Kapitel 5

Objekte und Klassen

Ziele

- Grundbegriffe objektorientierter Programmierung kennenlernen
- Klassen in Java deklarieren können
- Das Speichermodell für Objekte verstehen
- Typen, Ausdrücke und deren Auswertung im Kontext von Klassendeklarationen verstehen

Überblick Kapitel 3 und 5

Kapitel 3

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 5

Klassendeklarationen

Objekte und Objekthalde (Heap)

Objektorientierte Programmierung

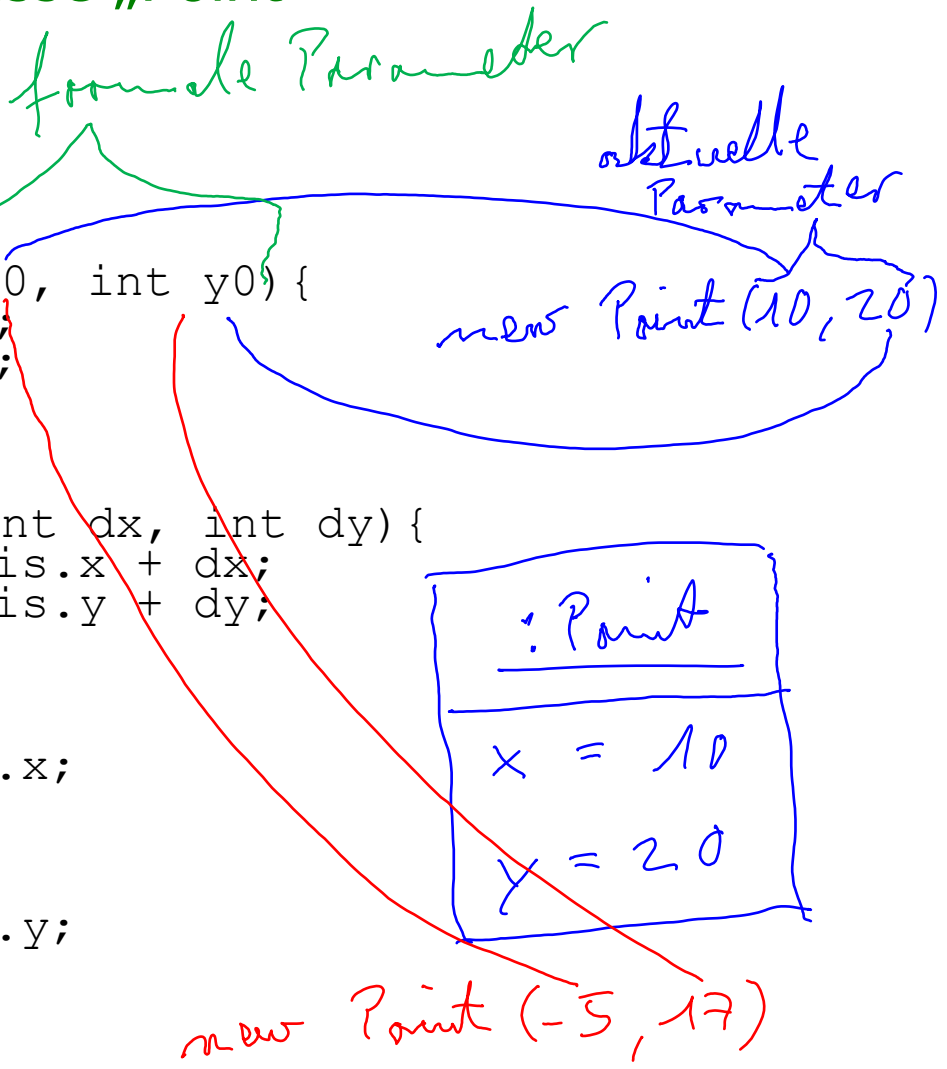
- In der objektorientierten Programmierung werden **Daten und Methoden**, die Algorithmen implementieren, zu geschlossenen Einheiten (**Objekten**) zusammengefasst.
- **Beispiele**:
  - Bankkonto
    Daten: Kontostand, Zinssatz; Methoden: einzahlen, abheben, …
  - Punkte, Linien, Kreise in einem Zeichenprogramm
    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;
        }
}
```



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

Methode mit Ergebnis vom Typ int

Return-Anweisung mit Ergebniswert

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;
          }
```

