

# 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 5*

#### **Klassendeklarationen**

#### **Objekte und Objekthalde (Heap)**

Klassentypen

Referenzen und `null`

`==`, `!=` für Referenzen und `null`

Attributzugriff,

Methodenaufruf mit Ergebnis,

Objekterzeugungsausdruck

Objekthalde (Heap)

### *Kapitel 3*

Grunddatentypen

erweitert um

Werte

erweitert um

Operationen

erweitert um

Ausdrücke

erweitert um

Typisierung

Auswertung bzgl.

Zustand (Stack)

erweitert um

## 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**

- 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**

- 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; 2 Attribute vom Typ int  
  
    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;  
    }  
    }  
}
```

Methodenkopf

Methodenrumpf

3 Methoden

## Beispiel: Klasse „Point“

Klassenname  
`public class Point {`

`private int x,y;}`

2 Attribute (Felder, Instanzvariable)  
vom Typ int

nur in derselben Klasse verwendbar

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

formale Parameter vom  
Typ int

Zugriff auf  
y-Koordinate von this

vordefinierte lokale  
Variable this  
bezeichnet das „gerade  
aktive“ Objekt

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

Methodendeklaration;  
die Methode liefert  
kein Ergebnis

Ergebnistyp  
Return-Anweisung  
mit Ergebniswert  
`public int getX(){  
return this.x;  
}`

Methodendeklaration;  
die Methode liefert ein Ergebnis vom  
Typ int

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

**Bemerkung:** In einer Datei kann höchstens eine „public“ Klasse deklariert sein.





## 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;
    private int 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

## Class Point

java.lang.Object  
vorlesung05.Point

---

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

Diese Klasse repräsentiert 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 müssen.

## Method Summary

### Methods

Modifier and Type	Method and Description
int	<code>getX()</code> Diese Methode gibt die x-Koordinate des Punkts zurueck
int	<code>getY()</code> Diese Methode gibt die y-Koordinate des Punkts zurueck
void	<code>move(int dx, int dy)</code> 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 zurück

#### Returns:

die x-Koordinate des Punkts

### getY

```
public int getY()
```

Diese Methode gibt die y-Koordinate des Punkts zurück

#### 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“

