Informatik B

Vorlesung 1

Objektorientierte Programmierung und UML

Organisation

- Übungen: Dipl.-Math. Dorothee Langfeld
- Übungsblätter
  - Ausgabe jeweils am Dienstag
  - Online auf der Webseite:
    http://www-lehre.inf.uos.de/~binf
  - Ein „Freischuss“
- Testate in Raum 31/339
- Scheinerwerb durch bestandene Klausur
- Mailingliste: http://rb.inf.uos.de
- Bitte in StudIP eintragen (Übung und Vorlesung)
- Vorlesungsaufzeichnung

Termine

- Vorlesung:
  - Montag: 16:15 Uhr, 31/E06
  - Dienstag: 12:15 Uhr 31/E06
- Übungen:
  - Donnerstags12:00 - 13:30, 31/449a
  - Donnerstags14:00 - 15:30, 31/E05
- Testate:
  - Montag bis Mittwoch (13:30 Uhr letztes Testat)
  - Eintragen für Testattermine: Tür Raum 31/339
- Klausur:
  - Montag, 16.07.2007, 10:00 Uhr (st!), 01/E01+E02

Inhalt der Veranstaltung

- OOP
- Java 5 (ev. Java 6)
  - Dateioperationen
  - Multithreading
  - Eventhandling
  - GUI-Programmierung
  - Netzwerkprogrammierung
  - ...
- Designpattern

Vom Problem zum Programm

- Intuitives Programmieren (aus dem Bauch heraus)
  - Keine Planung
  - Funktionsumfang wird immer wieder erweitert
  - Fehleranfälligkeiten
  - Schlecht zu warten
  - Schlecht zu erweitern
  - *Spaghetticode*

Vom Problem zum Programm

- Problem strukturieren
  - Analyse der Beteiligten
  - Erkennen von Beziehungen
  - Wer kommuniziert mit wem?
- Problem abstrahieren
  - Unbenötigte Teile weglassen
  - Modell vereinfachen
  - Aufgaben zuordnen

Vom Problem zum Programm

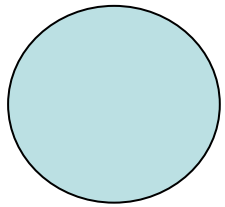
- Akteure ermitteln und benennen
- Attribute definieren
- Akteure ggf. vereinigen oder aufteilen
- Kommunikation vereinfachen
- Nachrichten vereinbaren
- Schnittstellen mit dem Benutzer erfassen
- Sichtbarkeiten (Attribute/Nachrichten) festlegen

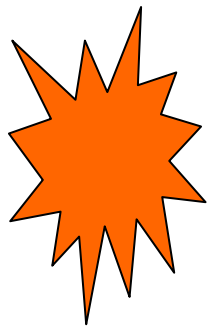
OOP

- Auf Konzept der Objektorientierung basierendes **Programmierparadigma**
- Entwicklung **flexibler** Programme
- **Wiederverwendbarkeit** von Programmen
- Daten und Funktionen möglichst eng in einem so genannten **Objekt** zusammenfassen
- Nach außen hin **kapseln**, so dass Methoden fremder Objekte diese Daten nicht versehentlich manipulieren können
- Der OOP-Ansatz ist für Computerneulinge im allgemeinen leichter zu erlernen

Grafik-Objekte (Akteure/Nachrichten)

