LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung

Wintersemester 2018/19

Kapitel 5: Objekte und Klassen

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 14. November 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- Einige Grundbegriffe **Objektorientierter Programmierung** kennenlernen
- **Klassen** in Java deklarieren
- Das **Speichermodell** für Objekte
- Typen, Ausdrücke und deren Auswertung im **Kontext von Klassendeklarationen**

Überblick Kapitel 3 und Kapitel 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 (einschließlich Typisierung und Auswertung)	erweitert um	Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck
Auswertung bezüglich Zustand (Stack)	erweitert um	Objekthalde (Heap)

Die objektorientierte Programmierung

In der objektorientierten Programmierung werden **Daten und Methoden**, die Algorithmen implementieren, zu geschlossenen Einheiten (**Objekten**) zusammengefasst.

Beispiele:

- Bankkonto

  Daten: Kontostand, Zinsatz

  Methoden: einzahlen, abheben, ...

- Punkte, Linien, Kreise in einem Zeichenprogramm

  Daten: geometrische Form

  Methoden: verschieben, rotieren ...

Die objektorientierte Programmierung (2)

- 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

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 charakterischen 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.

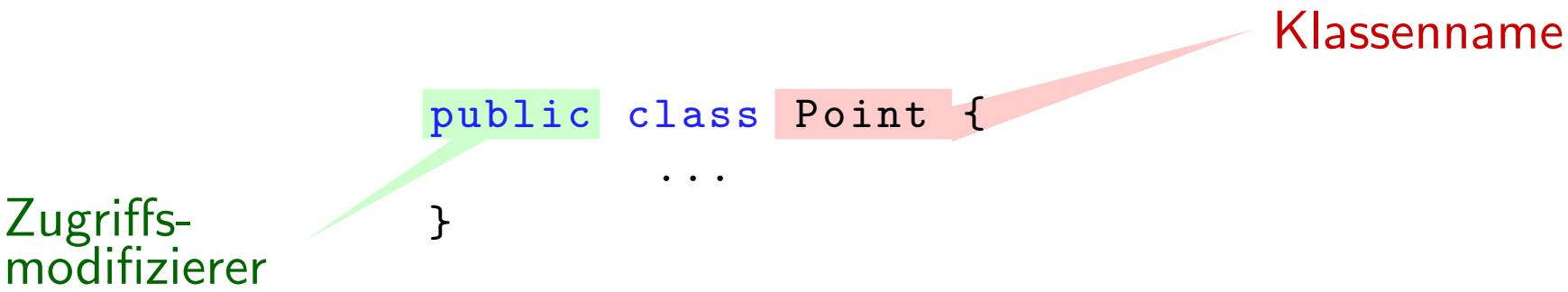
Beispiel: Klasse für Punkte

```
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 += dx;
                this.y += dy;
        }
        public int getX(){
                return this.x;
        }
        public int getY(){
                return this.y;
        }
}
```

Beispiel: Klasse für Punkte (2)

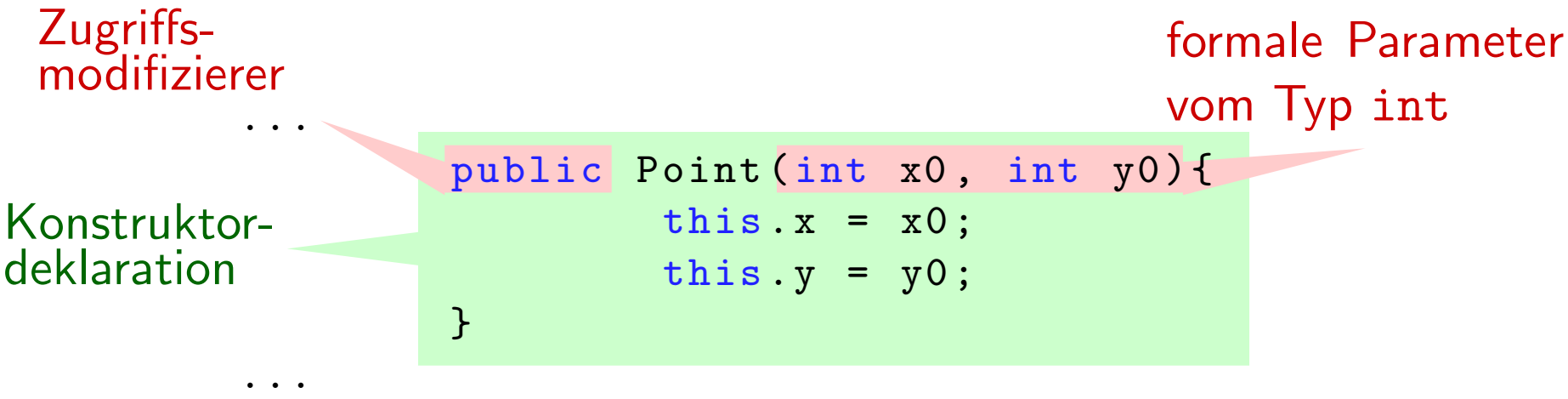


- Zugriffsmodifizierer für Klassen:
  Jede mit `public` gekennzeichnete Klasse muss in einer eigenen Datei gespeichert werden, die `Klassenname.java` heißt.
- public bedeutet „Zugriff erlaubt“,
  ohne public ist der Zugriff eingeschränkt