```

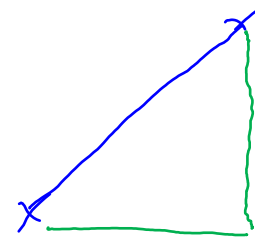
public class Line {
    private Point start;
    private Point end;
    public Line(Point s, Point e){
        this.start = s;
        this.end = e;
    }
    public void move(int dx, int dy){
        this.start.move(dx,dy);
        this.end.move(dx,dy);
    }
    public double length(){
        int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

```

*Handwritten notes:*

- Blue curly brace and text: } 2 Attribute vom Typ Point
- Blue arrow pointing from the curly brace to the two Point attributes in the constructor.
- Blue 'x' at the start of the line in the diagram.
- Blue arrow pointing from the curly brace in the length() method to the return statement.





## Grammatik für Klassendeklarationen (ohne Vererbung)

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

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

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

*Modifier* = "public " | "private"

*MethodDeclaration* = *Header* *Block*

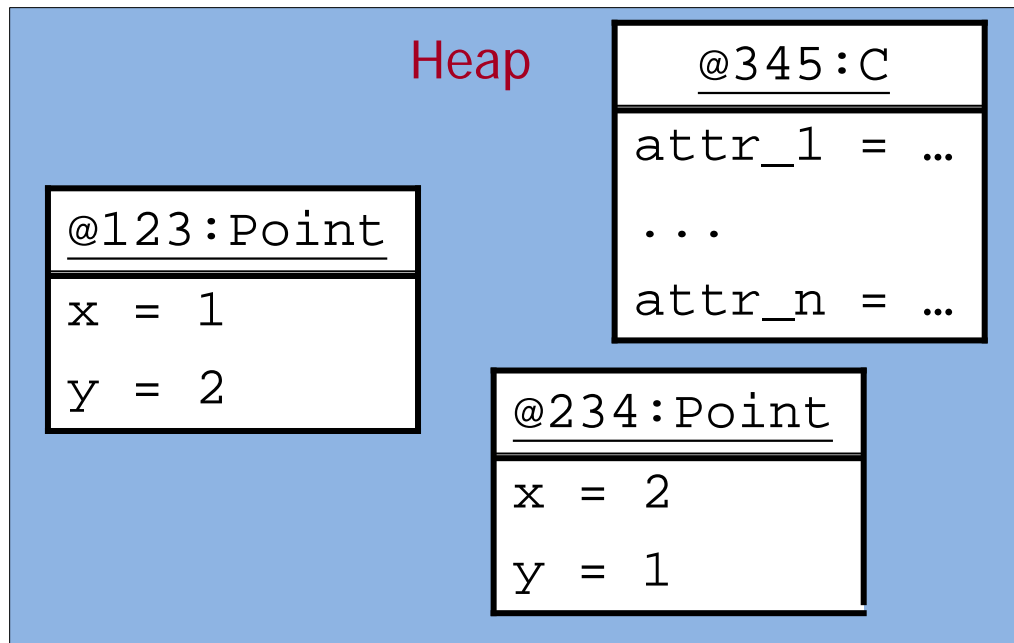
*Header* = [*Modifier*] (*Type* | "void") *Identifier* "(" [*FormalParameters*] ")"

*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 und 5

### *Kapitel 5*

Klassendeklarationen

Objekte und Objekthalde (Heap)

**Klassentypen**

**Referenzen und null**

**==, != für Referenzen und null**

Attributzugriff,

Methodenaufruf mit Ergebnis,

Objekterzeugungsausdruck

**Objekthalde (Heap)**

### *Kapitel 3*

Grunddatentypen

