

Die JGUIToolbox

Ein Werkzeugkasten zur Erzeugung von Graphikoberfläche in Java

Vorwort

Die Vermittlung der Programmiersprache Java hat den Weg in viele Schulen gefunden.

Leider zeigt sich schnell, dass die Programmierung graphischer Oberflächen mit großen Mühen verbunden ist (wie in vielen anderen Programmiersprachen auch).

Zwar gibt es mittlerweile viele (kostenlose)

Programmierungsumgebungen, die den Weg erleichtern.

Die meisten helfen, indem sie den eigenen Quellcode mit den nötigen Codefragmenten ergänzen. Der Quellcode wird dabei nicht übersichtlicher. Gerade bei Elementen, die auf Benutzeraktionen reagieren sollen, entsteht viel Code, deren Fülle Schülerinnen und Schülern, die gerade ihre ersten Schritte machen überfordert.

Die hier vorgestellte **Toolbox** entstand mit dem Ziel, auch im Anfangsunterricht auf graphische Komponenten zugreifen zu können, ohne den Quellcode undurchsichtig und den Aufwand übermäßig werden zu lassen.

Die Idee zu der Toolbox wurde vom BlueJ-Beispiel „**shapes**“ übernommen.

Die Toolbox versucht die Idee aus „**shapes**“ weiterzuführen.

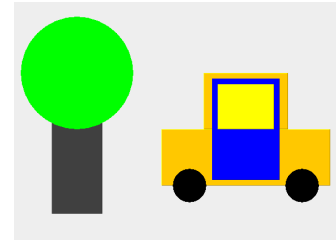
Dem Schüler steht eine große Palette von GUI-Elementen zur Verfügung. Ziel war vor allem eine einfache Integration von Elementen wie Tasten und Ausgabeelementen.

Die Klassen sind für die Verwendung in **BlueJ** entwickelt. Sie daher enthalten zusätzliche Variablen, die im Objektinspektor Informationen über den Status der Objekte geben. Einer Verwendung in jeder anderen Entwicklungsumgebung steht nichts im Weg.

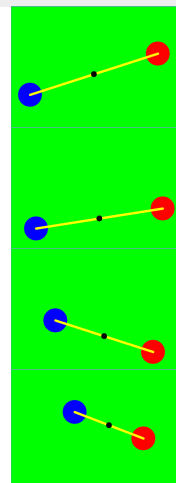
Für Experten:

Um die Anzahl der Methoden im Objektinspektor zu begrenzen, wurde bei der Entwicklung der Klassen ein mehrstufiger Aufbau gewählt. Näheres dazu im Anhang C.

Hans Witt



1 € kostet 1.5\$	
\$	75.0
€	50.0



Inhaltsverzeichnis

Die JGUIToolbox

Inhaltsverzeichnis.....	2
Übersicht über bisher vorhandenen die Komponenten.....	5
Einführung.....	7
Erste Schritte in BlueJ:.....	7
Transport von Codefragmenten:.....	7
Beispiele für den Einsatz der Toolbox:.....	8
Ein Graphik-Projekt:.....	8
Abfrage von IO-Komponenten	9
Callback von IO-Komponenten.....	10
Ein Währungsrechner.....	11
Die Klassen der Toolbox.....	12
Konstruktoren.....	12
Der Standardkonstruktor:.....	12
Weitere Konstruktoren:.....	12
Für Fortgeschrittene:.....	12
Eigene Dialoge:.....	12
Standardmethoden:.....	13
Die Farben:.....	13
Aktive Komponenten (z.B. Tasten).....	13
Programmieren mit Callback-Funktionen:.....	14
Graphik.....	15
Kreis.....	15
Ellipse.....	16
Bogen.....	16
Sektor.....	16
Rechteck.....	17
RechteckMitRundenEcken.....	18
Quadrat.....	19
Dreieck.....	20
Linie.....	21
Der Pfeil.....	22
FreiZeichnen.....	23
Taktgeber.....	25
GUI.....	26
Ausgabe.....	26
AusgabePanel.....	26
Taste.....	27
FeststellTaste.....	28
UmschaltTaste.....	29
UmschaltTasteMitAnzeige.....	30
ChkTaste.....	31
RadioTaste und Radiobehaelter.....	32
Eingabefeld.....	34
Rollbalken.....	35

Schieberegler.....	36
Combobox.....	37
Listbox.....	38
Siebensegment.....	39
Segment.....	39
Siebensegment.....	39
Segment8x.....	39
Behälter – Die „Klassen“ für graphische Elemente.....	40
Der Behälter.....	40
Der MausBehälter.....	41
Signalisieren von Ereignissen:.....	41
Die IDs der Mausereignisse:.....	41
Einschalten der Mausereignisse.....	42
Spuren.....	44
SpurNX.....	44
Standarddialoge:.....	45
Der Meldungsdialog.....	45
Der Bestätigungsdialog.....	46
Der Eingabedialog.....	47
Erstellung eigener Dialoge.....	48
Der modale Dialog.....	48
Der nicht modale Dialog.....	48
Kommunikation	48
Die Kommunikationsform der Toolbox.....	49
Dialogbeispiele.....	50
Beispiel für einen modalen Dialog.....	50
Das Hauptprogramm:.....	50
Der modale Dialog	51
Beispiel für einen nicht modalen Dialog:.....	52
Das Hauptprogramm:.....	52
Der nicht modale Dialog	53
Das Grundgerüst der Toolbox.....	54
Die Klasse Zeichnung.....	54
setzeFenstergroesse.....	54
setzeRasterEin.....	54
setzeRasterAus.....	54
setzeDeltaX.....	54
setzeDeltaY.....	54
Die Klasse StaticTools.....	55
getColor.....	55
jetzt_Minute.....	55
jetzt_Sekunde.....	55
jetzt_Stunde.....	55
jetzt.....	55
warte.....	55
Anhang A: Programmiersysteme.....	56
Anhang B: Für mich hilfreiche (elektronische) Literatur.....	57
Anhang C: Der Aufbau der Toolbox.....	58
Zum Schluss:.....	60

Übersicht über bisher vorhandenen die Komponenten



Kreis



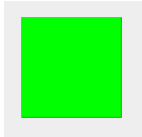
Ellipse



Rechteck



Rechteck
MitRundenEcken



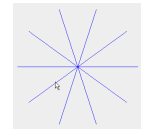
Quadrat



Dreieck



Linie



Freizeichnen

Ausgabebeispiel

Ausgabe



AusgabePanel

Taste

Taktgeber

Taktgeber



FeststellTaste

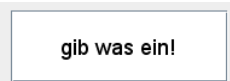


UmschaltTaste

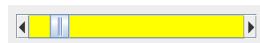


UmschaltTaste
MitAnzeige

ChkTaste



Eingabefeld



Rollbalken

Schieberegler

Behälter

Behaelter



Segment

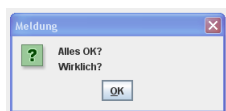
Siebensegment



Segment8x

Maus-sensitiver Behälter

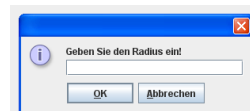
Mausbehaelter



Meldungsdialog



Bestätigungsdialog



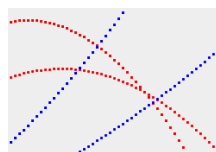
Eingabedialog

Selbst gestaltete Dialoge

eigene Dialoge



Pfeil



Spur



Bild

Audio

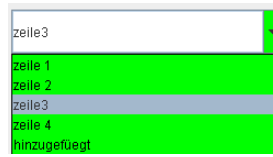
Audio



Behälter mit
Scrollern



RadioTasten
und Radiobehälter



Combobox



Listbox

Einführung

Mit Hilfe der Elemente der Toolbox sollen auch Programmieranfänger in die Lage versetzt werden, graphischen Benutzeroberflächen selbst zu gestalten.

Die **JGUIToolbox** ist eine Sammlung von Java-Klassen, die als Blackboxes verwendet werden können. Nach Außen sollen nur die unbedingt nötigen Details sichtbar sein.

Beim Einsatz der **JGUIToolbox** sind zwei Klassen immer dabei:

Die Klasse **Zeichnung** und die Klasse **StaticTools**.

Die Klasse Zeichnung erzeugt das Programmfenster. Das erfolgt automatisch durch das Erzeugen der ersten Komponente der Toolbox!

Die Klasse StaticTools ist ein Behälter für statische Methoden, die häufig gebraucht werden. Dazu gehören Methoden zu Umwandeln von Farbnamen (String) in die Java-Variante(Color).

Ein Beispiel sagt mehr als viele Worte.

Daher zuerst einige Beispiele, die den Einsatz einiger Komponenten (und den dafür nötigen Aufwand) zeigen.

Erste Schritte in BlueJ:

- Legen Sie mit BlueJ ein Projekt an.
- Kopieren (importieren) Sie den Quelltext der benötigten Klassen in das Projektverzeichnis. Nach dem Übersetzen können Sie die Klassen wie in BlueJ üblich verwenden.

Sie können in BlueJ **interaktiv** Objekte dieser Klassen anlegen und die erzeugten Objekte mit den Methoden der Klasse verändern.

Im **Objektinspektor** von BlueJ kann man jederzeit den Zustand der GUI-Objekte sehen. Dazu besitzen die Objekte Variablen, in denen die an die eigentlichen GUI-Objekte von Java weitergereichten Werte gespeichert sind.

Transport von Codefragmenten:

Löschen Sie vor der Weitergabe von Programmfragmenten die *.class-Dateien.

Sollte die Java-Version auf dem Zielsystem sich von der Version des Quellsystems unterscheiden, so treten **unvorhersehbare Effekte** auf!!

Die Klassen der **JGUIToolbox** sind für den Einsatz mit **BlueJ** angepasst. Sie können jedoch problemlos in anderen Entwicklungsumgebungen wie Javaeditor, Eclipse, Netbeans eingesetzt werden.

Beispiele für den Einsatz der Toolbox:

Ein Graphik-Projekt:

```
public class B_Auto {

    private Rechteck stamm;
    private Kreis krone;
    private Rechteck reUnten;
    private Rechteck reOben;

    private Rechteck tuer;
    private Rechteck fenster;
    private Kreis radLi;
    private Kreis radRe;

    public B_Auto() {

        stamm = new Rechteck(90, 200);
        stamm.setzePosition(105, 200);
        stamm.setzeFarbe("dunkelgrau");

        krone = new Kreis(100);
        krone.setzePosition(50, 50);
        krone.setzeFarbe("gruen");

        reUnten = new Rechteck(300, 100);
        reUnten.setzePosition(300, 250);
        reUnten.setzeFarbe("orange");

        reOben = new Rechteck(150, 100);
        reOben.setzePosition(375, 150);
        reOben.setzeFarbe("orange");

        tuer = new Rechteck(120, 180);
        tuer.setzePosition(390, 160);
        tuer.setzeFarbe("blau");

        fenster = new Rechteck(100, 80);
        fenster.setzePosition(400, 170);
        fenster.setzeFarbe("gelb");

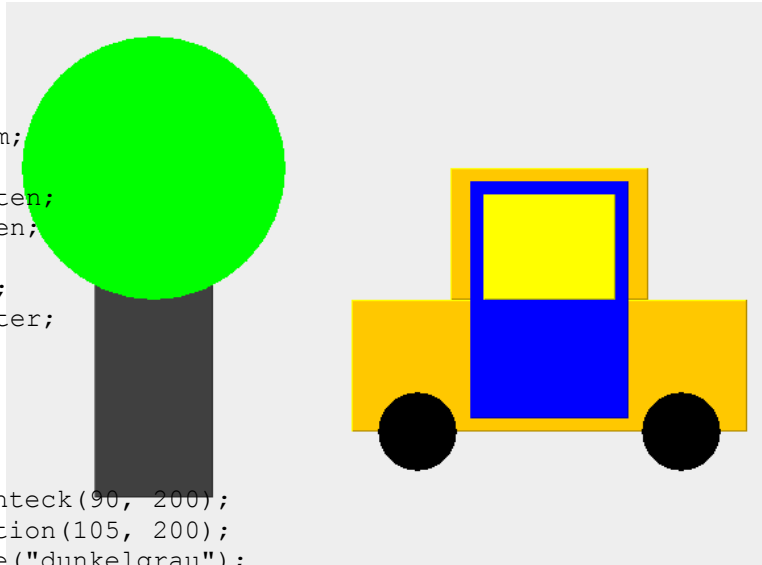
        radLi = new Kreis( 30 );
        radLi.setzePosition(320, 320 );
        radLi.setzeFarbe("schwarz");

        radRe = new Kreis( 30 );
        radRe.setzePosition(520, 320 );
        radRe.setzeFarbe("schwarz");

    }

    public static void main(String[] args)
        new B_Auto();

}
```



Mit main-Methode ist die Ausführung
in jeder Java-Umgebung möglich!
Bei BlueJ nicht nötig.

Abfrage von IO-Komponenten

```

public class B_Blinken_UT_Polling {

    private Kreis lampeLi ;
    private Kreis lampeRe ;

    private UmschaltTaste ut ;

    public B_Blinken_UT_Polling() {
        lampeLi = new Kreis(50);
        lampeLi.setzeFarbe("gelb");
        lampeLi.setzePosition(100, 100);

        lampeRe = new Kreis(50);
        lampeRe.setzeFarbe("gelb");

        lampeRe.setzePosition(300, 100);

        ut = new UmschaltTaste("Drück mich", 300, 100);
        ut.setzePosition(100, 220);
        ut.setzeHintergrundfarbe("gruen");
        ut.setzeGroesse(300, 60);
    }

    public void action() {
        while (true) {
            if (ut.istGewaehlt()) {
                lampeLi.fuellen();
                lampeRe.rand();
            }
            StaticTools.warte(200);
            if (ut.istGewaehlt()) {
                lampeLi.rand();
                lampeRe.fuellen();
            }
            StaticTools.warte(200);
        }
    }

    public static void main(String[] args) {
        B_Blinken_UT_Polling p = new B_Blinken_UT_Polling();
        p.action();
    }
}

```

Die Taste

Abfrage der Taste

Bei BlueJ muss nach dem Erzeugen eines Objekts der Klasse `B_Blinken_UT_Polling` die Methode **action()** aufgerufen werden.

