Kapitel ~~6~~ 7

Vererbungsprinzip

Ziele

- Das Vererbungsprinzip der objektorientierten Programmierung verstehen
- Und in Java umsetzen können
- Insbesondere folgende Begriffe verstehen und anwenden können:
  - Ober/Unterklassen
  - Subtyping
  - Überschreiben von Methoden
  - Dynamische Bindung
- Die Klasse `Object` kennenlernen
- Abstrakte Klassen und Interfaces kennenlernen

Beispiel: Klasse BankKonto

```
public class BankKonto {
  protected double kontoStand;

  public BankKonto(double anfangsBetrag) {
    this.kontoStand = anfangsBetrag;
  }
  public double getKontoStand() {
    return this.kontoStand;
  }
  public void einzahlen(double x) {
    this.kontoStand = this.kontoStand + x;
  }
  public void abheben(double x) {
    this.kontoStand = this.kontoStand - x;
  }
}
```

Grafische Darstellung in UML
(Unified Modeling Language)

"private" ≙ "–"

BankKonto
#double kontoStand Attribute
+BankKonto(double anfangsBetrag) +double getKontoStand() +void einzahlen(double x) +void abheben(double x)

Konstruktoren- und Methodenköpfe

Damit man in einer Unterklasse das Attribut `kontoStand` verwenden kann, wird die Sichtbarkeit des Attributs `kontoStand` auf `protected` gesetzt.

Vererbung

Vererbung ist ein Mechanismus zur Implementierung von Klassen durch Erweiterung existierender Klassen.

Beispiel:

- Die Klasse `BankKonto` stellt die Grundfunktionalität eines Bankkontos bereit.
- Möchte man verschiedene spezielle Arten von Bankkonten implementieren, so kann man die Klasse `BankKonto` durch **Vererbung** erweitern.
  - `SparKonto`: hat zusätzlich einen Zinsatz, Zinsen können gut geschrieben werden
  - `GiroKonto`: bei jeder Transaktion werden Gebühren berechnet

Sparkonto in Java

```
public class SparKonto extends BankKonto {
  private double zinsSatz;

  public SparKonto(double anfangsBetrag,  double zinsSatz) {
    super(anfangsBetrag);
    this.zinsSatz = zinsSatz;
  }

  public void zinsenAnrechnen() {
    double zinsen = this.kontoStand * this.zinsSatz / 100.0;
    this.kontoStand = this.kontoStand+zinsen;
  }
}
```