**erweitert um**

Werte

**erweitert um**

Operationen

**erweitert um**

Ausdrücke

erweitert um

Typisierung

Auswertung bzgl.

Zustand (Stack)

**erweitert um**

## Klassentypen

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

$Type = PrimitiveType \mid ClassType$  (← 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  $K \neq$  Werte des Klassentyps  $K$ .

## Zustand = Stack + Heap

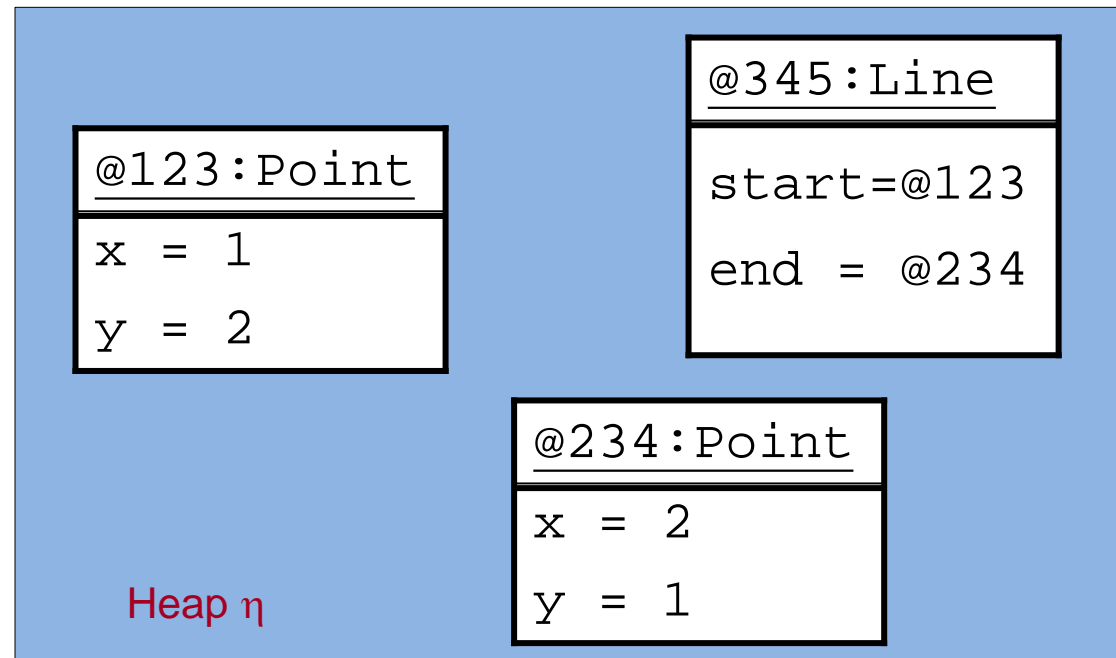
Ein Zustand  $(\sigma, \eta)$  eines objektorientierten Java-Programms besteht aus

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

<code>int i;</code>	<code>i</code>	<code>3</code>
<code>boolean b;</code>	<code>b</code>	<code>false</code>
<code>Line ln;</code>	<code>ln</code>	<code>@345</code>
<code>Point p;</code>	<code>p</code>	<code>@123</code>
<code>Point q;</code>	<code>q</code>	<code>@234</code>

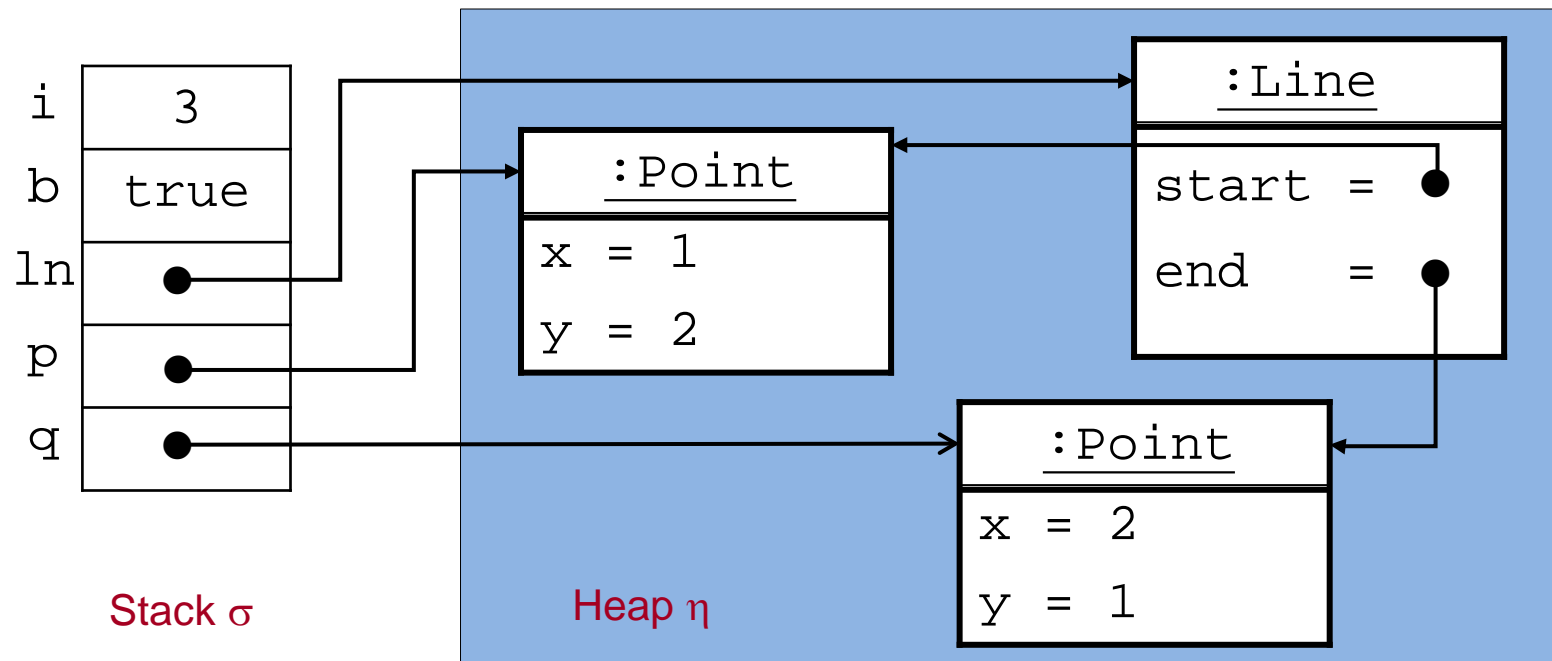
Lokale  
Variablen

Stack  $\sigma$



Heap  $\eta$

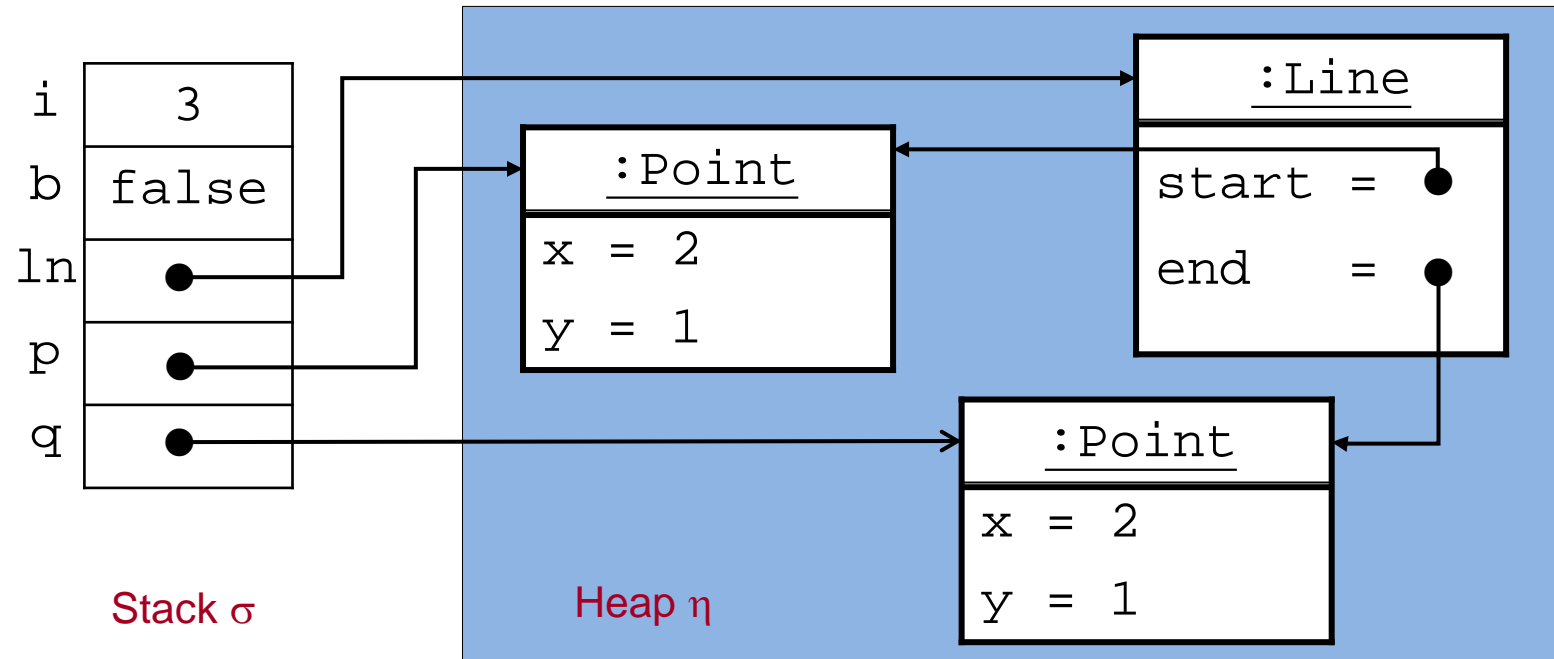
## Zustand mit Zeigerdarstellung



### Beachte:

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

## Gleichheit von Objektreferenzen



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

## Wdh: Überblick Kapitel 3 und 5

### *Kapitel 5*

Klassendeklarationen

Objekte und Objekthalde (Heap)

Klassentypen

Referenzen und `null`

`==`, `!=` für Referenzen und `null`

**Attributzugriff,  
Methodenaufruf mit Ergebnis,  
Objekterzeugungsausdruck**

Objekthalde (Heap)