- Formen
  - Kreis
  - Rechteck
  - Polygon



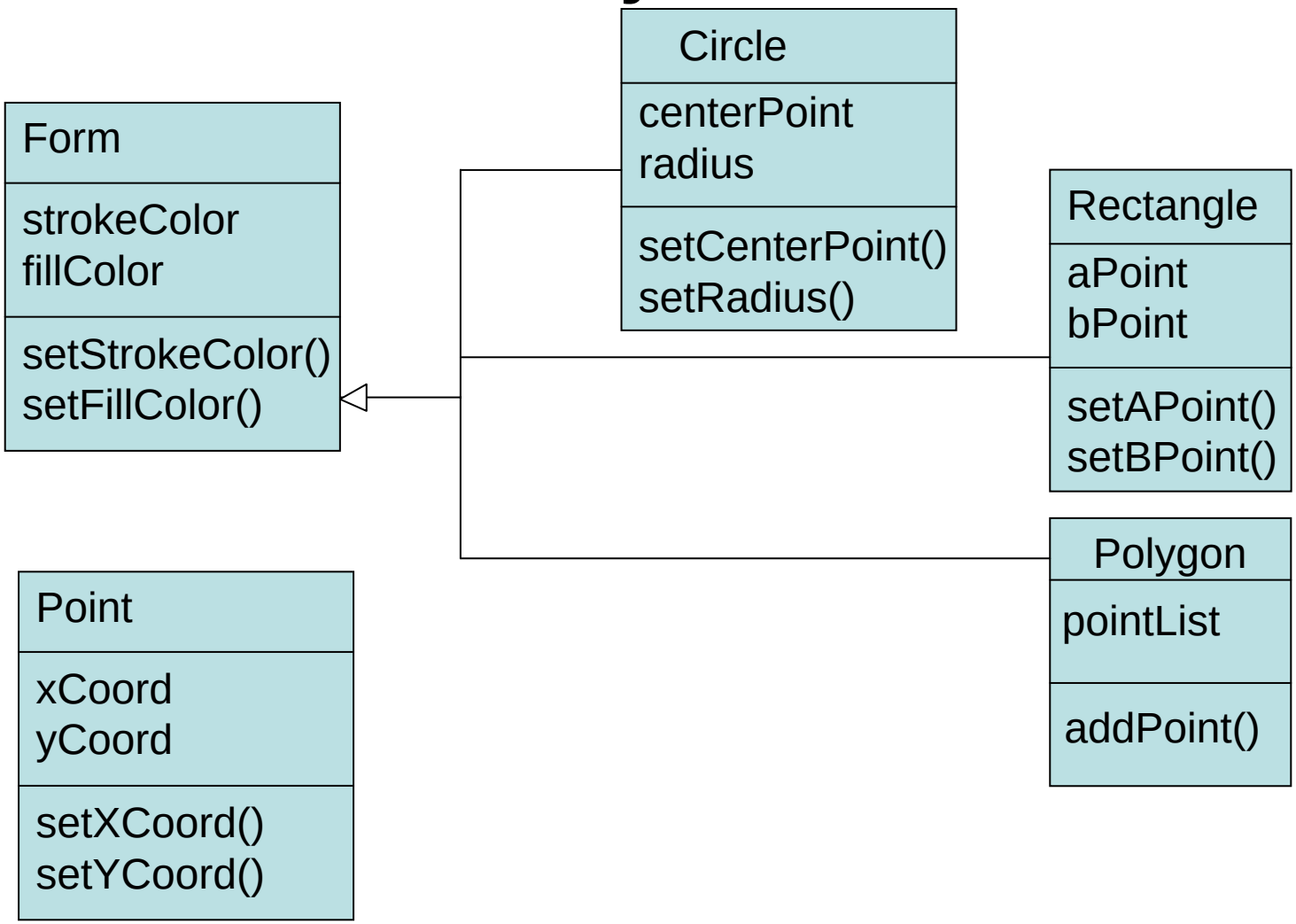


- Nachrichten
  - Linienfarbe bestimmen (setLineColor)
  - Füllfarbe abfragen (getFillColor)
  - ...

Grafik-Objekte (Attribute)

- Rechteck
  - Füllfarbe
  - Linienfarbe
  - Punkt oben links
  - Punkt unten rechts
- Polygon
  - Füllfarbe
  - Linienfarbe
  - Eckpunkte
- Kreis
  - Füllfarbe
  - Linienfarbe
  - Mittelpunkt
  - Radius

Grafik Objekte (Abstraktion)



Unified Modeling Language (UML)

- James Rumbaugh, Grady Booch, Ivar Jacobson
- Oktober 1995 (Rational)
- IBM,HP
- Januar 1997: UML Version 1.0
- 2005: UML2

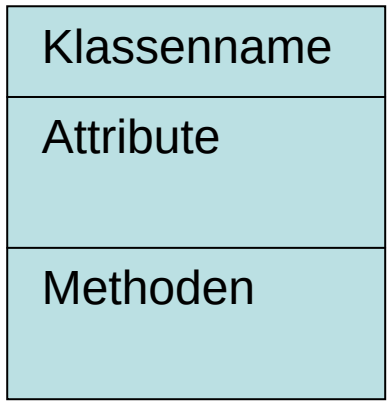
UML

- Notation zur Erstellung/Austausch von Modellen
- Unabhängig von Programmiersprachen
- Verständlichkeit
- Unterstützung von Werkzeugen für OOP
- Unterstützung von Komponenten, Zusammenarbeit, Muster

UML

- Grafische Symbole zur Bezeichnung von
  - Akteuren (Klassen, Attributen und Nachrichten)
  - Vererbungshierarchie
  - Assoziationen
  - Aggregationen
  - Kompositionen
  - Stereotypen

