

# Einführung in die Programmierung für NF

Vererbung

---

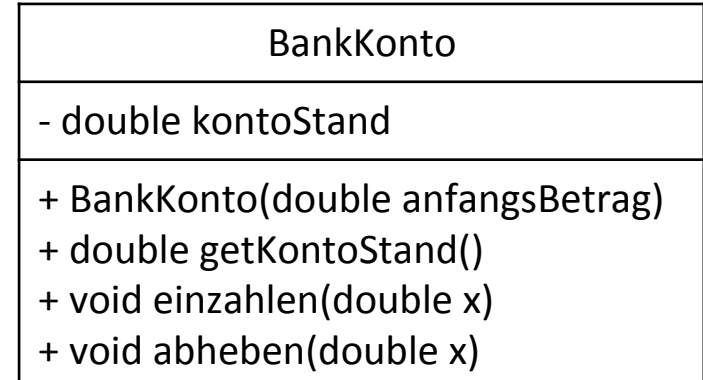
# Ziele

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

Grafische Darstellung in UML  
(Unified Modeling Language)



# 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 Zinssatz, Zinsen können gut geschrieben werden
  - GiroKonto: bei jeder Transaktion werden Gebühren berechnet

# Extends

- Die Vererbungshierarchie bei der Klassendefinition mit dem Schlüsselwort **extends** gekennzeichnet.

```
Class <<Namesub>> extends  
<<Namesuper>> { ... }
```

- Die **Subklasse** „erweitert“ (engl: extends) die **Superklasse**

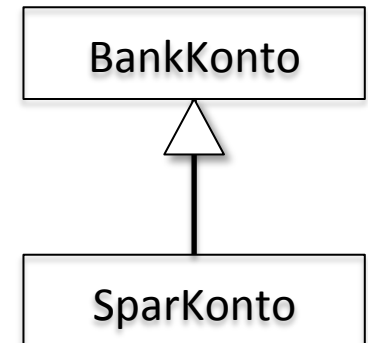
# SparKonto in Java

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

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

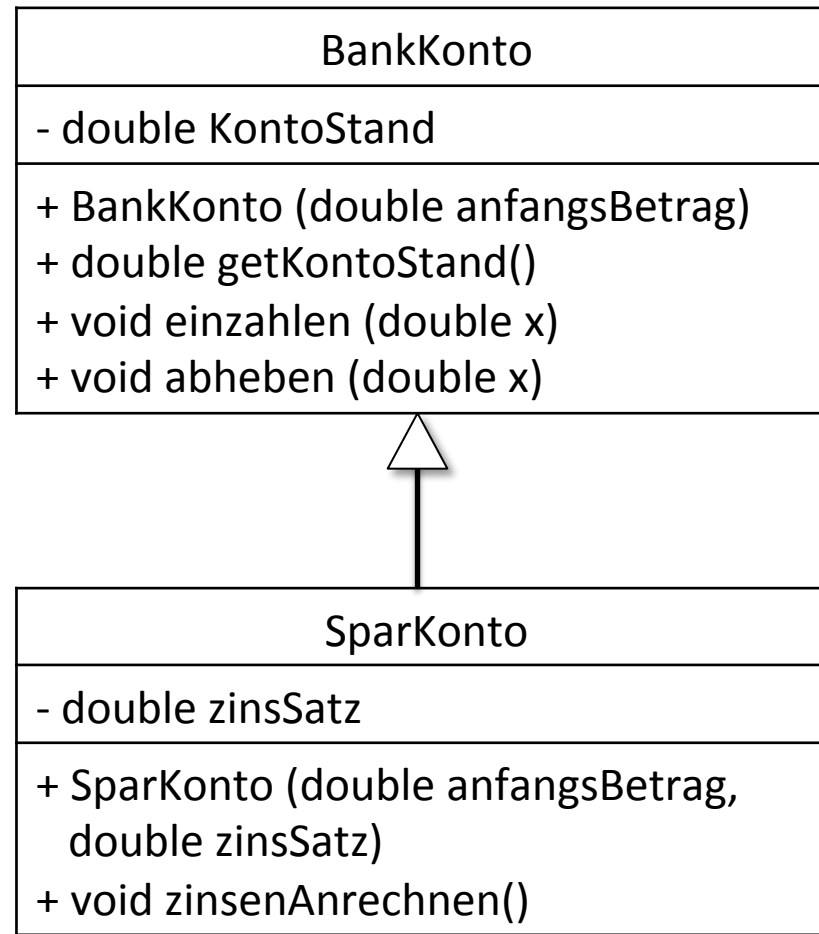
    public void zinsenAnrechnen() {
        double zinsen = this.getKontoStand() * zinsSatz/ 100;
        this.einzahlen(zinsen);
    }
}
```

Darstellung in UML



- SparKonto erbt alle Methoden von BankKonto.
- Die Attribute von BankKonto werden auch vererbt, sind aber in SparKonto nicht sichtbar (also nicht direkt zugreifbar) da in BankKonto privat deklariert.

# Grafische Darstellung der Vererbung in UML



# 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.

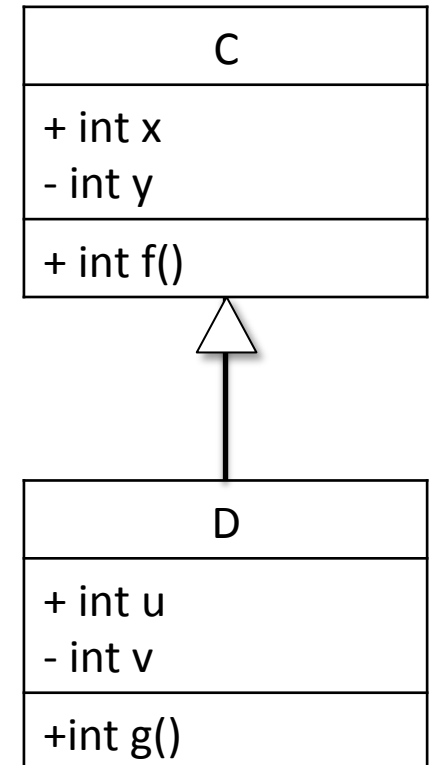


# Beispiel: Benutzung von SparKonto

