

MEHR ZU VARIABLEN, METHODEN UND DER *DRAW LOOP*

Ausdrücken, Rückgabe von Werten aus Methoden und dem *Draw Loop* in der GraphicsApp

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#) verwendet werden.

AKTUELLER SEMESTERFORTSCHRITT (WOCHE 4)

Kursabschnitt	Themen		
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
Klassenmodellierung	Grundlagen der Klassenmodellierung	Vererbung & Sichtbarkeit	
Interaktive Anwendungen	Event-basierten Programmierung	String- & Textverarbeitung	
Datenstrukturen	Listen, Maps & die Collections	Speicherverwaltung	Umgang mit Dateien
Software Engineering	Planhaftes Vorgehen bei der Softwareentwicklung	Qualitätsaspekte von Quellcode	

PINGO-QUIZ



<https://pingo.coactum.de/447102> 

RÜCKBLICK: VARIABLEN UND KLASSEN

- Variablen speichern Werte während der Programmausführung (Laufzeit). Sie sind Platzhalter für unterschiedliche Daten, funktionieren aber immer nur mit einem bestimmten Datentyp.
- Werte, die während der Implementierung festgelegt werden und zur Laufzeit unveränderlich sind, nennt man Konstanten.
- Klassen definieren Eigenschaften und Verhalten für ein oder mehrere Objekte.
- Objekte sind die Instanzen der Klassen.

RÜCKBLICK: GRAPHICSAPP

- Mit Hilfe der GraphicsApp können wir einfache graphische Elemente auf dem Bildschirm zeichnen.
- Alle sichtbaren Elemente werden durch Instanzen der entsprechenden Klassen repräsentiert.

DAS PROGRAMM FÜR HEUTE

- Informationen zur ersten Studienleistung
- Typenumwandlung (*Casts*)
- Ausrücke und Operatoren in Java
- Ergebnisse aus Methoden zurückgeben
- Mehr zum *Draw Loop*

INFORMATIONEN ZUR ERSTEN STUDIENLEISTUNG

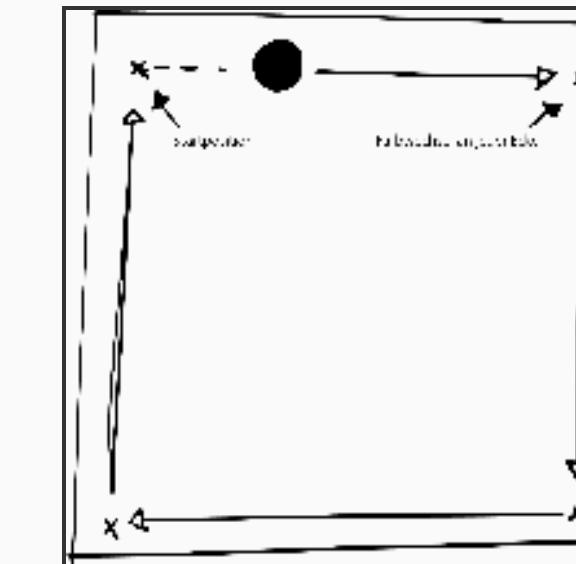
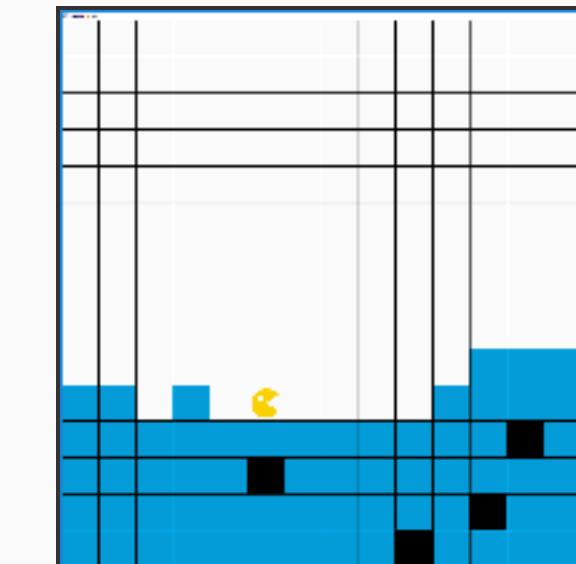
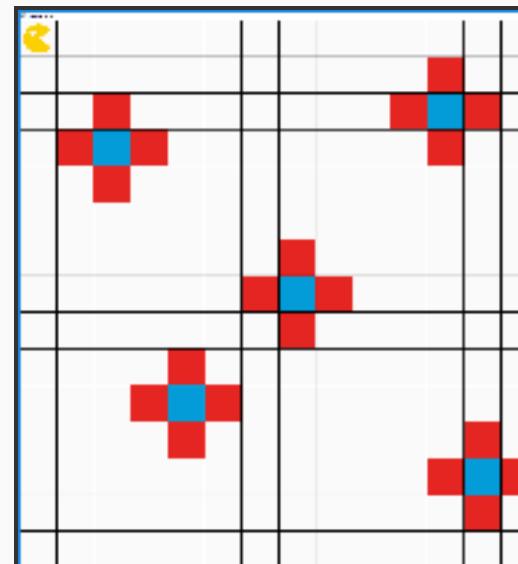
DIE ERSTE STUDIENLEISTUNG (1/2)

