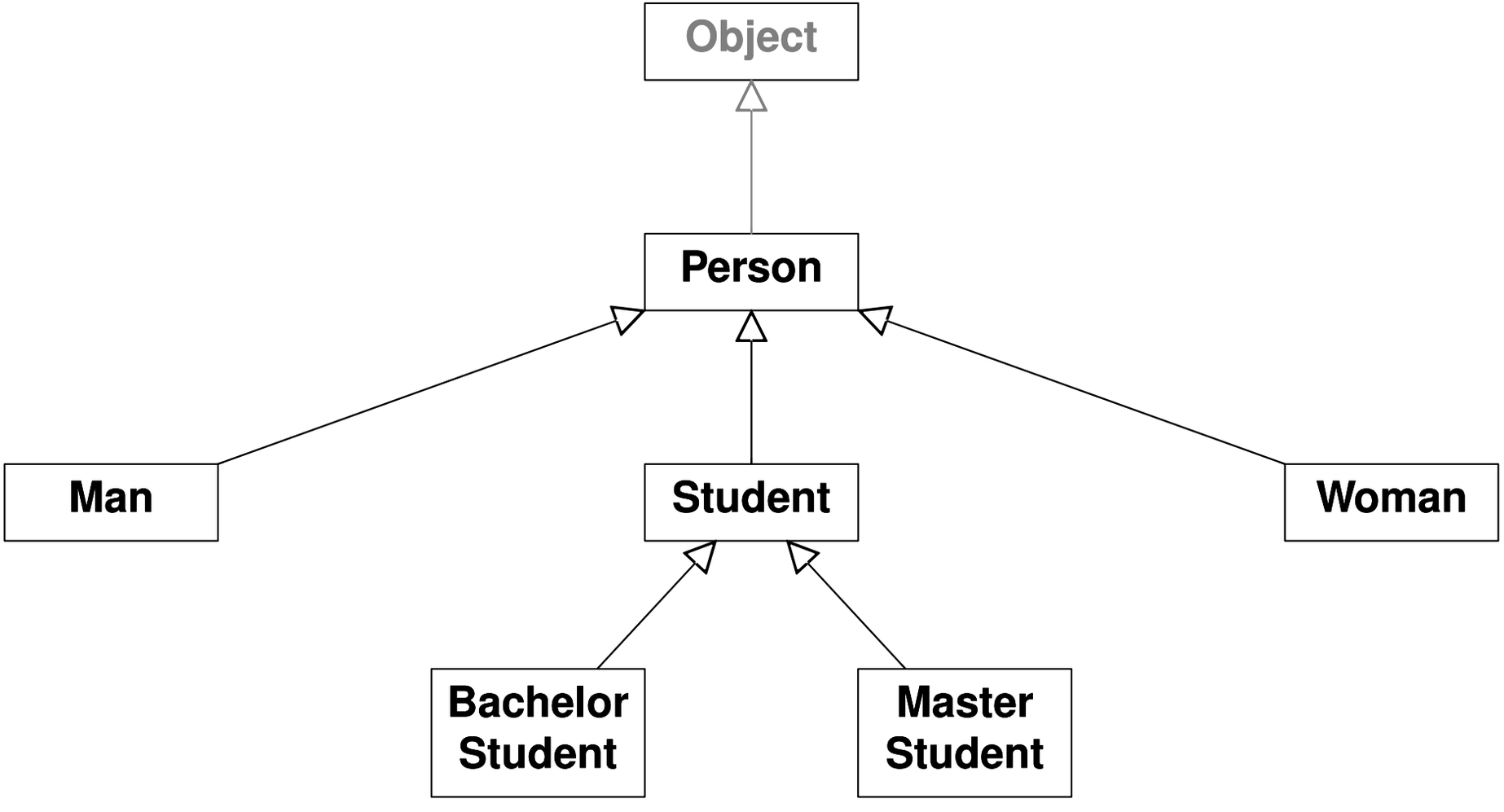
5 Unterklassen

5.1 Begriffe

- Ein Klasse kann eine andere Klasse *erweitern* (Schlüsselwort `extends`), d. h. ihre Felder und Methoden (sowie ihre geschachtelten Klassen und Schnittstellen, nicht jedoch ihre Konstruktoren und Initialisierungsblöcke) *erben*.
- Die erbende Klasse heißt (direkte) *Unterklasse*, die beerbte Klasse (direkte) *Oberklasse*.
- Eine Klasse kann beliebig viele direkte Unterklassen, aber nur eine direkte Oberklasse besitzen (*einfache Vererbung*).
- Eine Unterklasse einer Unterklasse wird auch als *indirekte Unterklasse*, eine Oberklasse einer Oberklasse als *indirekte Oberklasse* bezeichnet.
- Gelegentlich wird eine Klasse auch als *triviale Unter-* und *Oberklasse* ihrer selbst bezeichnet.
- Formal besteht die Menge aller Unter- bzw. Oberklassen einer Klasse daher aus:
  - der Klasse selbst
  - der Menge aller Unter- bzw. Oberklassen ihrer direkten Unter- bzw. Oberklasse(n).