Beispiel: Klasse für Punkte (3)

- Deklaration zweier **Attribute** `x` und `y`, beide vom Typ `int`
- Zugriffsmodifizierer für Attribute:
  - `private`: Zugriff nur innerhalb der Klasse
  - `public`: Würde beliebigen Zugriff von außen erlauben
  - ... (es gibt noch weitere)
- Wichtiger Begriff hier ist die **Kapselung**:
  Verhindere den direkten Zugriff auf Attribute von außen.
  Erlaube Zugriffe nur über Methoden.
  Grund: Z.B. um die „interne“ Implementierung zu verbessern, ohne bestehenden Code unbrauchbar zu machen.

Beispiel: Klasse für Punkte (4)



- Konstruktoren sind Methoden zum Erzeugen eines Objekts der Klasse (sie „konstruieren“ ein Objekt)
- Der Methodenname entspricht dabei dem Klassennamen.
- Es kann mehrere Konstruktoren mit unterschiedlichen Argumenten geben. Z.B. zusätzlich:

```
public Point () {
      this.x = 0;
      this.y = 0
}
```

Beispiel: Klasse für Punkte (5)

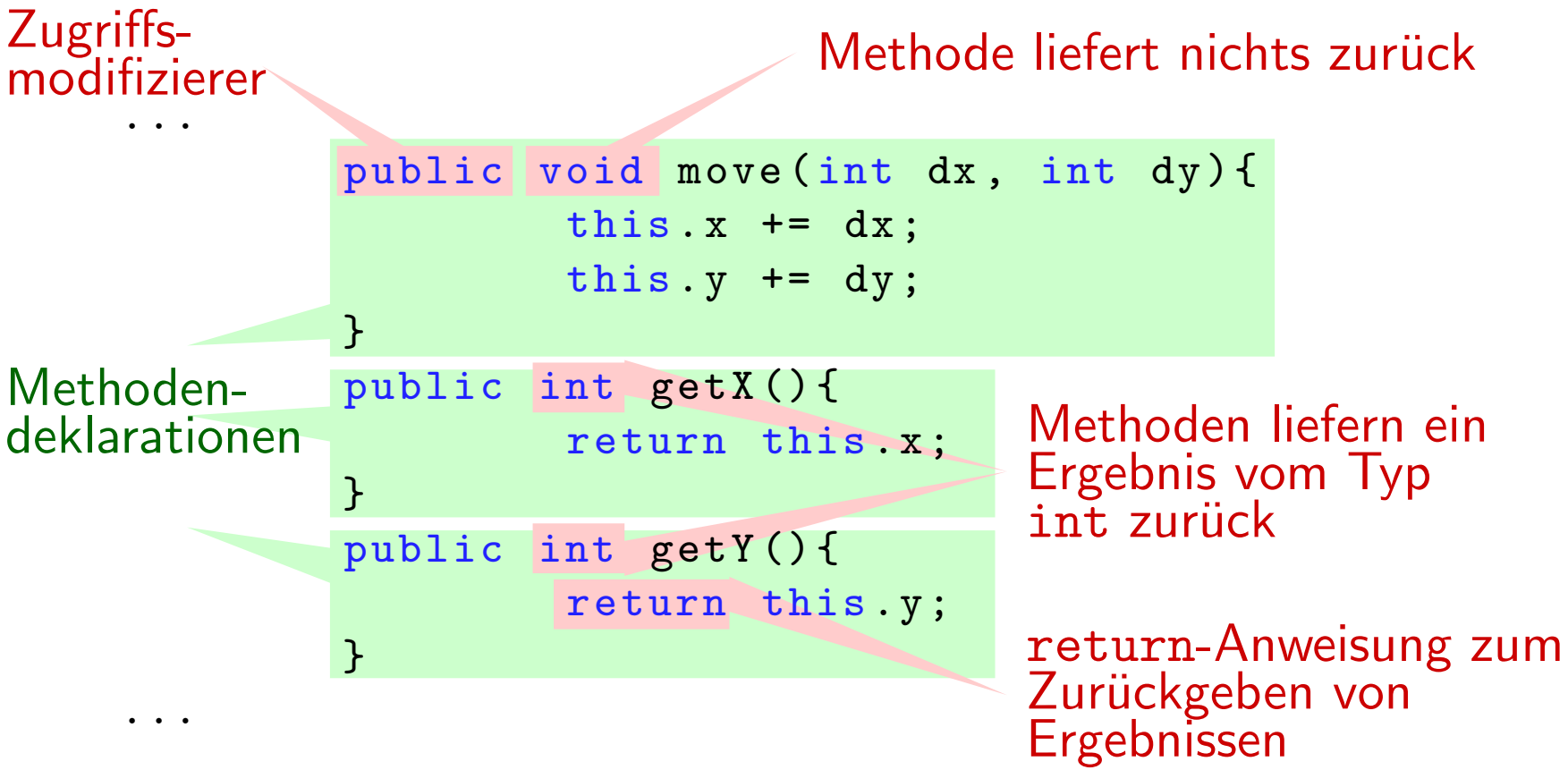
vordefinierte lokale Variable `this`

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

Zugriff auf y-Koordinate von `this`

- `this` bezeichnet immer das aktuelle Objekt selbst.
- Mit `this.x` wird das Attribut `x` des aktuellen Objekts bezeichnet.

