# Kapitel 5

# Objekte und Klassen

# Ziele

- Grundbegriffe objektorientierter Programmierung kennenlernen
- Klassen in Java deklarieren können
- Das Speichermodell für Objekte verstehen
- Typen, Ausdrücke und Anweisungen im Kontext von Klassendeklarationen erweitern
- Die Klasse `String` kennenlernen

# Überblick Kapitel 3 - 5

*Kapitel 3*

| | | *Kapitel 5* |
|---|---|---|
| | | Klassendeklarationen |
| | | Objekte und Objekthalde (Heap) |
| Grunddatentypen | erweitert um | Klassentypen |
| Werte | erweitert um | Referenzen und `null` |
| Operationen | erweitert um | `==`, `!=` für Referenzen und `null` |
| Ausdrücke | erweitert um | Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck |
| Typisierung | | |
| Auswertung bzgl. Zustand (Stack) | erweitert um | Objekthalde (Heap) |

*Kapitel 4*

| | | |
|---|---|---|
| Kontrollstrukturen | erweitert um | Return-Anweisung, Methodenaufruf, Objekterzeugung |

# Objektorientierte Programmierung

- In der objektorientierten Programmierung werden **Daten und Methoden**, die Algorithmen implementieren, zu geschlossenen Einheiten (**Objekten**) zusammengefasst.
- **Beispiele**:
  - Bankkonto
    Daten: Kontostand, Zinssatz; Methoden: einzahlen, abheben, …
  - Punkte, Linien, Kreise in einem Zeichenprogramm
    Daten: geometrische Form; Methoden: verschieben, rotieren, …
- Ein objektorientiertes System besteht aus einer Menge von Objekten, die Methoden bei anderen Objekten (oder bei sich selbst) aufrufen. Die Ausführung einer Methode führt häufig zu einer Änderung der gespeicherten Daten (**Zustandsänderung**).

# Objekte und Klassen

- Objekte → Laufzeit
  - Objekte speichern Informationen (Daten).
  - Objekte können Methoden ausführen zum Zugriff auf diese Daten und zu deren Änderung.
  - Während der Ausführung einer Methode kann ein Objekt auch Methoden bei (anderen) Objekten aufrufen.
- Klassen → Programmierzeit
  - Klassen definieren die charakteristischen Merkmale von Objekten einer bestimmten Art: **Attribute**, **Methoden** (und deren Algorithmen).
  - Jede Klasse kann Objekte derselben Art **erzeugen**.
  - Jedes Objekt gehört zu genau einer Klasse; es ist **Instanz** dieser Klasse.

# Beispiel: Klasse „Point“

```
public class Point {

        private int x,y;

        public Point(int x0, int y0){
                this.x = x0;
                this.y = y0;
        }

        public void move(int dx, int dy){
                this.x = this.x + dx;
                this.y = this.y + dy;
        }

        public int getX(){
                return this.x;
        }

        public int getY(){
                return this.y;
        }
}
```

2 Attribute beide vom Typ int

Konstruktor

3 Methoden

# Beispiel: Klasse „Point“

```
public class Point {

        private int x,y;

        public Point(int x0, int y0){
                this.x = x0;
                this.y = y0;
        }

        public void move(int dx, int dy){
                this.x = this.x + dx;
                this.y = this.y + dy;
        }

        public int getX(){
                return this.x;
        }

        public int getY(){
                return this.y;
        }
}
```

Klassenname

2 Attribute (Felder, Instanzvariable) vom Typ int

formale Parameter

Konstruktordeklaration

vordefinierte lokale Variable `this`; bezeichnet das „gerade aktive“ Objekt

Zugriff auf y-Koordinate von `this`

Methode ohne Ergebnis

Ergebnistyp

Return-Anweisung mit Ergebniswert

Methode mit Ergebnis vom Typ int

# Mit Javadoc kommentierte Klasse „Point"

Zur Erzeugung von Kommentaren zu Klassen, Konstruktoren, Methoden, …

```
/**
 * Diese Klasse repraesentiert einen Punkt in der Ebene.
 * @author Prof. Dr. Hennicker
 */
public class Point {
        private int x, y;

         /**
         * Konstruktor eines Punkts,
         * wobei dessen x- und y-Koordinate gegeben sein muessen.
         * @param x0
         *    x-Koordinate des Punkts
         * @param y0
         *    y-Koordinate des Punkts
         */
        public Point(int x0, int y0) {
            this.x = x0;
            this.y = y0;
        }
```

```
    /**
     * Diese Methode versetzt den Punkt um dx auf der x-Achse und dy auf der y-Achse.
     * @param dx
     *      gibt an, um wieviel der Punkt auf der x-Achse versetzt werden soll
     * @param dy
     *      gibt an, um wieviel der Punkt auf der y-Achse versetzt werden soll
     */
    public void move(int dx, int dy) {
        this.x = this.x + dx;
        this.y = this.y + dy;
    }
    /**
     * Diese Methode gibt die x-Koordinate des Punkts zurueck
     * @return die x-Koordinate des Punkts
     */
    public int getX() {
        return this.x;
    }
    /**
     * Diese Methode gibt die y-Koordinate des Punkts zurueck
     * @return die y-Koordinate des Punkts
     */
    public int getY() {
        return this.y;
    }
}
```

# Ansicht der Dokumentation

wichtig für die Wiederverwendung von Klassen

## Class Point

java.lang.Object
    vorlesung05.Point

---

```
public class Point
extends java.lang.Object
```

Diese Klasse repraesentiert einen Punkt in der Ebene.

**Author:**

Prof. Dr. Hennicker

### Constructor Summary

| Constructors | |
|---|---|
| **Constructor and Description** | |
| `Point(int x0, int y0)` | Konstruktor eines Punkts, wobei dessen x- und y-Koordinate gegeben sein muessen. |

## Method Summary

Methods

| Modifier and Type | Method and Description |
|---|---|
| int | getX()<br>Diese Methode gibt die x-Koordinate des Punkts zurueck |
| int | getY()<br>Diese Methode gibt die y-Koordinate des Punkts zurueck |
| void | move(int dx, int dy)<br>Diese Methode versetzt den Punkt um dx auf der x-Achse und dy auf der y-Achse. |

### Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Point

```
public Point(int x0,
      int y0)
```

Konstruktor eines Punkts, wobei dessen x- und y-Koordinate gegeben sein muessen.

**Parameters:**

x0 - x-Koordinate des Punkts

y0 - y-Koordinate des Punkts

## Method Detail

### move

```
public void move(int dx,
                 int dy)
```

Diese Methode versetzt den Punkt um dx auf der x-Achse und dy auf der y-Achse.

**Parameters:**

dx - gibt an, um wieviel der Punkt auf der x-Achse versetzt werden soll

dy - gibt an, um wieviel der Punkt auf der y-Achse versetzt werden soll

### getX

```
public int getX()
```

Diese Methode gibt die x-Koordinate des Punkts zurueck

**Returns:**

die x-Koordinate des Punkts

### getY

```
public int getY()
```

Diese Methode gibt die y-Koordinate des Punkts zurueck

**Returns:**

die y-Koordinate des Punkts

# Erzeugung der Dokumentation

Mit dem Befehl

```
javadoc Point.java
```

wird automatisch eine Beschreibung der Klasse `Point` erzeugt und in die Datei

```
Point.html
```

geschrieben.

# Spezielle Tags für Javadoc

- `@see` für Verweise
- `@author` für Namen des Autors
- `@version` für die Version
- `@param` für die Methodenparameter
- `@return` für die Ergebniswerte von Methoden

# Beispiel: Klasse „Line“ benützt die Klasse „Point“

```
public class Line {
    private Point start;
    private Point end;
    public Line(Point s, Point e){
        this.start = s;
        this.end = e;
    }
    public void move(int dx, int dy){
        this.start.move(dx,dy);
        this.end.move(dx,dy);
    }
    public double length(){
        int startX = this.start.getX();
        int endX = this.end.getX();
        int diffX = Math.abs(startX - endX);

        int startY = this.start.getY();
        int endY = this.end.getY();
        int diffY = Math.abs(startY - endY);
        //oder int diffY = Math.abs(this.start.getY()-this.end.getY());
        return Math.sqrt(diffX * diffX + diffY * diffY));
    }
}
```

} 2 Attribute

Aufruf der Methode "move" der Klasse Point für den Startpunkt Endpunkt

start.x geht nicht, weil x private ist in der Klasse Point

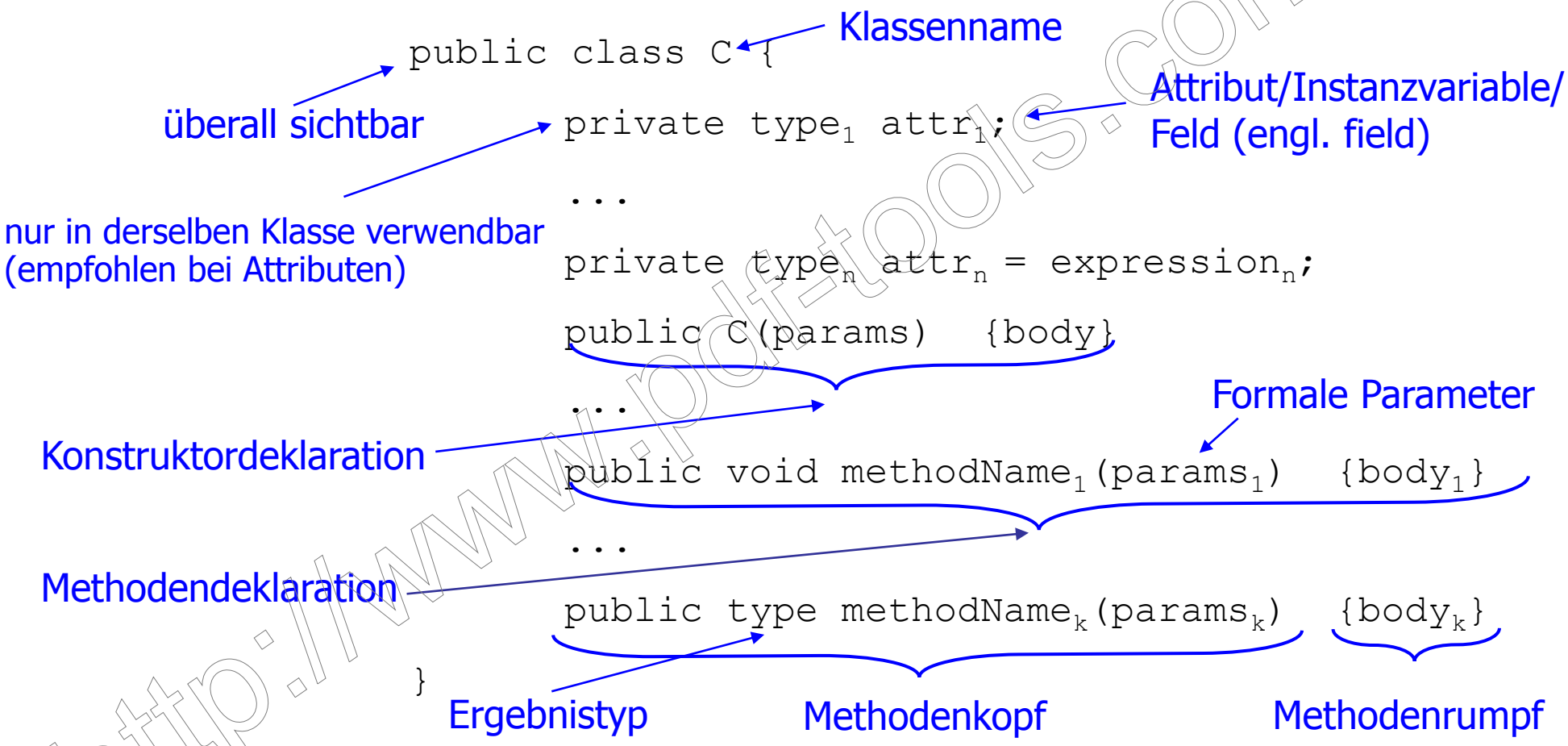
# Klassendeklarationen in Java (ohne Vererbung)

```
public class C {
        private type_1 attr_1;
        ...
        private type_n attr_n = expression_n;
        public C(params)  {body}
        ...
        public void methodName_1(params_1)   {body_1}
        ...
        public type methodName_k(params_k)   {body_k}
}
```