- Ab heute, 16 Uhr finden Sie die Aufgabenbeschreibung und das Starterpaket zur ersten Studienleistung im **GRIPS-Kurs**
- Bearbeiten Sie die drei Aufgaben und laden Sie Ihre Lösung **bis zum 17. November (23:55 Uhr)** über den vorbereiteten Link hoch.
- Allgemeine Informationen zum Ablauf der Studienleistung finden Sie **hier** .
- Die Bewertungskriterien finden Sie im GRIPS-Kurs (beim Hochladen der Lösung).

DIE ERSTE STUDIENLEISTUNG (2/2)

In der ersten Studienleistung müssen Sie:

- Zwei komplexere *Bouncer*-Programme erstellen
- und eine erste, einfache Animation mit der GraphicsApp-Umgebung realisieren



DATENTYPEN KONVERTIEREN: CASTS

KOMPATIBILITÄT ZWISCHEN (PRIMITIVEN) DATENTYPEN HERSTELLEN

Bei der gleichzeitigen Verwendung unterschiedlicher Datentypen müssen Sie einige Regeln beachten:

```
1 int x = 5;
2 // BUG: y bekommt hier den Wert 2.0 nicht 2.5 zugewiesen!
3 // Das Ergebnis einer ganzzahligen Division ist immer eine Ganzzahl!
4 double y = x/2;
```

Mithilfe von *Casts* kann eine Variable **zwischenzeitlich** so behandelt werden, als wäre Sie von einem anderen Typ:

```
1 int x = 5;
2 // y bekommt hier den Wert 2.5 zugewiesen, da x während der Auswertung,
3 // und nur da, als Gleitkommazahl (double) verwendet wird. Der Datentyp der
4 // Variable x wird nicht geändert!
5 double y = ((double) x) / 2;
```

CAST != RUNDEN

Beim **können** Informationen verloren gehen – Wird eine Gleitkommazahl (`double`) zu einer Ganzzahl (`int`) *gecastet*, wird der Wert am Komma abgeschnitten:

```
1 double g = 3.55
2 System.out.println((int) g); // gibt 3 aus
```

Hinweis: *Richtiges* Runden funktioniert mit der Funktion `round()` aus dem Math-Paket. Der Befehl `Math.round(3.55)` gibt 4 zurück.

EXPLIZITER VS IMPLIZITER CAST

Führen wir (die ProgrammiererInnen) den *Cast* einer Variable manuell durch, so handelt es sich um einen **expliziten Cast** (Zieldatentyp in runden Klammern VOR der Variable / dem Ausdruck, der *gecastet* werden soll):

```
1 double g = 3.55;  
2 (int) g; // Cast zu Ganzzahl
```

Die *Runtime* führt zur Laufzeit auch **implizite Casts**. In der Zeile `double x = 1 + 2.3;` wird der Wert 1 automatisch zu einem double-Wert (1.0) *gecastet*, da der andere Summand eine Gleitkommazahl ist.

REGELN FÜR DIE TYPUMWANDLUNG

- Beim expliziten *Cast* können Informationen verloren gehen, z.B. bei der Umwandlung von `double` zu `int` (aus `1.9` wird dann `1`).
- Beim impliziten *Cast* gehen keine Informationen verloren. Die *Runtime* interpretiert den ursprünglichen Wert als einen Typ mit höherem Informationsgehalt, dessen Informationsraum nicht voll ausgenutzt wird (z.B. bei `int` zu `double`, bzw. `1` zu `1.0`).