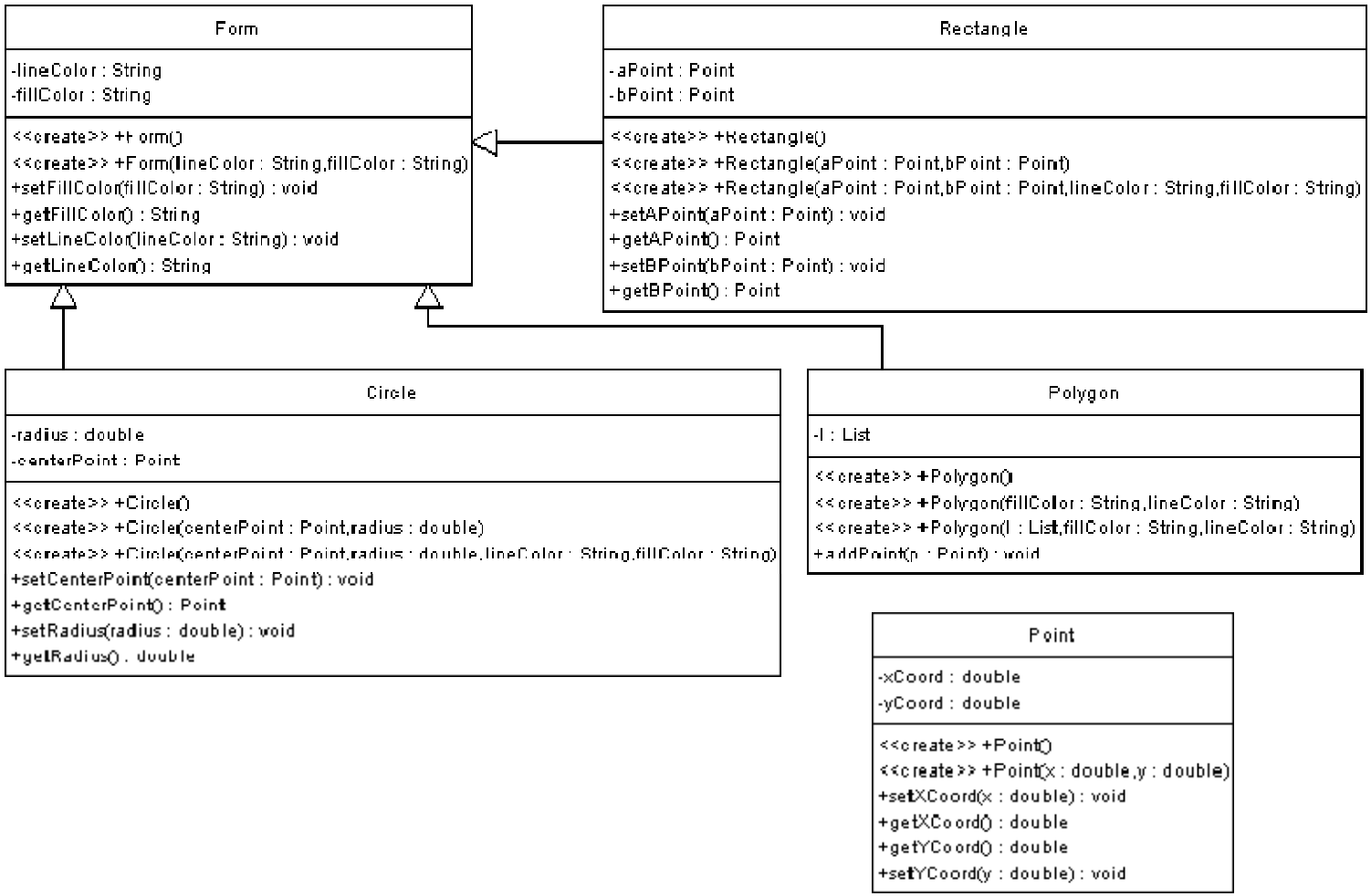
UML Klassendiagramm



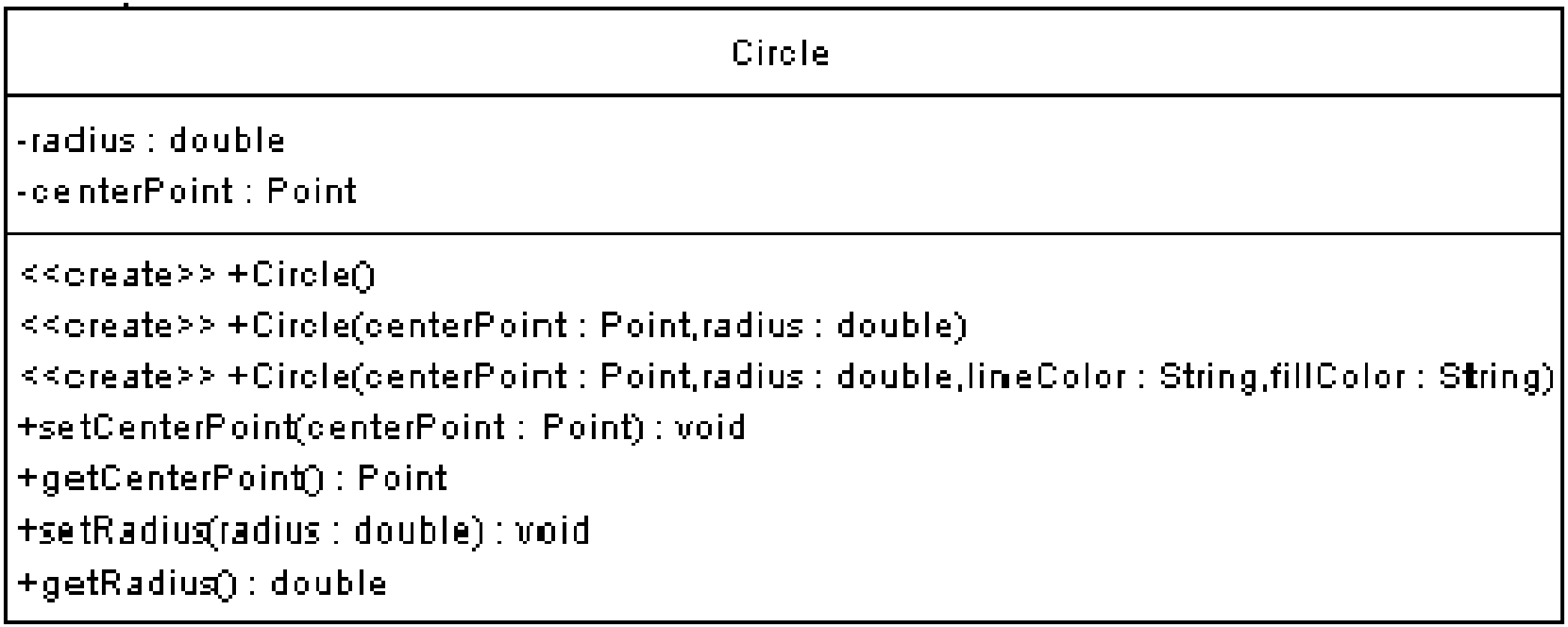
`Attributname: Attributtyp`

`Methodenname(Parametername: Parametertyp):Rückgabetyp`

UML Diagramm (Übersicht)



UML Diagramm (Klasse **`Circle`**)



