# Rahmenprogramm

Ein Basisprogramm in Java besteht aus mindestens den folgenden Komponenten:

1. Programm (Klasse)
2. Methoden
3. Variablen

Wir arbeiten von Anfang an mit einem Rahmenprogramm, dass uns eine Grafikausgabe erlaubt:

**JFrame**

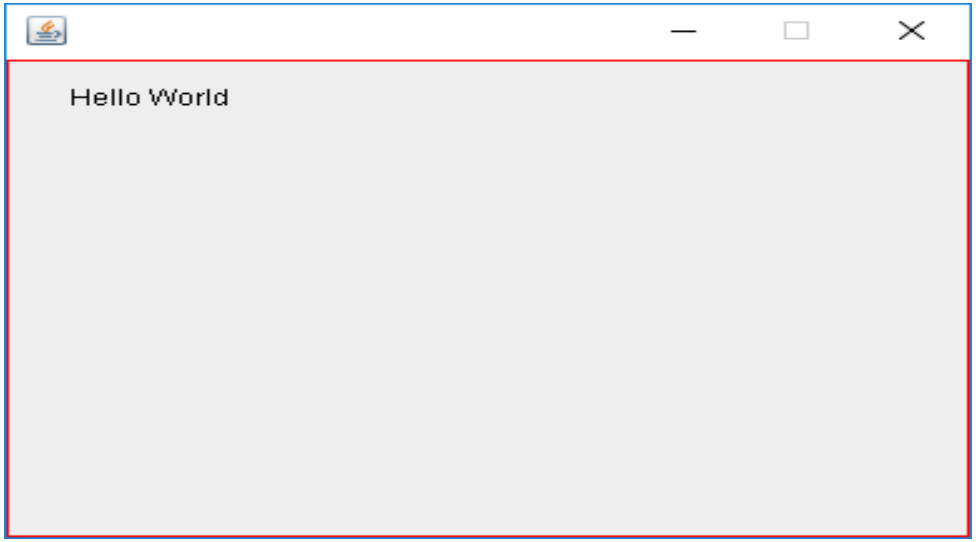
```
public class HelloWorldFrame extends JFrame {
  HelloWorldPanel view = new HelloWorldPanel();

  public HelloWorldFrame() {
    setTitle("HelloWorldFrame");
    add(view);
    setSize(400, 300);
    view.init();
  }

  public static void main(String args[]) {
    HelloWorldFrame frame = new HelloWorldFrame();
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(1);
      }
    });
    frame.setResizable(true);
    frame.setVisible(true);
  }
}
```

Obiger Code erzeugen den Rahmen (das Fenster), in dem dann ein Panel eingebettet wird.

Das eigentliche Programm ist im Panel zu finden:

**JPanel**

```
class HelloWorldPanel extends JPanel {

  public void init() {
    System.out.println("init()");
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawString("Hello World", 100, 50);
  }
}
```




Unser Fokus liegt auf dem Programm resp. auf der Klasse der Art JPanel, das vorweg über die beiden Methoden `init()` und `paintComponent(Graphics g)` verfügt:

- `init()` wird beim Aufstarten exakt einmal ausgeführt.
- `paintComponent()` wird so häufig ausgeführt, wie neue gezeichnet werden muss aufgerufen.

## Primitive Data Types

| Daten Typ | Grösse [Bit] | Wertebereich |
|---|---|---|
| *byte* | 8 | -128 ... 127 |
| *short* | 16 | -32'768 … 32’767 |
| *int* | 32 | -2’147’483’648 ... 2’147’483’647 |
| *long* | 64 | -9’223’372’036’854’775’808 ... 9’223’372’036’854’775’807 |
| *float* | 32 | -3.4e+38 ... 3.4e+38 |
| *double* | 64 | -1.7e+308 … 1.7e+308 |
| *char* | 16 | Kompletter Unicode Charakter Satz |
| *boolean* | 1 | true, false |

Alle Ganzzahlen sind im Zweierkomplement, alle Fliesskommazahlen im Format gemässe IEEE 754 abgelegt.

## Java Operators

| Operator Typ | Operatoren |
|---|---|
| Arithmetisch | +, – , *, ? , % |
| Zuweisung | =, +=, -=, *=, /=, %= |
| Logisch | &&, \|\| |
| Vergleichend | <, >, <=, >=,==, != |
| Unitär | **++x, --x, x++, x--, !, ~,-,+** |

## Java Variablen

In Java gibt es drei Typen von Variablen:

1. Lokale Variablen (leben in Methoden)
2. Attribute (leben in Klassen/Objekten)
3. Statische Variablen

Syntax:

{public | protected | private} [static] type name [= expression | value];

Beispiele: private double d = quadrat(20.0); int x = 25;

## Java Methoden

Eine Methode ist ein Satz von Code, der gruppiert wird, um eine bestimmte Operation auszuführen. Wir unterscheiden zwischen

1. Deklaration der Methode
2. Aufruf der Methode

Syntax:

{public | protected | private} [static] {type | void} name (arg1, ..., argN ){statements}

Beispiel:

