TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 03b: Systematische Abrundung bisheriges Java: Referenztypen

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Begriffsbildung: Referenztypen

Der Inhalt des ersten Abschnitts hat dem ganzen Kapitel seinen Namen gegeben. Man kann einfach sagen, Referenztypen sind alle Typen, die nicht primitive Datentypen sind.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

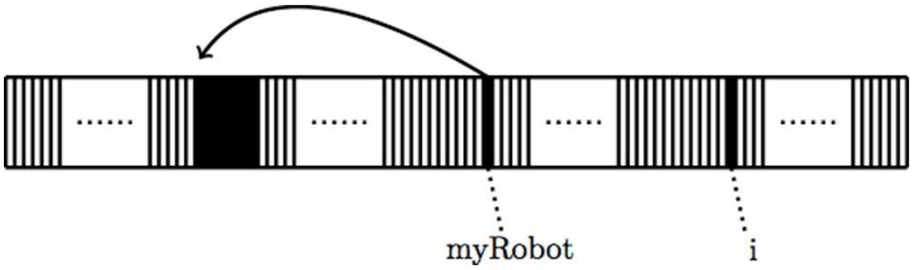
- Klassen
- Interfaces
- Arraytypen
- Enum-Typen

Einsichtsreicher ist es natürlich, die verschiedenen Arten von Referenztypen aufzuzählen; das sind sie. Wir gehen diese vier Arten der Reihe nach durch, um jeweils zu klären, warum sie unter dem Begriff Referenztypen subsumiert werden. Wir halten aber schon einmal fest: Abgesehen von Array-Typen sind das genau die Typen, die nicht fest in der Sprache Java eingebaut sind.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassen:



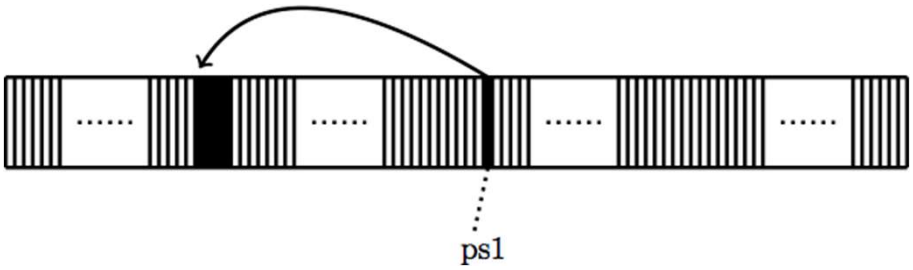

```
Robot myRobot = new Robot ( ..... );
int i = 1;
```

Bei Klassen hatten wir schon mehrfach gesehen, was Referenztypen sind, nämlich Typen, bei denen zwischen Referenz und eigentlichem Objekt unterschieden wird – eben im Gegensatz zu primitiven Datentypen, bei denen *kein* solcher Unterschied gemacht wird.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Interfaces:



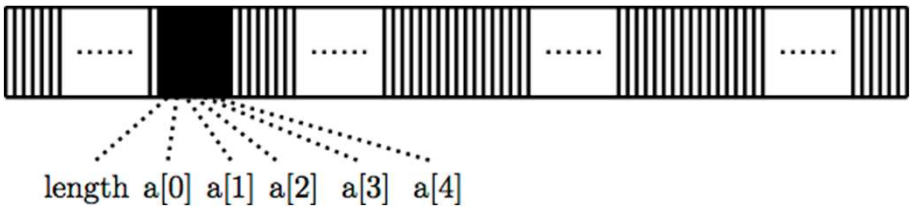
PacmanStrategy ps1 = new PacmanStrategy1 (.....);

Anhand des Beispiels unten hatten wir schon gesehen, dass das Bild bei Interfaces dasselbe ist, also ebenfalls Referenztypen.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytypen:




```
double[ ] a = new double [5];    // Typ „Array von double“
```