```
public class SparKontoTest{  
  
    public static void main (String[] args) {  
        SparKonto sk = new SparKonto(500, 3); // 500 EUR, 3% Zinsen  
  
        sk.einzahlen (50); // geerbte Methode  
        sk.zinsenAnrechnen(); // neue Methode  
        System.out.println(sk.getKontoStand()); // geerbte Methode  
    }  
}
```

# Vererbung von Attributen und Methoden

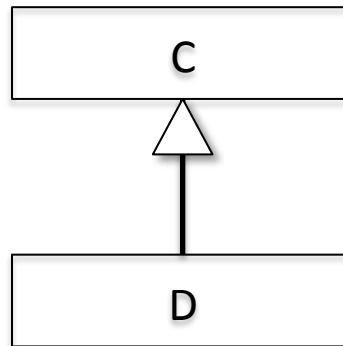
- Attribute von C: x, y
- Attribute von D: u, v, x, y (x, y sind geerbt)
- Methoden von C: f
- Methoden von D: g, f (f ist geerbt)
- Wenn C Subklasse einer anderen Klasse B ist, dann besitzen C und D auch noch alle Attribute und Methoden von B.
- Man kann von D aus **nicht direkt auf die privaten Attribute** von C zugreifen (hier y), sondern nur mittels nicht privater (geerbter) Methoden von C.



# SUBTYPING

# Subtyping

**Mit Vererbung drückt man „is-a“- Beziehungen zwischen Klassen aus, z.B. ist jedes Sparkonto ein spezielles Bankkonto.**



Die Klasse D besitzt nicht nur alle Methoden und Attribute von C.

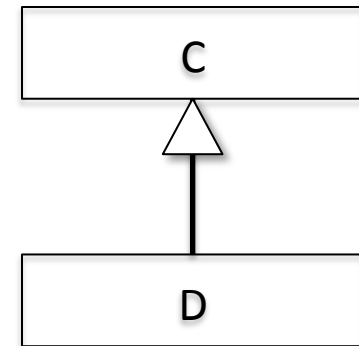
Es gilt stärker:

**„Jedes Objekt der Klasse D ist auch ein Objekt der Klasse C.“**

# Subtyping

- (1) Die Objekte der Klasse D können wie Objekte der Klasse C benutzt werden.
- (2) 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!



Beispiel:

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

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.

# Beispiel: BankKonto und SparKonto

