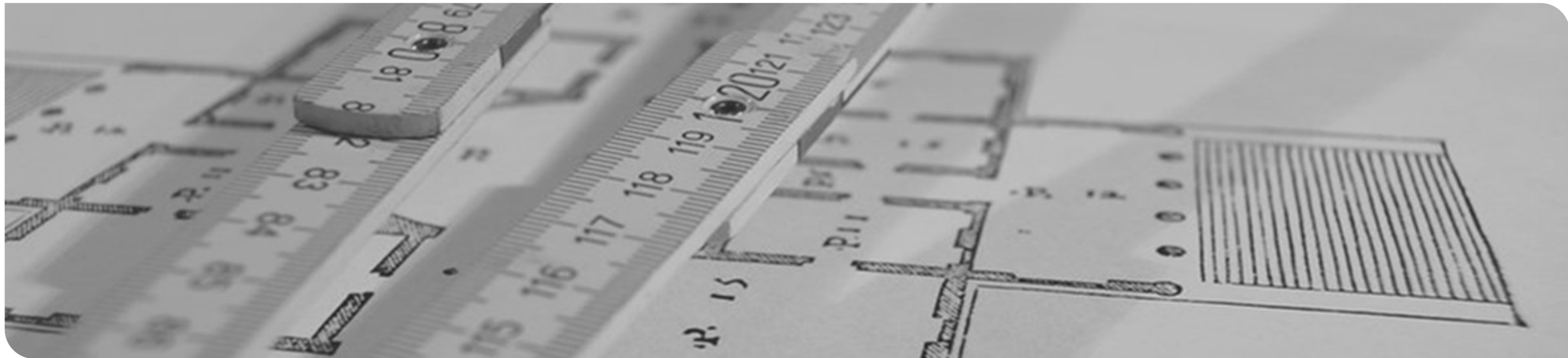


# Vorlesung Programmieren

## 4. Konstruktoren und Methoden

Prof. Dr.-Ing. Anne Koziolk



# Immatrikulationsbefragung im WS 2024/25

## Worum geht es?

- Fragen zur Hochschulwahl, Bewerbung und Zulassung sowie zur Studieneinstiegs-Phase am KIT

## Wer wird befragt?

- Alle Neu- und Erstimmatrikulierten Studierende am KIT im aktuellen Wintersemester

## Warum soll ich mitmachen?

- Gestalten Sie das KIT mit und helfen Sie, die Studieneingangsphase für Sie und nachfolgende Studierende zu optimieren
- Die Ergebnisse fließen direkt in das Qualitätsmanagement des KIT ein