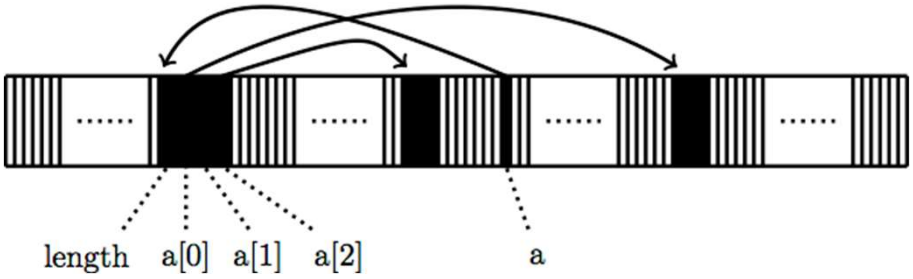
Schlussendlich haben wir ebenfalls schon gesehen, dass bei Arrays im Prinzip dasselbe Schema realisiert ist, also ebenfalls Referenztyp.

Was genau sind Arraytypen? Zu jedem Typ X – egal ob X ein primitiver Datentyp oder ein Referenztyp ist, gibt es einen Arraytyp, nämlich „Array von X“. Im Beispiel sehen wir also den Typ „Array von double“, das heißt, der Komponententyp ist double.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytypen:




```
Robot[ ] a = new Robot [3];      // Typ „Array von Robot“
a[0] = new Robot ( ..... );
a[2] = new Robot ( ..... );
```

Wie wir ebenfalls schon gesehen haben, können Arrays von Referenztypen gebildet werden. Die einzelnen Komponenten sind dann die Referenzen, die auf Objekte des Komponententyps verweisen. Im Beispiel ist der Komponententyp Robot, der Arraytyp ist also „Array von Robot“.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytypen:

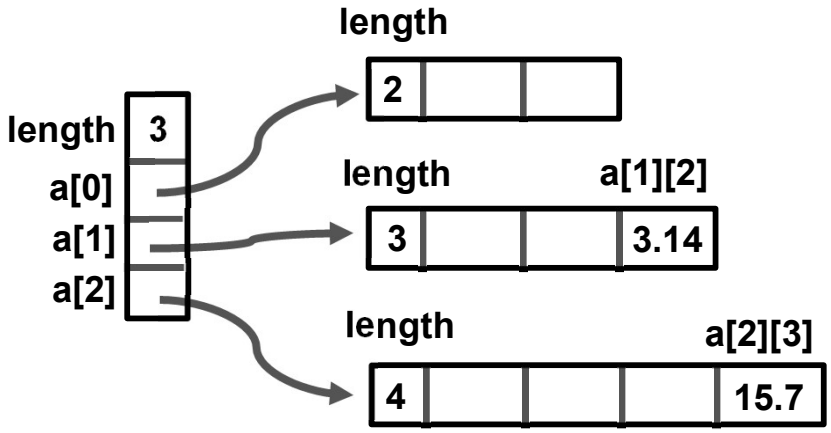
```
double[ ] a = new double [ 3 ];
a[0] = 3.14;
a[1] = 2.71;
a[2] = a[0] + 3 * a[1];
```

Hier noch einmal kurz ein Beispiel dafür, wie man mit den Komponenten eines Arrays umgeht. Es muss ja nicht immer eine Schleife über die Komponenten sein.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von double“:



Da man jeden Typ als Komponententyp haben kann, kann man natürlich auch einen Typ „Array von ...“ als Komponententyp haben. Links sehen Sie ein Arrayobjekt mit Länge 3, auf das eine Referenz namens a verweist. Der Komponententyp ist Array von double, das heißt, die einzelnen Komponenten verweisen auf Arrayobjekte oder enthalten den symbolischen Wert null.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von double“:

```
double[ ][ ] a = new double [ 3 ] [ ];
a[0] = new double [ 2 ];
a[1] = new double [ 3 ];
a[2] = new double [ 4 ];
a[1][2] = 3.14;
a[2][3] = 5 * a[1][2];
```

Mit diesen Anweisungen wird die Konstellation erzeugt, die auf der letzten Folie visualisiert wurde.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von double“:

```
double[ ][ ] a = new double [ 3 ] [ ];
a[0] = new double [ 2 ];
a[1] = new double [ 3 ];
a[2] = new double [ 4 ];
a[1][2] = 3.14;
a[2][3] = 5 * a[1][2];
```