Callback von IO-Komponenten

Das gleiche Beispiel mit **Callback**

```
public class B_Blinken_UT_Callback implements ITuWas {

    private Kreis lampeLi;
    private Kreis lampeRe;
    private UmschaltTaste ut;

    public B_Blinken_UT_Callback() {
        lampeLi = new Kreis(50);
        lampeLi.setzeFarbe("gelb");
        lampeLi.setzePosition(100, 100);
        lampeRe = new Kreis(50);
        lampeRe.setzeFarbe("gelb");
        lampeRe.setzePosition(300, 100);
        ut = new UmschaltTaste("Drück mich", 300, 100);
        ut.setzePosition(100, 220);
        ut.setzeHintergrundfarbe("gruen");
        ut.setzeGroesse(300, 60);
        ut.setzeLink(this, 0);
    }

    boolean bAction = false;
    public void tuWas(int ID) {
        if (ID == 0) {
            bAction = true;
            ut.setzeAnzeigetext("Anhalten");
        } else {
            bAction = false;
            ut.setzeAnzeigetext("Starten");
        }
    }

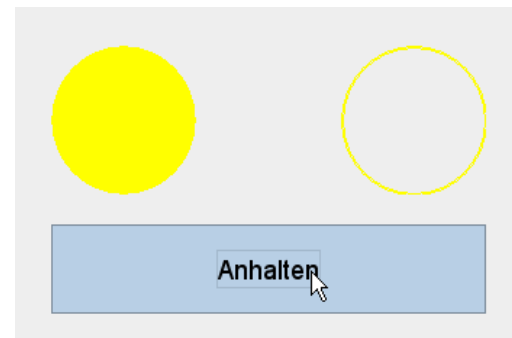
    public void action() {
        while (true) {
            if (bAction) {
                lampeLi.fuellen();
                lampeRe.rand();
            }
            StaticTools.warte(200);
            if (bAction) {
                lampeLi.rand();
                lampeRe.fuellen();
            }
            StaticTools.warte(200);
        }
    }

    public static void main(String[] args) {
        B_Blinken_UT_Callback p = new B_Blinken_UT_Callback();
        p.action();
    }
}
```

Interface
Erzwingt die Callbackfunktion

Callback-Link
für die Taste

Die Callback-Funktion:
ID == 0 --> gedrückt
ID == 1 --> nicht gedrückt



Bei BlueJ muss nach dem Erzeugen eines Objekts der Klasse B_Blinken_UT_Polling die Methode **action()** aufgerufen werden.

Ein Währungsrechner

```

public class B_Waehrungsrechner implements ITuWas {

    AusgabePanel titel;

    Ausgabe lDollar;
    Ausgabe lEuro;
    Eingabefeld dollar;
    Eingabefeld euro;

    double unmrrechnungskursEnachD = 1.5;

    public B_Waehrungsrechner() {

        titel = new AusgabePanel(
            "1 € kostet " + unmrrechnungskursEnachD + "$", 250, 100);
        titel.setzePosition(200, 0);

        lDollar = new Ausgabe("$", 100, 50);
        lDollar.setzePosition(200, 200);
        dollar = new Eingabefeld("Dollar", 200, 100);
        dollar.setzePosition(250, 200);
        dollar.setzeLink(this, 0);
        lEuro = new Ausgabe("€", 100, 50);
        lEuro.setzePosition(200, 300);

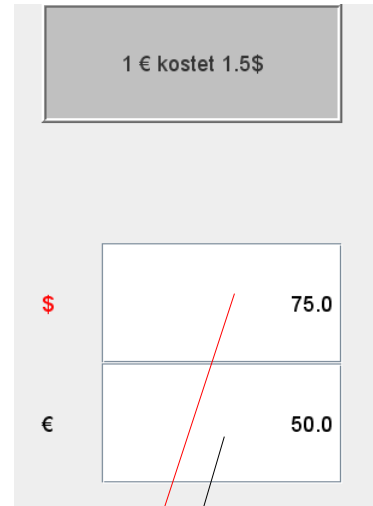
        euro = new Eingabefeld("Euro", 200, 100);
        euro.setzePosition(250, 300);
        euro.setzeLink(this, 1);

    }

    public void tuWas(int ID) {
        switch (ID) {
            case 0: // Auslöser: ENTER bei dollar
                euro.setzeDouble(dollar leseDouble(0) /
                    unmrrechnungskursEnachD);
                break;
            case 1: // Auslöser: ENTER bei euro
                dollar.setzeDouble(euro leseDouble(0) *
                    unmrrechnungskursEnachD);
                break;
            default:
                break;
        }
    }

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

```



Die Klassen der Toolbox

Konstrukturen

Der Standardkonstruktor:

Alle Klassen der Toolbox besitzen einen Standardkonstruktor ohne Parameter.

Weitere Konstrukturen:

Diese Konstrukturen haben als Parameter häufig gebrauchten Attribute. Das sind beispielsweise die Position, Größe oder Aufschriften (bei Tasten oder Labels).

Attribute können im Allgemeinen nur über Methoden gesetzt werden.

Ein direkter Zugriff auf Statusvariablen ist normalerweise nicht möglich. Bei einer Änderung der Statusvariablen wie z.B. der Position muss diese Information auch an die Graphik-Komponente im Hintergrund weitergeleitet werden. Das geschieht beim Aufruf der „setter“-Methoden.

Um in abgeleiteten Klassen Zugriff auf Statusvariablen zu ermöglichen, wurde die

„**Sichtbarkeit**“ der Zustandsvariablen auf **protected** gesetzt

Die Komponenten der Toolbox werden normalerweise **dem Hauptfenster** hinzugefügt.

Für Fortgeschrittene:

Komponenten werden normal dem Hauptfenster zugeordnet. Die Konstrukturen mit Attribut Behälter erlauben andere Zuordnungen, z.B. Dialogen.

Die Komponente **Behälter** ist ein alternativer Behälter für Elemente. Elemente werden relativ zum von Behälter abgeleiteten Objekt positioniert. Das erleichtert z.B. das Positionieren von in einem Behälter zusammengefassten Elementen. Beim Verschieben des Behälter-Objekts werden automatisch die eingebetteten Elemente verschoben.

Der **Maus-sensitive Behälter** ist eine aktive Komponente, die über Callback Mausereignisse signalisiert.

Eigene Dialoge:

Die Komponenten der Toolbox können zum Erzeugen **eigener Dialoge** verwendet werden. Beim Erstellen eigener Dialoge wird als Behälter der Behälter des Dialogs angegeben. Siehe Dialogkomponente.

Standardmethoden:

Alle graphischen Komponenten besitzen Methoden zum **Positionssetzen** bzw. zum **Verändern der Größe**:

```
public void setzePosition(int neuesX, int neuesY)
public void setzeGroesse(int neueBreite, int neueHoehe)
public void setzeDimensionen(int neuesX, int neuesY ,
                             int neueBreite, int neueHoehe){
public void setzeRadius(int neuerRadius) // beim Kreis
setzeGroesse gibt die Größe des umschließenden Rechtecks an.
```

Die Farben:

Komponenten besitzen Methoden zum **Setzen des Farbattributs**.

```
public void setzeFarbe(String neueFarbe) {
```

Je nach Komponente variieren die Bezeichnungen (z.B. setzeSchriftfarbe)

Farbattribute werden als Zeichenkette übergeben.

Als Farben stehen zur Verfügung:

"rot", "gelb", "blau", "gruen", "lila", "schwarz", "weiss",

"grau", "pink", "magenta", "orange", "cyan", "hellgrau".

Dazu stehen zur Verfügung die Namen "F01", "F02",... "F10".

Die Farbe dieser Namen wird durch die Methode

```
StaticTools.setzeFarbe(String farbname, int r, int g, int b)
```

selbst festgelegt. r,g,b sind Werte für rot,grün, blau aus dem Bereich 0 bis 255.

(In der Klasse StaticTools übersetzt eine Methode die Farbnamen in den entsprechenden Java (Color)-Wert.

Aktive Komponenten (z.B. Tasten)

Die **Klassendiagramme** der Klassen von **JGUIToolbox** finden Sie auf den folgenden Seiten mit kurzen Erläuterungen der Methoden. (Die Klassendiagramme sind aus dem Quelltext mit dem Javaeditor -siehe Anhang A- erzeugt)

Alle aktiven Klassen besitzen Methoden, mit denen ihr Status abgefragt werden kann bzw. der Status gesetzt werden kann. (Siehe Beschreibung der einzelnen Klassen)

Programmieren mit Callback-Funktionen:

Jede aktive Klasse besitzt eine Methode zum Übergeben eines übergeordneten Objekts.

```
public void setzeLink (ITuWas linkObj, int ID)
```

Will man direkt durch **Events (Ereignisse)** der Komponenten Aktionen in einem **Objekt obj** einer **Klasse Aktion** auslösen, so sind drei Schritte nötig:

- Die **Klasse Aktion** muss das Interface ITuWas implementieren:

```
public class Action implements ITuWas
```

Es muss also in dieser Klasse eine Methode

```
public void tuWas (int ID)
```

existieren.

- Das **Objekt Obj** der Klasse Aktion ruft die Methode

```
public void setzeLink (ITuWas linkObj, int ID)
```

der aktiven Komponente (z.B. einem Tastenobjekt taste) auf.

linkObj: Link auf eine Klasse mit Methode tuWas(int ID), meist sich selbst(this).

ID: eine Nummer, die die aktive Komponente identifiziert.

Codefragment: **taste.setzeLink(this,0);**

Wird nun bei dieser Komponente ein **Ereignis** ausgelöst, z.B. ein Tatendruck, so ruft diese Komponente die tuWas-Methode des Objekts **obj** auf.

Als ID erhält man die bei **setLink** übergebene ID.

Hat eine Komponente **mehrere** Ereignisquellen, wie z.B. die Umschalttaste (Taste_gedrückt und Taste_gelöst), so besitzt diese Komponente entsprechen viele IDs, im Fall der Umschalttaste also ID+0 und ID+1.

Dies muss bei der Vergabe der IDs für Komponenten berücksichtigt werden.

- In der Methode **tuWas(int ID)** wird für jede mögliche ID die gewünschte Aktion programmiert.
Innerhalb der Methode tuWas wird über die ID nach der Auslösequelle unterschieden.

Wird ein Objekt übergeben, so wird beim Eintritt des entsprechenden Ereignisses die **tuWas- Methode des übergebenen Objekts** aufgerufen.

Die zurückgegebene ID ist die Nummer des Ereignisses (beginnend mit 0) + die ID, die beim Aufruf von setLink übergeben wurde.

Bitte beachten:

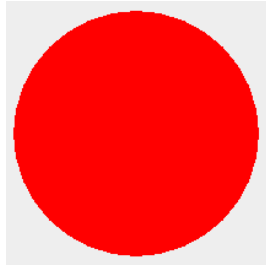
In Callback-Methoden sollten nur kurze/schnelle Aktionen aufgerufen werden.

Bei langen Schleifen sieht man eventuell nur das Ende!

Die Methode **setLink** einer Komponente erst dann aufrufen, wenn alle Komponenten verfügbar sind, die von einer Callback-Methode aufgerufen werden, am sichersten also am Ende des Konstruktors.

Graphik

Kreis



setzeMittelpunkt() setzt den Mittelpunkt des Kreises
 setzeRadius() ändert den Radius, der Mittelpunkt bleibt!

Wie bei anderen graphischen Komponenten setze ..

- setzePosition(...) die Linke obere Ecke.
- setzeGroesse(...) Ausdehnung

Kreis

obj: CKreis
 radius: int
 xPos: int
 yPos: int
 sichtbar: boolean
 gefuellt: boolean
 farbe: String

Kreis(...)
 Kreis(...)
 Kreis(...)
 Kreis(...)
 Kreis(...)
 sichtbarMachen(...)
 unsichtbarMachen(...)
 nachRechtsBewegen(...)
 nachLinksBewegen(...)
 nachObenBewegen(...)
 nachUntenBewegen(...)
 langsamVertikalBewegen(...)
 langsamHorizontalBewegen(...)
 setzeMittelpunkt(...)
 setzeRadius(...)
 setzeMittelpunktUndRadius(...)
 setzePosition(...)
 setzeGroesse(...)
 setzeDimensionen(...)
 setzeFarbe(...)
 horizontalBewegen(...)
 vertikalBewegen(...)
 fuellen(...)
 rand(...)

Ellipse

Das Element Ellipse umfasst auch die Elemente

Bogen

und

Sektor

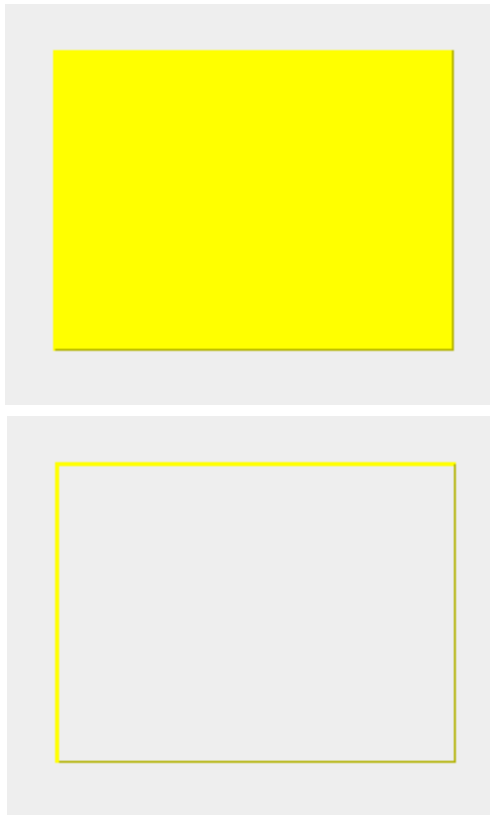


Startwinkel (0 = horizontal rechts) und
Bogenlänge sind im Gradmaß anzugeben.

Ist die Bogenlänge 360, so erhält man die Ellipse.
Ist die Bogenlänge kleiner als 360, so erhält man einen Sektor
(fuellen()) oder einen Bogen (rand()) beginnend ab dem
Startwinkel gegen den Uhrzeigersinn.