Beispiel: Klasse für Punkte (6)



- Methoden haben einen Rückgabetyp
- `void` bedeuted „nichts“
- `return` zur Rückgabe von Ergebnis und Beenden der Methode
- Beachte: Konstruktoren haben keinen expliziten Rückgabetyp

Benutzung der Klasse Point in einer main-Methode

```
public class PointTest {
 public static void main(String[] args) {
  Point p1 = new Point(7,-4);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
  p1.move(-2,1);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
 }
}
```

Bemerkungen:

Benutzung der Klasse Point in einer main-Methode

```
public class PointTest {
 public static void main(String[] args) {
  Point p1 = new Point(7,-4);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
  p1.move(-2,1);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
 }
}
```

Bemerkungen:

- new erzeugt ein Objekt

Benutzung der Klasse Point in einer main-Methode

```
public class PointTest {
 public static void main(String[] args) {
  Point p1 = new Point(7,-4);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
  p1.move(-2,1);
  System.out.println("x-Koordinate von p1=" + p1.getX());
  System.out.println("y-Koordinate von p1=" + p1.getY());
 }
}
```

Bemerkungen:

- `new` erzeugt ein Objekt
- `p1.getX()`, `p1.getY()` und `p1.move(-2,1)` sind Methodenaufrufausdrücke.
- Beides wird später noch genauer betrachtet.

Mit Javadoc kommentierte Klasse „Point“

```
/**
 * Diese Klasse dient zu Repraesentation eines Punkts in der Ebene
 * @author David Sabel
 * @version 1.0
 *
 */
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 verschiebt den Punkt um dx auf der x-Achse
   * und dy auf der y-Achse
   * @param dx
   *    Distanz der Verschiebung auf der x-Achse
   * @param dy
   *    Distanz der Verschiebung auf der y-Achse
   */
   ...
```

Mit Javadoc kommentierte Klasse „Point“

```
...
  /**
   * Diese Methode liefert die x-Koordinate des Punkts zurueck
   * @return
   *   die x-Koordinate des Punkts
   */
  public int getX(){
    return this.x;
  }
  /**
   * Diese Methode liefert die y-Koordinate des Punkts zurueck
   * @return
   *   die y-Koordinate des Punkts
   */
  public int getY(){
    return this.y;
  }
}
```

Aufruf zum Erstellen der Dokumentation:

```
> javadoc -author -version Point.java
```

Durch Javadoc erzeugte HTML-Seite

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Point

java.lang.Object
    Point

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

Diese Klasse dient zu Repraesentation eines Punkts in der Ebene

Version:

1.0

Author:

David Sabel

Constructor Summary

Constructors

Constructor and Description
`Point(int x0, int y0)` Konstruktor eines Punkts.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
int	`getX()` Diese Methode liefert die x-Koordinate des Punkts zurueck
int	`getY()` Diese Methode liefert die y-Koordinate des Punkts zurueck
void	`move(int dx, int dy)` Diese Methode verschiebt den Punkt um dx auf der x-Achse und dy auf der y-Achse

Methods inherited from class java.lang.Object

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

Durch Javadoc erzeugte HTML-Seite (2)

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 verschiebt den Punkt um dx auf der x-Achse und dy auf der y-Achse

Parameters:

`dx - Distanz der Verschiebung auf der x-Achse`

`dy - Distanz der Verschiebung auf der y-Achse`

getX

```
public int getX()
```

Diese Methode liefert die x-Koordinate des Punkts zurueck

Returns:

`die x-Koordinate des Punkts`

getY

```
public int getY()
```

Diese Methode liefert die y-Koordinate des Punkts zurueck

Returns:

`die y-Koordinate des Punkts`

Spezielle Tags für Javadoc

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

Klasse für Linien (benutzt Point)

```
public class Line {
  private Point start, 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);
    return Math.sqrt(diffX * diffX + diffY * diffY);
  }
}
```

Grammatik für Klassendeklarationen (ohne Vererbung)

ClassDecl = ["public"] "class" Identifier ClassBody
ClassBody = "{" { FieldDecl | ConstructorDecl | MethodDecl } "}"
FieldDecl = [Modifier] VariableDeclaration
Modifier = "public" | "private"
MethodDecl = Header Block
Header = [Modifier] (Type | "void") Identifier "(" [FormalParam] ")"
FormalParam = Type Identifier { "," Type Identifier }

- ConstructorDecl ist analog zu MethodDecl, jedoch ohne (Type | "void") im Header.
  Der Identifier muss hier gleich zum Klassennamen sein.
- Methoden, deren Header einen Ergebnistyp Type hat, nennt man Methoden mit Ergebnis(typ)