Hier sehen Sie, wie eine Referenz und ein Objekt vom Typ „Array von Array von double“ eingerichtet werden.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von double“:

```
double[ ][ ] a = new double [ 3 ] [ ];
a[0] = new double [ 2 ];
a[1] = new double [ 3 ];
a[2] = new double [ 4 ];
a[1][2] = 3.14;
a[2][3] = 5 * a[1][2];
```

Die einzelnen Komponenten sind ja Arrays von double und müssen dementsprechend eingerichtet werden. Solange das mit einer dieser Komponenten nicht passiert, steht darin der symbolische Wert null.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von double“:

```
double[ ][ ] a = new double [ 3 ] [ ];
a[0] = new double [ 2 ];
a[1] = new double [ 3 ];
a[2] = new double [ 4 ];
a[1][2] = 3.14;
a[2][3] = 5 * a[1][2];
```

So sieht dann der schreibende beziehungsweise lesende Zugriff auf eine einzelne Arraykomponente aus. Die eckigen Klammern können als Selektionsoperator verstanden werden, der auf eine Referenz vom Arraytyp angewendet wird und eine einzelne Komponente zurückliefert. Bei einem Array von Array von double muss dieser Selektionsoperator natürlich zweimal angewendet werden, um auf eine double-Komponente zuzugreifen.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Arraytyp „Array von Array von Robot“:

```
Robot[ ][ ] a = new Robot [ 3 ] [ ];
a[0] = new Robot [ 2 ];
a[1] = new Robot [ 3 ];
a[2] = new Robot [ 4 ];
a[1][2] = new Robot ( ..... );
a[1][2].move();
```

Analog dazu sieht ein Array von Array mit einem Referenztyp als Komponententyp aus.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Enum-Typen:

- **Sind alle abgeleitet von Klasse java.lang.Enum**
  - **Oracle Java Tutorials: Enum Types**
- **Alle von java.lang.Enum abgeleiteten Klassen sind Enum-Typen**
- **Besonderheit: neben den vordefinierten Konstanten keine weiteren Objekte**

Die offene Frage ist, warum Enum-Typen ebenfalls Referenztypen sind. Deren interne Realisierung hatten wir ja bisher nicht betrachtet.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Enum-Typen:

- **Sind alle abgeleitet von Klasse java.lang.Enum**
  - **Oracle Java Tutorials: Enum Types**
- **Alle von java.lang.Enum abgeleiteten Klassen sind Enum-Typen**
- **Besonderheit: neben den vordefinierten Konstanten keine weiteren Objekte**

Enum-Typen sind tatsächlich Klassen und damit Referenztypen.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Enum-Typen:

- **Sind alle abgeleitet von Klasse java.lang.Enum**
  - **Oracle Java Tutorials: Enum Types**
- **Alle von java.lang.Enum abgeleiteten Klassen sind Enum-Typen**
- **Besonderheit: neben den vordefinierten Konstanten keine weiteren Objekte**

Umgekehrt können Sie von Klasse Enum im Package java.lang keine Klassen einfach so auf die normale Weise ableiten, das verhindert der Compiler auf die übliche Art: Fehlermeldung und Abbruch der Übersetzung.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

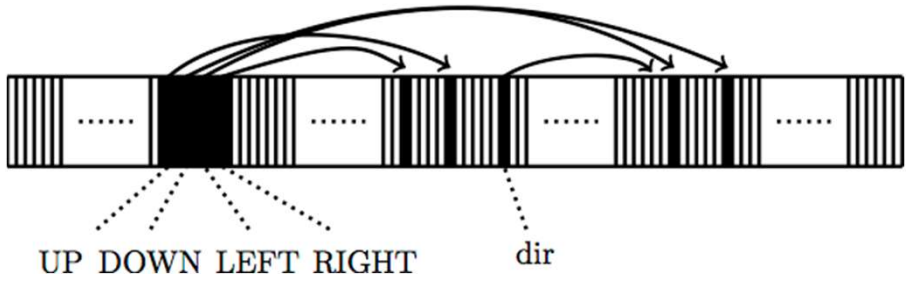
Enum-Typen:

- **Sind alle abgeleitet von Klasse java.lang.Enum**
  - **Oracle Java Tutorials: Enum Types**
- **Alle von java.lang.Enum abgeleiteten Klassen sind Enum-Typen**
- **Besonderheit: neben den vordefinierten Konstanten keine weiteren Objekte**

Enum-Typen sind spezielle Klassen: In der Definition des Enum-Typs definiert man ein paar Konstanten, darüber hinaus lassen sich keine Objekte des Enum-Typs definieren.