Ellipse
<div>[-] obj: CEllipse</div> <div>[-] breite: int</div> <div>[-] hoehe: int</div> <div>[-] xPos: int</div> <div>[-] yPos: int</div> <div>[-] startwinkel: int</div> <div>[-] bogenlaenge: int</div> <div>[-] sichtbar: boolean</div> <div>[-] gefuellt: boolean</div> <div>[-] farbe: String</div>
<div>[+] Ellipse()</div> <div>[+] Ellipse(int, int)</div> <div>[+] Ellipse(int, int, int, int)</div> <div>[+] setzeBogen(int, int): void</div> <div>[+] sichtbarMachen(): void</div> <div>[+] unsichtbarMachen(): void</div> <div>[+] nachRechtsBewegen(): void</div> <div>[+] nachLinksBewegen(): void</div> <div>[+] nachObenBewegen(): void</div> <div>[+] nachUntenBewegen(): void</div> <div>[+] langsamVertikalBewegen(int): void</div> <div>[+] langsamHorizontalBewegen(int): void</div> <div>[+] setzeGroesse(int, int): void</div> <div>[+] setzePosition(int, int): void</div> <div>[+] setzeFarbe(String): void</div> <div>[+] horizontalBewegen(int): void</div> <div>[+] vertikalBewegen(int): void</div> <div>[+] fuellen(): void</div> <div>[+] rand(): void</div> <div>[+] main(String[]): void</div>

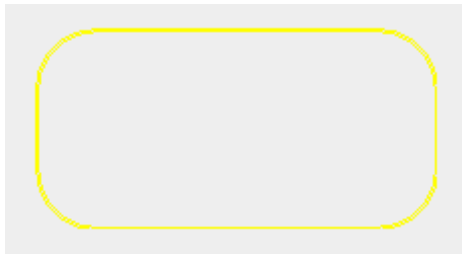
Rechteck



Rechteck
<ul style="list-style-type: none"> ▢ obj: CRechteck ▢ breite: int ▢ hoehe: int ▢ xPos: int ▢ yPos: int ▢ sichtbar: boolean ▢ gefuellt: boolean ▢ farbe: String
<ul style="list-style-type: none"> ⊞ Rechteck() ⊞ Rechteck(int, int) ⊞ sichtbarMachen(): void ⊞ unsichtbarMachen(): void ⊞ nachRechtsBewegen(): void ⊞ nachLinksBewegen(): void ⊞ nachObenBewegen(): void ⊞ nachUntenBewegen(): void ⊞ langsamVertikalBewegen(int): void ⊞ langsamHorizontalBewegen(int): void ⊞ setzeGroesse(int, int): void ⊞ setzePosition(int, int): void ⊞ setzeFarbe(String): void ⊞ horizontalBewegen(int): void ⊞ vertikalBewegen(int): void ⊞ fuellen(): void ⊞ rand(): void

RechteckMitRundenEcken

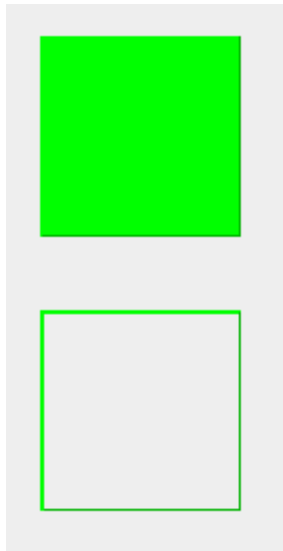
Mit der Größenangabe wird der Radius der Eck-Kreisbögen angegeben.



RechteckMitRundenEcken

- ▣ obj: CRechteckmitRundenEcken
- ▣ breite: int
- ▣ hoehe: int
- ▣ radius: int
- ▣ xPos: int
- ▣ yPos: int
- ▣ sichtbar: boolean
- ▣ gefuellt: boolean
- ▣ farbe: String

- ⊕ RechteckMitRundenEcken(...)
- ⊕ RechteckMitRundenEcken(...)
- ⊕ RechteckMitRundenEcken(...)
- ⊕ RechteckMitRundenEcken(...)
- ⊕ RechteckMitRundenEcken(...)
- ⊕ sichtbarMachen(...)
- ⊕ unsichtbarMachen(...)
- ⊕ nachRechtsBewegen(...)
- ⊕ nachLinksBewegen(...)
- ⊕ nachObenBewegen(...)
- ⊕ nachUntenBewegen(...)
- ⊕ langsamVertikalBewegen(...)
- ⊕ langsamHorizontalBewegen(...)
- ⊕ setzeGroesse(...)
- ⊕ setzePosition(...)
- ⊕ setzeDimensionen(...)
- ⊕ setzeFarbe(...)
- ⊕ horizontalBewegen(...)
- ⊕ vertikalBewegen(...)
- ⊕ fuellen(...)
- ⊕ rand(...)

Quadrat

Quadrat	
<input type="checkbox"/>	obj: CQuadrat
<input type="checkbox"/>	seite: int
<input type="checkbox"/>	xPos: int
<input type="checkbox"/>	yPos: int
<input type="checkbox"/>	sichtbar: boolean
<input type="checkbox"/>	gefullt: boolean
<input type="checkbox"/>	farbe: String
<input checked="" type="radio"/>	Quadrat()
<input checked="" type="radio"/>	Quadrat(int)
<input checked="" type="radio"/>	sichtbarMachen(): void
<input checked="" type="radio"/>	unsichtbarMachen(): void
<input checked="" type="radio"/>	nachRechtsBewegen(): void
<input checked="" type="radio"/>	nachLinksBewegen(): void
<input checked="" type="radio"/>	nachObenBewegen(): void
<input checked="" type="radio"/>	nachUntenBewegen(): void
<input checked="" type="radio"/>	langsamVertikalBewegen(int): void
<input checked="" type="radio"/>	langsamHorizontalBewegen(int): void
<input checked="" type="radio"/>	setzeGroesse(int): void
<input checked="" type="radio"/>	setzePosition(int, int): void
<input checked="" type="radio"/>	setzeFarbe(String): void
<input checked="" type="radio"/>	horizontalBewegen(int): void
<input checked="" type="radio"/>	vertikalBewegen(int): void
<input checked="" type="radio"/>	fuellen(): void
<input checked="" type="radio"/>	rand(): void

Dreieck

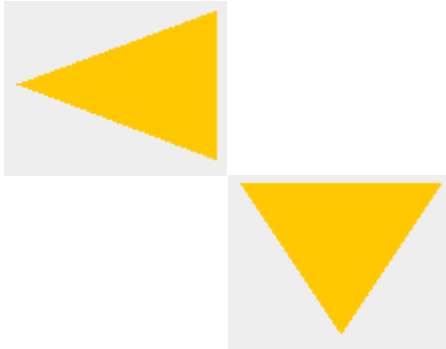
Die Ausrichtung gibt die Richtung der Spitze des Dreiecks an.

Mögliche Ausrichtungen:

- **N, O, S, W:**

Dreieck erzeugt ein symmetrisches Dreieck innerhalb des umschließenden Rechtecks.

Die Himmelsrichtung gibt die Spitze an.



- **NW, NO, SO, SW:**

rechtinkeliges Dreieck. Basis ist eine Diagonale. Die Himmelsrichtung gibt den rechten Winkel an.



Dreieck
<div>obj: CDreieck</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>sichtbar: boolean</div> <div>gefullt: boolean</div> <div>farbe: String</div> <div>ausrichtung: StaticTools.Richtung</div>
<div>Dreieck()</div> <div>Dreieck(int, int)</div> <div>setzeAusrichtung(String): void</div> <div>sichtbarMachen(): void</div> <div>unsichtbarMachen(): void</div> <div>nachRechtsBewegen(): void</div> <div>nachLinksBewegen(): void</div> <div>nachObenBewegen(): void</div> <div>nachUntenBewegen(): void</div> <div>langsamVertikalBewegen(int): void</div> <div>langsamHorizontalBewegen(int): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div> <div>setzeFarbe(String): void</div> <div>horizontalBewegen(int): void</div> <div>vertikalBewegen(int): void</div> <div>fuellen(): void</div> <div>rand(): void</div> <div>main(String[]): void</div>

Linie

Linien haben **Endpunkte**, eine **Farbe** und eine **Dicke**



Linie
<div><div></div>obj: CLinie</div> <div><div></div>x1: int</div> <div><div></div>y1: int</div> <div><div></div>x2: int</div> <div><div></div>y2: int</div> <div><div></div>sichtbar: boolean</div> <div><div></div>farbe: String</div> <div><div></div>linienDicke: int</div>
<div><div></div>Linie(...)</div> <div><div></div>Linie(...)</div> <div><div></div>Linie(...)</div> <div><div></div>Linie(...)</div> <div><div></div>sichtbarMachen(...)</div> <div><div></div>unsichtbarMachen(...)</div> <div><div></div>setzeEndpunkte(...)</div> <div><div></div>setzeFarbe(...)</div> <div><div></div>setzeLinienDicke(...)</div>

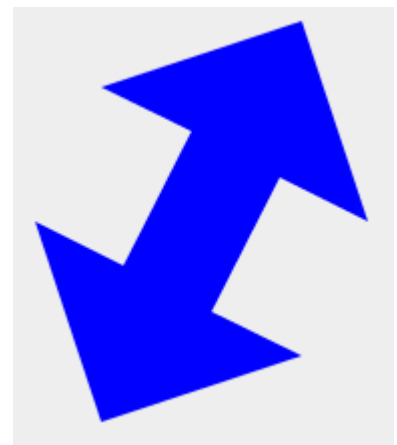
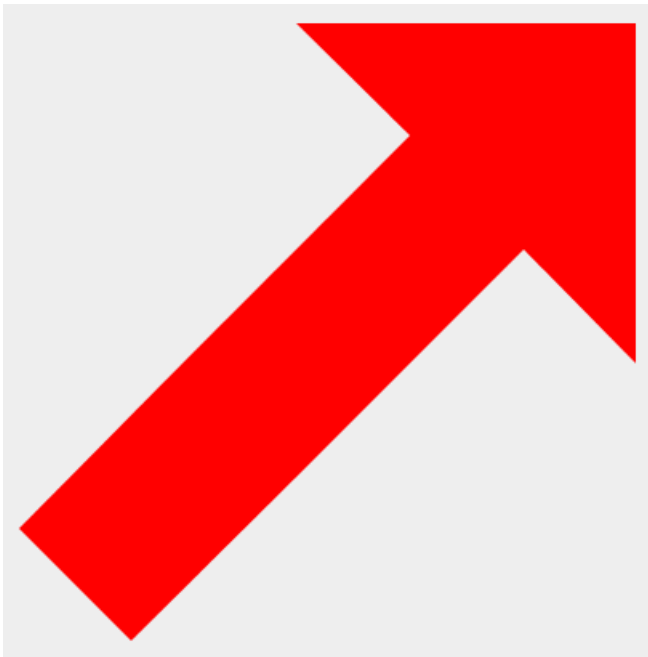
Der Pfeil

Pfeile haben **Endpunkte**, eine **Farbe** und eine **Dicke**

Die Methode

```
public void setzeEnden(boolean start, boolean ende) {
```

aktiviert an Anfang bzw. Ende die Spitze



Pfeil	
	obj: CPfeil
	x1: int
	y1: int
	x2: int
	y2: int
	sichtbar: boolean
	farbe: String
	breite: int
	spitzeStart: boolean
	spitzeEnde: boolean
	Pfeil(...)
	Pfeil(...)
	Pfeil(...)
	Pfeil(...)
	sichtbarMachen(...)
	unsichtbarMachen(...)
	setzeEndpunkte(...)
	setzeFarbe(...)
	setzeBreite(...)
	setzeEnden(...)

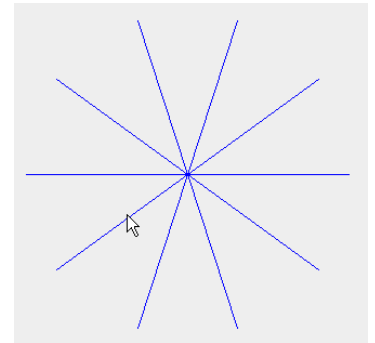
FreiZeichnen

Die Komponente FreiZeichnen ist ein Spezialfall.

Sie ist keine „fertige“ Komponente, die als Blackbox eingesetzt werden soll.

Sie ist vielmehr als Muster für weitere graphische Komponenten.

Will man sich eine neue graphische Komponente schaffen, so müssen die (im folgenden Quelltext hervorgehobenen) Teile geändert werden:



- Der **Name der Komponente/Klasse**.
BlueJ ändert dabei auch den Namen der Quelldatei. In anderen IDEs muss das selbst erledigt **werden**.
- Die **Namen der Konstruktoren**
- In der Methode **paintComponentSpezial** werden die eigenen Graphikbefehle programmiert.
Das Beispiel zeichnet einen Stern

FreiZeichnen
<div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>strFarbe: String</div> </div>
<div> <div>FreiZeichnen(...)</div> <div>FreiZeichnen(...)</div> <div>paintComponentSpezial(...)</div> <div>setzeGroesse(...)</div> <div>setzePosition(...)</div> <div>setzeFarbe(...)</div> </div>

```

public class FreiZeichnen extends BasisKomponente {
    private int breite = 100;
    ...

    public FreiZeichnen() {
        ...
    }

    public FreiZeichnen(int neueBreite, int neueHoehe) {
        ...
    }
    /**
     * Die Darstellung der Komponente wird hier programmiert.
     */
    public void paintComponentSpezial(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;
        // Graphik-Abmessungen
        breite = getSize().width ; // Breite der Komponente
        hoehe = getSize().height ; // Höhe der Komponente
        g.setColor(farbe);          // Setze die Zeichenfarbe
        .....
        // Hier stehen dann die eigenen Zeichenbefehle.
        // Beispiel: g2.drawLine(0, 0, 100, 200 );
    }

```

Ohne Anspruch auf Vollständigkeit eine Auswahl der Methoden.

Weitere Informationen in der API in den Klassen Graphics und Graphics2D.

Da Graphics2D von Graphics abgeleitet ist, können Methoden von Graphics auch bei g2 angewendet werden.