Beispiel



- Wenn eine Klasse (wie z. B. `Person`) keine explizite Oberklasse besitzt, besitzt sie implizit die vordefinierte Wurzelklasse `Object` als Oberklasse (vgl. § 5.10).

5.2 Zugriffskontrolle

- Eine Unterklasse kann auf folgende Elemente ihrer (direkten und indirekten) Oberklasse(n) zugreifen (vgl. § 4.3):
  - öffentliche Elemente (`public`)
  - unterklassenöffentliche Elemente (`protected`)
    (mit gewissen Einschränkungen, siehe § 5.13)
  - paketöffentliche Elemente (keine Angabe), wenn sich die Unterklasse im selben Paket wie die Oberklasse befindet (vgl. Kap. 9)
- Auf private Elemente (`private`) einer Oberklasse kann grundsätzlich nicht zugegriffen werden.

5.3 Untertyp-Polymorphie

- Da ein Objekt einer Unterklasse alle Merkmale seiner Oberklasse besitzt, kann es überall verwendet werden, wo ein Objekt der Oberklasse erwartet wird (*Ersetzbarkeit*). Insbesondere kann ein Objekt der Unterklasse an eine Variable der Oberklasse zugewiesen werden (aber nicht umgekehrt).

- Somit können Variablen eines bestimmten Klassentyps zur Laufzeit nicht nur Objekte dieser Klasse, sondern auch Objekte von (direkten und indirekten) Unterklassen referenzieren (*Untertyp-Polymorphie*).

- Daher muss zwischen dem *statischen Typ* einer Variablen (oder eines Ausdrucks) und ihrem *dynamischen Typ* unterschieden werden:

  - Der statische Typ ist der im Programmcode deklarierte Typ einer Variablen (bzw. der aus dem Programmcode ableitbare Typ eines Ausdrucks).
    Er ist dem Übersetzer bekannt und während der gesamten Programmausführung konstant.

  - Der dynamische Typ einer Variablen (oder eines Ausdrucks) ist der Typ des tatsächlich referenzierten Objekts.
    Er ist dem Übersetzer nicht bekannt und kann sich im Laufe der Programmausführung ändern.
    Es handelt sich jedoch immer um einen Untertyp des statischen Typs.

Beispiel

```
class Person { ..... }
class Student extends Person { ..... }

class Test {
  public static void main (String [] args) {
    // Der statische und dynamische Typ von p ist Person.
    Person p = new Person();

    // Der statische und dynamische Typ von s ist Student.
    Student s = new Student();

    // Das von s referenzierte Student-Objekt
    // kann an die Person-Variable p zugewiesen werden.
    // Der statische Typ von p bleibt Person,
    // der dynamische Typ wird Student.
    p = s;

    // Die umgekehrte Zuweisung ist nicht möglich.
    s = p;       // Fehler!!
  }
}
```

5.4 Konstruktoren

- Konstruktoren werden grundsätzlich nicht vererbt, d. h. jede Klasse definiert ihre eigenen Konstruktoren.

- Ein Konstruktor kann als erste Anweisung einen Aufruf eines Konstruktors der Oberklasse enthalten, um die Objektvariablen der Oberklasse zu initialisieren. Als Name dieses Konstruktors wird nicht der Name der Oberklasse, sondern das Schlüsselwort `super` verwendet.

- Wenn die erste Anweisung eines Konstruktors weder ein Aufruf eines anderen Konstruktors der eigenen Klasse (vgl. § 4.7) noch ein Aufruf eines Konstruktors der Oberklasse ist, wird automatisch ein Aufruf `super()` des parameterlosen Konstruktors der Oberklasse eingefügt.

- Dies impliziert, dass die Oberklasse in diesem Fall einen (entsprechend zugreifbaren) parameterlosen Konstruktor besitzen muss.
  (Wenn eine Klasse keine explizite Oberklasse besitzt, besitzt sie implizit die Wurzelklasse `Object` als Oberklasse, die einen solchen Konstruktor besitzt; vgl. § 5.10.)

- Der explizite oder implizite Aufruf des Oberklassenkonstruktors findet zwischen den in § 4.9 beschriebenen Schritten 3 und 4 statt (d. h. nach der Auswertung der eigenen Konstruktorargumente) und führt (nach der Auswertung seiner Argumente) für die Oberklasse die dort genannten Schritte 4 und 5 aus, d. h. die Initialisierungen der Oberklasse finden vor den eigenen Initialisierungen statt.

- Wenn die Oberklasse selbst wieder eine Oberklasse besitzt (d. h. wenn sie verschieden von der Wurzelklasse `Object` ist), wird zuvor der passende `super`-Aufruf ihres Konstruktors ausgeführt, usw.
  Somit werden die Initialisierungsschritte 4 und 5 für alle Oberklassen der Klasse „von oben nach unten“ ausgeführt.