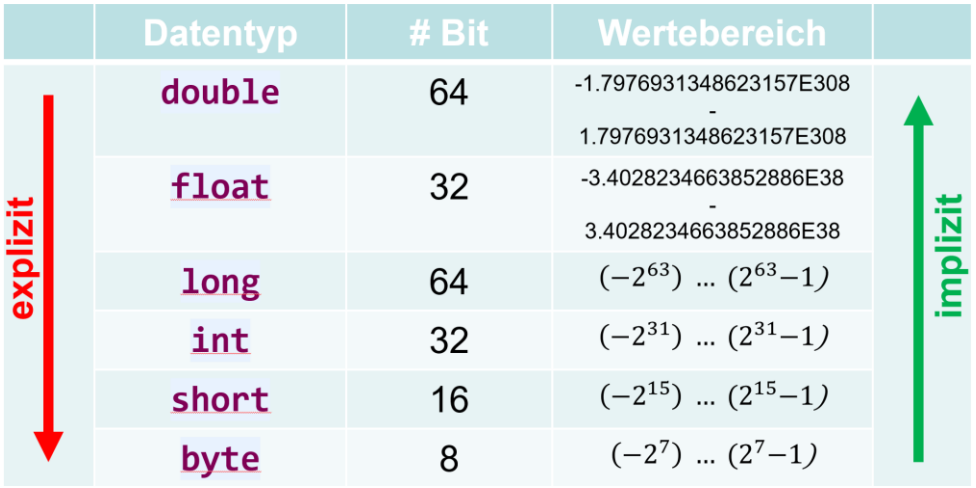
```
public double quadrat(double x){
  return x*x;
}

double d = quadrat(5.0);
```

# Datentyp Konversion

Die Änderung eines Wertes von einem Datentyp in einen anderen Typ wird als Datentypkonvertierung oder Type-Casting bezeichnet. Bei der Datentypkonvertierung gibt es zwei Arten:

1. Ein Datentyp mit geringerem Wertebereich wird in einen Datentyp mit grösserem Wertebereich umgewandelt.
2. Ein Datentyp mit grösserem Wertebereich wird in einen Datentyp mit geringerem Wertebereich umgewandelt.

Die erste Datentypkonvertierung kann implizit erfolgen, die zweite muss explizit erfolgen.




# Schleifen

Schleifen werden verwendet, wenn eine Reihe von Anweisungen wiederholt werden müssen, bis die Bedingung für die Beendigung erfüllt ist.

```
// for Schleife
for (initialization; condition; increment) {expression}

// for each Schleife
for (int i: someArray) {}

// while loop
while (condition) {expression}

// do while loop
do {expression} while(condition)
```

Beispiele sind unter Pseudo-Code zu finden.

# Entscheidungen

Entscheidungen werden verwendet, wenn während der Ausführung des Programms Entscheide notwendig sind:

```
//if statement
if (condition) {expression}

//if-else statement
if (condition) {expression} else {expression}

//switch statement
switch (var)
{ case 1: expression; break; default: expression; break; }
```

Beispiele sind unter Pseudo-Code zu finden.

# Eindimensionale (1-D) Arrays

Eindimensionales oder 1-D-Array ist eine Art lineares Array, in dem Elemente in einer kontinuierlichen Reihe gespeichert werden.

```
// Deklaration
type[] varName= new type[size];

// Oder
type[] varName= {values1, value2,...};

// Oder
type[] varName= new type[]{values1, value2,...};

// Oder
g.drawPolygon(new int[] {24, 48, 36}, new int[] {50, 50, 10}, 3);
```

```
// Zugriff auf Elemente:
varName[index]

// Zugriff auf die Länge des Arrays:
varName.length
```

## Zeichenketten erzeugen

String in Java ist ein Objekt, das eine Folge von Zeichen darstellt. Ein String kann auf zwei Arten erstellt werden:

```
String str1 = "Welcome"; // Using literal
String str2 = new String("Java"); // Using new keyword
```

## Pseudo-Code

Mit Pseudo-Code lässt sich eine Abfolge von Befehlen sprachunabhängig formulieren. Pseudo-Code kann insbesondere bei der Entwicklung und Programmierung von Algorithmen wertvoll sein. Dabei wird in fünf Schritten verfahren:

1. Der Algorithmus wird in Prosatext formuliert.
2. Aufgrund des Prosatextes wird von Hand ein Beispiel vernünftiger Grösse erstellt.
3. Der zum Handbeispiel passende Code wird als Pseudo-Code entworfen.
4. Der Pseudo-Code wird in der verwendeten Sprache ausprogrammiert.
5. Der programmierte Algorithmus wird gegen das Handbeispiel getestet.

```
/**
 * TODO: while - Schleife
 *
 * <pre>
 * - Lokale Ganzzahlen n gleich Null und x = 10 erzeugen.
 * - Solange n kleiner 8:
 *   - Sternchen an der Stelle x, 20 zeichnen.
 *   - x gleich x plus 10 setzen.
 *   - n inkrementieren.
 * </pre>
 */
int n = 0, x = 10;
while (n < 8) {
  g.drawString("*", x, 20);
  x = x + 10;
  n++;
}

/**
 * TODO: for - Schleife
 *
 * <pre>
 * - Lokale Ganzzahl y = 10 erzeugen.
 * - Für m gleich Null bis kleiner 8:
 *   - Sternchen an der Stelle y, 40 zeichnen.
 *   - y gleich y plus 10 setzen.
 * </pre>
 */

int y = 10;
for (int m = 0; m < 8; m++) {
  g.drawString("*", y, 40);
  y = y + 10;
}

/**
 * TODO: if-Anweisung
 *
 * <pre>
 * - Lokale Ganzzahl alter gleich 19 erzeugen.
 * - Falls alter grösser 18:
 *   - Zeichenkette "Sie dürfen wählen!" an der Stelle 100,
 *     100 zeichnen.
 * </pre>
 */
int alter = 19;
if (alter > 18) {
  g.drawString("Sie dürfen wählen!", 100, 100);
}


/**
 * TODO: if-else-Anweisung
```

```
 *
 * <pre>
 * - Falls alter grösser 18:
 *   - Zeichenkette "Sie dürfen wählen!" an der Stelle 200,
 *     200 zeichnen.
 * - Sonst
 *   - Zeichenkette "Sorry, Sie dürfen nicht rein!" an der
 *     Stelle 200, 200 zeichnen.
 * </pre>
 */
if (alter > 18) {
  g.drawString("Sie dürfen rein.", 200, 200);
} else {
  g.drawString("Sorry, Sie dürfen nicht rein!", 200, 200);
}
```

## Grundlegende Mathe-Funktionen

| Methode | Beschreibung |
|---|---|
| Math.abs() | Es wird der Absolutwert des angegebenen Wertes zurückgegeben. |
| Math.max() | Es wird der größte von zwei Werten zurückgegeben. |
| Math.min() | Es wird der kleinste von zwei Werten zurückgegeben. |
| Math.round() | Wird verwendet, um die Dezimalzahlen auf den nächstgelegenen Wert zu runden. |
| Math.sqrt() | Wird verwendet, um die Quadratwurzel einer Zahl zurückzugeben. |
| Math.cbrt() | Wird verwendet, um die Cube-Root einer Zahl zurückzugeben. |
| Math.pow() | Gibt den Wert des ersten Arguments hoch das zweite Argument zurück. |
| Math.signum() | Wird verwendet, um das Vorzeichen einer Zahl zu erhalten. |
| Math.ceil() | Wird verwendet, um den kleinsten ganzzahligen Wert zu finden, der größer oder gleich dem Argument ist. |
| Math.floor() | Wird verwendet, um den größten ganzzahligen Wert zu finden, der kleiner oder gleich dem Argument ist. |
| Math.random() | Gibt eine Pseudozufallszahl im Intervall Null exklusive Eins zurück. |

## Logarithmische Mathe-Funktionen

| Methode | Beschreibung |
|---|---|
| Math.log() | Gibt den natürlichen Logarithmus eines double Wertes zurück. |

| | |
|---|---|
| Math.log10() | Gibt den Logarithmus der Basis 10 eines double Wertes zurück. |
| Math.exp() | Gibt die Eulersche Zahl hoch dem Argument zurück. |

## Trigonometrische Mathe-Funktionen

| Methode | Beschreibung |
|---|---|
| Math.sin() | Gibt den Sinus des Arguments zurück. |
| Math.cos() | Gibt den Kosinus des Arguments zurück. |
| Math.tan() | Gibt den Tangens des Arguments zurück. |
| Math.asin() | Gibt den Arkussinus des Arguments zurück. |
| Math.acos() | Gibt den Arkuskosinus des Arguments zurück. |
| Math.atan() | Gibt den Arkustangens des Arguments zurück. |