Hinweis:

Die Methode **drawXXX** zeichnet den **Umriss** der Figur, während die entsprechende Methode **fillXXX** eine **gefüllte Figur** zeichnet.

Die Sinn der Parameter ist im Allgemeinen offensichtlich. Besondere Parameter werden kurz erläutert. Ansonsten wird auf die Java-API verwiesen.

```
public void drawLine(int x1, int y1, int x2, int y2)
```

```
public void fillRect(int x, int y, int width, int height)
```

```
public void drawRect(int x, int y, int width, int height)
```

```
public void draw3DRect(int x, int y, int width, int height, boolean raised)
```

```
public void fill3DRect(int x, int y, int width, int height, boolean raised)
```

```
public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

```
public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

drawString zeichnet den Text an der angegebenen Position.

```
public void drawString(String str, int x, int y)
```

drawPolyline zeichnet ein Polygon.

Die Koordinaten werden in einem Feld `int[] xPoints` für die x-Koordinaten und `int[] yPoints` für die y-Koordinaten übergeben. `nPoints` gibt die Anzahl der Punkte an.

```
public void drawPolyline(int[] xPoints, int[] yPoints, int nPoints)
```

drawPolygon zeichnet ein geschlossenes Polygon. Wenn Anfangs- und Endpunkt nicht übereinstimmen wird das Polygon automatisch geschlossen.

```
public void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

```
public abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)
```

Taktgeber

Der Taktgeber kann bei graphischen Aktionen als Ersatz für Schleifen verwendet werden. Der Taktgeber ruft in der tuWas-Methode wiederholt die gewünschte Aktion auf.

Sollen mehrere Animationsstränge parallel laufen, bereitet die Wiederholung Probleme. Alles muss in einer Schleife integriert sei, da diese Schleife alle anderen anhält.

Jede Komponente kann einen eigenen Taktgeber installieren. Es ist keine zentrale Logik nötig. Die Methode tuWas zeichnet. Sie unterbricht das System nur kurz.

Der Methode **mehrfach** wird als Parameter die gewünschte Anzahl übergeben. Die Methode **endlos** ersetzt die Endlos-Schleife.

Die Methode **einmal** kann als definierte Zeitverzögerung benutzt werden.

Taktgeber	
wurdeSignalisiert	Seit der letzten Abfrage hat der Timer signalisiert
warteBisTaktsignal	Pause bis Timer signalisiert
setzeZeitZwischenAktionen	Zeit zwischen zwei Timersignalen
setzeAnfangszeitverzoeigerung	Zeit von Start() bis zum ersten Timersignal
mehrfach(int anzahl)	anzahl Timersignale
einmal() und einmal(int delay)	Ein einziges Timersignal (delay in ms)
endlos	immer
stop	Stop des Timers
boolean laufend()	Der Timer ist noch aktiv

Die Methoden mehrfach, einmal, endlos starten den Timer.

Taktgeber
<div> <div> <div></div> <div>t: Timer</div> </div> <div> <div></div> <div>delay: int</div> </div> <div> <div></div> <div>startDelay: int</div> </div> <div> <div></div> <div>timerSignal: boolean</div> </div> <div> <div></div> <div>anzahl: int</div> </div> <div> <div></div> <div>begrenzteAnzahl: boolean</div> </div> <div> <div></div> <div>linkObj: ITuWas</div> </div> <div> <div></div> <div>id: int</div> </div> </div>
<div> <div>Ⓢ</div> <div>Taktgeber()</div> </div> <div> <div>Ⓢ</div> <div>Taktgeber(ITuWas, int)</div> </div> <div> <div>Ⓢ</div> <div>tuWas(): void</div> </div> <div> <div>Ⓢ</div> <div>setzeLink(ITuWas, int): void</div> </div> <div> <div>Ⓢ</div> <div>wurdeSignalisiert(): boolean</div> </div> <div> <div>Ⓢ</div> <div>warteBisTaktsignal(): void</div> </div> <div> <div>Ⓢ</div> <div>setzeZeitZwischenAktionen(int): void</div> </div> <div> <div>Ⓢ</div> <div>setzeAnfangszeitverzoeigerung(int): void</div> </div> <div> <div>Ⓢ</div> <div>mehrfach(int): void</div> </div> <div> <div>Ⓢ</div> <div>einmal(): void</div> </div> <div> <div>Ⓢ</div> <div>einmal(int): void</div> </div> <div> <div>Ⓢ</div> <div>endlos(): void</div> </div> <div> <div>Ⓢ</div> <div>stop(): void</div> </div> <div> <div>Ⓢ</div> <div>laufend(): boolean</div> </div>

GUI

Ausgabe

Ausgabebeispiel

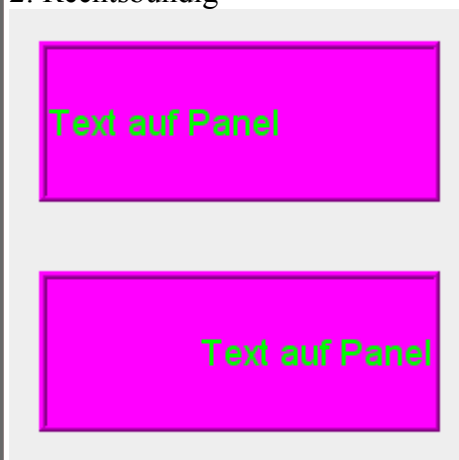
Ausgabe
<div>obj: CAusgabe</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>anzeigeText: String</div> <div>fontGroesse: int</div> <div>farbe: String</div>
<div>Ausgabe()</div> <div>Ausgabe(String, int, int)</div> <div>setzeAusgabertext(String): void</div> <div>setzeSchriftgroesse(int): void</div> <div>setzeFarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div>

AusgabePanel



setzeAusrichtung bestimmt die Ausrichtung des Texts innerhalb des Panels:

- 0: Linksbündig
- 1: Zentriert
- 2: Rechtsbündig



AusgabePanel
<div>obj: CAusgabePanel</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>anzeigeText: String</div> <div>fontGroesse: int</div> <div>hintergrundFarbe: String</div> <div>schriftFarbe: String</div> <div>ausrichtung: int</div>
<div>AusgabePanel()</div> <div>AusgabePanel(String, int, int)</div> <div>setzeAnzeigetext(String): void</div> <div>setzeSchriftgroesse(int): void</div> <div>setzeAusrichtung(int): void</div> <div>setzeSchriftfarbe(String): void</div> <div>setzeHintergrundfarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div>

Taste

wurdegedrueckt liefert **true**, wenn seit der letzten Abfrage die Taste gedrückt wurde.

Bitte beachten:

Die Callbackfunktion setzt diese Information NICHT zurück!

Taste
<div>obj: CTaste</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>anzeigeText: String</div> <div>fontGroesse: int</div> <div>schriftFarbe: String</div> <div>hintergrundFarbe: String</div> <div>gedrueckt: boolean</div>
<div>Taste()</div> <div>Taste(String, int, int)</div> <div>setzeLink(ITuWas, int): void</div> <div>setzeSchriftgroesse(int): void</div> <div>setzeSchriftfarbe(String): void</div> <div>setzeHintergrundfarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div> <div>wurdeGedrueckt(): boolean</div> <div>ruecksetzenGedrueckt(): void</div> <div>warteBisGedrueckt(): void</div>

FeststellTaste

Die FeststellTaste hat **zwei Zustände**:

StatusFunktion für Abfragen:

```
public boolean istGedrueckt()
```

Die FeststellTaste hat **zwei Anzeigetexte** für **gedrückt** und **nichtgedrückt**, genauso **zwei Farben**.

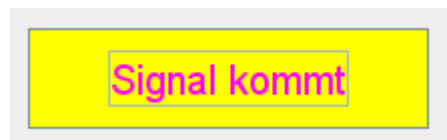
Sie können durch den Konstruktor und durch Methoden gesetzt werden.

Der Text und die Farbe wechselt automatisch!

Die FeststellTaste wird **durch die Maus gesetzt**. Sie kann ist dann verriegelt.

Sie kann **nur** durch die Methode **setzeNichtGewaeht zurückgesetzt** werden.

Über **Callback** kann der Feststell-Tastendruck weitergeleitet werden



FeststellTaste

- [-] knopf: Taste
- [-] breite: int
- [-] hoehe: int
- [-] xPos: int
- [-] yPos: int
- [-] fontGroesse: int
- [-] gedrueckt: boolean
- [-] anzeigeTextNichtGedrueckt: String
- [-] anzeigeTextGedrueckt: String
- [-] knopfFarbeNichtGedrueckt: String
- [-] knopfFarbeGedrueckt: String
- [-] schriftFarbe: String

- [+] FeststellTaste(...)
- [+] FeststellTaste(...)
- [+] FeststellTaste(...)
- [+] FeststellTaste(...)
- [+] FeststellTaste(...)
- [+] setzePosition(...)
- [+] setzeGroesse(...)
- [+] setzeDimensionen(...)
- [+] setzeTextNichtGedrueckt(...)
- [+] setzeTextGedrueckt(...)
- [+] setzeSchriftgroesse(...)
- [+] warteBisGedrueckt(...)
- [+] istGedrueckt(...)
- [+] setzeGewaeht(...)
- [+] setzeNichtGewaeht(...)
- [+] setzeFarbeNichtGedrueckt(...)
- [+] setzeFarbeGedrueckt(...)
- [-] zeigeStatus(...)
- [+] setzeSchriftfarbe(...)
- [+] tuWas(...)

UmschaltTaste

Die Umschalttaste hat **zwei Zustände**:

StatusFunktion für Abfragen:

```
public boolean istGewaeht()
```

Callback-ID:

Taste wurde gedrückt: ID + 0

Taste wurde gelöst: ID + 1



UmschaltTaste
<div>obj: CUmschaltTaste</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>anzeigeText: String</div> <div>fontGroesse: int</div> <div>schriftFarbe: String</div> <div>hintergrundFarbe: String</div> <div>gedrueckt: boolean</div>
<div>UmschaltTaste()</div> <div>UmschaltTaste(String, int, int)</div> <div>setzeLink(ITuWas, int): void</div> <div>setzeSchriftgroesse(int): void</div> <div>setzeAnzeigetext(String): void</div> <div>setzeSchriftfarbe(String): void</div> <div>setzeHintergrundfarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div> <div>istGewaeht(): boolean</div> <div>setzeGewaeht(): void</div> <div>setzeNichtGewaeht(): void</div>

UmschaltTasteMitAnzeige

Die UmschaltTasteMitAnzeige hat **zwei Zustände**:

StatusFunktion für Abfragen:

```
public boolean istGewaeht()
```

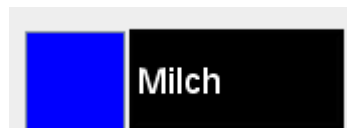
Callback-ID:

Taste wurde gedrückt: ID + 0

Taste wurde gelöst: ID + 1

Ausrichtung:

0 : Knopf Links Text Links



1 : Knopf Links Text Mitte



2 : Knopf Links Text Rechts



4 : Knopf Rechts Text Links

















5 : Knopf Rechts Text Mitte,

















6 : Knopf Rechts Text Rechts



UmschaltTasteMitAnzeige

-  knopf: UmschaltTaste
-  anzeige: AusgabePanel
-  breite: int
-  hoehe: int
-  xPos: int
-  yPos: int
-  fontGroesse: int
-  gedrueckt: boolean
-  anzeigeText: String
-  knopfFarbe: String
-  hintergrundFarbe: String
-  schriftFarbe: String
-  linkObj: ITuWas
-  ausrichtung: int

-  UmschaltTasteMitAnzeige(...)
-  UmschaltTasteMitAnzeige(...)
-  setzeAusrichtung(...)
-  setzePosition(...)
-  setzeGroesse(...)
-  setzeLink(...)
-  setzeAnzeigetext(...)
-  setzeSchriftgroesse(...)
-  istGewaeht(...)
-  setzeGewaeht(...)
-  setzeNichtGewaeht(...)
-  setzeFarbeknopf(...)
-  setzeFarbeAnzeigetext(...)
-  setzeSchriftfarbe(...)

ChkTaste

Die ChkTaste hat **zwei Zustände**:

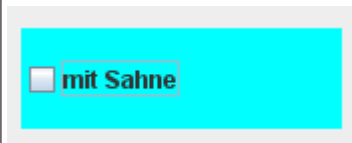
StatusFunktion für Abfragen:

```
public boolean istGewaeht()
```

Callback-ID:

Select: ID + 0

DeSelect: ID + 1



ChkTaste
<div>obj: CChkTaste</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>anzeigeText: String</div> <div>fontGroesse: int</div> <div>hintergrundFarbe: String</div> <div>schriftFarbe: String</div> <div>gewaeht: boolean</div>
<div>ChkTaste()</div> <div>ChkTaste(String, int, int)</div> <div>setzeLink(ITuWas, int): void</div> <div>setzeAnzeigetext(String): void</div> <div>istGewaeht(): boolean</div> <div>setzeSelect(): void</div> <div>setzeDeSelect(): void</div> <div>setzeSchriftfarbe(String): void</div> <div>setzeHintergrundfarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div>

RadioTaste und Radiobehaelter

Radio-Tasten treten im Gegensatz zu Chk-Tasten nie allein auf. Das besondere an Radio-Tasten ist, dass sie sich gegenseitig ausschließen. Wird eine Radio-Taste aktiv, so werden automatisch alle anderen Tasten der Gruppe deaktiviert.

Das hat auch Auswirkungen auf die Programmierung von Radio-Tasten. Man benötigt zwei Klassen, die Klasse RadioBehaelter und die Klasse RadioTaste

Objekte vom Typ **Radiotaste** werden in ein Objekt vom Typ **Radiobehaelter** eingefügt.

(Ein spezieller Behälter! Näheres siehe dort)

Der Radiobehälter ist ein spezieller Behälter für Radiotasten.

Durch das Einfügen wird gleichzeitig die Radio-Funktionalität implementiert. Radiotasten werden relativ zu ihrem Behälter positioniert.

Kommunikation mit den Radio-Tasten:

Jede Radio-Taste kann abgefragt werden:

if(r1.istGewählt()) ...

Jede Radio-Taste ist eine aktive Komponente. Durch das Einfügen in den Radiobehaelter wird automatisch ein Link auf den Radiobehälter gesetzt. Die gewählte Radio-Taste signalisiert automatisch dem Behälter.

Übergibt man dem Behälter einen Link, so signalisiert der Behälter jeden Wechsel in der Gruppe. Die dabei übergebene ID ist die Summe aus der ID des Behälters und der ID der Taste.

Spezialfall: Ändert man nach dem Einfügen in den Behälter den Link einer Radio-Taste, so signalisiert diese Taste nicht mehr dem Behälter, sondern dem neuen Ziel.

RadioTaste
obj: CRadioTaste breite: int hoehe: int xPos: int yPos: int anzeigeText: String fontGroesse: int hintergrundFarbe: String schriftFarbe: String gewaehlt: boolean
RadioTaste(Radiobehaelter) RadioTaste(Radiobehaelter, String, int, int, int, int) getBasisComponente(): BasisComponente setzeID(int): void setzeLink(ITuWas): void setzeLink(ITuWas, int): void setzeAnzeigetext(String): void istGewählt(): boolean setzeGewählt(): void setzeNichtGewählt(): void setzeSchriftfarbe(String): void setzeHintergrundfarbe(String): void setzeGroesse(int, int): void setzePosition(int, int): void verschieben(int, int): void setzeSchriftgroesse(int): void setzeDimensionen(int, int, int, int): void

Radiobehaelter
obj: CRadioBehaelter breite: int hoehe: int xPos: int yPos: int sichtbar: boolean zoomInhalt: double anzeigeText: String fontGroesse: int farbe: String
RadioBehaelter() RadioBehaelter(int, int) RadioBehaelter(int, int, int, int) RadioBehaelter(Container) RadioBehaelter(Container, int, int, int, int) setzeZoomfaktor(double): void getBehaelterZoom(): double hinzufuegen(RadioTaste): void setzeAusgabebetext(String): void setzeSchriftgroesse(int): void setzeFarbe(String): void setzeID(int): void setzeLink(ITuWas): void setzeLink(ITuWas, int): void getBasisComponente(): BasisComponente setzeGroesse(int, int): void setzePosition(int, int): void setzeDimensionen(int, int, int, int): void sichtbarMachen(): void unsichtbarMachen(): void setzeMitRand(boolean): void add(Component, int): Component setzeKomponentenKoordinaten(JComponent, int, int, int, int): void setzeKomponentenGroesse(JComponent, int, int): void setzeKomponentenPosition(JComponent, int, int): void validate(): void getPanel(): JPanel

Codefragment:

```

...

RadioBehaelter    radiobehaelter;
RadioTaste        r1;
RadioTaste        r2;
RadioTaste        r3;

...

// Im Konstruktor:
...
radiobehaelter = new RadioBehaelter(0, 152, 200, 47);
radiobehaelter.setzeBeschreibungstext("Farbe");
radiobehaelter.setzeFarbe("schwarz");
radiobehaelter.setzeHintergrundfarbe("gelb");

r1 = new RadioTaste(radiobehaelter, "r", 7, 15, 60, 30);
r1.setzeSchriftgroesse(10);
r1.setzeHintergrundfarbe("rot");
r1.setzeID(3);

r2 = new RadioTaste(radiobehaelter, "g", 67, 15, 60, 30);
r2.setzeSchriftgroesse(10);
r2.setzeHintergrundfarbe("gelb");
r2.setzeID(4);

r3 = new RadioTaste(radiobehaelter, "b", 127, 15, 60, 30);
r3.setzeSchriftgroesse(10);
r3.setzeHintergrundfarbe("blau");
r3.setzeID(5);

r1.setzeGewaeHLT();
radiobehaelter.setzeLink(this);
...

public void tuWas(int ID) {
    switch (ID) {

        ...

        case 3:
            ball.setzeFarbe("rot");
            radiobehaelter.setzeHintergrundfarbe(null);
            break;
        case 4:
            ball.setzeFarbe("gelb");
            break;
        case 5:
            ball.setzeFarbe("blau");
            radiobehaelter.setzeHintergrundfarbe("gelb");
            break;

        default :

    }
}
...

```

Die Taste wird in den Behälter gesetzt

Eingabefeld

Die **Callback-Methode** signalisiert, dass im Eingabefenster die **>>RETURN<<-Taste** gedrückt wurde.



Der Inhalt kann mit **leseText**, **leseInteger**, **leseIntegerGerundet** und **leseDouble** abgefragt werden.

Lesen von Zahlen:

Beim **Lesen von Zahlen** wird das Textfeld **mit dem gelesenen Wert überschrieben**.

Zahlen werden **rechtsbündig** in das Textfeld geschrieben.

Keine Zahl im Eingabefeld:

Die Methoden **leseInteger**, **leseIntegerGerundet** und **leseDouble** haben einen Parameter, der als Wert zurückgegeben wird, wenn das Textfeld der Komponente keinen Integer- bzw. Double-Wert enthält.

Eingabefeld
<ul style="list-style-type: none"> [-] obj: CEingabefeld [-] breite: int [-] hoehe: int [-] xPos: int [-] yPos: int [-] anzeigeText: String [-] fontGroesse: int [-] schriftFarbe: String [-] hintergrundFarbe: String
<ul style="list-style-type: none"> [+] Eingabefeld() [+] Eingabefeld(String, int, int) [+] setzeLink(ITuWas, int): void [+] setzeSchriftgroesse(int): void [+] setzeSchriftfarbe(String): void [+] setzeHintergrundfarbe(String): void [+] setzeGroesse(int, int): void [+] setzePosition(int, int): void [+] setzeAusgabetext(String): void [+] leseText(): String [+] zentrieren(): void [+] setzeInteger(int): void [+] leseInteger(int): int [+] setzeDouble(double): void [+] leseDouble(double): double

Rollbalken

Einstellungen:

**setzeBereich(double neuesMin, double neuesMax,
double neuerWert)**

und

setzeWert(double neuerWert)

Lesen:

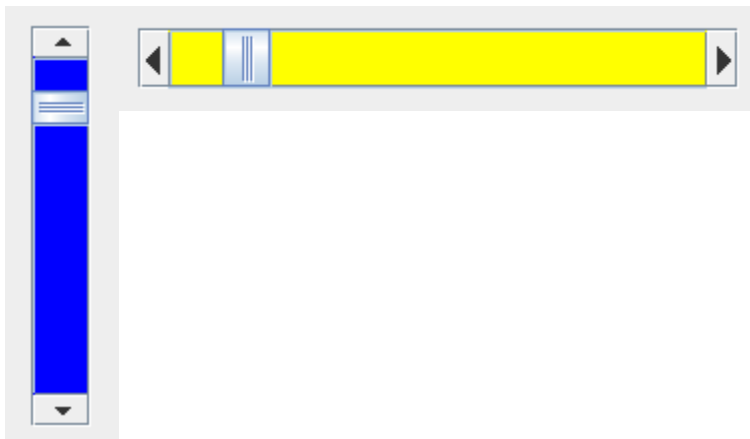
leseDoubleWert ()

und

leseIntWert ()

Über die Methode **hatSichGeaendert** kann der Status abgefragt werden. Statusabfrage, **Lesen** und **warteBisAenderung** setzen den Status automatisch zurück

Übergibt man Taste den **Link** auf eine Klasse mit dem Interface **IuWas**, so wird bei **Änderung** **tuWas(id)** aufgerufen wird



Rollbalken	
	obj: CRollbalken
	breite: int
	hoehe: int
	xPos: int
	yPos: int
	hintergrundFarbe: String
	min: double
	max: double
	wert: double
	Rollbalken(...)
	Rollbalken(...)
	Rollbalken(...)
	Rollbalken(...)
	Rollbalken(...)
	Rollbalken(...)
	Rollbalken(...)
	setzeBereich(...)
	setzeWert(...)
	leseDoubleWert(...)
	leseIntWert(...)
	setzeLink(...)
	setzeOrientierungHorizontal(...)
	setzeOrientierungSenkrecht(...)
	setzeFarbe(...)
	setzeGroesse(...)
	setzePosition(...)
	setzeDimensionen(...)
	hatSichGeaendert(...)
	ruecksetzenAenderung(...)
	warteBisAenderung(...)

Schieberegler

Einstellungen:

**setzeBereich(double neuesMin, double neuesMax,
double neuerWert)**

und

setzeWert(double neuerWert)

Lesen:

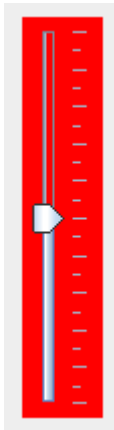
leseDoubleWert ()

und

leseIntWert()

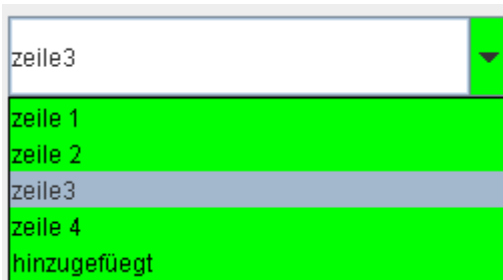
Über die Methode **hatSichGeaendert** kann der Status abgefragt werden. Statusabfrage, **Lesen** und **warteBisAenderung** setzen den Status automatisch zurück

Übergibt man Taste den **Link** auf eine Klasse mit dem Interface **Ituwas**, so wird bei **Änderung** **tuWas(id)** aufgerufen wird



Schieberegler
<div> <div>obj: CSchieberegler</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>hintergrundFarbe: String</div> <div>min: double</div> <div>max: double</div> <div>wert: double</div> </div>
<div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>Schieberegler(...)</div> </div> <div> <div>+</div> <div>setzeBereich(...)</div> </div> <div> <div>+</div> <div>setzeWert(...)</div> </div> <div> <div>+</div> <div>leseDoubleWert(...)</div> </div> <div> <div>+</div> <div>leseIntWert(...)</div> </div> <div> <div>+</div> <div>setzeLink(...)</div> </div> <div> <div>+</div> <div>setzeOrientierungHorizontal(...)</div> </div> <div> <div>+</div> <div>setzeOrientierungSenkrecht(...)</div> </div> <div> <div>+</div> <div>setzeFarbe(...)</div> </div> <div> <div>+</div> <div>setzeGroesse(...)</div> </div> <div> <div>+</div> <div>setzePosition(...)</div> </div> <div> <div>+</div> <div>setzeDimensionen(...)</div> </div> <div> <div>+</div> <div>hatSichGeaendert(...)</div> </div> <div> <div>+</div> <div>ruecksetzenAenderung(...)</div> </div> <div> <div>+</div> <div>warteBisAenderung(...)</div> </div> </div>

Combobox



Der Konstruktor

```
public Combobox( String [] texte ,int neuesX,
                int neuesY, int neueBreite, int neueHoehe)
```

übergibt die Einträge als String-Feld.

Spezielle Methoden

textHinzufuegen(String text)

bzw.

textHinzufuegenAnPosition(String text, int index)

ergänzen die Auswahltexte.

setzeEditierbar erlaubt, neue Einträge hinzuzufügen.

leseAuswahl liefert den ausgewählten Text.

leseAuswahlindex liefert den Index des ausgewählten Texts.

objektEntfernenAnPosition(int anIndex) und
alleObjekteEntfernen löscht Einträge

Combobox
<div> <div>obj: CCombobox</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>fontGroesse: int</div> <div>schriftFarbe: String</div> <div>hintergrundFarbe: String</div> </div>
<div> <div>Combobox()</div> <div>Combobox(int, int)</div> <div>Combobox(int, int, int, int)</div> <div>Combobox(String[], int, int, int, int)</div> <div>Combobox(IContainer)</div> <div>Combobox(IContainer, int, int, int, int)</div> <div>Combobox(IContainer, String[], int, int, int, int)</div> <div>getBasisKomponente(): BasisKomponente</div> <div>setzeID(int): void</div> <div>setzeLink(ITuWas): void</div> <div>setzeLink(ITuWas, int): void</div> <div>setzeSchriftgroesse(int): void</div> <div>setzeSchriftfarbe(String): void</div> <div>setzeHintergrundfarbe(String): void</div> <div>setzeGroesse(int, int): void</div> <div>setzePosition(int, int): void</div> <div>verschieben(int, int): void</div> <div>setzeDimensionen(int, int, int, int): void</div> <div>setzeEditierbar(): void</div> <div>setzeNichtEditierbar(): void</div> <div>leseAuswahl(): String</div> <div>setzeAuswahl(Object): void</div> <div>leseAuswahlIndex(): int</div> <div>setzeAuswahlIndex(int): void</div> <div>textHinzufuegen(String): void</div> <div>textHinzufuegenAnPosition(String, int): void</div> <div>objektEntfernenAnPosition(int): void</div> <div>alleObjekteEntfernen(): void</div> </div>

Listbox

zeile 1

zeile 2

zeile3

zeile 4

zeile 5

zeile 6

zeile 7

Der Konstruktor