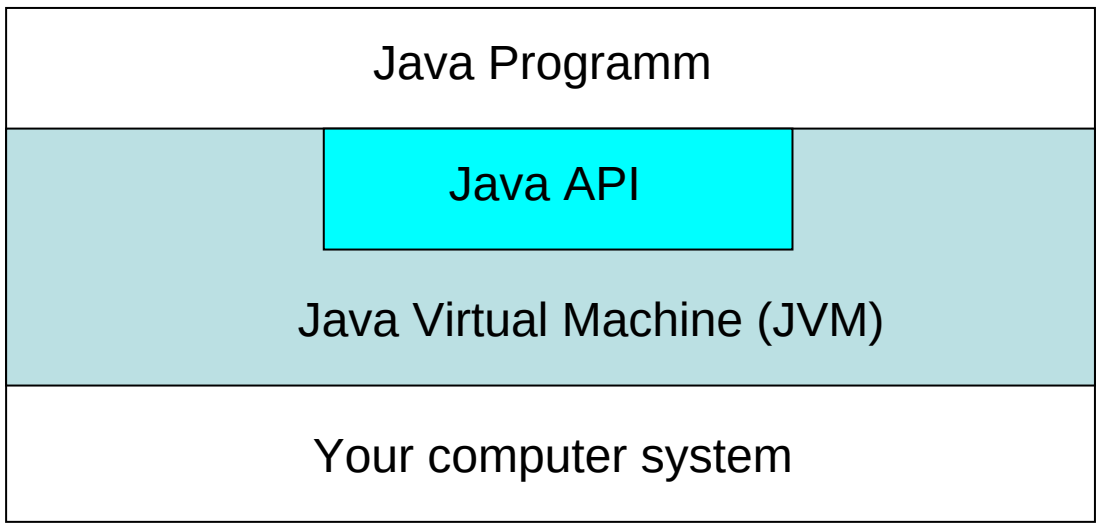
Java ist …

- eine einfache und kleine
- objektorientierte
- dezentrale
- interpretierte
- stabil laufende
- architekturneutrale
- portierbare
- dynamische

Sprache, die Hochgeschwindigkeitsanwendungen und Multithreading unterstützt.

