Java Zusammenfassung von Simon Flüeli

GUI
```
import java.awt.*
```
AWT – Abstract Window Toolkit

Combobox: Choice (ItemListener)

Static
Static-Variablen: Klassenvariablen
Ein Datenelement soll nur an einem Ort abgespeichert werden, egal wie viele Objekte erzeugt worden sind – selbst wenn noch gar keine Objekte bestehen.
```
class StaticTest
{ public static int number=100; }

int x = StaticTest.number;
```

Zugriffsspezifikatoren
Keiner: Zugriff für Klassen / Objekte aus dem gleichen Package
Public: Zugriff ohne Einschränkung
Protected: wie Package-Gültigkeit & Zugriff von abgeleiteten Klassen (Vererbung)
Private: Kein Zugriff von Aussen (nur eigene Klasse)

Strings
Strings können nicht mit „==“ auf gleichen Inhalt geprüft werden. -> string1.equals(string2);
Wichtige methoden:
```
equalsIgnoreCase(String str)

int length()

char charAt(int index)
String sExpression = "((x+1) * y)";
char c = sExpression.charAt(2);
```
→ *c enthält "x"*

```
int indexOf(String str, int index)
```
gibt die Position eines Sub-Strings innerhalb eines Strings zurück (die Suche beginnt bei index)

```
int pos = "Winterthur".indexOf("thur",
index);
```
pos = 6
pos = -1 wenn Sub-String nicht gefunden wurde

```
boolean startsWith(String str)
boolean endsWith(String str)

String toLowerCase() / toUpperCase

String trim()

String substring(int start, int end + 1)
```
erzeugt einen neuen String, indem es einen Teil des Strings kopiert:
```
String sOrt1 = "Winterthur
Versicherung";
String sOrt2 =
sOrt1.substring(0,10);
```
- sOrt2 -> "Winterthur"

Events
```
import java.awt.event.*
```
Event-Sources: Komponenten (Buttons, Scrollbars)
Event-Listener: Klassen / Methoden zum abfangen / bearbeiten von Events
… implements ActionListener
txtFeld.addActionListener(this)
public void actionPerformed(ActionEvent e)
Scrollbar: AdjustmentListener / adjustmentValueChanged(AdjustmentEvent e)

Enumerations
- Vom Programmierer definierte Datentypen für Aufzählungen (z.B. für Wochentage)
- Vorteil: Es können nur gültige Wochentage verwendet werden
Public enum Geschlecht { MAENNLICH, WEIBLICH }
Wochentag tag = Wochentag.MONTAG

StringTokenizer
```
import java.util.*
```
Wichtige Methoden:
String nextToken()
Gibt einen Substring zurück, der durch ein Spezialzeichen vom nächsten Substring getrennt ist.
```
String namen = {"Albrecht;Brunner;Fischer;Rohrer"};
StringTokenizer namenList;
namenList = new StringTokenizer(namen,";");
while(namenList.hasMoreTokens())
{
   namenRecords[index] = namenList.nextToken();
   index++;
}
```

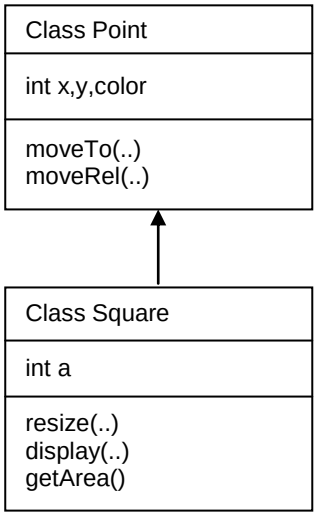
Split
Alternative zu StringTokenizer
```
String st = "aaa;bbb ccc!ddd";
String[] stRecords;
stRecords = st.split("[; !]");

for(int i=0; i < stRecords.length; i++)
{
   System.out.println(stRecords[i]);
}
```

Arrays
Eindimensional: String[] array = new String[10];
Zweidimensional: String[][] array = new String [10][10];
Grösse des Arrays: `array.length;`

Garbage Collector
Mechanismus, de rim Hintergrund läuft und die Objekte markiert und entfernt, auf die nicht mehr verwiesen wird.