5.5 Überschreiben und dynamisches Binden von Objektmethoden

- Eine Unterklasse kann von einer Oberklasse geerbte Objektmethoden *überschreiben*, d. h. neu implementieren.
  Eine solche Überschreibung kann optional mit der Annotation `@Override` gekennzeichnet werden. Wenn es dann keine passende geerbte Methode gibt, wird der Code vom Übersetzer nicht akzeptiert, während ohne diese Annotation eine *Überladung* anstelle einer Überschreibung definiert wird, was häufig nicht beabsichtigt ist.

- Beim Aufruf einer Objektmethode wird (sofern der Aufruf zur Übersetzungszeit anhand der statischen Typen korrekt ist) *zur Laufzeit* anhand des *dynamischen Typs* `T` des Aufrufobjekts (d. h. unabhängig von seinem statischen Typ) diejenige Implementierung der Methode ausgewählt, die sich entweder in der Klasse `T` oder in der „nächstgelegenen“ Oberklasse von `T` befindet (*dynamisches Binden*),

- Für das aktuelle Objekt `this` kann eine überschriebene Methode `objmeth` (nur) mittels `super.objmeth(arguments)` aufgerufen werden.
  (Das heißt, die Verwendung von `super` setzt das dynamische Binden außer Kraft und ruft direkt die Methode der Oberklasse auf.)

- Ein Aufruf der Art `((superclass)object).objmeth(arguments)` ist aufgrund des normalen dynamischen Bindens äquivalent zu `object.objmeth(arguments)`, da sich der *dynamische* Typ von `object` durch die Umwandlung des *statischen* Typs nach `superclass` nicht ändert.

Beispiel

```
class Person {
  private String name;
  public Person (String n) { name = n; }
  public void print () { System.out.println(name); }
}

class Student extends Person {
  private int matr;

  public Student (String n, int m) {
    super(n);
    matr = m;
  }

  @Override
  public void print () {
    super.print();
    System.out.println(matr);
  }
}
```

```
class Test {
  public static void main (String [] args) {
    // Der statische Typ von p und s ist jeweils Person.
    // Der dynamische Typ von p ist Person, der von s ist Student.
    Person p = new Person("Person");
    Person s = new Student("Student", 12345);

                          // Aufruf der Methode print der Klasse ...
    p.print();            // Person
    s.print();            // Student
  }
}
```

Weitere Regeln

- Die Anzahl und die Typen der Parameter der überschreibenden Methode müssen mit denen der überschriebenen Methode übereinstimmen.
- Der Resultattyp der überschreibenden Methode muss mit dem Resultattyp der überschriebenen Methode übereinstimmen oder ein Untertyp davon sein.
- Die überschreibende Methode muss mindestens so viel Zugriff erlauben wie die überschriebene Methode, das heißt:
  - Eine öffentliche Methode kann nur von einer öffentlichen Methode überschrieben werden.
  - Eine unterklassenöffentliche Methode kann nur von einer öffentlichen oder unterklassenöffentlichen Methode überschrieben werden.
  - Eine paketöffentliche Methode kann nur von einer öffentlichen, unterklassenöffentlichen oder paketöffentlichen Methode überschrieben werden.
- Private Methoden werden grundsätzlich nicht überschrieben, d. h. wenn eine Methode einer Klasse dieselbe Signatur wie eine private Methode einer Oberklasse besitzt, stehen diese Methoden in keinerlei Beziehung zueinander, und die o. g. Regeln sind in diesem Fall bedeutungslos.

5.6 Überschreiben und Überladen von Methoden

- *Überschreiben* und *Überladen* von Methoden (vgl. § 3.5) sind voneinander unabhängige Konzepte, die bei Bedarf auch kombiniert werden können.

- Die Auswahl der ausgeführten Methode erfolgt dann in mehreren Schritten:
  - Der statische Typ des Aufrufobjekts bestimmt die Klasse, in der der Übersetzer nach dem Methodennamen sucht.
  - Die statischen Typen der Aufrufparameter bestimmen (zur Übersetzungszeit) die am besten passende Methode.
  - Wenn diese Methode in Unterklassen überschrieben ist, bestimmt der dynamische Typ des Aufrufobjekts zur Laufzeit die passende Implementierung dieser Methode.

Beispiel

```
class A {
  void m (double x) { System.out.println("A.m(double)"); }
}

class B extends A {
  // Keine Überschreibung, sondern eine Überladung der
  // geerbten Methode m, da der Parametertyp anders ist.
  void m (int x) { System.out.println("B.m(int)"); }
}

class C extends B {
  // Überschreibung der indirekt von A geerbten Methode m.
  @Override
  void m (double x) { System.out.println("C.m(double)"); }

  // Überschreibung der von B geerbten Methode m.
  @Override
  void m (int x) { System.out.println("C.m(int)"); }
}
```

```
class Test {
  public static void main (String [] args) {
    A a = new C(); a.m(1); // C.m(double)
    B b = new C(); b.m(1); // C.m(int)
    C c = new C(); c.m(1); // C.m(int)
  }
}
```

- Weil `a` den statischen Typ `A` besitzt, findet der Übersetzer für den Aufruf `a.m(1)` nur die in der Klasse `A` definierte Methode.
  Weil `a` den dynamischen Typ `C` besitzt, wird die in der Klasse `C` definierte Überschreibung dieser Methode ausgeführt.