(Definition nach SUN Microsystems)

Java


Java Programm
Java API
Java Virtual Machine (JVM)
Your computer system

Java Entwicklertools

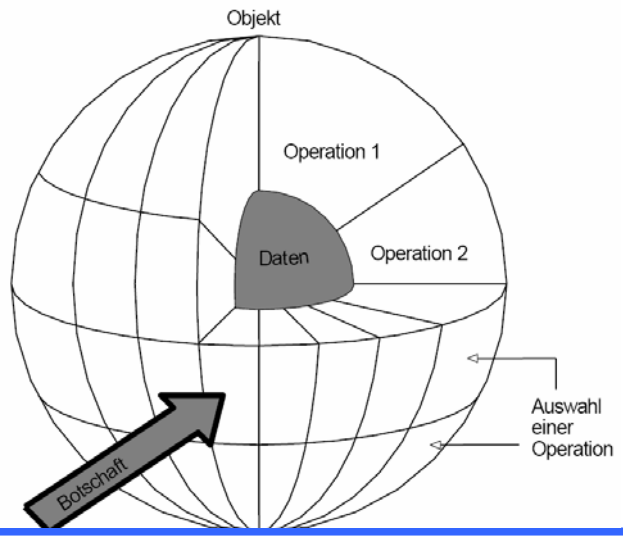
- **`javac`** Java Compiler
  (**`javac Example.java`**)
- **`java`** Java-Interpreter
  (**`java Example`**)
- **`appletviewer`** Applet-Viewer
  (**`appletviewer Example.html`**)
- **`javadoc`** Dokumentationswerkzeug
  (**`javadoc Example.java`**)
- **`jdb`** Textmode-Debugger
  (**`jdb Example`**)
- **`javap`** Disassembler
  (**`javap Example`**)

Klassen und Objekte

- Javaprogramme bestehen aus Klassen
- Allgemeine Beschreibung von Dingen mit gleicher Eigenschaft
- Definition einer gemeinsamen Struktur
- Definition eines gemeinsamen Verhaltens
- Bauplan für konkrete Ausprägungen
- Nachrichten (Methoden) beschreiben das Verhalten

Klassen und Objekte

- Objekt ist eine Instanz/Ausprägung einer Klasse
- Attribute besitzen konkrete Werte und beschreiben den Zustand eines Objektes



Klassen und Objekte

Allgemeiner Aufbau einer Javaklasse:

[Modifikator] **`class`** Klassenname [**`extends`** Oberklasse]
[**`implements`** Interface]

`{`

Attributdeklarationen

Methodendeklarationen

`}`

Die Teile in eckigen Klammern werden zu einem späteren Zeitpunkt erläutert.

Klassen und Objekte

```
class Person {
  String name;
  int alter;
  int schuhgroesse;
}
```

In der Klasse sind keine Operationen vorhanden, sondern lediglich Attribute

Klassen und Objekte

- Um ein Objekt zu erzeugen, muss eine Referenz angelegt werden (Variablendeklaration):
  ```
  Person p;
  ```
- Ein Objekt wird mit dem Konstruktor und new erzeugt:
  ```
  p = new Person();
  ```
- Die Instanzvariablen werden mit Standardwerten belegt
- In der Variablen p wird nur eine Referenz („Speicheradresse“) abgelegt
- Der Standardwert für eine Referenz ist **`null`**

Zugriff auf Attribute

- Mittels der Punktnotation kann auf Attributen zugegriffen werden (sofern sie öffentlich zugänglich sind):

```
p.name = „Ralf Kunze“
p.alter = 33;
p.schuhgroesse = 44;
System.out.println(„Name: „ + p.name);
System.out.println(„Alter: „ + p.alter);
System.out.println(„Schuhgroesse: „ + p.schuhgroesse);
```

Methoden

- Methoden definieren das Verhalten von Objekten
- Sie werden innerhalb einer Klasse definiert
- Methoden haben Zugriff auf alle Daten des aktuellen Objektes
- Methoden sind das Gegenstück zu den Funktionen in imperativen Programmiersprachen (z.B. C)
- Methoden können klassenbezogen sein und sind nicht an ein bestimmtes Objekt gebunden
- Es existieren keine globalen Methoden, die sich weder auf eine Klasse noch ein Objekt beziehen.

Methoden

Syntax:

```
[Modifikatoren]
    Rückgabetyp Methodenname([Parameterliste]) {
        // Anweisungen
}
```

- Der Methodenname und die Typen der Parameterliste zusammengenommen sind die Signatur einer Methode
- Anhand der Signatur wird zur Laufzeit entschieden welche Methode aufgerufen werden soll

Methoden

```
class Person {
  String name;
  int alter;
  int schuhgroesse;

  int jahrgang() {
     return 2007 – alter;
  }
}
```

Methoden

- Mittels der Punktnotation kann auf Methoden zugegriffen werden (sofern sie öffentlich zugänglich sind)
- Zur Unterscheidung zu einem Variablenzugriff dienen die Klammern um die (ev. leere) Parameterliste

```
Person p = new Person();
p.name = „Ralf Kunze“
p.alter = 33;
p.schuhgroesse = 44;
int gebJahr = p.jahrgang();
System.out.println(p.jahrgang());
```