Vererbung
Durch Vererbung können Klassen definiert werden, die auf einer anderen Klasse basieren. Neue Klasse: Erweiterung und Spezialisierung der Basisklasse.
Neue Klasse kann: Elemente / Methoden hinzufügen; bestehende Methoden überschreiben.



In Klasse Square ist nun alles verfügbar, was auch in der Klasse Point verfügbar ist, also auch x, y, color, moveTo und moveRel.

```
class Point
{
   protected int x, y;
   protected Color color;

   public Point() {
        x = 0;
        y = 0;
        color = Color.black;
   }

   public Point(int x, int y, Color
color){
        this.x = x;
   this.y = x;
        this.color = color;
   }

   public void moveTo(int newX, int newY)
   {
        x = newX;
        y = newY;
   }

   public void moveRel(int dx, int dy)
   {
        x = x + dx;
        y = y + dy;
   }
}
```

```
class Square extends Point
{
   protected int a;

   public Square(int x, int y, int a, Color
   color)
   {
        super(x, y, color); // Konstr. des
        Vorfahren
        this.a = a;
   }

   public void resize(int a)
   {
        this.a = a;
   }

   public void display(Graphics g)
   {
        g.setColor(g, color);
        g.drawRect(x, y, a, a);
   }

   public int getArea()
   {
        return (a * a);
   }
}
```

Überschreiben von Methoden
Oft notwendig, dass Unterklasse eine geerbte Methode anders implementieren muss, als es beim Vorfahren der Fall ist.
→ Methode überschreiben (gleiche Parameterliste und Rückgabewert). Die Anpassung abgeleiteter Klassen durch Überschreibung geerbter Methoden nennt man **Polymorphismus**

```
class Rectangle extends Square {
   protected int b;

   public Rectangle(int x, int y, int a,
   int b, Color color) {
        super(x, y, a, color);
        this.b = b;
   }

   public void resize(int a, int b){
   // überschreibt resize von Square? ->
   Nein, andere Parameterliste
        super.resize(a);
        this.b = b;
   }

   public void display(Graphics g)
   { // überschreibt display von Square
        g.setColor(g, color);
        g.drawRect(x, y, a, b);
   }

   public int getArea()
   { // überschreibt getArea von Square
        return (a * b);
   }
}
```

Beispiel:

```
class Mitarbeiter {
   protected int persNr;
   protected String name;
   protected float gehalt;

   public Mitarbeiter(int persNr,
   String name, float gehalt) {
       this.persNr = persNr;
       this.name = name;
       this.gehalt = gehalt;
   }

   public void gehalt_erhoehen(float
   erhoehung) {
       gehalt += erhoehung;
   }
}

class Chef extends Mitarbeiter {
   // keine Erweiterungen
   public Chef(int persNr, String
   name, float gehalt) {
       super(persNr, name, gehalt);
   }

   public void gehalt_erhoehen(float
   erhoehung) {
       gehalt += 1.1f * erhoehung;
   }
}
```

Anwendung von 'Mitarbeiter / Chef'

```
Mitarbeiter mueller = new
Mitarbeiter(4711, "Müller", 8000f);
Mitarbeiter meier = new Chef(4712,
"Meier", 9000f);

if(meier instanceof Chef)
{
   System.out.println(„Meier ist Chef");
}
```

Abstrakte Klassen

Datenfelder und Methoden, die bei mehreren Klassen gemeinsam vorhanden sind, werden in einer Basisklasse zusammengefasst.

Wird in einer Basisklasse nur die Schnittstelle von Methoden (Methodenkopf) festgelegt und die Implementierung erst in einer abgeleiteten Klasse vorgenommen, so liegt eine abstrakte Basisklasse vor.

Abstrakte Klassen können NICHT instanziert werden.

Methoden, für die nur die Schnittstelle festgelegt wird (also keine Implementierung) müssen mit dem Schlüsselwort abstract deklariert werden.

