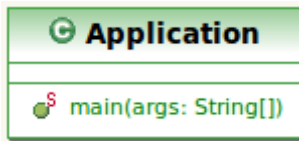


Inhaltsverzeichnis

Application.....	2
Methoden	2
Konstruktor:	3
Methoden	3
FunctionFrame	7
Konstruktor:	9
Methoden	11
Innere Klassen:	12
FunctionInputPanel	14
Konstruktor:	15
Innere Klassen:	15
Graph	16
Konstruktor:	17
Methoden	17
Innere Klassen:	21
GraphData	23
Konstruktor:	24
Methoden	24
GraphSettingPanel.....	27
Konstruktor:	28
Methoden	28
Innere Klassen:	29
MainFrame	30
Konstruktor:	30
SaveLoadPanel.....	31
Konstruktor:	31
Innere Klassen:	32
Function	32
Konstruktor:	33
Methoden	34
DPoint	44
Konstruktoren:.....	44
Methoden	44
ObjectSerializer	45
Methoden	45
Calculator.....	46
Methoden	46
Element	52
Konstruktoren.....	53
Methoden	53
Reflexion.....	55

Application



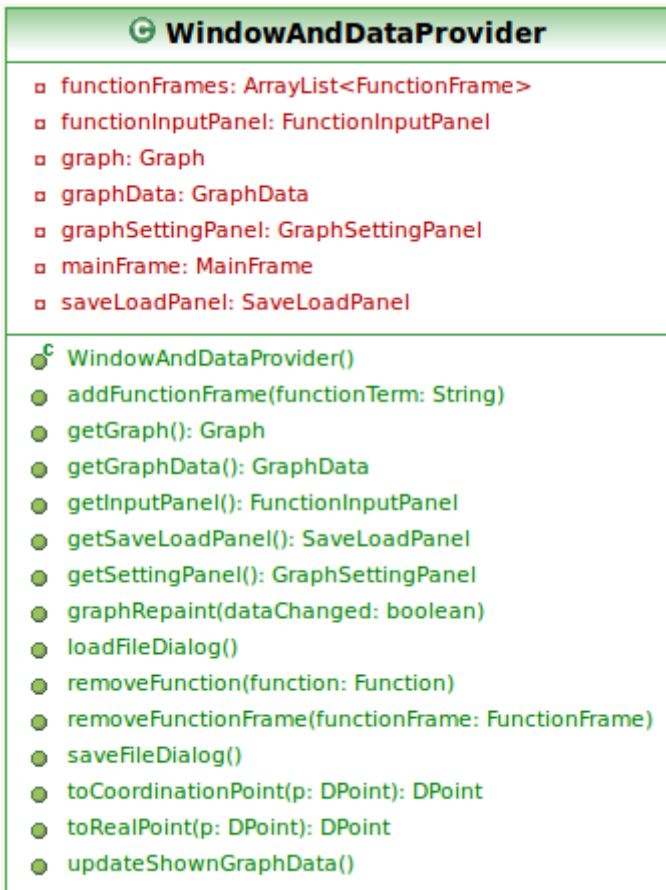
Die Klasse „Application“ dient als Programmeinstiegspunkt für die gesamte Anwendung.

Methoden

```
public static void main(String args[])
{
    new WindowAndDataProvider();
}
```

Dies ist die einzige und auch erste Methode welche in dieser Klasse existiert und aufgerufen wird. Durch den Aufruf wird ein Objekt der Klasse „WindowAndDataProvider“ erstellt.

WindowAndDataProvider



Hier werden alle Grafischen Elemente erstellt und verwaltet. Außerdem regelt die Klasse die Verteilung der Daten an die einzelnen Oberflächenelemente.

Konstruktor:

```
public WindowAndDataProvider()
{
    graphData = new GraphData();
    functionFrames = new ArrayList<FunctionFrame>();

    graph = new Graph(this);
    functionInputPanel = new FunctionInputPanel(this);
    graphSettingPanel = new GraphSettingPanel(this);
    saveLoadPanel = new SaveLoadPanel(this);
    mainFrame = new MainFrame(this);
}
```

Wird der Konstruktor aufgerufen, werden zunächst Objekte der folgenden Klassen erstellt:

- ◆ GraphData
- ◆ Graph
- ◆ FunctionInputPanel
- ◆ GraphSettingPanel
- ◆ SaveLoadPanel
- ◆ MainFrame

Auffallend ist, dass alle aufgerufenen Konstruktoren die aktuelle Instanz der Klasse „WindowAndDataProvider“ übergeben bekommen. Neben den bisher erwähnten Objekten wird auch eine ArrayList vom Typ der Klasse „FunctionFrame“ angelegt.

Methoden

```
public void addFunctionFrame(String functionTerm)
{
    Function function = graphData.addFunction(functionTerm);
    functionFrames.add(new FunctionFrame(this, function));
}
```

Mit der Methode „addFunctionFrame(...)“ wird ein Fenster erzeugt, welches mit dem übergebenen Funktionsterm und dessen Function-Objekt verknüpft wird.

```
public DPoint toCoordinationPoint(DPoint p)
{
    DPoint pNew = new DPoint();

    pNew.setX(graphData.getX0() + p.getX()*graphData.getXFactor());
    pNew.setY(graphData.getY0() - p.getY()*graphData.getYFactor());

    return pNew;
}
```

Diese Methode ist erforderlich, um einen Punkt aus dem mathematischen Koordinatensystem in einen Punkt für das Java-Koordinatensystem umzurechnen. Als Übergabeparameter wird ein Punkt des Typs „DPoint“ erwartet. Als Rückgabe erhält man ebenfalls einen Punkt vom selben Typ, jedoch mit dem Inhalt der Java-Koordinaten.

```
public DPoint toRealPoint(DPoint p)
{
    DPoint pNew = new DPoint();

    pNew.setX((graphData.getX0()-p.getX())/graphData.getXFactor()*(-1));
    pNew.setY((graphData.getY0()-p.getY())/graphData.getYFactor());

    return pNew;
}
```

Die Methode „toRealPoint(...)“ stellt das Gegenstück zur vorherigen Methode dar. Der Übergabeparameter und der Rückgabewert sind wieder vom Selben Typ, mit dem

Unterschied dass in diesem Fall eine Java-Koordinate in das Mathematische Koordinatensystem umgerechnet und zurückgegeben wird.

```
public Graph getGraph()  
{  
    return graph;  
}
```

Gibt das Objekt der Klasse Graph zurück.

```
public GraphData getGraphData()  
{  
    return graphData;  
}
```

Gibt das Klassenattribut „graphData“ zurück.

```
public FunctionInputPanel getInputPanel()  
{  
    return functionInputPanel;  
}
```

Gibt das Klassenattribut „functionInputPanel“ zurück.

```
public SaveLoadPanel getSaveLoadPanel()  
{  
    return saveLoadPanel;  
}
```

Gibt das Klassenattribut „saveLoadPanel“ zurück.

```
public GraphSettingPanel getSettingPanel()  
{  
    return graphSettingPanel;  
}
```

Gibt das Klassenattribut „graphSettingPanel“ zurück.

```
public void graphRepaint(boolean dataChanged)  
{  
    if(dataChanged)  
    {  
        for(int i=0; i<graphData.getNumberOfFunctions(); i++)  
        {  
            graphData.getFunction(i).calcFunctionData(graphData.getDomainLeft(),  
graphData.getDomainRight());  
        }  
        graph.repaint();  
    }  
}
```

Diese Methode wird aufgerufen, wenn der Graph neu gezeichnet werden soll. Dabei kann über einen Parameter bestimmt werden, ob die Daten aller Funktionen neu berechnet werden sollen oder nicht. Dies geschieht über eine Boolean-Variable. Ist der Wert „true“, werden alle Daten neu berechnet und anschließend neu im Graphen dargestellt. Wird jedoch „false“ übergeben, so entfällt die Neuberechnung und es wird lediglich anhand der vorhandenen Daten neu gezeichnet.

```

public void saveFileDialog()
{
    JFileChooser fc = new JFileChooser();

    if(fc.showSaveDialog(mainFrame) != JFileChooser.CANCEL_OPTION)
    {
        try
        {
            ObjectSerializer.writeToFile(fc.getSelectedFile().getPath(),
graphData);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

Der Aufruf „saveFileDialog()“ ermöglicht es einen Graphen mit all seinen Funktionen und dazugehörigen Daten zu speichern. Dazu erscheint ein Dialog, in dem der Benutzer auswählen kann, wo er seine Daten gerne speichern möchte. Außerdem hat er die Möglichkeit das Speichern abubrechen. Wird der Speichern Button betätigt, so werden die Daten mittels der Klasse „ObjectSerializer“ gespeichert.

```

public void loadFileDialog()
{
    JFileChooser fc = new JFileChooser();

    if(fc.showOpenDialog(mainFrame) != JFileChooser.CANCEL_OPTION)
    {
        try
        {
            GraphData data;
            data = (GraphData)
ObjectSerializer.readFromFile(fc.getSelectedFile().getPath());

            if(data != null)
            {
                for(int i=0; i<functionFrames.size(); i++)
                {
                    functionFrames.get(i).dispose();
                }

                graphData = data;

                for(int i=0; i<graphData.getNumberOfFunctions(); i++)
                {
                    functionFrames.add(new FunctionFrame(this,
graphData.getFunction(i)));
                }

                updateShownGraphData();
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
    }
}

```

Hat man durch die vorherige Methode Daten gespeichert, ist es sinnvoll diese auch wieder Auslesen zu können. Und genau diese Möglichkeit wird mit der Methode „loadFileDialog()“ geboten. Durch ihren Aufruf wird der Benutzer diesmal dazu aufgefordert in einem Dialog die zu öffnende Datei auszuwählen. Auch hier hat er wieder die Möglichkeit Abzubrechen. Ist der Dialog jedoch mit „Öffnen“ bestätigt worden, werden alle Fenster zu den bereits vorhandenen Funktionen geschlossen, und die Daten mit denen aus der Datei überschrieben. Daraufhin werden Abschließend noch die Fenster zu den geladenen Funktionen erstellt und angezeigt.

```
public void removeFunction(Function function)
{
    graphData.removeFunction(function);
}
```

Die Methode „removeFunction(...)“ wird benutzt, um die als Parameter übergebene Funktion aus dem „graphData“ Objekt zu löschen.

```
public void removeFunctionFrame(FunctionFrame functionFrame)
{
    functionFrames.remove(functionFrame);
}
```

Auch hierbei handelt es sich um eine Methode welche ein Objekt löscht. Dazu wird das zu löschende Objekt der Klasse „FunktionFrame“ übergeben, und anschließend aus der ArrayList aller FunctionFrame’s gelöscht.

```
public void updateShownGraphData()
{
    graphSettingPanel.setTextFields(graphData.getCodomainLeft(),
    graphData.getCodomainRight(), graphData.getDomainLeft(), graphData.getDomainRight());
}
```

Mithilfe von „updateShownGraphData()“ wird die Anzeige des Werte- und Definitionsbereichs unter dem Graphen auf den neuesten Stand gebracht. Dazu werden die benötigten Daten aus dem Objekt „graphData“ als Parametern dem Methodenaufruf „setTextFields(...)“ der Klasse „GraphSettingPanel“ übergeben.

FunctionFrame

FunctionFrame
<ul style="list-style-type: none"> cancelButton: JButton dataPanel: JPanel drawCheck: JCheckBox drawLowerSumCheck: JCheckBox drawSettingsPanel: JPanel drawUpperSumCheck: JCheckBox editButton: JButton function: Function functionEditPanel: JPanel functionInput: JTextField functionLabel: JLabel integralLabel: JLabel intervallA: JTextField intervallB: JTextField intervallLabel: JLabel lowerSumLabel: JLabel outputPanel: JPanel slider: JSlider steps: JTextField stepsLabel: JLabel upperSumLabel: JLabel windowAndDataProvider: WindowAndDataProvider
<ul style="list-style-type: none"> FunctionFrame(newWindowProvider: WindowAndDataProvider, newfunction: Function) closeWindow() updateData()
CListener
<ul style="list-style-type: none"> stateChanged(in e: ChangeEvent)
KListener
<ul style="list-style-type: none"> keyPressed(in e: KeyEvent) keyReleased(in e: KeyEvent) keyTyped(in e: KeyEvent)
Listener
<ul style="list-style-type: none"> actionPerformed(in e: ActionEvent)



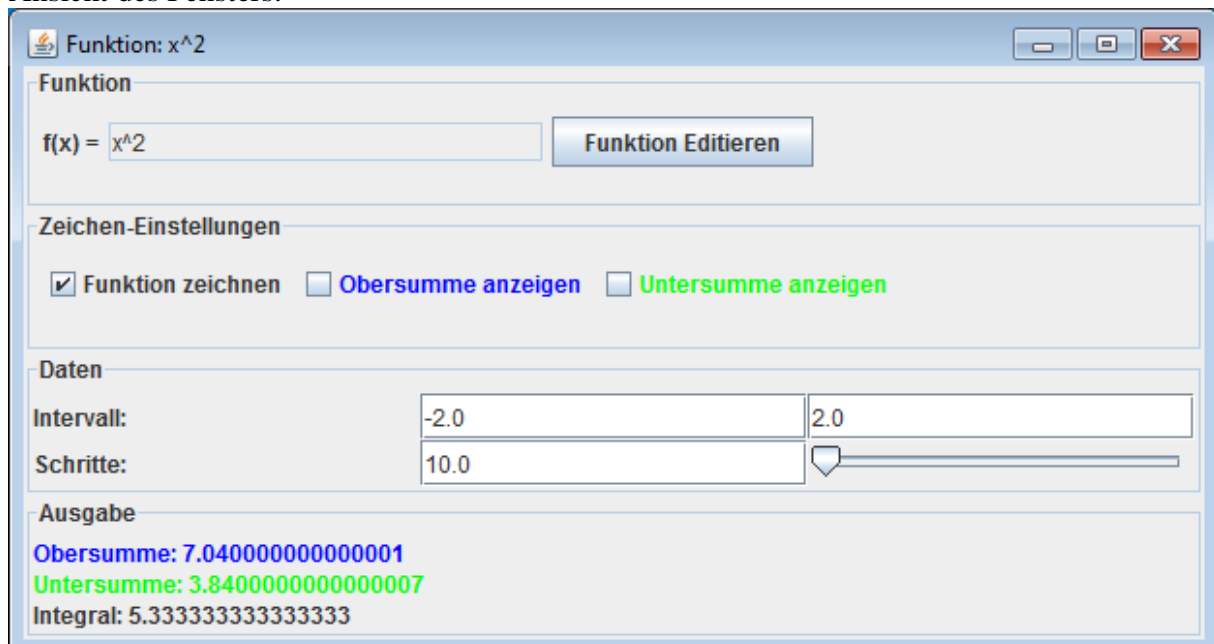
Die Klasse „FunctionFrame“ stellt das Interface zur Konfiguration einer Funktion dar. Dem Nutzer des Programms erscheint sie als Fenster, welches diverse Einstellungsmöglichkeiten bietet. Dazu gehören:

- ◆ Editieren des Funktionsterms
- ◆ Ausblenden der Funktion im Graphen
- ◆ Anzeigen der Ober- und Untersumme
- ◆ Festlegung des Intervalls
- ◆ Anzahl der Schritte bzw. Teilflächen der Ober- und Untersumme

Unter den Einstellungen befinden sich drei weitere Zeilen, welche als Ergebnisanzeige der Funktion dienen.

1. Zeile: Obersumme in Abhängigkeit der Einstellungen
2. Zeile: Untersumme in Abhängigkeit der Einstellungen
3. Zeile: Integral in Abhängigkeit der Einstellungen

Ansicht des Fensters:



Konstruktor:

```
public FunctionFrame(WindowAndDataProvider newWindowProvider, Function newfunction)
{
    function = newfunction;
    windowAndDataProvider = newWindowProvider;

    setTitle("Funktion: "+function.getTerm());
    setLayout(new GridLayout(0, 1));
    addWindowListener(new WListener());
    setVisible(true);
    setAlwaysOnTop(true);

    functionEditPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    functionEditPanel.setBorder(new TitledBorder("Funktion"));

    functionLabel = new JLabel("f(x) =");
    functionEditPanel.add(functionLabel);

    functionInput = new JTextField(function.getTerm());
    functionInput.setEditable(false);
    functionInput.setColumns(20);
    functionInput.addKeyListener(new KListener());
    functionEditPanel.add(functionInput);

    editButton = new JButton("Funktion Editieren");
    editButton.addActionListener(new Listener());
    functionEditPanel.add(editButton);

    cancelButton = new JButton("Abbrechen");
    cancelButton.addActionListener(new Listener());
    cancelButton.setVisible(false);
    functionEditPanel.add(cancelButton);

    drawSettingsPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    drawSettingsPanel.setBorder(new TitledBorder("Zeichen-Einstellungen"));

    drawCheck = new JCheckBox();
    drawCheck.setText("Funktion zeichnen");
    drawCheck.setSelected(function.isVisible());
    drawCheck.addActionListener(new Listener());
    drawSettingsPanel.add(drawCheck);

    drawUpperSumCheck = new JCheckBox();
    drawUpperSumCheck.setText("Obersumme anzeigen");
    drawUpperSumCheck.setSelected(function.isObersummeVisible());
    drawUpperSumCheck.addActionListener(new Listener());
    drawUpperSumCheck.setForeground(Color.blue);
    drawSettingsPanel.add(drawUpperSumCheck);

    drawLowerSumCheck = new JCheckBox();
    drawLowerSumCheck.setText("Untersumme anzeigen");
    drawLowerSumCheck.setSelected(function.isUntersummeVisible());
    drawLowerSumCheck.addActionListener(new Listener());
    drawLowerSumCheck.setForeground(Color.green);
    drawSettingsPanel.add(drawLowerSumCheck);

    dataPanel = new JPanel(new GridLayout(0, 3));
    dataPanel.setBorder(new TitledBorder("Daten"));

    intervallLabel = new JLabel("Intervall:");
    dataPanel.add(intervallLabel);

    intervallA = new JTextField("-2");
    intervallA.setText(String.valueOf(function.getIntervall1()));
    intervallA.addKeyListener(new KListener());
    dataPanel.add(intervallA);

    intervallB = new JTextField();
    intervallB.setText(String.valueOf(function.getIntervall2()));
    intervallB.addKeyListener(new KListener());
    dataPanel.add(intervallB);
}
```

```

stepsLabel = new JLabel("Schritte:");
dataPanel.add(stepsLabel);

steps = new JTextField("10");
steps.setText(String.valueOf(function.getSteps()));
steps.addKeyListener(new KListener());
dataPanel.add(steps);

slider = new JSlider(1, 1000, 10);
slider.addChangeListener(new CListener());
dataPanel.add(slider);
slider.setValue(10);

outputPanel = new JPanel(new GridLayout(0, 1));
outputPanel.setBorder(new TitledBorder("Ausgabe"));

upperSumLabel = new JLabel("Obersumme: ");
upperSumLabel.setForeground(Color.blue);
outputPanel.add(upperSumLabel);

lowerSumLabel = new JLabel("Untersumme: ");
lowerSumLabel.setForeground(Color.green);
outputPanel.add(lowerSumLabel);

integralLabel = new JLabel("Integral: ");
outputPanel.add(integralLabel);

add(functionEditPanel);
add(drawSettingsPanel);
add(dataPanel);
add(outputPanel);

pack();

updateData();
}

```

Der Konstruktor erstellt alle Oberflächenelemente, die für das anzuzeigende Fenster nötig sind. Um das Fenster mit der Richtigen Funktion zu verknüpfen, erwartet er zwei Übergabeparameter. Zum einen, ein Objekt der Klasse „WindowAndDataProvider“ und zum anderen eines vom Typ „Function“. Ersterer ermöglicht das Neuzeichnen der Funktion und Letzterer das Einstellen der Funktion. Darüber hinaus werden den Objekten welche Benutzerinteraktion erfordern verschiedene Listener zugewiesen. Ist die Oberfläche soweit initialisiert, werden abschließend noch die Inhalte der Textfelder mittels der Methode „updateData()“ in der betreffenden Funktion gespeichert.

Methoden

```
public void updateData()
{
    try
    {
        function.setIntervall1(Double.parseDouble(intervallA.getText()));
        intervallA.setBackground(Color.white);
    }
    catch (NumberFormatException e)
    {
        intervallA.setBackground(Color.red);
    }

    try
    {
        function.setIntervall2(Double.parseDouble(intervallB.getText()));
        intervallB.setBackground(Color.white);
    }
    catch (NumberFormatException e)
    {
        intervallB.setBackground(Color.red);
    }

    try
    {
        function.setSteps(Double.parseDouble(steps.getText()));
        slider.setValue((int) Double.parseDouble(steps.getText()));
        steps.setBackground(Color.white);
    }
    catch (NumberFormatException e)
    {
        steps.setBackground(Color.red);
    }

    windowAndDataProvider.graphRepaint(true);
    windowAndDataProvider.updateShownGraphData();

    upperSumLabel.setText("Obersumme: "+String.valueOf(function.calcObersumme()));
    lowerSumLabel.setText("Untersumme: "+String.valueOf(function.calcUntersumme()));
    integralLabel.setText("Integral: "+String.valueOf(function.calcIntegral()));
}
```

Wie bereits erwähnt, werden mithilfe dieser Methode die Benutzereingaben in das Objekt „function“ geschrieben. Schlägt dieser Vorgang bei einem der Textfelder fehl, wird das Speichern abgebrochen, und das betreffende Textfeld wird rot hinterlegt. Dadurch wird der Benutzer auf eine ungültige Eingabe hingewiesen.

Nachdem das Speichern der Eingabedaten beendet ist, wird der Graph anschließend neu gezeichnet, damit die Änderungen auch sichtbar werden. Des Weiteren werden Obersumme, Untersumme und der Integral der Funktion neu berechnet und angezeigt.

```
public void closeWindow()
{
    windowAndDataProvider.removeFunction(function);
    windowAndDataProvider.removeFunctionFrame(this);
    windowAndDataProvider.graphRepaint(true);
    windowAndDataProvider.updateShownGraphData();
}
```

Die Methode „closeWindow()“ wird aufgerufen, wenn das Fenster und die dazugehörige Funktion geschlossen bzw. gelöscht werden sollen. Dazu wird sowohl die Funktion, wie auch das dazu gehörende „FunctionFrame“-Objekt mittels Referenzobjekt der Klasse „WindowAndDataProvider“ entfernt. Zu guter letzt wird noch der Graph neu gezeichnet und dessen Daten aktualisiert.

Innere Klassen:

```
private class Listener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        Object source = e.getSource();

        if(source.equals(drawCheck))
        {
            function.setVisible(((JCheckBox)source).isSelected());
        }
        else if(source.equals(drawUpperSumCheck))
        {
            function.setObersummeVisible(((JCheckBox)source).isSelected());
        }
        else if(source.equals(drawLowerSumCheck))
        {
            function.setUntersummeVisible(((JCheckBox)source).isSelected());
        }
        else if(source.equals(editButton))
        {
            if(functionInput.isEditable() == false)
            {
                functionInput.setEditable(true);
                editButton.setText("Speichern");
                cancelButton.setVisible(true);
            }
            else
            {
                function.setTerm(functionInput.getText(),
                windowAndDataProvider.getGraphData().getDomainLeft(),
                windowAndDataProvider.getGraphData().getDomainRight());
                setTitle("Funktion: "+function.getTerm());
                functionInput.setEditable(false);
                editButton.setText("Funktion Editieren");
                cancelButton.setVisible(false);
                updateData();
            }
        }
        else if(source.equals(cancelButton))
        {
            functionInput.setText(function.getTerm());
            functionInput.setEditable(false);
            editButton.setText("Funktion Editieren");
            editButton.setEnabled(true);
            cancelButton.setVisible(false);
            functionInput.setBackground(Color.white);
        }

        windowAndDataProvider.graphRepaint(true);
    }
}
```

Die Klasse „Listener“ implementiert einen „ActionListener“, welcher in diesem Fall genutzt wird, um auf die Interaktionen des Benutzers zu reagieren.

```
private class WListener implements WindowListener
{
    public void windowActivated(WindowEvent e) {}

    public void windowClosed(WindowEvent e) {}

    public void windowClosing(WindowEvent e)
    {
        closeWindow();
    }

    public void windowDeactivated(WindowEvent e) {}

    public void windowDeiconified(WindowEvent e) {}

    public void windowIconified(WindowEvent e) {}

    public void windowOpened(WindowEvent e) {}
}
```

Die Klasse „WListener“ implementiert einen „WindowListener“. Die Methode „windowClosing(...)“ wird aufgerufen, bevor das Fenster geschlossen ist, und ruft die Methode „closeWindow()“ auf, welche wie bereits beschrieben das Löschen der Funktion und des Fensters einleitet.

```
private class KListener implements KeyListener
{
    public void keyPressed(KeyEvent e) {}

    public void keyReleased(KeyEvent e)
    {
        if(e.getSource().equals(functionInput))
        {
            if(Function.inputIsFunction(functionInput.getText()))
            {
                functionInput.setBackground(Color.white);
                editButton.setEnabled(true);
            }
            else
            {
                functionInput.setBackground(Color.red);
                editButton.setEnabled(false);
            }
        }
        else
        {
            updateData();
        }
    }

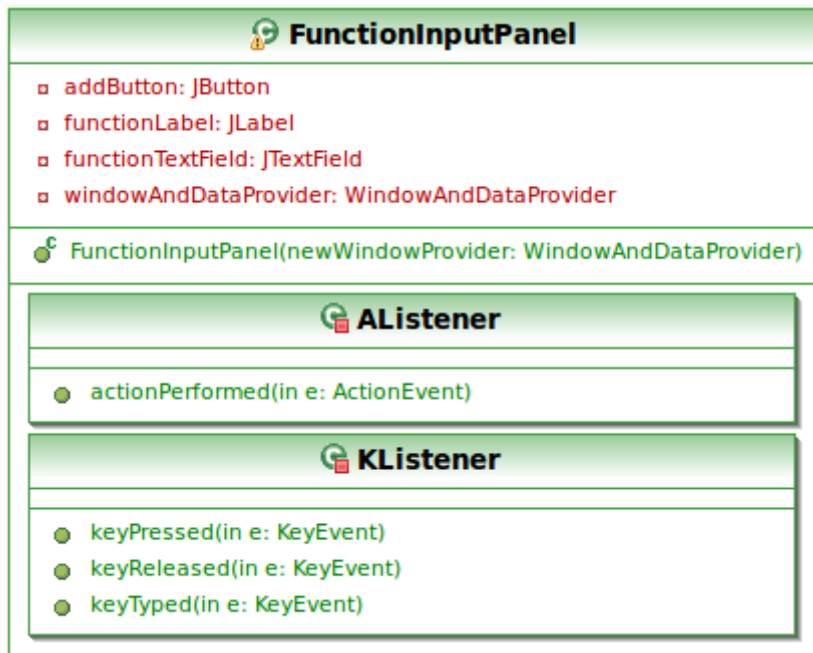
    public void keyTyped(KeyEvent e) {}
}
```

Die Klasse „KListener“ implementiert einen „KeyListener“ und ist für die Handhabung bei Tastendrücken der entsprechenden Objekte zuständig.