Methoden

- Schlüsselwort **`this`**
  - In der Methode **`jahrgang`** wird deutlich, dass auf das Attribut ohne Punktnotation zugegriffen werden kann
  - Der Kompiler bezieht Zugriffe auf nicht lokale Variablen **`x`** ohne Punktnotation auf das Objekt (**`this`**), was als **`this.x`** interpretiert wird
  - **`this`** ist eine Referenzvariable, die sich immer auf das aktuelle Objekt verweist, um die eigenen Attribute und Methoden anzusprechen.
  - **`this`** ist immer verfügbar
  - **`this`** wird an jede Methode versteckt übergeben, insofern es sich nicht um eine Klassenmethode handelt

Methoden

Die Methoden Jahrgang hätte also auch folgendermaßen implementiert werden können:

```
class Person {
   String name;
   int alter;
   int schuhgroesse;

   int jahrgang() {
       return 2007 – this.alter;
   }
}
```

Methoden

- Innerhalb von Methoden verdecken lokale Variablen und formale Parameter die Attribute gleichen Namens
- Mit Hilfe von **`this`** kann zwischen einer lokalen Variable und einem Attribut gleichen namens unterschieden werden

```
class Person {
   String name;
   …
   void setName(String name) {
       this.name = name;
   }
}
```

Methoden

- Parameter
  - Eine Methode kann mit Parametern definiert werden
  - Parameter werden in Klammern angegeben
  - Jeder **formale Parameter** besteht aus dem Typ und dem Namen des Parameters
  - Sollen mehrere Parameter angegeben werden, sind diese mit Kommata voneinander zu trennen

Methoden

- Parameter (call-by-value)
  - Alle Parameter werden in Java per call-by-value übergeben
  - Bei Aufruf einer Methode wird der aktuelle Wert in die Parametervariable kopiert
  - Der Wert der Parametervariable ist der **aktuelle Parameter**
  - Veränderungen der Parametervariablen wirken sich nur lokal aus

Methoden

- Parameter (call-by-value)

```
class Person {
    String name;
    …
    void printName(int count) {
         while(count > 0){
              count--;
              System.out.println(name);
         }
    }
}
```

Methoden

- Parameter (call-by-value)

```
int a = 3;
   …
p.printName(a);
p.printName(a);
p.printName(a);
```

  - Obwohl der Parameter **`count`** innerhalb der Methode verändert wird, merkt der Aufrufer der Methode nichts davon. Der Wert wurde ja kopiert (call-by-value)
  - Im obigen Beispiel wird der Name neun mal ausgegeben

Methoden

- Parameter (call-by-value mit Seiteneffekt)
  - Nicht primitive Datentypen (Referenzen) werden ebenfalls per call-by-value übergeben
  - Da die Referenz kopiert wird, steht innerhalb der Methode ein Verweis auf das Originalobjekt zur Verfügung
  - Veränderung an diesem Objekt sind auch für den Aufrufer der Methode sichtbar (**Seiteneffekt**)

Methoden

- Parameter (call-by-value mit Seiteneffekt)

```
class Verwaltung  {
    static void changeName(Person p) {
          p.name = „Herr/Frau „ + p.name;
    }
}
```

```
Person p = new Person();
p.name = „Ralf Kunze“;
Verwaltung.changeName(p);
System.out.println(p.name); // Der Name wurde
veraendert
```

Methoden

- Parameter (call-by-value mit Seiteneffekt)
  - Die Methode arbeitet mit dem Originalobjekt
  - Die Übergabe von beliebig großen Objekten ist performant, da nur eine Referenz übergeben wird
  - Sollen Seiteneffekte vermieden werden, müssen die Objekte explizit kopiert werden (z.B. **`clone()`**)

Methoden

- Rückgabewert
  - Beliebiger Typ (primitiv, Referenz oder **`void`**)
  - Der Rückgabetyp **`void`** bedeutet, dass die Methode keinen Wert zurückliefert
  - **`void`**-Methoden sind wegen ihrer Seiteneffekte von Interesse.

Methoden

- Rückgabewert (nicht **`void`**-Methoden)
  - Hat eine Methode einen Rückgabewert, muss dieser mittels **`return`** an den Aufrufenden zurückgegeben werden: **`return wert;`**
  - Nach dem **`return`**-Statement ist die Methode beendet
  - Der Rückgabewert muss mit dem Rückgabetyp kompatibel sein

Überladen von Methoden

- In Java ist es erlaubt Methoden zu überladen, d.h. es dürfen Methoden mit dem gleichen Namen existieren
- Die Methoden werden dann anhand der Anzahl, der Typen und der Reihenfolge der Parameter unterschieden
- Es ist nicht erlaubt Methoden mit dem gleichen Namen und der exakt gleichen Parameterliste (also gleicher Signatur) zu definieren
- Zwei Methoden, die sich nur im Rückgabetyp unterscheiden werden als gleich angesehen und sind nicht erlaubt