```
public class SparKontoTest{

    public static void main(String[] args) {
        SparKonto sk = new SparKonto(500, 3);
        BankKonto k = sk;

        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());
        System.out.println(sk.getKontoStand());
    }
}
```

# 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 `(C)e` wird **geprüft** ob das Objekt, auf das `e` im aktuellen Zustand verweist, tatsächlich Instanz einer von `C` abgeleiteten Klasse ist.  
Ist dies nicht der Fall -> Laufzeitfehler (`ClassCastException`).
- **Hinweis:** Typkonversionen zwischen Klassentypen sollten möglichst vermieden werden.

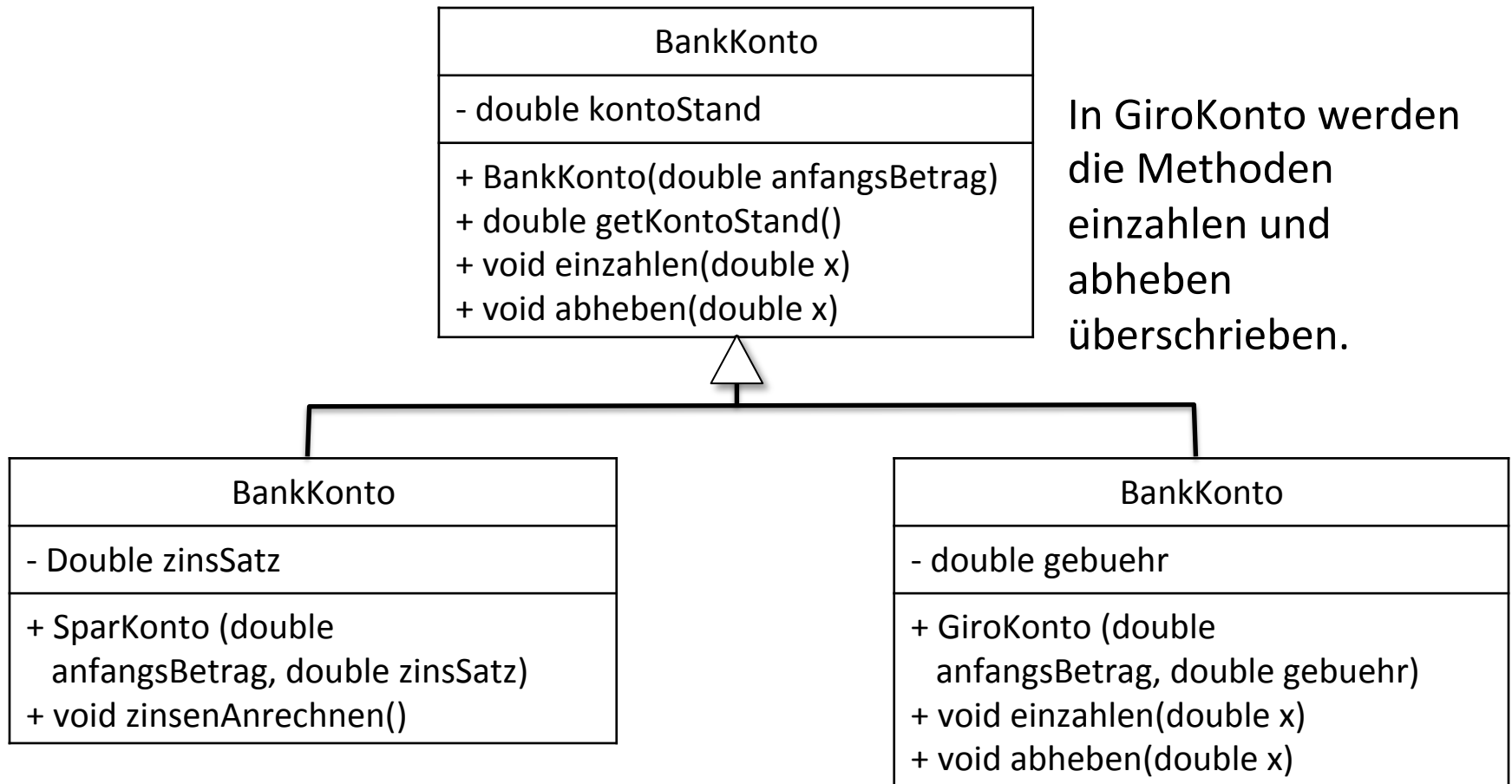
# Überschreiben von Methoden (Overriding)

- 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.
- Möglichkeit in der Unterklasse Methoden der Oberklasse neu zu implementieren, d.h. zu **überschreiben**.
- Überschriebene Methode muss in Unterklasse denselben Namen und dieselben Parameter (in Anzahl und Typ) wie Methode der Oberklasse haben  
Sichtbarkeit darf nicht eingeschränkt werden  
Ergebnistyp (falls vorhanden) muss gleich oder höchstens ein Subtyp des bisherigen Ergebnistyps sein.
- Überschreiben in UML: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.



# OVERRIDING

# Überschreiben von Methoden: Beispiel



# Überschreiben von Methoden: Beispiel