### *Kapitel 3*

Grunddatentypen

erweitert um

Werte

erweitert um

Operationen

erweitert um

Ausdrücke

**erweitert um**

Typisierung

Auswertung bzgl.

Zustand (Stack)

erweitert um



## 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* = *Identifizier*

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

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

## Grammatik für Methodenaufruf- und Objekterzeugungs-Ausdrücke

*MethodInvocation* =

*Expression* "." *Identifer* "(" [*ActualParameters*] ")" "

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

*InstanceCreation* = *ClassInstanceCreation*

*ClassInstanceCreation* =

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

## 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  $e =_{(\sigma, \eta)} \dots$
- 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* "." *Identifizier*

- Der Ausdruck *Expression* muss einen Klassentyp haben und der *Identifizier* 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 *Identifizier*.

### Beispiel:

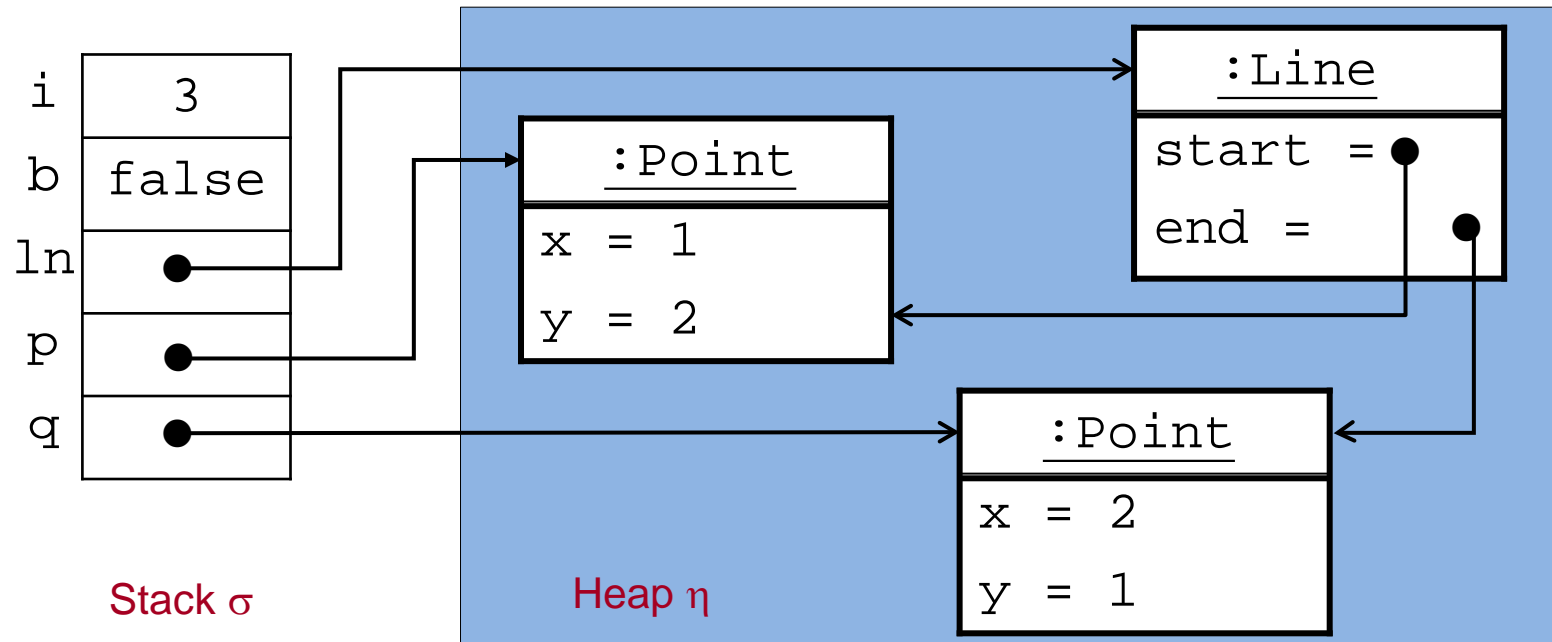
Seien `Point p`; `Line ln`; lokale Variable.

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

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

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

## Attributzugriff: Auswertung



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

### Achtung:

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

## Methodenaufruf-Ausdruck

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