Summary

Details

Summary / Details

```
    /**
     * 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

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()` Diese Methode gibt die x-Koordinate des Punkts zurueck
int	`getY()` Diese Methode gibt die y-Koordinate des Punkts zurueck
void	`move(int dx, int dy)` 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“

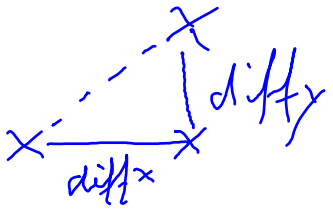
```
public class Line {
    private Point start;
    private Point end;
    public Line(Point s, Point e){
        this.start = s;
        this.end = e;
    }
    public void move(int dx, int dy){
        this.start.move(dx,dy);
        this.end.move(dx,dy);
    }
    public double length(){
        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 vom Typ Point

Methodenkopf

Methodenrumpf

Aufruf der Methode "move" der Klasse Point für das Point-Objekt bezeichnet mit "start"



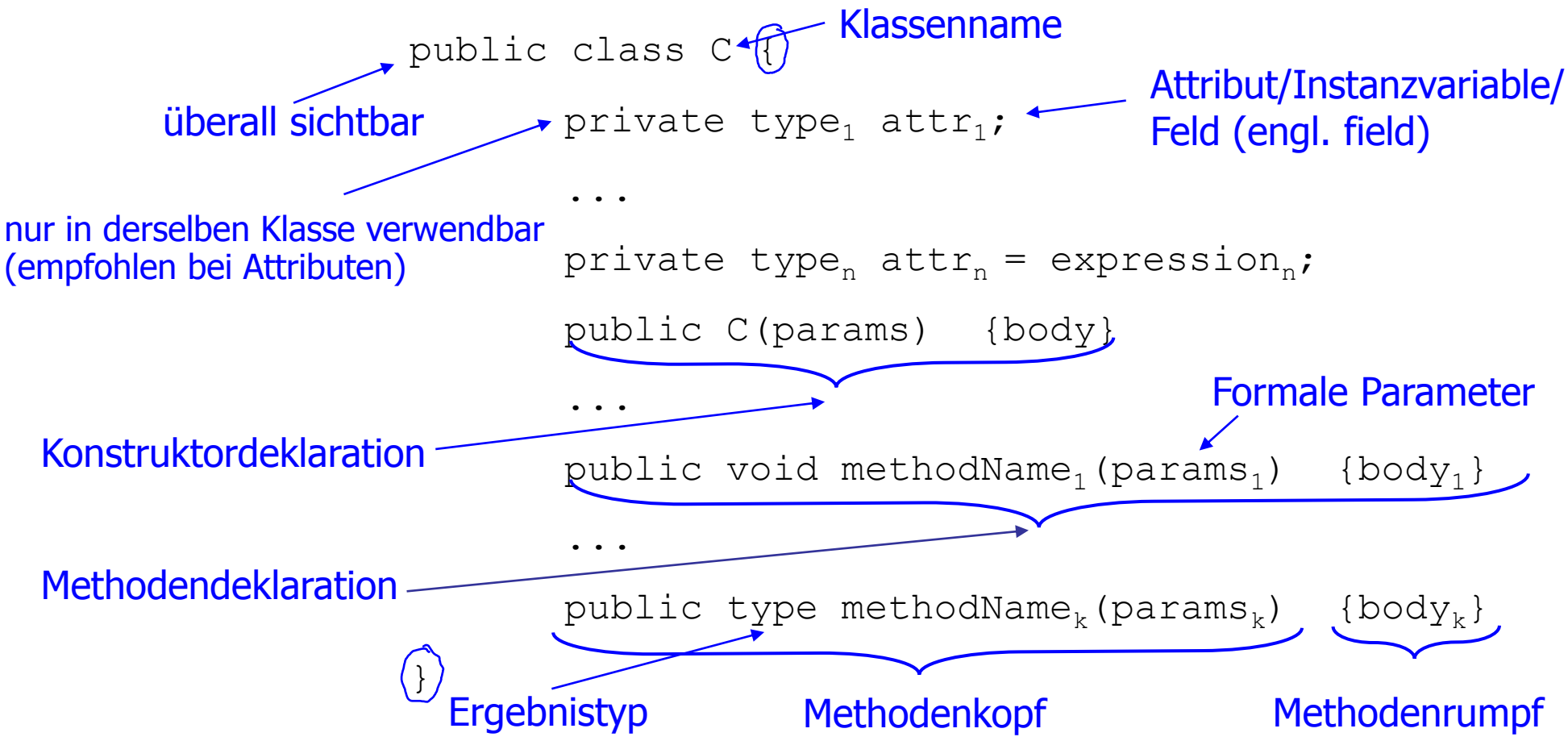
Klassendeklarationen in Java (ohne Vererbung)

```
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*

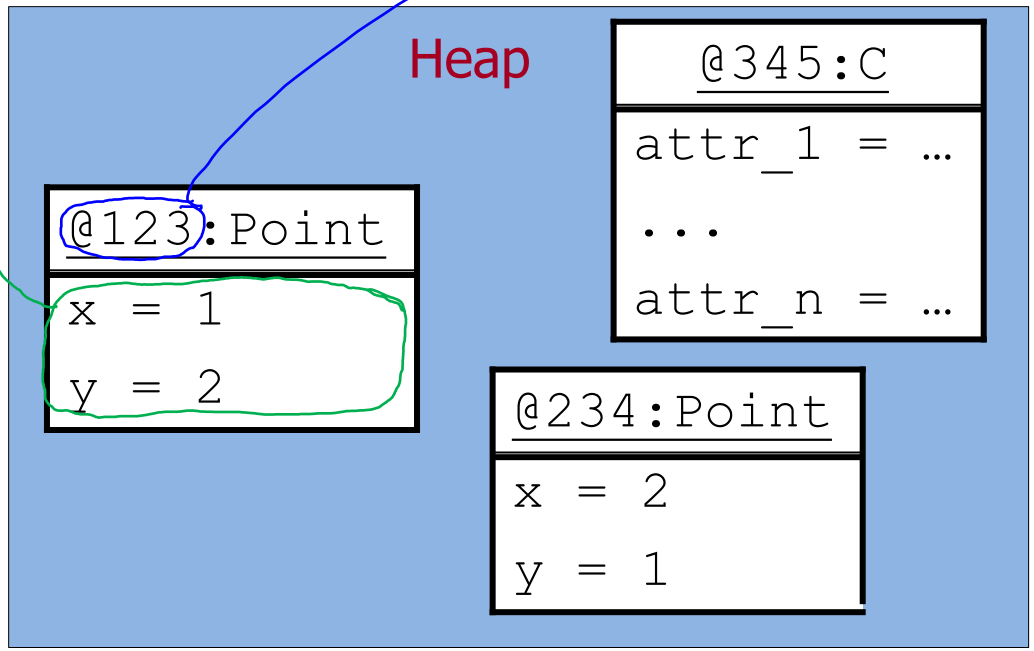
Header = [*Modifier*] (*Type* | "void") *Identifier* "(" [*FormalParameters*] ")"

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,…)]>
}
```

Wdh: Überblick Kapitel 3 - 5

Kapitel 3

Grunddatentypen
Werte
Operationen
Ausdrücke
Typisierung
Auswertung bzgl.
Zustand (Stack)

erweitert um
erweitert um
erweitert um
erweitert um
erweitert um

Kapitel 5

Klassendeklarationen
Objekte und Objekthalde (Heap)