**Beachte:**

1. In einer Datei kann höchstens eine „public" Klasse deklariert sein.
2. Zu jeder konkreten Klasse `C` gibt es einen vordefinierten Standardkonstruktor `C()`.
3. Der Ergebnistyp einer Methode kann auch leer sein, dargestellt durch `void`.

# Klassendeklarationen in Java (ohne Vererbung)



# Grammatik für Klassendeklarationen (ohne Vererbung)

*ClassDeclaration* = ["public"] "class" *Identifier ClassBody*

*ClassBody* = "{"{*FieldDeclaration* | *ConstructorDeclaration* | *MethodDeclaration*}"}"

*FieldDeclaration* = [*Modifier* ] *VariableDeclaration*

*Modifier* = "public " | "private"

*MethodDeclaration* = *Header Block*

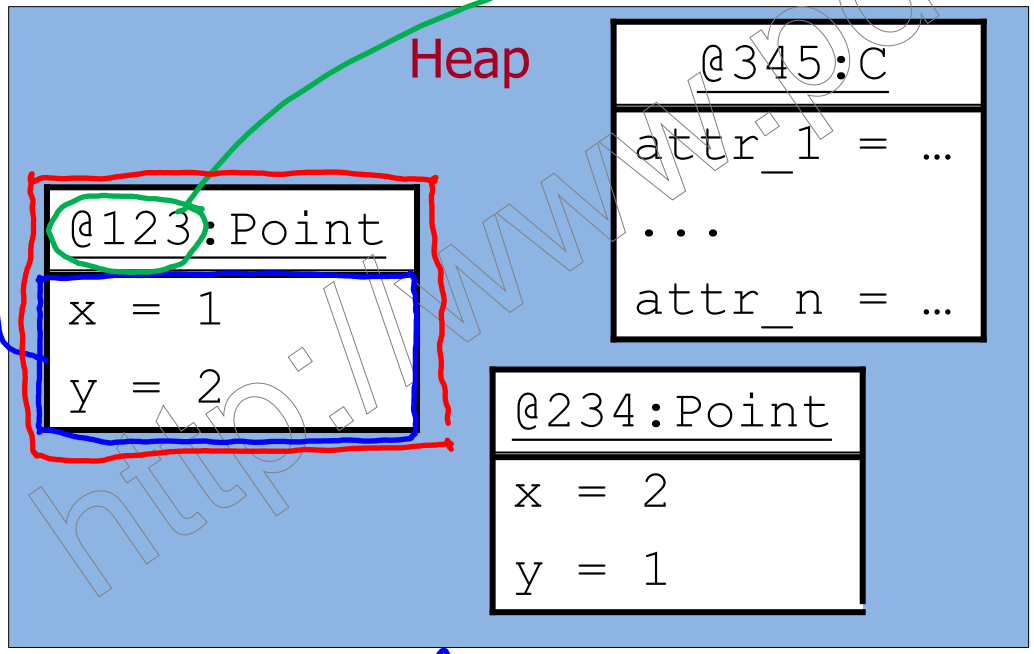
*Header* = [*Modifier* ] (*Type* | "void") *Identifier* "(" [*FormalParameters*] ")"

*FormalParameters* = *Type Identifier* {"," *Type Identifier*}

- *ConstructorDeclaration* ist wie *MethodDeclaration*, jedoch ohne (*Type* | "void") im *Header*. Der *Identifier* im *Header* muss hier gleich dem Klassennamen sein.
- Methoden, deren *Header* einen Ergebnistyp *Type* hat, nennt man Methoden mit Ergebnis(typ).

# Objekte und ihre Speicherdarstellung

- Ein Objekt ist ein Behälter mit einer eindeutigen **Objektidentität** (Adresse), unter der man die Daten (Attributwerte) des Objekts findet => **Objektzustand**.
- Die aktuell während eines Programmlaufs existierenden Objekte werden mit ihrem aktuellen Zustand auf einem **Heap** („Halde") abgelegt.




**Abstrakte Darstellung des Heaps:**

```
{ <(@123,Point),[(x,1),(y,2)]>,
  <(@234,Point),[(x,2),(y,1)]>,
  <(@345,C),[(attr_1,…), ... ,(attr_n,…)]>
}
```

grafische Notation für Objekte a la UML

# Wdh: Überblick Kapitel 3 - 5

| | | *Kapitel 5* |
|---|---|---|
| | | Klassendeklarationen |
| *Kapitel 3* | | Objekte und Objekthalde (Heap) |
| Grunddatentypen | erweitert um | **Klassentypen** |
| Werte | erweitert um | **Referenzen und `null`** |
| Operationen | erweitert um | **`==`, `!=` für Referenzen und `null`** |
| Ausdrücke | erweitert um | Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck |
| Typisierung | | |
| Auswertung bzgl. Zustand (Stack) | erweitert um | **Objekthalde (Heap)** |
| *Kapitel 4* | | |
| Kontrollstrukturen | erweitert um | Return-Anweisung, Methodenaufruf, Objekterzeugung |

# Klassentypen

Im Folgenden werden die in Kapitel 3 und 4 eingeführten Konzepte für *Typen*, *Ausdrücke* und *Anweisungen* **erweitert**. (Eine nochmalige Erweiterung erfolgt später bei der Einführung von Arrays.)

*Type* = *PrimitiveType* | ***ClassType*** (← neu)

*ClassType* = *Identifier*

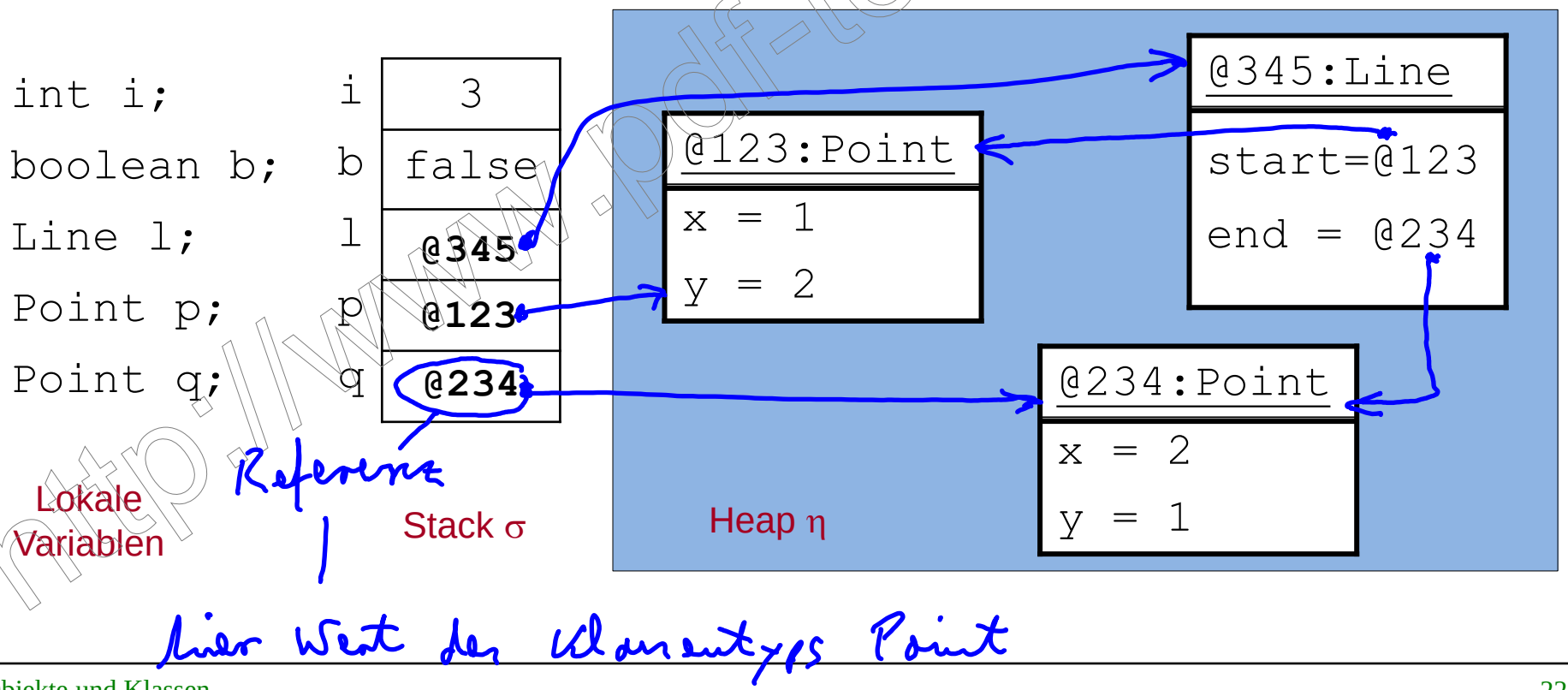
- Mit jeder Klassendeklaration wird ein neuer Typ eingeführt (**Klassentyp**), der den Namen der Klasse hat.
- Die **Werte** eines Klassentyps sind **Referenzen** (Verweise, Zeiger, Links) auf Objekte der Klasse sowie das Element `null` („leere" Referenz).
- Dementsprechend speichern lokale Variable eines Klassentyps Referenzen auf Objekte oder den Wert `null`.
- Objekt-Referenzen können mit den Operationen `==` und `!=` auf Gleichheit bzw. Ungleichheit getestet werden.