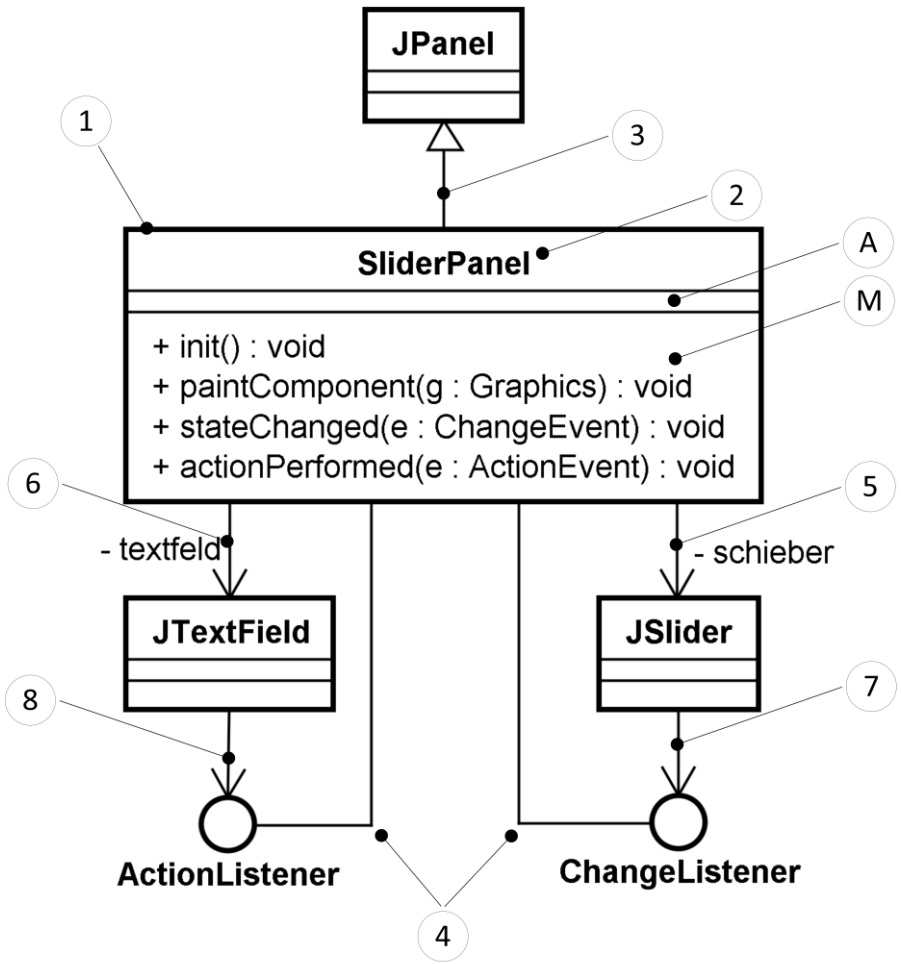
## Hyperbolische Mathe-Funktionen

| Methode | Beschreibung |
|---|---|
| Math.sinh() | Gibt den Sinus hyperbolicus des Arguments zurück. |
| Math.cosh() | Gibt den Kosinus hyperbolicus des Arguments zurück. |
| Math.tanh() | Gibt den Tangens hyperbolicus des Arguments zurück. |

## Winkelkonversion

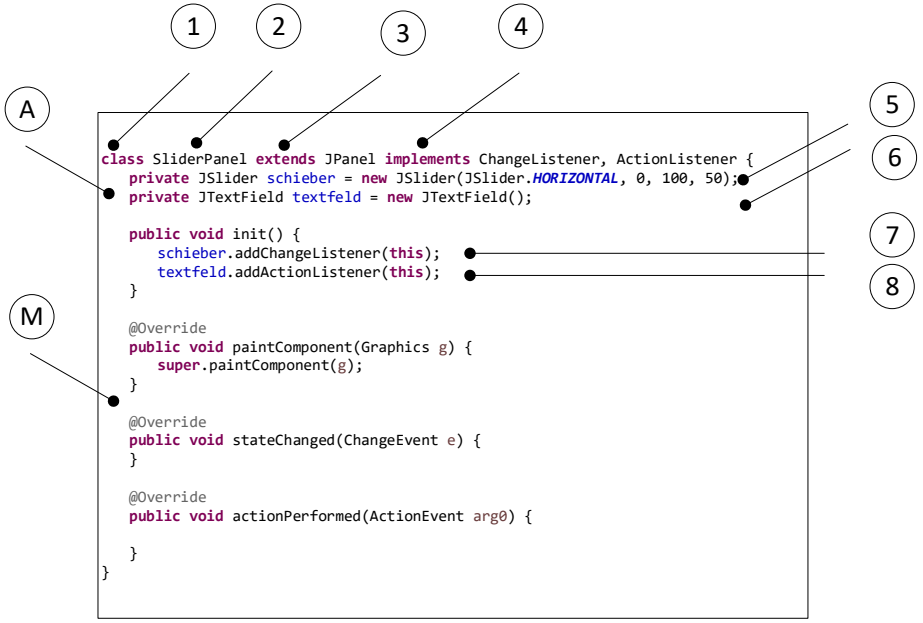
| Methode | Beschreibung |
|---|---|
| Math.toDegrees | Wird verwendet, um den angegebenen Radiuswinkel in einen äquivalenten Winkel, gemessen in Grad, umzuwandeln. |
| Math.toRadians | Wird verwendet, um den angegebenen Gradwinkel in einen äquivalenten Radiuswinkel umzuwandeln. |