OPERATOREN UND AUSDRÜCKE

DER MODULO-OPERATOR (1/2)

Um den Rest einer ganzzahligen Division zu erhalten existiert in Java (und in den meisten anderen Programmiersprachen) der Modulo-Operator %:

$$15 \% 3 = 8 \text{ (} 15/3 = 5; \text{Rest ist } 0\text{)}$$

$$14 \% 8 = 6 \text{ (} 14/8 = 1; \text{Rest ist } 6\text{)}$$

$$14 \% 17 = 14 \text{ (} 14/17 = 0; \text{Rest ist } 14\text{)}$$

DER MODULO-OPERATOR (2/2)

Mit dem Modulo-Operator können Sie leicht überprüfen, ob eine Zahl gerade oder ungerade (durch 2 teilbar) ist:

```
1 int num = 12;
2 System.out.println("Die Zahl " + num + " ist ...");
3 if(num % 2 == 0) {
4     System.out.println("... gerade!");
5 } else {
6     System.out.println("... ungerade!");
7 }
```

AUSDRÜCKE (EXPRESSIONS)

- Als Ausdruck (oder *Expression*) wird die Verknüpfung von Sprachelementen durch Operatoren bezeichnet
- Ausdrücke werden beim Codieren definiert und zur Laufzeit ausgewertet
- Ausdrücke haben *immer* ein Ergebnis. D.h. nach der kompletten Auswertung eines Ausdrucks bleibt ein einzelner Datenwert übrig

```
1 int num1 = 2;
2 int num2 = 3;
3 // Im Ausdruck auf der rechten Seite werden zwei Sprachelemente (hier Variablen) durch einen Operator (hier +)
4 // verknüpft. Nach Auswertung wird das Ergebnis (hier "5") der Variable auf der linken Seite zugewiesen.
5 int total = num1 + num2;
```

AUSDRÜCKE IN JAVA

Ein **Ausdruck** besteht aus einem **Term** und beliebig vielen (auch keinen!) weiteren, durch **Operatoren** verknüpfte **Terme**:

AUSDRUCK => TERM [+ OPERATOR TERM]

- **Terme** können dabei **Litereale** (Werte), **Konstanten**, **Variablen**, **Methodenaufrufe** oder Ausdrücke (beliebig tiefe Verschachtlung) sein.
- **Operatoren** können mathematisch (+,-,*,/ oder %) oder logisch (&&, ||, !) sein.

BOOLESCHE AUSDRÜCKE UND VERGLEICHSOPERATOREN (1/2)

Ein boolescher Ausdruck ist eine Auswertung eines oder mehrerer passender Terme und ergibt immer einen eindeutigen Wahrheitswert: wahr (true) oder falsch (false). Für die Verwendung innerhalb boolescher Ausdrücke existieren spezielle Vergleichsoperatoren, die zum Teil aus der Mathematik bekannt sind.

Hinweis: Sie kennen das bereits (ausführlicher) aus dem EIMI-Kurs! In diesem Kurs beschränken wir uns auf einen Teilbereich der Booleschen-Logik

BOOLESCHE AUSDRÜCKE UND VERGLEICHSOPERATOREN (2/2)

Operator *Übersetzung*

<code>==</code>	<i>ist gleich</i>	Nicht mit dem Zuweisungsoperator = verwechseln!
<code>!=</code>	<i>ist ungleich</i>	
<code>></code>	<i>größer als</i>	
<code><</code>	<i>kleiner als</i>	
<code>>=</code>	<i>größer oder gleich</i>	
<code><=</code>	<i>kleiner oder gleich</i>	
<code>!</code>	<i>nicht</i>	Negation des nachfolgenden Ausdrucks

BEISPIEL: ZAHLEN PRÜFEN

```
1
2 public static final int UPPER_LIMIT = 50;
3
4 // ...
5
6 int x = 42;
7 if(x <= UPPER_LIMIT) {
8     // HINWEIS: Versuchen Sie IMMER sinnvolle Ausgaben und Rückmeldungen zu produzieren!
9     System.out.println("Der Wert " + x + " entspricht oder liegt unter dem Limit von " + LIMIT);
10 }
```

Im Beispiel wird die in x gespeicherte Zahl mit dem Wert aus der Konstante UPPER_LIMIT verglichen. Zum Einsatz kommt der *kleiner gleich*-Operator. Ist die Zahl gleich dem oder kleiner als der in UPPER_LIMIT festgehaltene Grenzwert, wird dies dem Nutzer/der Nutzerin über eine Bildschirmausgabe mitgeteilt.

VERKNÜPFUNG VON BOOLESCHEN AUSDRÜCKEN

- Boolesche Ausdrücke können durch besondere Operatoren verknüpft bzw. kombiniert werden.
- Für die logische Verknüpfung stehen uns die Operatoren `!`, `&&` und `||` zur Verfügung.

