Grundlagen der Informatik für Ingenieure I

8. AWT - Abstract Window Toolkit Teil 1

8 AWT-Abstract Windowing Toolkit

◆ ***AWT*** ist ein graphisches Interface sowohl für

- Java Applets als auch für
- Java-Applikationen

◆ Ein wesentlich umfangreicheres *package* zum Design von graphischen Oberflächen ist ***swing***.

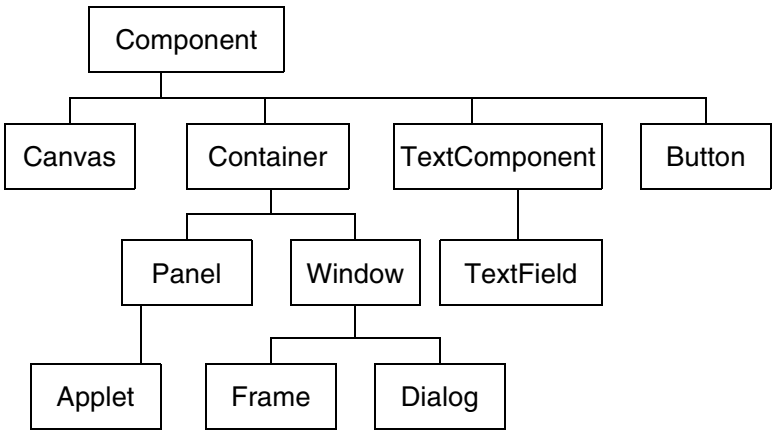
- Die grundsätzlichen Mechanismen sind jedoch ähnlich.
- Im Rahmen dieser einführenden Vorlesung wird der Verwendung von ***AWT*** der Vorzug gegeben.

8 AWT-Abstract Windowing Toolkit

- ***AWT*** enthält ***GUI*** (*Graphical User Interface*)- **Komponenten** wie
  - *Labels*
  - *Buttons*
  - *Check Boxes*
  - *Textfields*
  - *Scroll Bars*
  - *Windows*
  - *Menus*
  - *Container*
  - *Eventhandling*
  - *Mechanismen zur plattformunabhängigen GUI-Gestaltung*

8 AWT-Abstract Windowing Toolkit

- ***AWT*** unterstützt beim Design eine hierachische Struktur.
- ***Container*** können wiederum Container enthalten.
- Jeder ***Container*** kann beliebige Basiskomponenten enthalten.
- ***Component class*** ist die Superklasse aller Komponenten.



8 AWT-Abstract Windowing Toolkit

◆ Die allgemeine Form des ***Containers*** ist das ***Panel***:

- ein ***Container***, der ein Fenster darstellt.
- ***Applet*** z. B. ist eine Form des ***Panels***.
- Konkret ist ***Applet*** *subclass* der ***Panel-class.***

◆ ***Canvases*** sind *"Draw"*-Oberflächen für Strichgraphik oder für Images.

◆ Window-Konstruktions-Komponenten, wie

– *windows,*

– *frames* oder

– *menu bars*

werden nur bei Java-Applikation gebraucht; bei ***Java-Applets*** ist dies Sache des ***Browsers*** (Siehe auch *AWT*-Teil 2).

8 AWT-Abstract Windowing Toolkit

◆ Das Kreieren einer Komponente läuft immer nach dem gleichen Schema ab:

- Kreieren eines Komponenten-Objekts
- Hinzufügen dieser Komponente auf das Panel z. B.:

```
public void init(){
    Button b = new Button( "OK" );
    add(b);
// oder auch
    add( new Button( "OK" ) );
}
```

- Der Aufbau der graphischen Oberfäche wird in der Regel in der ***init-method()*** vorgenommen.
- Im Folgenden werden wir jeweils an Beispielen die Komponenten einführen und Layouttechniken diskutieren.

8.1 Basiskomponenten

- Basiskomponenten sind Komponenten, die selbst keine anderen Kompo*nenten mehr enthalten können.*
- *Wir behandeln hier*
  - *Labels*
  - *Buttons*
  - *Check Boxes*
  - *Radio Buttons*
  - *Choice Menus*
  - *Text Fields*

8.1 Basiskomponenten

- Die einzelnen Abschnitte sind wie folgt strukturiert:
  - Kurze Beschreibung der Eigenschaften
  - Konstruktoren
  - Methoden (eine Auswahl). Ausführlich in der Dokumentation unter:

    java.awt.komponentenname
  - ***Event Typ***. Anwendung im Kapitel 10 *Eventhandling.*
  - Jeweils ein kleines Beispiel; die verwendeten ***layout manager*** werden in Abschnitt 8.2 behandelt.