Ist in einer Klasse auch nur eine abstrakte Methode, so ist die Klasse zwangsläufig abstrakt und muss als solche deklariert werden.

Eine Klasse, in der alle Methoden implementiert sind, kann mit dem Schlüsselwort abstract zu einer abstrakten Klasse gemacht werden. Damit kann die Instanzierung einer solchen Klasse verhindert werden. Eine solche Klasse ist zwar vollständig, sie stellt aber eine Abstraktion dar, von der es in der Realität keine Objekte gibt.
Eine solche abstrakte Klasse dient allein Ziel der Generalisierung in der Klassenhierarchie.

Implementiert eine abgeleitete Klasse nicht alle abstrakten Methoden, ist sie wiederum wie die Basisklasse abstrakt.

Die abstrakten Methoden in der Basisklasse müssen von den abgeleiteten Klassen implementiert werden → **Zwang zur Konformität und Vollständigkeit**

```
abstract class Figure {
   protected int x, y;

   public Figure(int newX, int newY){
       x = newX;
       y = newY;
   }

   public void moveTo(int newX, int newY) {
       x = newX;
       y = newY;
   }

   public abstract void display(Graphics g);
   public abstract void getArea();
}

class Square extends Figure {
   protected int a;

   public Square(int x, int y, int a) {
       super(x, y);
       this.a = a;
   }

   public void display(Graphics g) {
       g.drawRect(x, y, a, a);
   }

   public int getArea() {
       return a * a;
   }
}
```

Interfaces
Interfaces enthalten Methodenköpfe.
Klassen, die sich auf ein Interface beziehen (..implements Interface_XY) müssen die im Interface aufgeführten Methoden implementieren:

Beispiel: Source-Code von ActionListener

```
public interface ActionListener extends
EventListener {
    //Invoked when an action occurs.
    public void actionPerformed(
       ActionEvent e);
}
```

Interfaces können angewendet werden, um Konformität zu erzwingen; zudem liefern sie eine Übersicht über die Methoden, die eine gewisse Kategorie von Klassen haben muss.
Beispiel: Verschieden Motoren müssen folgende Methoden implementieren:
- setOn
- setOff
- setForward
- setReverse

Das Interface Motor sieht nun so aus:
```
public interface Motor
{
   public abstract void setOn();
   public abstract void setOff();
   public abstract void setFroward();
   public abstract void setReverse();
}
```

De Klassen, welche Motor implementieren müssen ALLE Methoden implementieren, die in Motor aufgeführt sind.

Interfaces können ferner dazu benützt werden, gemeinsame Konstanten
in mehrere Klassen zu importieren:

```
interface CommonConstants {
   final boolean ON = true;
   final boolean OFF = false;
   final int OK = 0;
   final int ERROR1 = -1;
}
```

In diesem Fall bedeutet *implements CommonConstants* nicht, dass etwas implementiert werden muss.

Interface Motor
{
    …
}

Class ElektroMotor implements Motor
{
    …
}

Class DieselMotor implements Motor
{
    …
}

Anwendung:
```
Motor elektroMotor = new ElektroMotor(1295, 3000);
Motor dieselMotor = new DieselMotor(1240, 2000);
```

Model-View-Controller
• Das Modell enthält den funktionalen Kern der Anwendung; es modelliert die eigentliche Anwendung (Verarbeitung der Daten, Steuerung eines Ablaufs).
Ferner kapselt es die Daten und bietet Operationen an, welche die für die Anwendung spezifischen Verarbeitungen vornehmen.
Es ist normalerweise nicht sichtbar für den Benützer

Das Modell stellt Operationen zum Zugriff auf seine Daten zur Verfügung, welche von der View (get...) und vom Controller (set...) verwendet werden.

• Die View (Ansicht / Ausgaben) präsentiert dem Anwender die vom Modell verarbeiteten Daten.

• Der Controller (mehrdeutig / Varianten) enthält Steuerungskomponenten für Bedienereingaben (Buttons,Textfelder etc.).

Views und Controller umfassen die Benutzerschnittstelle, diese ist für die Kommunikation zwischen dem Benützer und dem Modell verantwortlich.