LOGISCHES "NICHT" (!)

Ein logisches **NICHT** kehrt den Wahrheitswert des betroffenen Terms um:

$\neg p$ ist wahr (true) wenn p falsch (false) ist
 $\neg p$ ist falsch (false) wenn p wahr (true) ist

Damit können Sie einige *fehlende* Bedingungsprüfungen in der *Bouncer*-Umgebung ergänzen:

```
1 /**
2  * Bouncer bewegt sich solange vorwärts, bis er auf einem grünen Feld ankommt.
3  */
4 while(!bouncer.isOnFieldWithColor(FieldColor.GREEN)) {
5     move();
6 }
```

LOGISCHES "UND" (`&&`)

Ein logisches **UND** kombiniert die Wahrheitswerte mehrerer Terme:

`p && q` ist wahr (true) wenn p und q wahr sind

Für die Auswertung gilt dabei die folgende *Wahrheitstabelle*:

&	true	false
true	true	false
false	false	false

LOGISCHES "ODER" (\parallel)

Ein logisches **ODER** kombiniert die Wahrheitswerte mehrerer Terme:

$p \parallel q$ ist wahr (true) wenn

- 1) p oder q wahr sind oder
- 2) p und q wahr sind

Für die Auswertung gilt dabei die folgende *Wahrheitstabelle*:

&	true	false
true	true	true
false	true	false

BEISPIEL: ZAHLEN PRÜFEN

```
1
2 public static final int UPPER_LIMIT = 50;
3 public static final int LOWER_LIMIT = 40;
4
5 // ...
6
7 int x = 42;
8 if(x >= LOWER_LIMIT && x <= UPPER_LIMIT) {
9     // HINWEIS: Versuchen Sie IMMER sinnvolle Ausgaben und Rückmeldungen zu produzieren!
10    System.out.println("Der Wert " + x + " liegt im Toleranzbereich von " + LOWER_LIMIT + " bis " + UPPER_LIMIT);
11 }
```

Im Beispiel wird ein komplexer Ausdruck verwendet um die Zahl in der Variable x zuerst mit dem unteren und dann mit dem oberen Limit zu vergleichen. Durch die Kombination der beiden *größer gleich* und *kleiner gleich* Vergleiche mit einem logischen **UND** wird geprüft, ob die Zahl zwischen den beiden Werten liegt.

OPERATOR PRECEDENCE: AUSWERTUNGSREGELN

Bei der Auswertung von mathematischen Ausdrücken gilt in Java diese Reihenfolge: [Klammern] vor [$*$, $/$, $\%$] vor [+ und -]. Operatoren mit gleicher *Precdence* (z.B. + und - werden von *links nach rechts* ausgewertet).

```
1 int x = 1 + 4 * 5 / 2;
```

Auswertungsreihenfolge im Beispiel:

- $4*5$ (Zwischenergebnis ist 20)
- $/2$ (Zwischenergebnis ist 10)
- $1+$ (Endergebnis ist 11)

OPERATOR PRECEDENCE: LESBARKEIT UND KLAMMERUNG

Nutzen Sie die Möglichkeiten von Klammern, um Ihre Intention beim Programmieren möglichst explizit auszudrücken:

```
1 int x = 1000;
2 int y = 100;
3 int i = 0;
4
5 // Automatische Auswertung
6 i = x + y / 100; // Ergebnis: 1001, Intention unklar
7
8 // Implizites Vorziehen der Addition
9 i = (x + y) / 100; // Ergebnis: 11, Intention klar
10
11 // Implizite Nutzung der automatischen Auswertung
12 i = x + (y / 100); // Ergebnis 1001, Intention klar
```

METHODEN MIT PARAMETERN UND RÜCKGABEWERTEN

FÜR WELCHEN ZWECK HABEN WIR BIS JETZT METHODEN EINGESETZT?

Bis jetzt dienen Methoden dazu, zusammengehörige Anweisungen zu gruppieren und den Code durch Decomposition verständlicher zu machen:

- `turnRight()` als Ergänzung der fehlenden Möglichkeit, *Bouncer* nach Rechts zu drehen
- `moveToObstacle` zum Auslagern eines Teilproblems unseres *Algorithmus*
- `drawCircle()` (von letztem Übungsblatt) um häufig verwendete Befehlsmuster nur einmal implementieren zu müssen

Methoden können mehr!