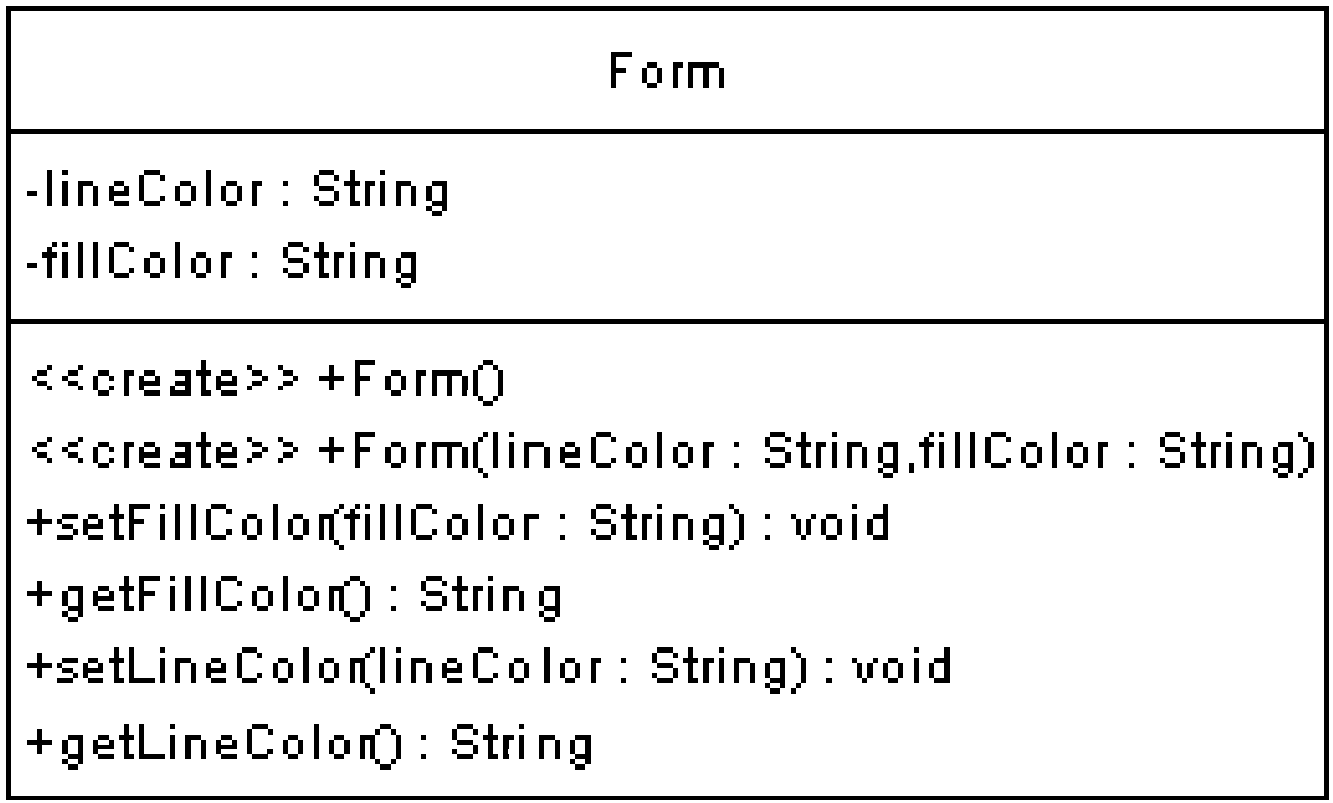
Überladen von Methoden

- Überladen ist dann sinnvoll, wenn die gleichnamigen Operationen auch die gleiche Funktionalität besitzen
- Typische Anwendung ist die Simulation variabler Parameterlisten, die seit Java5 in der Sprachsyntax enthalten sind
- Muss eine Methode um einen Parameter erweitert werden, müssen bereits vorhandene Methodenaufrufe nicht angepasst werden

Überladen von Methoden

```
class Person {
   String name;
        …
   void printName(int count) {
        while(count > 0){
               count--;
               System.out.println(name);
        }
   }
   void printName(String begruessung, int count) {
        while(count > 0){
               count--;
               System.out.println(begruessung + name);
        }
   }
}
```