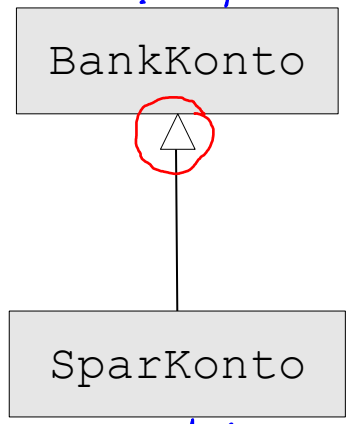
← zusätzlich

← ruft den Konstruktor der Oberklasse auf

← zusätzliche Methode

Darstellung in UML

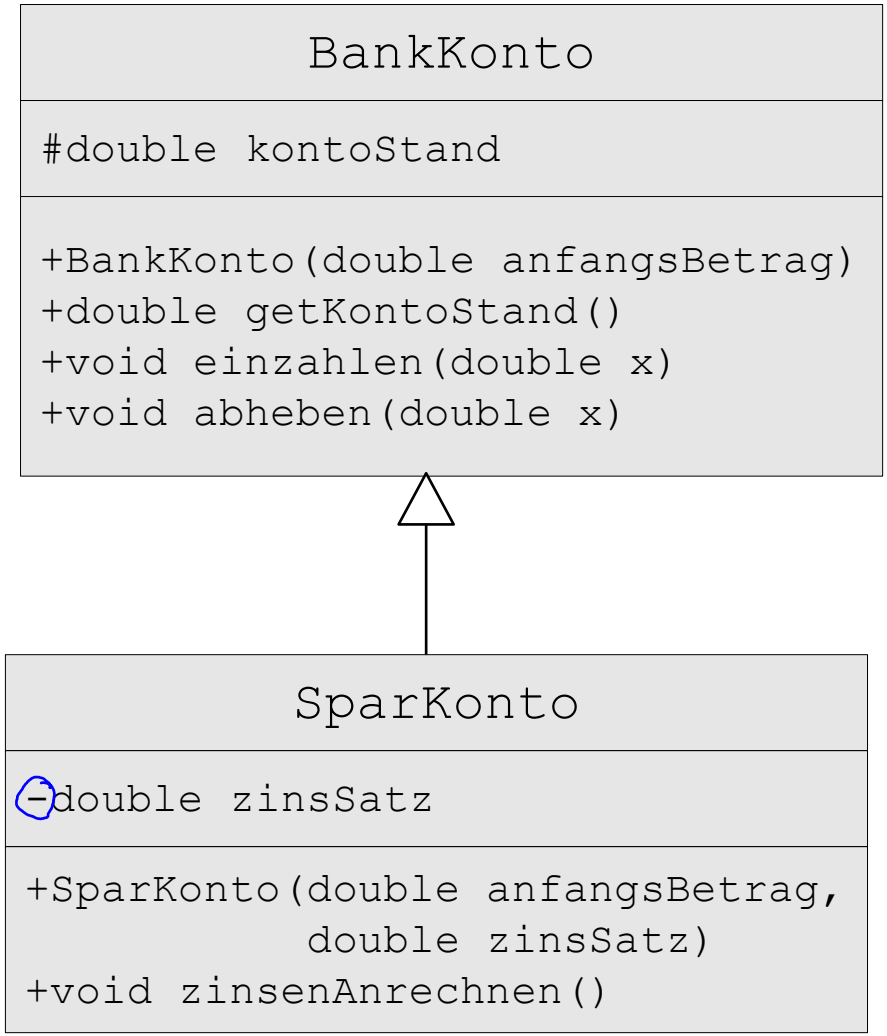
Oberklasse / Superklasse



Unterklasse / Subklasse

- Ein Sparkonto erbt alle Methoden und Attribute eines Bankkontos und fügt eigene hinzu.

Grafische Darstellung der Vererbung in UML



private

Vererbung und Konstruktoren

- **Konstruktoren werden nicht vererbt**
  (im Gegensatz zu Attributen und Methoden).
- Die Konstruktoren der Oberklasse können jedoch in neu definierten Konstruktoren der Unterklasse aufgerufen werden.
- Aufruf des Konstruktors der Oberklasse:

```
super();              // parameterloser Konstruktor
```

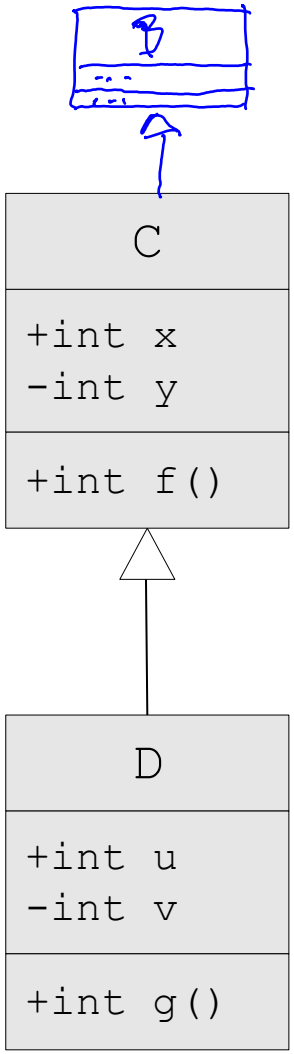
  bzw.

```
super(a1,…,an);       // Konstruktor mit n Parametern
```

- Dieser Aufruf muss die erste Anweisung des Unterklassen-Konstruktors sein.

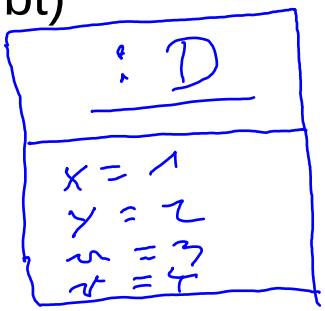
Beispiel: Benutzung von SparKonto

```
public class SparKontoTest {

  public static void main(String[] args) {
    SparKonto sk = new SparKonto(500.0, 3.0);  // 500 EUR, 3% Zinsen

    sk.einzahlen(50.0);                        // geerbte Methode
    sk.zinsenAnrechnen();                      // neue Methode
    System.out.println(sk.getKontoStand());    // geerbte Methode
  }
}
```



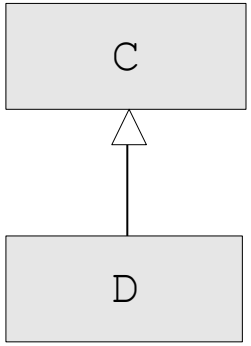
Vererbung von Attributen und Methoden

- Attribute von `C`: `x`, `y`
- Attribute von `D`: `u`, `v`, `x`, `y` (`x`, `y` sind geerbt)
- Methoden von `C`: `f`
- Methoden von `D`: `g`, `f` (`f` ist geerbt)



- Wenn `C` Subklasse einer anderen Klasse `B` ist, dann besitzen `C` und `D` auch noch alle Attribute und Methoden von `B`.
- Man kann von `D` aus **nicht direkt auf die privaten Attribute** von `C` zugreifen (hier `y`), sondern nur mittels nicht privater (geerbter) getter-Methoden von `C`.
- Auf geschützte (protected) Attribute kann man in Subklassen schon zugreifen.