```
private class CListener implements ChangeListener
{
    public void stateChanged(ChangeEvent e)
    {
        steps.setText(String.valueOf(slider.getValue()));
        updateData();
    }
}
```

Die Klasse „CListener“ implementiert einen „ChangeListener“ und enthält die Methode „stateChanged(...)“, welche aufgerufen wird, sobald sich die Position des Sliders zum Einstellen der Schritte ändert. Der neue Wert wird dann in das zugehörige Textfeld geschrieben und der Graph wird mit neuen Daten gezeichnet.

FunctionInputPanel



Zur Eingabe und dem Hinzufügen einer Funktion wurde die Klasse „FunctionInputPanel“ entworfen. Sie erbt von der Klasse „JPanel“ und bietet zum Definieren einer Funktion ein Textfeld. Zum abschließenden Hinzufügen steht es dem Benutzer frei, nach der Eingabe die Return-Taste oder den Button „Funktion hinzufügen“ zu klicken. Um die Möglichkeit eine falsch definierte Funktion hinzuzufügen weitgehend zu minimieren, wird der Eingabestring auf funktionstypische Zeichen überprüft. Im Fall einer Abweichung wird das Textfeld rot hinterlegt, und alle Möglichkeiten zum Hinzufügen deaktiviert.

Ansicht des Panels:

The screenshot shows the user interface of the **FunctionInputPanel**. It features a label **Funktion** above a text input field. The input field is preceded by the text **f(x) =**. To the right of the input field is a button labeled **Funktion hinzufügen**.

Konstruktor:

```
public FunctionInputPanel(WindowAndDataProvider newWindowProvider)
{
    windowAndDataProvider = newWindowProvider;

    setLayout(new FlowLayout(FlowLayout.LEFT));
    setBorder(new TitledBorder("Funktion"));

    functionLabel = new JLabel("f(x) =");
    add(functionLabel);

    functionTextField = new JTextField();
    functionTextField.setColumns(20);
    functionTextField.addKeyListener(new KListener());
    add(functionTextField);

    addButton = new JButton("Funktion hinzufügen");
    addButton.setEnabled(false);
    addButton.addActionListener(new ActionListener());
    add(addButton);
}
```

Im Konstruktor werden das Textfeld und der Button erstellt, um sie anschließend dem Panel hinzuzufügen. Hierzu wird ein so genanntes FlowLayout verwendet, welches die Elemente hintereinander, entsprechend ihrer hinzugefügten Reihenfolge anordnet. Damit die Oberflächenelemente später auch auf Interaktionen reagieren, werden ihnen entsprechende Listener hinzugefügt.

Innere Klassen:

```
private class KListener implements KeyListener
{
    public void keyPressed(KeyEvent e) {}

    public void keyReleased(KeyEvent e)
    {
        if(Function.inputIsFunction(functionTextField.getText()))
        {
            addButton.setEnabled(true);
            functionTextField.setBackground(Color.white);

            if(e.getKeyCode() == KeyEvent.VK_ENTER)
            {
                windowAndDataProvider.addFunctionFrame(functionTextField.getText());
                functionTextField.setText("");
                functionTextField.setBackground(Color.white);
            }
        }
        else
        {
            addButton.setEnabled(false);
            functionTextField.setBackground(Color.red);
        }
    }

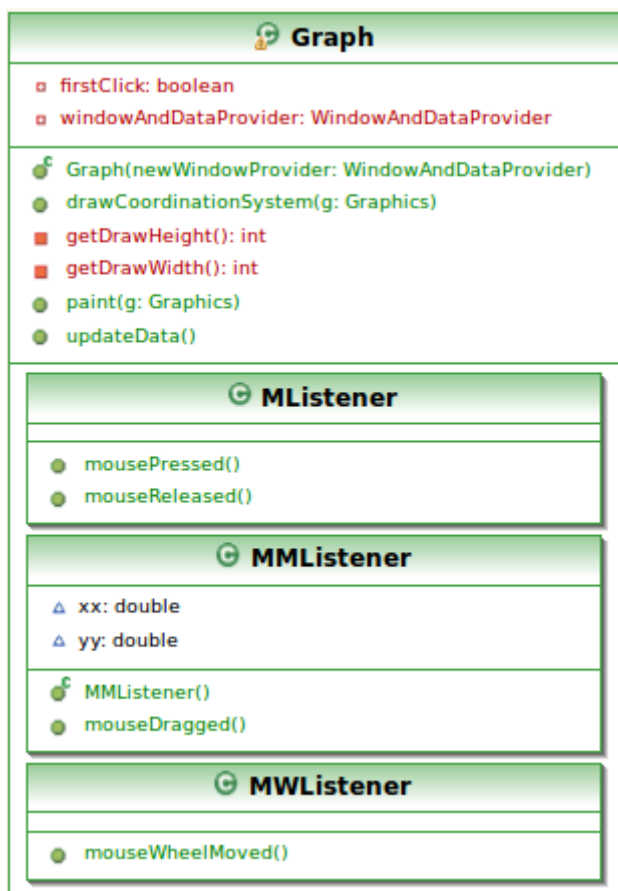
    public void keyTyped(KeyEvent e) {}
}
```

Die Klasse „KListener“ implementiert einen „KeyListener“. Sie beinhaltet die Methode „keyReleased(...)“, welche aufgerufen wird, nachdem ein Tastendruck im Textfeld erfolgt ist. In der Methode wird dann überprüft, ob es sich beim Eingabestring des Textfeldes um funktionstypische Zeichen handelt oder nicht. Ist dem nicht so, wird der Eingabehintergrund Rot gefärbt und der Button ausgegraut. Im anderen Fall wird der Button benutzbar und ein Drücken der Return-Taste führt zum hinzufügen der Funktion.

```
private class AListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        windowAndDataProvider.addFunctionFrame(functionTextField.getText());
        functionTextField.setText("");
        functionTextField.setBackground(Color.white);
    }
}
```

Mit dem „AListener“ welcher die Klasse „ActionListener“ implementiert, wird lediglich festgelegt, dass beim drücken des Button die Polynomfunktion hinzugefügt wird.

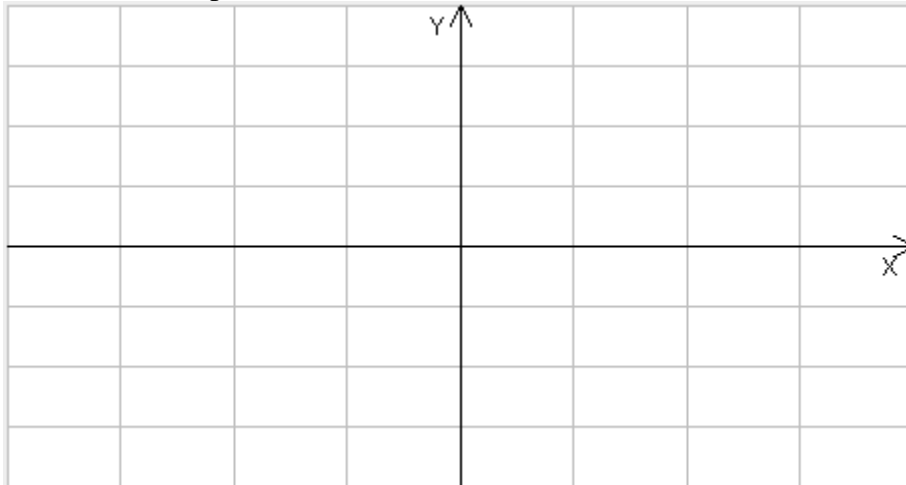
Graph



Auch die Klasse „Graph“ erbt von der Klasse „JPanel“, und ist für das Darstellen eines Graphen verantwortlich.

Dem Benutzer gegenüber erscheint er als Koordinatensystem mit einem Gitternetz, wobei ein Kästchen eine Flächeneinheit repräsentiert.

Ansicht des Graphen:



Konstruktor:

```
public Graph(WindowAndDataProvider newWindowProvider)
{
    windowAndDataProvider = newWindowProvider;

    addMouseMotionListener(new MMListener());
    addMouseWheelListener(new MWListener());
    addMouseListener(new MListener());

    setBackground(Color.white);
    setDoubleBuffered(true);
}
```

Mit dem Konstruktoraufruf wird dem Panel ein „MouseMotionListener“, ein „MouseWheelListener“ und ein „MouseListener“ hinzugefügt. Um den Graphen auch optisch anscheinlicher zu machen, wird dessen Hintergrund auf die Farbe Weiß gesetzt.

Methoden

```
private int getDrawWidth()
{
    return getWidth()-2;
}
```

Die Methode „getDrawWidth()“ gibt die Breite des Panels subtrahiert mit Zwei als Integer Wert zurück.

```
private int getDrawHeight()
{
    return getHeight()-2;
}
```

Im Gegensatz zur vorherigen Methode wird hier die Höhe mit Zwei subtrahiert und ebenfalls als Integer zurückgegeben.

```

public void paint(Graphics g)
{
    GraphData data = windowAndDataProvider.getGraphData();

    super.paint(g);
    updateData();
    drawCoordinationSystem(g);

    for(int i=0; i<data.getNumberOfFunctions(); i++)
    {
        data.getFunction(i).drawObersumme(windowAndDataProvider, g, this);
        data.getFunction(i).drawUntersumme(windowAndDataProvider, g, this);
    }

    for(int i=0; i<data.getNumberOfFunctions(); i++)
    {
        data.getFunction(i).drawFunction(windowAndDataProvider, g,
data.getDomainLeft(), data.getDomainRight());
    }
}

```

Die „paint(...)“ Methode dient dem Panel zum Zeichnen von Grafiken auf dessen Oberfläche. Sie überschreibt die geerbte Methode der Klasse „JPanel“, und bekommt ein Objekt der Klasse „Graphics“ als Parameter übergeben. Auf Dieses Objekt kann man nun zeichnen. Dabei ist zu beachten, dass zuerst gezeichnete Grafiken von später Gezeichneten überlagert werden. Deshalb wird zuerst das Koordinatensystem inklusive Gitternetz gezeichnet. Darüber wird dann zuerst die Obersumme und dann die Untersumme sowie die eigentliche Funktion gezeichnet.

```

public void updateData()
{
    GraphData data = windowAndDataProvider.getGraphData();
    data.setXFactor(getDrawWidth() / (data.getDomainRight() - data.getDomainLeft()));
    data.setYFactor(getDrawHeight() / (data.getCodomainRight() - data.getCodomainLeft()));

    data.setX0(getDrawWidth() - (data.getDomainRight() * data.getXFactor()));
    data.setY0(data.getCodomainRight() * data.getYFactor());
}

```

Mit dem Aufruf der Methode „updateData()“ werden die Daten zu Skalierung des Graphen neu berechnet und gespeichert.