## Konzept und Teilnahme

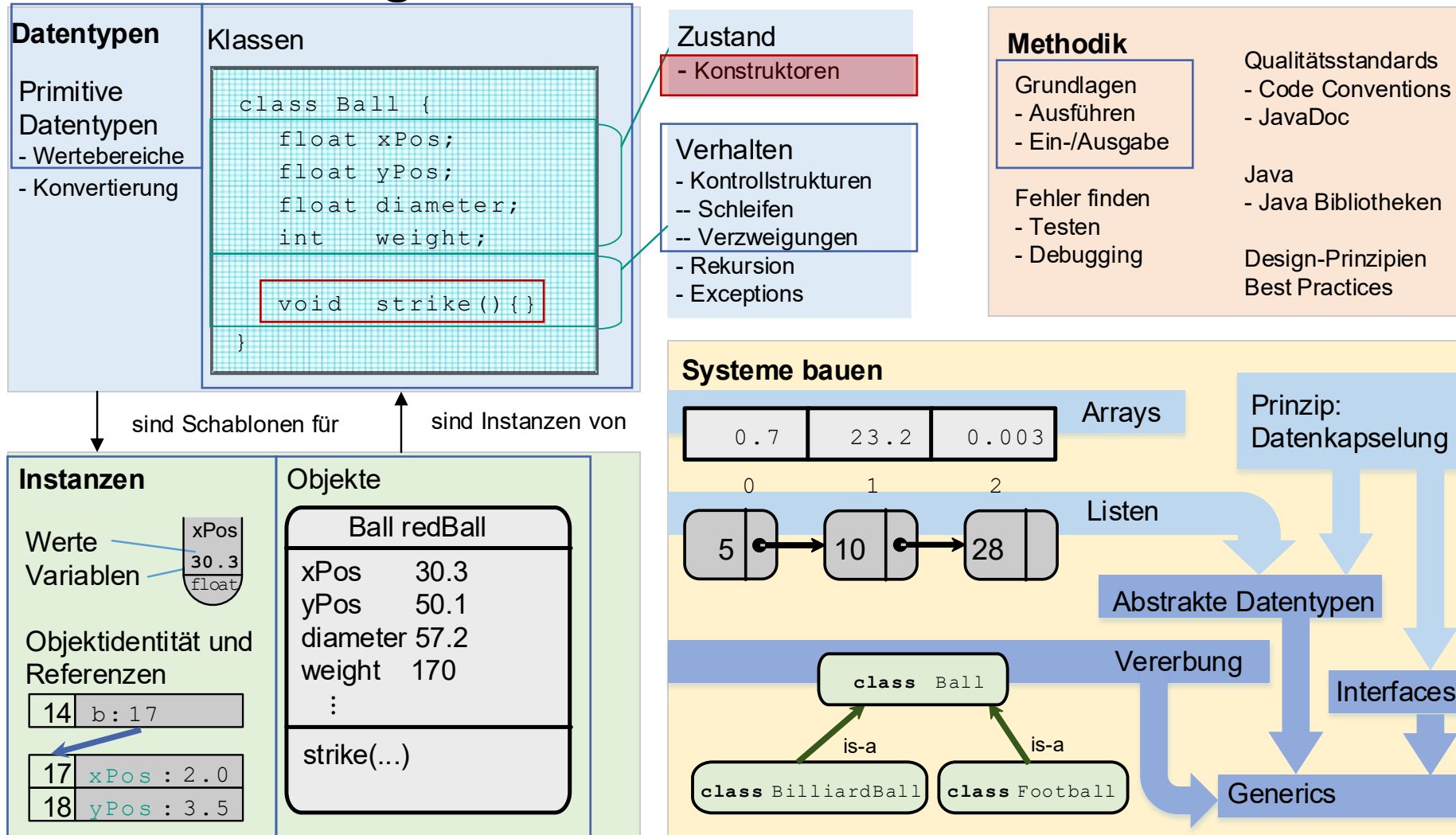
## Wie kann ich teilnehmen?

- Die Einladung erfolgt per E-Mail an Ihre @student.kit.edu Adresse
- Absender: [befragungen@sts.kit.edu](mailto:befragungen@sts.kit.edu)
- Weitere Informationen können Sie der Einladungsmail entnehmen

## Wann wird befragt?

- Die Einladungen werden Anfang November versendet
- Sie haben Fragen? Dann können Sie sich gerne an Anne Pfeifer wenden: [befragungen@sts.kit.edu](mailto:befragungen@sts.kit.edu)

# Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



# Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung <b>Audimax 50.34 Raum -101 und Raum -102</b>
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

# Lernziele heute

## Konstruktoren

- Sie können mit Konstruktoren Objekte erzeugen und gleichzeitig initialisieren.
- Sie können die `this`-Referenz verwenden, um auf Attribute und Methoden eines Objekts zuzugreifen.



Quelle: <http://phdcomics.com>

## Methoden

- Sie können das Verhalten Ihres Programms in Methoden aufteilen.
  - Sie können Methoden definieren
  - Sie können Methoden aufrufen
- Sie können Klassen- und Objekt-Methoden anlegen und aufrufen.
- Sie können lokale Variablen für Zwischenergebnisse verwenden.
- Sie können Rückgabewerte von Methoden definieren.

# Objekte erzeugen und initialisieren

- Wir hatten bereits gesehen:
- Die Anweisungsfolge

```
Vector2D p = new Vector2D();  
p.x = 4.0f;  
p.y = 22.5f;
```

kreiert ein neues Objekt und initialisiert dessen Attribute.