Subtyping



(1) Vererbung drückt eine „is-a“ Beziehung aus.

(2) Die Objekte der Klasse `D` können wie Objekte der Klasse `C` benutzt werden.

(3) Eine Referenz auf ein Objekt der Unterklasse `D` kann überall dort eingesetzt werden, wo eine Referenz auf ein Objekt der Oberklasse `C` erwartet wird.

Wir sagen: Der Klassentyp D ist ein **Subtyp** von C! (D „passt“ zu C.)

Beispiel:

```
C c = new D();
```

BankKonto b = new SparKonto(500.0, 3.0);

Der Variablen `c` mit Typ `C` wird ein Ausdruck des Subtyps `D` zugewiesen. Die Variable `c` verweist dann auf ein Objekt der Klasse `D` (und damit auf ein Objekt von `C`).

Umgekehrt kann man einer Variablen vom Typ `D` **keine** Referenz auf ein Objekt der Oberklasse `C` zuweisen.

`D d = c;` ist unzulässig.

~~SparKonto sk = new BankKonto(500.0);~~

Beispiel: BankKonto und SparKonto

```
public class SparKontoTest {

  public static void main(String[] args) {
    SparKonto sk = new SparKonto(500.0, 3.0);
    BankKonto k = sk;      ✓ wegen Subtyping

    k.einzahlen(50);
    sk.zinsenAnrechnen();
//  k.zinsenAnrechnen();          wäre nicht richtig:
//                                Die Variable k hat den Typ BankKonto
//                                und diese Klasse hat keine Methode
//                                zinsenAnrechnen().
//  SparKonto sk1 = k;            wäre auch falsch, da ein Konto nicht
//                                notwendigerweise ein Sparkonto ist.
    System.out.println(k.getKontoStand());      550.0 + 16.5 = 566.5
    System.out.println(sk.getKontoStand());     566.5
  }
}
```

Typkonversion durch Einengung

- Im letzten Beispiel wäre `k.zinsenAnrechnen();` nicht möglich gewesen, obwohl wir wissen, dass `k` auf ein Objekt der Klasse `SparKonto` verweist.
- In solchen Situationen kann man den Typ ausdrücklich konvertieren:

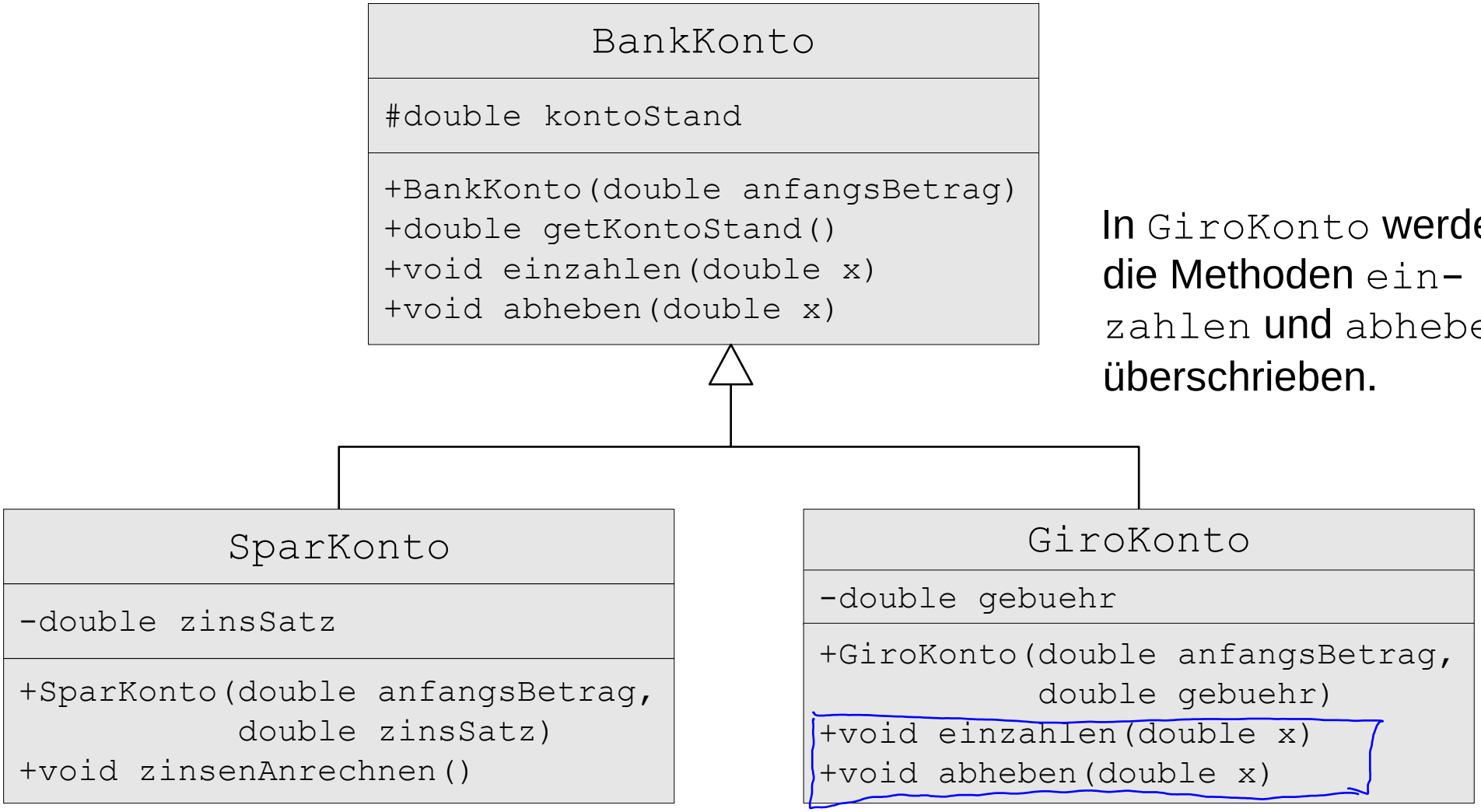
```
SparKonto sk1 = (SparKonto)k;
sk1.zinsenAnrechnen();
```

- Bei der Auswertung einer Typkonversion `(D)e` wird **geprüft**, ob das Objekt, auf das `e` im aktuellen Zustand verweist, tatsächlich Instanz von `D` oder einer Subklasse von `D` ist.
  Ist dies nicht der Fall, so kommt es zu einem Laufzeitfehler (`ClassCastException`).
- **Hinweis**: Typkonversionen zwischen Klassentypen sollten möglichst vermieden werden.

Überschreiben von Methoden (Overriding)

- In vielen Fällen möchte man bei der Vererbung die Implementierung von Methoden der Oberklasse etwas anpassen.
  Beispiel: In einem Girokonto soll bei jeder Ein- und Auszahlung eine Gebühr vom Kontostand abgezogen werden.
- Deshalb ist es möglich, in der Unterklasse Methoden der Oberklasse neu zu implementieren, d.h. zu **überschreiben**.
- Eine überschriebene Methode muss in der Unterklasse denselben Namen und dieselben Parameter (in Anzahl und Typ) wie die Methode der Oberklasse haben. Die Sichtbarkeit darf nicht eingeschränkt werden. Der Ergebnistyp (falls vorhanden) muss zum bisherigen Ergebnistyp passen.
- Überschreiben in UML: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.

Überschreiben von Methoden: Beispiel (1)



In `GiroKonto` werden die Methoden `einzahlen` und `abheben` überschrieben.

Überschreiben von Methoden: Beispiel (2)

```
public class GiroKonto extends BankKonto {
  private double gebuehr;

  public GiroKonto(double anfangsBetrag, double gebuehr) {
    super(anfangsBetrag);
    this.gebuehr = gebuehr;
  }
  public void abheben(double x) {
    this.kontoStand = this.kontoStand - x - this.gebuehr;
  }
  public void einzahlen(double x) {
    this.kontoStand = this.kontoStand + x - this.gebuehr;
  }
}
```

Überschreiben von Methoden: Beispiel (3)

```
public class BankKonto {
  ...
  public void ueberweisenAuf(BankKonto ziel, double x) {
    ziel.einzahlen(x);
    this.abheben(x);
  }
}
public class KontoTest {
  public static void main(String[] args) {
    BankKonto sk = new SparKonto(0.0, 3.0);
    BankKonto gk = new GiroKonto(0.0, 0.02); // 2 Cent Transaktionsgebühr
    sk.einzahlen(100.0);  // sk.getKontoStand() == 100.0
    gk.einzahlen(100.0);  // gk.getKontoStand() == 99.98
    gk.ueberweisenAuf(sk, 50.0);
        // sk.getKontoStand() == 150.0, gk.getKontoStand() == 49.96
  }
}
```

Bei gk.einzahlen(100.0) bewirkt die dynamische Bindung, dass der Rumpf der Methode einzahlen der Klasse GiroKonto ausgeführt wird, da der Wert von gk zum aktuellen Zeitpunkt auf ein GiroKonto-Objekt verweist.