Klassentypen
Referenzen und `null`
`==`, `!=` für Referenzen und `null`
Attributzugriff,
Methodenaufruf mit Ergebnis,
Objekterzeugungsausdruck

Objekthalde (Heap)

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*

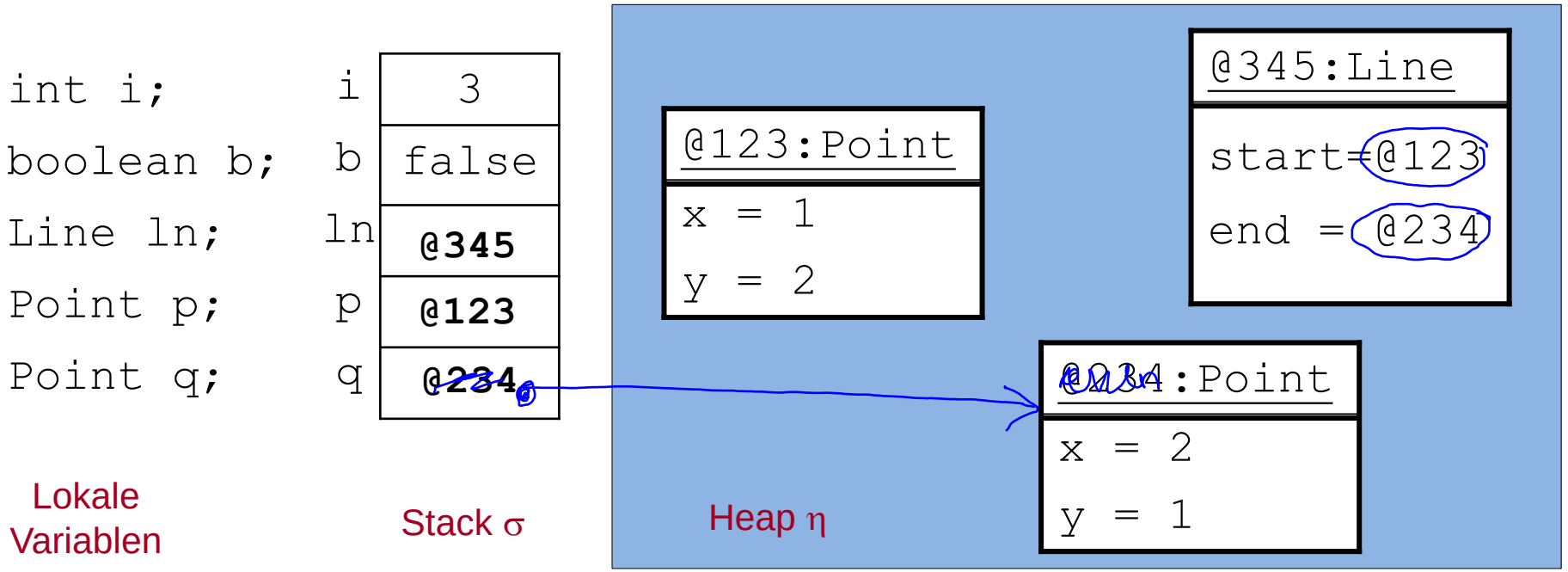
- Mit jeder Klassendeklaration wird ein neuer Typ eingeführt (**Klassentyp**), der den Namen der Klasse hat.
- Die **Werte** eines Klassentyps sind **Referenzen** (Verweise, Zeiger, Links) auf Objekte der Klasse sowie das Element `null` („leere" Referenz).
- Dementsprechend speichern lokale Variable 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.