```
public Listbox( String [] texte ,int neuesX,
               int neuesY, int neueBreite, int neueHoehe)
```

übergibt die Einträge als String-Feld.

Spezielle Methoden

textHinzufuegen(String text)

bzw.

textHinzufuegenAnPosition(String text, int index)

ergänzen die Auswahltexte.

SetzeMehrfachauswahl läßt die Auswahl von mehreren Einträgen zu (Strg-Taste gedrückt)

leseAuswahl liefert den (ersten)ausgewählten Text.

leseAuswahlindex liefert den (ersten) Index des ausgewählten Texts.

leseAuswahlindices liefert ein Feld von ausgewählten Indices

leseAuswahlen liefert ein Textfeld mit ausgewählten Texten

auswahlAufheben löscht die Auswahlen

objektEntfernenAnPosition(int anIndex) und
alleObjekteEntfernen löscht Einträge

Listbox
<div> <div>obj: CListbox</div> <div>breite: int</div> <div>hoehe: int</div> <div>xPos: int</div> <div>yPos: int</div> <div>fontGroesse: int</div> <div>schriftFarbe: String</div> <div>hintergrundFarbe: String</div> </div>
<div> <div>Ⓢ Listbox()</div> <div>Ⓢ Listbox(int, int)</div> <div>Ⓢ Listbox(int, int, int, int)</div> <div>Ⓢ Listbox(String[], int, int, int, int)</div> <div>Ⓢ Listbox(IContainer)</div> <div>Ⓢ Listbox(IContainer, int, int, int, int)</div> <div>Ⓢ Listbox(IContainer, String[], int, int, int, int)</div> <div>Ⓢ getBasisKomponente(): BasisKomponente</div> <div>Ⓢ setzeID(int): void</div> <div>Ⓢ setzeLink(ITuVWas): void</div> <div>Ⓢ setzeLink(ITuVWas, int): void</div> <div>Ⓢ setzeSchriftgroesse(int): void</div> <div>Ⓢ setzeSchriftfarbe(String): void</div> <div>Ⓢ setzeHintergrundfarbe(String): void</div> <div>Ⓢ setzeGroesse(int, int): void</div> <div>Ⓢ setzePosition(int, int): void</div> <div>Ⓢ verschieben(int, int): void</div> <div>Ⓢ setzeDimensionen(int, int, int, int): void</div> <div>Ⓢ leseAuswahl(): String</div> <div>Ⓢ leseAuswahlen(): String[]</div> <div>Ⓢ setzeAuswahl(Object): void</div> <div>Ⓢ leseAuswahlIndex(): int</div> <div>Ⓢ setzeAuswahlIndex(int): void</div> <div>Ⓢ leseAuswahlIndices(): int[]</div> <div>Ⓢ textHinzufuegen(String): void</div> <div>Ⓢ textHinzufuegenAnPosition(String, int): void</div> <div>Ⓢ objektEntfernenAnPosition(int): void</div> <div>Ⓢ alleObjekteEntfernen(): void</div> <div>Ⓢ setzeEinfachauswahl(): void</div> <div>Ⓢ setzeMehrfachauswahl(): void</div> <div>Ⓢ auswahlAufheben(): void</div> </div>

Siebensegment

Segment



Siebensegment

Segment8x

Ein Feld von Siebensegmentanzeigen.



Der Konstruktor:

```
public Segment8x(int stellen, int hoehe) {
```

stelle: **Anzahl** der Siebensegmentelemente

hoehe: **Höhe** eines Elements

Anzeige einer **ganzen Zahl**:

```
public void anzeige(long zahl)
```

Anzeige einer **Dezimalzahl** dzahl mit dp **Nachkommastellen**

```
public void anzeige(double dzahl, int dp)
```

dp = -1 : ganzzahlige Darstellung ohne Dezimalpunkt.

dp = 0 : Dezimalpunkt neben der letzten Ziffer.

Kein Runden !

Segment
<input type="checkbox"/> obj: CSegment
<input type="checkbox"/> breite: int
<input type="checkbox"/> hoehe: int
<input type="checkbox"/> xPos: int
<input type="checkbox"/> yPos: int
<input type="checkbox"/> sichtbar: boolean
<input type="checkbox"/> gefuellt: boolean
<input type="checkbox"/> farbe: String
<input type="checkbox"/> farbeRand: String
<input checked="" type="radio"/> Segment(...)
<input checked="" type="radio"/> Segment(...)
<input checked="" type="radio"/> sichtbarMachen(...)
<input checked="" type="radio"/> unsichtbarMachen(...)
<input checked="" type="radio"/> setzeGroesse(...)
<input checked="" type="radio"/> setzePosition(...)
<input checked="" type="radio"/> setzeFarbe(...)
<input checked="" type="radio"/> setzeRandfarbe(...)
<input checked="" type="radio"/> fuellen(...)
<input checked="" type="radio"/> rand(...)

SiebenSegment
<input type="checkbox"/> sa: Segment
<input type="checkbox"/> sb: Segment
<input type="checkbox"/> sc: Segment
<input type="checkbox"/> sd: Segment
<input type="checkbox"/> se: Segment
<input type="checkbox"/> sf: Segment
<input type="checkbox"/> sg: Segment
<input type="checkbox"/> dp: Kreis
<input type="checkbox"/> sBreit: int
<input type="checkbox"/> sHoch: int
<input type="checkbox"/> positionX: int
<input type="checkbox"/> positionY: int
<input type="checkbox"/> farbe: String
<input checked="" type="radio"/> SiebenSegment(...)
<input checked="" type="radio"/> zeigeDP(...)
<input checked="" type="radio"/> anzeige(...)
<input checked="" type="radio"/> anzeigeMinus(...)
<input checked="" type="radio"/> anzeige(...)
<input checked="" type="radio"/> SiebenSegment(...)
Segment8x
<input type="checkbox"/> s: SiebenSegment[]
<input type="checkbox"/> hoehe: int
<input type="checkbox"/> stellen: int
<input type="checkbox"/> xPos: int
<input type="checkbox"/> yPos: int
<input checked="" type="radio"/> Segment8x(...)
<input checked="" type="radio"/> setzePosition(...)
<input checked="" type="radio"/> setzeFarbe(...)
<input checked="" type="radio"/> setzeGroesse(...)
<input checked="" type="radio"/> anzeige(...)
<input checked="" type="radio"/> anzeige(...)
<input checked="" type="radio"/> laufen(...)
<input checked="" type="radio"/> main(...)

Behälter – Die „Klassen“ für graphische Elemente

Der Behälter

Der Behälter ist eine **Container-Klasse**.

Die Klasse Behälter kapselt die GUI-Darstellung der Komponenten.

Sie ist in einer "normalen" Java-Klasse die Ergänzung, in der die **Graphische Darstellung der Objekte** gekapselt ist.

Sie nimmt **graphische-Elemente** auf.

Der Behälter bestimmt einen rechteckigen Bereich, in dem die Objekte des Behälters dargestellt werden. Diese werden **relativ zum Behälter positioniert**.

Dies vereinfacht vor allem die Positionierung der im Behälter zusammengefassten Elemente. Verschiebt man den Behälter, so werden alle Komponenten im Behälter mit verschoben.

Wie die Klasse Zeichnung kann auch der Behälter ein **Raster** einschalten.

Die Standard-Konstruktoren fügen Elemente der Toolbox automatisch dem „Behälter“ des Programms hinzu.

Zusätzliche Konstruktoren erzeugen die Komponenten gleich im Ziel-Behälter. Setzt man diesen Behälter unsichtbar, so sieht man den Behälterinhalt erst nach dem Sichtbar-Mache. Der Bildaufbau wird ruhiger.

Mit der Methode **hinzufuegen**(IComponente obj) löst man die Komponente obj aus dem bisherigen Behälter und fügt ihn dem neuen Behälter hinzu.

Die Methode **setzeZoomfaktor**(double zf) zoomt alle Komponenten innerhalb des Behälters.

Beispiele:

SiebenSegment_B
Segment8x_B

Anwendung bei eigenen Dialogen.

Behaelter
☺ Behaelter() ☺ Behaelter(int, int) ☺ Behaelter(int, int, int, int) ☺ Behaelter(IContainer) ☺ Behaelter(IContainer, int, int, int) ☺ setzeZoomfaktor(double): void ☺ getBehaelterZoom(): double ☺ hinzufuegen(IComponente): void ☺ hinzufuegenUndAnpassen(IComponente): void ☺ getBasisComponente(): BasisComponente ☺ setzeGroesse(int, int): void ☺ setzePosition(int, int): void ☺ setzeDimensionen(int, int, int, int): void ☺ sichtbarMachen(): void ☺ unsichtbarMachen(): void ☺ horizontalBewegen(int): void ☺ vertikalBewegen(int): void ☺ nachRechtsBewegen(): void ☺ nachLinksBewegen(): void ☺ nachObenBewegen(): void ☺ nachUntenBewegen(): void ☺ langsamVertikalBewegen(int): void ☺ langsamHorizontalBewegen(int): void ☺ setzeMitRaster(boolean): void ☺ setzeDeltaX(int): void ☺ setzeDeltaY(int): void ☺ add(Component, int): Component ☺ setzeKomponentenKoordinaten(JComponent, int, int, int, int): void ☺ setzeKomponentenGroesse(JComponent, int, int): void ☺ setzeKomponentenPosition(JComponent, int, int): void ☺ validate(): void ☺ getPanel(): JPanel

Der MausBehälter

Der MausBehälter ergänzt den Behälter um Mausereignisse.

Wie beim Behälter werden Komponenten entweder beim Erzeugen im Behälter erzeugt oder später mit der Methode `hinzufuegen()` aus dem Ursprungsbehälter (meist dem Hauptfenster) hinzugefügt.

Signalisieren von Ereignissen:

Der Mausbehälter kann 8 verschiedene Ereignisse signalisieren.

Die beim Aufruf von `tuWas(int ID)` signalisierte ID ist die Summe aus der durch

`setzeLink(ITuWas linkObj, int BasisID)` übergebenen BasisID und der ID des Mausereignisses.

Wird das MausBehälterObjekt mit BasisID 10 initialisiert, so signalisiert der Mausbehälter das Ereignis RELEASE durch ID $(10+2) = 12$.

Die IDs der Mausereignisse:

Ereignis	ID	Beschreibung
CLICK	0	Drücken und Loslassen
PRESS	1	Drücken der Maustaste
RELEASE	2	Loslassen der Maustaste
ENTER	3	Maus bewegt sich in den Behälter
EXIT	4	Maus verlässt den Behälter
DRAGGED	5	Bewegung mit gedrückter Maustaste
MOVED	6	Bewegung der Maus
WHEEL	7	Das Mousrad wurde bewegt

MausBehaelter

```

+ CLICK: int
+ PRESS: int
+ RELEASE: int
+ ENTER: int
+ EXIT: int
+ DRAGGED: int
+ MOVED: int
+ WHEEL: int

+ MausBehaelter()
+ MausBehaelter(int, int)
+ MausBehaelter(int, int, int, int)
+ MausBehaelter(IContainer)
+ MausBehaelter(IContainer, int, int, int, int)
+ getBasisComponente(): BasisComponente
+ hinzufuegen(IComponente): void
+ hinzufuegenUndAnpassen(IComponente): void
+ setzeZoomfaktor(double): void
+ setzeLink(ITuWas, int): void
+ ruecksetzenMaus(): void
+ setzeMausClick(): void
+ setzeMausPressRelease(): void
+ setzeMausEnterExit(): void
+ setzeMausDraggedMoved(): void
+ setzeMausRad(): void
+ setzeMausereignisse(int): void
+ setzeAlleMausereignisse(): void
+ mausAktion(): boolean
+ getMX(): int
+ getMY(): int
+ getClickCount(): int
+ getButton(): int
+ getShift(): boolean
+ getCtrl(): boolean
+ getAlt(): boolean
+ getRotation(): int
+ getXPos(): int
+ getYPos(): int
+ setzeGroesse(int, int): void
+ setzePosition(int, int): void
+ setzeDimensionen(int, int, int, int): void
+ verschieben(int, int): void
+ sichtbarMachen(): void
+ unsichtbarMachen(): void
+ setzeMitRaster(boolean): void
+ setzeDeltaX(int): void
+ setzeDeltaY(int): void
+ add(Component, int): Component
+ setzeKomponentenKoordinaten(JComponent, int, int, int, int): void
+ setzeKomponentenGroesse(JComponent, int, int): void
+ setzeKomponentenPosition(JComponent, int, int): void
+ validate(): void
+ getPanel(): JPanel
+ getBehaelterZoom(): double

```


Einschalten der Mausereignisse

Die Maus signalisiert bis zu 8 verschiedene Ereignisse. Oft braucht man nur einzelne. Daher muss vor der Verwendung der Ereignisse das gewünschte Ereignis eingeschaltet werden. Folgende Methoden stellen Gruppen von Ereignissen ein. Gruppen werden zusätzlich zu bereits eingestellten gesetzt.

ruecksetzenMaus()	Alle Mausereignisse deaktivieren
setzeMausClick()	Click-Ereignis
setzeMausPressRelease()	Drücken und Loslassen
setzeMausEnterExit()	Mit der Maus die Komponente betreten bzw. verlassen
setzeMausDraggedMoved()	Mit gedrückter Maustaste (dragged) und ohne gedrückte Maustaste (moved) bewegen. Die Ereignisse finden bei gedrückter Maustaste bis zum Loslassen der Maustaste statt, also auch außerhalb des Mausbehälters!
setzeMausRad	Aktiviere das Mousrad
setzeAlleMausereignisse	Alle Ereignisse setzen.
setzeMausereignisse(int ereignisse)	Beim Parameter ereignisse wird pro Ereignis ein Bit gesetzt. Folgende Aufzählung aktiviert all. Für die eigene Zusammenstellung die gewünschten Zeilen übernehmen: (1 << MausBehaelter.CLICK) (1 << MausBehaelter.PRESS) (1 << MausBehaelter.RELEASE) (1 << MausBehaelter.ENTER) (1 << MausBehaelter.EXIT) (1 << MausBehaelter.DRAGGED) (1 << MausBehaelter.MOVED) (1 << MausBehaelter.WHEEL)

Zu Beachten:

Einige Mausereignisse führen zu mehreren Ereignissen:
Ein Click-Ereignis geht immer ein Press- und ein Release-Ereignis voraus.
Einem Doppelclick geht eine EinfachClick voraus usw.

Status beim Eintreten des Mausereignisses:

Methode mit Rückgabotyp	Beschreibung
<code>public boolean</code> <code>mausAktion()</code>	Zeigt, dass eine Mausaktion stattgefunden hat. Diese Methode kann verwendet werden, wenn ohne Callback der mausstatus abgefragt werden soll.
<code>public int</code> <code>getMX()</code>	x-Koordinate der Maus relativ zum Behälter
<code>public int</code> <code>getMY()</code>	y-Koordinate der Maus relativ zum Behälter
<code>public int</code> <code>getClickCount()</code>	Anzahl der Clicks (für Doppelclick usw.)
<code>public int</code> <code>getButton()</code>	Auslösende Maustaste: Links = 0 ...
<code>public boolean</code> <code>getShift()</code>	War die Umschalttaste gedrückt?
<code>public boolean</code> <code>getCtrl()</code>	War die Steuerungstaste gedrückt?
<code>public boolean</code> <code>getAlt()</code>	War die Alt Taste gedrückt?
<code>public int</code> <code>getRotation()</code>	Anzahl der Rotations-Ticks (rückwärts negativ)
<code>public int</code> <code>getXPos()</code>	x-Position des Mausbehälters
<code>public int</code> <code>getYPos()</code>	y-Position des Mausbehälters

Die Angabe sind die Angaben beim Eintritt des letzten Mausereignisses.

(Man sollte daher nicht benötigte Ereignisse deaktivieren. Sie verfälschen eventuell benötigte (Positions-)Daten.)

Spuren

SpurNX

Die Klasse SpurNX ist ein Objekt, auf dem man mehrere verschiedenfarbige Spuren hinterlassen kann. In den verschiedenen Konstruktoren gibt man die Anzahl der anzuzeigenden Spuren an:

```
public SpurNX(int anzahl)
oder
...
public SpurNX(IContainer behaelter, int
anzahl, int neuesX, int neuesY, int
neueBreite, int neueHoehe)
```

Die Methode

```
hinzufuegen(int nrSpur, int x, int y)
```

fügt einen Punkt der Spur **nrSpur** hinzu.