**Achtung:** Objekte einer Klasse K ≠ Werte des Klassentyps K.

# Zustand = Stack + Heap

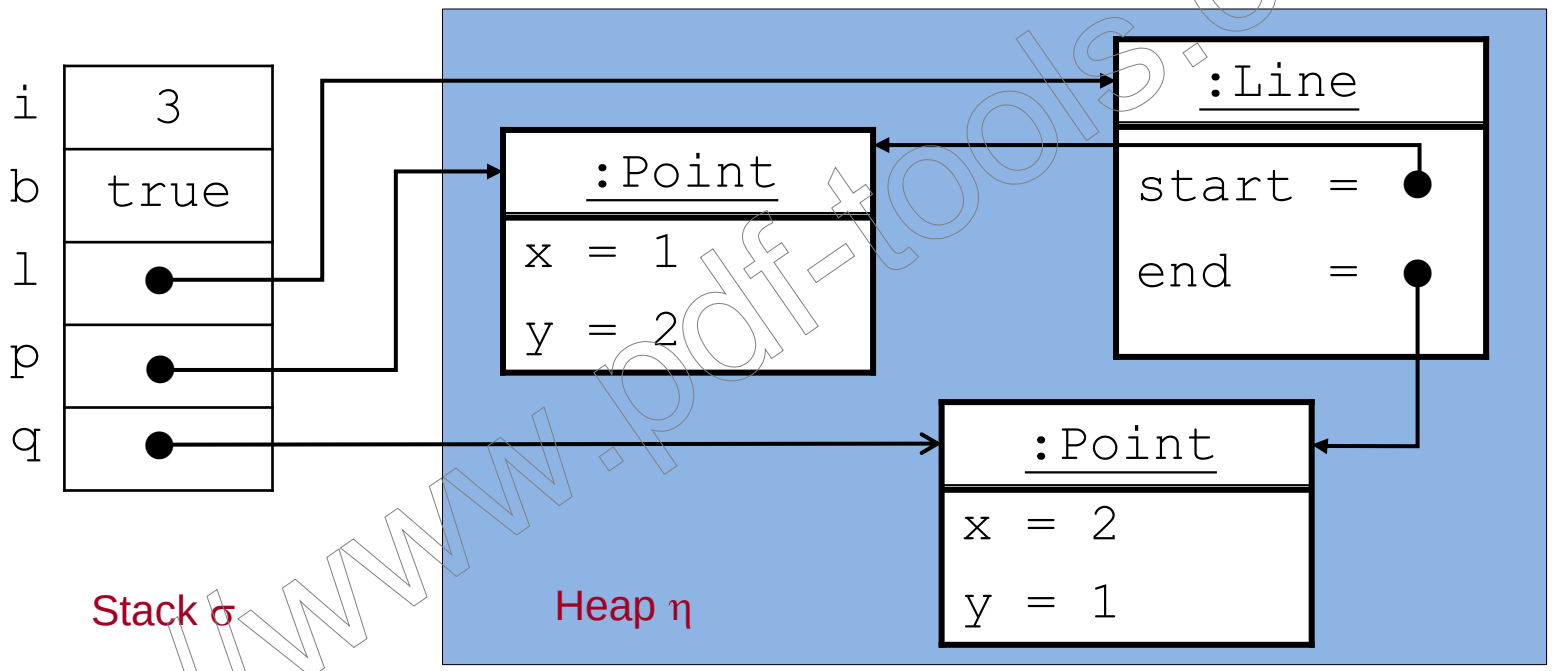
Ein Zustand ($\sigma$,$\eta$) eines objektorientierten Java-Programms besteht aus

- einem Stack (Keller) $\sigma$ für die lokalen Variablen und
- einem Heap (Halde) $\eta$ für die aktuell existierenden Objekte



# Zustand mit Zeigerdarstellung

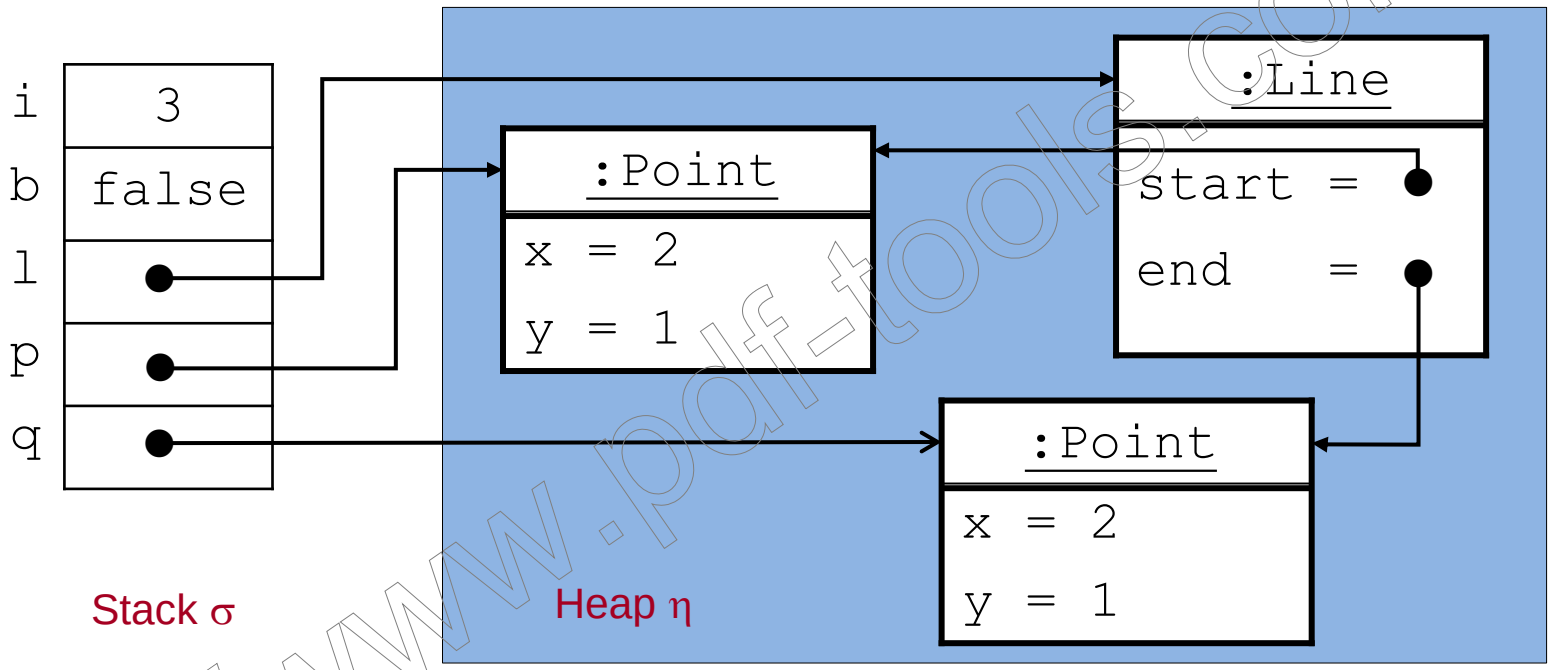
abstrahiert von den konkreten Adressen




**Beachte:**

Der Attributwert eines Objekts kann selbst wieder ein Verweis auf ein (anderes) Objekt sein.

# Gleichheit von Objektreferenzen




(σ,η)

field access

(p==q) $=_{(\sigma,\eta)}$ false, (p!=q) $=_{(\sigma,\eta)}$ true, (q==l.end) $=_{(\sigma,\eta)}$ true

Ausdruck vom Typ boolean

# Wdh: Überblick Kapitel 3 - 5

| *Kapitel 3* | | *Kapitel 5* |
|---|---|---|
| | | Klassendeklarationen |
| | | Objekte und Objekthalde (Heap) |
| Grunddatentypen | erweitert um | Klassentypen |
| Werte | erweitert um | Referenzen und `null` |
| Operationen | erweitert um | `==`, `!=` für Referenzen und `null` |
| Ausdrücke | erweitert um | **Attributzugriff**, **Methodenaufruf mit Ergebnis**, **Objekterzeugungsausdruck** |
| Typisierung | | |
| Auswertung bzgl. Zustand (Stack) | erweitert um | **Objekthalde (Heap)** |
| *Kapitel 4* | | |
| Kontrollstrukturen | erweitert um | Return-Anweisung, Methodenaufruf, Objekterzeugung |

# Erweiterte Grammatik für Ausdrücke im Kontext von Klassendeklarationen

*Expression* = *Variable* | *Value* | *Expression BinOp Expression* |
*UnOp Expression* | "(" *Expression* ")" |
***MethodInvocation*** | (← neu)
***InstanceCreation*** (← neu)

*Variable* = *NamedVariable* |
***FieldAccess*** (← neu)

*NamedVariable* = *Identifier*

*FieldAccess* = *Expression* "." *Identifier* (← neu)

p.x

*Value* = *IntegerValue* | *FloatingPointValue* | *CharacterValue* | *BooleanValue* |
"**null**" (← neu)

# Grammatik für Methodenaufruf- und Objekterzeugungs-Ausdrücke

*MethodInvocation* =
*Expression* "." *Identifier* "(" [*ActualParameters*] ") "

p.getX()

*ActualParameters* = *Expression* {"," *Expression*}

*InstanceCreation* = *ClassInstanceCreation*

*ClassInstanceCreation* =
"new" *ClassType* "(" [*ActualParameters*] ") "

new Point(-7, 8+1)
↑ ↑
aktuelle Parameter

# Typ und Auswertung der neuen Ausdrücke

- Ein Ausdruck ist (wie bisher) **typkorrekt**, wenn ihm ein Typ zugeordnet werden kann.
- Die **Auswertung** eines Ausdrucks `e` erfolgt (jetzt) unter einem **Zustand** $(\sigma,\eta)$, d.h. wir berechnen $e =_{(\sigma,\eta)}$ ...
- Der Attributzugriff mit "." und der Methodenaufruf mit "." haben die höchste Präzedenz 15.

