Einführung in die Programmierung für NF

Klassen und Objekte

OBJEKTORIENTIERTE PROGRAMMIERUNG

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

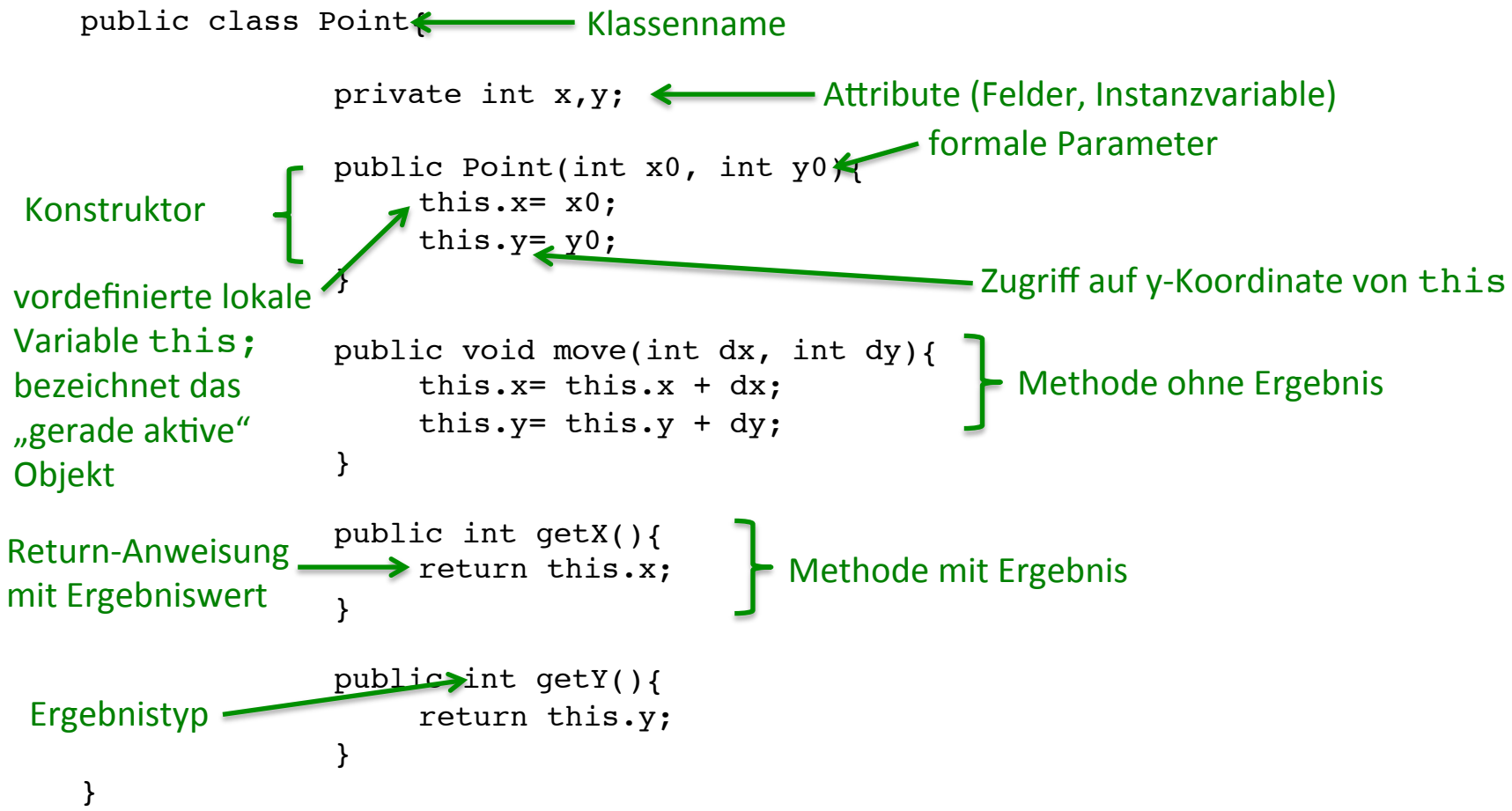
- Objekte
  - Objekte speichern Informationen (Daten)
  - Objekte können Methoden ausführen zum Zugriff auf diese Daten und zu deren Änderung
  - Während der Ausführung einer Methode kann ein Objekt auch Methoden bei (anderen) Objekten aufrufen

- 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;

     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“



Erzeugen von Objekten

- Mit der Anweisung

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

  wird ein neues Objekt der Klasse Point erzeugt.

- Die Attribute werden dabei mit Standardwerten belegt, genau wie bei der Erzeugung von Arrays. Zustand nach der obigen Anweisung:

p : Point
x = 0
y = 0

- Statt Point-Objekten könnte man auch int[]-Arrays der Länge 2 verwenden. Der Array-Typ enthält jedoch auch Arrays anderer Längen; p.x ist besser lesbar als p[0].
- Weiterhin können in Arrays nur Werte des gleichen Typs gespeichert werden. In ColoredPoint werden Daten verschiedener Typen zusammengefasst.