```
public class GiroKonto extends BankKonto{

    private double gebuehr;

    public GiroKonto(double anfangsBetrag, double gebuehr) {
        super(anfangsBetrag);
        this.gebuehr = gebuehr;
    }

    public void abheben(double x) {
        super.abheben(x); // Zugriff auf Methode der Oberklasse:
                          // Ohne super würde sich die Methode selbst aufrufen.
        super.abheben(gebuehr);
    }

    public void einzahlen(double x) {
        super.einzahlen(x);
        super.abheben(gebuehr);
    }
}
```

# Überschreiben von Methoden: Beispiel

```
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, 3);
        BankKonto gk = new GiroKonto(0, 0.02); // 2 Cent Transaktionsgebühr

        sk.einzahlen(100); // sk.getKontoStand() == 100
        gk.einzahlen(100); // gk.getKontoStand() == 99.98

        gk.ueberweisenAuf(sk, 50);
        // gk.getKontoStand() == 49.96, sk.getKontoStand() == 150
    }
}
```

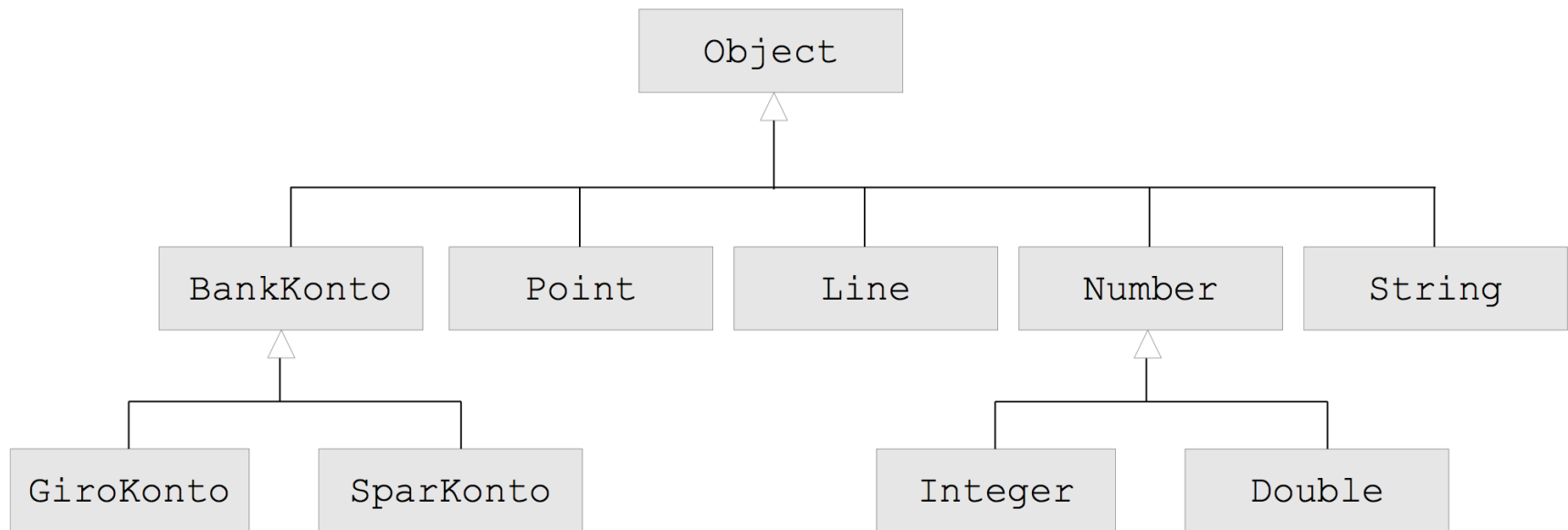
# DYNAMISCHE BINDUNG

# Dynamische Bindung

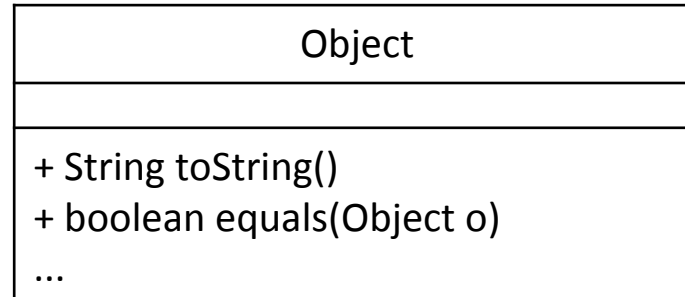
- Methoden haben in Java dynamische Bindungen. D.h. Methoden hängen nicht von der Klasse der Variablen ab, sondern von der Klasse des Objekts, das die Variable im Augenblick enthält.
- Die Entscheidung, welche Implementierung der Methode benutzt wird, wird dynamisch (d.h. **erst zur Laufzeit**) getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

# Die Klasse Object

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



# Die Klasse Object



- 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.



# instanceof

- Um zu überprüfen, welche Unterklasse sich in einer Variablen verbirgt verwendet man den Begriff **instanceof**.
- Beispiel:  
Wir haben jetzt Objekte und Variablen folgender Art