- Weil `b` den statischen Typ `B` besitzt, findet der Übersetzer für den Aufruf `b.m(1)` sowohl die in der Klasse `B` definierte als auch die von `A` geerbte Methode. Gemäß der Regeln in § 3.5 passt die in `B` definierte Methode besser.
  Weil `b` den dynamischen Typ `C` besitzt, wird die in der Klasse `C` definierte Überschreibung dieser Methode ausgeführt.

- Dto. für den Aufruf `c.m(1)`.

5.7 Beispiel: Limitierte Konten (vgl. § 2.2 und § 4.4)

```
// Unterklasse von Account: Limitiertes Konto.
class LimitedAccount extends Account {
  // Zusätzliche Objektvariable:
  private int limit;              // Kreditlinie in Cent.

  // Konstruktoren:
  // Limitiertes Konto mit Inhaber h, ggf. Anfangsbetrag b,
  // Kreditlinie l und eindeutiger Nummer initialisieren.
  public LimitedAccount (String h, int b, int l) {
    super(h, b); // Konstruktor der Oberklasse Account aufrufen,
                 // um deren Objektvariablen zu initialisieren.
    limit = l;   // Zusätzliche Objektvariable limit initialisieren.
  }
  public LimitedAccount (String h, int l) {
    // Entweder:                  // Oder:
    super(h);                     // this(h, 0, l);
    limit = l;                    //
  }

  // Zusätzliche Objektmethode: Kreditlinie abfragen.
  public int limit () { return limit; }
```

```
  // Hilfsmethode: Kann Betrag amount abgezogen werden,
  // ohne die Kreditlinie zu überschreiten?
  private boolean check (int amount) {
    if (balance() - amount >= -limit) return true;
    System.out.println("Unzulässige Kontoüberziehung!");
    return false;
  }

  // Überschreiben geerbter Objektmethoden:
  // Betrag amount abheben/überweisen.
  public void withdraw (int amount) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.withdraw(amount);
    }
  }
  public void transfer (int amount, Account that) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.transfer(amount, that);
    }
  }
}
```

```
// Testklasse.
class Test {
  public static void main (String [] args) {
    // Objekte erzeugen und durch Konstruktoraufrufe initialisieren.
    Account a = new LimitedAccount("Hans Maier", 500);
    Account b = new Account("Fritz Müller");

    // Da deposit in LimitedAccount nicht überschrieben ist,
    // wird die Account-Implementierung der Methode ausgeführt.
    a.deposit(1000);

    // Da a den dynamischen Typ LimitedAccount besitzt,
    // wird die LimitedAccount-Implementierung von transfer
    // und withdraw ausgeführt (was bei withdraw zu einer
    // unzulässigen Kontoüberziehung führt).
    a.transfer(300, b);
    a.withdraw(2000);

    // Da b den dynamischen Typ Account besitzt,
    // wird die Account-Implementierung von withdraw ausgeführt.
    b.withdraw(2000);
  }
}
```

5.8 Dynamische Typtests und statische Typumwandlungen

- Der Ausdruck `object instanceof type` liefert genau dann `true`, wenn der dynamische Typ des Objekts `object` ein (trivialer, direkter oder indirekter) Untertyp von `type` ist (woraus folgt, dass `object` nicht `null` ist).

- Wenn die Bedingung `object instanceof type` erfüllt ist (oder `object` gleich `null` ist), liefert der Ausdruck `(type)object` das Objekt `object` mit statischem Typ `type` (und unverändertem dynamischen Typ); andernfalls erhält man eine Ausnahme des Typs `ClassCastException`.

- Wenn `type` ein Obertyp des statischen Typs von `object` ist, bezeichnet man die Typumwandlung `(type)object` als *Aufwärtsumwandlung* (up-cast).
  Da `object` in diesem Fall sowieso überall verwendet werden kann, wo ein Objekt des Typs `type` erwartet wird (vgl. § 5.3), ist eine solche Umwandlung normalerweise nicht erforderlich; sie kann jedoch notwendig sein, um den Aufruf einer überladenen Methode eindeutig zu machen (vgl. § 3.5) oder um auf ein verborgenes Feld zuzugreifen (vgl. § 5.9).

- Wenn `type` ein Untertyp des statischen Typs von `object` ist, bezeichnet man die Typumwandlung `(type)object` als *Abwärtsumwandlung* (down-cast).