METHODEN IN DER OBJEKTORIENTIERTEN PROGRAMMIERUNG

Kontstruktormethode

- Meist möchte man die Felder eines Objekts direkt nach seiner Erzeugung initialisieren, d.h. nicht die Standardwerte nutzen.
- Eine Klasse kann spezielle Methoden, sogenannte Konstruktoren, definieren, in denen die Felder bei der Objekterzeugung gleich richtig initialisiert werden.

```
public class Point {
 public int x;
 public int y;

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

Konstruktor:
gleicher Name wie Klasse,
kein Rückgabetyp

- this bezeichnet das gerade betrachtete Objekt.
- Die Anweisung `Point p = new Point(4, 3);` erzeugt ein neues Point-Objekt und führt den obigen Konstruktor aus.
- Wenn kein Konstruktor definiert wird, erstellt Java einen parameterlosen Standardkonstruktor

Objekte — Methoden

- Objekte speichern nicht nur Daten, sie können auch Methoden für den Zugriff und zur Manipulation dieser Daten bereitstellen.

- Eine Klasse definiert die Methoden, die mit allen Objekten dieser Klassen benutzt werden können.
- Objektmethoden können eine vordefinierte lokale Variable this benutzen, die das gerade betrachtete Objekt bezeichnet.
- Objektmethoden können nicht ohne ein Objekt ausgeführt werden, im Gegensatz zu statischen Methoden.
- Aufruf einer Objektmethode: object.methodenName(args), z.B.:

```
Point p = new Point();
p.x = 3; p.y = 4;
p.move(-1, 2);
```

```
public class Point {
   public int x;
   public int y;
public void move(int dx, int dy) {
    this.x = this.x + dx;
    this.y = this.y + dy;
   }
public double distance(Point q) {
    int dx = this.x - q.x;
    int dy = this.y - q.y;
    return Math.sqrt(dx*dx+dy*dy);
   }
}
```

Warum Objektmethoden und nicht so?

```
public class Point {
    public int x;
    public int y;
}
```

Funktioniert nicht mit private

```
public class PointFunctions {
    static void move(Point p, int dx, int dy) {
        p.x = p.x + dx;
        p.y = p.y + dy;
    }

    static double distance(Point p, Point q) {
        int dx = p.x - q.x;
        int dy = p.y - q.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

- **Antwort**: Eine wichtige Idee der objektorientieren Programmierung ist das **Verstecken von Informationen.**
- Man deklariert Attribute/Felder **so gut wie immer** als private.
- Der Zugriff auf die Felder erfolgt **nur** über die Objektmethoden.

KAPSELUNG

Kapselung und das Verstecken von Information

- Warum ist das Verstecken von Informationen so wichtig?
  - Modularität
  - Änderbarkeit
  - Abstraktion von Implementierungsdetails

- Die **Kapselung von Daten** ist eine der grundlegenden Ideen der objektorientierten Programmierung:
  - Ein Objekt sollte seine Schnittstelle vollständig von seiner Implementierung trennen.
  - Man sollte ein Objekt benutzen können, ohne etwas über seine Implementierung zu wissen. Ein Objekt ist wie eine „Black Box“.
  - Solange die Schnittstelle eines Objekts gleich bleibt, kann die Implementierung beliebig verändert werden.