1 Labels

◆ ***Labels*** sind die einfachsten Komponenten
- Es handelt sich um Text-Strings,
- sie sind nicht editierbar

◆ **Konstruktoren:**

Constructors	Action
Label()	Constructs an empty label
Label(String)	Constructs a new label with the specified string of text, left justified.
Label(String, int)	Constructs a new label with the specified string of text. The second parameter specifies the alignment: Label.LEFT Label.CENTER Label.RIGHT

1 Labels

◆ ***Label Methods:***

Methods	Action
getText()	Returns a string containing this label's text
setText(String)	Changes the text of this label
getAlignment()	Returns an integer representing the alignment of this label: Label.LEFT Label.CENTER Label.RIGHT
setAlignment(int)	Changes the alignment of this label to the given integer: Label.LEFT Label.CENTER Label.RIGHT

◆ ***Eventhandling*:**
- keine *Events*

1 Labels

◆ Ein Beispiel:

```
/* Labels */

import java.awt.*;

public class LabelTest extends java.applet.Applet {

  public void init() {
    setFont( new Font( "Helvetica", Font.BOLD, 14 ) );
    setLayout( new GridLayout( 3,1 ) );
    add( new Label( "aligned left", Label.LEFT ) );
    add( new Label( "aligned center", Label.CENTER ) );
    add( new Label( "aligned right", Label.RIGHT ) );
  }
}
```


1 Labels

◆ Ergebnis mit Appletviewer:

2 Buttons

◆ ***Buttons*** sind Komponenten mit denen man durch Mausbetätigung Aktionen auslösen kann:

- Sie können einen Bezeichner (*label*) tragen.
- Ihre Darstellung ist häufig zustandsabhängig gestaltet:
  - out focus (Maus außerhalb)
  - in focus (Maus innerhalb)
  - clicked

◆ **Konstruktoren***:*

Constructors	Action
ButtonI()	Constructs a Button with no label
Button(String)	Constructs a Button with th specified label

2 Buttons

◆ ***Button Methods:***

Methods	Action
setLabel(String)	Sets the button's label to be the specified string.
getLabel()	Returns the buttons label, or null if the button has no label
paramString()	Returns the parameter string representing the state of this button. (usefull for debugging)

◆ ***Eventhandling:***

- ***ActionEvent****:* wird generiert, wenn der ***Button*** mit der ***Mouse*** betätigt wird.
- ***MouseEvent****:* Wenn der exakte Zustand der ***Mouse*** von Interesse ist.

2 Buttons

◆ Ein Beispiel:

```
/* create a few buttons */

import java.awt.*;

public class ButtonTest extends java.applet.Applet {

  public void init() {

    setLayout( new FlowLayout( FlowLayout.LEFT ) );

    add( new Button( "Rewind" ) );
    add( new Button( "Play" ) );
    add( new Button( "Fast Forward" ) );
    add( new Button( "Stop" ) );
  }
}
```


2 Buttons

• Ergebnis mit Appletviewer:

3 Check Boxes

◆ Mit ***Check Boxes*** ist es möglich, alternativ Kästchen *(Boxes)* zu betätigen und damit geeignet Aktionen auszulösen.

- ***Check Boxes*** können zwei Stati annehmen
  - selektiert/nicht selektiert
  - eine oder mehrere können selektiert sein (nonexclusive)

◆ **Konstruktoren:**

Constructors	Action
Checkbox()	Creates a check box with no label
Checkbox(String)	Creates a check box with the specified label
Checkbox(String, boolean)	Creates a check box with the specified label; true/false: initial status.

3 Check Boxes

◆ ***Check Box Methods:***

Methods	Action
getLabel()	Returns a string containing this check box's label
setLabel(String)	Changes the text of the check box's label
getState()	Returns true or false, based on whether the check box is selected
setState(boolean)	Changes the check box's state to selected ***(true)*** or unselected ***(false)***

◆ ***Eventhandling:***

- *ItemEvent*

3 Check Boxes

◆ Ein Beispiel:

```
/* check boxes */

import java.awt.*;

public class CheckboxTest extends java.applet.Applet {

  public void init() {
    setLayout( new FlowLayout( FlowLayout.LEFT ) );
    add( new Checkbox( "Shoes" ) );
    add( new Checkbox( "Socks" ) );
    add( new Checkbox( "Pants" ) );
    add( new Checkbox( "Underwear", true ) );
    add( new Checkbox( "Shirt" ) );
  }
}
```


3 Check Boxes

• Ergebnis mit Appletviewer:

4 CheckboxGroups; Radio Buttons

- ***Radio Buttons*** sind ***CheckBox***-Gruppen bei denen nur jeweils **eine** Alternative selektiert werden kann (exclusive).

- Zunächst ist es notwendig eine ***CheckboxGroup*** zu kreieren:

```
CheckboxGroup cbg = new CheckboxGroup();
```

dann werden die *Boxes* mit *add()* hinzugefügt:

```
add( new Checkbox ( "RED", false, cbg ) );
```

Durch den dritten Parameter wird die entsprechende Gruppe referenziert.

5 CheckboxGroups; Radio Buttons

- **Konstruktoren:**

Constructors	Action
CheckboxGroup()	Creates a new instance of CheckboxGroup

- Eine weiterer **Konstruktor der *Checkbox class*:**

Constructors	Action
Checkbox(String, boolean, CheckboxGroup)	Creates a check box with the specified label, in the specified check box group, and set to the specified state.