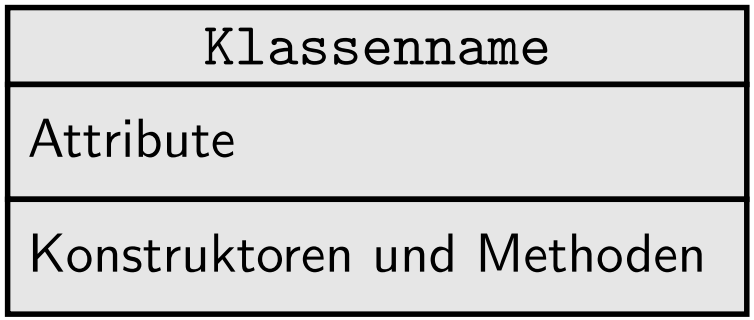
UML-Darstellung von Klassen

- UML = Unified Modeling Language
- Referenz `http://uml.org/`
- Grafische Modellierungssprache zur Spezifikation von Software-Teilen
- Wir lernen einige grundlegende Notationen von UML im Laufe der Veranstaltung.

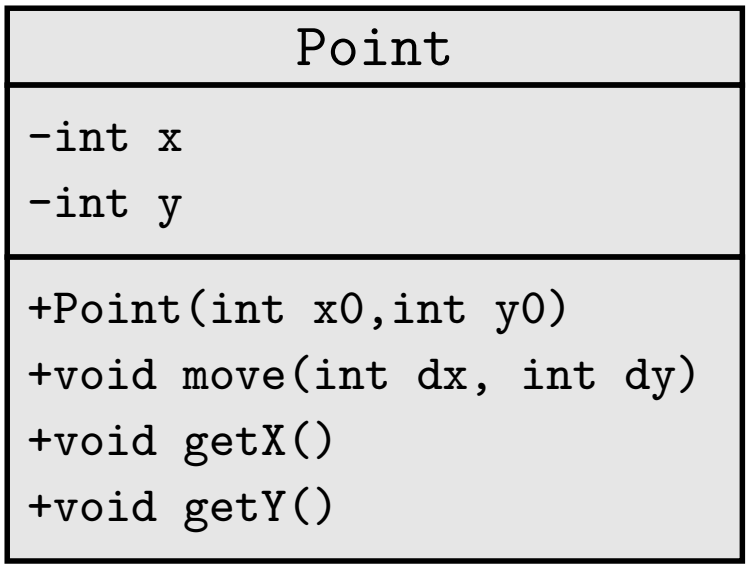
UML-Darstellung von Klassen

- UML = Unified Modeling Language
- Referenz `http://uml.org/`
- Grafische Modellierungssprache zur Spezifikation von Software-Teilen
- Wir lernen einige grundlegende Notationen von UML im Laufe der Veranstaltung.

Ein **Klassendiagramm** in UML hat die Form



Beispiel:

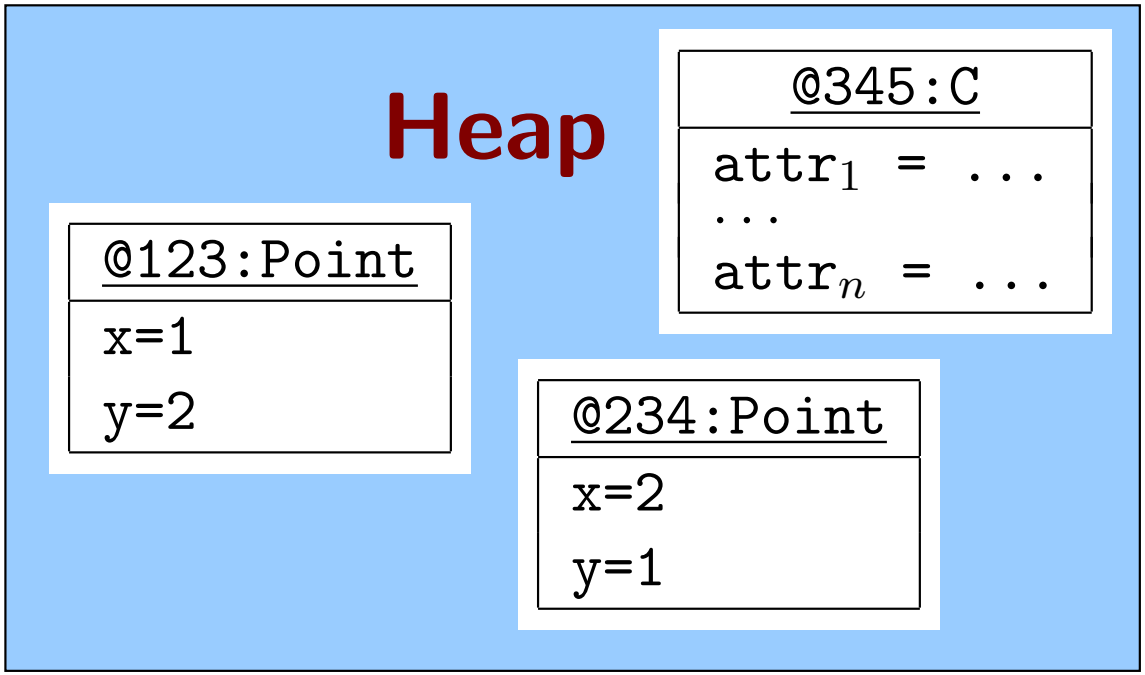


Beachte: + steht für `public`, - steht für `private`, # steht für `protected`

Objekte und ihre Speicherdarstellung

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

Objekte und ihre Speicherdarstellung (2)