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

Ein Methodenaufruf-Ausdruck hat also die Form  $e.m(a_1, \dots, a_n)$

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

$$\text{Type } m (T_1 \ x_1, \dots, T_n \ x_n) \{ \text{body} \}$$

- Die **aktuellen Parameter**  $a_1, \dots, a_n$  sind Ausdrücke, die in Anzahl und Typ zu den formalen Parametern der Methodendeklaration passen müssen.
- Der Ausdruck  $e.m(a_1, \dots, a_n)$  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  $(\sigma, \eta)$  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  $\text{new } C(a_1, \dots, a_n)$

- $C$  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:

$$C(T_1 \ x_1, \dots, T_n \ x_n) \{ \text{body} \}$$

- Die aktuellen Parameter  $a_1, \dots, a_n$  sind Ausdrücke, deren Typen zu den Typen  $T_1, \dots, T_n$  passen müssen.
- Der Ausdruck  $\text{new } C(a_1, \dots, a_n)$  hat dann den Typ  $C$ .

Beachte: Wenn in einer konkreten Klasse kein Konstruktor deklariert wurde, dann kann man einen „Standard-Konstruktor“  $C()$  ohne Parameter verwenden, mit dem ein Objekt der Klasse ohne explizite Initialisierung der Attribute erzeugt wird.



## Objekterzeugungs-Ausdruck: Beispiele und Auswertung

Sei `int i`; eine lokale Variable.

`new Point()` hat den Typ `Point`, //Annahme: Es gibt keinen benutzer-definierten Konstruktor

`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.

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