# Klassendiagramme



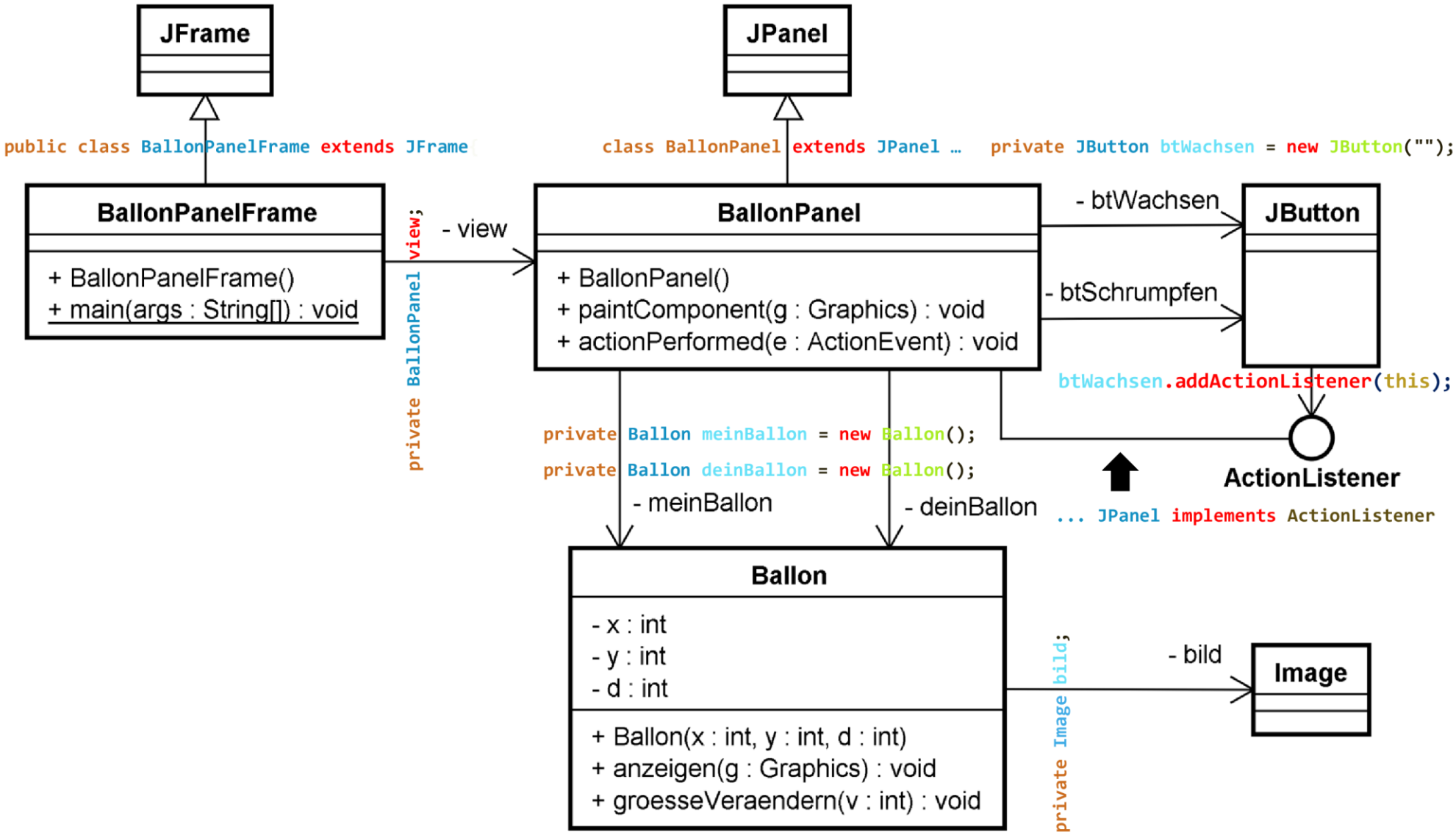

```
Zugriffs-Modifier
+  public / #   protected  / -  private
```

| | | |
|---|---|---|
| A | **Attribute** | Attribut – Kompartiment |
| M | **Methoden** | Methoden - Kompartiment. |




| Nr. | Keyword | Bedeutung / Phrasologie |
|---|---|---|
| 1 2 | **class** *SliderPanel* | *Deklaration* einer *Klasse* mit entsprechendem Namen. / *SliderPanel* ist eine Klasse. |
| 3 | **extends** *JPanel* | Erweitert *JPanel* / *SliderPanel* **ist ein** *JPanel.* |
| 4 | **implements** *ChangeListener*, *ActionListener* | Implementiert die Schnittstellen *ChangeListener* und *ActionListener* / *SliderPanel* **ist ein** ChangeListener und ein ActionListener. |
| 5 | **private JSlider** *schieber;* | *SliderPanel* **hat einen** privaten *JSlider* mit der Bezeichnung *schieber*. |
| 6 | **private JTextField** *textFeld;* | *SliderPanel* **hat ein** privates *JTextField* mit der Bezeichnung *textFeld*. |
| 7 | **schieber.**addChangeListener(**this**); | Registriert die Klasse *SliderPanel* als *ChangeListener* für den *JSlider schieber*. / *SliderPanel* ist *ChangeListener* für den *JSlider schieber*. |
| 8 | **textFeld.**addActionListener(**this**) | Registriert die Klasse *SliderPanel* als *ActionListener* für das *JTextField textFeld*. / *SliderPanel* ist *ActionListener* für das *JTextField textFeld*. |

**Beispiel Implementationsklassendiagramm**[1]




[1] With courtesy of Simon Bieli …

# Klassen und Objekte

Ein Klasse ist der Bauplan für Objekte gleicher Art. Durch die Instanziierung der Klasse entsteht ein Objekt und damit das zugehörige Abbild im Speicher des Computers.