Abstrakte Darstellung des Heaps:

$$\{\langle(\texttt{@123}, \texttt{Point}), [(\texttt{x}, 1), (\texttt{y}, 2)]\rangle,$$
$$\langle(\texttt{@234}, \texttt{Point}), [(\texttt{x}, 2), (\texttt{y}, 1)]\rangle,$$
$$\langle(\texttt{@345}, \texttt{C}), [(\texttt{attr}_1, \ldots), \ldots, (\texttt{attr}_n, \ldots)]\rangle\}$$

Überblick Kapitel 3 und Kapitel 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 (einschließlich Typisierung und Auswertung)	erweitert um	Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck
Auswertung bezüglich Zustand (Stack)	**erweitert um**	**Objekthalde (Heap)**

Klassentypen

Im Folgenden werden die in Kapitel 3 eingeführten Konzepte für Typen und Ausdrücke **erweitert**.

$$\begin{aligned} \text{Type} &= \text{PrimitiveType} \mid \text{ClassType} \\ \text{ClassType} &= \text{Identifier} \end{aligned}$$

- Mit jeder Klassendeklarationen wird ein neuer Typ eingeführt (**Klassentyp**), der den Namen der Klasse hat

Klassentypen (2)

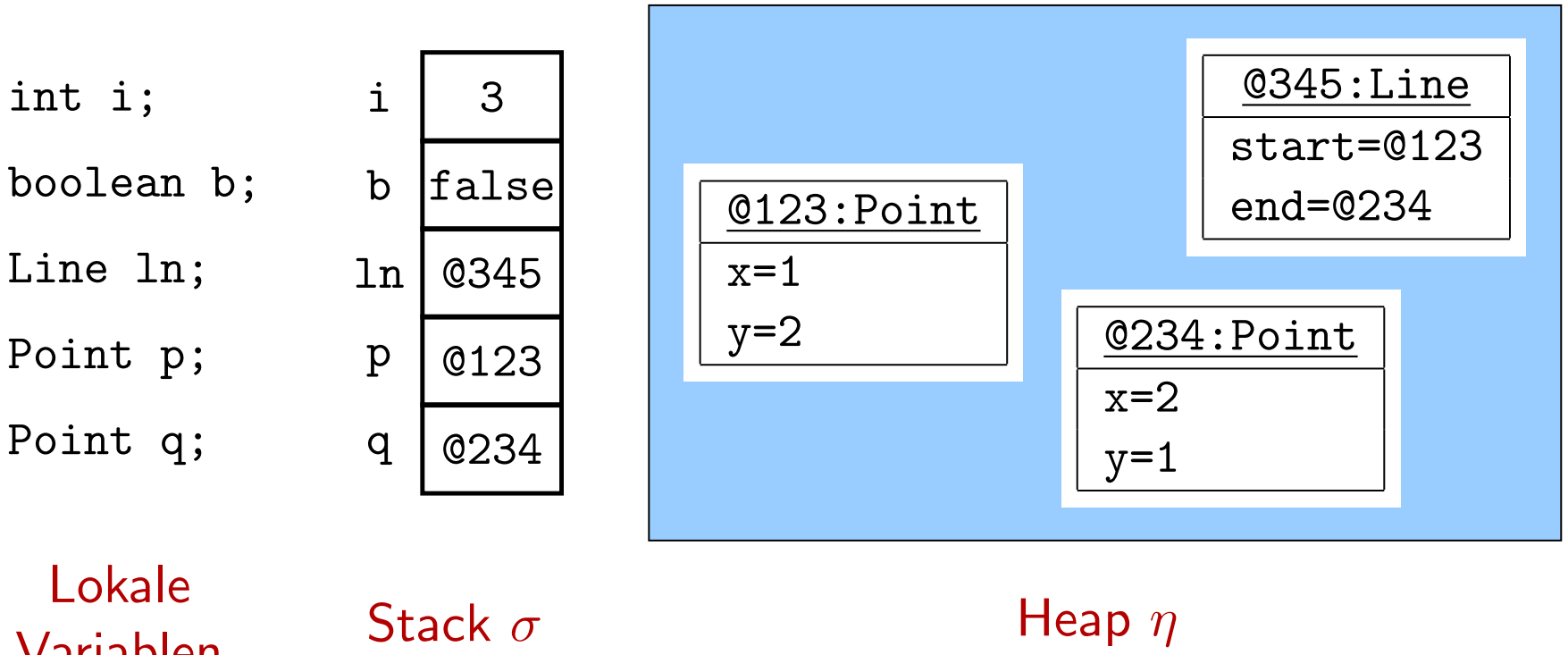
- Die **Werte** eines Klassentyps sind **Referenzen** (Verweise, Zeiger, Links) auf Objekte der Klasse sowie das Element `null` („leere“ Referenz)
- Dementsprechend speichern lokale Variablen eines Klassentyps Referenzen auf Objekte oder den Wert `null`.
- Objekt-Referenzen können mit den Operationen == und != auf Gleichheit bzw. Ungleichheit getestet werden.
  (**Achtung:** Die Zeiger nicht die Inhalte werden erglichen)