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public enum MyDirection {
   UP, DOWN, LEFT, RIGHT
}
```

```
MyDirection dir = MyDirection.DOWN;
```

Hier sehen Sie noch einmal wie in Kapitel 01e, Abschnitt „Erste eigene Enumeration“, wie man sich das im Computerspeicher vorstellen kann. Es werden genau diese vier Objekte von MyDirection zu Beginn der Ausführung des Programms erzeugt,

***Nebenbemerkung*: Wenn die Enumeration wie hier nur wenige Konstanten enthält und diese auch noch kurze Namen haben, ist es nicht unüblich, diese Namen nicht untereinander, sondern nebeneinander zu schreiben, so wie Sie das links unten in der Definition von MyDirection sehen.**

Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public enum MyEnum {
    CONSTANT1, CONSTANT2, CONSTANT3
}

MyEnum myVar =  MyEnum.CONSTANT1;
myVar = MyEnum.CONSTANT2;
final MyEnum myConst = MyEnum.CONSTANT1;
```

Hier sehen Sie zur Abrundung noch ein weiteres, rein illustratives Beispiel. Wie Sie sehen, gilt alles, was wir für MyDirection gesagt haben, sinngemäß auch für jeden anderen Enum-Typ.

TECHNISCHE UNIVERSITÄT DARMSTADT

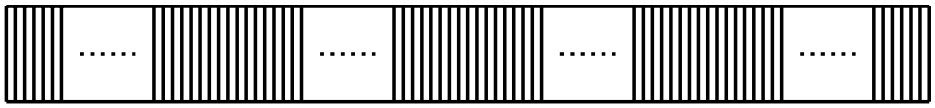
Referenzen und Objekte

Oracle Java Tutorials: Objects

Die strikte Unterscheidung zwischen Referenzen und Objekten ist *die* wichtige Grundlage für objektorientierte Abstraktion, quasi der Blick in den Maschinenraum. Der Inhalt dieses Abschnitts enthält wenig Neues, sondern arbeitet das Thema noch einmal systematisch auf.

Computerspeicher

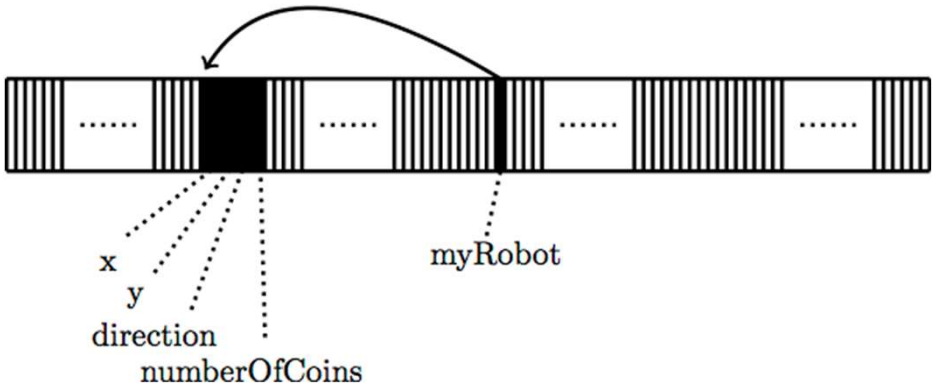
TECHNISCHE UNIVERSITÄT DARMSTADT



Wir haben schon öfters gesehen, wie wir uns die Daten, mit denen wir im Quelltext hantieren, in unserem vereinfachten Modell eines Computerspeichers prinzipiell vorstellen können. Diese Vorstellung entspricht nicht wirklich der Realität im Computer, ist aber ausreichend für unsere Zwecke.

