Wortdatei mit Gui

Dieses einfache Beispiel mit einer Gui soll einen Benutzerzugriff auf eine Datei mit Worten liefern.

strings

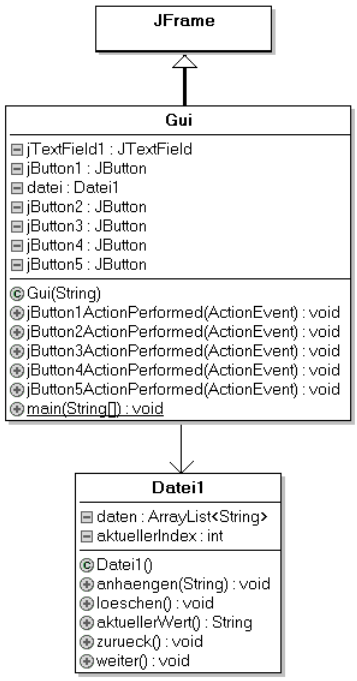
Worte sind allgemein englisch strings, also Zeichenketten, in Java sind das Objekte, also Instanzen der Klasse **`String`**. Da dieser Datentyp in einer Klasse realisiert ist, können wir davon ausgehen, dass er eine große Zahl von Methoden schon bereitstellt, mit denen typische Aufgaben realisiert sind, die ein Benutzer von Strings üblicherweise durchführt. Ein Blick in die javadoc wäre also sinnvoll. Eine Anmerkung ist aber ganz wichtig: Ein erstelltes Stringobjekt ist nicht veränderbar. Die Inhalte kann man also nicht bearbeiten, will man das tun, muss man jeweils ein neues Stringobjekt erzeugen. Bei sehr vielen Bearbeitungen an einem Stringobjekt sollte man sich vorher daher ein anderes Hilfsmittel besorgen, nämlich einen **`StringBuffer`**, zu dem es in der javadoc heißt:

> A thread[1]-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

Von einem solchen StringBuffer kann man am Ende der Bearbeitung einen dann statischen string erzeugen.

gui

Hier in diesem Text soll es aber um die Gui gehen. Strings sind nur Hilfsmittel, um die eingegebenen Worte zu speichern, wieder abzurufen und sie in der entsprechenden Komponente der Oberfläche darzustellen. Bei dieser sehr einfachen Anwendung werden wir das MVC (model view controller) – Konzept sicher nicht streng einhalten und nur die Modellkomponente ausgliedern. Unsere Datei wird also eine eigene Klasse haben, so dass wir ein UML – Diagramm der rechts dargestellten Art zu unserer Anwendung zeichnen können[2].
Das Schreiben sowohl der Gui-Klasse mit Hilfe der Funktionen des Javaeditors als auch das Schreiben der Datei-Klasse sollte keine großen Schwierigkeiten bereiten.



Versuchen Sie einmal, die Liste zu sortieren durch die sort – Methode, die unsere ArrayList<String> bereitstellt.

> Interface Comparable<T>
> Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`).

1 threads – deutsch: nebenläufige Prozesse – erlauben Anwendungen, mehrere Aufgaben gleichzeitig auszuführen. Eine wirkliche Gleichzeitigkeit ist natürlich nicht möglich, Java weist den verschiedenen Prozessen nach einander jeweils Rechenzeit zu und sorgt dafür, dass nach einer gewissen Zeit andere threads diese übernehmen. Probleme kann es natürlich bei Zugriffen auf gleiche Daten geben und thread-Sicherheit und Synchronisation sind in dem Zusammenhang wichtige Fragestellungen.
Wir arbeiten nicht mit threads.

2 Anders als BlueJ stellt der Javaeditor auch Systemklassen dar, von denen geerbt wird.

Sichern in eine Datei

Der Dateityp ArrayList hält die Daten nur temporär, nur solange das Programm läuft. Wollen wir die Daten dauerhaft sichern, beispielsweise auf der Festplatte, dann brauchen wir einen anderen Dateityp. Auch dafür stellt Java natürlich Werkzeuge bereit, sie sind aber nicht so einfach selbst erklärend wie bei der ArrayList. Mit dem Javaeditor können wir per Mausklick allerdings die „FileChooser“ einfügen und zwar sowohl für den Öffnen-Dialog als auch für den Speichern-Dialog. Klicken wir beide an, ohne die Vorgaben im Dialogfenster zu verändern, dann finden wir zwei Attribute

```
  private JFileChooser jfco = new JFileChooser();
  private JFileChooser jfcs = new JfileChooser();
```

und zwei fast gleiche Methoden für open (save entsprechend):

```
  public String jfcoOpenFilename() {
    jfco.setDialogTitle("Öffne Datei");
    if (jfco.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
      return jfco.getSelectedFile().getPath();
    } else {
      return null;
    }
  }
```