5 CheckboxGroups; Radio Buttons

◆ ***Check Box Methods:***

Methods	Action
getLabel()	Returns a string containing this check box's label
setLabel(String)	Changes the text of the check box's label
getState()	Returns true or false, based on whether the check box is selected
getCheckboxGroup()	Determines this check box's group.
setState(boolean)	Changes the check box's state to selected ***(true)*** or unselected ***(false)***

◆ ***Eventhandling:***

- *ItemEvent*

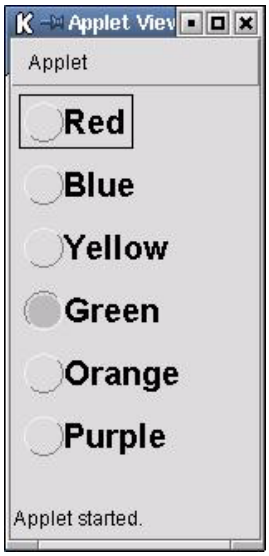
5 CheckboxGroups; Radio Buttons

◆ Beispiel:

```
/* radio buttons */

import java.awt.*;

public class CheckboxGroupTest extends java.applet.Applet {

  public void init() {
     setLayout( new FlowLayout( FlowLayout.LEFT ) );

     CheckboxGroup cbg = new CheckboxGroup();

     add( new Checkbox( "RED", false, cbg ) );
     add( new Checkbox( "BLUE", false, cbg ) );
     add( new Checkbox( "YELLOW", false, cbg ) );
     add( new Checkbox( "GREEN", true, cbg ) );
     add( new Checkbox( "ORANGE", false, cbg ) );
     add( new Checkbox( "PURPLE", false, cbg ) );
  }
}
```

5 CheckboxGroups; Radio Buttons

- Ergebnis mit Appletviewer:



5 Choice Menus

◆ ***Choice Menus*** sind ***Pop-Up***-Listen von denen man einen Wert auswählen kann.

◆ Zunächst ist es notwendig ein ***Choice Menu*** - Objekt zu kreiern:

```
Choice c = new Choice();
```

dann werden die ***Items*** mit ***add()*** hinzugefügt:

```
c.add( "Apples" );
```

◆ **Konstruktoren:**

Constructors	Action
Choice()	Creates a new choise menu

5 Choice Menus

◆ ***Choice Menu Methods:***

Methods	Action
add(String)	Adds an Item to this Choice menu
addItem(String)	Adds an Item to this Choice
getItem(int)	Returns the string item at the given position (items inside a choice begin at 0, just like arrays)
getItemCount()	Returns the number of items in the menu
getSelectedIndex()	Returns the index position of the item that's selected
getSelectedItem()	Returns the currently selected item as a string
select(int)	Selects the item at the given position
select(String)	Selects the item with the given string

◆ **Eventhandling:**

- ItemEvent

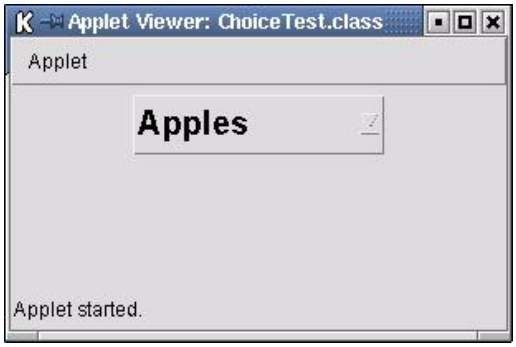
5 Choice Menus

◆ Beispiel:

```
/* choice menus */

import java.awt.*;

public class ChoiceTest extends java.applet.Applet {

  public void init() {
    setFont( new Font( "Helvetica", Font.BOLD, 20 ) );

    Choice c = new Choice();
    c.add( "Apples" );
    c.add( "Oranges" );
    c.add( "Strawberries" );
    c.add( "Blueberries" );
    c.add( "Bananas" );

    add(c);
  }
}
```

5 Choice Menus

- Ergebnis mit Appletviewer:



6 Text Fields

◆ ***Text Fields*** sind einzeilige Textfelder in denen man Text einbringen und ändern kann.

◆ **Konstruktoren:**

Constructors	Action
TextField()	Constructs a new text field
TextField(int)	Constructs a new empty text field with the specified number of columns (characters).
TextField(String)	Constructs a new text field initialized with the specified text
TextField(String, int)	Constructs a new text field initialized with the specified text and wide enough for the specified numbers of characters.