Objekt einer Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

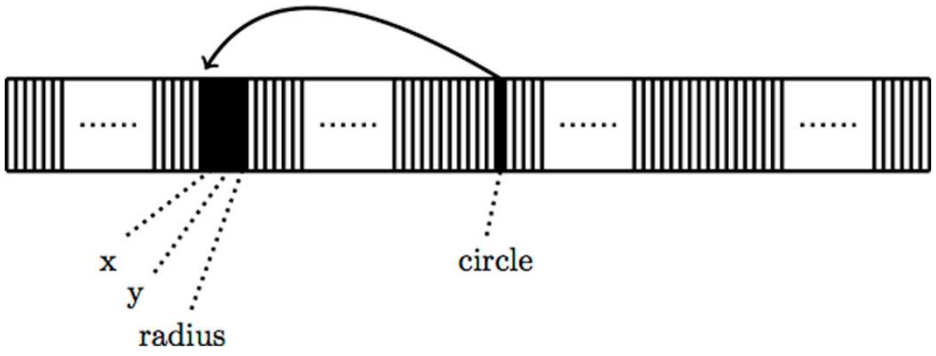


Robot myRobot = new Robot (.....);

Dieses Bild sollte Ihnen sehr vertraut vorkommen, so dass wohl nichts mehr hier dazu zu sagen ist.

Objekt einer Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

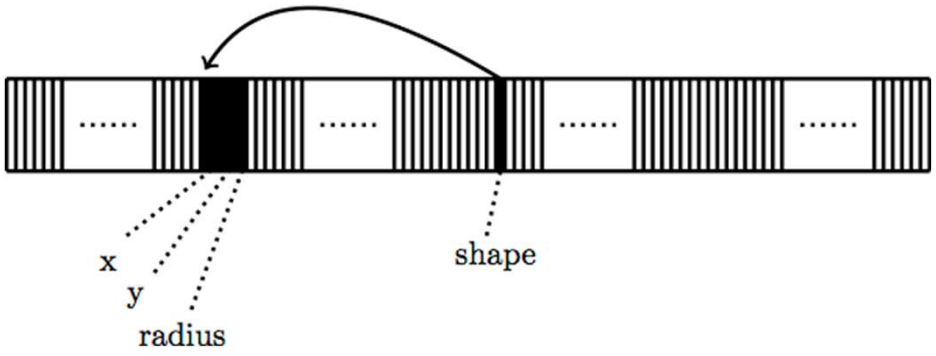


Circle circle = new Circle (300, 400, 100);

Dasselbe gilt natürlich für alle anderen Klassen ebenfalls, zum Beispiel für die Klassen in unserem geometrischen Fallbeispiel.

Objekt einer Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT

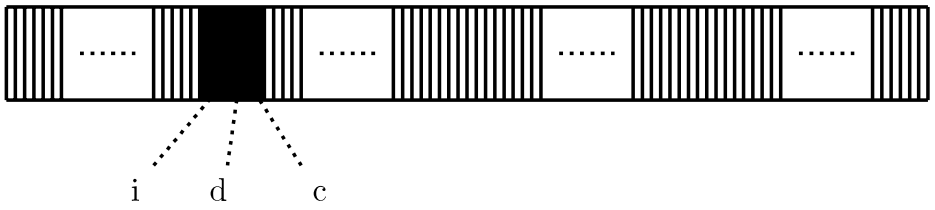


GeomShape2D shape = new Circle (300, 400, 100);

Wir hatten auch schon gesehen, dass Referenz und Objekt nicht vom selben Typ sein müssen, sondern dass das Objekt von einem direkt oder indirekt abgeleiteten Typ sein kann. Sie erinnern sich: Klasse Circle war im geometrischen Fallbeispiel von Klasse GeomShape2D abgeleitet. An der Realisierung im Speicher ändert sich nichts.