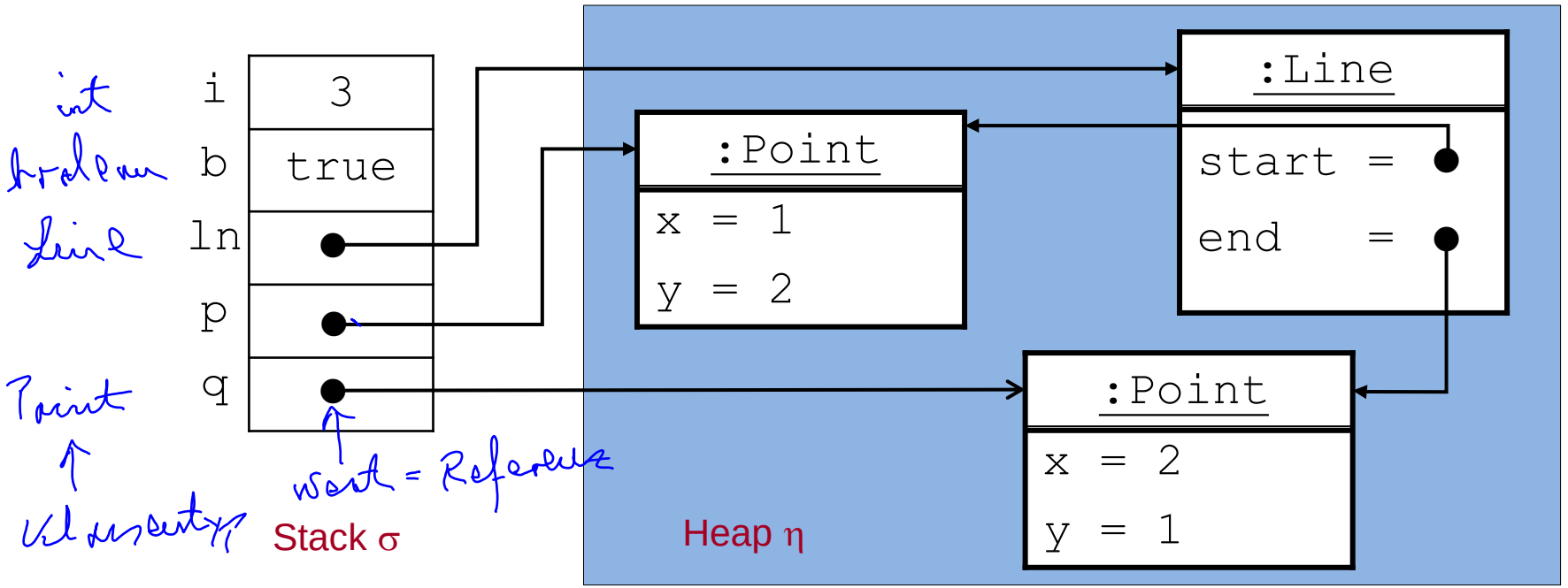
Zustand = Stack + Heap

Ein Zustand (σ,η) eines objektorientierten Java-Programms besteht aus

- einem Stack (Keller) σ für die lokalen Variablen und
- einem Heap (Halde) η für die aktuell existierenden Objekte