RÜCKGABEWERTE UND PARAMETER

Methoden haben in Java noch zusätzliche Möglichkeiten, die wir bis jetzt nur indirekt genutzt haben: Methoden können – wie Funktionen* in der Mathematik – Ergebnisse zurückliefern, die auf der Basis von an die Methode übergebene Eingabewerte (Parameter) berechnet wurden:

wert <= FUNKTION(PARAMETER)

oder

sinus <= sin(WERT)

*In prozeduralen Programmiersprachen wie *C* oder *Pascal* heißen Subroutinen tatsächlich Funktionen und sind unabhängig von Objekten. In Java muss eigentlich immer von Methoden gesprochen werden, da Subroutinen werden hier immer im Kontext einer Klasse oder eines Objekts ausgeführt.

BEISPIELE AUS DER MATHEMATIK: SINUS

```
x <= sin(90); x = 1
```

```
x <= sin(70); x = 0,939692620786
```

```
x <= sin(45); x = 0,707106781187
```

```
x <= sin(90); x = 1
```

Die Sinusfunktion berechnet ein Ergebnis aus dem Eingabeparameter (hier: Winkel) und liefert das Ergebnis zurück. Aus gleichen Eingabeparametern wird das gleiche Ergebnis berechnet (Zeile 1 und 4).

METHODEN IN JAVA DEFINIEREN

- Eine Methode besteht in Java aus dem **Methodenkopf** (oder *Signatur*) und dem **Methodenrumpf**
- Im Methodenkopf wird die Methode und deren Parameter beschrieben
- Im Methodenrumpf stehen die eigentlichen Befehle, die beim Aufruf der Methode ausgeführt werden sollen
- Das Erstellen einer Methode wird auch **Deklaration** genannt (hier wird in manchen Programmiersprachen auch noch zwischen *Deklaration* und *Definition* unterschieden)

AUFBAU DES METHODENKOPFS

```
1 SICHTBARKEIT[1] RÜCKGABETYP[2] NAME[3] (PARAMETERTYP[4]* PARAMETERNAME[5]*) {  
2     // ...  
3 }
```

[1]: Sichtbarkeit der Methode, z.B. private (nur innerhalb der Klasse) oder public (überall sichtbar)

[2]: Was für Informationen gibt die Methode zurück? void bedeutet hier, es gibt keinen Rückgabewert

[3]: Name der Methode für deren späteren Aufruf

[4]: Typ des Parameters (Eingabedaten)

[5]: Name des Parameters (Variable) über den er INNERHALB der Methode angesprochen werden kann (*Scope*)

*Eine Methode kann über mehrere Parameter (jeweils mit Typ & Namen) verfügen. Diese werden dann durch Komma getrennt angeben.

ERGEBNISSE ZURÜCKLIEFERN

Innerhalb der Methode wird die Rückgabe eines Werts durch das Schlüsselwort `return` eingeleitet:

```
return AUSDRUCK*;
```

Hinweis: *Richtiges* Die Methode wird mit der `return`-Anweisung sofort beendet und gibt den Wert des Ausdrucks zurück. Sie können auch Methoden ohne Rückgabewert (`void`) an jeder Stelle abbrechen, indem Sie den `return`-Befehl ohne nachfolgenden Ausdruck verwenden.

*Hier kann ein beliebiger Ausdruck, eine Variable oder ein Wert stehen, der aber mit dem Rückgabetypen der Methode übereinstimmt

BEISPIEL: KILOMETER IN METER UMRECHNEN

```
1 private double kmToMeters(double km) {  
2     return 1000 * km;  
3 }
```

Beim Aufruf der Methode wird ein double-Wert übergeben, der eine Strecke in Kilometern repräsentiert. Im Rumpf der Methode wird dieser Wert mit dem Faktor multipliziert und das entsprechende Ergebnis zurückgegeben. Die Umrechnung von km in m erfolgt hier kombiniert mit der Rückgabe des Ergebnis (return).

BEISPIEL: MEHRERE PARAMETER UND * RETURNS*

```
1 private int max(int value1, int value2) {  
2     if (value1 > value2) {  
3         return value1;  
4     } else {  
5         return value2;  
6     }  
7 }
```

Die Methode bekommt zwei Parameter (value1 und value2) übergeben und überprüft, welcher Wert größer ist. Die Rückgabe ist abhängig von den Parametern, die beim Aufruf übergeben werden (z.B. `int max = max(12,14);`).