Objekt einer Klasse

TECHNISCHE UNIVERSITÄT DARMSTADT



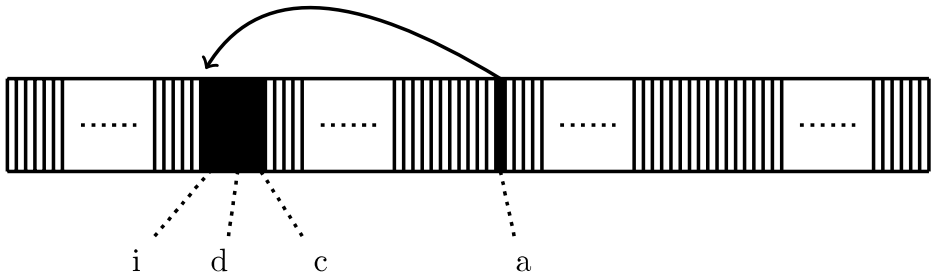

```
public class X {
  public int i;
  public double d;
  public char c;
}
```

```
new X();
```

Hier noch zur Abrundung ein drittes, rein illustratives Beispiel. Sie sehen noch einmal exemplarisch, was der Operator new macht, nämlich ausreichend großen ungenutzten Speicherplatz für das Objekt suchen und diesen für das Objekt reservieren. Mehr macht er nicht.

Variable = Referenz

TECHNISCHE UNIVERSITÄT DARMSTADT



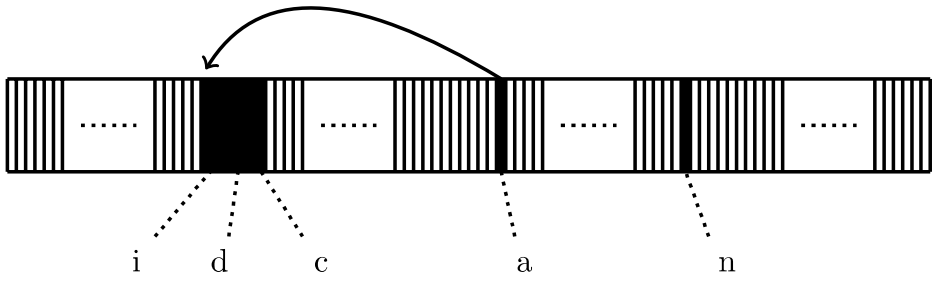

```
public class X {
   public int i;
   public double d;
   public char c;
}
```

```
X a = new X();
```

Nun kommt die Referenz hinzu, in der die von Operator new zurückgelieferte Adresse des neuen Objektes durch die Zuweisung gespeichert wird.

Vergleich mit primitiven Datentypen

TECHNISCHE UNIVERSITÄT DARMSTADT

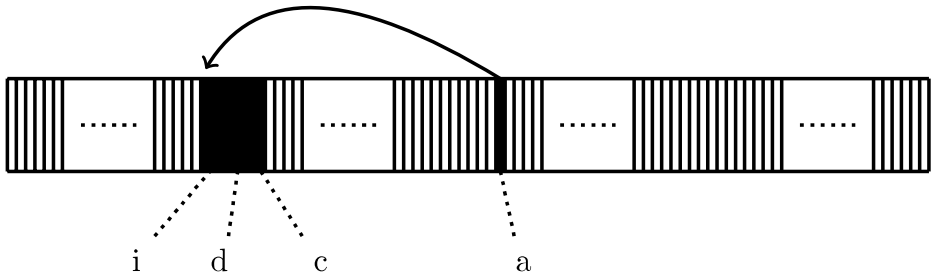



```
int n = 1;
X a = new X();
```

Hier noch einmal der Vergleich mit primitiven Datentypen, bei denen es eben *keine* Trennung von Variable und Objekt gibt.

Variable = Referenz

TECHNISCHE UNIVERSITÄT DARMSTADT



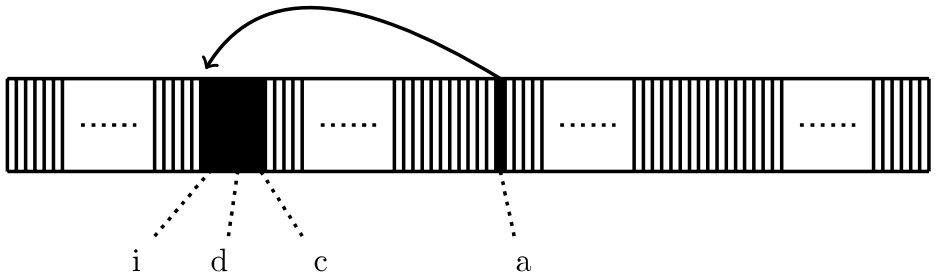

```
public class X {
  public int i;
  public double d;
  public char c;
}
```

```
X a = new X();
a.i = 1;
a.d = 3.14159;
a.c = ´z´;
```