Beispiel Auto:
Control: Gaspedal, Bremspedal, Kupplungspedal, Ganghebel, Steuerrad, Tachometer, Tourenzähler, Benzinanzeige, Lichtschalter etc.
Modell: Motor, Kupplung, Bremsanlage, Lenkung, Lichtanlage
View: Instrumente, Warnlampen

Vorteile:
Einfache und sichere Bedienung und Überwachung durch Benützer.
Verstecken der Komplexität vor dem Benützer.
Getrennte und damit einfachere Entwicklung und Wartung der Teilsysteme.

Ziel und Vorteile des MVC-Modells
Ziel beim Design:
- möglichst starke Entkopplung von Modell und Benützerschnittstelle

Vorteile:
- Unabhängige Entwicklung/Test von Modell und Benützerschnittstelle möglich
- Es können gleichzeitig mehrere Views und Controllers für dasselbe Modell aktiv sein.
- Änderungen an Benützerinterface oder Modell sind leichter möglich.

Canvas
Separate Zeichenfläche

```
import java.awt.*;
import java.awt.applet.*;

public class CanvasDemo extends Applet {
   private MyCanvas canvas = new
       MyCanvas();

   public void init() {
       canvas.setBackground(Color.gray);
       canvas.setForeground(Color.white);
       canvas.setSize(200, 100);
       add(canvas);
   }
}

class MyCanvas extends Canvas {

   public void paint(Graphics g) {
       g.drawString("Ich bin in einer
       Canvas");
   }
}
```

Erklärung:
- MyCanvas: Subklasse von Canvas
- Um zu zeichnen: paint(Graphics g) von Canvas überschreiben

List

```
List liste = new List();
oder new List(5, true);

liste.add("Liste Zeile 1");
oder liste.addItem("...");
```

ItemEvent (ItemListener): bei jeder neuen Auswahl einer Variante mit Einfach-Klick
ActionEvent (ActionListener): Bei Doppelklick eines Items

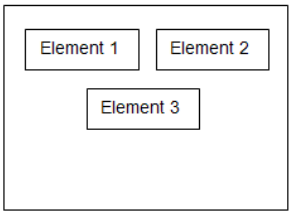
Benötigte Methoden:
itemStateChanged(ItemEvent ie)
itemActionPerformed(ActionEvent e)

Funktionen des List-Objekts
Int getSelectedIndex() → beginnend bei 0
Int[] getSelectedIndexes() → beginnend bei 0
String getSelectedItem()
String[] getSelectedItems()
Add(String str, int index)
Add(String str)
Remove(int index)
Select(int index) } z.B. für Default
Deselect(int index) }

Panel
Unsichtbare Komponente, welche andere enthalten kann.
→ Gruppierung von Komponenten

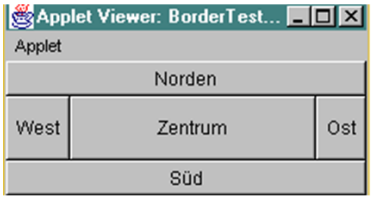
Vorgehen um Komponenten in Panel zu gruppieren:
- Panel erzeugen
- Komponenten zu Panel hinzufügen
- Panel zum Applet / anderem Panel hinzufügen

Layout Manager
- FlowLayout



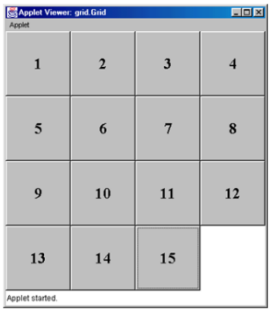
Standardlayout für Applets/Panels
FlowLayout layout = new FlowLayout() → zentriert
Optional new FlowLayout(FlowLayout.LEFT, 10, 20) → 10px horizontal, 20px vertikal

- BorderLayout



Standardlayout für selbstständige Applikationen
BorderLayout layout = new BorderLayout()
Oder new BorderLayout(10, 20) → 10px horizontal, 20px vertical
- Pro Feld je eine Komponente (aber auch Panel möglich)
- Platz Zuweisung:
Button nord = new Button(„Norden”);
add(nord, BorderLayout.NORTH);