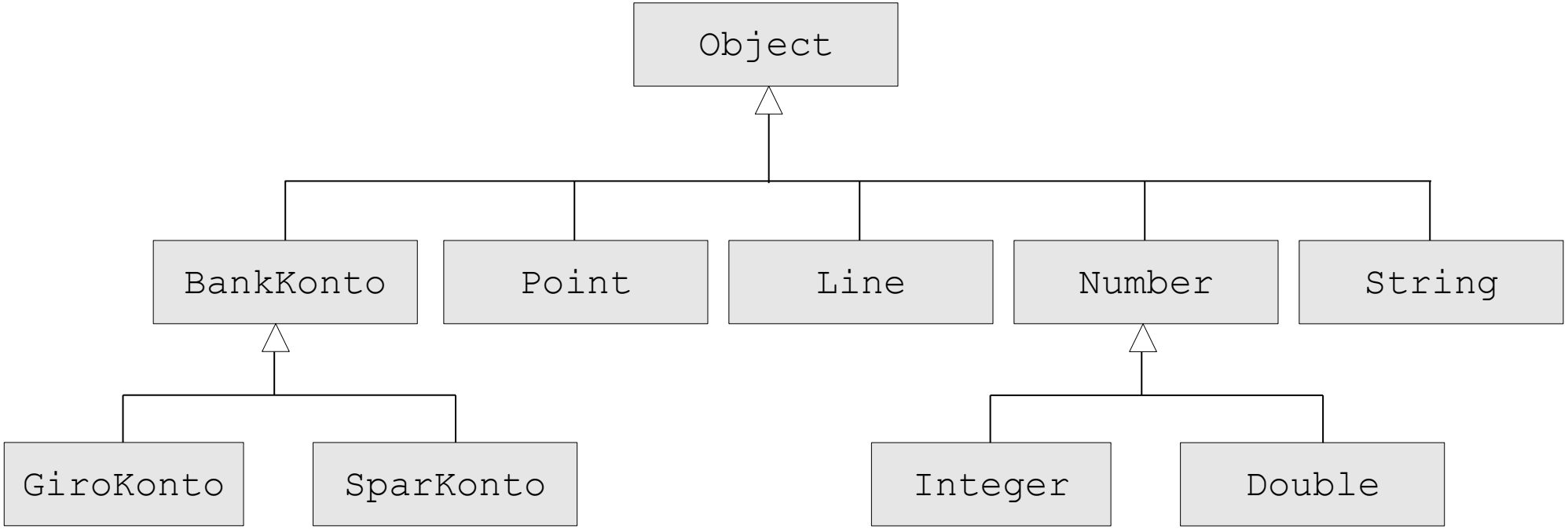
Dynamische Bindung

- Bei einem Methodenaufruf `e.m(a1,…,an)` muss eine Implementierung der Methode `m` ausgewählt und ausgeführt werden.
- Die Auswahl der Implementierung richtet sich nach der Klasse `C` des Objekts, auf welches `e` im aktuellen Zustand zeigt und **nicht** nach dem Klassentyp des Ausdrucks `e`.
- Ist die Methode `m` in der Klasse `C` definiert, so wird diese Implementierung ausgeführt. Ansonsten wird nach der kleinsten Oberklasse von `C` gesucht, in der die Methode definiert ist und diese Implementierung wird dann ausgeführt.

Die Entscheidung, welche Implementierung der Methode benutzt wird, wird dynamisch (d.h. **erst zur Laufzeit**) getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

Die Klasse `Object` (1)

In Java sind alle Klassen (implizit) von der Klasse `Object` abgeleitet.



Die Klasse `Object` (2)

Object

`+String toString()`
`+boolean equals(Object o)`
`...`

der Verweisen auf ein

- Die Methode `toString` liefert die Textrepräsentation ~~eines~~ Objekts.
- Viele Klassen überschreiben die Methoden `toString` und `equals`, zum Beispiel `Integer`, `Double`, `Boolean`, `String`.
- Die Methode `equals` wird meist so überschrieben, dass sie nicht die Gleichheit von Objektreferenzen sondern die Gleichheit von Objektzuständen testet.