- Die **Objekterzeugung** und **Initialisierung der Attribute** tritt in Programmen fast immer gemeinsam auf!
- **Außerdem:** Initialisierung könnte **vergessen** werden, dann...
  - ... sind Attribute nicht richtig initialisiert
  - ... ist der Objektzustand undefiniert

→ **Konstruktoren: Fassen Objekterzeugung und Initialisierung der Attribute zusammen**

# Konstruktoren

- Werden bei der Erzeugung von Objekten (mittels `new`) aufgerufen
- Dienen der **Initialisierung** eines Objekts
  - Initialisierung = Setzen von Anfangswerten der Attribute
  - Legen damit den Startzustand eines Objekts fest
- Müssen denselben Namen haben wie die zugehörige Klasse
- Erhalten die Anfangswerte des Objekts als **Parameter**

# Parameter

- Ein **Parameter** ist ein Platzhalter für beliebige Werte (eines Typs)
- ...ist also eine Ausprägung einer Variablen
- Prinzip bekannt aus der Mathematik: *Parameter von Funktionen*

$$f(x) = x \cdot \sin(x^2)$$

oder

$$g(x, y) = x^2 + y^2$$

- In der Programmierung unterscheidet man:
  - **Formaler Parameter:**  
Der Name, der bei der Deklaration der (Konstruktor-)Methode gewählt wurde.
  - **Aktueller Parameter:**  
Der Wert, der beim Aufruf für den formalen Parameter eingesetzt wird, wird in Java auch **Argument** genannt.



# Syntax von Konstruktoren

- Klassen-Name(formale Parameter) {  
    // Konstruktor-Rumpf  
}

- Beispiel: Klasse Vector2D

```
class Vector2D {  
    // Attribute: x/y-Koordinaten  
    float x;  
    float y;  
  
    // Konstruktor  
    Vector2D(float x0, float y0) {  
        x = x0;  
        y = y0;  
    }  
}
```

Formale Parameter

Aktuelle Parameter

- Aufruf:

- Beispiel: `new Vector2D(2.0f, 3.0f);`

- Schema: `new Klassen-Name(aktuelle Parameter);`



# this-Referenz

## ■ Vordefinierte Referenz-Variable:

**this** bezeichnet das aktuell zu erstellende Objekt

Gerade gesehen:

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x0, float y0) {  
        x = x0;  
        y = y0;  
    }  
    ...  
}
```

Häufiger verwendete Notation:

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x, float y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

- **this** erlaubt Unterscheidung zwischen Parameter und Attribut
- Bei Namensgleichheit und ohne Verwendung von **this** beziehen sich Variablennamen auf Parameter

# Default-Konstruktor

- Falls kein Konstruktor angegeben wird, ergänzt Java automatisch einen parameterlosen Konstruktor:
  - **Im Beispiel:** `Vector2D() { }` wird erzeugt
- Wie sind dann die Attribute initialisiert?

Typ	Default-Wert
boolean	false
byte, short, int	0
long	0L
float	0.0f
double	0.0
char	'\u0000'
Objekt-Referenz	null

# Mehrere Konstruktoren

- Es können auch mehrere Konstruktoren in einer Klasse deklariert werden:

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x0, float y0) {  
        x = x0;  
        y = y0;  
    }  
  
    Vector2D(float x0) {  
        x = x0;  
        y = 0.0f;  
    }  
}
```

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x0, float y0) {  
        x = x0;  
        y = y0;  
    }  
  
    Vector2D(float x0) {  
        this(x0, 0.0f);  
    }  
}
```

- Sobald man einen Konstruktor explizit angibt, verschwindet der Default-Konstruktor
- **Im Beispiel:** `p = new Vector2D()` ist nicht mehr möglich!

# Jetzt sind Sie gefragt: Konstruktoren

- Schreiben Sie den Konstruktor der Klasse Ball.
- Schreiben Sie eine Main-Methode, die ein neues Objekt vom Typ Ball erzeugt und alle Attribute mit beliebigen sinnvollen Werten initialisiert und dann ausgibt.

```
class Ball {  
    float diameter;  
    int weight;  
    float xPos;  
    float yPos;  
  
    //TODO: Konstruktor  
  
    //TODO: Main-Methode  
}
```

# Jetzt sind Sie gefragt: Konstruktoren

```
class Ball {  
    float diameter; int weight; float xPos; float yPos;  
  
    Ball(float diameter, int weight, Vector2D pos){  
        this.diameter = diameter;  
        this.weight = weight;  
        this.xPos = pos.x;  
        this.yPos = pos.y;  
    }  
  
    public static void main (String[] args){  
        Ball b = new Ball(10f, 5, new Vector2D(10f, 15f));  
        System.out.println("My ball: "+b.diameter+" "+b.weight+" "+b.xPos+" "+b.yPos);  
    }  
}
```

# Methoden

## ■ Methoden

- realisieren das dynamische Verhalten von Objekten
- führen Berechnungen durch

**Beispiel:** Temperatur-Umrechnung:  
 $\text{fahrenheit}(c) = c \cdot 9/5 + 32$