6 Text Fields

◆ ***Text Field Methods:***

Methods	Action
getText()	Returns the text that this text filed contains (as a string)
setText(String)	Puts the given text string into the field
getColumns()	Returns the width of this text filed
select(int, int)	Selects the text between the two integer positions (positions start from 0)
selectAll()	Selects all the text in the field
isEditable()	Returns ***true*** or ***false*** based on whether the text is editable
setEditable(boolean)	***true*** (the default) enables text to be edited; ***false*** freezes the text
getEchoChar()	Returns the character used for masking input

◆ **Eventhandling:**

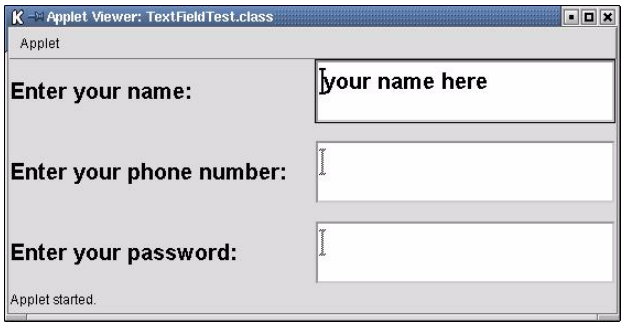
- ActionEvent

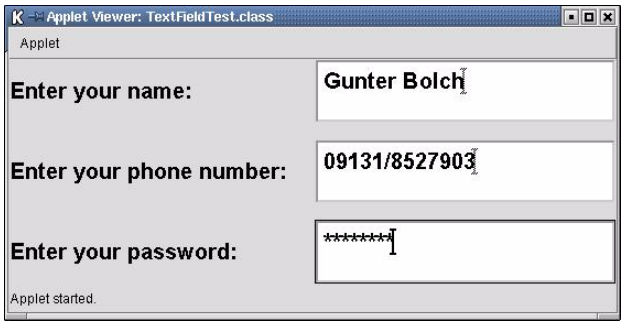
6 Text Fields

◆ Beispiel:

```
/* text fields */

import java.awt.*;

public class TextFieldTest extends java.applet.Applet {

  public void init() {

    setLayout( new GridLayout( 3,2,5,15 ) );

    add( new Label( "Enter your name:" ) );
    add( new TextField( "your name here",45 ) );
    add( new Label( "Enter your phone number:" ) );
    add( new TextField( 12 ) );
    add( new Label( "Enter your password:" ) );

    TextField t = new TextField( 20 );

    t.setEchoChar( '*' );
    add(t);
  }
}
```

6 Text Fields





8.2 Layout Manager

- ***Layout Manager*** helfen das Layout von ***Panels*** - also die Plazierung einzelner Komponenten - zu organisieren.
- Da Java den Anspruch erhebt, plattformunabhängig zu sein, vermeidet der ***Layout Manager*** Angaben fester Positionen, sondern gestattet nur “relative” Angaben.
- Er regelt dann dynamisch in Abhängigkeit physikalischer Gegebenheiten, wie
  - verschieden auflösende Displays,
  - verschiedene Fonts,
  - etc.

  die jeweils tatsächliche Form der Darstellung.

8.2 Layout Manager

- Jedes *Panel* auf dem *Screen* (Bildschirm) hat seinen eigenen ***Layout Manager*.**
- Das ***AWT-package*** stellt (z. Zt.) fünf verschiedene Manager bereit:
  - *FlowLayout*
  - *GridLayout*
  - *BorderLayout*
  - *CardLayout* (wird nicht behandelt)
  - *GridBagLayout* (wird nicht behandelt)
- Typischerweise erzeugt man den ***Layout Manger*** in der Initialisierungsphase eines Applets.

1 FlowLayout

- Mit dem ***FlowLayout*** werden Komponenten von links nach rechts in Reihen angeordnet.
- Man kann sie links, rechts oder zentriert ausrichten **(*align*).**
- **Konstruktoren:**

Constructors	Action
FlowLayout()	Constructs a new Flow Layout with a centered alignment and a default 5-pixel horizontal and vertical gap.
FlowLayout(int)	Constructs a new Flow Layout with the specified alignment and a default 5-pixel horizontal and vertical gap: FlowLayout.CENTER FlowLayout.RIGHT FlowLayout.LEFT
FlowLayout(int, int, int)	Constructs a new Flow Layout with the specified alignment and the indicated horizontal and vertical gaps.

1 FlowLayout

◆ ***FlowLayout Methods:***

Methods	Action
addLayoutComponent (String, Component)	Adds the specified component to the layout.
getAlignment()	Gets the alignment for this layout
getHgap()	Gets the horizontal gap between components
getVgap()	Gets the vertical gap between components
setAlignment()	Sets the alignment for this layout
setHgap()	Sets the horizontal gap between components
setVgap()	Sets the vertical gap between components