- Wenn `type` weder ein Ober- noch ein Untertyp des statischen Typs von `object` ist, wird die Umwandlung vom Übersetzer zurückgewiesen (siehe jedoch § 6.2.2).

- Der Ausdruck `object instanceof type var` liefert dasselbe wie der Ausdruck `object instanceof type`. Wenn dieser Wert `true` ist, wird außerdem die dadurch nebenbei definierte lokale Variable `var` mit Typ `type` mit `(type)object` initialisiert. Wenn der Wert `false` ist, bleibt die Variable jedoch uninitialisiert.

- Dementsprechend ist die Variable anschließend nur an Stellen sicht- und verwendbar, die bei der Programmausführung garantiert nur erreicht werden können, wenn der Ausdruck den Wert `true` hatte.

- Zum Beispiel:

```
Account a = ......; // a kann auch ein LimitedAccount enthalten.
if (a instanceof LimitedAccount la) {
  // Hier ist la sicht- und verwendbar, weil dieser Code nur
  // ausgeführt wird, wenn a instanceof LimitedAccount wahr ist.
}
else {
  // Hier ist la nicht sichtbar, weil dieser Code ausgeführt
  // wird, wenn a instanceof LimitedAccount nicht wahr ist.
}

if (!(a instanceof LimitedAccount la)) return;
// Hier ist la sicht- und verwendbar, weil auch dieser Code nur
// ausgeführt wird, wenn a instanceof LimitedAccount wahr ist.
```

5.9 Verbergen von Feldern

- Wenn ein Feld (Objekt- oder Klassenvariable) einer Klasse denselben Namen wie ein geerbtes Feld besitzt, ist das geerbte Feld in dieser Klasse zwar vorhanden, aber *verborgen* (engl. *hidden*), d. h. nicht direkt über seinen Namen ansprechbar.

- Eine verborgene Objektvariable `objvar` des aktuellen Objekts kann über `super.objvar` angesprochen werden.

- Eine verborgene Objektvariable eines beliebigen Objekts `object` kann durch eine explizite Typumwandlung in die passende Oberklasse angesprochen werden: `((superclass)object).objvar`

- Allgemein: Zugriffe auf Objektvariablen werden *statisch gebunden*, d. h. bei einem Zugriff `object.objvar` bestimmt allein der *statische* Typ von `object`, welche Variable ausgewählt wird.

- Eine verborgene Klassenvariable `classvar` kann wie folgt aufgerufen werden:
  - `superclass.classvar`
  - `super.classvar` (innerhalb einer Objektmethode)
  - `((superclass)object).classvar` (unüblich)

5.10 Die Wurzelklasse `Object`

- Wenn eine Klasse keine explizite Oberklasse besitzt, besitzt sie implizit die Wurzelklasse `Object` als Oberklasse.
- Daraus folgt, dass `Object` eine direkte oder indirekte Oberklasse jeder Klasse ist (vgl. Abbildung in § 5.1).
- Daher kann eine Variable mit statischem Typ `Object` zur Laufzeit Objekte beliebiger Klassen referenzieren.
- Die Klasse `Object` besitzt einen öffentlichen parameterlosen Konstruktor und (unter anderem) die folgenden öffentlichen Methoden (die von Unterklassen überschrieben werden können):

- `boolean equals (Object other)`
  Dient grundsätzlich zum Vergleich des aktuellen Objekts `this` mit irgendeinem anderen Objekt `other`.
  Wenn die Methode nicht überschrieben ist, liefert sie genau dann `true`, wenn `this` und `other` *dasselbe* Objekt referenzieren.
  Um die Methode zu überschreiben, muss man (wie immer) eine Methode mit derselben Signatur definieren; insbesondere muss als Parametertyp `Object` (und nicht die aktuelle Klasse) verwendet werden.
- `int hashCode ()`
  Liefert grundsätzlich einen Streuwert für das aktuelle Objekt `this`. Für Objekte, die bzgl. `equals` „gleich“ sind, muss derselbe Streuwert geliefert werden (d. h. wenn man `equals` überschreibt, muss man i. d. R. auch `hashCode` überschreiben). „Verschiedene“ Objekte sollten nach Möglichkeit verschiedene Streuwerte besitzen (was aber häufig nicht möglich ist).
  Wenn die Methode nicht überschrieben ist, liefert sie typischerweise die interne Adresse des aktuellen Objekts als ganze Zahl.
- `String toString ()`
  Liefert grundsätzlich eine Zeichenkettendarstellung des aktuellen Objekts `this` und wird automatisch aufgerufen, um das Objekt in eine Zeichenkette umzuwandeln (vgl. § 3.2.4).
  Wenn die Methode nicht überschrieben ist, liefert sie den Namen der Klasse des Objekts, gefolgt von einem @-Zeichen, gefolgt von einer hexadezimalen Darstellung des Streuwerts des Objekts.