- GridLayout



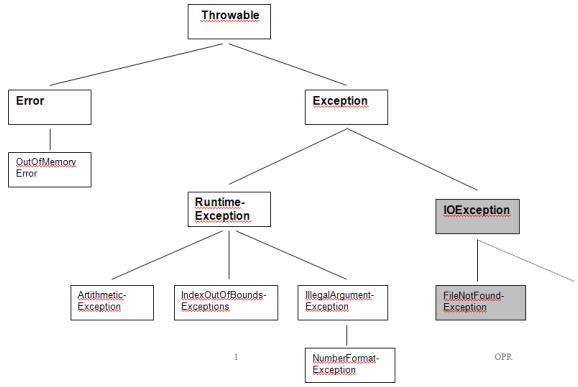
- Gridlayout ordnet die Komponenten den Feldern eines Gitters zu
GridLayout layout = new GridLayout(); → 1 Kolonne
= new GridLayout(rows, cols);
= new GridLayout(rows, cols, hSp, vSp);

- GridBagLayout

Exceptions
- Fehlerhafte Eingaben müssen klar definierte Reaktion auslösen.
- Unkorrekte Werte von Argumenten, die an Methoden übergeben werden, müssen ebenfalls eine klar definierte Reaktion auslösen.

```
try {
a =
   Integer.parseInt(txtValue.getText());
   repaint();
}
catch(NumberFormatException x) {
   showStatus("Enter an integer value
   !");
   return;
}

class Integer {
   public static int parseInt(String s)
        throws NumberFormatException
   {
        throw new NumberFormatException();
   }
}
```



Methoden von throwable
- getMessage() → liefert Fehlermeldungstext
- toString() gibt Typ der Exception und Text aus
- printStackTrace() gibt zusätzlich den Aufrufpfad der vorgängig aufgerufenen Methoden aus

Fehler erzeugen / weitergeben
• Java bietet die Möglichkeit, die Exceptions nicht selber zu behandeln sondern an den Aufrufer weiterzuleiten. Diese Kette kann soweit fortgesetzt werden, bis schlussendlich der Interpreter die Exception mit einer allgemeinen Fehlermeldung quittiert.
• Damit aber die entsprechende Methode eine Exception an den Aufrufer weiterleiten kann, muss im entsprechenden Methodenkopf zusammen mit dem Schlüsselwort throws alle möglichen Exceptions, getrennt durch Kommas aufgelistet werden.
• Werden nun die try- und catch-Blöcke weggelassen, so wird eine eventuell auftretende Exception einfach an den Aufrufer weitergeleitet.

```
public void paint(Graphics g) {
   try {
        c = divide(a,b);
        txtResult.setText(Integer.toString(c));
   }
   catch(ArithmeticException e) {
        showStatus("Fehler: +e.getMessage());
   }
}

private int divide(int x, int y) throws
   ArithmeticException
{
   if(y != 0)
        return x/y;
   else
        throw new ArithmeticException("Division
        by zero not permitted!");
}
```

Eigene Exceptions

```
class EingabeException extends Exception
{
   public EingabeException (String msg) {
        super(msg);
   }
}
```

Applets und Applikationen

Applets:
Java-Programme, die in HTML-Seiten eingebunden werden.
Interpreter: Im Browser / Appletviewer enthalten
→ Kann nicht allein ausgeführt werden
→ Müssen nicht auf lokaler Maschine installiert sein
→ Begrenzer Schreibzugriff auf Dateisystem