```
BankKonto konto;  
GiroKonto giro = new GiroKonto();  
SparKonto spar = new SparKonto();
```

Es ist erlaubt, sowohl `giro` als auch `spar` an die Variable `konto` zuzuweisen.

Die Klasse der Variable kann anschließend mit `instanceof` überprüft werden.

```
if (konto instanceof GiroKonto){...}  
else if (konto instanceof SparKonto){...}
```

# ABSTRAKTE UND KONKRETE KLASSEN

# 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 „instanzierbar“ 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.

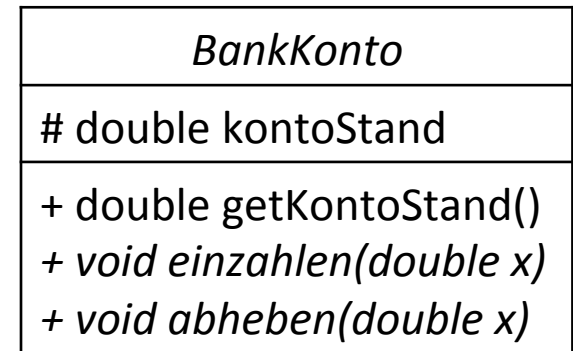
# Beispiel: Abstrakte Klasse BankKonto

```
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“



- Die konkrete Implementierung der Methoden `einzahlen` und `abheben` erfolgt in Unterklassen.
- Damit man in einer Unterklasse den Kontostand ändern kann, wird die Sichtbarkeit des Attributs `kontoStand` auf `protected` gesetzt. Damit ist `kontoStand` auch in Unterklassen von `BankKonto` verwendbar.

