

*First things first,
but not necessarily in that order.
- Dr. Who, Meglos*

Klassen

Martin Wirsing

in Zusammenarbeit mit
Matthias Hölzl, Piotr Kosciuzenko, Dirk Pattinson

Ziele

- Verstehen des Syntax einer Java-Klasse und ihre graphische Beschreibung in UML
- Verstehen des Speichermodells von Java
- Lernen Objekte zu erzeugen und einfache Methoden zu schreiben

Einfache Klassen in Java und UML

▪ Objekte

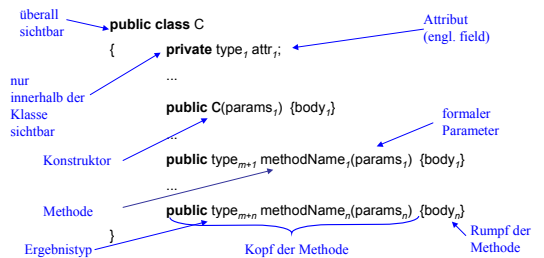
Objekte sind kleine Programmstücke.
Jedes Objekt hat spezifische Fähigkeiten.
Objekte kooperieren, um eine umfangreiche Aufgabe gemeinsam zu erfüllen

▪ Klassen

Klassen sind Fabriken für Objekte.
Jede Klasse kann einen ganz bestimmten Typ von Objekten erzeugen.
Jedes Objekt gehört zu genau einer Klasse; es ist **Instanz** dieser Klasse

Einfache Klassen in Java und UML

Eine Klassendeklaration in Java hat die Gestalt



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;
    }

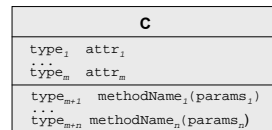
    public int getX()
    {
        return this.x;
    }

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

Annotations in the image:
 - 'Klassenname' points to 'Point' in 'public class Point'.
 - 'vordefinierte lokale Variable this bezeichnet das gerade betrachtete Objekt' points to 'this.x' and 'this.y' in the constructor.
 - 'y-Koordinate von this' points to 'this.y' in the 'move' method.
 - 'Rückgabe des Ergebniswerts' points to 'return this.x;' and 'return this.y;'.

Einfache Klassen in Java und UML

In UML wird eine Klasse C folgendermaßen repräsentiert (angepasst an Java-Syntax):



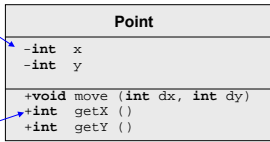
Bemerkung

- In UML wird eine Pascal-ähnliche Syntax für Attribut- und Methodendeklarationen verwendet. Wir haben dies zugunsten einer einheitlichen Syntax an Java angepaßt.
- Die Konstruktoren werden meist nicht im Klassendiagramm aufgeführt.
- Die Methodenrümpfe erscheinen nicht im UML-Klassendiagramm, da UML-Diagramme zur abstrakteren Repräsentation von Klassen verwendet werden.
- Man kann Methodenrümpfe als Notizen an das Diagramm hängen.

Beispiel:

In UML wird die Klasse `Point` folgendermaßen repräsentiert:

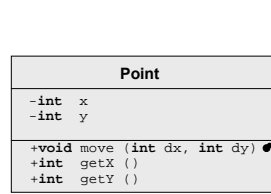
„-“ für private



„+“ für public

Notizen in UML

Beispiel

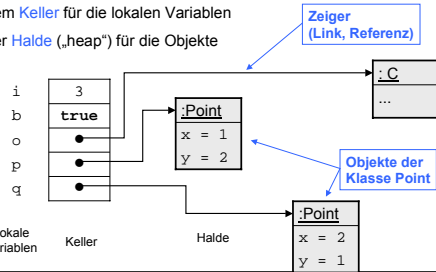


Notiz
(für Rumpf von move)

Objekte und deren Speicherdarstellung

Jeder Zustand eines sequentiellen Java-Programms besteht aus

- einem Keller für die lokalen Variablen
- einer Halde („heap“) für die Objekte



Objekte sind Referenztypen

- In einer Objektvariable wird nur ein Zeiger (Link, Referenz) auf das wirkliche Objekt gespeichert

▪ Point p; ← Zeiger auf Objekt von Point

- Die Java Maschine kümmert sich um Platz für die Daten des Objekts.

- Dazu gehört
 - Besorgung von zusätzlichem Platz bei Bedarf
 - Recyclen von nicht mehr benötigtem Platz
Dies heißt: Garbage Collection

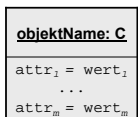
- Der Programmierer hat stets einen Link auf das Objekt zur Verfügung



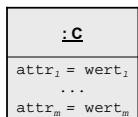
Objekte und deren Speicherdarstellung

In UML wird ein Objekt der Klasse C folgendermaßen repräsentiert:

Objektinstanz mit Name



Objektinstanz OHNE Name („anonymes“ Objekt)



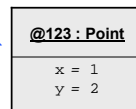
textuell:

objektName:C[attr1 = wert1, ..., attrm = wertm]
C[attr1 = wert1, ..., attrm = wertm]