Wir bestimmen nun Regeln für Typkorrektheit und Auswertung für jeden neu hinzugekommenen Ausdruck.

"**null**" :

`null` ist ein Wert, dessen (namenloser) Typ passend zu jedem Klassentyp ist.

# Attributzugriff

*FieldAccess* = *Expression* "." *Identifier*

- Der Ausdruck *Expression* muss einen Klassentyp haben und der *Identifier* muss ein Attribut der Klasse (oder einer Oberklasse, vgl. später) bezeichnen.
- Das Attribut muss im aktuellen Kontext sichtbar sein.
- *FieldAccess* hat dann denselben Typ wie das Attribut *Identifier*.

Beispiel:

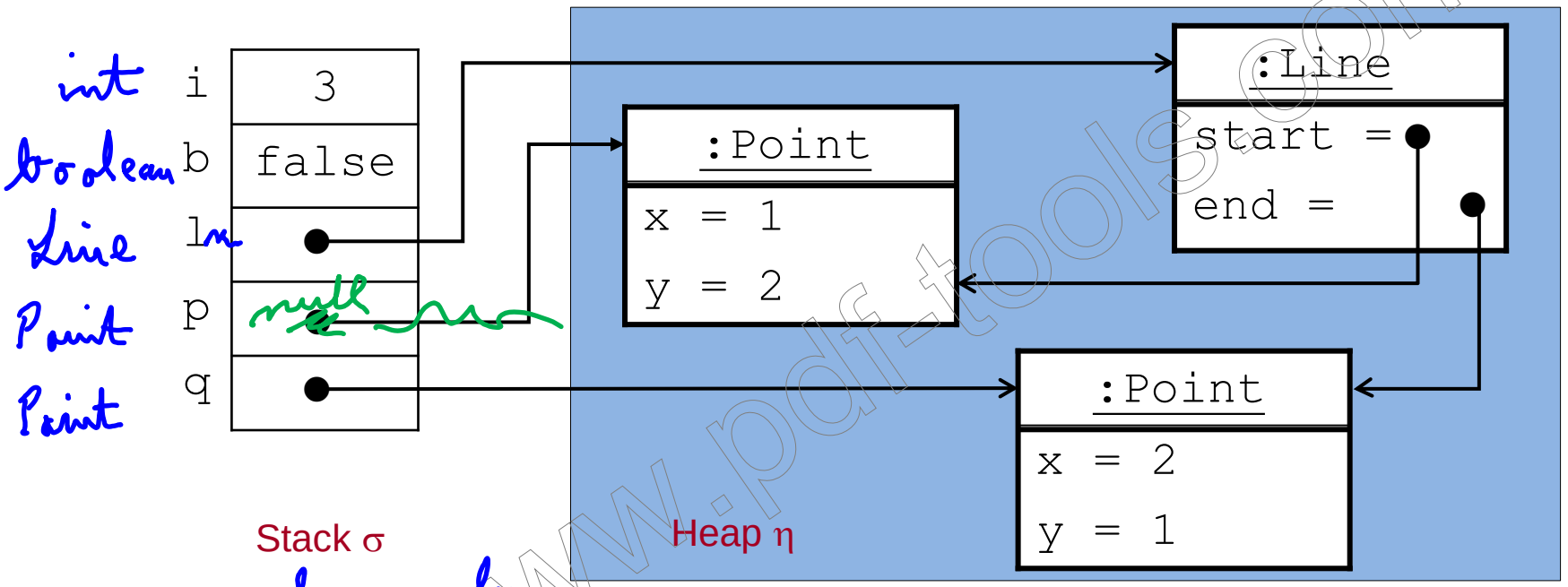
Seien `Point p; Line ln;` lokale Variable.

`p.x` hat den Typ `int`,

`ln.start` hat den Typ `Point`,

`ln.start.y` hat den Typ `int`.

# Attributzugriff: Auswertung




- `p.x`, `q.y`, `l.end`, `l.end.x`, … sind **Variablen**, deren Werte in einem Zustand $(\sigma,\eta)$ die Attributwerte der referenzierten Objekte sind.
- `p.x` $=_{(\sigma,\eta)}$ `1`, `q.y` $=_{(\sigma,\eta)}$ `1`, `l.end.x` $=_{(\sigma,\eta)}$ `2`, …

**Achtung:**

- Falls kein Objekt referenziert wird, z.B. falls `p` $=_{(\sigma,\eta)}$ `null`, dann erfolgt bei der Auswertung von `p.x` ein Laufzeitfehler.

# Methodenaufruf-Ausdruck

*MethodInvocation* = *Expression* "." *Identifier* "(" [*ActualParameters*] ")"

*ActualParameters* = *Expression* {"," *Expression*}

Ein Methodenaufruf-Ausdruck hat also die Form `e.m(a1,…,an)`

- Der Ausdruck `e` muss einen Klassentyp `C` haben und der Identifier `m` muss eine in der Klasse `C` (oder einer Oberklasse von `C`, vgl. später) deklarierte Methode **mit Ergebnis** bezeichnen:

  ```
  Type m (T1 x1,…,Tn xn) {body}
  ```

- Die **aktuellen Parameter** $a_1,\ldots,a_n$ sind Ausdrücke, die in Anzahl und Typ zu den formalen Parametern der Methodendeklaration passen müssen.

- Der Ausdruck $e.m(a_1,\ldots,a_n)$ hat dann als Typ den Ergebnistyp der Methode.

# Methodenaufruf-Ausdruck: Beispiele und Auswertung

Seien `Point p; Line l;` lokale Variable.

`p.getX()` hat den Typ `int`,

`l.start.getY()` den Typ `int`.

Typ Point

Sei $(\sigma,\eta)$ der Zustand von oben.

`p.getX()` $=_{(\sigma,\eta)}$ `1`,

`l.start.getY()` $=_{(\sigma,\eta)}$ `2`.

**Bemerkungen:**

- Die Berechnung der Ergebnisse von Methodenaufrufen basiert auf der Ausführung von Methodenrümpfen (vgl. unten).
- Im allgemeinen ist es möglich, dass der Aufruf einer Methode mit Ergebnistyp nicht nur einen Ergebniswert liefert sondern auch eine Zustandsänderung bewirkt. "Seiteneffekt"

# Objekterzeugungs-Ausdruck

*ClassInstanceCreation* = "new" *ClassType* "(" [*ActualParameters*] ") "

Eine Objekterzeugungs-Ausdruck hat also die Form `new C(a1,…,an)`

- `C` muss eine deklarierte Klasse sein.
- Wenn die aktuelle Parameterliste nicht leer ist, muss in der Klasse `C` ein Konstruktor definiert sein mit `n` formalen Parametern:

```
C (T1 x1,…,Tn xn) {body}
```

- Die aktuellen Parameter $a_1,\ldots,a_n$ sind Ausdrücke, deren Typen zu den Typen $T_1,\ldots,T_n$ passen müssen.
- Der Ausdruck `new C(a1,…,an)` hat dann den Typ `C`.

**Beachte:**

Zu jeder Klasse `C` gibt es implizit einen Standard-Konstruktor `C()` ohne Parameter.

# Objekterzeugungs-Ausdruck: Beispiele und Auswertung

Sei `int i;` eine lokale Variable.

`new Point()` hat den Typ `Point`,

`new Point(1,2)` hat den Typ `Point`,

`new Point(1,i)` hat den Typ `Point`,

`(new Point(1,i)).getX()` hat den Typ `int`.

Mit dem Ausdruck `new Point()` wird

1. ein neues Objekt der Klasse `Point` erzeugt und auf den Heap gelegt,
2. die Felder des Objekts mit Defaultwerten initialisiert (`0` bei `int`, `false` bei `boolean`, `null` bei Klassentypen),
3. eine Referenz auf das neu erzeugte Objekt als **Ergebniswert** geliefert.

Mit dem Ausdruck `new Point(1,2)` wird der Rumpf des benutzer-definierten Konstruktors ausgeführt und damit den Attributen `x`, `y` des neu erzeugten Objekts die Werte `1` und `2` zugewiesen.

(Eine allgemeine Vorschrift zur Ausführung von Objekterzeugung vgl. unten).

# Wdh: Überblick Kapitel 3 - 5

| | | *Kapitel 5* |
|---|---|---|
| | | Klassendeklarationen |
| *Kapitel 3* | | Objekte und Objekthalde (Heap) |
| Grunddatentypen | erweitert um | Klassentypen |
| Werte | erweitert um | Referenzen und `null` |
| Operationen | erweitert um | `==`, `!=` für Referenzen und `null` |
| Ausdrücke | erweitert um | Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck |
| Typisierung | | |
| Auswertung bzgl. Zustand (Stack) | erweitert um | Objekthalde (Heap) |
| *Kapitel 4* | | |
| Kontrollstrukturen | erweitert um | **Return-Anweisung**, **Methodenaufruf**, **Objekterzeugung** |

# Erweiterte Grammatik für Anweisungen im Kontext von Klassendeklarationen

*Statement* =

*VariableDeclaration*

| *Assignment*

| *Block*

| *Conditional*

| *Iteration*

| ***ReturnStatement*** (← neu)