BEISPIEL: OBJEKTE ALS RÜCKGABEWERT

Auch Objekte können aus Methoden zurückgegeben werden. Der Mechanismus ist der gleiche wie bei primitiven (Rückgabe-)Typen

```
1 private static final int BORDER_WIDTH = 2;
2 private static final Color BORDER_COLOR = Colors.GREY;
3
4 public void draw() {
5     drawBackground(Colors.WHITE);
6     Circle myCircle = createCircleWithBorder(100,100,70);
7     myCircle.draw();
8 }
9
10 private Circle createCircleWithBorder(int xPos, int yPos, int radius) {
11     Circle circle = new Circle(xPos, yPos, 2*radius, 2*radius, Colors.RED);
12     circle.setBorder(BORDER_COLOR, BORDER_WIDTH);
13     return circle;
14 }
```

VARIABLEN INNERHALB VON METHODEN

- Variablen die innerhalb einer Methode deklariert werden heißen lokale Variablen.
- Sie leben vom Eintrittspunkt in die Methode bis zum Austritt (*Scope*: Sichtbarkeit und Gültigkeit).
- Sie sind nur innerhalb der Methode sichtbar, in der sie definiert wurden.
- Parameter sind lokale Variablen, die mit den übergebenen Werten initialisiert werden.
- Das Set (die Menge) an lokalen Variablen bei der Ausführung einer Methode nennt man Stack Frame.

BEISPIEL: FAKULTÄT BERECHNEN

```
1 private static final int MAX_NUM = 4;
2
3 private void doSomeMath() {
4     for (int i = 0; i < MAX_NUM; i++) {
5         System.out.println(i + "!" + factorial(i));
6     }
7 }
8
9 private int factorial(int n) {
10    // Die lokale Variable result ist nur in dieser Methode verwendbar
11    int result = 1;
12    // Wir können hier den Namen der Zählervariable i (aus doSomeMath) wiederverwenden, da
13    // die Methoden unterschiedliche Scopes haben.
14    for (int i = 1; i <= n; i++) {
15        result *= i;
16    }
17    return result;
18 }
```

In der Methode factorial wird die Fakultät der übergebenen Zahl berechnet. Für die Berechnung des Produktes wird eine for-Schleife eingesetzt, die von 1 bis zum in n gespeicherten Wert läuft und in jeder Iteration den in result gespeicherten Wert mit dem aktuellen Wert der Zählervariable i multipliziert und den ursprünglichen Wert von result mit diesem Ergebnis überschreibt.

HÄUFIGE PROBLEME MIT PARAMETERN, VARIABLEN UND BUGS (1/3)

Betrachten Sie dieses Beispiel:

```
1 private void doSomeMath() {  
2     int x = 3;  
3     addFive(x);  
4     System.out.println("x = " + x);  
5 }  
6  
7 private void addFive(int x) {  
8     x += 5;  
9 }
```

Das Programm gibt $x = 3$ aus. Warum ist das so?

HÄUFIGE PROBLEME MIT PARAMETERN, VARIABLEN UND BUGS (2/3)

Bug-Gefahr: An die Methode wird ein Parameter mit dem Wert von `x` übergeben, nicht die Variable `x` selber. Das heißt alle Veränderungen innerhalb der Methode werden nur in deren *Scope* angewandt: `x` aus `doSomeMath` bleibt unverändert. Im Beispiel existieren zwei verschiedene Variablen mit dem Namen `x` – syntaktisch ist dies so zulässig (verschiedene Namensräume). Oft sind unterschiedliche Namen aber sinnvoller!

Hinweis: Java arbeitet mit dem *Call by Value* – Prinzip, d.h. für primitive Datentypen gilt immer: Übergeben wird immer der Wert (eine Kopie) der betreffenden Variable, nicht die Variable (bzw. Ihre Speicheradresse) selbst! Das gilt eigentlich auch für Objekte, hat dort aber andere Konsequenzen: Manipulationen an übergebenen Objekten ändern die ursprünglichen Objekte!

HÄUFIGE PROBLEME MIT PARAMETERN, VARIABLEN UND BUGS (2/3)

Ohne Bugs:

```
1 private int addFive(int x) {  
2     x += 5;  
3     return x;  
4 }  
5  
6 private void doSomeMath() {  
7     int x = 3;  
8     x = addFive(x);  
9     System.out.println("x = " + x);  
10 }
```

Das Programm gibt x = 8 aus.