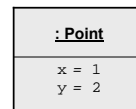
Objekte und deren Speicherdarstellung

Beispiel: Objekte der Klasse Point

Objektinstanz mit Name



Objektinstanz OHNE Name („anonymes“ Objekt)



Zwei VERSCHIEDENE Objekte der Klasse Point mit gleichen Attributwerten!

textuell:

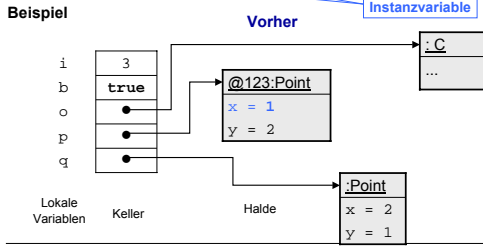
@123 : Point[x = 1, y = 2]
Point[x = 1, y = 2]

Objekte und deren Speicherdarstellung

Durch Zuweisung verändert sich der Wert der Instanzvariable

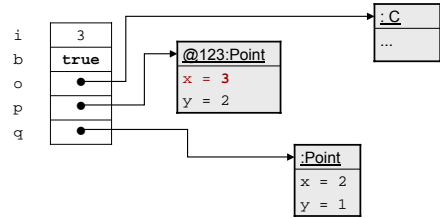
$$p.x = p.x + 2;$$

Beispiel



Objekte und deren Speicherdarstellung

Nachher

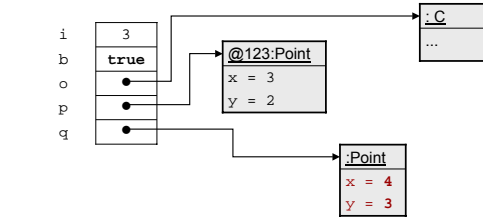


Objekte und deren Speicherdarstellung

Methodenaufruf verändert den Zustand

Beispiel

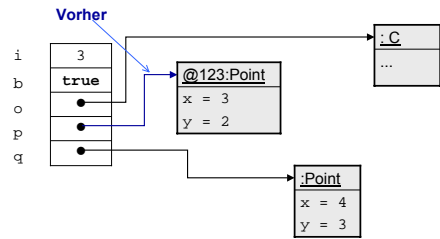
q.move(2, 2); ergibt



Zuweisung von Objekten

Bei Zuweisung von Objekten werden nur die Referenzen übernommen

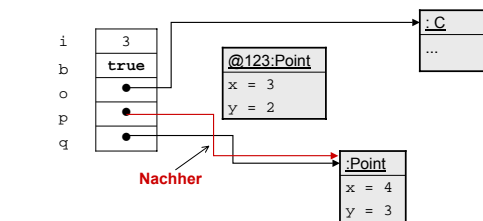
p = q;



Zuweisung von Objekten

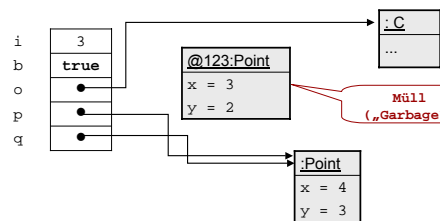
Bei Zuweisung von Objekten werden nur die Referenzen übernommen

p = q;



Datenmüll

- Nach der Zuweisung ist ein Objekt un erreichbar geworden
 - Kein Link zeigt mehr darauf
 - Es ist Müll (engl.: garbage) und wird automatisch vom Speicherbereinigungsalgorithmus („Garbage Collector“) gelöscht

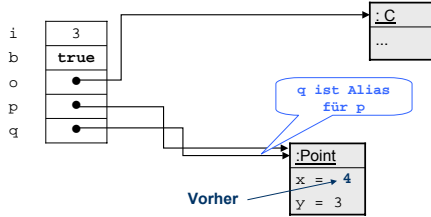


Beeinflußung, Manipulation

- Wenn zwei Referenzen auf das gleiche Objekt zeigen
 - Jedes kann die Felder des anderen beeinflussen
 - Ein Name ist dann ein sog. **Alias** für den anderen

Beispiel

q.x = 25; ändert die x-Koordinate von p

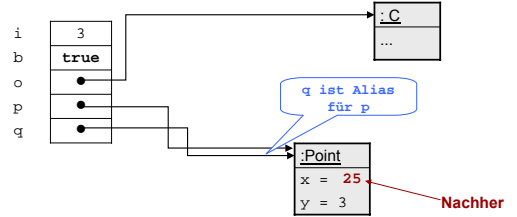


Beeinflußung, Manipulation

- Wenn zwei Referenzen auf das gleiche Objekt zeigen
 - Jedes kann die Felder des anderen beeinflussen
 - Ein Name ist dann ein sog. **Alias** für den anderen

Beispiel

q.x = 25; ändert die x-Koordinate von p



Erzeugung von Objekten und Konstruktoren

Will man einer lokalen Variablen `var` ein neues Objekt der Klasse C zuweisen, schreibt man

```
var = new C(x0);
```