**Objekte erzeugen**

Objekte werden mittels dem Key-Word `new` erzeugt:

```
private Ballon meinBallon = new Ballon(250, 150, 75);
```

und lässt sich in Gedanken in zwei Schritte übersetzen:

- Auf `new` hin wird der entsprechende Speicherplatz alloziert und die Attribute im Speicher erstellt und initialisiert.
- Mit `Ballon(250, 150, 75)` wird der passende Konstruktor der Klasse aufgerufen.

**Konstruktor**

Der Konstruktor dient eigens dem Erzeugen von Objekten. Er ist als spezielle Methode zu verstehen und trägt den gleichen Namen wie die Klassen. Der Konstruktor deklariert keinen Return-Type, da er per Definition die Referenz des neu erzeugten Objektes zurück gibt.

**Zugriff auf Elemente eines Objektes**

Jedes Objekt erhält bei der Erzeugung eine Referenz, resp. einen Hash-Code, der den effizienten Zugriff auf das Objekt ermöglich. Diese Referenz wir im entsprechenden Attribut (`meinBallon, deinBallon`) abgelegt.

Der Zugriff erfolgt mittels

`objektName`.methodenName() resp. `objektName`.attributName

und setzt voraus, dass die entsprechenden Elemente **`public`**[2] sind.

Beispiele:

```
meinBallon.anzeigen(g);
deinBallon.anzeigen(g);
```

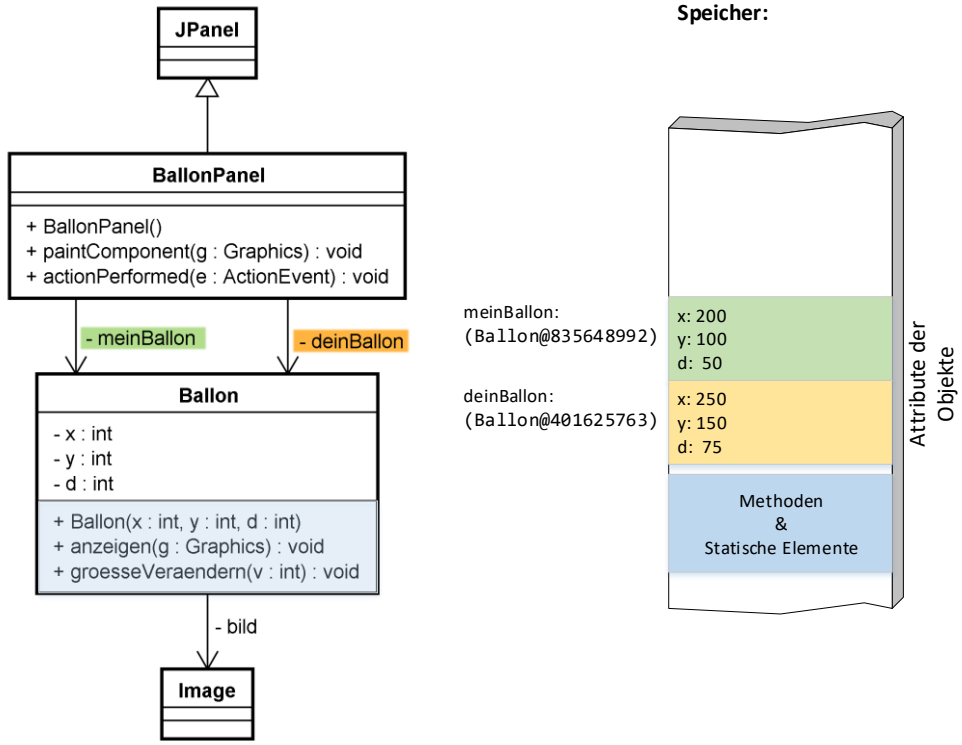



*Abbildung 1: Erzeugen eines Objektes*

### Benennung von Klassen, Objekten und Methoden

- Klassen beginnen mit einem Grossbuchstaben!
- Objekte, Methoden und Variablen mit einem Kleinbuchstaben!

[2] Die Zugriffs-Modifier package-wide und **`protected`** werden im Kontext Vererbung eingeführt.

### Schlüsselwort `this`

Das Schlüsselwort **`this`** meint das Objekt, das wir im Moment sehen, resp. in dem wir uns aktuell befinden.

Beispiel:

Wird die Methode anzeigen() der Klasse Ballon mittels `meinBallon.anzeigen(g)` aufgerufen, nimmt **`this`** innerhalb von anzeigen() den Wert der Referenz `meinBallon` an und verweist damit auf das Objekt mit Hash-Code `Ballon@835648992`, Wird die Methode mittels mittels `deinBallon.anzeigen(g)` aufgerufen, nimmt **`this`** innerhalb von anzeigen() den Wert der Referenz `deinBallon` und verweist damit auf das Objekt mit Hash-Code `Ballon@401625763`. Dadurch wird das Objekt mit den entsprechenden Attributen angezeigt!