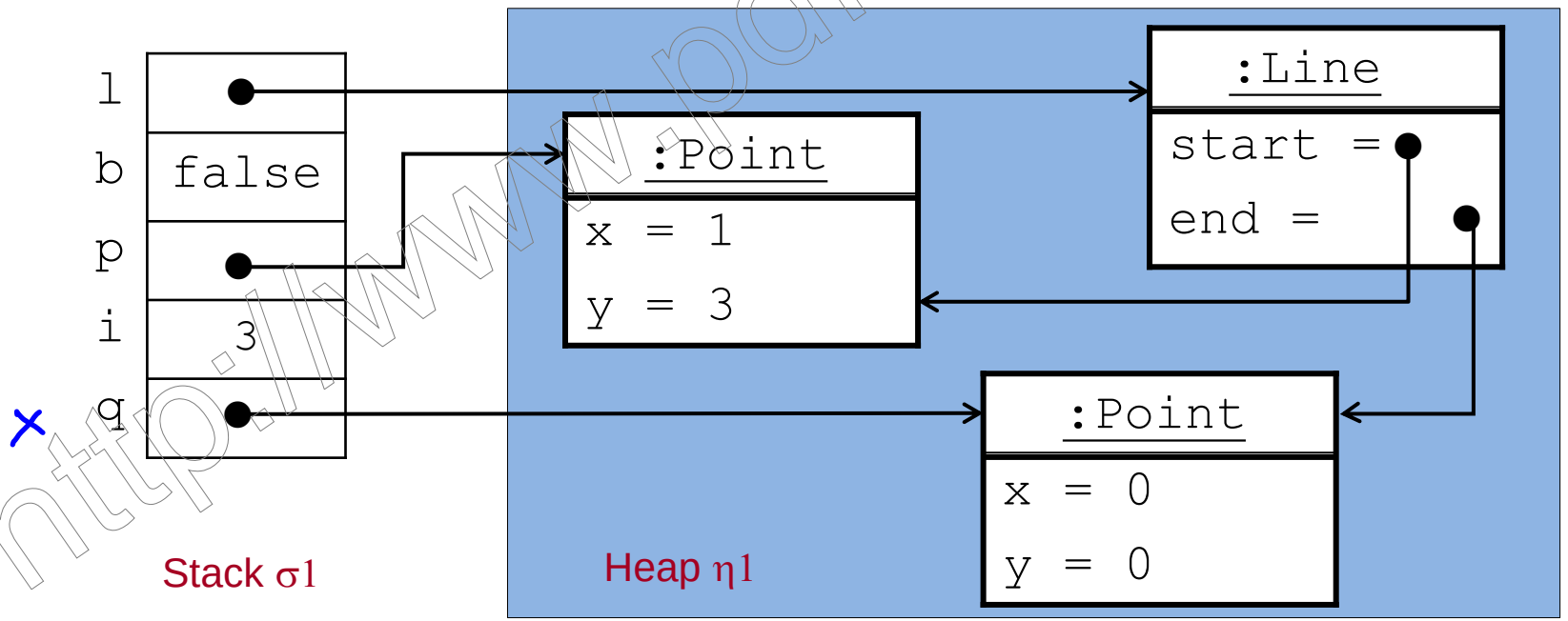
| ***MethodInvocation*** **";"** (← neu)

| ***ClassInstanceCreation*** **";"** (← neu)

# Deklarationsanweisungen und Zustandsänderung

```
Point q = new Point(0,0);
int i = 3;
Point p = new Point(1,i);
boolean b = false;
Line l = new Line(p,q);
```

führt zu folgendem Zustand:

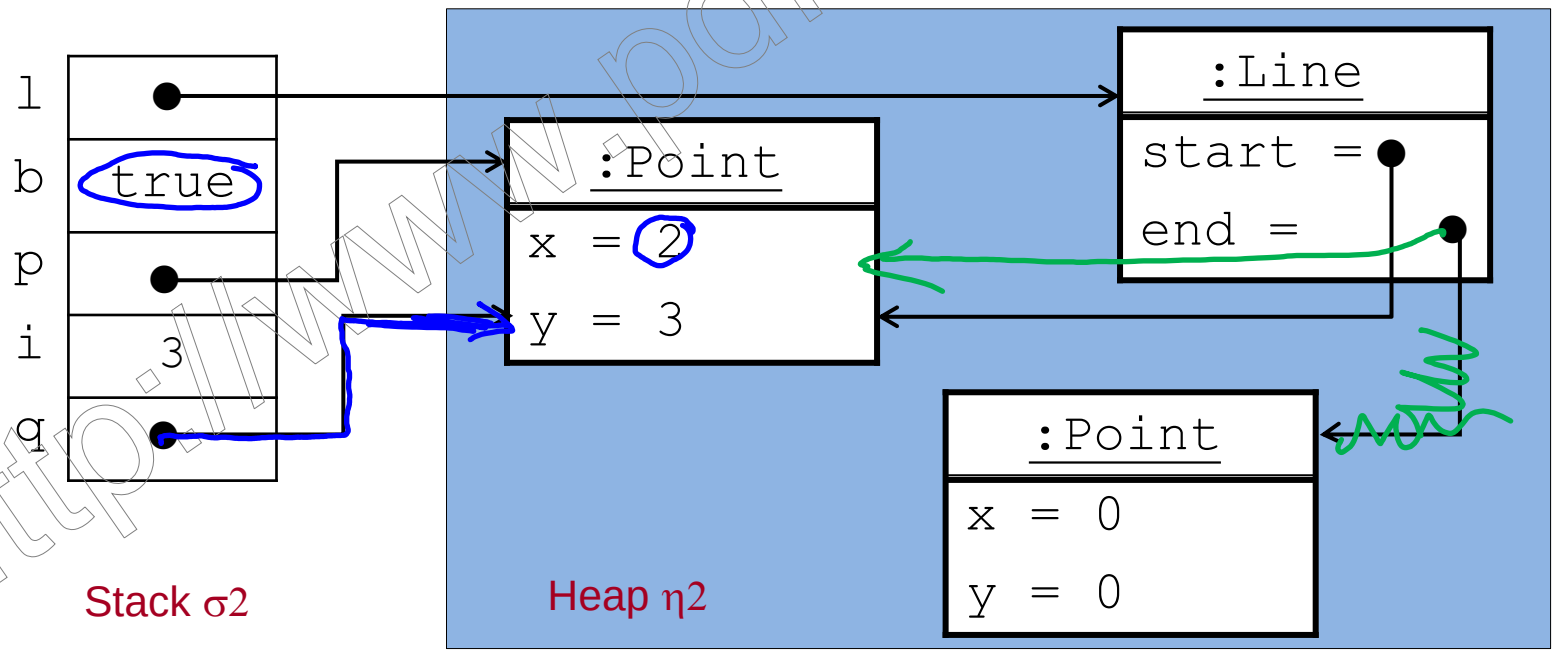


# Zuweisungen und Zustandsänderung

Im Zustand $(\sigma 1, \eta 1)$ der letzten Folie werden folgende Zuweisungen durchgeführt:

```
q = p;  //Aliasing! q und p zeigen auf dasselbe Objekt!
p.x = p.x + 1;
b = (q.getX() == 2);
```

Dies führt zu folgendem Zustand:

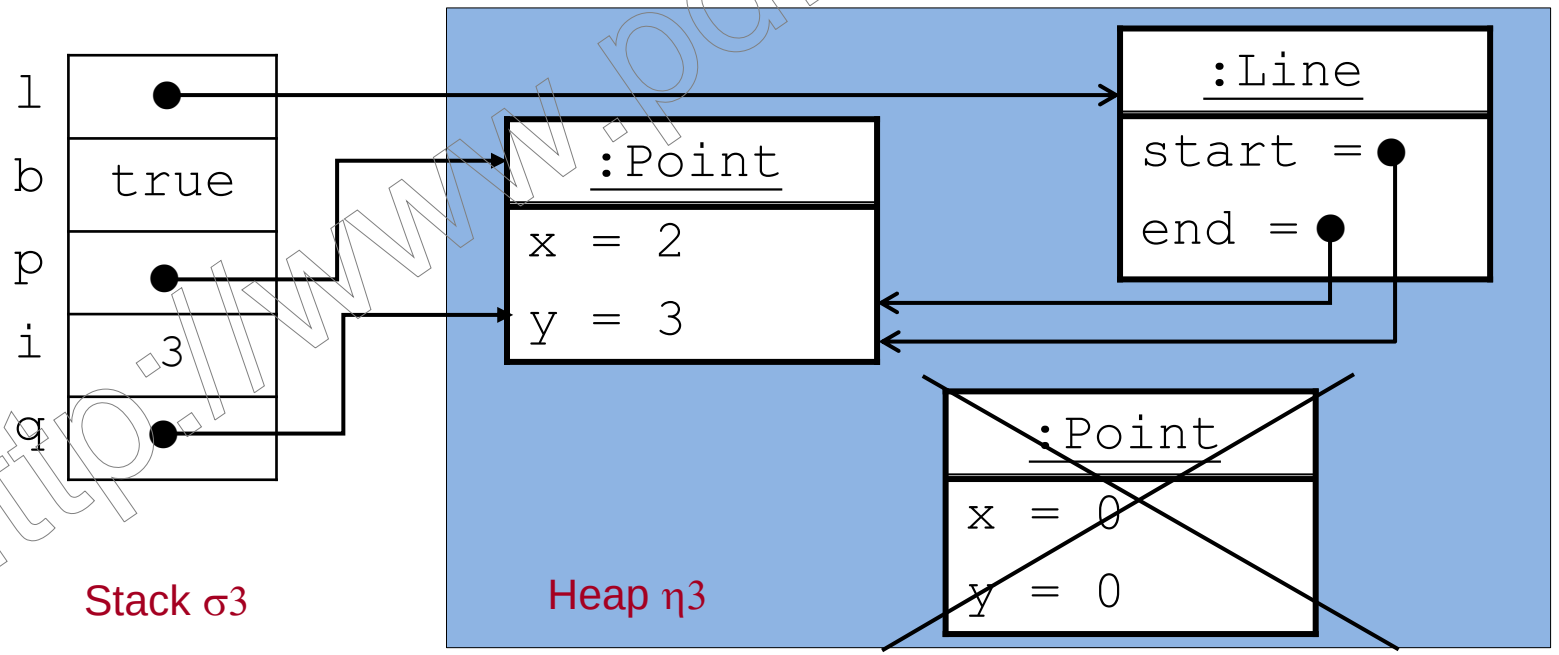


# Datenmüll

Im Zustand $(\sigma 2, \eta 2)$ der letzten Folie führen wir durch: `l.end = p;`

Im Zustand $(\sigma 3, \eta 3)$ nach der Zuweisung ist ein Objekt unerreichbar geworden.

- Keine Referenz zeigt mehr darauf.
- Es ist Müll (engl.: Garbage) und wird automatisch vom Speicherbereinigungsalgorithmus („Garbage Collector“) gelöscht.



# Return-Anweisung

Syntax: *ReturnStatement* = "return" [*Expression*] ";"

- Eine Return-Anweisung mit einem Ergebnisausdruck muss in jedem Ausführungspfad einer Methode mit Ergebnis vorhanden sein.
- Der Typ von *Expression* muss zum Ergebnistyp der Methode passen.