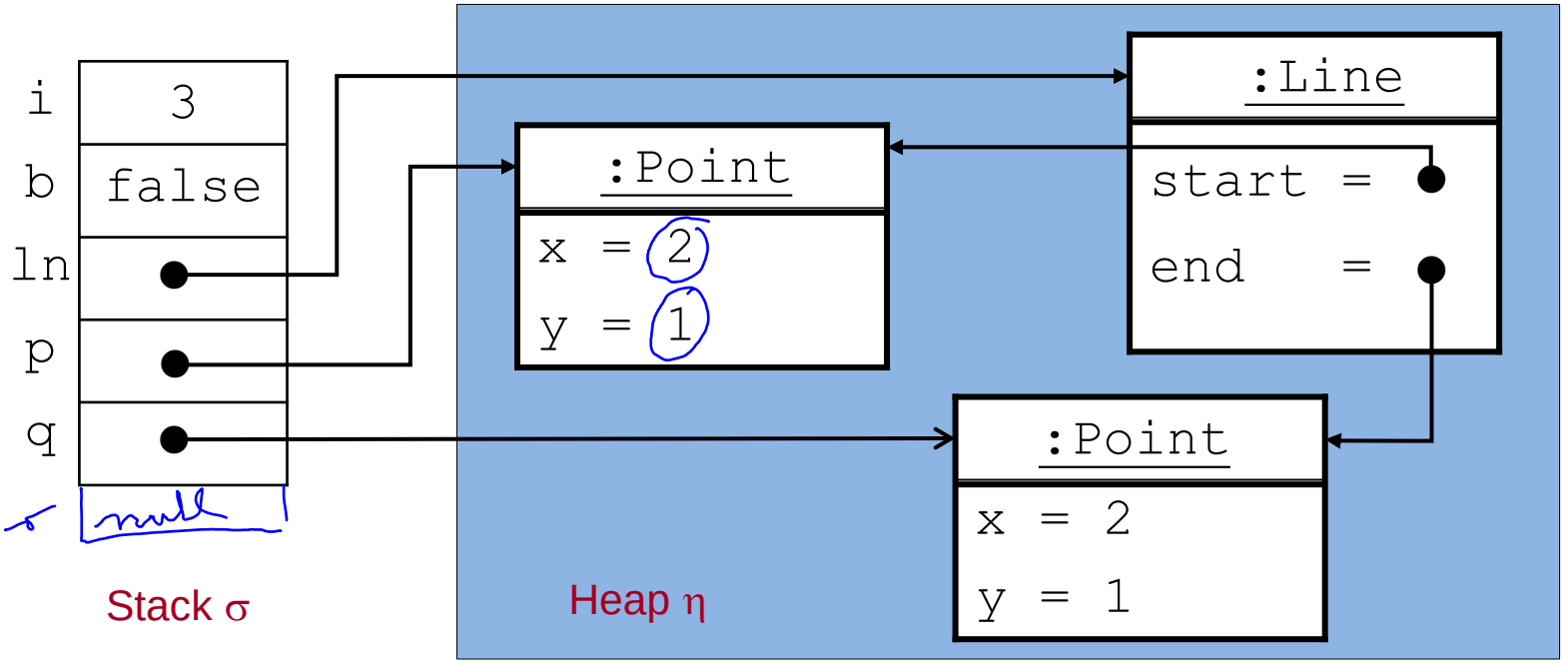
Zustand mit Zeigerdarstellung



Beachte:

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

Gleichheit von Objektreferenzen



Auswertung

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

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)

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)

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

Grammatik für Methodenaufruf- und Objekterzeugungs-Ausdrücke

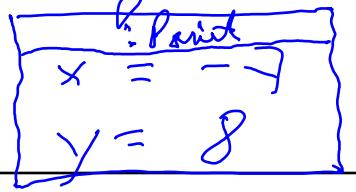
MethodInvocation =
Expression "." *Identifier* "(" [*ActualParameters*] ") "

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

InstanceCreation = *ClassInstanceCreation*

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

z.B. new Point(-7, 8) hat den Typ Point
und liefert einen Verweis auf
ein neu angelegtes Point-Objekt
mit den Attributwerten



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** (σ,η), d.h. wir berechnen $e_{=(\sigma,\eta)}$ ...
- Der Attributzugriff mit "." und der Methodenaufruf mit "." haben die höchste Präzedenz 15.

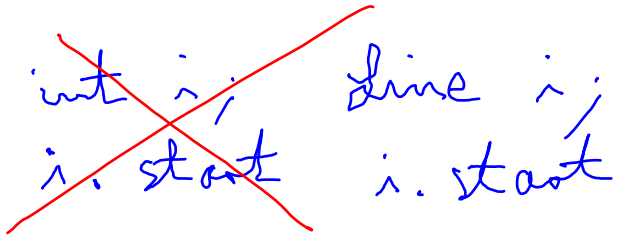
- p.x
- (p.x)

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*.