Beispiel

```
// Punkt im zweidimensionalen Raum.
class Point {
  // Koordinaten des Punkts.
  public final double x, y;

  // Punkt mit Koordinaten x und y initialisieren.
  public Point (double x, double y) {
    this.x = x;
    this.y = y;
  }

  // Zeichenkettendarstellung des aktuellen Objekts liefern.
  @Override
  public String toString () {
    return "(" + x + ", " + y + ")";
  }
```

```
  // Vergleich des aktuellen Objekts this (mit Typ Point)
  // mit irgendeinem anderen Objekt other (mit beliebigem
  // dynamischem Typ).
  @Override
  public boolean equals (Object other) {
    // Wenn other kein Point that ist,
    // kann es nicht gleich this sein.
    if (!(other instanceof Point that)) return false;

    // Andernfalls können this und that
    // inhaltlich verglichen werden.
    return this.x == that.x && this.y == that.y;
  }

  // Streuwert für das aktuelle Objekt liefern.
  @Override
  public int hashCode () {
    // Für Punkte, die gemäß equals gleich sind,
    // erhält man den gleichen Streuwert.
    return (int)(x + y);
  }
}
```

Unterschied zwischen Überschreiben und Überladen von `equals`

- Wenn man `a` und `b` jeweils durch eine der folgenden Variablen ersetzt, liefert `a.equals(b)` die in der Tabelle dargestellten Resultatwerte (T bedeutet `true`, F bedeutet `false`):

```
Point p1 = new Point(3, 4);
Point p2 = new Point(2, 2);
Object p3 = new Point(3, 4);

String s1 = "hello";
String s2 = "world";
Object s3 = "hello";
```

a \ b	p1	p2	p3	s1	s2	s3
p1	T	F	**T**	F	F	F
p2	F	T	F	F	F	F
p3	**T**	F	T	F	F	F
s1	F	F	F	T	F	T
s2	F	F	F	F	T	F
s3	F	F	F	T	F	T

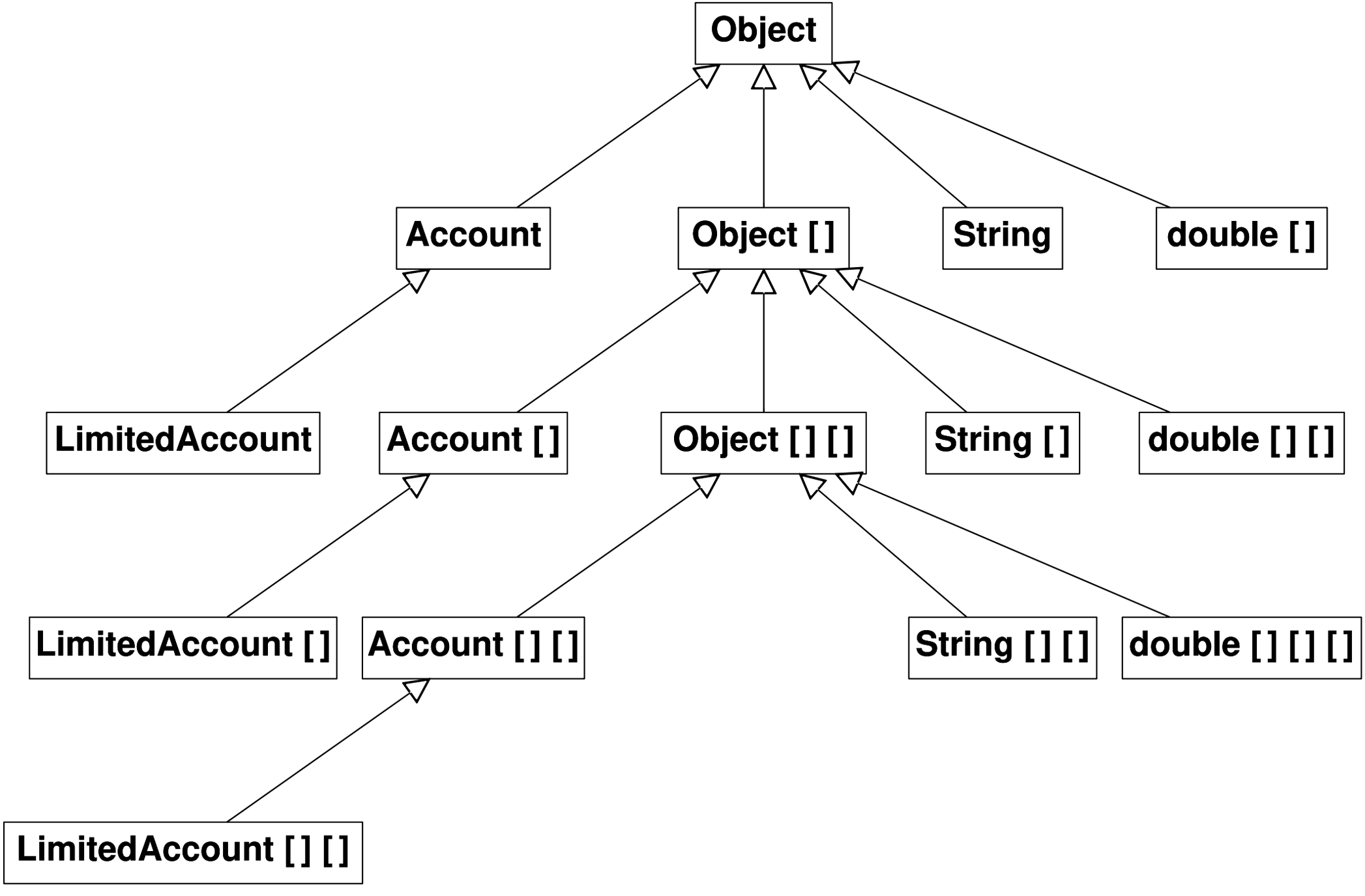
- Wenn man die Methode `equals` in der Klasse `Point` nicht überschreibt, sondern durch die folgende andere Methode mit Parametertyp `Point` *überlädt* – was auf den ersten Blick einfacher und naheliegender erscheint –, dann wird bei `p3.equals(b)` immer die Methode der Klasse `Object` ausgeführt, die für `b` gleich `p1` den unzutreffenden Wert `false` liefert. Ebenso wird bei `p1.equals(p3)` die „falsche“ Methode mit dem falschen Ergebnis ausgeführt. (Das heißt, die beiden in der Tabelle hervorgehobenen Werte stimmen dann nicht mehr.)

```
// Vergleich des aktuellen Objekts this (mit Typ Point)
// mit einem anderen Objekt that des Typs Point.
public boolean equals (Point that) {
  return this.x == that.x && this.y == that.y;
}
```