```

public void drawCoordinationSystem(Graphics g)
{
    GraphData data = windowAndDataProvider.getGraphData();
    g.setColor(Color.cyan);

    g.setColor(data.getNetColor());

    DPoint p1 = new DPoint();
    DPoint p2 = new DPoint();

    for(double i=0; i<=data.getDomainRight(); i++)
    {
        if(i!=0)
        {
            p1.setX(i);
            p1.setY(data.getCodomainLeft());
            p2.setX(i);
            p2.setY(data.getCodomainRight());

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int)
p2.getY());
        }
    }

    for(double i=0; i>=data.getDomainLeft(); i--)
    {
        if(i!=0)
        {
            p1.setX(i);
            p1.setY(data.getCodomainLeft());
            p2.setX(i);
            p2.setY(data.getCodomainRight());

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int)
p2.getY());
        }
    }

    for(double i=0; i<=data.getCodomainRight(); i++)
    {
        if(i!=0)
        {
            p1.setX(data.getDomainLeft());
            p1.setY(i);
            p2.setX(data.getDomainRight());
            p2.setY(i);

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int)
p2.getY());
        }
    }

    for(double i=0; i>=data.getCodomainLeft(); i--)
    {
        if(i!=0)
        {
            p1.setX(data.getDomainLeft());
            p1.setY(i);
            p2.setX(data.getDomainRight());
            p2.setY(i);

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int)
p2.getY());
        }
    }
}

```

```

g.setColor(data.getCrossColor());

//Koordinatenkreuz
g.drawLine((int) data.getX0(), 0, (int) data.getX0(), getDrawHeight());
g.drawLine(0, (int) data.getY0(), getDrawWidth(), (int) data.getY0());

//Koordinatenpfeile
p1.setX(data.getX0());
p1.setY(0);

p2.setX(data.getX0()-5);
p2.setY(10);

g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int) p2.getY());

p1.setX(data.getX0());
p1.setY(0);

p2.setX(data.getX0()+5);
p2.setY(10);

g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int) p2.getY());

p1.setX(getDrawWidth());
p1.setY(data.getY0());

p2.setX(getDrawWidth()-10);
p2.setY(data.getY0()-5);

g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int) p2.getY());

p1.setX(getDrawWidth());
p1.setY(data.getY0());

p2.setX(getDrawWidth()-10);
p2.setY(data.getY0()+5);

g.drawLine((int) p1.getX(), (int) p1.getY(), (int) p2.getX(), (int) p2.getY());

//Koordinatenbezeichnung
p1.setX(getDrawWidth()-15);
p1.setY(data.getY0()+15);

p2.setX(data.getX0()-15);
p2.setY(0+15);

g.drawString("X", (int) p1.getX(), (int) p1.getY());
g.drawString("Y", (int) p2.getX(), (int) p2.getY());
}

```

Das Unterprogramm „drawCoordinationSystem(...)“ bekommt als Parameter ein Graphics-Objekt übergeben, auf welches im folgenden Verlauf das Koordinatensystem gezeichnet wird. Dies geschieht in der folgenden Reihenfolge:

1. zuerst werden die senkrechten Linien auf der rechten Seite der Y-Achse im Abstand von einer Einheit gezeichnet
2. danach werden die senkrechten Linien auf der linken Seite der Y-Achse im Abstand gezeichnet
3. Anschließend geschieht das selbe in waagrechtter Anordnung zuerst über der X-Achse und dann unter ihr

Innere Klassen:

```
public class MMListener extends MouseMotionAdapter
{
    double xx;
    double yy;

    public MMListener()
    {
        xx = 0;
        yy = 0;
        firstClick = true;
    }

    public void mouseDragged(MouseEvent e)
    {
        GraphData data = windowAndDataProvider.getGraphData();
        DPoint p1 = new DPoint(e.getX(), e.getY());
        DPoint p2 = new DPoint(xx, yy);

        p1 = windowAndDataProvider.toRealPoint(p1);

        if(firstClick == true)
        {
            xx = p1.getX();
            yy = p1.getY();
            firstClick = false;
        }
        else
        {
            double xdiff, ydiff;

            xdiff = p1.getX() - p2.getX();
            ydiff = p1.getY() - p2.getY();

            data.setDomainLeft(data.getDomainLeft() - xdiff);
            data.setDomainRight(data.getDomainRight() - xdiff);
            data.setCodomainLeft(data.getCodomainLeft() - ydiff);
            data.setCodomainRight(data.getCodomainRight() - ydiff);

            windowAndDataProvider.updateShownGraphData();
            windowAndDataProvider.graphRepaint(true);
        }
    }
}
```

Diese Klasse wird genutzt, um auf Mausbewegungen zu reagieren. Wird die Maus mit gedrückter Taste auf dem Graphen gezogen, wird der Nullpunkt des Graphen mit der Bewegung verschoben.

```
public class MListener extends MouseAdapter
{
    public void mouseReleased(MouseEvent e)
    {
        firstClick = false;
    }

    public void mousePressed(MouseEvent e)
    {
        firstClick = true;
    }
}
```

Mit dieser Klasse wird festgelegt, ob es sich bei einem Mausklick um den ersten handelt der wahrgenommen wird oder nicht.

```

public class MWListener extends MouseAdapter
{
    public void mouseWheelMoved(MouseWheelEvent e)
    {
        GraphData data = windowAndDataProvider.getGraphData();
        double x = 0;
        DPoint p = new DPoint(e.getX(), e.getY());

        p = windowAndDataProvider.toRealPoint(p);

        if(e.getWheelRotation() == -1)
        {
            x = -(data.getDomainRight()-data.getDomainLeft())/10;
        }
        else
        {
            x = (data.getDomainRight()-data.getDomainLeft())/10;
        }


        if(data.getDomainLeft() - x < data.getDomainRight() + x &&
        data.getCodomainLeft() + x < data.getCodomainRight())
        {
            data.setDomainLeft(data.getDomainLeft() - x);
            data.setDomainRight(data.getDomainRight() + x);
            data.setCodomainLeft(data.getCodomainLeft() - x);
            data.setCodomainRight(data.getCodomainRight() + x);
        }

        windowAndDataProvider.updateShownGraphData();
        windowAndDataProvider.graphRepaint(true);
    }
}

```

Die Klasse ist dafür zuständig, dass der Graph je nach Richtung des gedrehten Mausekads entweder vergrößert oder verkleinert wird.

GraphData

 GraphData	
▣	<code>codomainLeft: double</code>
▣	<code>codomainRight: double</code>
▣	<code>crossColor: Color</code>
▣	<code>domainLeft: double</code>
▣	<code>domainRight: double</code>
▣	<code>functions: ArrayList<Function></code>
▣	<code>netColor: Color</code>
▣	<code>x0: double</code>
▣	<code>xFactor: double</code>
▣	<code>y0: double</code>
▣	<code>yFactor: double</code>
●	<code>GraphData()</code>
●	<code>addFunction(functionTerm: String): Function</code>
●	<code>getCodomainLeft(): double</code>
●	<code>getCodomainRight(): double</code>
●	<code>getCrossColor(): Color</code>
●	<code>getDomainLeft(): double</code>
●	<code>getDomainRight(): double</code>
●	<code>getFunction(index: int): Function</code>
●	<code>getNetColor(): Color</code>
●	<code>getNumberOfFunctions(): int</code>
●	<code>getXFactor(): double</code>
●	<code>getY0(): double</code>
●	<code>getYFactor(): double</code>
●	<code>removeFunction(function: Function)</code>
●	<code>setCodomainLeft(newCodomainLeft: double)</code>
●	<code>setCodomainRight(newCodomainRight: double)</code>
●	<code>setCrossColor(newCrossColor: Color)</code>
●	<code>setDomainLeft(newDomainLeft: double)</code>
●	<code>setDomainRight(newDomainRight: double)</code>
●	<code>setX0(newX0: double)</code>
●	<code>setXFactor(newXFactor: double)</code>
●	<code>setY0(newY0: double)</code>
●	<code>setYFactor(newYFactor: double)</code>

„GraphData“ ist die Zentrale Klasse zum Speichern der Daten für den Graphen sowie zum Speichern der Funktionen. Zu diesen Daten gehören:

- ◆ Wertebereich
- ◆ Definitionsbereich
- ◆ Liste der Polynomfunktionen
- ◆ Die Faktoren zur Umrechnung der Koordinaten
- ◆ Und die Koordinaten des Nullpunktes
- ◆ Gitternetzfarbe des Koordinatensystems
- ◆ Farbe des Koordinatenkreuzes

Konstruktor:

```
public GraphData()
{
    functions = new ArrayList<Function>();
    crossColor = Color.black;
    netColor = Color.LIGHT_GRAY;
}
```

Der Konstruktor ist hier lediglich dafür zuständig, dass die Farben für das Gitternetz und die des Koordinatenkreuzes gesetzt werden. Zudem wird die Liste für die Funktionen Initialisiert, sodass sie bereit ist mit Objekten vom Typ „Function“ befüllt zu werden.

Methoden

```
public Function addFunction(String functionTerm)
{
    Function function = new Function(functionTerm, domainLeft, domainRight);
    functions.add(function);

    return function;
}
```

„addFunction(...)“ erstellt aus dem übergebenen String ein Objekt von der Klasse „Function“ und fügt es an das Ende der ArrayList „functions“ an.

```
public double getCodomainLeft()
{
    return codomainLeft;
}
```

Liefert als Rückgabewert die untere Grenze des Wertebereichs als Double zurück.

```
public double getCodomainRight()
{
    return codomainRight;
}
```

Liefert als Rückgabewert die obere Grenze des Wertebereichs als Double zurück.

```
public Color getCrossColor()
{
    return crossColor;
}
```

Gibt die Farbe des Koordinatenkreuzes als Color-Objekt zurück.

```
public double getDomainLeft()
{
    return domainLeft;
}
```

Liefert als Rückgabewert die linke Grenze des Definitionsbereichs als Double zurück.

```
public double getDomainRight()
{
    return domainRight;
}
```

Liefert als Rückgabewert die rechte Grenze des Definitionsbereichs als Double zurück.

```
public Function getFunction(int index)
{
    return functions.get(index);
}
```


Mit dem Aufruf des Unterprogramms „getFunction(...)“ wird das Function-Objekt mit dem übergebenen Index aus der ArrayList zurückgegeben.

```
public Color getNetColor()
{
    return netColor;
}
```

Gibt die Farbe des Gitternetzes als Color-Objekt zurück.

```
public int getNumberOfFunctions()
{
    return functions.size();
}
```

Der Methodenaufruf „getNumberOfFunctions()“ bewirkt, dass die Anzahl der gespeicherten Funktionen zurückgegeben wird.

```
public double getX0()
{
    return x0;
}
```

Gibt den X-Wert des Nullpunkts als Pixelangabe zurück.

```
public double getXFactor()
{
    return xFactor;
}
```

„getXFactor()“ liefert als Rückgabewert den Faktor in X-Richtung zum Umrechnen der Koordinaten.

```
public double getY0()
{
    return y0;
}
```

Gibt den Y-Wert des Nullpunkts als Pixelangabe zurück.

```
public double getYFactor()
{
    return yFactor;
}
```

„getYFactor()“ liefert als Rückgabewert den Faktor in Y-Richtung zum Umrechnen der Koordinaten.

```
public void removeFunction(Function function)
{
    functions.remove(function);
}
```

Mit der Methode lässt sich das übergebene Function-Objekt aus der ArrayList eines DataGraph-Objektes löschen.

```
public void setCodomainLeft(double newCodomainLeft)
{
    codomainLeft = newCodomainLeft;
}
```

Setzt die untere Grenze des Wertebereichs auf den übergebenen Double-Wert.

```
public void setCodomainRight(double newCodomainRight)
{
    codomainRight = newCodomainRight;
}
```

Setzt die obere Grenze des Wertebereichs auf den übergebenen Double-Wert.

```
public void setCrossColor(Color newCrossColor)
{
    crossColor = newCrossColor;
}
```

Setzt die Farbe des Gitternetzes auf die übergebene Farbe.

```
public void setDomainLeft(double newDomainLeft)
{
    domainLeft = newDomainLeft;
}
```

Setzt die linke Grenze des Definitionsbereichs auf den übergebenen Double-Wert.

```
public void setDomainRight(double newDomainRight)
{
    domainRight = newDomainRight;
}
```

Setzt die rechte Grenze des Definitionsbereichs auf den übergebenen Double-Wert.

```
public void setX0(double newX0)
{
    x0 = newX0;
}
```

Mit dieser Funktion kann der X-Wert des Nullpunktes neu gesetzt werden.

```
public void setXFactor(double newXFactor)
{
    xFactor = newXFactor;
}
```

Mit der Methode „setXFactor(...)“ kann der Faktor zur Skalierung der X-Achse gesetzt werden.

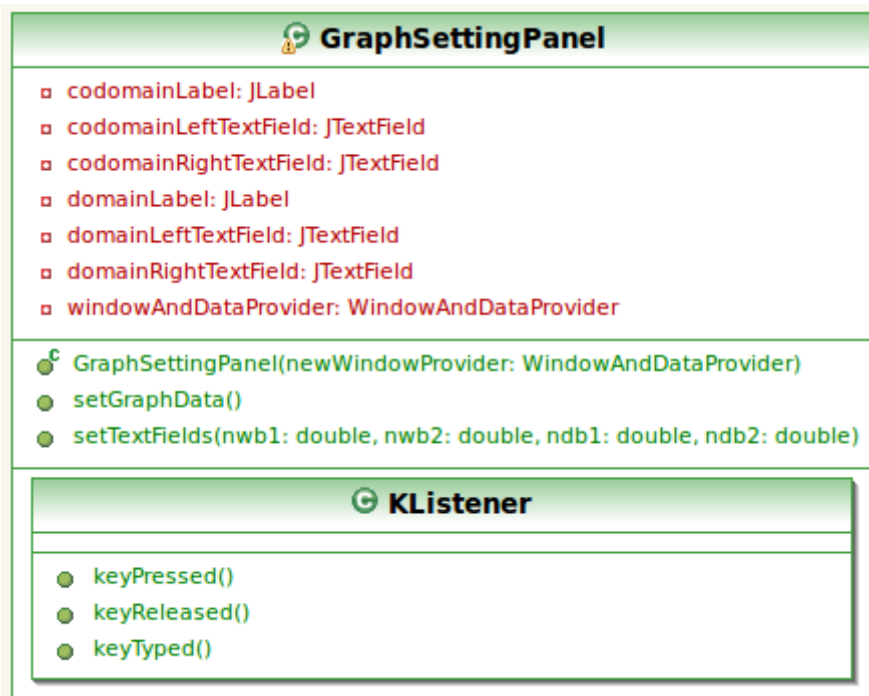
```
public void setY0(double newY0)
{
    y0 = newY0;
}
```

„setY0(...)“ setzt den übergebenen Double-Wert als neue Y-Wert des Nullpunktes.