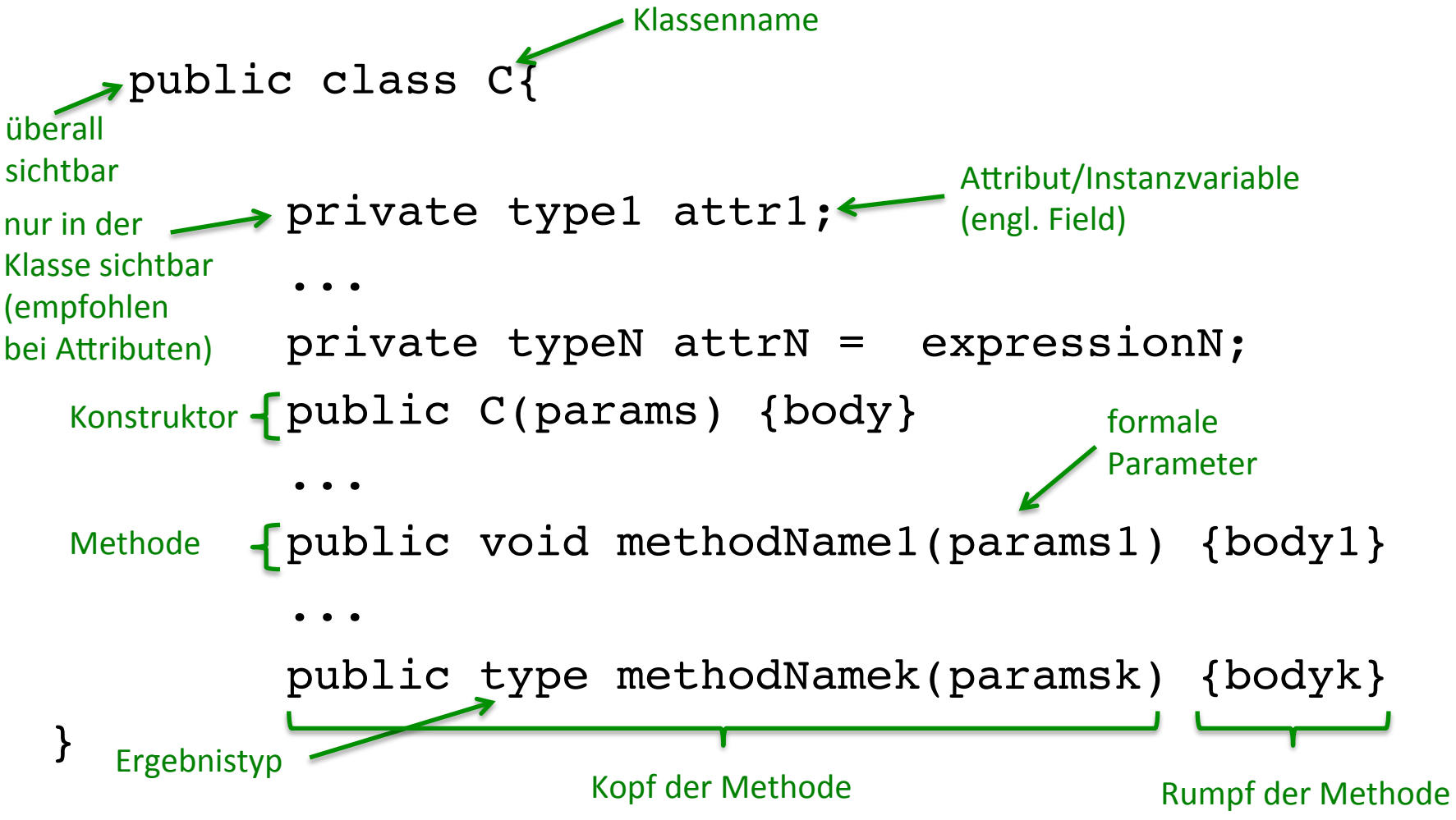
Kapselung

- Implementierungsdetails von Objekten werden versteckt (gekapselt).
- Objektmethoden stellen eine Schnittstelle zum Zugriff auf diese Daten dar.
- Vorteile:
  - Modularität: Ein Programm wird in Module mit klar definierten Schnittstellen aufgeteilt (z.B. IntSet), die dann unabhängig voneinander entwickelt werden können.
  - Änderbarkeit: Programmteile können ausgetauscht und verbessert werden, ohne dass Änderungen an anderen Teile des Programms nötig wären.
  - Softwarequalität: Kapselung fördert die genaue Spezifikation von Schnittstellen und dem Verhalten von Programmteilen. Dadurch wird systematisches Testen und die Verifikation von Programmen möglich.

Beispiel: Klasse „Line“

```
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));
	}
}
```

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)**.

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 l;` lokale Variable.
- `p.x` hat den Typ `int`,
- `l.start` hat den Typ `Point`,
- `l.start.y` hat den Typ `int`.

Grammatik für Methodenaufruf – und Objekterzeugungs-Ausdrücke

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

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

InstanceCreation = *ClassInstanceCreation*

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

Null-Objekt

- spezielle Referenz: null
- Literal null lässt sich zur Initialisierung von Referenzvariablen verwenden
- null-Referenz ist typenlos, kann also jeder Referenzvariablen zugewiesen und jeder Methode übergeben werden, die ein Objekt erwartet.
  → null verhält sich also so, als ob es Untertyp jeden anderen Typs wäre.

- Daher ist Folgendes gültig:

```
Point  p = null;
String s = null;
System.out.println( null );
```

- Da es nur ein null gibt, ist zum Beispiel `(Point) null == (String) null`.
- null hat viele Einsatzgebiete
  - Der Haupteinsatz sieht vor, damit uninitialisierte Referenzvariablen zu kennzeichnen, also auszudrücken, dass eine Referenzvariable auf kein Objekt verweist.
  - In Listen oder Bäumen kennzeichnet null aber auch das Fehlen eines gültigen Nachfolgers; null ist dann ein gültiger Indikator und kein Fehlerfall.
- Ist bei einem Methodenaufruf der Wert des Objekts `null`, erfolgt ein Laufzeitfehler

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 Instanz-Variablen 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++;
            }
            ...
}
```

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!).

Umwandlungen in Strings

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`

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 = ... ; String s = Integer.toString(x);`
- 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`.

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 = ... ; int x = Integer.parseInt(s);`

Der String `s` muss eine ganze Zahl repräsentieren; 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).

Vielen Dank für Ihre Aufmerksamkeit