Wirkung:

- Die Ausführung des Methodenrumpfs wird beendet.
- Bei Methoden mit Ergebnistyp wird der Ausdruck *Expression* im zuletzt erreichten Zustand ausgewertet und dessen Wert als Ergebnis bereit gestellt.

```
int w = p.getX();
```

# Methodenaufruf-Anweisung

Syntax: *MethodInvocation* ";"

*MethodInvocation* = *Expression* "." *Identifier* "(" [*ActualParameters*] ") "

Eine Methodenaufruf-Anweisung hat also die Form

$e.m(a_1,\ldots,a_n);$

- Der Ausdruck `e` muss einen Klassentyp haben und der Identifier `m` muss der Name einer Methode der Klasse (oder einer Oberklasse, vgl. später) sein:

  `void m (`$T_1$ $x_1$`,…,`$T_n$ $x_n$`) {body}` **oder**

  `Type m (`$T_1$ $x_1$`,…,`$T_n$ $x_n$`) {body}`

- Die aktuellen Parameter $a_1,\ldots,a_n$ sind Ausdrücke, die in Anzahl und Typ zu den formalen Parametern der Methodendeklaration passen müssen.

Beispiel: Sei `e` ein Ausdruck vom Typ `Point`.

Methodenaufruf-Anweisung: `e.move(10,15);`

# Methodenaufruf-Anweisung: Wirkung

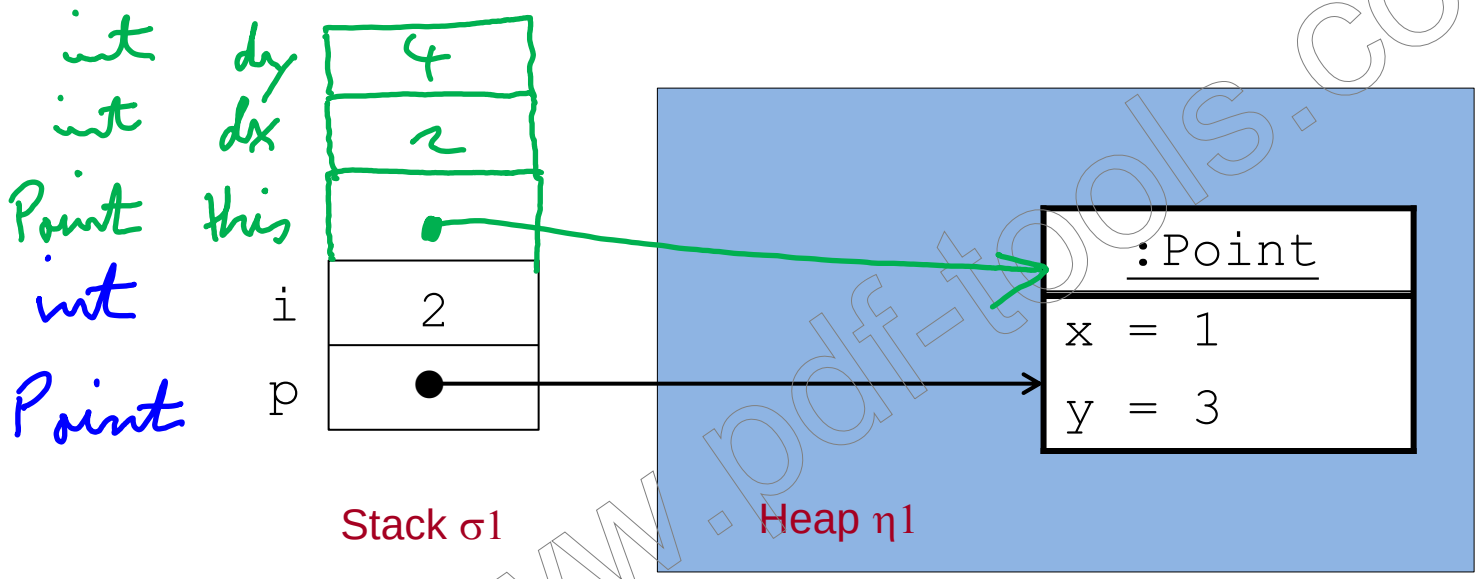
`e.m(a1,…,an);` hat folgende Wirkung:

Sei `e` ein Ausdruck mit Klassentyp `C`.

1. Der Ausdruck `e` wird im aktuellen Zustand ausgewertet.
   Falls der Wert `null` ist, erfolgt ein Laufzeitfehler (`NullPointerException`), andernfalls wird eine lokale Variable `this` vom Typ `C` angelegt und mit der erhaltenen Objektreferenz initialisiert.
2. Analog werden die Werte aller aktuellen Parameter $a_1,\ldots,a_n$ berechnet, lokale Variable für die formalen Parameter der Methode angelegt und mit den erhaltenen Werten der aktuellen Parameter initialisiert („**Call by Value**“).
3. Der Rumpf der Methode wird (als Block) ausgeführt.
4. Die lokalen Variablen `this`, $x_1, \ldots, x_n$ werden vom Stack genommen.

**Beachte:** Von einer anderen Klasse aus, sind Methodenaufrufe nur gemäß den spezifizierten Sichtbarkeiten zulässig.

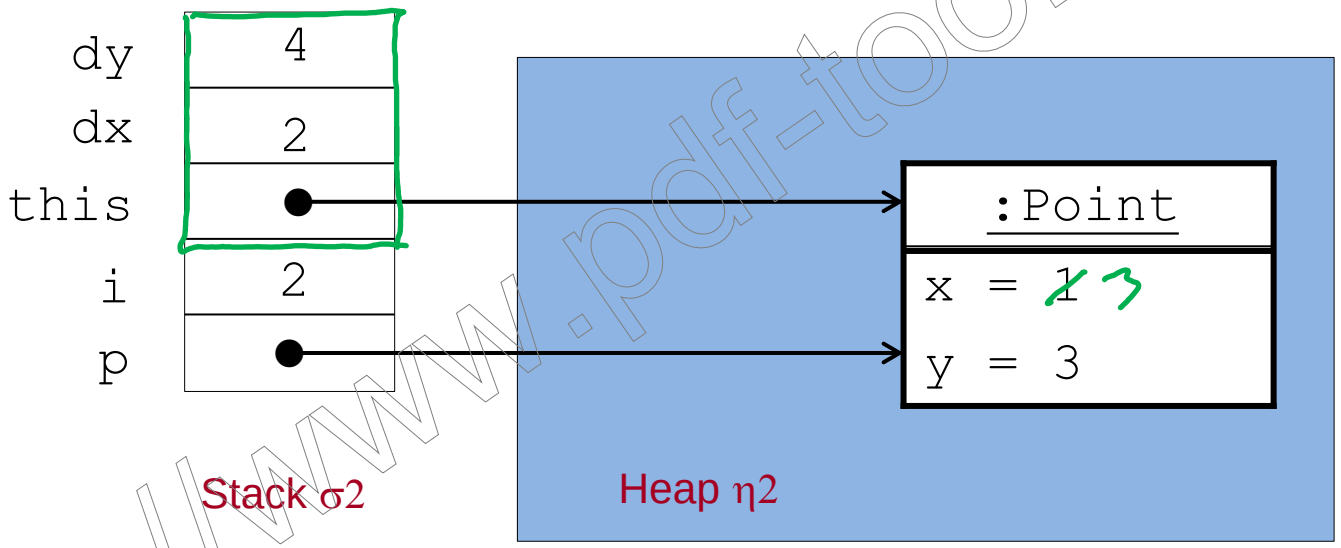
# Call-by-Value Parameterübergabe: Beispiel (1)




Im Zustand $(\sigma 1, \eta 1)$ werde `p.move(i,2+2);` aufgerufen.

# Call-by-Value Parameterübergabe: Beispiel (2)

Zustand nach Parameterübergabe:

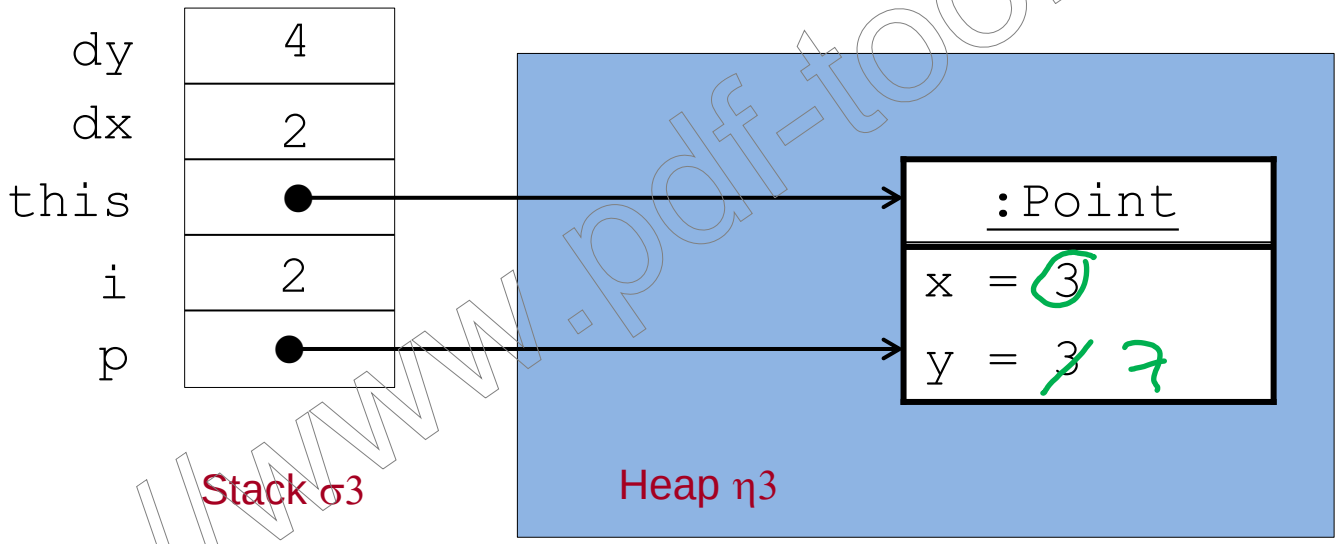



Nun wird der Rumpf der Methode `move` ausgeführt:

```
{ this.x = this.x + dx;
  this.y = this.y + dy; }
```