```
loescheSpur(int nrSpur)
```

löscht die Spur **nrSpur**

```
loescheSpuren()
```

löscht alle Spuren

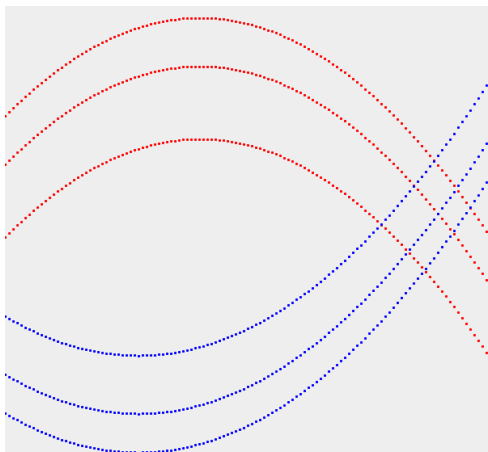
```
setzeFarbe(int nrSpur, String neueFarbe)
```

und

```
setzeDurchmesser(int nrSpur, int
durchmesser)
```

setzt die Farbattribute bzw. den Durchmesser der Spur „punkte“.

füllen() zeigt die Punkte als gefüllte Kreise, rand() als Kreisränder.



SpurNX

```
- obj: CSpurNX
- anzahl: int
# breite: int
# hoehe: int
# xPos: int
# yPos: int
# sichtbar: boolean
# gefuellt: boolean
# farbe: String
- ausrichtung: StaticTools.Richtung
```

```
+ SpurNX(...)
+ SpurNX(...)
+ SpurNX(...)
+ SpurNX(...)
+ SpurNX(...)
+ hinzufuegen(...)
+ loescheSpur(...)
+ loescheSpuren(...)
+ mitRahmen(...)
+ ohneRahmen(...)
+ setzeFarbe(...)
+ setzeDurchmesser(...)
+ sichtbarMachen(...)
+ unsichtbarMachen(...)
+ setzeGroesse(...)
+ setzePosition(...)
+ setzeDimensionen(...)
+ fuellen(...)
+ rand(...)
```

Standarddialoge:

Die folgenden Klassen kapseln Standarddialoge.

Der Meldungsdialog

setzeTitel und setzeMeldungstext

setzen die entsprechenden Texte

Die Methoden

hinweis()

warnung()

frage()

ohneIcon()

setzen die Icons des Meldungsfensters.

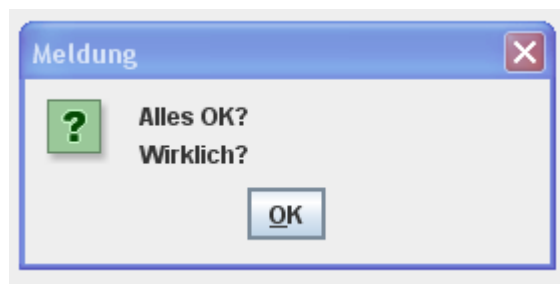
Die Methode **zeigeMeldung()** startet den Dialog.

Ein Zeilenumbruch für mehrzeiliger Meldungstext erreicht man durch die Zeichenfolge `\n`

Die Methode „setzeMeldungstext“ für folgendes Fenster ist:

setzeMeldungstext(“Alles OK?\nWirklich?“);

D_Meldung
titel: String
meldung: String
typ: int
D_Meldung(...)
zeigeMeldung(...)
setzeTitel(...)
setzeMeldungstext(...)
fehlermeldung(...)
hinweis(...)
warnung(...)
frage(...)
ohneIcon(...)



Der Bestätigungsdialog

Der Typ des Dialogs:

typJaNein()

typOkAbbruch()

typJaNeinAbbruch()

setzen die Anzahl und die Beschriftung der Tasten

setzeTitel und **setzeMeldungstext**

setzen die entsprechenden Texte

Die Methoden

hinweis()

warnung()

frage()

ohneIcon()

setzen die Icons des Meldungsfensters.

Die Methode **zeigeMeldung()** startet den Dialog.

Das Ergebnis des Meldungsfensters

`public char leseErgebnis ()`

Ergebniswerte sind Zeichen

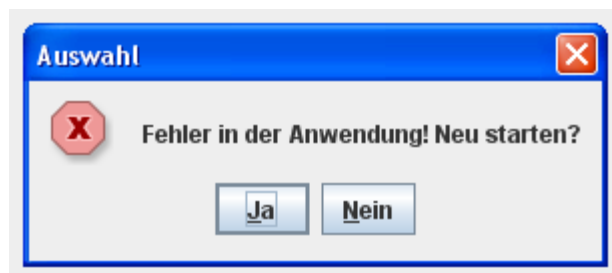
JA oder OK: '**J**'

NEIN : '**N**'

ABBRUCH oder Fenster schliesen: '**A**'

noch nicht aufgerufen: ' ' (Leerzeichen)

D_Bestaetigung
titel: String
meldung: String
optionsType: int
meldungsTyp: int
ergebnis: char
D_Bestaetigung(...)
zeigeMeldung(...)
leseErgebnis(...)
setzeTitel(...)
setzeMeldungstext(...)
fehlermeldung(...)
hinweis(...)
warnung(...)
frage(...)
ohneIcon(...)
typJaNein(...)
typOkAbbruch(...)
typJaNeinAbbruch(...)



Der Eingabedialog

setzeTitel und **setzeMeldungstext**
setzen die entsprechenden Texte

Die Methoden

hinweis()

warnung()

frage()

ohneIcon()

setzen die Icons des Meldungsfensters.

Die Methode **zeigeMeldung()** startet den Dialog.

Das Ergebnis des Meldungsfensters

```
public String leseErgebnis()
```

Ergebnis ist die Zeichenkette. Sie wird nach dem Lesen geleert

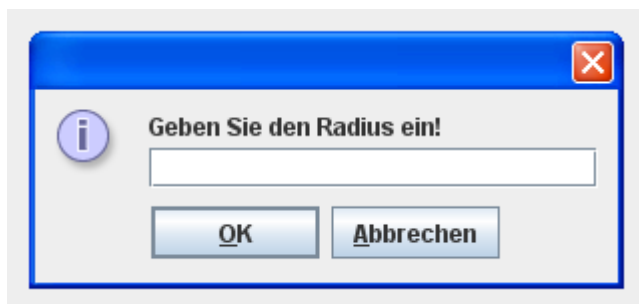
Die beiden folgenden Methoden übersetzen die Zeichenkette in einen int- bzw. double-Typ. Falls dabei ein **Fehler** auftritt, wird stattdessen der Wert **def** zurückgegeben.

```
public int leseInteger(int def)
```

```
public int leseIntegerGerundet(int def)
```

```
public double leseDouble(double def)
```

D_Eingabe
titel: String
meldung: String
meldungsTyp: int
ergebnis: String
D_Eingabe(...)
zeigeMeldung(...)
leseErgebnis(...)
setzeTitel(...)
setzeMeldungstext(...)
fehlermeldung(...)
hinweis(...)
warnung(...)
frage(...)
ohneIcon(...)



Erstellung eigener Dialoge

Ein Dialogfenster ist ein eigenes Programmfenster, das parallel zu Hauptfenster erscheint.

Es gibt zwei Arten von Dialogen:

Der modale Dialog blockiert die Eingabe zum Hauptfenster bis er geschlossen ist.

Der nicht modale Dialog ist ein Fenster, das über dem Hauptprogramm schwebt. Er ist ein Fenster, das parallel zum Hauptfenster erscheint. Es kann alles, was auch das Hauptfenster kann.

Die wichtigste Aufgabe: Die **Kommunikation** der beiden Fenster.

Der modale Dialog

Die Eingabe des Hauptfensters ist während der Sichtbarkeit des modalen Dialogs blockiert.

Der Status des Dialogfelds kann vom Hauptfenster aus erst nach dem Schließen des Dialogfensters des Dialogs abgefragt werden.

Der Dialog ist ein Objekt, das nach Beenden des Dialogs existiert. Die Elemente des Dialogs können abgefragt werden.

Der nicht modale Dialog

Die Eingabe auf das Hauptfenster ist parallel zum Dialog möglich.

Startet man den **nicht modalen Dialog**, so wird das Fenster des nicht modalen Dialogs sichtbar. Dann läuft das Hauptprogramm weiter.

Damit beide Fenster aufeinander wirken können, muss eine Kommunikationsmöglichkeit zwischen beiden installiert sein.

Kommunikation

Hauptfenster → DialogObjekt.

Das Hauptfenster hat aus der Erzeugung des Dialogfensters eine Referenz auf das Dialogobjekt. Darüber können Methoden des Dialogobjekts aufgerufen werden und Attribute des Dialogobjekts gelesen/verändert werden.

Dialogobjekt → Hauptfenster (auch beim modalen Dialog möglich)

Dem Dialogobjekt wird das Hauptfenster bekannt gemacht. Das kann z:b: eine Referenz auf das Hauptfenster sein. Dadurch kann dann auch das Dialogobjekt Methoden des Hauptfensters verwenden.

Auch beim modalen Dialog kann der Dialog die Kommunikationsformen des nicht modalen Dialogs verwenden! Der modale Dialog blockiert die Eingabe des Hauptfensters, das Programm läuft natürlich weiter!

Die Kommunikationsform der Toolbox

Als Spezialfall kann die Kommunikationsform der Toolbox installiert werden.

Die Dialogklasse enthält dann das folgende Code-Fragment:

```
ITuWas linkObj;
int id = 0; // ID der Komponente für Callback

public void setzeLink(ITuWas linkObj, int ID) {
    this.linkObj = linkObj;
    id = ID;
}
```

In der `public void tuWas(int ID)`, die von den aktiven Komponenten des Dialogs aufgerufen wird, wird dann die `tuWas`-Methode des Hauptprogramms aufgerufen. Die dem Hauptprogramm übergebene ID kann man selbst bestimmen.

```
if (linkObj != null)
    linkObj.tuWas(id);
```

Dieses Vorgehen ist dann zwingend, wenn die Dialogklasse „universell“ sein soll, also nichts von der Klasse des aufrufenden Objekts weiß.

Dialogbeispiele

Die Beispiele sind recht einfach gehalten.

Aspekte wie Datenkapselung können natürlich umgesetzt werden.

Beispiel für einen modalen Dialog

Das Hauptfenster hat ein Ausgabepanel, das durch einen Dialog mit Eingabefeld verändert wird.

Das Hauptprogramm:

```
public class B_Dialog_Modal_Ausgabe implements ITuWas{
```

Das Dialog-Objekt

```
    B_Dialog_Modal dialogModal;
    // Dass Ausgabepanel des Hauptfensters
    AusgabePanel anzeige ;
    Taste taste;

    /**
     * Konstruktor
     */
    public B_Dialog_Modal_Ausgabe () {

        dialogModal = new B_Dialog_Modal();
        anzeige = new AusgabePanel("Ausgabe",100,100,300,100);
        taste = new Taste("Dialog", 200 , 250 , 100 , 50 );
        taste.setzeLink(this, 0);
    }
```

Diese Taste startet den Dialog

```
    public void tuWas(int ID) {
        dialogModal.setzeSichtbar();
        // Nach Beenden des Dialogs weiter:
        if (dialogModal.dialog leseCloseID() == 1 ) {
            anzeige.setzeAnzeigetext(dialogModal.eingabe leseText());
        }
    }
```

Start des Dialogs

Abfrage auf ID der
Übernehmen-Taste

```
    public static void main(String[] args) {
        B_Dialog_Ausgabe t = new B_Dialog_Ausgabe();
    }
}
```

Lesen des Eingabefelds

Der modale Dialog

```

public class B_Dialog_Modal {

    // Dialog
    public D_JGUIDialog    dialog;
    // Container der Dialog-Komponente
    private IContainer     behaelter;

    Eingabefeld             eingabe ;
    private Taste           ja ;
    private Taste           nein ;

    public B_Dialog_Modal() {
        // Dialog anlegen
        // Ein modaler Dialog blockiert andere Fenster des Programms !
        // Dialoge werden verborgen, nicht geschlossen!
        dialog = new D_JGUIDialog("Einstellungen",
                                   D_JGUIDialog.MODAL );

        // Behälter für Dialogkomponenten lesen
        behaelter = dialog leseContainer();

        eingabe = new Eingabefeld(behaelter, "....", 0, 0, 400, 80);

        // Elemente in den Dialog-Behälter einfügen
        ja = new Taste(behaelter, "Übernehmen", 50, 100, 150, 50);
        // Die Taste erhält als Link die Dialog-Komponente
        // Die Taste schließt dadurch den Dialog
        ja.setzeLink(dialog, 1);

        nein = new Taste(behaelter, "Abbruch", 200, 100, 150, 50);
        // Die Taste erhält als Link die Dialog-Komponente
        // Die Taste schließt dadurch den Dialog
        nein.setzeLink(dialog, 0);

        // Größe des Dialogfensters einstellen
        dialog.setSize(400, 300);
    }

    /**
     * Der Dialog wird sichtbar.
     * Der modale Dialog blockiert die Eingabe des Hauptprogramms
     */
    public void setzeSichtbar() {
        // Dialog anzeigen
        dialog.setVisible(true);
    }
}

```

Das Dialog-Fenster

Der Behälter
des Dialog-FenstersDie Elemente
des Dialogs

Modal!

Beispiel für einen nicht modalen Dialog:

Das Hauptfenster hat ein Ausgabepanel, das durch einen Dialog mit Eingabefeld verändert wird.

Das Hauptprogramm:

Das Dialog-Objekt