### Vergleich von Objekten

Mit

```
if(meinBallon == deinBallon) {
  // ...
}
```

werden die Referenzen der beiden Objekte auf Gleichheit verglichen und nicht deren Eigenschaften. So man die Eigenschaften vergleichen will, muss die Methode `equals()` überschrieben werden, um Gleichheit der Objekte festzulegen.

Beispiel:

```
public boolean equals(Ballon ballon) {
    return (ballon.flaeche() == this.flaeche());
}
private double flaeche() {
  return Math.PI * Math.pow(d / 2.0, 2.0);
}

if(meinBallon.equals(deinBallon)) {
  // ...
}
```

### Überladen von Methoden

Eine Klasse kann mehrere Methoden mit gleichen Namen haben, die sich bezüglich Signatur, d.h. bezüglich formaler Parameterliste unterscheiden.

Beispiel:

```
public void groesseVeraendern() {
  d = d + 5;
}
public void groesseVeraendern(int v) {
  d = d + v;
}
```

Zur Signatur zählt nur die Art und Abfolge der Paramater, nicht aber der Return-Type!

### Schlüsselwort `static`

Mit dem Schlüssel ist es möglich Elemente einer Klasse und nicht Objekten zuzuordnen. Es gibt dann nur eine Kopie des Elementes, das direkt zur Klasse gehört. Im Gegensatz zu den Elementen der Objekte werden statische Elemente direkt mit

`Klasse.methodenName()` resp. `Klasse.attributName`

aufgerufen!

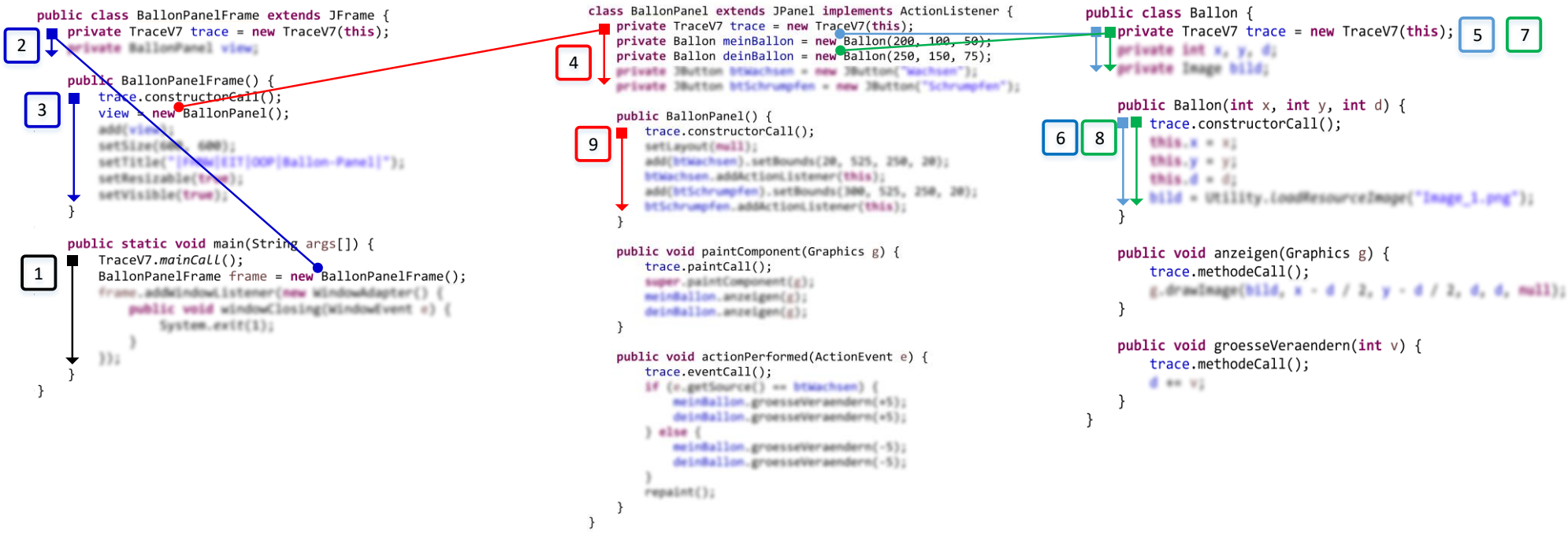
Beispiel:

```
class MyMath {
 public static final double bessel(int o, double a) {
  // hier käme die Berechnung der Besselfunktion
  return value;
 }
}
double d = MyMath.bessel(3, 3.78);
```

# Aufstarten der Applikation

Durch das Aufstarten der Applikation, wird die gemäss Klassendiagramm festgelegte Struktur gebaut indem entsprechende Objekte erzeugt und in gewünschter Weise miteinander verbunden werden. Anfangspunkt ist dabei immer die Methode main()! Anschliessend werden, allenfalls auch geschachtelt, Attribute initialisiert und Konstruktoren ausgeführt. Die entsprechende  Spur lässt sich mittels der Klasse TraceV7 auch in der Konsole beobachten.