```
public void setYFactor(double newYFactor)
{
    yFactor = newYFactor;
}
```

Mit der Methode „setYFactor(...)“ kann der Faktor zur Skalierung der Y-Achse gesetzt werden.

GraphSettingPanel



Die Klasse „GraphSettingPanel“ erbt von der Klasse „JPanel“. Sie ist ein Panel, auf dem sich vier Textfelder befinden. Diese stellen eine Möglichkeit für den Benutzer dar, den Wertebereich und Definitionsbereich des Graphen festzulegen. Wird eines dieser Felder editiert, so werden die Inhalte ausgelesen und für den Graphen gespeichert.

Ansicht des Panels:

Graph-Einstellungen	
Definitionsbereich:	<input type="text" value="-4"/> <input type="text" value="4"/>
Wertebereich:	<input type="text" value="-4"/> <input type="text" value="4"/>

Konstruktor:

```
public GraphSettingPanel(WindowAndDataProvider newWindowProvider)
{
    windowAndDataProvider = newWindowProvider;

    setLayout(new GridLayout(2, 3));
    setBorder(new TitledBorder("Graph-Einstellungen"));

    domainLabel = new JLabel("Definitionsbereich:");
    add(domainLabel);

    domainLeftTextField = new JTextField("-4");
    domainLeftTextField.addKeyListener(new KListener());
    add(domainLeftTextField);

    domainRightTextField = new JTextField("4");
    domainRightTextField.addKeyListener(new KListener());
    add(domainRightTextField);

    codomainLabel = new JLabel("Wertebereich:");
    add(codomainLabel);

    codomainLeftTextField = new JTextField("-4");
    codomainLeftTextField.addKeyListener(new KListener());
    add(codomainLeftTextField);

    codomainRightTextField = new JTextField("4");
    codomainRightTextField.addKeyListener(new KListener());
    add(codomainRightTextField);

    setGraphData();
}
```

Der Konstruktor ist für die Erstellung der Grafischen Oberfläche zuständig. Um die einzelnen Elemente anzuordnen wird dem Panel ein GridLayout mit zwei Zeilen und jeweils drei Spalten zugewiesen. Danach wird das Label zur Beschriftung des Definitionsbereichs angelegt und dem Panel hinzugefügt. Damit hinter dem Label die beiden Textfelder erscheinen, müssen diese nun erstellt und hinzugefügt werden. Mit den Steuerelementen des Wertebereichs muss genauso verfahren werden. Außerdem ist noch zu sagen, das sowohl der Definitionsbereich wie auch der Wertebereich mit $[-4;4]$ initialisiert werden.

Methoden

```
public void setTextFields(double nwb1, double nwb2, double ndb1, double ndb2)
{
    codomainLeftTextField.setText(String.valueOf(nwb1));
    codomainRightTextField.setText(String.valueOf(nwb2));
    domainLeftTextField.setText(String.valueOf(ndb1));
    domainRightTextField.setText(String.valueOf(ndb2));
}
```

Die Methode „setTextFields(...)“ setzt die Textfelder des Panels auf die Werte der übergebenen Parameter. Die Parameter sind in folgender Reihenfolge zu übergeben:

1. untere Grenze des Wertebereichs
2. obere Grenze des Wertebereichs
3. linke Grenze der Definitionsbereichs
4. rechte Grenze der Definitionsbereichs

```

public void setGraphData()
{
    if(codomainLeftTextField.getBackground().equals(Color.white) &&
codomainRightTextField.getBackground().equals(Color.white) &&
domainLeftTextField.getBackground().equals(Color.white) &&
domainRightTextField.getBackground().equals(Color.white))
    {

        windowAndDataProvider.getGraphData().setDomainLeft(Double.parseDouble(domainLeftText
extField.getText()));

        windowAndDataProvider.getGraphData().setDomainRight(Double.parseDouble(domainRightT
extField.getText()));

        windowAndDataProvider.getGraphData().setCodomainLeft(Double.parseDouble(codomainLef
tTextField.getText()));

        windowAndDataProvider.getGraphData().setCodomainRight(Double.parseDouble(codomainRi
ghtTextField.getText()));
    }
}

```

„setGraphData(...)“ ist eine Methode welche die aktuellen Werte in den Textfeldern ausliest und sie im GraphData-Objekt speichert. Dazu werden sie jedoch zuvor in Double-Werte umgewandelt.

Innere Klassen:

```

public class KListener implements KeyListener
{
    public void keyPressed(KeyEvent e)
    {
        ;
    }

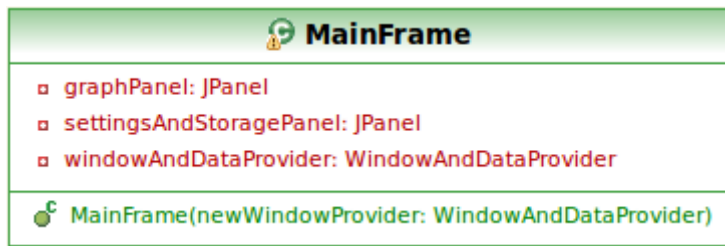
    public void keyReleased(KeyEvent e)
    {
        try
        {
            Double.parseDouble(((JTextField)e.getSource()).getText());
            ((JTextField)e.getSource()).setBackground(Color.white);
            setGraphData();
            windowAndDataProvider.graphRepaint(true);
        }
        catch(NumberFormatException ee)
        {
            ((JTextField)e.getSource()).setBackground(Color.red);
        }
    }

    public void keyTyped(KeyEvent e)
    {
        ;
    }
}

```

Mit dieser Klasse wird der Hintergrund des aktiven Textfeldes rot gefärbt, wenn eine ungültige Eingabe getätigt wird.

MainFrame



Die Klasse „MainFrame“ fügt alle Oberflächenelemente der Anwendung zusammen. Dazu erbt sie von der Klasse „JFrame“, welche es unter anderem ermöglicht Objekte der Klasse „JPanel“ hinzuzufügen.

Konstruktor:

```
public MainFrame(WindowAndDataProvider newWindowProvider)
{
    windowAndDataProvider = newWindowProvider;

    setTitle("Lernpfad: Integralrechnung");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setMinimumSize(new Dimension(480, 500));
    setLayout(new BorderLayout());
    setVisible(true);

    settingsAndStoragePanel = new JPanel();
    settingsAndStoragePanel.setLayout(new GridLayout(0, 1));
    settingsAndStoragePanel.add(windowAndDataProvider.getSettingPanel());
    settingsAndStoragePanel.add(windowAndDataProvider.getSaveLoadPanel());

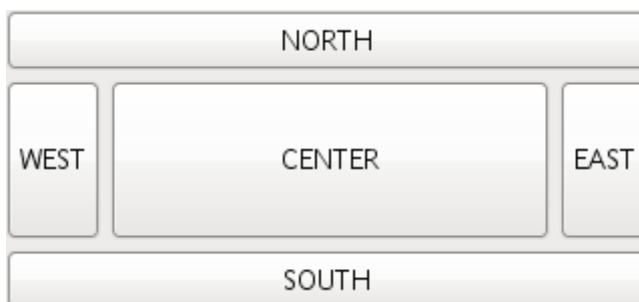
    graphPanel = new JPanel();
    graphPanel.setLayout(new GridLayout(1, 1));
    graphPanel.setBorder(new TitledBorder("Graph"));

    graphPanel.add(windowAndDataProvider.getGraph());

    add(BorderLayout.NORTH, windowAndDataProvider.getInputPanel());
    add(BorderLayout.CENTER, graphPanel);
    add(BorderLayout.SOUTH, settingsAndStoragePanel);

    pack();
}
```

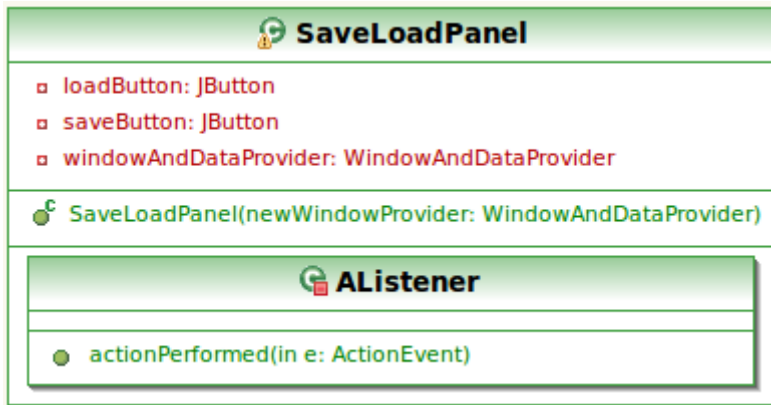
Der Konstruktor legt zuerst den Fenstertitel fest. Danach wird bestimmt, was passieren soll, wenn das Fenster geschlossen wird. Des Weiteren wird eine Minimalgröße von 480x500 Pixel festgelegt. Damit die Anwendung auch beim verändern der Fenstergröße ihre Oberfläche anpasst, wird an dieser Stelle auf ein so genanntes BorderLayout zurückgegriffen. Dies ist wie folgt strukturiert:



An der Position North wurde dem JFrame ein Objekt der Klasse „FunctionInputPanel“ hinzugefügt. Darunter also auf der Position Center ein Panel welches den Graphen enthält. Und zu guter letzt noch ein JPanel an letzter Stelle, welches ein Objekt der Klasse

„GraphSettingPanel“ und „SaveLoadPanel“ enthält. Die Wahl eines Layouts hat den Vorteil, dass sich die Fensterinhalte bei Veränderungen wie zum Beispiel der Größe neu ausrichten.

SaveLoadPanel



Wie alle anderen Klassen welche das Wort „Panel“ in ihrem Klassennamen tragen, handelt es sich auch hier wieder um ein Oberflächenelement der Anwendung, welches Methoden und Attribute von der Klasse „JPanel“ erbt. Die Aufgabe besteht darin, ein Panel zu erstellen, worauf sich zwei Buttons befinden. Wobei Einer zum Speichern und der Andere für das Laden zuständig ist.

Ansicht des Panels:



Konstruktor:

```
public SaveLoadPanel(WindowAndDataProvider newWindowProvider)
{
    windowAndDataProvider = newWindowProvider;

    setLayout(new FlowLayout(FlowLayout.LEFT));
    setBorder(new TitledBorder("Graph Speichern und Laden"));

    saveButton = new JButton("Speichern unter");
    saveButton.addActionListener(new AListener());
    add(saveButton);

    loadButton = new JButton("Aus Datei Laden");
    loadButton.addActionListener(new AListener());
    add(loadButton);
}
```


Wie schon in den vorherigen Klassen zur Anzeige von Benutzerelementen wird die Oberfläche auch hier wieder im Konstruktor gestaltet und mit dem entsprechenden Listener verknüpft.

Innere Klassen:

```
private class AListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource().equals(saveButton))
        {
            windowAndDataProvider.saveFileDialog();
        }
        else if(e.getSource().equals(loadButton))
        {
            windowAndDataProvider.loadFileDialog();
        }
    }
}
```

Diese Innere Klasse implementiert einen ActionListener, welcher die Methode „actionPerformed(...)“ enthält. Diese wird wie bereits im Konstruktor definiert aufgerufen, wenn der Benutzer einen Button drückt. In Abhängigkeit vom gedrückten Button wird dann entweder die Methode „saveFileDialog()“ oder „loadFileDialog()“ der Klasse „WindowAndDataProvider“ aufgerufen. Diese leitet dann die entsprechenden Schritte zum Speichern bzw. Laden der Daten ein.

Function

 **Function**

- ▣ aufleitung: String
- ▣ color: Color
- ▣ drawLowerSum: boolean
- ▣ drawUpperSum: boolean
- ▣ dx: double
- ▣ intervalLeft: double
- ▣ intervalRight: double
- ▣ points: ArrayList<DPoint>
- ▣ steps: double
- ▣ term: String
- ▣ visible: boolean

- 🔗 Function(newTerm: String, domainLeft: double, domainRight: double)
- 🔗 calcFunctionData(domainLeft: double, domainRight: double)
- 🔗 calcIntegral(): double
- 🔗 calcObersumme(): double
- 🔗 calcUntersumme(): double
- 🔗 drawFunction(windowAndDataProvider: WindowAndDataProvider, g: Graphics, domainLeft: double, domainRight: double)
- 🔗 drawObersumme(windowAndDataProvider: WindowAndDataProvider, g: Graphics)
- 🔗 drawUntersumme(windowAndDataProvider: WindowAndDataProvider, g: Graphics)
- 🔗 getAufleitung(): String
- 🔗 getIntervall1(): double
- 🔗 getIntervall2(): double
- 🔗 getSteps(): double
- 🔗 getTerm(): String
- 🔗 inputsFunction(function: String): boolean
- 🔗 isObersummeVisible(): boolean
- 🔗 isUntersummeVisible(): boolean
- 🔗 isVisible(): boolean
- 🔗 setIntervall1(int1: double)
- 🔗 setIntervall2(int2: double)
- 🔗 setObersummeVisible(visible: boolean)
- 🔗 setSteps(newSteps: double)
- 🔗 setTerm(newTerm: String, domainLeft: double, domainRight: double)
- 🔗 setUntersummeVisible(visible: boolean)
- 🔗 setVisible(newVisible: boolean)

Die Klasse „Function“ ist einer der wichtigsten Bestandteile der Anwendung. Sie speichert die Daten einer, vom Benutzer definierten, Funktion und bietet diverse Methoden um Berechnungen durchzuführen oder grafische Darstellungen zu handhaben. In der Klasse werden folgende Daten gespeichert:

- ◆ der Funktionsterm
- ◆ der integrierte Funktionsterm
- ◆ die Farbe in welcher die Funktion gezeichnet wird
- ◆ der Abschnitt, in welchem die Obersumme, die Untersumme und der Integral berechnet und gezeichnet werden soll
- ◆ die Punkte zum Zeichnen der Kurve
- ◆ die Breite der Rechteckstreifen für Ober- und Untersumme
- ◆ die Anzahl der Rechteckstreifen
- ◆ Variablen, die festlegen ob die Funktion und die Ober- und Untersumme gezeichnet werden sollen

Konstruktor:

```
public Function(String newTerm, double domainLeft, double domainRight)
{
    visible = true;
    term = newTerm;
    aufleitung = Calculator.aufleiten(term);
    color = Color.red;
    intervalLeft = -2;
    intervalRight = 2;
    steps = 10;
    drawUpperSum = false;
    drawLowerSum = false;

    calcFunctionData(domainLeft, domainRight);
}
```

Der Konstruktor erwartet als Übergabeparameter einen Funktionsterm als String wie auch die linke und rechte Grenze des Definitionsbereiches in Form zweier Double-Variablen. Die übergebenen Werte werden in die entsprechenden Klassenattribute gesichert und der Abschnitt für Ober- und Untersumme sowie den Integral wird auf $[-2; 2]$ festgelegt. Da zu diesem Zeitpunkt der Funktionsterm bereits bekannt ist, wird mithilfe der Klasse „Calculator“ gleich noch die Integrierte Funktion erstellt und abgespeichert. Auch die Anzahl der Rechteckstreifen wird auf zehn voreingestellt. Außerdem ist anzumerken, dass das Zeichnen der Rechteckstreifen standardmäßig deaktiviert ist. Sind all diese Attribute soweit festgelegt, wird abschließend noch die Methode „calcFunctionData(...)“ aufgerufen.

Methoden

```
public static boolean inputIsFunction(String function)
{
    boolean right = true;
    char c;

    if(function.length() == 0)
    {
        return false;
    }

    for(int i=0; i<function.length(); i++)
    {
        c = function.charAt(i);

        if(c != 'x' && c != '+' && c != '-' && c != '*' && c != '/' && c != '^' &&
c != '^' && c != '.' && c != ',' && c != '(' && c != ')' && (c < '0' || c > '9'))
        {
            right = false;
            break;
        }
    }

    return right;
}
```

Mithilfe dieser Methode lässt sich feststellen ob der, in Form eines Parameters übergebener, Funktionsterm Zeichen enthält die nicht für eine Polynomfunktion typisch sind. Ist dies der Fall, gibt die Methode „false“ zurück. Andererseits wird „true“ zurückgegeben. Jedoch heißt dies nicht, dass es sich wirklich um eine Polynomfunktion handelt.

```

public void calcFunctionData(double domainLeft, double domainRight)
{
    double tmp;
    String var;

    dx = (intervalRight-intervalLeft)/steps;

    if(intervalLeft<domainLeft)
    {
        domainLeft = intervalLeft;
    }

    if(intervalRight>domainRight)
    {
        domainRight = intervalRight;
    }

    if(points == null)
    {
        ArrayList<DPoint> newPoints = new ArrayList<DPoint>();

        for(double i=domainLeft; i<=domainRight; i+=0.0005)
        {
            NumberFormat nf = new DecimalFormat("####");
            String h = nf.format(i);

            var = term.replaceAll("x", "("+h+")");
            tmp = Calculator.allesRechnen(var);

            newPoints.add(new DPoint(i, tmp));
        }

        points = newPoints;
    }

    if(domainLeft<points.get(0).getX())
    {
        ArrayList<DPoint> newPoints = new ArrayList<DPoint>();

        for(double i=domainLeft; i<points.get(0).getX(); i+=0.0005)
        {
            NumberFormat nf = new DecimalFormat("####");
            String h = nf.format(i);

            var = term.replaceAll("x", "("+h+")");
            tmp = Calculator.allesRechnen(var);

            newPoints.add(new DPoint(i, tmp));
        }

        newPoints.addAll(points);

        points = newPoints;
    }

    if(domainRight>points.get(points.size()-1).getX())
    {
        ArrayList<DPoint> newPoints = new ArrayList<DPoint>();

        for(double i=points.get(points.size()-1).getX(); i<domainRight; i+=0.0005)
        {
            NumberFormat nf = new DecimalFormat("####");
            String h = nf.format(i);

            var = term.replaceAll("x", "("+h+")");
            tmp = Calculator.allesRechnen(var);

            newPoints.add(new DPoint(i, tmp));
        }

        points.addAll(newPoints);
    }
}

```

Mit der Methode „calcFunctionData(...)“ werden die zum Zeichnen benötigten Punkte berechnet. Übergeben werden ihr die linke und rechte Grenze des Definitionsbereiches als

Double-Variablen. Damit nicht bei jeder Änderung des Definitionsbereiches alle Punkte neu berechnet werden müssen, ist die Funktionsweise so ausgelegt, dass immer nur die Punkte berechnet werden, welche noch nicht in der Liste vorhanden sind. Das hat den Vorteil, dass Rechenzeit gespart wird, aber auch den Nachteil, dass die Datenmengen schnell anwachsen können.

```
public double calcIntegral()
{
    String rechnung, term1, term2;

    term1 = aufleitung.replaceAll("x", "("+String.valueOf(intervalLeft)+")");
    term2 = aufleitung.replaceAll("x", "("+String.valueOf(intervalRight)+")");

    rechnung = "("+term2+")-("+term1+")";

    return Calculator.allesRechnen(rechnung);
}
```

Wie der Name der Methode „calcIntegral()“ schon deutlich macht, wird hier das Integral der Polynomfunktion berechnet. Das Ergebnis wird anschließend als Double-Wert zurückgegeben.

```
public double calcObersumme()
{
    double hoehe, erg;

    erg = 0.0;

    for(double i=intervalLeft; i<=intervalRight-dx+(dx/4); i+=dx)
    {
        int j1, j2;

        j1 = 0;

        while(points.get(j1).getX()<=i && j1 < points.size()-1)
        {
            j1++;
        }

        j2 = j1;

        while(points.get(j2).getX()<=i+dx && j2 < points.size()-1)
        {
            j2++;
        }

        hoehe = points.get(j1).getY();

        for(int j=j1; j<=j2; j++)
        {
            if(points.get(j).getY()>hoehe)
            {
                hoehe = points.get(j).getY();
            }
        }

        erg += dx*hoehe;
    }

    return erg;
}
```

„calcObersumme()“ ist für die Berechnung der Obersumme zuständig. Für die Berechnung greift die Methode auf die Daten zu, welche mittels „calcFunctionData(...)“ erstellt wurden. Ist das Ergebnis fertig berechnet, wird es als Rückgabeparameter vom Typ double zurückgegeben.

```

public double calcUntersumme()
{
    double hoehe, erg;
    erg = 0.0;

    for(double i=intervalLeft; i<=intervalRight-dx+(dx/4); i+=dx)
    {
        int j1, j2;

        j1 = 0;

        while(points.get(j1).getX()<=i && j1 < points.size()-1)
        {
            j1++;
        }

        j2 = j1;

        while(points.get(j2).getX()<=i+dx && j2 < points.size()-1)
        {
            j2++;
        }

        hoehe = points.get(j1).getY();

        for(int j=j1; j<=j2; j++)
        {
            if(points.get(j).getY()<hoehe)
            {
                hoehe = points.get(j).getY();
            }
        }

        erg += dx*hoehe;
    }

    return erg;
}

```

Auch in diesem Fall lässt dich über den Name des Unterprogramms auf seine Aufgabe schließen. Wie auch schon bei der Obersummenberechnung wird hier ebenfalls auf die Daten zugegriffen, welche von der Methode „calcFunctionData(...)“ erstellt wurden. Das Ergebnis der Berechnungen wird auch hier wieder in Form eines Doubles zurückgegeben.

```

public void drawFunction(WindowAndDataProvider windowAndDataProvider, Graphics g, double
domainLeft, double domainRight)
{
    if(visible == true)
    {
        DPoint p1, p2;
        int j1, j2;

        p1 = new DPoint();
        p2 = new DPoint();
        g.setColor(color);

        j1 = 0;

        while(points.get(j1).getX()<=domainLeft && j1 < points.size()-1)
        {
            j1++;
        }

        j1--;

        j2 = j1;

        while(points.get(j2).getX()<=domainRight && j2 < points.size()-1)
        {
            j2++;
        }

        for(int i=j1; i<j2; i++)
        {
            if(points.size()-1 != i)
            {
                p1 =
windowAndDataProvider.toCoordinationPoint(points.get(i));
                p2 =
windowAndDataProvider.toCoordinationPoint(points.get(i+1));
            }
            else
            {
                p1 =
windowAndDataProvider.toCoordinationPoint(points.get(i));
                p2 =
windowAndDataProvider.toCoordinationPoint(points.get(i));
            }

            g.drawLine((int) Math.round(p1.getX()), (int) Math.round(p1.getY()),
(int) Math.round(p2.getX()), (int) Math.round(p2.getY()));
        }
    }
}

```

Die Methode „drawFunction(...)“ ist für das Zeichnen der Polynomfunktion zuständig. Deshalb ist einer der Übergabeparameter ein Graphics-Objekt, welches als Zeichenfläche dient. Außerdem wird auch ein Objekt der Klasse „WindowAndDataProvider“ übergeben, um die Methode zur Umrechnung der mathematischen Koordinaten in Java-Koordinaten benutzen zu können. Zu guter letzt wird der Definitionsbereich des Graphen in Form zweier Double-Variablen übergeben. Damit nur der Sichtbare Teil der Polynomfunktion gezeichnet wird, sucht die Methode mit Hilfe einer Schleife den Punkt mit dem X-Wert, welcher am den kleinsten Abstand zur linken Grenze des Definitionsbereiches hat. Dasselbe wird auch für den rechten Grenzwert durchgeführt. Nun müssen nur noch die Punkte, welche zwischen den beiden Grenzen liegen immer mit dem jeweils nächsten verbunden werden. Doch bevor dies geschieht, müssen die zu verbindenden Punkte immer in das Java-Koordinatensystem umgerechnet werden.

```

public void drawObersumme(WindowAndDataProvider windowAndDataProvider, Graphics g)
{
    if(drawUpperSum == true)
    {
        double hoehe;
        DPoint p1, p2;

        p1 = new DPoint();
        p2 = new DPoint();

        for(double i=intervalLeft; i<=intervalRight-dx+(dx/4); i+=dx)
        {
            int j1, j2;

            j1 = 0;

            while(points.get(j1).getX()<=i && j1 < points.size()-1)
            {
                j1++;
            }

            j2 = j1;

            while(points.get(j2).getX()<=i+dx && j2 < points.size()-1)
            {
                j2++;
            }

            hoehe = points.get(j1).getY();

            for(int j=j1; j<=j2; j++)
            {
                if(points.get(j).getY()>hoehe)
                {
                    hoehe = points.get(j).getY();
                }
            }

            p1.setX(i);
            p1.setY(0);
            p2.setX(i+dx);
            p2.setY(hoehe);

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            int[] x = new int[4];
            int[] y = new int[4];