```
double convertToFahrenheit(double celsius) {
    return (celsius * 9.0) / 5.0 + 32.0;
}
```

**Schema:**

```
Rückgabetyp Methoden-Name(formale Parameter) {
    // Methodenrumpf
}
```

**return-Anweisung:**

Verlässt die Methode und liefert den angegebenen Wert als Ergebnis

Schema: **return** Ausdruck;

# Methoden-Deklaration in Java

- Der **Methoden-Kopf** ist die formale Schnittstelle einer Methode
- Sie besteht u.a. aus Methoden-Signatur und Rückgabetyt
  
- Die **Methoden-Signatur** identifiziert eine Methode und besteht aus:
  - dem Namen der Methode
  - der Anzahl der Parameter
  - der Reihenfolge der Parameter
  - den Typen der Parameter

## Beispiel:

```
double convertToFahrenheit(double celsius) {  
    return (celsius * 9.0) / 5.0 + 32.0;  
}
```

- Randbemerkung: In einigen Programmiersprachen ist auch der Rückgabetyt Teil der Signatur



# Methoden-Aufruf

<b>Beispiel:</b>	<b>Temperatur-Umrechnung</b>
<b>Methode:</b>	<pre>double convertToFahrenheit(double celsius) {     return (celsius * 9.0) / 5.0 + 32.0; }</pre>
<b>Aufruf:</b>	<pre>double c = 15, f; f = convertToFahrenheit(c);</pre>
<b>Schema:</b>	Methoden-Name(aktuelle Parameter)
<b>Ergebnis:</b>	Ein Wert des Rückgabetyps

# Beispiel-Methoden

## Beispiel

### Mathematische Notation:

fahrenheit(c) =  $c \cdot 9/5 + 32$   
volumenZylinder(r, h) =  $\pi \cdot r^2 \cdot h$

### Methoden-Deklaration in Java: Fahrenheit-Konvertierung bereits gesehen

```
double areaCircle(double radius) {  
    return Math.PI * radius * radius;  
}  
  
double volumeCylinder(double height, double radius) {  
    return height * areaCircle(radius);  
}
```

### Aufrufe:

- convertToFahrenheit(38.0) liefert als Ergebnis den Wert 100.4
- areaCircle(2.0) liefert als Ergebnis den Wert 12.5664
- volumeCylinder(1.5, 2.0) berechnet  $1.5 \cdot 12.5664$  und liefert daher 18.8496

# Der Rückgabotyp `void`

- Der Rückgabotyp `void` wird verwendet, wenn eine Methode keinen Wert zurückliefern soll.
- Wieso ist so eine Methode sinnvoll?
  - zustandsändernde Methode

## Beispiele:

```
void setXCoordinate(float newX) {  
    this.x = newX;  
}  
  
void alarm(String message) {  
    System.out.println("ALARM: " + message);  
}
```

# Zugriffsfunktionen (getter/setter)

- Unter Zugriffsfunktionen (auch Zugriffsmethoden) versteht man Methoden, die ein Attribut eines Objekts abfragen bzw. ändern
- **Abfragemethoden (Getter):**
  - Dienen der Abfrage von Attributen, z.B.: `float getXCoordinate()` in Klasse `Vector2D`
  - Der Methodename ist meist "get" konkateniert mit dem Attributnamen
    - **Ausnahme:** bei Attributen vom Typ `boolean` verwendet man "is" / "has" anstelle von "get", z.B.: `isMale()` für ein Attribut `boolean male`
- **Änderungsmethoden (Setter):**
  - Dienen dem Setzen / Verändern von Attributwerten, z.B.: `void setXCoordinate(float x)` in Klasse `Vector2D`
  - Der Methodename ist meist "set" plus Attributname
- **Vorteile der Verwendung von Zugriffsfunktionen:**
  - Beim Setzen ist die Prüfung auf gültige Werte möglich
  - Seiteneffekte (z.B. Log-Meldungen) sind möglich
  - Konsistenz bei abhängigen Attributen
  - Leichtere Änderbarkeit von klasseninternen Datenstrukturen

# Bezugsobjekt

- **Normalfall:** Methode ist abhängig vom Zustand des aktuellen Objekts  
 → Beim Aufruf muss das **Bezugsobjekt** angegeben werden

<b>Beispiel:</b> Methode shift in der Klasse Vector2D	<pre>void shift(float dx, float dy) {     this.x = this.x + dx;     this.y = this.y + dy; }</pre>
<b>Aufruf:</b>	<pre>Vector2D p = new Vector2D(3.0f, 4.0f); p.shift(2.0f, 4.0f);</pre>
<b>Aufruf-Schema:</b>	<i>Bezugsobjekt.Methoden-Name (Parameter)</i>

- Ist kein Bezugsobjekt angegeben wird implizit **this** verwendet!
- Bezugsobjekt kann auch Rückgabewert einer anderen Methode sein

<b>Beispiel:</b> Methode in der Klasse Ball	<pre>Vector2D getPosition() {...}</pre>
Möglicher Aufruf:	<pre>Ball b = new Ball(); b.getPosition().shift(2.0f, 4.0f);</pre>

# Was passiert beim Aufruf einer Methode?

## Betrachte Methoden-Aufruf „shift“:

1. Das Bezugsobjekt und dessen Typ wird ermittelt:  
→ Bezugsobjekt ist p vom Typ Vector2D
2. Bestimmung der aufzurufenden Methode aus der ermittelten Klasse:  
→ Verwende Methode shift aus der Klasse Vector2D
3. Die Referenz des ermittelten Bezugsobjekts wird der **this**-Variable der aufgerufenen Methode zugewiesen  
→ **this = p**
4. Die aktuellen Parameter werden den formalen Parametern der Methode zugewiesen:  
→  $dx = -1.0f$  und  $dy = 5.0f$
5. Der Rumpf der Methode wird abgearbeitet
6. Danach wird das Programm direkt nach dem Methodenaufruf fortgesetzt

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x, float y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    void shift(float dx, float dy) {  
        this.x = this.x + dx;  
        this.y = this.y + dy;  
    }  
}
```

### Anweisungen:

```
Vector2D p = new Vector2D(3.0f, 4.0f);  
p.shift(-1.0f, 5.0f);  
System.out.println("My vector: x="+p.x + " y="+p.y);
```

Ausgabe: My vector: x=2.0 y=9.0

# Jetzt sind Sie gefragt: Methoden

- Was gibt der folgende Programmteil aus?

```
Vector2D v = new Vector2D(10f, 11f);  
v.shift(1f, 1f);  
Vector2D p = v;  
p.shift(1f, 1f);  
  
System.out.println("v.x = " + v.x  
    + " and v.y = " + v.y);
```

## Mögliche Antworten

- a) v.x = 10.0 and v.y = 11.0
- b) v.x = 11.0 and v.y = 12.0
- c) v.x = 12.0 and v.y = 13.0

```
//Zur Erinnerung  
void shift(float dx, float dy) {  
    this.x = this.x + dx;  
    this.y = this.y + dy;  
}
```

# Statische Methoden

- Manche Methoden sind **unabhängig vom Zustand** des Objekts, d.h.
    - es findet kein Zugriff auf die Attribute des Objekts statt,
    - auch nicht indirekt über Methodenaufrufe
    - keine Verwendung von **this** möglich
- **Statische Methode (= Klassen-Methode)**

<b>Beispiel:</b>	<pre>class WeatherStation {     static double convertToFahrenheit(double celsius) {         return (celsius * 9.0) / 5.0 + 32.0;     } }</pre>
<b>Schema:</b>	<pre>static Rückgabetyp Methoden-Name(formale Parameter) {     // Methodenrumpf }</pre>
<b>Aufruf:</b>	<pre>Klassen-Name.Methoden-Name(Parameter);</pre>



# Klassen- vs. Objekt-Methode

## Klassen-Methode

```
class WeatherStation {  
  
    static double convertToFahrenheit(double celsius) {  
        return (celsius * 9.0) / 5.0 + 32.0;  
    }  
  
}
```

Anweisungen:

```
WeatherStation.convertToFahrenheit(15.0);
```

## Objekt-Methode

```
class Vector2D {  
    float x;  
    float y;  
  
    Vector2D(float x0, float y0) {  
        x = x0;  
        y = y0;  
    }  
  
    void shift(float dx, float dy) {  
        this.x = this.x + dx;  
        this.y = this.y + dy;  
    }  
}
```