Beachte: Objekte einer Klasse $\neq$ Werte des Klassentyps K

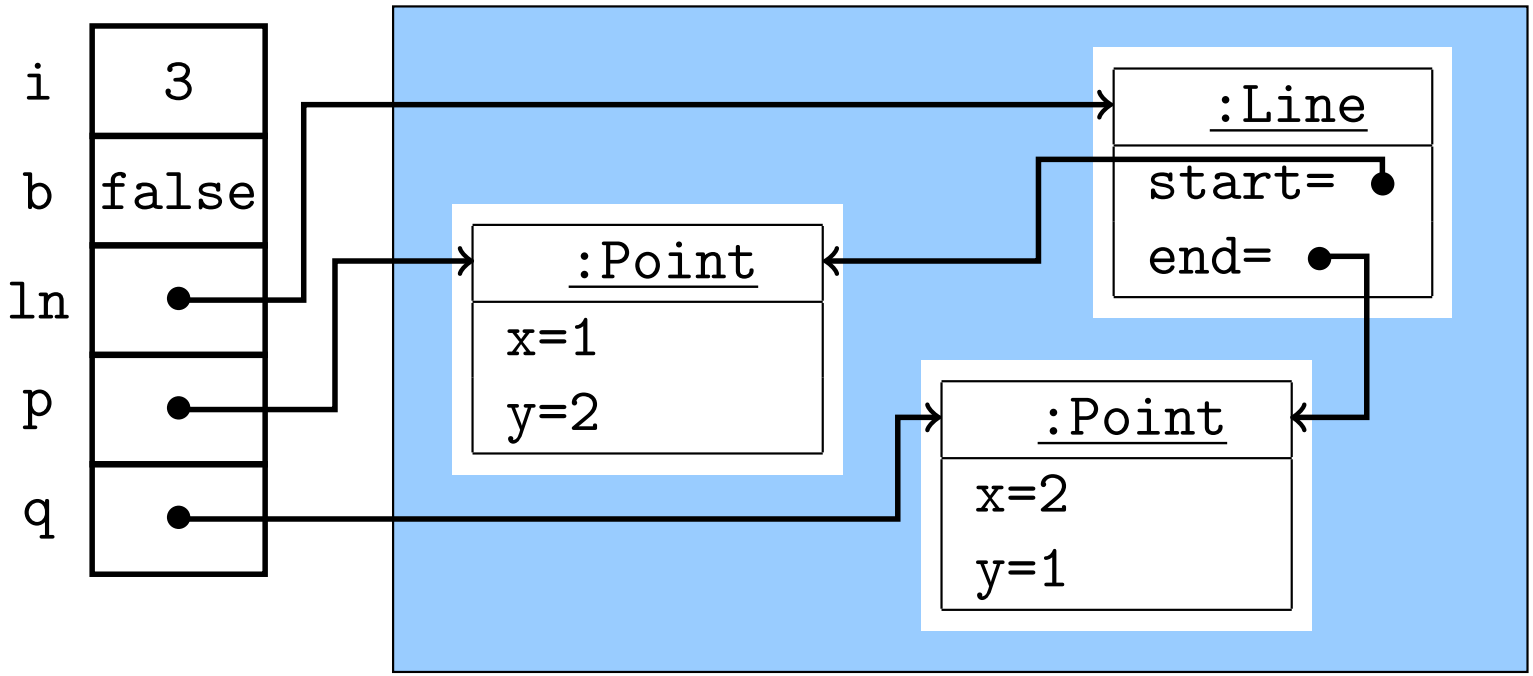
Zustand = Stack + Heap

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

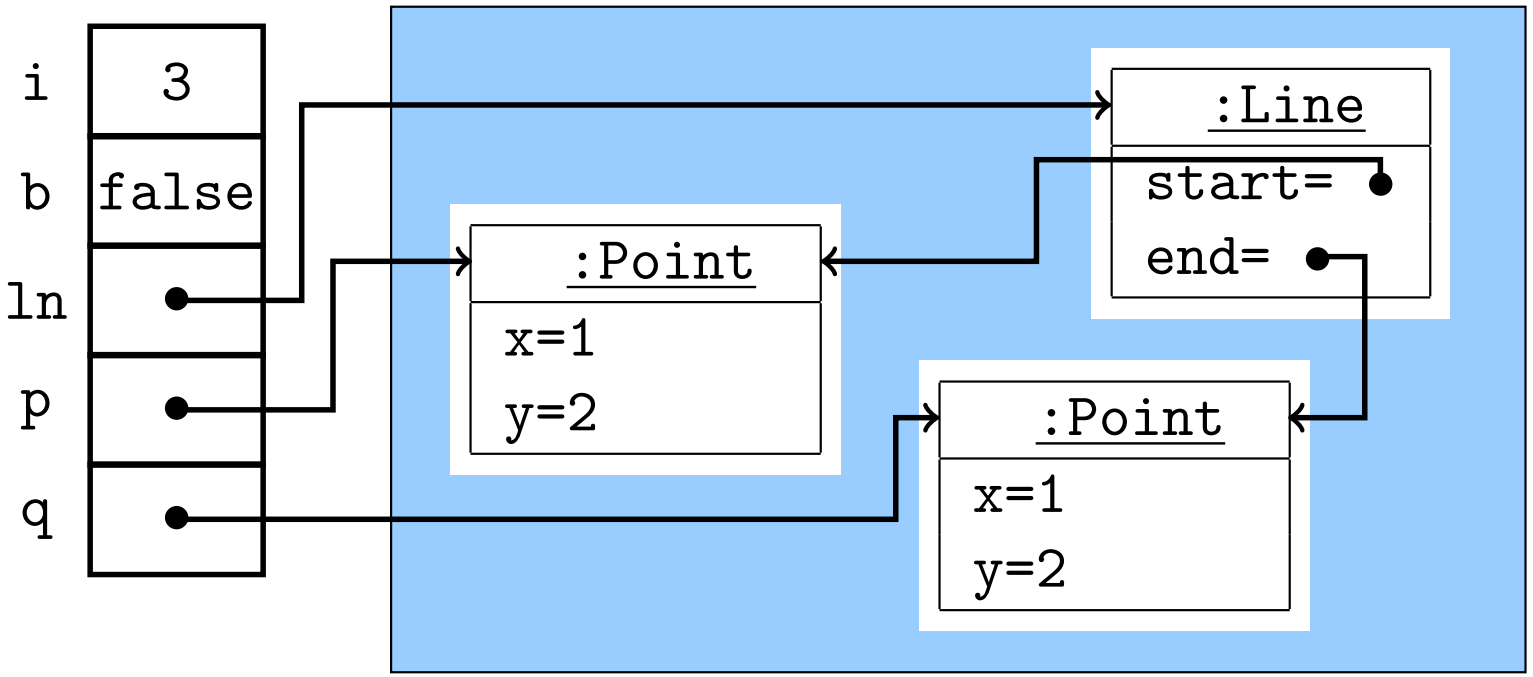


Stack σ Heap η

Beachte:

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

Gleichheit von Objektreferenzen



Stack σ Heap η

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

Überblick Kapitel 3 und Kapitel 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 (einschließlich Typisierung und Auswertung)	**erweitert um**	**Attributzugriff, Methodenaufruf mit Ergebnis, Objekterzeugungsausdruck**
Auswertung bezüglich Zustand (Stack)	erweitert um	Objekthalde (Heap)

Erweiterte Grammatik für Ausdrücke im Kontext von Klassendeklarationen

```
Expression     = Variable | Value | Expression BinOp Expression
               | UnOp Expression | "(" Expression ")"
               | Expression "?" Expression ":" Expression
               | MethodInvocation
               | InstanceCreation
Variable       = NamedVariable | FieldAccess
NamedVariable  = Identifier
FieldAccess    = Expression "." Identifier
Value          = IntegerValue | FloatingPointValue
               | CharacterValue | BooleanValue | "null"
```

Syntax für Methodenaufruf- & Objekterzeugungsausdrücke

MethodInvocation = Expression "." Identifier "(" [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** (σ, η), d.h. wir berechnen $e =_{(\sigma,\eta)} \ldots$
- Der Attributzugriff mit . und der Methodenaufruf mit . haben die höchte Präzedenz 15.

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 $e =_{(\sigma,\eta)} \ldots$
- Der Attributzugriff mit `.` und der Methodenaufruf mit `.` haben die höchte 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