Applikationen:
Java-Applikationen sind eigenständige Programme, die vom Betriebssystem gestartet werden.
Brauchen Java-Interpreter welcher Java-Byte-Code zur Laufzeit in Maschinencode umwandelt.
→ Braucht eine „main“ Methode

```
public class Editor extends Frame
implements ActionListener,
WindowListener {
   private TextArea txtArea;
   private Button cmdLaden;
   private static String file;

 public static void main(String args[])
 {
   if(args.length > 0)
        file = args[0];
   else
        file = "test.txt";

   Editor editor = new Editor();
   editor.setSize(400,400);
   editor.setTitle("Editor");
   editor.makeGui();
   editor.setVisible(true);
 }
```

```
private void makeGui()
{
    Panel header = new Panel();
    cmdLaden = new Button("Laden");
    header.add(cmdLaden);
    cmdLaden.addActionListener(this);
}

// Folgende Event-Handler müssen
// implementiert sein (weil wir "Frame"
// geerbt haben)

public void windowClosing(WindowEvent e)
{
   if(printWriter != null)
       printWriter.close();
   System.exit(0);
}
public void windowOpened(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public void windowClosed(WindowEvent e){}
public void windowDeiconified(WindowEvent
e){}
public void windowActivated(WindowEvent e){}
public void windowDeactivated(WindowEvent
e){}
```

Main Methode muss static sein, weil sie aufgerufen wird, bevor ein Objekt erzeugt wird.

Dateien

Das Grundkonzept der Ein- und Ausgabe bildet in Java der Stream: eine geordnete Folge von Bytes.
(Sequentieller Zugriff auf eine Datenquelle im Gegensatz zu willkürlichem/random Zugriff)

Der Stream ist ein Art Kommunikationskanal für den Datentransfer.

Je nachdem, welche Arten von Sender/Empfänger man ansprechen will, gibt es verschiedene Arten von Stream-Klassen: Byte-Streams, Character-Streams (für Text).

Für Text-Dateien gibt es folgende Klassen und Methoden:

Lesen von einer Datei: Klasse BufferedReader, Methode readLine

Schreiben auf eine Datei: Klasse PrintWriter, Methode print/println

Ferner brauchen wir Instanzen der Klassen FileReader/FileWriter, welche die Dateien repräsentieren.

Dazu folgenden Beispiele (ohne ExceptionHandling wegen Übersichtlichkeit):

Lesen einer Datei

```
// Klasse, die den Zugang zu einer
// Eingabedatei erschliesst
// (zeichenorientiert)
private FileReader inputFile;

// Klasse, die Methoden zum Lesen
// enthält
private BufferedReader bufferedReader;

// "open file for read"
inputFile = new FileReader("a:\Testdaten.txt");
bufferedReader = new BufferedReader(inputFile);

String zeile;

while ((zeile = bufferedReader.readLine()) != null)
{
   txtArea.append(zeile + "\n");
}
bufferedReader.close();
```

Ausgabe an eine Datei

```
// Klasse, die den Zugang zu einer
// Ausgabedatei erschliesst
// (zeichenorientiert)
private FileWriter outputFile ;

// Klasse, die Methoden zur Textausgabe enthält
private PrintWriter printWriter;

// "open file for write"
outputFile = new FileWriter("a:\Test.txt");
printWriter = new PrintWriter(outputFile,true);
//true: autoflush
printWriter.print(textArea.getText());
printWriter.close();
```

Mit Exceptionhandling:

```
try
{
   bufferedReader = new BufferedReader(new
       FileReader(dateiName));
   String zeile;

   while ((zeile = bufferedReader.readLine()) != null)
   {
       txtArea.append(zeile+"\n");
   }
   bufferedReader.close();
}
catch(IOException e)
{
   System.err.println("Dateifehler: " + e.toString());
   System.exit(1);
}
```

Anmerkungen
Printwriter:
- Daten werden nicht unmittelbar in Datei geschrieben,
 sondern im Buffer gespeichert.
Geschrieben werden sie, wenn der Buffer voll ist oder
mit Kommandos „close“ und „flush“

BufferedReader:
- Daten werden in grösseren Portionen eingelesen, obwohl
Anwenderprogramm nur Zeile um Zeile einliest (readline).

File:
- Klasse mit nützlichen Datei-Funktionen

```
File file = new File(fileName);
file.exists();
file.isDirectory()
file.getAbsolutePath();
```

→ File als Ganzes, nicht Inhalt