Anweisungen:

```
Vector2D p = new Vector2D(3.0f, 4.0f);  
p.shift(2.0f, 4.0f);
```

# Klassen- vs. Objekt-Methode

## Objektmethode:

- verwendet **Bezugsobjekt**
- ohne expliziten Bezug wird `this` als Bezugsobjekt verwendet
- **Beispiel:** `p.shift(2.0f, 4.0f);`

## Klassenmethode:

- verwendet **Bezugsklasse**
- wird mit dem Schlüsselwort `static` deklariert
- kein Zugriff auf Attribute der Klasse möglich
- `this`-Referenz kann im Klassenrumpf nicht verwendet werden
- **Beispiel:** `WeatherStation.convertToFahrenheit(15.0);`

Klassenmethoden  
werden in Eclipse  
kursiv angezeigt

- **Klassenmethoden sollten sparsam verwendet werden!**

# Lokale Variablen

## ■ Lokale Variablen

- dienen als Hilfsspeicher für Zwischengrößen in Methoden
- werden auf dem sogenannten **Stack** (*Stapel*) gespeichert
- existieren nur, solange die Methode abgearbeitet wird

### Beispiel (Berechnung Kreisfläche)

```
double areaCircle(double radius) {  
    return Math.PI * radius * radius;  
}
```

mit lokaler Variable:

```
double areaCircle(double radius) {  
    double radiusSquared = radius * radius;  
    return Math.PI * radiusSquared;  
}
```

**Achtung:** lokale Variablen werden nicht automatisch initialisiert!

D.h. sie haben einen undefinierten Anfangswert, sofern keine explizite Initialisierung vorgenommen wird.

# Parameter als lokale Variablen

- **Parameter** werden in Java behandelt wie **lokale Variablen**, die beim Methodenaufruf initialisiert werden

## Beispiel:

```
int foo(int a) {  
    int x = a + 1;  
    a = x * x;           // schlechter Programmierstil!  
    return a + x;  
}
```

# Methodenaufrufe und Objektreferenzen

## Achtung bei Objektreferenzen als Methodenparameter:

- Beim Methodenaufruf werden die aktuellen Parameter auf den **Objektreferenzwert** gesetzt
- D.h. die **Objektidentität** wird **nicht kopiert!**
- Dadurch ist es möglich, den Objektzustand in einer Methode zu verändern; der geänderte Objektzustand bleibt auch nach Abarbeitung der Methode erhalten

## Beispiel:

```
class ParameterTest {
    static void func(int y) {
        y = 5;
    }
    static void foo(Vector2D q) {
        q.x = 5.0f;
    }

    public static void main(String[] args) {
        int x = 7;
        func(x);
        // x ist immer noch 7!

        Vector2D p = new Vector2D(1.0f, 2.0f);
        foo(p);
        // p.x ist jetzt 5.0!
    }
}
```

# Lokale Variablen vs. Attribute

	<b>Lokale Variable</b>	<b>Attribut</b>
<b>Deklaration</b>	innerhalb von Methoden	außerhalb von Methoden
<b>Lebensdauer</b>	Methoden-Aufruf	Lebensdauer des zugehörigen Objekts
<b>Zugänglichkeit</b>	nur innerhalb einer Methode	für alle Methoden der Klasse
<b>Zweck</b>	Zwischenspeicher für Werte	Zustand des Objekts

# Klassenvariablen (Statische Attribute)

- Das Schlüsselwort **static** kann auch für Attribute verwendet werden und zeigt dann eine **Klassenvariable** (auch *statisches Attribut* genannt) an, z.B.:

```
static int nrObjects = 0;
```

- Klassenvariablen

- gehören zur Klasse und nicht zum Objekt
- werden nur einmal für alle Objekte einer Klasse angelegt
- sind während der gesamten Laufzeit des Programms verwendbar
- sollten immer direkt bei der Deklaration initialisiert werden
- sollten sparsam verwendet werden!