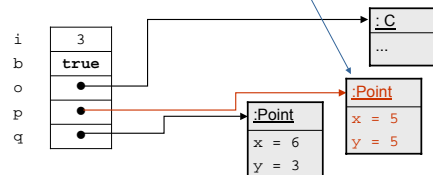
↖ Konstruktor
↘ erzeugt ein neues Objekt der Klasse C, auf das var zeigt und das durch den Konstruktoraufufruf C(x0) initialisiert wird

Erzeugung von Objekten: Beispiel Point

Konstruktor von Point erzeugt neues Point-Objekt

```
p = new Point(5,5);
```

ergibt



Konstruktor

- Ein Konstruktor dient zur Initialisierung der Attribute eines neu erzeugten Objekts.
- Ein Konstruktor hat den gleichen Namen wie seine Klasse
- Er ist keine Methode, da er nicht auf ein Objekt angewendet werden kann, sondern immer nach „new“ stehen muß.
- Beispiel: `p.Point(5,5)` ergibt einen Syntaxfehler.

Konstruktor

- Wird ein Attribut durch einen Konstruktor nicht explizit initialisiert, so wird es mit dem Standardwert seines Typs implizit initialisiert.

- Der Standardwert für

```
int           ist 0,
double        ist 0.0,
boolean       ist false,
C (Klassentyp) ist null,
```

wobei `null` das NICHT-ERZEUGTE Objekt repräsentiert.

- Beispiel: Deklariere Konstruktor `Point(){}` in Klasse `Point`

`p = new Point();` initialisiert die Attribute `x,y` des neuen Objekts mit 0.



Null

- `null` bezeichnet das NICHT-ERZUGTE Objekt ; es ist ein Zeiger, der auf **kein** Objekt zeigt.



Der Aufruf `null.m`; einer Methode mit dem null-Zeiger ist **verboten** und führt zu einem (Compilezeit-)Fehler.

- Ebenso führt

```
Point p1 = null;
// i 3
// p1 [ ]
```

und dann

```
p1.move(1,1);
```

zu einem Laufzeit-Fehler, bei dem eine „NullPointerException“ erzeugt wird und das Programm abbricht.

Mehrere Konstruktoren

- Mehrere Konstruktoren sind möglich
- Überladene Konstruktoren müssen aber eine unterschiedliche Parameterliste aufweisen, durch die sie eindeutig unterschieden werden können.

Beispiel:

Ein zweiter Konstruktor in der Klasse `Point` ist der Standardkonstruktor `Point()`

Implementierung einer UML-Klasse

Für jede Methode einer Klasse muß eine Implementierung (in Java) angegeben werden. Z.B. zur [Implementierung](#) der Klasse

Point	
<code>int</code>	<code>x</code>
<code>int</code>	<code>y</code>
<code>int</code>	<code>getX ()</code>
<code>int</code>	<code>getY ()</code>
<code>void</code>	<code>move (int dx, int dy)</code>

müssen für alle 3 Methoden Implementierungen durch Methodenrumpfe angegeben werden; außerdem müssen die Konstruktoren implementiert werden.

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: 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;
}
```

Benutzen von Klassen

Eine Klasse besteht aus einer Menge von Attributen und Methodenrumpfen. Um die Methoden ausführen zu können, braucht man eine Klasse mit einer Methode `main`. Im einfachsten Fall hat diese die Gestalt einer einfachen [Klasse](#).

Beispiel

```
public class PointMain
{
    public static void main (String[] args)
    {
        Point p = new Point(10,20);
        Point p1 = new Point();

        System.out.println("p=Point[x = "+p.getX()+", y = "+p.getY()+"]");
        System.out.println("p1=Point[x = "+p1.getX()+", y = "+p1.getY()+"]");

        p1.move(10,10);

        System.out.println("p1=Point[x = "+p1.getX()+", y = "+p1.getY()+"]");
    }
}
```

Benutzen von Klassen

Achtung:

- Wenn man zwei oder mehr Klassen in einer Datei vereinbart, darf genau eine dieser Klassen eine Methode `main` besitzen. Der Name der Datei muß der Name dieser Klasse mit Suffix `.java` sein

- Beispiel: Die Datei `PointMain.java` enthält eine Klasse `PointMain` mit Methode `main`. Mögliche andere Klassen in dieser Datei dürfen keine Methode `main` enthalten.

Zusammenfassung

- Klassen werden graphisch durch UML-Diagramme dargestellt und in Java implementiert.
- Jede Instanz-Methode hat einen impliziten Parameter – das Objekt mit dem die Variable aufgerufen wird – und 0 oder mehr explizite Parameter.
- Objekte werden mit dem `new`-Operator erzeugt, gefolgt von einem Konstruktor.
- Zahlartige Variablen haben Zahlen als Werte, lokale Variablen vom Objekttyp haben Zeiger (Referenzen) als Werte. Um Aliasing zu vermeiden, muß man die betreffenden Objekte kopieren.
- Der `null`-Zeiger zeigt auf **kein** Objekt. Der Aufruf einer Methode `null` (als implizitem Parameter) führt zu einem Fehler.
- Instanzvariablen (Attribute) werden in Java implizit initialisiert; dagegen müssen lokale Variablen explizit initialisiert werden.