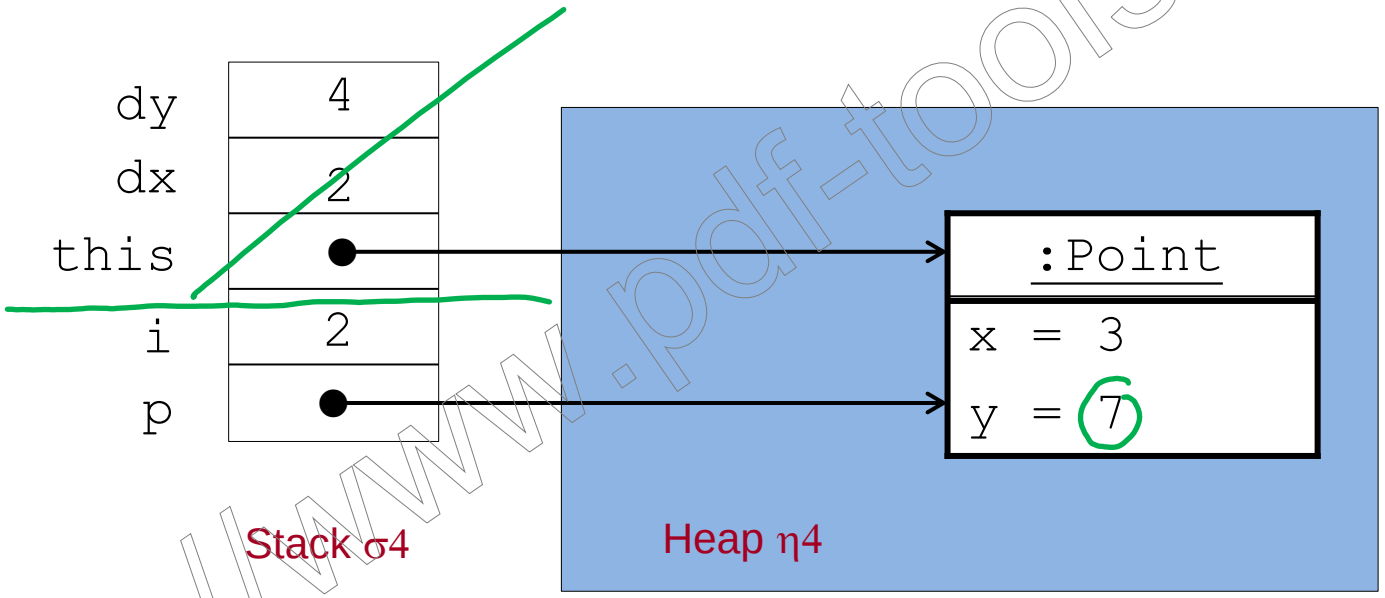
# Call-by-Value Parameterübergabe: Beispiel (3)

Zustand nach Ausführung von `this.x = this.x + dx;` :



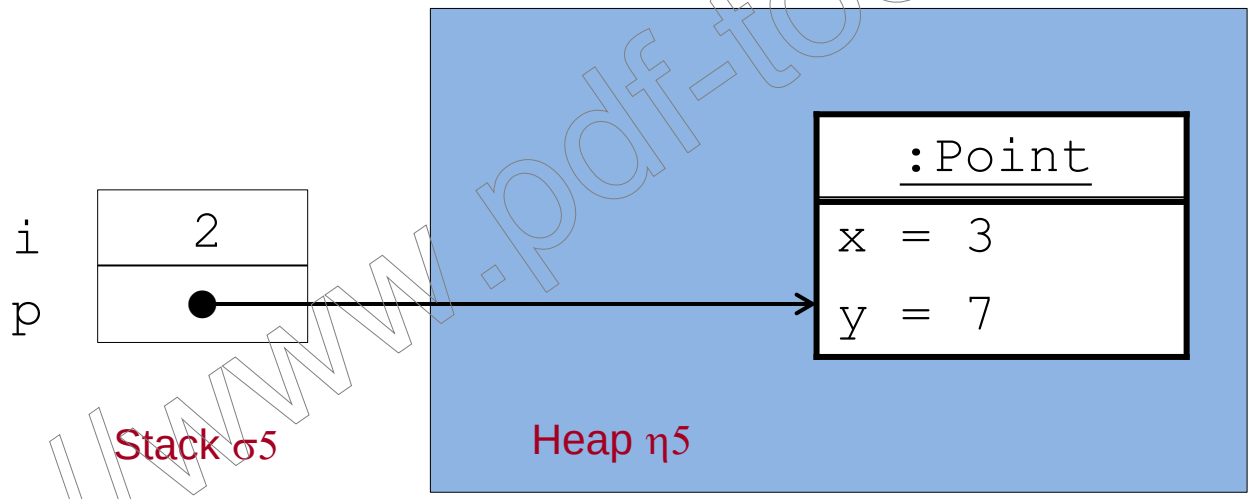
# Call-by-Value Parameterübergabe: Beispiel (4)

Zustand nach Ausführung von `this.y = this.y + dy;` :



# Call-by-Value Parameterübergabe: Beispiel (5)

Anschließend werden die lokalen Variablen `this, dx, dy` vom Stack entfernt:



# Objekterzeugungs-Anweisung

Syntax: *ClassInstanceCreation* ";"

Wdh.: *ClassInstanceCreation* = "new" *ClassType* "(" [*ActualParameters*] ") "

Eine Objekterzeugungs-Anweisung hat also die Form

```
new C(a1,…,an);
```

wobei `new C(a1,…,an)` ein Objekterzeugungs-Ausdruck ist (vgl. oben).

# Objekterzeugungs-Anweisung: Wirkung

`new C(`$a_1$`,…,`$a_n$`);` hat folgende Wirkung:

1. Ein neues Objekt der Klasse `C` wird erzeugt und auf den Heap gelegt.
2. Die Felder des Objekts werden mit Default-Werten initialisiert.
   (`0` bei `int`, `false` bei `boolean`, `null` bei Klassentypen).
3. Die Referenz auf das neue Objekt wird als Ergebniswert bereit gestellt.

Falls ein benutzerdefinierter Konstruktor aufgerufen wird, erfolgt vor 3.:

i. Eine lokale Variable `this` mit Typ `C` wird angelegt und mit der Referenz auf das neue Objekt initialisiert.
ii. Die Werte aller aktuellen Parameter $a_1,\ldots,a_n$ werden berechnet, lokale Variable für die formalen Parameter des Konstruktors werden angelegt und mit den erhaltenen Werten der aktuellen Parameter initialisiert.
iii. Der Rumpf des Konstruktors wird (als Block) ausgeführt.
iv. Die lokalen Variablen `this,` $x_1$`, … ,`$x_n$ werden vom Stack genommen.

# Benutzung von Klassen und Objekten

Objekte werden (meist) in Methoden von anderen Klassen erzeugt und benutzt. Die Benutzung geschieht (meist) durch Methodenaufruf.

**Beispiel "Point":**

```
public class PointMain {
   public static void main(String[] args) {
      Point p1 = new Point(10, 20);
      Point p2 = new Point(0,0);
      int x1 = p1.getX(), y1 = p1.getY() ;
      int x2 = p2.getX(), y2 = p2.getY();
      System.out.println("p1=(" + x1 + ", " + y1 + ")");
      System.out.println("p2=(" + x2 + ", " + y2 + ")");
      p1.move(10, 10);
      System.out.println("p1=(" + p1.getX() + ", " + p1.getY() + ")");
   }
}
```

Aufruf einer Methode mit Rückgabewert

Methodenaufruf (ohne Rückgabewert)

p1 = (10, 20)

p2 = (0, 0)

p1 = (20, 30)

# Klasse „Point“ mit öffentlichen Attributen

```
public class Point {
    public int x,y;
    public Point(int x0, int y0){
       this.x = x0;
        this.y = y0;
    }
    public void move(int dx, int dy){
        this.x = this.x + dx;
        this.y = this.y + dy;
    }
    public int getX(){
        return this.x;
    }
     public int getY(){
        return this.y;
    }
}
```

Auf öffentliche Attribute kann von anderen Objekten aus zugegriffen werden! Dies verletzt die Idee des **Geheimnisprinzips**, nach dem Änderungen an Objektzuständen nur unter Kontrolle von Methodenaufrufen geschehen sollen.

# Benutzung von Objekten/Klassen bei öffentlichem Attributzugriff

**Beispiel "Point":**

```
public class PointMain {
    public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        Point p2 = new Point();
        int x1 = p1.x; y1 = p1.y;
        int x2 = p2.x; y2 = p2.y;
        System.out.println("p1=(" + x1 + ", " + y1 + ")");
        System.out.println("p2=(" + x2 + ", " + y2 + ")");
        p1.x = p1.x + 10;
        System.out.println("p1=(" + p1.x + ", " + p1.y + ")");
    }
}
```

0, 0

Zugriff auf das Attribut eines anderen Objekts

Änderung des Attributwerts eines anderen Objekts

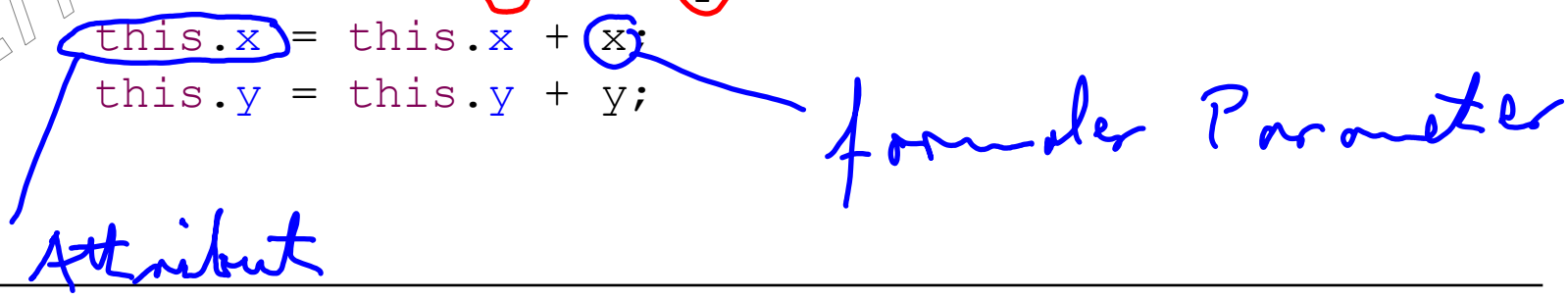
# Methodenimplementierung: Abkürzung

Innerhalb einer Methodenimplementierung ist der Name von `this` eindeutig und kann weggelassen werden, wenn keine Namenskonflikte auftreten.

```
public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
}
```