# 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 * zinsSatz / 100;
        einzahlen(zinsen);
    }
}
```

# Implementierung von Methoden in Unterklassen: GiroKonto

```
public class GiroKonto extends BankKonto{

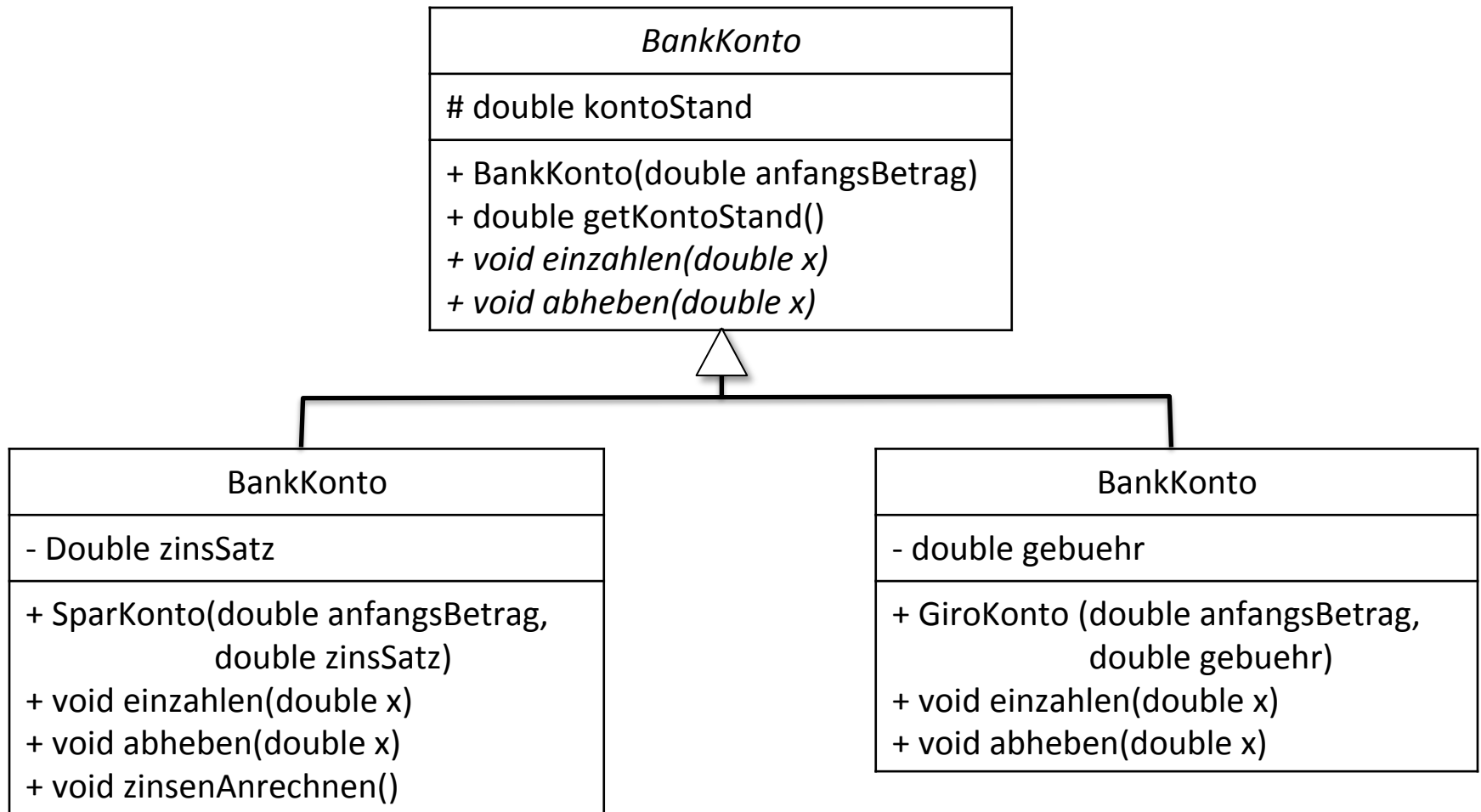
    private double gebuehr;

    public GiroKonto(double anfangsBetrag, double gebuehr) {
        this.kontoStand = anfangsBetrag;
        this.gebuehr = gebuehr;
    }

    public void einzahlen(double x) {
        this.kontoStand = this.kontoStand + x - gebuehr;
    }

    public void abheben(double x) {
        this.kontoStand = this.kontoStand - x - gebuehr;
    }
}
```

# Abstrakte und konkrete Klassen



# INTERFACES



# 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);  
}  
  
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*

## Beachte:

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

Vielen Dank für Ihre Aufmerksamkeit