1 FlowLayout

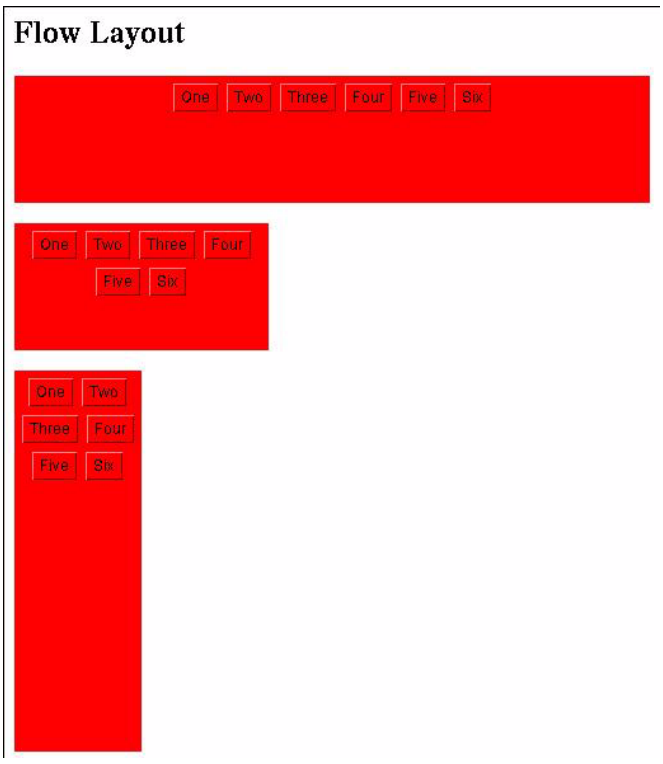
◆ Beispiel:

```
/* flowlayout test */

import java.awt.*;

public class FlowLayoutTest extends java.applet.Applet {

  public void init() {

    setLayout( new FlowLayout() );

    add( new Button( "One" ) );
    add( new Button( "Two" ) );
    add( new Button( "Three") );
    add( new Button( "Four" ) );
    add( new Button( "Five" ) );
    add( new Button( "Six" ) );
  }
}
```

1 FlowLayout

- HTML-File:

```
<HTML>
<HEAD>
<TITLE>Flow Layout</TITLE>
</HEAD>
<BODY>
<H2>Flow Layout</H2>
<P>
<APPLET CODE="FlowLayoutTest.class" WIDTH=500 HEIGHT=100>
</APPLET>
<P>
<APPLET CODE="FlowLayoutTest.class" WIDTH=200 HEIGHT=100>
</APPLET>
<P>
<APPLET CODE="FlowLayoutTest.class" WIDTH=100 HEIGHT=300>
</APPLET>
<P>
<A HREF="FlowLayoutTest.java">The Source</A>
</BODY>
</HTML>
```

2 Grid Layout

- Mit dem ***GridLayout*** wird das Panel in eine Anzahl Felder unterteilt.
- Die Komponenten werden in der Reihenfolge ihrer ***add()'s*** von links nach rechts und von oben nach unten angeordnet.
- **Konstruktoren**:

Constructors	Action
GridLayout()	Constructs a new Grid Layout with the defaul of one column per component, in a single row.
GridLayout(int, int)	Constructs a new Grid Layout with the specified numbers of rows and columns.
GridLayout(int, int, int, int)	Constructs a new Grid Layout with the specified numbers of rows and columns and the specified h-gaps and v-gaps.

2 Grid Layout

- ***GridLayout Methods***:

Methods	Action
addLayoutComponent (String, Component()	Adds the specified component to the layout.
getHgap()	Gets the horizontal gap between components
getVgap()	Gets the vertical gap between components
setHgap()	Sets the horizontal gap between components
setVgap()	Sets the vertical gap between components
getColoumns()	Get the number of columns in this layout.
getRows()	Get the number of rows in this layout.
settColoumns()	Set the number of columns in this layout.
settRows()	Set the number of rows in this layout.

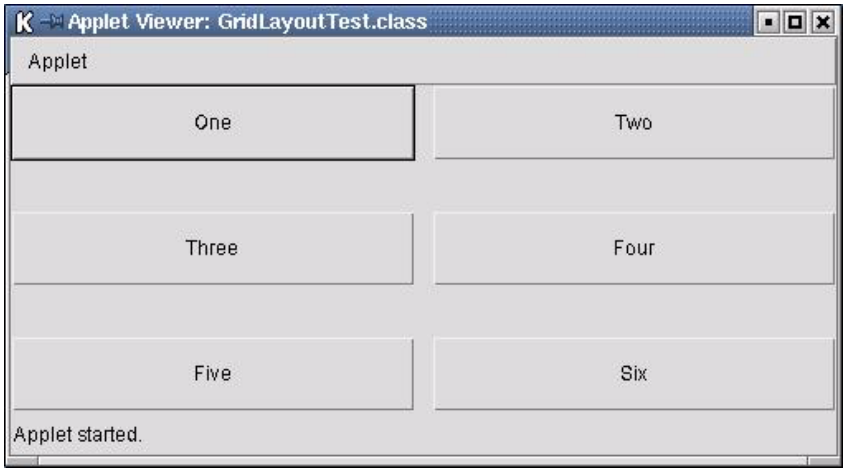
2 Grid Layout

◆ Beispiel:

```
/* grid layouts */

import java.awt.*;

public class GridLayoutTest extends java.applet.Applet {

  public void init() {

    setLayout( new GridLayout( 3,2,10,30 ) );
    add( new Button( "One" ) );
     add( new Button( "Two" ) );
     add( new Button( "Three" ) );
     add( new Button("Four" ) );
     add( new Button( "Five" ) );
     add( new Button( "Six" ) );
  }
}
```