- Zugriff auf Klassenvariable mit: **Bezugsobjekt.Name** oder **Klasse.Name** (bevorzugt)

```
class Ball {
    // class variable static
    static int nrBalls = 0;
    // attribute
    double size;

    // constructor
    Ball(double size) {
        this.size = size;
        nrBalls++;
    }

    // main method
    public static void main(String[] args) {
        System.out.println(Ball.nrBalls);
        Ball b1 = new Ball(4.0);
        Ball b2 = new Ball(1.2);
        System.out.println(Ball.nrBalls);

        // works, but not recommended:
        System.out.println(b1.nrBalls);
    }
}
```

Klassenvariablen werden in Eclipse ebenfalls kursiv angezeigt

# Überladen von Methoden

- In Java kann es verschiedene Methoden mit **dem gleichen Namen** geben, wenn sich diese in den *Typen und/oder der Anzahl ihrer Parameter* (d.h. in ihrer *Signatur*) unterscheiden.
- **Beispiel (Klasse Vector2D):**

```

void rotate(float angle) {
    float phi = (float) Math.toRadians(angle);
    float xOld = this.x;
    float yOld = this.y;
    this.x = (float) (xOld * Math.cos(phi) - yOld * Math.sin(phi));
    this.y = (float) (xOld * Math.sin(phi) + yOld * Math.cos(phi));
}

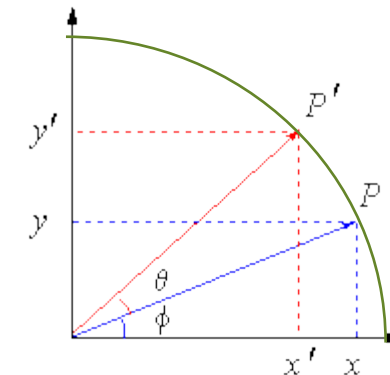
```

Typecast-Operator, da Math-Methoden double zurückgeben

```

void rotate(Vector2D center, float angle) {
    float xOfCenter = center.x;
    float yOfCenter = center.y;
    this.shift(-xOfCenter, -yOfCenter);
    this.rotate(angle);
    this.shift(xOfCenter, yOfCenter);
}

```



Um einen anderen Punkt drehen:  
verschieben, um Ursprung drehen, verschieben

- Auch Konstruktoren können überladen werden (hatten wir bereits gesehen)
- Methoden gleichen Namens, die sich nur im Rückgabetyt unterscheiden, sind nicht möglich



# Hilfsmethoden

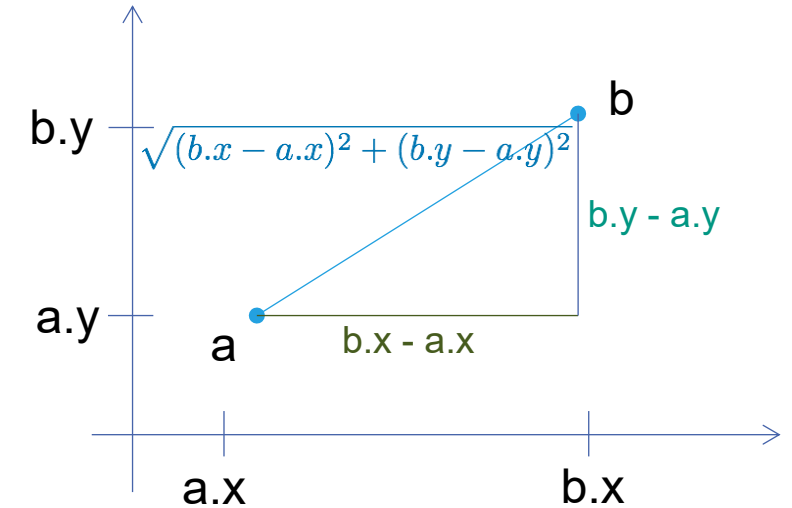
- **Beispiel:** Berechne die Länge einer Linie

```
class Line {
    Vector2D a; // start
    Vector2D b; // end
}
```

- **Mathematische Lösung:**  $\sqrt{(b.x - a.x)^2 + (b.y - a.y)^2}$

- **Methode in Java:**

```
double length() {
    return Math.sqrt((b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y));
}
```



Basis für die Grafik:  
<http://scienceblogs.de/mathlog/2008/04/04/topologie-von-flachen-viii/>

# Hilfsmethoden

- **Beispiel:** Berechne die Länge einer Linie

```
class Line {  
    Vector2D a; // start  
    Vector2D b; // end  
}
```

- **Mathematische Lösung:**  $\sqrt{(b.x - a.x)^2 + (b.y - a.y)^2}$

- **Methode in Java (besser):**

```
double length() {  
    return Math.sqrt(square(b.x - a.x) + square(b.y - a.y));  
}
```

- **Mit Hilfsmethode:**

```
double square(double x) {  
    return x * x;  
}
```

→ Wichtig: Keinen Code duplizieren!  
Besser Hilfsmethoden einführen.