Damit ist es aber noch nicht getan und das ist auch sinnvoll, denn dies ist der Teil, welcher der View-Komponente zugeordnet ist. Die tatsächlichen Schreib- und Lesebefehle sollten wieder in der Klasse Datei zu finden sein. Hier wird es nun etwas komplizierter:

```
  /**
   * Der in der ArrayList vorhandene Text wird in die Datei geschrieben.
   * ACHTUNG: bestehende Dateien werden überschrieben !!!
   */
  public void dateiSchreiben(){
    try {
        PrintWriter ausgabe
        = new PrintWriter(new BufferedWriter(new FileWriter(dateiName)));
        for (int i=0; i<text.size();i++) {
            ausgabe.write(text.get(i));
            if (i<text.size()-1) ausgabe.write("\n");
        }
        ausgabe.close();
    } catch (IOException e){
        System.out.println("Fehler beim Erstellen der Datei !");
    }
  }
```

Da beim schreibenden und lesenden Zugriff immer mit Fehlern gerechnet werden muss, sollten wir diese mit einem try – catch – Block abfangen. Ob das Abfangen, wie oben angegeben, allein in einer Konsolenmeldung besteht oder besser ein Dialogfenster (showDialog...) bemüht wird, sei dahin gestellt.

Klasse Gui

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
  * @version 1.0 vom 07.01.2007
  * @author claus albowski
  */

public class Gui extends JFrame {
  // Anfang Variablen
  private JTextField jTextField1 = new JTextField();
  private JButton jButton1 = new JButton();
  private Datei1 datei;
  private JButton jButton2 = new JButton();
  private JButton jButton3 = new JButton();
  private JButton jButton4 = new JButton();
  private JButton jButton5 = new JButton();
  private JButton jButton6 = new JButton();
  // Ende Variablen

  public Gui(String title) {
    // Frame-Initialisierung
    super(title);
    datei=new Datei1();
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt) { System.exit(0); }
    });
    int frameWidth = 399;
    int frameHeight = 191;
    setSize(frameWidth, frameHeight);
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    int x = (d.width - getSize().width) / 2;
    int y = (d.height - getSize().height) / 2 ;
    setLocation(x, y);
    Container cp = getContentPane();
    cp.setLayout(null);
    // Anfang Komponenten

    jTextField1.setBounds(8, 64, 121, 24);
    jTextField1.setText("");
    cp.add(jTextField1);
    jButton1.setBounds(272, 64, 75, 25);
    jButton1.setText("Ende");
    cp.add(jButton1);
    jButton1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        jButton1ActionPerformed(evt);
      }
    });

    jButton2.setBounds(136, 48, 123, 25);
    jButton2.setText("übernehmen");
    cp.add(jButton2);
    jButton2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        jButton2ActionPerformed(evt);
      }
    });

    jButton3.setBounds(136, 80, 123, 25);
```

```
  jButton3.setText("entfernen");
  cp.add(jButton3);
  jButton3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      jButton3ActionPerformed(evt);
    }
  });

  jButton4.setBounds(24, 32, 75, 25);
  jButton4.setText("zurück");
  cp.add(jButton4);
  jButton4.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      jButton4ActionPerformed(evt);
    }
  });

  jButton5.setBounds(24, 96, 75, 25);
  jButton5.setText("weiter");
  cp.add(jButton5);
  jButton5.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      jButton5ActionPerformed(evt);
    }
  });

  jButton6.setBounds(24, 126, 75, 25);
  jButton6.setText("sortiere");
  cp.add(jButton6);
  jButton6.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      jButton6ActionPerformed(evt);
    }
  });

  // Ende Komponenten

  setResizable(false);
  setVisible(true);
}

// Anfang Ereignisprozeduren
public void jButton1ActionPerformed(ActionEvent evt) {
    System.exit(0);
}

public void jButton2ActionPerformed(ActionEvent evt) {
   if (!jTextField1.getText().equals(""))
      datei.anhaengen(jTextField1.getText());
}

public void jButton3ActionPerformed(ActionEvent evt) {
   datei.loeschen();
   jTextField1.setText(datei.aktuellerWert());
}

public void jButton4ActionPerformed(ActionEvent evt) {
   datei.zurueck();
   jTextField1.setText(datei.aktuellerWert());
}

public void jButton5ActionPerformed(ActionEvent evt) {
   datei.weiter();
```

```
      jTextField1.setText(datei.aktuellerWert());
  }

  public void jButton6ActionPerformed(ActionEvent evt) {
     datei.sortiere();
     jTextField1.setText(datei.aktuellerWert());
  }

  // Ende Ereignisprozeduren

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

Klasse Datei1

```
import java.util.ArrayList;
import java.util.Collections;

public class Datei1 {
  private ArrayList<String> daten;
  private int aktuellerIndex;

  public Datei1(){
    super();
    daten = new ArrayList<String>();
    aktuellerIndex=-1;
  }

  public void anhaengen(String dat){
    daten.add(dat);
    aktuellerIndex=daten.size()-1;
  }

  public void loeschen(){
    if ((aktuellerIndex>-1)&&(aktuellerIndex<daten.size()))
    daten.remove(aktuellerIndex);
  }

  public String aktuellerWert(){
    if (aktuellerIndex==-1) return new String("");
    else if (aktuellerIndex<daten.size()) return daten.get(aktuellerIndex);
    else return new String("");
  }

  public void zurueck(){
    if (aktuellerIndex>=0) aktuellerIndex--;
  }

  public void weiter(){
    if (aktuellerIndex<daten.size()) aktuellerIndex++;
  }

  public void sortiere(){
    Collections.sort(daten);
  }

}
```