Abstrakte Klassen und abstrakte Methoden

- Häufig möchte man abstrakte Begriffe beschreiben, deren Realisierungen nur in speziellen Ausprägungen sinnvoll sind.
  Beispiel: Die Konten, die von einer Bank verwaltet werden, treten nur in einer bestimmten Form (Sparkonto, Girokonto, etc.) auf. „Bankkonto" ist also eigentlich ein abstrakter Begriff.
- **Abstrakte Klassen** sind Klassen, für die keine Objekte direkt erzeugt werden können, die also nicht direkt „instanziierbar" sind.
- Instanzen können dann für konkrete Subklassen einer abstrakten Klasse erzeugt werden.
- **Abstrakte Methoden** sind Methoden, die in einer (abstrakten) Klasse keine Implementierung haben.
- Eine Klasse, die eine abstrakte Methode hat, muss abstrakt sein (jedoch nicht umgekehrt).
- Implementierungen von abstrakten Methoden werden in Unterklassen angegeben.

Beispiel: Abstrakte Klasse BankKonto

```
public abstract class BankKonto {
   protected double kontoStand;

   public double getKontoStand() {
       return kontoStand;
   }
  public abstract void einzahlen(double x);
  public abstract void abheben(double x);
}
```

Darstellung in UML:
Kursive Schreibweise für „abstract"

BankKonto {abstract}
#double kontoStand
+double getKontoStand() *+void einzahlen(double x)*{abstract} *+void abheben(double x)*{abstract}

- Die konkrete Implementierung der Methoden `einzahlen` und `abheben` erfolgt in Unterklassen.

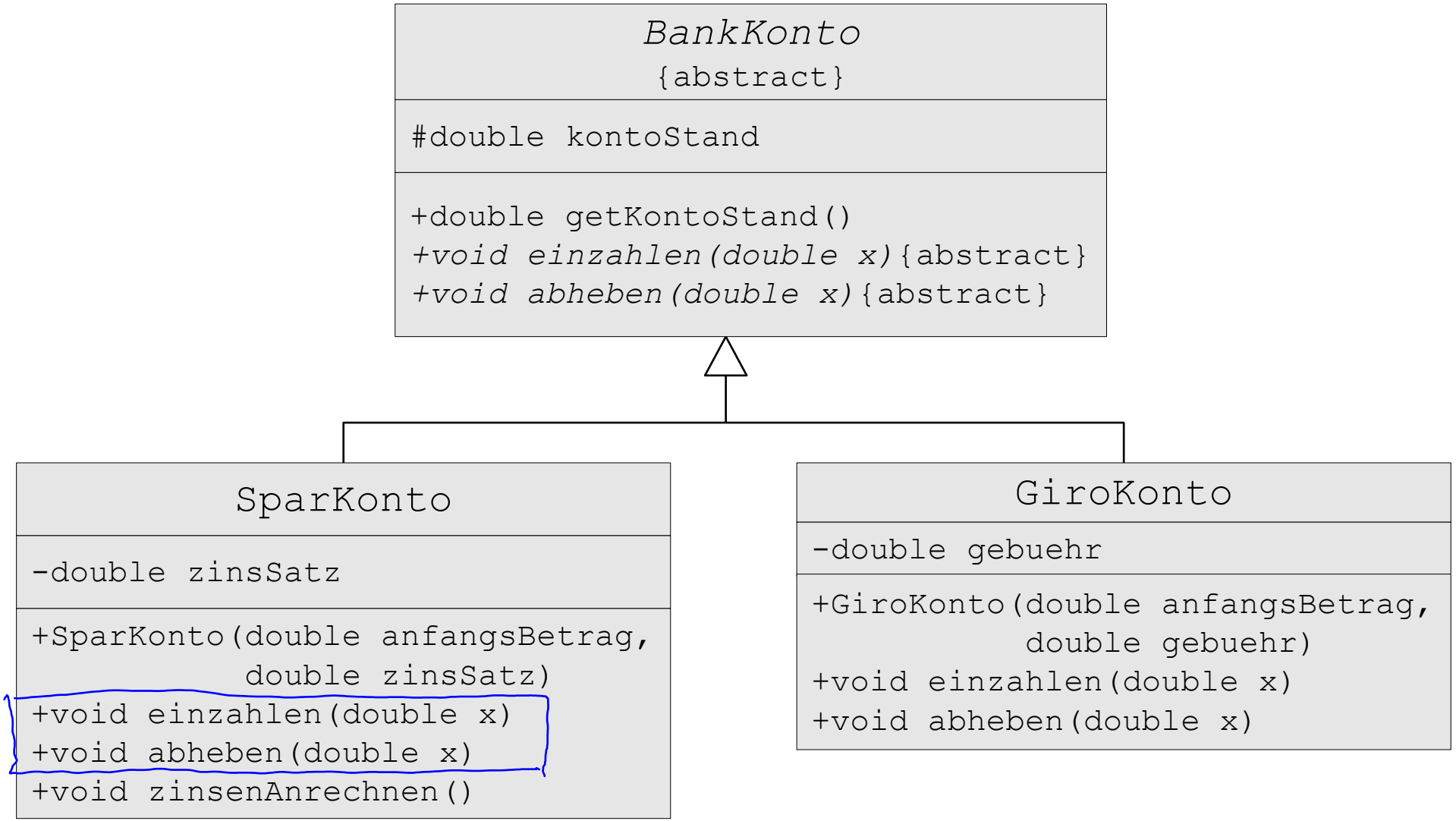
Implementierung von Methoden in Unterklassen: `SparKonto`

```
public class SparKonto extends BankKonto {
  private double zinsSatz;

  public SparKonto(double anfangsBetrag, double zinsSatz) {
    this.kontoStand = anfangsBetrag;
    this.zinsSatz = zinsSatz;
  }
  public void einzahlen(double x) {
    this.kontoStand = this.kontoStand + x;
  }
  public void abheben(double x) {
    this.kontoStand = this.kontoStand - x;
  }
  public void zinsenAnrechnen() {
    double zinsen = this.kontoStand * this.zinsSatz / 100;
    this.einzahlen(zinsen);
  }
}
```