ANIMATIONEN IN DER GRAPHICSAPP: DER DRAW LOOP

ANIMATIONEN IN DER GRAPHICSAPP (1/2)

- Bis jetzt: Unsere Anwendungen zeichnen fast immer ein statisches Bild: Kreis, *Target*, ...
- Was wir eigentlich wollen: Interaktive, animierte Anwendungen!
- Wie wir das erreichen: Zeichnen von graphischen Objekte, verändern von Position, Farbe und Größe und erneutes Zeichnen
- Die GraphicsApp erlaubt so etwas über den *draw loop*

Hinweis: Die GraphicsApp verfügt über eine öffentliche (public) Methode `draw`. Diese Methode wird automatisch und regelmäßig vom System aufgerufen (Das Intervall wird über einen festen FPS-Wert bestimmt (*frames per second*)). Mit jedem neuen Aufruf können wir Teile des vorher gezeichneten Bildschirms überschreiben. Das Prinzip ähnelt dabei der Animation in einem Zeichentrickfilm.

ANIMATIONEN IN DER GRAPHICSAPP (2/2)



BEISPIEL: FLYINGBALL

```
1 public class FlyingBall extends GraphicsApp {  
2     // Instanzvariable für den Ball  
3     private Circle ball;  
4  
5     public void initialize() {  
6         setCanvasSize(500,500);  
7         // Erstellen des Balls  
8         ball = new Circle( 250,250, 50, 50, Colors.RED);  
9     }  
10  
11    public void draw() {  
12        drawBackground(Colors.WHITE);  
13        // Bewegen des Balls  
14        ball.move(1, 1);  
15        // Zeichnen des Ball  
16        ball.draw();  
17    }  
18 }
```

INSTANZVARIABLEN (1/2)

Häufig benötigen verschiedene Methoden einer Klasse Zugriff auf dieselben Variablen:

- Wenn wir Objekte in `initialize` anlegen und in `draw` manipulieren bzw. zeichnen wollen, müssen die entsprechenden Variablen in beiden Methoden zugänglich machen.
- Instanzvariablen werden in der Klasse und außerhalb von Methoden (typischerweise zu Beginn) definiert und stehen einer Instanz in jeder Methode zur Verfügung (jede Instanz hat ihre eigenen Eigenschaftswerte): `private TYP NAME;`

INSTANZVARIABLEN (2/2)

Wir nutzen Instanzvariablen (Eigenschaften), um Informationen zu speichern, die:

- Innerhalb der Klasse bzw. des Objekts („überall“) verfügbar sein müssen.
- Nicht als Parameter übergeben oder als lokale Variablen abgespeichert werden können.

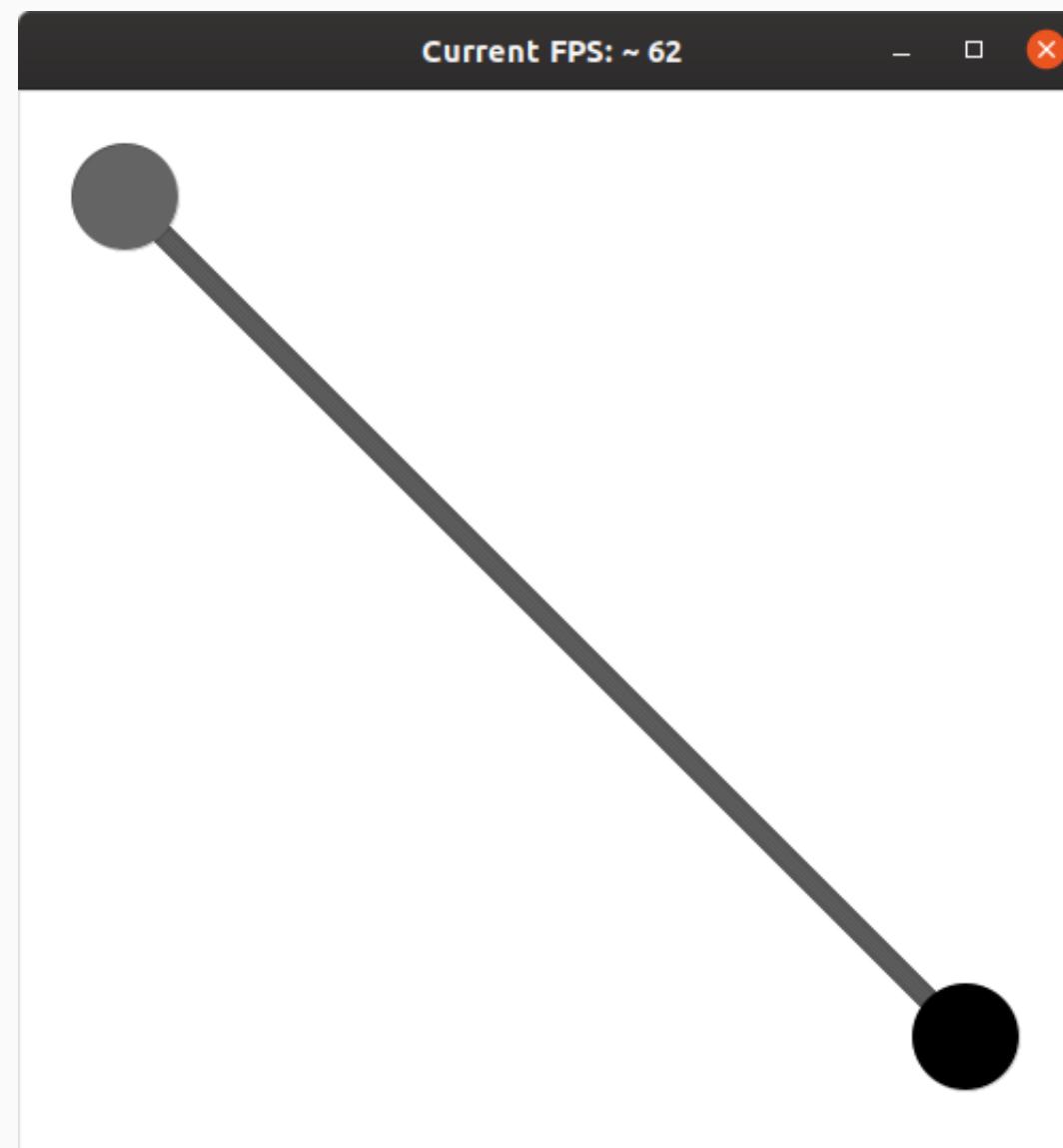
Hinweis: Instanzvariablen werden nur dann eingesetzt, wenn die gespeicherte Information tatsächlich an unterschiedlichen Stellen der Klasse verwendet wird. Temporäre Informationen werden lokal (in Methoden) gespeichert und/oder über Parameter und Rückgabewerte verarbeitet.