```
public class B_Dialog_NichtModal_Ausgabe implements ITuWas {
```

```
    B_Dialog_NichtModal    dialogNichtModal;
```

```
    AusgabePanel           anzeige;
```

```
    Taste                  taste; _____
```

Diese Taste startet den Dialog

```
/**
```

```
 * Konstruktor
```

```
 */
```

```
public B_Dialog_NichtModal_Ausgabe() {
```

```
    dialogNichtModal = new B_Dialog_NichtModal();
```

```
// Verbindung zwischen Dialog und Hauptfenster
```

```
    dialogNichtModal.hauptfenster = this ;
```

```
    anzeige = new AusgabePanel("Ausgabe", 100, 100, 300, 100);
```

```
    taste = new Taste("Dialog", 200, 250, 100, 50);
```

```
    taste.setzeLink(this, 0);
```

```
}
```

```
public void tuWas(int ID) {
```

```
    // Die Methode beendet sofort nach dem Start des Dialogs!
```

```
    dialogNichtModal.setzeSichtbar();
```

```
    // nichts weiter zu tun!
```

```
}
```

```
public static void main(String[] args) {
```

```
    B_Dialog_NichtModal_Ausgabe t =
```

```
        new B_Dialog_NichtModal_Ausgabe();
```

```
}
```

```
}
```

Start des Dialogs

Der nicht modale Dialog

```

public class B_Dialog_NichtModal implements ITuWas {
    // Dialog
    public D_JGUIDialog    dialog;
    // Container der Dialog-Komponente
    private IContainer     behaelter;

    Eingabefeld            eingabe ;
    private Taste          ende ;

    public B_Dialog_NichtModal() {
        // Dialog anlegen
        // Ein nicht modaler Dialog blockiert nicht !
        // Dialoge werden verborgen, nicht geschlossen!
        dialog =
            new D_JGUIDialog("Einstellungen",
                            D_JGUIDialog.NICHTMODAL );

        // Behälter für Dialogkomponenten lesen
        behaelter = dialog leseContainer();

        eingabe = new Eingabefeld(behaelter, "....", 0, 0, 400, 80);
        // Callback bei >>Enter<<
        eingabe.setzeLink(this, 0);

        // Elemente in den Dialog-Behälter einfügen
        ende = new Taste(behaelter, "Verbergen", 50, 100, 150, 50);
// Die Taste erhält als Link die Dialog-Komponente
// Die Taste verbirgt dadurch den Dialog
        ende.setzeLink(dialog, 1);

        // Größe des Dialogfensters einstellen
        dialog.setSize(400, 300);
    }
    /**
     * Der Dialog wird sichtbar.
     */
    public void setzeSichtbar() {
        // Dialog anzeigen
        dialog.setVisible(true);
    }
// Verbindung zwischen Dialog und Hauptfenster
    B_Dialog_NichtModal_Ausgabe hauptfenster ;
    public void tuWas(int ID) {
        // Auch bei der Erzeugung des Dialogs können Ereignisse auftreten!
        if ( hauptfenster != null) {
            hauptfenster.anzeige.
                setzeAnzeigetext(eingabe leseText());
        }
    }
}

```

Das Grundgerüst der Toolbox

Die Klassen **Zeichnung** und **StaticTools** bilden die Basis der JGUIToolbox.

Die Klasse **Zeichnung** bildet das Programmfenster der Anwendung.

Die Klasse **StaticTools** beinhaltet statische Methoden, die man an vielen Stellen einsetzen kann.

Die Klasse Zeichnung

Ein Programm hat genau ein Fenster.

**Diese Fenster wird automatisch erzeugt,
wenn eine Komponente der Toolbox angelegt wird.**

Die Klasse **Zeichnung** hat statische Methoden, die das einzige von dieser Klasse angelegte Objekt beeinflussen:

setzeFenstergroesse

```
public static void setzeFenstergroesse(int breite, int hoehe)
```

Raster

Beim Hauptfenster kann ein Raster ein- und ausgeschaltet werden. Folgende Methoden beeinflussen das Raster

setzeRasterEin

```
public static void setzeRasterEin()
```

setzeRasterAus

```
public static void setzeRasterAus()
```

setzeDeltaX

```
public static void setzeDeltaX(int deltaX). Default: 100
```

setzeDeltaY

```
public static void setzeDeltaY(int deltaY). Default: 100
```

Die Klasse StaticTools

Die Klasse StaticTools beinhaltet einige statisch Methoden, die an vielen Stellen verwendet werden.

getColor

public static java.awt.Color **getColor**(java.lang.String farbname)

Gibt den Farbwert (Color) für Java zurück. Gültige Parameter sind "rot", "gelb", "blau", "gruen", "lila", "schwarz", "weiss", "grau", "pink", "magenta", "orange", "cyan", "hellgrau"

jetzt_Minute

public static int **jetzt_Minute**()

liefert die aktuelle Zeit

Returns: Minute in der aktuellen Stunde

jetzt_Sekunde

public static int **jetzt_Sekunde**()

liefert die aktuelle Zeit

Returns: Sekunde in der aktuellen Minute

jetzt_Stunde

public static int **jetzt_Stunde**()

liefert die aktuelle Zeit

Returns: Stunde

jetzt

public static long **jetzt**()

liefert die aktuelle Tageszeit

Returns: Tageszeit in Sekunden

warte

public static void **warte**(int millisekunden)

Warte für die angegebenen Millisekunden. Mit dieser Operation wird eine Verzögerung definiert, die für animierte Zeichnungen benutzt werden kann.

Parameter: millisekunden - die zu wartenden Millisekunden

Anhang A: Programmiersysteme

Die Aufzählung ist unvollständig und zeigt nur meine Favoriten.

Die Beschreibung der Eigenschaften soll nur den Teil hervorheben, der für mich diese Programme von anderen abhebt.

Da alle diese Systeme Java-Code erzeugen, ist es für den Experten ein Leichtes, die verschiedenen Stärken der System parallel auszuschöpfen.

BlueJ: Quelle: <http://www.bluej.org/>

Mein favorisiertes Programmiersystem für die Arbeit mit Schülern.

Herausragende Eigenschaften: „**Objects First**“

Javaeditor : Quelle: <http://lernen.bildung.hessen.de/informatik/javaeditor/index.htm>

Das Programm erspart dem Anwender viele Mühe, weil es gerade beim Arbeiten mit GUI-Elementen durch automatisches Einfügen viel Arbeit abnimmt.

Dazu kommt eine hervorragende Integration von Hilfe-Systemen in das Programm. Von der IDE aus erhält man Zugriff auf die Java-API und das „Handbuch der Java-Programmierung“ von Guido Krüger und Thomas Stark

Neben den Hilfen bei der GUI-Programmierung sind für mich besonders erwähnenswert:

Ein **UML-Editor**, der sowohl bei der Erzeugung von Java-Code aus dem Klassendiagramm wie auch dem Erzeugen von Klassendiagrammen aus Java-Code hervorragende Ergebnisse liefert.

Eclipse : Quelle: <http://www.eclipse.org/>

Mein favorisiertes Programmierwerkzeug.

Für GUI-Programmierung: Das Visual Editor Project

Quelle: <http://www.eclipse.org/vep/WebContent/main.php>

Netbeans : Quelle: <http://www.netbeans.org/>

Eine alternative große IDE für Java.

In Netbeans kann ein Plugin (BlueJ Project Support) integriert werden, das BlueJ-Projekte direkt bearbeiten kann. Man kann dadurch die bei BlueJ vermissten Editierfähigkeiten über Netbeans ergänzen. Editieren in BlueJ. Parallel dazu auf den gleichen Dateien(!) arbeiten mit BlueJ.

Anhang B: Für mich hilfreiche (elektronische) Literatur

Die Reihenfolge ist ohne Wertung. Ich verwende alle parallel. Jedes von Ihnen hat seine eigenen Stärken. In der Zusammenschau wird vieles klarer.

Die Dokumentationen und Tutorials von Sun (in Englisch)

<http://java.sun.com/docs/books/tutorial>.

Guido Krüger-Thomas Stark: Handbuch der Java-Programmierung

<http://www.javabuch.de>

Christian Ullenboom: Java ist auch eine Insel

http://download.galileo-press.de/openbook/javainsel7/galileocomputing_javainsel7.zip

Stefan Middendorf, Reiner Singer, Jörn Heid :

Java Programmierhandbuch und Referenz für die JavaTM-2-Plattform, Standard Edition

<http://www.dpunkt.de/java/index.html>

Anhang C: Der Aufbau der Toolbox

Dieser Teil ist für den Programmierer, der neue Komponenten für die Toolbox erzeugen möchte oder auch nur den Aufbau der Toolbox nachvollziehen möchte.

Die Toolbox erzeugt ein Objekt vom Typ **Zeichnung**. **Zeichnung** ist von **JFrame** abgeleitet.

Genau ein Objekt dieser Klasse **Zeichnung** wird von den Komponenten erzeugt, wenn es noch nicht existiert.

Dieses **Zeichnung**-Objekt hat ein Objekt vom Typ **Zeichenflaeche**.

Dieses wird bei der Erzeugung des **Zeichnung**-Objekts erzeugt.

Die Klasse **Komponente** implementiert das Interface **Icontainer**.

Es gibt **genau ein Objekt vom Typ Zeichnung und vom Typ Zeichenfläche**.

Ruft eine Komponente die statische Methode **Zeichnung.gibZeichenflaeche** auf. So werden beide erzeugt, wenn sie noch nicht existieren..

Zu dieser **Zeichenfläche** (Klassenname für Objektname) werden die Komponenten hinzugefügt.

Die Komponenten der Toolbox sind von der abstrakten Klasse **BasisKomponente** abgeleitet.

BasisKomponente ist wiederum von **JPanel** abgeleitet.

BasisKomponente hat eine abstrakte Methode

```
public abstract void paintComponentSpezial (Graphics g);
```

Sie wird von **paintComponent(Graphics g)** aufgerufen. Hier stehen die speziellen Zeichenbefehle der Komponente.

Eine **aktive Komponente** (Komponente mit Callback) enthält eine Methode

```
public void setzeLink (ITuWas linkObj, int ID)
```

Wir der Komponente eine Klasse übergeben, die das Interface **ITuWas** implementiert, so wird beim Eintreten des Komponenten-Ereignisses die durch das Interface garantierte Methode **tuWas(int ID)** des übergebenen Objekts aufgerufen.

Soll die Komponente eine Standardkomponente von Java wie z.B. ein **Button** sein, so wird diese Komponente der **BasisKomponente** hinzugefügt.

Die **BasisKomponente** erhält dann das **Border-Layout**. Dadurch passt sich die Standardkomponente der Größe der **Basiskomponente** an.

Versteckspielen:

Mit Ausnahme der **FreiZeichen**-Komponente werden die anderen **Swing**-Komponenten vom Benutzer verborgen.

Wer ein Objekt der **FreiZeichnen**-Komponente im Objektinspektor untersucht wird von einer Fülle von Methoden überwältigt. **Basiskomponente** ist von **JPanel** abgeleitet und erbt daher alle Methoden von **JPanel** und das sind nicht wenige.

Für jede Komponente wird daher eine (von Objekt abgeleitete) Komponente (**Xxxxxxx**) erzeugt, die für den Außenstehenden sichtbar ist (**public**). Die eigentliche von

Basiskomponente abgeleitete **Graphikkomponente** (**C Xxxxxxx**) liegt im gleichen Quellfile (**Xxxxxxx.java**) **BlueJ** zeigt dann in seiner Klassenübersicht nur die **public**-Klasse **Xxxxxxx**.

Das gleiche Versteckspiel geschieht im übrigen in der Datei **Zeichnen.java**.

Sie enthält den Quelltext für die **public-Klasse Zeichnen**. Dazu noch den Quelltext für die **Klasse Zeichenfläche** und für die **Interface ITuWas** sowie **Interface IContainer**.

Natürlich würde es sich anbieten, die umschließenden Klassen von einer gemeinsamen Klasse abzuleiten. Aus obigen Gründen wurde das unterlassen.

Da die eingebundenen CXXX... - Klassen dort die Bezeichnung **obj** erhalten haben, kann man die meisten Methoden durch kopieren übernehmen.

Will man die Toolbox neu strukturieren und einige Teile in ein Package umgruppieren, so kann es nötig sein, dieses Versteckspiel aufzuheben.

In einer Quelldatei kann nur eine Klasse **public** sein, die Klasse, deren Namen der Name der Datei ist (!)

Zum Schluss:

Ich habe diese Toolbox für mich geschaffen. Ich habe gesehen, dass sie auch anderen nützlich sein kann.

Jede / jeder ist eingeladen, die Toolbox zu verwenden.

Ich würde mich freuen, wenn sie möglichst Vielen hilfreich ist.

Jede/Jeder ist dazu eingeladen, die Toolbox zu Verbessern und/oder zu/Ergänzen.

Ich würde mich freuen, wenn auch ich diese Änderungen/Ergänzungen erhalten könnte. Ich werde sie nach Möglichkeit in die Toolbox einarbeiten.

Ich würde mich freuen, wenn ich gelungene Anwendungen oder Erweiterungen der Toolbox erhalten würde.

Sollte die Toolbox in einer kommerziellen Umgebung eingesetzt werden, so erwarte ich eine Mitteilung.

Ich habe mich bemüht, kann aber keine Garantie dafür übernehmen, dass diese Komponenten in jedem Fall das tun, was man von ihnen erwartet. Sie werden so weitergegeben, „wie sie sind“. Mitteilungen über Probleme helfen bei ihrer Beseitigung.

Ich habe bei der Erstellung der Toolbox nur auf freie Informationsquellen zurückgegriffen.

Die in der Toolbox angesprochenen Markennamen gehören ihrer Eigentümer.

Sollte ich trotzdem Rechte anderer verletzt haben, bitte ich um einen Hinweis. Ich werde mich bemühen, das schnellstens zu bereinigen.

Hans Witt im August 2008