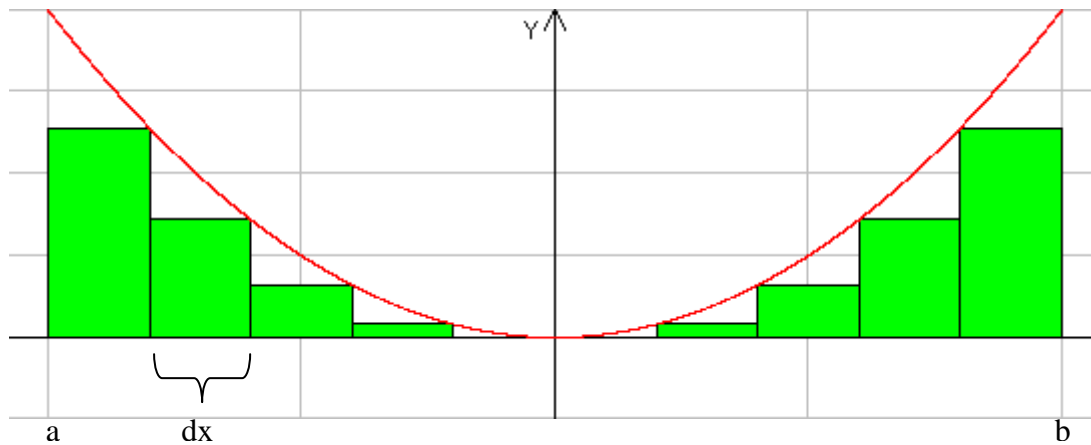
            x[0] = (int) Math.round(p1.getX());
            x[1] = (int) Math.round(p1.getX());
            x[2] = (int) Math.round(p2.getX());
            x[3] = (int) Math.round(p2.getX());

            y[0] = (int) Math.round(p1.getY());
            y[1] = (int) Math.round(p2.getY());
            y[2] = (int) Math.round(p2.getY());
            y[3] = (int) Math.round(p1.getY());

            g.setColor(Color.blue);
            g.fillPolygon(x, y, x.length);
            g.setColor(Color.black);
            g.drawPolygon(x, y, x.length);
        }
    }
}

```

Die Methode „drawObersumme(...)“ wird aufgerufen, um die Obersumme der Funktion zeichnen zu lassen. Daher ist einer der Übergabeparameter ein Graphics-Objekt, das die Zeichenfläche darstellt. Außerdem wird ein Objekt der Klasse „WindowAndDataProvider“ erwartet, dessen Methode zur Umrechnung der Koordinatenpunkte benötigt wird.



Um die Rechteckstreifen der Obersumme einzuzichnen, wird von der linken Grenze des Intervalls $[a; b]$ begonnen, den größten Y-Wert im Bereich dx zu suchen. Ist dieser gefunden, wird ein Punkt erzeugt, welcher den eben ermittelten Y-Wert und den kleinsten X-Wert in diesem Bereich enthält. Danach wird ein weiterer Punkt erstellt, welcher als Y-Koordinate den Wert null und als X-Koordinate den größten X-Wert im Bereich dx hat. Diese Punkte werden nun in Java-Koordinaten konvertiert. Mithilfe dieser wird dann ein ausgefülltes Rechteck und dessen Rand gezeichnet. Um die restlichen Rechteckstreifen darzustellen, wird der zu bearbeitende Bereich Abschnitt um dx nach rechts verschoben und es wird wieder von vorne begonnen.


```

public void drawUntersumme(WindowAndDataProvider windowAndDataProvider, Graphics g)
{
    if(drawLowerSum == true)
    {
        double hoehe;
        DPoint p1, p2;

        p1 = new DPoint();
        p2 = new DPoint();

        for(double i=intervalLeft; i<=intervalRight-dx+(dx/4); i+=dx)
        {
            int j1, j2;

            j1 = 0;

            while(points.get(j1).getX()<=i && j1 < points.size()-1)
            {
                j1++;
            }

            j2 = j1;

            while(points.get(j2).getX()<=i+dx && j2 < points.size()-1)
            {
                j2++;
            }

            hoehe = points.get(j1).getY();

            for(int j=j1; j<=j2; j++)
            {
                if(points.get(j).getY()<hoehe)
                {
                    hoehe = points.get(j).getY();
                }
            }

            p1.setX(i);
            p1.setY(0);
            p2.setX(i+dx);
            p2.setY(hoehe);

            p1 = windowAndDataProvider.toCoordinationPoint(p1);
            p2 = windowAndDataProvider.toCoordinationPoint(p2);

            int[] x = new int[4];
            int[] y = new int[4];

            x[0] = (int) Math.round(p1.getX());
            x[1] = (int) Math.round(p1.getX());
            x[2] = (int) Math.round(p2.getX());
            x[3] = (int) Math.round(p2.getX());

            y[0] = (int) Math.round(p1.getY());
            y[1] = (int) Math.round(p2.getY());
            y[2] = (int) Math.round(p2.getY());
            y[3] = (int) Math.round(p1.getY());

            g.setColor(Color.green);
            g.fillPolygon(x, y, x.length);
            g.setColor(Color.black);
            g.drawPolygon(x, y, x.length);
        }
    }
}

```

Die Methode „drawUntersumme(...)“ funktioniert genau nach dem gleichen Muster wie ihr Gegenstück „drawObersumme(...)“, mit dem Unterschied, dass hier nicht der größte Wert im Bereich dx, sondern der kleinste ermittelt wird.

```
public String getAufleitung()
{
    return aufleitung;
}
```

Die Methode „getAufleitung()“ liefert einen String, welcher den Integrierten Funktionsterm enthält.

```
public double getIntervall1()
{
    return intervalLeft;
}
```

Gibt den einen Double Wert zurück, der die linke Integrationsgrenze beinhaltet.

```
public double getIntervall2()
{
    return intervalRight;
}
```

Gibt den einen Double Wert zurück, der die rechte Integrationsgrenze beinhaltet.

```
public double getSteps()
{
    return steps;
}
```

Mit „getSteps()“ wird die Anzahl der Rechteckstreifen für Ober- und Untersumme zurückgegeben.

```
public String getTerm()
{
    return term;
}
```

Gibt den Funktionsterm in einem String zurück.

```
public boolean isObersummeVisible()
{
    return drawUpperSum;
}
```

Hier wird ein Boolean zurückgegeben. Er gibt an, ob die Obersumme gezeichnet wird oder nicht.

```
public boolean isUntersummeVisible()
{
    return drawLowerSum;
}
```

Es wird wieder ein Boolean zurückgegeben, der angibt, ob die Untersumme angezeigt wird.

```
public boolean isVisible()
{
    return visible;
}
```

Gibt true oder false zurück, je nach dem ob die Polynomfunktion im Graph eingezeichnet werden soll oder nicht.

```
public void setIntervall1(double int1)
{
    intervalLeft = int1;
}
```

Mithilfe der Methode „setIntervall1(...)“ wird die linke Integrationsgrenze festgelegt.

```
public void setIntervall2(double int2)
{
    intervalRight = int2;
}
```

Mithilfe der Methode „setIntervall2(...)“ wird die rechte Integrationsgrenze festgelegt.

```
public void setObersummeVisible(boolean visible)
{
    drawUpperSum = visible;
}
```

Die Methode erlaubt es festzulegen, ob die Obersumme dargestellt wird oder nicht.

```
public void setSteps(double newSteps)
{
    steps = newSteps;
}
```

Die Methode erlaubt es festzulegen, ob die Untersumme dargestellt wird oder nicht.

```
public void setTerm(String newTerm, double domainLeft, double domainRight)
{
    term = newTerm;
    aufleitung = Calculator.aufleiten(term);
    points = null;
    calcFunctionData(domainLeft, domainRight);
}
```

„setTerm(...)“ erlaubt es den Funktionsterm festzulegen, bzw. zu ändern. Sinnvollerweise werden nach den Änderungen auch gleich noch die Daten und die Integrierte Funktion neu berechnet. Um all diese Dinge tun zu können werden der Methode beim Aufruf der neue Term, sowie die linke und rechte Grenze des Definitionsbereiches übergeben werden.

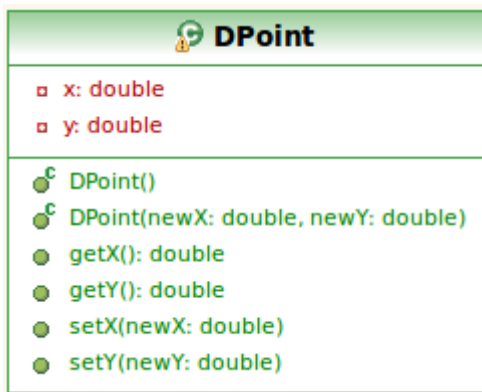
```
public void setUntersummeVisible(boolean visible)
{
    drawLowerSum= visible;
}
```

Anhand des Übergebenen Boolean wird festgelegt, ob die Obersumme gezeichnet werden soll oder nicht.

```
public void setVisible(boolean newVisible)
{
    visible = newVisible;
}
```

Anhand des Übergebenen Boolean wird festgelegt, ob die Untersumme gezeichnet werden soll oder nicht.

DPoint



Die Klasse „DPoint“ stellt dieselben Möglichkeiten dar, wie die Javaeigene Klasse „Point“. Der unterschied liegt in den Speichermöglichkeiten der Koordinaten. In der Mitgelieferten Klasse können nur Integer Zahlen benutzt werden. Da jedoch für diese Anwendung eine höhere Genauigkeit erforderlich ist, wurde eine ähnliche geschrieben, welche mit Double Werten umgehen kann.

Konstruktoren:

```
public DPoint()  
{  
    ;  
}
```

Dieser Konstruktor führt keinerlei Anweisungen aus und ist nur vorhanden, um ein Leeres Objekt erzeugen zu können.

```
public DPoint(double newX, double newY)  
{  
    x = newX;  
    y = newY;  
}
```

Im Gegensatz zum Vorherigen werden hier zwei Double-Werte übergeben. Hierbei ist ersterer der X und letzterer der Y-Wert für die zu initialisierende Koordinate.

Methoden

```
public double getX()  
{  
    return x;  
}
```

Gib den X-Wert als Double zurück.

```
public double getY()  
{  
    return y;  
}
```

Gib den Y-Wert als Double zurück.



```
public void setX(double newX)
{
    x = newX;
}
```

Setzt den X-Wert des Punktes.

```
public void setY(double newY)
{
    y = newY;
}
```

Setzt den Y-Wert des Punktes.

ObjectSerializer

ObjectSerializer	
	readFromFile(filename: String): Object
	writeToFile(filename: String, object: Object)

Die Klasse „ObjectSerializer“ stellt eine Möglichkeit dar, ganze Objekte speichern oder laden zu können. Die Methoden wurden in diesem Fall als static deklariert, da in der Klasse keine Daten gespeichert werden.

Methoden

```
public static void writeToFile(String filename, Object object) throws IOException
{
    ObjectOutputStream objOut = new ObjectOutputStream(new BufferedOutputStream(new
    FileOutputStream(filename)));
    objOut.writeObject(object);
    objOut.close();
}
```







Wie der Methodenname schon verrät, ist diese Methode dafür zuständig Objekte in eine Datei zu speichern/schreiben. Dazu muss der Methode als erster Parameter der Dateiname bzw. Dateipfad übergeben werden. Als zweites wird noch das abzuspeichernde Objekt erwartet.

```
public static Object readFromFile(String filename) throws IOException,
    ClassNotFoundException
{
    ObjectInputStream objIn = new ObjectInputStream(new BufferedInputStream(new
    FileInputStream(filename)));
    Object object = (Object) objIn.readObject();
    objIn.close();

    return object;
}
```

Die Methode „readFromFile(...)“ ist das Gegenstück zu „writeToFile(...)“. Daher erwartet es auch nur den Pfad zur Datei mit dem zu ladenden Objekt, und gibt dieses als Object-Objekt zurück.

Calculator

Calculator	
	<code>allesRechnen(input: String): double</code>
	<code>aufleiten(input: String): String</code>
	<code>calcArea(functionString1: String, functionString2: String): double</code>
	<code>grundoperatoren(rechnung: ArrayList<Element>)</code>
	<code>parseInput(input: String): ArrayList<Element></code>
	<code>parseInputFunction(input: String): ArrayList<Element></code>

Methoden

```
public static double allesRechnen(String input)
{
    double ergebnis = 0;
    ArrayList<Element> rechnung;

    rechnung = parseInput(input);

    for(int i=0; i<rechnung.size(); i++)
    {
        if(rechnung.get(i).isTerm())
        {
            rechnung.set(i, new Element(allesRechnen(rechnung.get(i).term)));
        }
    }

    grundoperatoren(rechnung);

    ergebnis = rechnung.get(0).zahl;

    return ergebnis;
}
```

Die Methode „allesRechnen(...)“ ist dazu gedacht, eine Rechnung, welche in einem String gespeichert ist auszurechnen. Dazu muss der String geparkt und dabei in seine Einzelteile zerlegt werden. Diese Aufgabe übernimmt die Methode „parseInput(...)“. Danach werden die einzelnen Rechnungselemente ausgerechnet und deren Ergebnisse wiederum zu einem Einzeigen verrechnet. Das wird anschließend als Double-Zahl zurückgegeben.