Implementierung von Methoden in Unterklassen: `GiroKonto`

```
public class GiroKonto extends BankKonto {
  private double gebuehr;

  public GiroKonto(double anfangsBetrag, double gebuehr) {
    this.kontoStand = anfangsBetrag;
    this.gebuehr = gebuehr;
  }
  public void einzahlen(double x) {
    this.kontoStand = this.kontoStand + x - this.gebuehr;
  }
  public void abheben(double x) {
    this.kontoStand = this.kontoStand - x - this.gebuehr;
  }
}
```

Abstrakte und konkrete Klassen



Interfaces und deren Implementierung

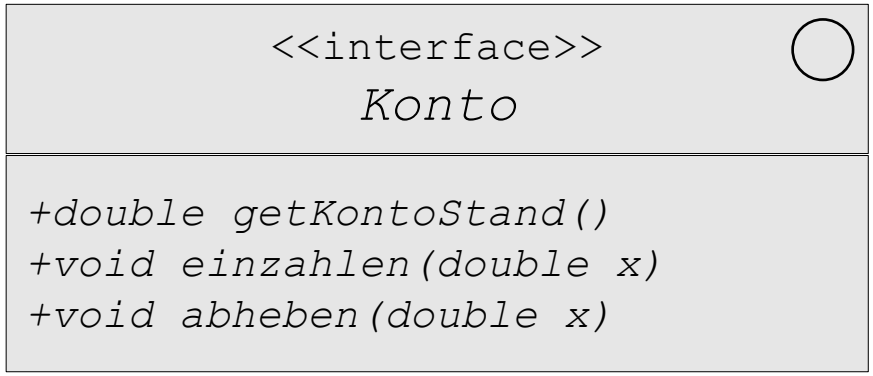
- Interfaces (Schnittstellen) sind ein Hilfsmittel zur Aufteilung von Systemen in einzelne Komponenten (bzw. zur Entkopplung von Klassen).
- Interfaces bieten Dienste (in Form von Methoden) an, die von bestimmten Klassen implementiert und von anderen Klassen benutzt werden können.
- Interfaces haben **nur öffentliche, abstrakte Methoden** und **keine** Objektattribute.
- Mit jeder Interface-Deklaration wird ein **Interfacetyp** definiert, der wie ein Klassentyp verwendet werden kann.

Beispiel: Interface für Konten

```
public interface Konto {
  public double getKontoStand();
  public void einzahlen(double x);
  public void abheben(double x);
}
```

alles abstrakte Methoden, da Interface.