2 Grid Layout

• Ergebnis mit Appletviewer:



3 Border Layout

- Mit dem ***Border Layout*** arrangiert man die Komponenten an den Begrenzungskanten.
- Der ***layout manager*** reserviert dafür “angemessen” Platz.
- Der restliche Platz wird einem zentralen Bereich zugeordnet.
- Bei der Plazierung der Komponenten *(**add()**)* gibt man als String zusätzlich die gewünschte Position an:
  - *North*
  - *East*
  - *South*
  - *West*
  - *Center*

3 Border Layout

- Folgende **Konstruktoren** stehen zur Verfügung:

• `setLayout( new BorderLayout() ):`

Komponenten werden im *BorderLayout* angeordnet.

• `setLayout( new BorderLayout( h,v ) ):`

Komponenten werden im ***BorderLayout*** angeordnet mit:

- horizontalem Zwischenraum *h Pixel*
- vertikalem Zwischenraum *v Pixel* .

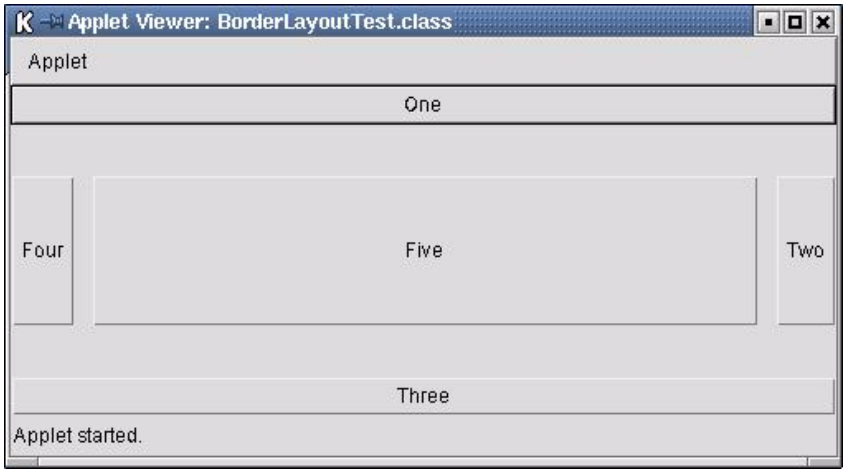
3 Border Layout

◆ Beispiel:

```
/* border layouts */

import java.awt.*;

public class BorderLayoutTest extends java.applet.Applet
{

  public void init() {

    setLayout( new BorderLayout( 10,30 ) );

    add( "North", new Button( "One" ) );
    add( "East", new Button( "Two" ) );
    add( "South", new Button( "Three" ) );
    add( "West", new Button( "Four" ) );
    add( "Center", new Button( "Five" ) );
  }
}
```


3 Border Layout

• Ergebnis mit Appletviewer:



4 Zugabe: Grid Bag Layout

- Das *GridBagLayout* ist zunächst einmal dem *GridLayout* sehr ähnlich. Zusätzlich sind jedoch die Größen der Zellen, die Proportionen der Spalten und Zeilen untereinander und das Layout der einzelnen Zelle individuell veränderbar, wobei die Festlegungen (*Constraints*) immer **relativ** erfolgen.
- Für das *GridBagLayout* benötigen wir zwei Klassen
  - *GridBagLayout:* für das Gesamtlayout
  - *GridBagConstrains:* für das interne Layout
- Im folgenden soll in einzelnen Schritten ein Entwurf am Beispiel erläutert werden:
  - Layout auf dem Papier
  - Gridlayout mit Platzhaltern als Komponenten
  - Festlegung der Proportionen
  - Layout der Einzelzellen
  - ggf. Überarbeitung des Designs

4 Grid Bag Layout

- Layout auf dem Papier:

4 Grid Bag Layout

- *Gridlayout* mit Platzhaltern als Komponenten
  - Zunächst einmal definieren wir uns eine “Hilfsmethode” um das Setzen der Constraints zu vereinfachen:

```
void buildConstraints(GridBagConstraints gbc, int gx, int gy,
      int gw, int gh, int wx, int wy) {
      gbc.gridx = gx;          // x - index des Grids
      gbc.gridy = gy;          // y - index des Grids
      gbc.gridwidth = gw;      // Anzahl der Grids pro Kompo-
                               // nente in x - Richtung
      gbc.gridheight = gh;     // Anzahl der Grids pro Kompo-
                               // nente in y - Richtung
      gbc.weightx = wx;        // Zeilenproportionen
      gbc.weighty = wy;        // Spaltenproportionen
}
```

  - Global legen wir zunächst fest, dass die zu definierenden Komponenten den gesamten ihnen zur Verfügung stehenden Bereich sowohl in x-Richtung als auch in y-Richtung (BOTH) belegen sollen.

```
constraints.fill = GridBagConstraints.BOTH;
```


4 Grid Bag Layout

- Die Einzelkomponenten werden in 4 Schritten kreiert:
  - Setzen der *Constraints*

```
buildConstraints(constraints, 0, 0, 1, 1, 100, 100);
```