```
public static String aufleiten(String input)
{
    String ableitung = "";
    ArrayList<Element> rechnung;
    int zahlen = 0;

    rechnung = parseInputFunction(input);

    for(int i=0; i<rechnung.size(); i++)
    {
        if(rechnung.get(i).isCalcable())
        {
            rechnung.set(i, new
            Element(allesRechnen(rechnung.get(i).getString())));
        }
    }

    for(int i=0; i<rechnung.size(); i++)
    {
        if(rechnung.get(i).isZahl())
        {
            zahlen++;
        }
    }
}
```

```

if(zahlen > 1)
{
    String a, b, s;
    boolean aa, bb;

    a = "";
    b = "";
    s = "";
    aa = false;
    bb = false;

    for(int i=0; i<rechnung.size(); i++)
    {
        if(rechnung.get(i).isZahl())
        {
            if(aa == false)
            {
                a = rechnung.get(i).getString();
                rechnung.set(i, new Element(0.0));
                aa = true;
            }
            else
            {
                s = rechnung.get(i-1).getString();
                b = rechnung.get(i).getString();
                bb = true;
            }
        }

        if(aa && bb)
        {
            rechnung.set(i, new
Element(Calculator.allesRechnen("(" + a + ")" + s + "(" + b + ")" ));
            aa = false;
            bb = false;
        }
    }

    for(int i=0; i<rechnung.size(); i++)
    {
        if(rechnung.get(i).isTerm())
        {
            String term = rechnung.get(i).term;
            char c;
            double e;
            boolean exp = false;

            for(int j=0; j<term.length(); j++)
            {
                c = term.charAt(j);

                if(c == '^')
                {
                    exp = true;
                    break;
                }
            }

            if(exp == false)
            {
                term = term.replaceAll("x", "x^1");
            }

            for(int j=0; j<term.length(); j++)
            {
                c = term.charAt(j);

                if(c == 'x')
                {
                    if(term.charAt(j+1) == '^')
                    {

```

```

        term.length()-1));
        e = Double.valueOf(term.substring(j+2,
        term = term.substring(0, term.length()-
1)+"*x/"+(e+1.0)+"");
    }
}
rechnung.set(i, new Element(term));
}
for(int i=0; i<rechnung.size(); i++)
{
    if(rechnung.get(i).isZahl())
    {
        rechnung.set(i, new
Element("(" +rechnung.get(i).getString()+"*x"+""));
    }
}
for(int i=0; i<rechnung.size(); i++)
{
    ableitung += rechnung.get(i).getString();
}
return ableitung;
}

```

Die Methode „aufleiten(...)“ tut genau das was auch schon ihr Name verkündet. Will man eine Polynomfunktion aufleiten, muss diese als einziger Parameter übergeben werden. Diese wird dann mittels einer anderen Methode in Teile aufgeteilt, welche dann aufgeleitet werden. Als Ergebnis erhält man dann einen String der die integrierte Polynomfunktion enthält.

```

public static ArrayList<Element> parseInputFunction(String input)
{
    ArrayList<Element> rechnung;
    String buffer;
    char c;
    boolean firstChar;

    rechnung = new ArrayList<Element>();
    buffer = "";
    firstChar = true;

    input = input.replace(',', '.');
    input = input.replace(':', '/');

    for(int i=0; i<input.length(); i++)
    {
        c = input.charAt(i);

        if(!firstChar && (c == '+' || c == '-'))
        {
            rechnung.add(new Element("(" +buffer+""));
            rechnung.add(new Element(c, false));

            buffer = "";
        }
        else
        {
            buffer += c;
        }

        firstChar = false;
    }

    if(!buffer.equals(""))
    {
        rechnung.add(new Element("(" +buffer+""));
    }

    return rechnung;
}

```


Die Methode „parseInputFunction(...)“ geht den Übergebenen String Zeichen für Zeichen durch und wenn sie ein Plus oder Minus findet wird der Term zwischen dem letzten und neuen Fund in ein Element-Objekt gespeichert und der ArrayList angehängt. Danach wird auch das gefundene Rechenzeichen selbst noch in einem neuen Objekt der Klasse „Element“ gesichert. Ist der Suchvorgang am Ende der Zeichenkette angelangt, wird auch noch der letzte Term in der ArrayList hinterlegt. Eben diese Liste wird auch als Rückgabewerte erwendet.

```
public static ArrayList<Element> parseInput(String input)
{
    ArrayList<Element> rechnung;
    String buffer;
    char c;
    int klammern;
    boolean first;

    rechnung = new ArrayList<Element>();
    buffer = "";
    klammern = 0;
    first = true;

    input = input.replaceAll("t", "");
    input = input.replace('.', ',');
    input = input.replace(':', '/');

    for(int i=0; i<input.length(); i++)
    {
        c = input.charAt(i);

        if((c >= '0' && c <= '9') || c == '.')
        {
            buffer += c;
        }
        else if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
        {
            if(first)
            {
                buffer += c;
            }
            else
            {
                if(klammern == 0)
                {
                    if(!buffer.equals(""))
                    {
                        rechnung.add(new
Element(Double.parseDouble(buffer)));
                    }

                    rechnung.add(new Element(c, false));

                    buffer = "";
                }
                else
                {
                    buffer += c;
                }
            }
        }
        else if(c == '(')
        {
            if(klammern == 0)
            {
                c = 't';
                buffer += c;
            }
            else
            {
                buffer += c;
            }

            klammern++;
        }
    }
}
```

```

        else if(c == ')')
        {
            klammern--;

            if(klammern == 0)
            {
                rechnung.add(new Element(buffer));

                buffer = "";
            }
            else
            {
                buffer += c;
            }
        }

        first = false;
    }

    if(!buffer.equals(""))
    {
        rechnung.add(new Element(Double.parseDouble(buffer)));
    }

    return rechnung;
}

```

Das Unterprogramm „parseInput(...)“ erfüllt fast den gleichen Effekt wie „parse InputFunction(...)“ mit dem Unterschied, dass der Parser der Liste mit Rechnungselementen bei jedem Rechenzeichen ein Element-Objekt hinzufügt. Eine Ausnahme stellen hier Terme dar, die Von Klammern eingeschlossen sind. Diese Werden als ein Element abgespeichert. Auch hier wird die erstellte Liste als Rückgabeparameter verwendet.

```

public static void grundoperatoren(ArrayList<Element> rechnung)
{
    for(int i=0; i<rechnung.size(); i++)
    {
        Element e = rechnung.get(i);

        if(e.isSign())
        {
            if(e.getString().equals("^"))
            {
                double a,b, erg;

                a = rechnung.get(i-1).zahl;
                b = rechnung.get(i+1).zahl;

                erg = Math.pow(a, b);

                i--;

                rechnung.remove(i);
                rechnung.remove(i);

                rechnung.set(i, new Element(erg));
            }
        }
    }

    for(int i=0; i<rechnung.size(); i++)
    {
        Element e = rechnung.get(i);

        if(e.isSign())
        {
            if(e.getString().equals("*"))
            {
                double a,b, erg;

                a = rechnung.get(i-1).zahl;
                b = rechnung.get(i+1).zahl;
            }
        }
    }
}

```

```

        erg = a*b;

        i--;

        rechnung.remove(i);
        rechnung.remove(i);

        rechnung.set(i, new Element(erg));
    }
    else if(e.getString().equals("/"))
    {
        double a,b, erg;

        a = rechnung.get(i-1).zahl;
        b = rechnung.get(i+1).zahl;

        erg = a/b;

        i--;

        rechnung.remove(i);
        rechnung.remove(i);

        rechnung.set(i, new Element(erg));
    }
}

for(int i=0; i<rechnung.size(); i++)
{
    Element e = rechnung.get(i);

    if(e.isSign())
    {
        if(e.getString().equals("+"))
        {
            double a,b, erg;

            a = rechnung.get(i-1).zahl;
            b = rechnung.get(i+1).zahl;

            erg = a+b;

            i--;

            rechnung.remove(i);
            rechnung.remove(i);

            rechnung.set(i, new Element(erg));
        }
        else if(e.getString().equals("-"))
        {
            double a,b, erg;

            a = rechnung.get(i-1).zahl;
            b = rechnung.get(i+1).zahl;

            erg = a-b;

            i--;

            rechnung.remove(i);
            rechnung.remove(i);

            rechnung.set(i, new Element(erg));
        }
    }
}
}

```

„grundoperationen(...)“ ist eine Methode, welche als Übergabewert eine ArrayList mit Element-Objekten erwartet. Die in diesen Elementen Rechnung wird dann nach der Regel „Punkt vor Strich“ berechnet. Dazu werden zuerst alle Elemente der Rechnung auf das '^'-Zeichen durchsucht und ausgerechnet, falls eines oder mehrere auftreten.

Wenn dies durchgeführt ist, wird das Gleiche erst mit `*` und `/` -Zeichen und anschließend mit `+` und `-` gemacht. Jetzt sind alle Rechnungsbestandteile miteinander berechnet worden und das Ergebnis wird in das erste Element der ArrayList zurückgespeichert.

```
public static double calcArea(String functionString1, String functionString2)
{
    double result, tmp1, tmp2;
    String aufleitung1, aufleitung2;

    aufleitung1 = aufleiten(functionString1);
    aufleitung2 = aufleiten(functionString2);

    tmp1 = Math.abs(allesRechnen(aufleitung1.replaceAll("x", "(-2)")) -
    allesRechnen(aufleitung2.replaceAll("x", "(-2)")));
    tmp2 = Math.abs(allesRechnen(aufleitung1.replaceAll("x", "(2)")) -
    allesRechnen(aufleitung2.replaceAll("x", "(2)")));


    result = tmp1 - tmp2;

    System.out.println("tmp1: "+tmp1);
    System.out.println("tmp2: "+tmp2);

    return result;
}
```

Die Methode „calcArea(...)“ ist noch ein Relikt aus den ersten Versuchen der Integral- und Flächenberechnung. Allerdings wurde sie nicht entfernt, da sie zu gegebener Zeit vielleicht doch noch realisiert wird.

Element

 Element
<ul style="list-style-type: none"> ▣ isCalcable: boolean ▣ isSign: boolean ▣ isTerm: boolean ▣ isVar: boolean ▣ isZahl: boolean ◇ sign: char ◇ term: String ◇ variable: char ◇ zahl: double
<ul style="list-style-type: none"> ⦿ Element(newTerm: String) ⦿ Element(c: char, isVariable: boolean) ⦿ Element(newZahl: double) ● getString(): String ● isCalcable(): boolean ● isSign(): boolean ● isTerm(): boolean ● isVar(): boolean ● isZahl(): boolean ● setCalcable(newCalcable: boolean)

Ist eine Klasse zum speichern eines Terms, Rechenzeichens oder einer Zahl. Durch die verschiedenen Konstruktoren werden die Klassenattribute so gesetzt, dass das Objekt nach außen hin als eines der genannten Möglichkeiten erscheint.

Konstrukturen

```
public Element(String newTerm)
{
    term = newTerm;

    isCalcable = true;

    for(int i=0; i<newTerm.length(); i++)
    {
        if(newTerm.charAt(i) == 'x')
        {
            isCalcable = false;
        }
    }

    isTerm = true;
    isZahl = false;
    isSign = false;
}
```

Dieser Konstruktor setzt die Attribute so, dass das Objekt als Term gehandhabt werden kann.

```
public Element(double newZahl)
{
    zahl = newZahl;

    isTerm = false;
    isZahl = true;
    isSign = false;
}
```

Dieser Konstruktor setzt die Attribute so, dass das Objekt als ZahlTerm gehandhabt werden kann.

```
public Element(char c, boolean isVariable)
{
    if(isVariable == true)
    {
        System.out.println(c);
        variable = c;
        isVar = true;
        isSign = false;
        isCalcable = false;
    }
    else
    {
        sign = c;
        isVar = false;
        isSign = true;
    }

    isTerm = false;
    isZahl = false;
}
```

Dieser Konstruktor setzt die Attribute so, dass das Objekt als Variable gehandhabt werden kann.

Methoden

```
public boolean isTerm()
{
    return isTerm;
}
```

Gibt an, ob es sich um einen Term handelt.

```
public boolean isZahl()  
{  
    return isZahl;  
}
```

Gibt an, ob es eine Zahle gespeichert hat.

```
public boolean isSign()  
{  
    return isSign;  
}
```

Gibt an, ob es ein Rechenzeichen ist.

```
public boolean isVar()  
{  
    return isVar;  
}
```

Gibt an, ob es eine Variable darstellt.

```
public boolean isCalcable()  
{  
    return isCalcable;  
}
```

Gibt an, ob das Element berechnet werden kann.

```
public String getString()  
{  
    if(isTerm)  
        return term;  
    else if(isZahl)  
        return String.valueOf(zahl);  
    else if(isSign)  
        return String.valueOf(sign);  
    else if(isVar)  
        return String.valueOf(variable);  
  
    return null;  
}
```

Gibt den gespeicherten Inhalt unabhängig vom Typ als String zurück.

```
public void setCalcable(boolean newCalcable)  
{  
    isCalcable = newCalcable;  
}
```

Mit dem übergebenen Boolean kann festgelegt werden, ob das Element berechnet werden darf oder nicht.

Reflexion

Bei dieser Projektarbeit wurden die Ziele, welche zu Beginn festgelegt wurden ohne größere Probleme erreicht. Sicher gab es hin und wieder kleine Unregelmäßigkeiten, die jedoch schnell behoben werden konnten. Da die Arbeit jedoch nicht nur aus dem Programmieren an sich besteht, sondern auch aus dem Dokumentieren des Projektverlaufs, hatte ich dort größere Probleme als beim Umsetzen der programmtechnischen Ziele. Wie schon bei diversen anderen Projekten habe ich auch diesmal wieder bemerkt, dass man früher anfangen sollte Alles schriftlich festzuhalten. Ein weiteres Problem das sich mir stellte war, dass ich nicht wusste wie ich ein Sequenzdiagramm zu solch einem umfangreichen Projekt erstellen sollte. Das lag meines Erachtens daran, dass der Benutzer ein zu großer Faktor ist, sodass sich keinen geregelten Ablauf ergibt, den welchen man als Ganzes darstellen könnte. Abschließend kann ich sagen, dass ich mir vorgenommen habe für die Zukunft die in Eclipse integrierten Dokumentationstools zu nutzen, um schon während der Entwicklungsphase die Dokumentation immer auf dem neuesten Stand zu halten.