- Deshalb sollte `equals` immer durch eine Methode mit Parametertyp `Object` *überschrieben* werden, auch wenn dies die Implementierung etwas verkompliziert.

5.11 Reihen (arrays)

- Reihentypen sind formal ebenfalls Klassen (mit einer öffentlichen, unveränderlichen Objektvariablen `length`) und somit Unterklassen von `Object`.
- Dementsprechend besitzen sie auch die Methoden `equals`, `hashCode` und `toString`, die jedoch nicht überschrieben sind, sodass ihr Verhalten nicht besonders nützlich ist. (Insbesondere vergleicht `a.equals(b)` nur die zwei Referenzen `a` und `b` und führt keinen elementweisen Vergleich durch.)
- Außerdem können Reihentypen, genauso wie Klassen, in dynamischen Typtests (`instanceof`) und statischen Typumwandlungen (Cast) verwendet werden.
- Wenn `S` ein Untertyp von `T` ist, dann ist auch `S []` ein Untertyp von `T []`, d. h. eine `S`-Reihe kann überall verwendet werden, wo eine `T`-Reihe erwartet wird.
- Durch wiederholte Anwendung dieser Regel ergibt sich auch, dass `S [] []` ein Untertyp von `T [] []` ist, usw.
- Außerdem folgt für jede Klasse (oder Schnittstelle) `T`, dass der Reihentyp `T []` ein Untertyp von `Object []` ist.
- Abgesehen von diesen Sonderregeln, können Reihentypen jedoch nicht als Oberklassen anderer Klassen verwendet werden.


Object
Account
Object [ ]
String
double [ ]
LimitedAccount
Account [ ]
Object [ ] [ ]
String [ ]
double [ ] [ ]
LimitedAccount [ ]
Account [ ] [ ]
String [ ] [ ]
double [ ] [ ] [ ]
LimitedAccount [ ] [ ]

- Wenn `a` eine Variable des Typs `T []` ist, akzeptiert der Übersetzer Zuweisungen der Art `a[i] = t`, sofern `t` den (statischen) Typ `T` besitzt. Wenn `a` zur Laufzeit jedoch eine Reihe des Typs `S []` referenziert, ist eine solche Zuweisung nur dann semantisch korrekt, wenn `t` den dynamischen Typ `S` besitzt; wenn dies nicht der Fall ist, erhält man eine Ausnahme des Typs `ArrayStoreException`.

```
// LimitedAccount-Reihe las erzeugen und mit Objekten füllen.
LimitedAccount [] las = new LimitedAccount [10];
las[0] = new LimitedAccount(...);
......

// las an Account-Reihenvariable as zuweisen.
Account [] as = las;

// Lesezugriffe über as sind unkritisch.
Account a = as[0];

// Zuweisungen von LimitedAccount-Objekten sind korrekt.
as[0] = new LimitedAccount(...);

// Zuweisungen von Account-Objekten sind nicht zulässig,
// da die von as referenzierte Reihe eine LimitedAccount-Reihe ist.
as[0] = new Account(...);       // ArrayStoreException!!
```

5.12 Unveränderliche Methoden und Klassen

- Eine Methode, die `final` deklariert ist, kann in Unterklassen nicht überschrieben werden.
  (Dementsprechend sind private Methoden quasi `final`; vgl. § 5.5.)