    Für die linke obere Komponente (0, 0,..) wird vereinbart, dass sie einen Grid belegt (., 1, 1, ..);
    die Zeilen- und Spaltenproportionen werden wir im nächsten Schritt betrachten
  - Instantiieren der Komponente

```
Button label1 = new Button("Name:");
```

  - Zuordnung der *Constraints* zu den Komponenten

```
gridbag.setConstraints(label1, constraints);
```

  - Hinzufügen der Komponente auf das *Panel*

```
add(label1);
```

4 Grid Bag Layout

◆ Das komplette Layout:

```
/* border layouts */

import java.awt.*;

public class GridBagTestStep1 extends java.applet.Applet {

  void buildConstraints(GridBagConstraints gbc, int gx, int gy,
      int gw, int gh, int wx, int wy) {
    gbc.gridx = gx;         // x - index des Grids
    gbc.gridy = gy;         // y - index des Grids
    gbc.gridwidth = gw;     // Anzahl der Grids pro Komponente
                            // in x - Richtung
    gbc.gridheight = gh;    // Anzahl der Grids pro Komponente
                            // in y - Richtung
    gbc.weightx = wx;       // Zeilenproportionen
    gbc.weighty = wy;       // Spaltenproportionen
  }
```


4 Grid Bag Layout

```
public void init() {
  GridBagLayout gridbag = new GridBagLayout();
  GridBagConstraints constraints = new GridBagConstraints();
  setLayout(gridbag);

  constraints.fill = GridBagConstraints.BOTH;

  // Name label
  buildConstraints(constraints, 0, 0, 1, 1, 100, 100);
  Button label1 = new Button("Name:");
  gridbag.setConstraints(label1, constraints);
  add(label1);

  // Name text field
  buildConstraints(constraints, 1, 0, 1, 1, 100, 100);
  Button tfname = new Button();
  gridbag.setConstraints(tfname, constraints);
  add(tfname);
```

4 Grid Bag Layout

```
        // password label
        buildConstraints(constraints, 0, 1, 1, 1, 100, 100);
        Button label2 = new Button("Password:");
        gridbag.setConstraints(label2, constraints);
        add(label2);

        // password text field
        buildConstraints(constraints, 1, 1, 1, 1, 100, 100);
        Button tfpass = new Button();
        gridbag.setConstraints(tfpass, constraints);
        add(tfpass);

        // OK Button
        buildConstraints(constraints, 0, 2, 2, 1, 100, 100);
        Button okb = new Button("OK");
        gridbag.setConstraints(okb, constraints);
        add(okb);
    }
}
```


4 Grid Bag Layout

- Proportionen
  Die Integerzahlenwerte geben die relativen Längen und Höhen bezogen auf die Gesamtlänge bzw. Gesamthöhe an. Denkt man an Prozentangaben, so muss die Summe aller Einzelangaben 100 sein; braucht man es feiner ist Promille eine Möglichkeit. Grundsätzlich ist man völlig frei in der Wahl.
  Wir werden im Weiteren mit “Prozenten” arbeiten.
  - Eine Integerzahl bedeutet also der prozentuale Anteil.
  - Eine “0” bedeutet, dass die Proportionen “übernommen” werden.
- Im ersten Schritt legen wir die Proportionen der **Spalten** fest. Wir machen dies, indem wir die Elemente der oberen Zeile zueinander ins Verhältnis setzen und zwar 10 : 90:

```
buildConstraints(constraints, 0, 0, 1, 1, 10, ?);
buildConstraints(constraints, 1, 0, 1, 1, 90, ?);
```

4 Grid Bag Layout

- Die 2. Zeile soll das gleiche Layout übernehmen, es wird also eine “0” angegeben für Zelle (0,1) und (1,1).

```
buildConstraints(constraints, 0, 1, 1, 1, 0, ?);
buildConstraints(constraints, 1, 1, 1, 1, 0, ?);
```

- Die 3. Zeile besteht nur aus einer Zelle; eine Aufteilung ist nicht möglich. Auch hier wird eine “0” eingegeben.

```
buildConstraints(constraints, 0, 2, 2, 1, 0, ?);
```

- Das **Zeilen**layout designen wir jeweils mit den Elementen der 1. Spalte und zwar im Verhältnis 40:40:20. Die anderen Elemente - soweit vorhanden - übernehmen das Layout:

```
buildConstraints(constraints, 0, 0, 1, 1, 10, 40);
buildConstraints(constraints, 1, 0, 1, 1, 90, 0);
buildConstraints(constraints, 0, 1, 1, 1, 0, 40);
buildConstraints(constraints, 1, 1, 1, 1, 0, 0);
buildConstraints(constraints, 0, 2, 2, 1, 0, 20);
```