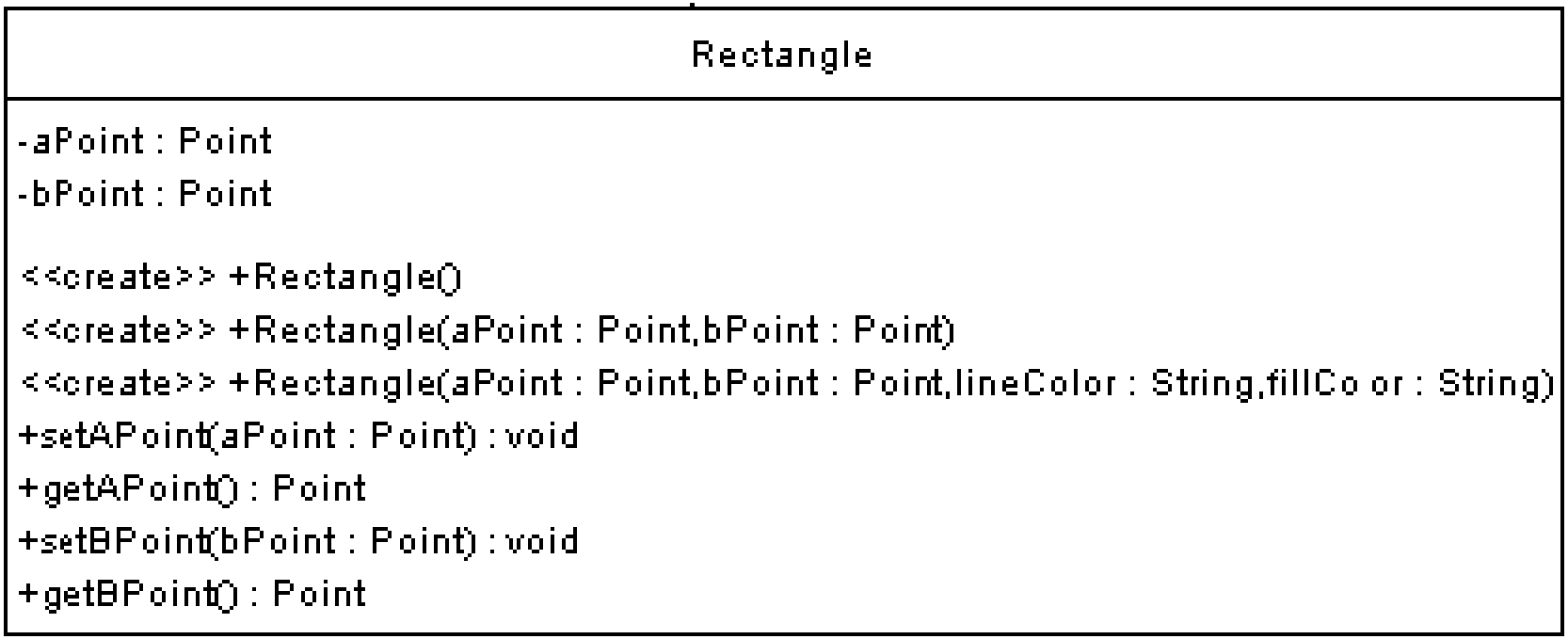
UML2Java


Form
-lineColor : String
-fillColor : String
<<create>> +Form()
<<create>> +Form(lineColor : String,fillColor : String)
+setFillColor(fillColor : String) : void
+getFillColor() : String
+setLineColor(lineColor : String) : void
+getLineColor() : String

UML2Java

```
public class Form {

    private String lineColor;
    private String fillColor;

    public Form() {
          fillColor = "black";
          lineColor ="black";
    }

    public Form(String lineColor, String fillColor) {
          this.lineColor = lineColor;
          this.fillColor = fillColor;
    }

    public void setFillColor(String fillColor) { this.fillColor = fillColor; }

    public String getFillColor() { return fillColor; }

    public void setLineColor(String lineColor) { this.lineColor = lineColor; }

    public String getLineColor() { return lineColor; }
}
```

UML2Java


Rectangle
-aPoint : Point
-bPoint : Point
<<create>> +Rectangle()
<<create>> +Rectangle(aPoint : Point,bPoint : Point)
<<create>> +Rectangle(aPoint : Point,bPoint : Point,lineColor : String,fillCo or : String)
+setAPoint(aPoint : Point) : void
+getAPoint() : Point
+setBPoint(bPoint : Point) : void
+getBPoint() : Point

UML2Java

```
import util.Point;

public class Rectangle extends Form {
    private Point aPoint;
    private Point bPoint;

    public Rectangle() {
          this(new Point(), new Point(), "black", "black");
    }

    public Rectangle(Point aPoint, Point bPoint) {
          this(aPoint, bPoint, "black", "black");
    }

    public Rectangle(Point aPoint, Point bPoint, String lineColor, String fillColor) {
          super(lineColor, fillColor);
          this.aPoint=aPoint; this.bPoint=bPoint;
    }

    public void setAPoint(Point aPoint) { this.aPoint = aPoint; }

    public Point getAPoint() { return aPoint; }

    public void setBPoint(Point bPoint) { this.bPoint = bPoint; }

    public Point getBPoint() { return bPoint; }
}
```

Nächste Woche

- (statische) Konstruktoren
- Destruktoren
- Exceptions
- Vererbung
- Abstrakte Klassen
- Interfaces