<u>Beispiel:</u>

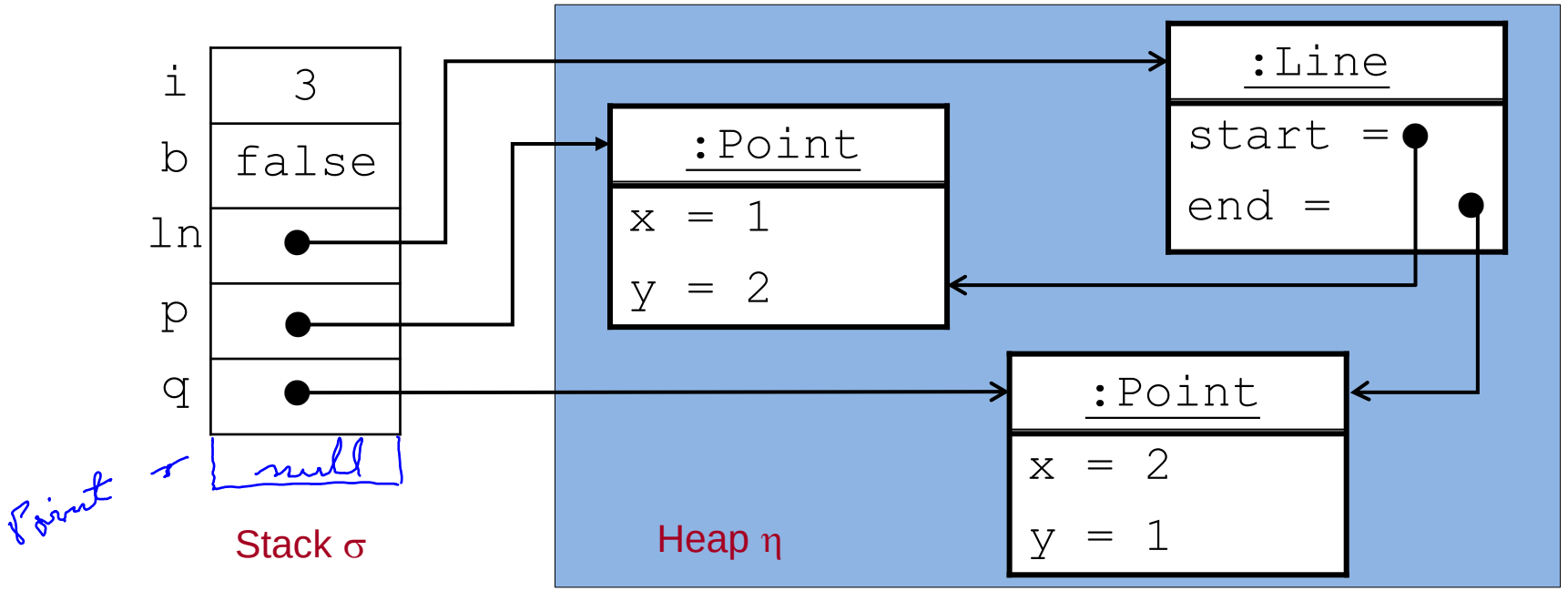
Seien `Point p; Line ln;` lokale Variable.

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

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

`ln.start.y` hat den Typ `int`. (falls y public wäre)

Attributzugriff: Auswertung



- `p.x, q.y, ln.end, ln.end.x, …` sind **Variablen**, deren Werte in einem Zustand (σ,η) die Attributwerte der referenzierten Objekte sind.
- `p.x` $=_{(\sigma,\eta)}$ `1, q.y` $=_{(\sigma,\eta)}$ `1, ln.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(a1,…,an)` hat dann als Typ den Ergebnistyp der Methode.

Methodenaufruf-Ausdruck: Beispiele und Auswertung

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

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

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

Sei (σ,η) der Zustand von oben.

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

`ln.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 (vgl. Kapitel 6).

Objekterzeugungs-Ausdruck

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

Eine Objekterzeugungs-Ausdruck hat also die Form `new C(`a_1`,…,`a_n`)`

- `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 (`T_1 x_1`,…,`T_n x_n`) {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(`a_1`,…,`a_n`)` 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`.

Ausdruck vom Typ Point

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.

(Allgemeine Vorschrift zur Ausführung von Objekterzeugung vgl. Kapitel 6).