4 Grid Bag Layout

- Das komplette Layout:

```
/* grid layouts */

import java.awt.*;

public class GridBagTestStep2 extends java.applet.Applet {

  void buildConstraints(GridBagConstraints gbc, int gx,
            int gy, int gw, int gh, int wx, int wy) {

    gbc.gridx = gx;
    gbc.gridy = gy;
    gbc.gridwidth = gw;
    gbc.gridheight = gh;
    gbc.weightx = wx;
    gbc.weighty = wy;
  }
```

4 Grid Bag Layout

```
public void init() {
  GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
setLayout(gridbag);

constraints.fill = GridBagConstraints.BOTH;

// Name label
buildConstraints(constraints, 0, 0, 1, 1, 10, 40);
Button label1 = new Button("Name:");
gridbag.setConstraints(label1, constraints);
add(label1);

// Name text field
buildConstraints(constraints, 1, 0, 1, 1, 90, 0);
Button tfname = new Button();
gridbag.setConstraints(tfname, constraints);
add(tfname);
```


4 Grid Bag Layout

```
    // password label
    buildConstraints(constraints, 0, 1, 1, 1, 0, 40);
    Button label2 = new Button("Password:");
    gridbag.setConstraints(label2, constraints);
    add(label2);

    // password text field
    buildConstraints(constraints, 1, 1, 1, 1, 0, 0);
    Button tfpass = new Button();
    gridbag.setConstraints(tfpass, constraints);
    add(tfpass);

    // OK Button
    buildConstraints(constraints, 0, 2, 2, 1, 0, 20);
    Button okb = new Button("OK");
    gridbag.setConstraints(okb, constraints);
    add(okb);
  }
}
```

4 Grid Bag Layout

- Als letzten Schritt designen wir die einzelnen Zellen. Hierfür sind zwei Klassenvariable in der *GridBagConstraits class* definiert und zwar

  `fill`

  - mit dem Wertebereich
    - strecke die Komponente so, dass sie horizontal und vertikal die Zelle ausfüllt:

      `GridBagConstraints.BOTH`
    - Stelle die Komponente “minimal” dar:

      `GridBagConstraints.NONE`
    - strecke die Komponente nur horizontal:

      `GridBagConstraints.HORIZONTAL`
    - strecke die Komponente nur vertikal:

      `GridBagConstraints.VERTICAL`

4 Grid Bag Layout

- Als letzten Schritt ...und

  `anchor`

  - mit dieser Variablen wird die Position innerhalb einer Zelle festgelegt, wobei die jeweilige Dimension nicht bereits durch die *fill*-Variable “besetzt” sein darf. Der Wertebereich ist mit

    `GridBagConstraints.NORTH; .NORTHEAST; .EAST;... bis .NORTHWEST und .CENTER (alle 45 Grad)`

    beschrieben.

4 Grid Bag Layout

- Das finale Layout:

```
/* grid bag layouts */

import java.awt.*;

public class GridBagTestFinal extends java.applet.Applet {

  void buildConstraints(GridBagConstraints gbc, int gx,
              int gy, int gw, int gh, int wx, int wy) {

    gbc.gridx = gx;
    gbc.gridy = gy;
    gbc.gridwidth = gw;
    gbc.gridheight = gh;
    gbc.weightx = wx;
    gbc.weighty = wy;
  }
```


4 Grid Bag Layout

```
  public void init() {
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new
GridBagConstraints();
    setLayout(gridbag);

    // Name label
    buildConstraints(constraints, 0, 0, 1, 1, 10, 40);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.EAST;
    Label label1 = new Label("Name:", Label.LEFT);
    gridbag.setConstraints(label1, constraints);
    add(label1);

    // Name text field
    buildConstraints(constraints, 1, 0, 1, 1, 90, 0);
    constraints.fill = GridBagConstraints.HORIZONTAL;
    TextField tfname = new TextField();
    gridbag.setConstraints(tfname, constraints);
    add(tfname);
```

4 Grid Bag Layout

```
    // password label
    buildConstraints(constraints, 0, 1, 1, 1, 0, 40);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.EAST;
    Label label2 = new Label("Password:", Label.LEFT);
    gridbag.setConstraints(label2, constraints);
    add(label2);

    // password text field
    buildConstraints(constraints, 1, 1, 1, 1, 0, 0);
    constraints.fill = GridBagConstraints.HORIZONTAL;
    TextField tfpass = new TextField();
    tfpass.setEchoCharacter('*');
    gridbag.setConstraints(tfpass, constraints);
    add(tfpass);

    // OK Button
    buildConstraints(constraints, 0, 2, 2, 1, 0, 20);
    constraints.fill = GridBagConstraints.NONE;
    constraints.anchor = GridBagConstraints.CENTER;
    Button okb = new Button("OK");
    gridbag.setConstraints(okb, constraints);
    add(okb);
    }
}
```


4 Grid Bag Layout

- Die *Constraints ipadx* und *ipady*
  Mit diesen *Constraints* kann man zusätzlich um eine Komponente einen Abstand schaffen.
- *Insets*
  Mit *Insets* wird der Abstand der Komponenten eines Panels vom äußeren Rand des Panels bestimmt, und zwar mit vier Parametern, die den Abstand der jeweiligen Ecken festlegen