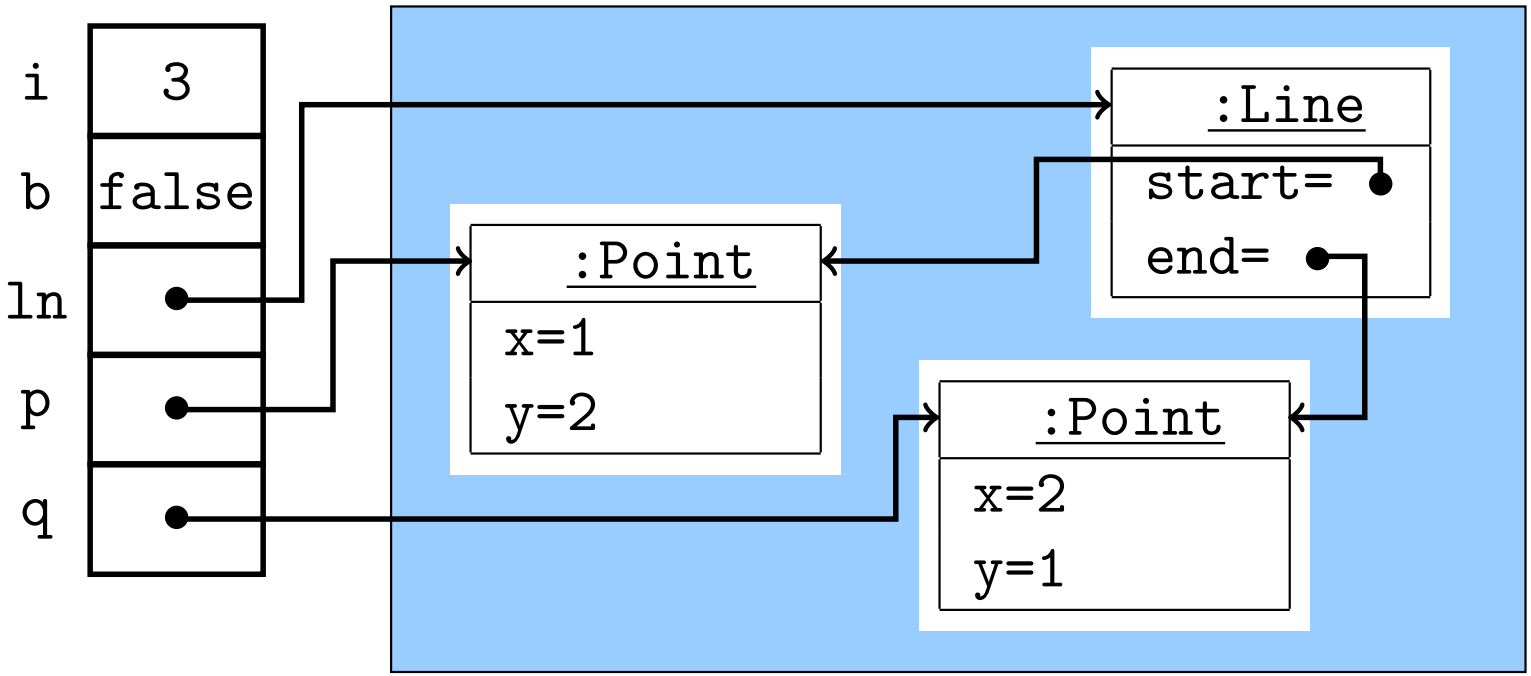
- Der Ausdruck Expression muss einen Klassentyp haben und der Identifier muss ein Attribut der Klasse (oder einer Oberklasse, vgl. später) bezeichnen.
- Das Attribut muss im aktuellen Kontext sichtbar sein.
- FieldAccess hat dann denselben Typ wie das Attribut Identifier

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



Stack σ Heap η

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

Attributzugriff: Auswertung (2)

Beachte: Falls kein Objekt referenziert wird,

$$\text{z.B. falls } \texttt{p} =_{(\sigma,\eta)} \texttt{null},$$

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(`a_1`,...,`a_n`)`

Methodenaufruf-Ausdruck

MethodInvocation = Expression "." Identifier "(" [ActualParameters] ")"
ActualParameters = Expression { "," Expression }

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

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

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

Methodenaufruf-Ausdruck

MethodInvocation = Expression "." Identifier "(" [ActualParameters] ")"
ActualParameters = Expression { "," Expression }

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

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

  `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.

Methodenaufruf-Ausdruck

MethodInvocation = Expression "." Identifier "(" [ActualParameters] ")"
ActualParameters = Expression { "," Expression }

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

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

  `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.

- Der Ausdruck `e.m(`a_1`,...,`a_n`)` hat dann als **Typ den Ergebnistyp** der Methode.

Methodenaufruf-Ausdruck: Beispiele und Auswertung

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

- `p.getX()` hat den Typ `int`,
- `ln.start.getY()` hat den Typ `int`

Sei (σ, η) der Zustand von oben.

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

Methodenaufruf-Ausdruck: Beispiele und Auswertung

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

- `p.getX()` hat den Typ `int`,
- `ln.start.getY()` hat den Typ `int`

Sei (σ, η) der Zustand von oben.

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

Bemerkungen:

- Die Berechnung der Ergebnisse von Methdodenaufrufen basiert auf der Ausführung von Methodenrümpfen (vgl. später)
- Im allgemeinen ist es möglich, dass der Aufruf einer Methode nicht nur einen Ergebniswert liefert, sondern auch eine Zustandsänderung bewirkt (vgl. Kapitel 6)

Objekterzeugungsausdruck

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

Ein Objekterzeugungsausdruck hat also die Form `new C(`a_1`,...,`a_n`)`

Objekterzeugungsausdruck

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

Ein Objekterzeugungsausdruck hat also die Form `new C(`a_1`,...,`a_n`)`

- `C` muss eine deklarierte Klasse sein.

Objekterzeugungsausdruck

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

Ein Objekterzeugungsausdruck 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}`

Objekterzeugungsausdruck

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

Ein Objekterzeugungsausdruck 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.

Objekterzeugungsausdruck

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

Ein Objekterzeugungsausdruck 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`

Objekterzeugungsausdruck (2)

Beachte:

Wenn in der 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.

Objekterzeugungsausdruck: 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`

Objekterzeugungsausdruck: Beispiele und Auswertung (2)

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 benutzerdefinierten Konstruktors ausgeführt und damit den Attributen `x` und `y` des neu erzeugten Objekts die Werte `1` und `2` zugewiesen.

(Allgemeine Vorschrift zur Ausführung von Objekterzeugung folgt in Kapitel 6).

Zusammenfassung

- Klassen: Attribute, Methoden
- Objekte: Instanzen von Klassen
- Klassendeklarationen
- Speicherdarstellung von Objekten
  (Zustand als Paar von Stack und Heap)
- Methodenaufruf und Objekterzeugung
- Typisierung von Methdodenaufrufen und Objekterzeugung