**Aber**: Parameter und lokale Variablen überdecken Attribute gleichen Namens. Die folgende Implementierung von `move` benötigt die explizite Verwendung von `this`.

```
public void move(int x, int y) {
        this.x = this.x + x;
        this.y = this.y + y;
}
```



# Statische Attribute und statische Methoden

- **Statische Attribute** (**Klassenattribute**) sind (globale) Variablen einer Klasse, die unabhängig von Objekten Werte speichern.
- **Statische Methoden** (**Klassenmethoden**) sind Methoden einer Klasse, die unabhängig von Objekten aufgerufen und ausgeführt werden.
- Syntax:

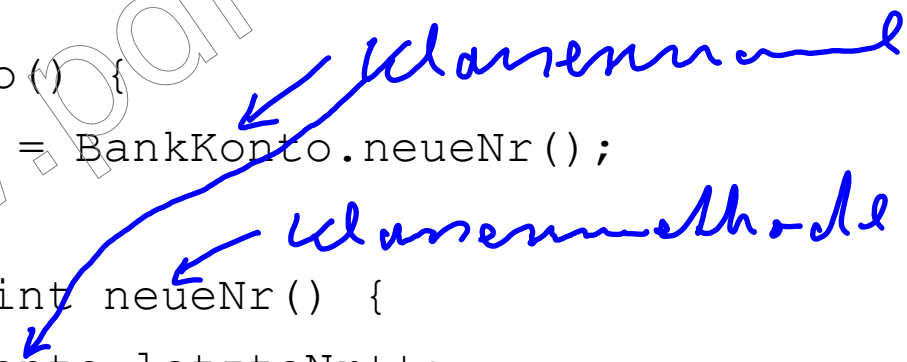
```
class C {
   private static type attribute = … ;
   public static void method( … ) {body};
   ...  }
```

- Im Rumpf einer statischen Methode dürfen keine Instanzvariablen verwendet werden.
- Zugriff auf ein Klassenattribut: `C.attribute` z.B. `System.out`
- Aufruf einer Klassenmethode: `C.method( … )` z.B. `Math.sqrt(7)`

# Klassenattribute und –methoden: Beispiel

```
class BankKonto {
   private double kontoStand;
   private int kontoNr;
   private static int letzteNr = 0;

   public BankKonto() {
      this.kontoNr = BankKonto.neueNr();
   }
   private static int neueNr() {
      return BankKonto.letzteNr++;
   }
   ...
}
```



# Klassenmethoden: Beispiele

```
class NumFunktionen {

public static int quersumme(int x){
    int qs = 0;
    while (x > 0) {
      qs = qs + x % 10;
      x = x / 10;
    }
    return qs;
  }
public static int fakultaet(int n){
    int akk = 1;
    while (n > 1) {
      akk = akk * n;
      n--;
    }
    return akk;
  }
}
```

((1 * 4) * 3) * 2

**Benutzung:**

```
class NumAnwendung {

public static void main(String[] args){
   int x = 352;
   int q = NumFunktionen.quersumme(x);
   System.out.println("Quersumme von"
+ x + ": " + q);

   int x = 6;
   System.out.println("Fakultät von"
+ x + ": " + NumFunktionen.fakultaet(x));
  }
}
```

klasse

# Konstanten

- **Konstanten** sind Klassenattribute mit einem festen, unveränderlichen Wert.
- Syntax:

```
class C {
  public static final type attribute = value;
  ...  }
```

- Konstanten werden meist mit Großbuchstaben geschrieben und meist als `public` deklariert.
- Beispiel:

```
class Math {
  public static final double PI = 3.14159265358979323846;
  ...  }
```

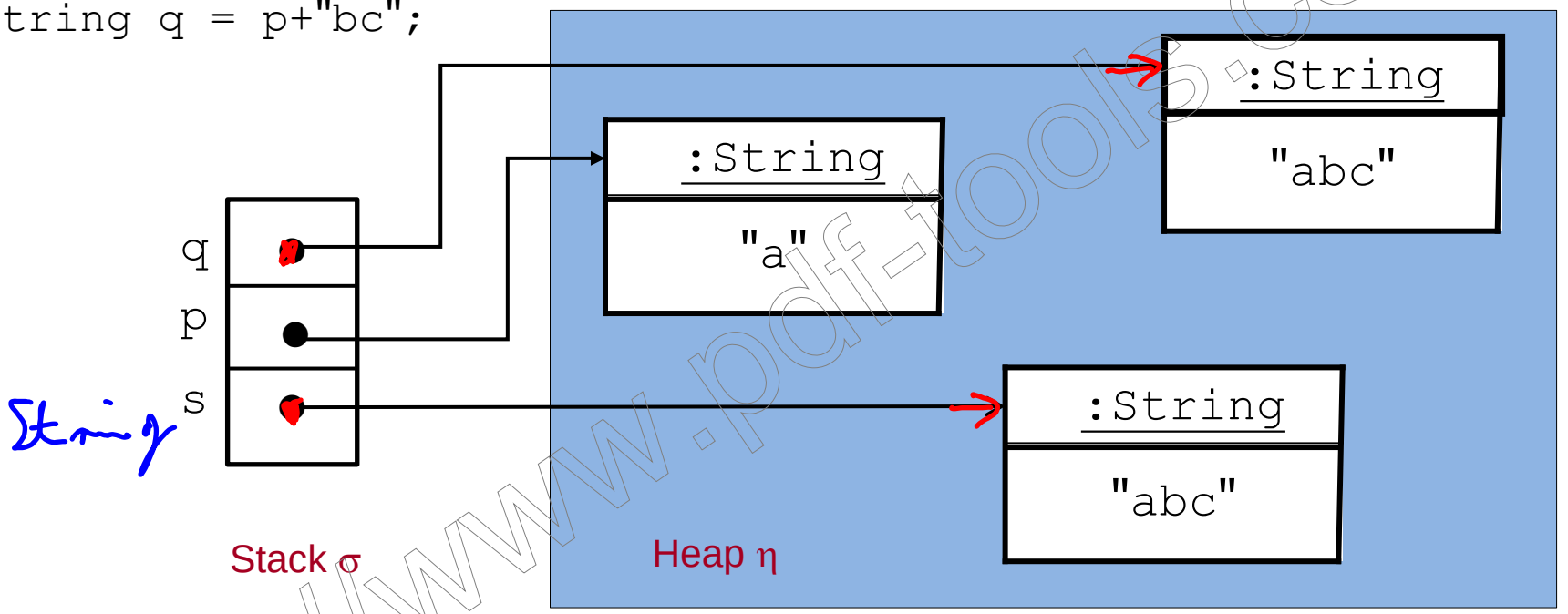
Math.PI

# Die Klasse String

- Zeichenketten (Strings) werden in Java durch Objekte der Klasse `String` repräsentiert. Diese Objekte speichern eine (unveränderbare) Folge von Zeichen (Characters).
- Infolgedessen sind die Werte des Klassentyps `String` Referenzen auf String-Objekte.
- Referenzen auf String-Objekte können durch String-Literale angegeben werden: z.B. `"WS 2011/12"`, `"M-XY 789"`, `"\""`, `""` (leerer String).
- Operationen auf Strings sind:
  `==`, `!=` Vergleich von Referenzen (**nicht empfohlen!**)
  `+` Zusammenhängen zweier Strings zu einem neuen String
- Die Klasse `String` enthält eine Vielzahl von Konstruktoren und Methoden, z.B. `public boolean equals(Object anObject)` für den Vergleich der Zeichenketten („Inhalte“) zweier String-Objekte (**empfohlen!**).

# Gleichheit von Strings

```
String s = "abc";
String p = "a";
String q = p+"bc";
```




- Gleichheit von String-Referenzen:
  `(s==p)`$=_{(\sigma,\eta)}$ `false`, `(s==q)`$=_{(\sigma,\eta)}$ **`false`** (!!),
- Gleichheit von String-Inhalten (Zeichenketten):
  `s.equals(p)`$=_{(\sigma,\eta)}$ `false`, `s.`**`equals`**`(q)`$=_{(\sigma,\eta)}$ **`true`**

# Ausschnitt aus der Java-Dokumentation der Klasse String

| Method Summary | |
|---|---|
| char | charAt(int index)<br>Returns the char value at the specified index. |
| boolean | isEmpty()<br>Returns true if, and only if, length() is 0. |
| int | length()<br>Returns the length of this string. |
| String | replace(char oldChar, char newChar)<br>Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar. |
| String | substring(int beginIndex, int endIndex)<br>Returns a new string that is a substring of this string. |
| String | toString()<br>This object (which is already a string!) is itself returned. |

Es gibt ca. 50 Methoden
in der Klasse String

# Umwandlung von Strings in Werte der Grunddatentypen

## Statische Methoden

`public static int parseInt(String s)` der Klasse `Integer`,

`public static double parseDouble(String s)` der Klasse `Double`, etc.

z.B. `String s = "67"; int x = Integer.parseInt(s);` x = 67

Der String `s` muss eine ganze Zahl repäsentieren; ansonsten kommt es zu einem Laufzeitfehler (`NumberFormatException`).

Nötig beim Einlesen von numerischen Werten aus Textfeldern.

(z.B. Methoden

`public static String showInputDialog(Object message) throws ..`
`public String getText()` der Klassen `JTextField`, `JTextArea`,

vgl. später).

"Zu" + "u"

# Umwandlungen in Strings

**Statische Methoden** (zur Umwandlung von Werten von Grunddatentypen)

`public static String toString(int i)` der Klasse `Integer`,

`public static String toString(double d)` der Klasse `Double`, etc.

z.B. `int x = 14; String s = Integer.toString(x);` `s = "14"`

Nötig beim Ausgeben von numerischen Werten in Textfeldern (Methode `public void setText(String t)` der Klassen `JTextField`, `JTextArea`, vgl. später).
Nicht nötig für Ausgaben mit `System.out.println`.

**Methode** `public String toString()`
kann auf Objekte aller Klassen angewendet werden.

z.B. `BankKonto b = new BankKonto(); String s = b.toString();`

Liefert einen String, bestehend aus dem Namen der Klasse, zu der das Objekt gehört, dem Zeichen @ sowie einer Hexadezimal-Repräsentation des Objekts,
z.B. `BankKonto@a2b7ef43`