# Die Methode `main`

- Was passiert beim Ausführen eines Java-Programms?
  - **Aufruf auf Kommandozeile:** `java Klasse`
- `main`-Methode von *Klasse* wird ausgeführt
- Signatur: `public static void main(String[] args)`
- `args`:
  - Kommandozeilen-Argumente
  - **Ausblick:** `args` ist eine Array-Variable: `args[0]`, `args[1]`, ...

## Beispiel:

```
java MyAdder 2.0 3.5  
args[0] = "2.0"  
args[1] = "3.5"
```

# Methoden zur Ein- und Ausgabe

■ Java hat verschiedene Methoden zur Ein- und Ausgabe bereits „eingebaut“, z.B.:

- `System.out.println(String);`
- `scanner.nextLine();` – Rückgabewert `String`
- `scanner.nextInt();` – Rückgabewert `int`
- `Integer.parseInt(String);` – Rückgabewert `int`
- `Double.parseDouble(String);` – Rückgabewert `double`

<b>Beispiel:</b>	<pre> class MyAdder {     public static void main(String[] args) {         double x, y, z;         x = Double.parseDouble(args[0]);         y = Double.parseDouble(args[1]);         z = x + y;         System.out.println(z); // automatische Konvertierung     } } </pre>
<b>Aufruf:</b>	<code>java MyAdder 2.0 3.5</code>
<b>Ausgabe:</b>	<code>5.5</code>

# Typische Struktur eines Java-Programms

## ■ Beispiel: Billard-Spiel

```
class Billard {  
    // main method  
    public static void main(String[] args) {  
        Table table = new Table(...);  
        Ball ball = new Ball(table, Ball.Color.RED);  
        table.addBall(ball);  
        ...  
        table.startSimulation();  
    }  
}
```

```
class Table {  
    // constructor  
    Table(...) { ... }  
  
    // methods  
    void addBall(Ball b) { ... }  
    void startSimulation() { ... }  
    ...  
}
```

```
class Ball {  
    // enum for ball color  
    enum Color { RED, BLUE, WHITE, ... }  
  
    // attributes  
    Table t;  
    Vector2D position;  
  
    // constructor  
    Ball(Table t, Color c) { ... }  
  
    // methods  
    void setPosition(Vector2D p) { ... }  
    ...  
}
```

# Jetzt sind Sie gefragt: Methodenrätsel

Betrachten Sie den Programmausschnitt rechts. Was können Sie über die Signaturen der Methoden foo, bar, baz und bom aussagen und in welchen Klassen sind sie deklariert?

```

class Unknown { ... }

class Quiz2 {
  Unknown u; int x; boolean b;
  ...
  void test() {
    b = u.foo().bar(u);
    x = u.foo().foo().bom();
    u = baz(u.bar(u));
  }
}
  
```

foo gehört zur Klasse  und hat die Signatur  foo()

bar gehört zur Klasse  und hat die Signatur  bar()

baz gehört zur Klasse  und hat die Signatur  baz()

bom gehört zur Klasse  und hat die Signatur  bom()

# Zusammenfassung

## Konstruktoren:

- Fassen Objekterzeugung und Initialisierung der Attribute zusammen
- Syntax: `Klassen-Name(formale Parameter) { ... }`
- `this`-Referenz erlaubt Ansprechen der Attribute bei Namensgleichheit

## Methoden:

- Zweck:
  - (a) realisieren das dynamische Verhalten von Objekten
  - (b) führen Berechnungen durch
- Klassen- vs. Objekt-Methoden
- Lokale Variablen dienen als Speicher für Zwischenergebnisse
- Ergebnis einer Methode wird mit `return` zurückgegeben

# Literaturhinweis – Weiterlesen

- Dietmar Ratz, Jens Scheffler, Detlef Seese und Jan Wiesenberger "Grundkurs Programmieren in Java", 7. Auflage, 2014 (mit Java 8), Hansa-Verlag
  - "Methoden und Unterprogramme"
  - "Der grundlegende Umgang mit Klassen"