Darstellung in UML:
alles kursiv, Kreis als Icon, nur abstrakte Methoden

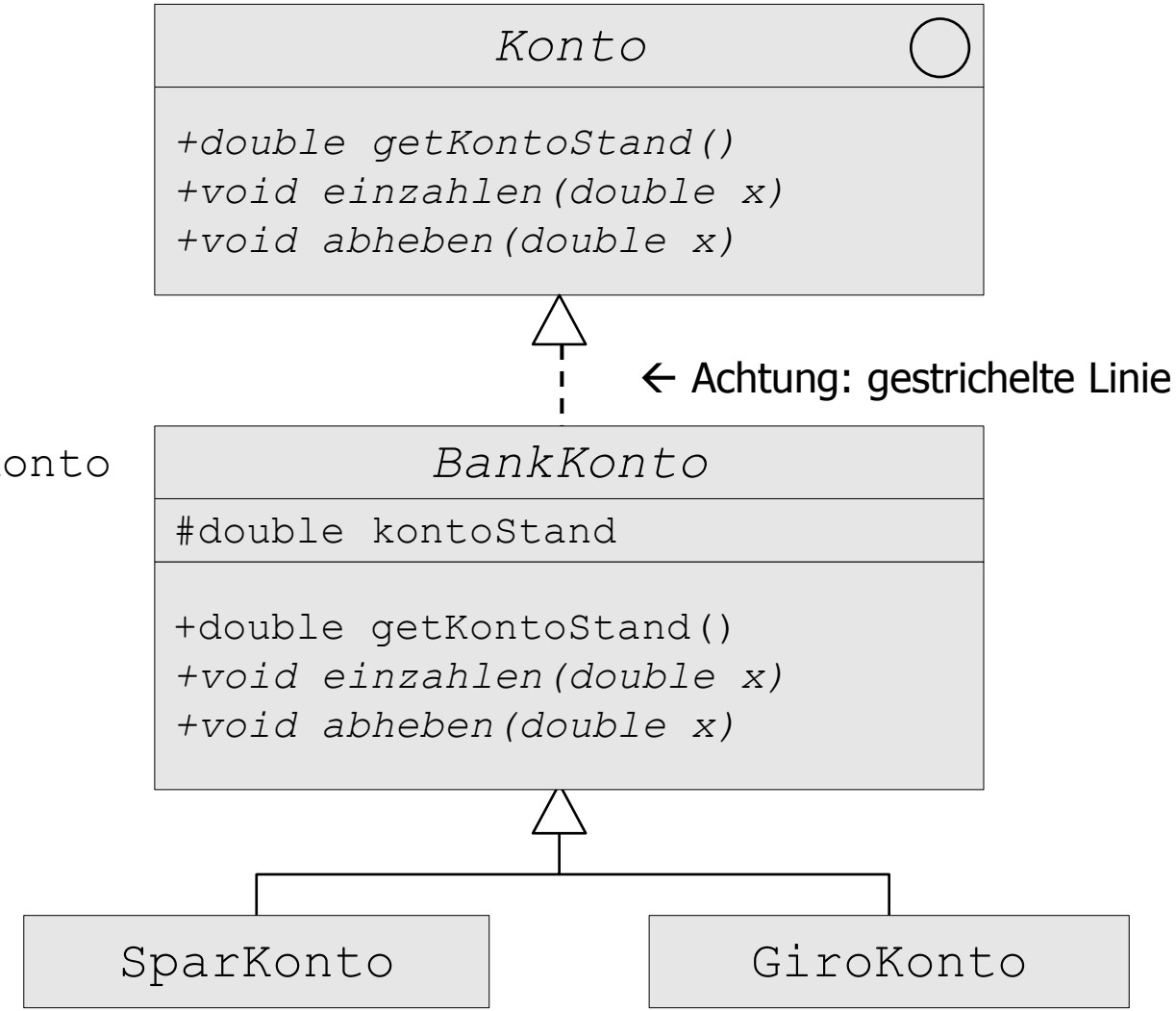


Implementierung von Interfaces

```
public interface Konto {
    …
}
```

```
public abstract class BankKonto
    implements Konto {
    …
}
```

```
public class SparKonto
    extends BankKonto {
    …
}
```



Erweiterte Grammatik für Klassendeklarationen in Java

ClassDeclaration =
["public"] ["**abstract**"] "class" *Identifier* [***Super***] [***Interfaces***] *ClassBody*

Super = "**extends**" *ClassType*

Interfaces = "**implements**" *InterfaceType* {"," *InterfaceType*}
InterfaceType = *Identifier*

ClassBody (vgl. Kapitel 5, aber jetzt mit abstrakten Methoden).

Beachte:

- In Java kann eine Klasse <u>höchstens eine</u> Oberklasse erweitern, eine Klasse kann aber <u>mehrere</u> Interfaces implementieren.
- `D implements C` induziert (genauso wie `D extends C`) eine Subtypbeziehung: `D` ist <u>Subtyp</u> von `C`.