- Eine Klasse, die `final` deklariert ist, kann keine Unterklassen besitzen (und dementsprechend können ihre Methoden nicht überschrieben werden, d. h. sie sind implizit quasi ebenfalls `final`).

- Beide Arten von `final`-Deklarationen erlauben einem Übersetzer u. U., effizienteren Code zu generieren, weil Methoden, die nicht überschrieben werden können, statisch gebunden und ggf. inline expandiert werden können.

- Andererseits erschweren oder verhindern derartige `final`-Deklarationen spätere (möglicherweise unvorhergesehene) Erweiterungen eines bestehenden Software-Systems um neue Unterklassen und widersprechen damit eigentlich einem Grundprinzip der Objektorientierung.

- Die Bibliotheksklasse `String` ist `final`.

- Reihentypen sind „semi-`final`“, da sie nicht als Oberklassen anderer Klassen verwendet werden können, aber trotzdem andere Reihentypen als Untertypen besitzen können (vgl. § 5.11).

5.13 Präzise Regeln für unterklassenöffentliche Elemente

- Eine Klasse darf auf ein unterklassenöffentliches Element ihrer Oberklasse(n) zugreifen,
  - wenn sich die Klasse im selben Paket wie die Oberklasse befindet (vgl. Kap. 9)
  - oder wenn das Element statisch ist
  - oder wenn der (statische) Typ des Objekts, auf dessen Element zugegriffen wird, ein (trivialer, direkter oder indirekter) Untertyp der Klasse ist (d. h. wenn die Klasse an der Implementierung des Objekts „beteiligt“ ist)

- Im Beispiel von § 5.1 darf Code der Klasse `Student`, sofern diese Klasse nicht zum selben Paket wie `Person` gehört, die unterklassenöffentlichen Objektvariablen, Objektmethoden und Konstruktoren von `Person` nur verwenden, wenn das betroffene Objekt Typ `Student`, `BachelorStudent` oder `MasterStudent` besitzt, aber nicht, wenn es Typ `Person`, `Man` oder `Woman` besitzt.

5.14 Beispiele: Listen (vgl. § 4.13)

5.14.1 Doppelt verkettete Listen

```
// Unterklasse von List: Doppelt verkettete Liste.
class BiList extends List {
  // Zusätzliche Objektvariable:
  protected BiList prev; // Verweis auf voriges Listenelement.

  // Konstruktoren: Doppelt verkettete Liste mit erstem
  // Element h und ggf. Restliste t initialisieren.
  public BiList (int h, BiList t) {
    // Konstruktor der Oberklasse List aufrufen,
    // um deren Objektvariablen head und tail zu initialisieren.
    super(h, t);          // Implizite Aufwärtsumwandlung von t.

    // Rückwärtsverkettung über Objektvariable prev herstellen.
    if (t != null) t.prev = this;
    prev = null;
  }
  public BiList (int h) {
    this(h, null);
  }
```

```
  // Zusätzliche Objektmethoden: Vorgänger und Nachfolger abfragen.
  public BiList next () {
    return (BiList)tail; // Explizite Abwärtsumwandlung notwendig,
                         // da tail vom (statischen) Typ List ist.
  }
  public BiList prev () {
    return prev;
  }

  // Die geerbten Objektmethoden head, tail, length und print
  // werden unverändert übernommen.

}
```

5.14.2 Zirkuläre Listen

```
// Unterklasse von List: Zirkuläre Liste.
class CircList extends List {
  // Keine zusätzlichen Objektvariablen.

  // Konstruktoren: Zirkuläre Liste mit erstem
  // Element h und ggf. Restliste t initialisieren.
  public CircList (int h, CircList t) {
    // Konstruktor der Oberklasse List aufrufen,
    // um deren Objektvariablen head und tail zu initialisieren.
    super(h, t);          // Implizite Aufwärtsumwandlung von t.

    // Zirkuläre Verkettung herstellen.
    if (t == null) {
      tail = this;        // Implizite Aufwärtsumwandlung von this.
    }
    else {
      List p = t;         // Implizite Aufwärtsumwandlung von t.
      while (p.tail != t) p = p.tail;   // Dto.
      p.tail = this;      // Implizite Aufwärtsumwandlung von this.
    }
  }
```

```
  public CircList (int h) { this(h, null); }

  // Überschriebene Objektmethode mit kovarianter Anpassung
  // des Resultattyps (CircList statt List).
  public CircList tail () { return (CircList)tail; }

  // Überschriebene Objektmethode:
  // Länge der zirkulären Liste ermitteln.
  public int length () {
    int n = 1;
    for (List p = tail; p != this; p = p.tail) n++;
    return n;
  }

  // Überschriebene Objektmethode:
  // Zirkuläre Liste ausgeben.
  public void print () {
    System.out.println(head);
    for (List p = tail; p != this; p = p.tail) {
      System.out.println(p.head);
    }
  }
}
```