GESCHWINDIGKEIT UND ANIMATION

Eine Animation kommt durch die Änderung der x- bzw y-Position und des anschließenden Neuzeichnens eines Objekts zustande. Die Geschwindigkeit ergibt sich durch die Größe der Positionsveränderungen: *pixels per frame*. Die Geschwindigkeit wird oft als dx bzw. dy (Delta, Differenz) bezeichnet.

Frame	Code	Position	Geschwindigkeit
1	ball.move(1,1); ball.draw();	x = 251, y= 251	dx = 1, dy = 1
2	ball.move(1,1); ball.draw();	x = 252, y= 252	dx = 1, dy = 1
3	ball.move(1,1); ball.draw();	x = 253, y= 253	dx = 1, dy = 1
4	ball.move(1,1); ball.draw();	x = 254, y= 254	dx = 1, dy = 1
5	ball.move(1,1); ball.draw();	x = 255, y= 255	dx = 1, dy = 1

KOLLISIONEN (MIT DER WAND) ERKENNEN



In jedem *Frame* stehen Ihnen die notwendigen Informationen zur Verfügung, um eine mögliche Kollision eines Objektes mit dem Rand der Zeichenfläche zu prüfen: die **Position** des Objektes, dessen **Größe** und die **Breite** bzw. **Höhe** der Zeichenfläche.

In unserem Beispiel: *Wenn die x-Position des Balls addiert mit seinem Radius größer gleich der Breite der Zeichenfläche ist, heißt das: Der Ball berührt den Rand oder hat ihn bereits überschritten.*

ZUSAMMENFASSUNG (1/2)

- Ausdrücke liefern einen Wert zurück, Operatoren verknüpfen Ausdrücke.
- Zur Laufzeit kann der Typ eines Wertes durch Typumwandlung geändert werden.

ZUSAMMENFASSUNG (2/2)

- Methoden können einen oder mehrere Eingabewerte (Parameter) verwenden und ein Ergebnis (Rückgabewert) ausgeben. Parameter und Rückgabe können primitive oder komplexe Datentypen sein, der jeweilige Typ wird in der Methodensignatur angegeben.
- In Java gilt das *Call by Value*-Prinzip: Primitive Werte werden als Kopien übergeben.
- Animationen werden in der GraphicsApp durch Zustandsmanipulation und Neuzeichnen von Objekten erreicht. Die draw-Methode wird zur Laufzeit regelmäßig aufgerufen.

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo (alexander.bazo@ur.de). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach (florin.schwappach@ur.de). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter mioop@mailman.uni-regensburg.de.

QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.