Beispiel mit Code:

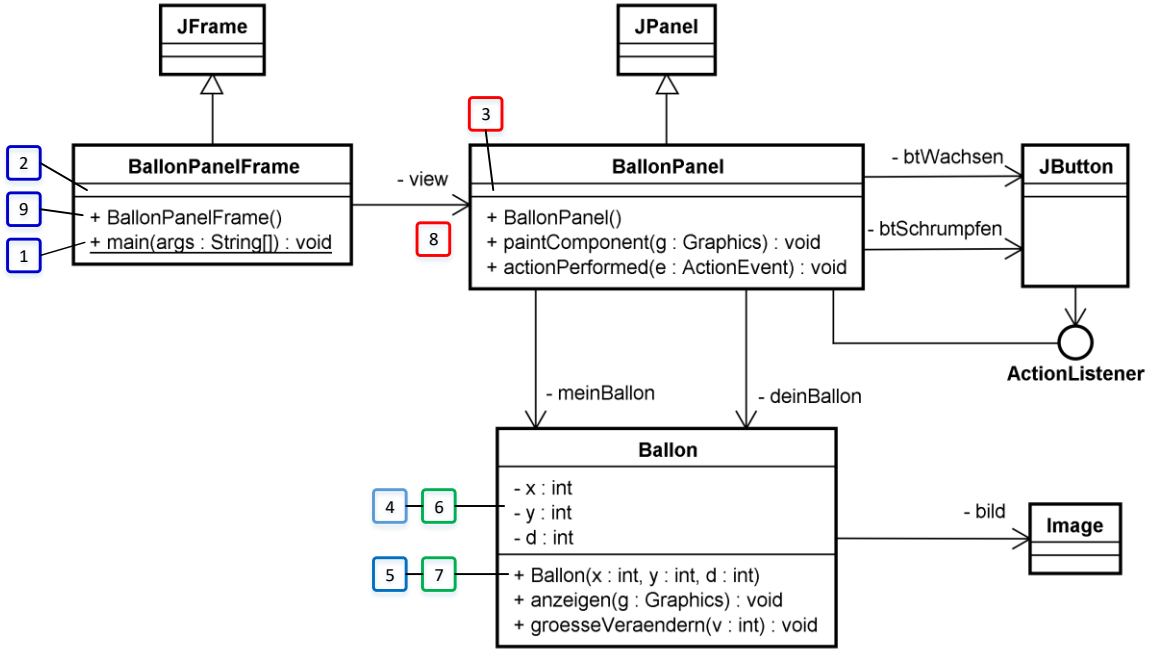



Ausgabe in der Konsole mittels TraceV7:

| | |
|---|---|
| 1 | Start via BallonPanelFrame.main(String args[]) |
| 2 | Attribute von BallonPanelFrame@1560911714 werden initialisiert ... |
| 3 | Konstruktor BallonPanelFrame():BallonPanelFrame@1560911714 wird ausgeführt ... |
| 4 | Attribute von BallonPanel@548246552 werden initialisiert ... |
| 5 | Attribute von Ballon@835648992 werden initialisiert ... |
| 6 | Konstruktor Ballon():Ballon@835648992 wird ausgeführt ... |
| 7 | Attribute von Ballon@401625763 werden initialisiert ... |
| 8 | Konstruktor Ballon():Ballon@401625763 wird ausgeführt ... |
| 9 | Konstruktor BallonPanel():BallonPanel@548246552 wird ausgeführt ... |

Beispiel mit Klassendiagramm:

Aufgrund des Klassendiagramms lässt sich bereits eine Aussage bezüglich der vermeintlichen Instanziierungssequenz machen. Noch unscharf bleibt dabei die Frage beantwortet, ob Objekte allenfalls bei den Attributen oder im Konstruktor erzeugt werden: Werden zum Erzeugen Parameter gebraucht, die erst im Konstruktor bekannt sind, werden die Objekte zwingend im Konstruktor erzeugt. Ansonsten können sie bereits bei den Attributen erzeugt werden. Sind benötigte Parameter erst zur Laufzeit bekannt, wird zu gegebener Zeit an entsprechender Stelle das Objekt erzeugt.




In obigem Beispiel wurde angenommen, dass das *BallonPanel* bei den Attributen von *BallonPanelFrame* erzeugt wird.

Ausgabe in der Konsole mittels TraceV7:

| | |
|---|---|
| 1 | Start via BallonPanelFrame.main(String args[]) |
| 2 | Attribute von BallonPanelFrame@1560911714 werden initialisiert ... |
| 3 | Attribute von BallonPanel@548246552 werden initialisiert ... |
| 4 | Attribute von Ballon@835648992 werden initialisiert ... |
| 5 | Konstruktor Ballon():Ballon@835648992 wird ausgeführt ... |
| 6 | Attribute von Ballon@401625763 werden initialisiert ... |
| 7 | Konstruktor Ballon():Ballon@401625763 wird ausgeführt ... |
| 8 | Konstruktor BallonPanel():BallonPanel@548246552 wird ausgeführt ... |
| 9 | Konstruktor BallonPanelFrame():BallonPanelFrame@1560911714 wird ausgeführt ... |