Auf die einzelnen Attribute können Sie lesend und schreibend mit dieser Notation zugreifen: zuerst der Name der Variable, dann der Name des Attributs, beides durch einen Punkt voneinander getrennt. In diesem ersten kleinen Beispiel sind die Zugriffe auf die Attribute i, d und c allesamt schreibend: Die Werte dieser Attribute werden durch Zuweisung auf die jeweils rechts vom Zuweisungszeichen stehenden Werte gesetzt.

Variable = Referenz

TECHNISCHE UNIVERSITÄT DARMSTADT



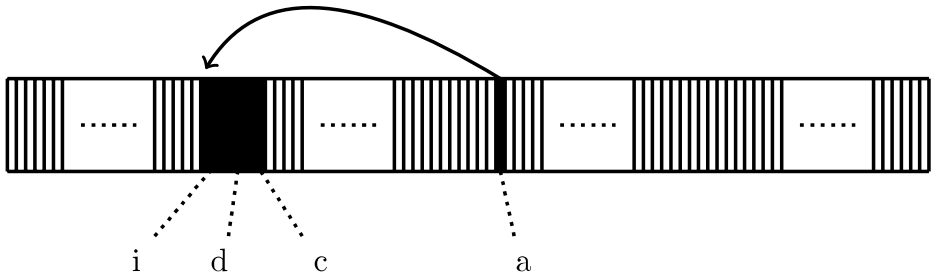

```
public class X {
  public int i;
  public double d;
  public char c;
}
```

```
a.i = a.i + 1;
a.d = a.i;
```

Hier sehen Sie zwei Beispiele für *lesenden* Zugriff auf das Attribut i: In der ersten Zeile wird auf das Attribut i des Objektes lesend, dann schreibend zugegriffen; in der zweiten Zeile auf das Attribut i lesend und auf das Attribut d schreibend.

Variable = Referenz

TECHNISCHE UNIVERSITÄT DARMSTADT



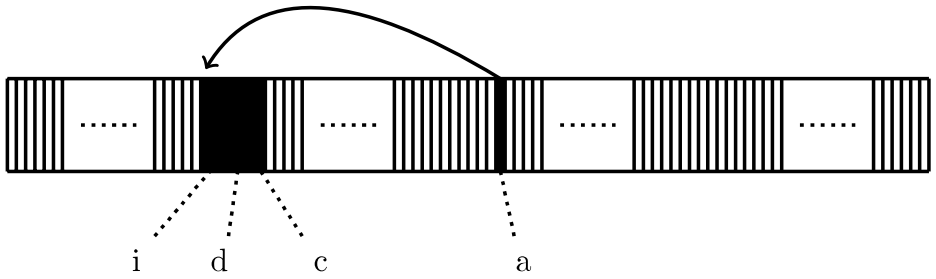

```
public class X {
  public final int i = 1;
  public double d;
  public final char c = ´z´;
}
```

```
X a = new X();
a.i = 1;
a.d = 3.14159;
a.c = ´z´;
```

Es gibt aber auch die Möglichkeit, ein Attribut in der Definition einer Klasse mit Schlüsselwort final als Konstante zu definieren, so wie hier das int-Attribut i und das char-Attribut c. Dann darf später auf diese Attribute nur noch lesend, nicht mehr schreibend zugegriffen werden. Ein final-definiertes Attribut muss zwingend initialisiert werden.
***Vorgriff*: In Kapitel 03c, Abschnitt „Konstruktoren und Static Initializer“, werden wir eine alternative Möglichkeit zur Initialisierung eines Attributs sehen.**

Variable = Referenz

TECHNISCHE UNIVERSITÄT DARMSTADT



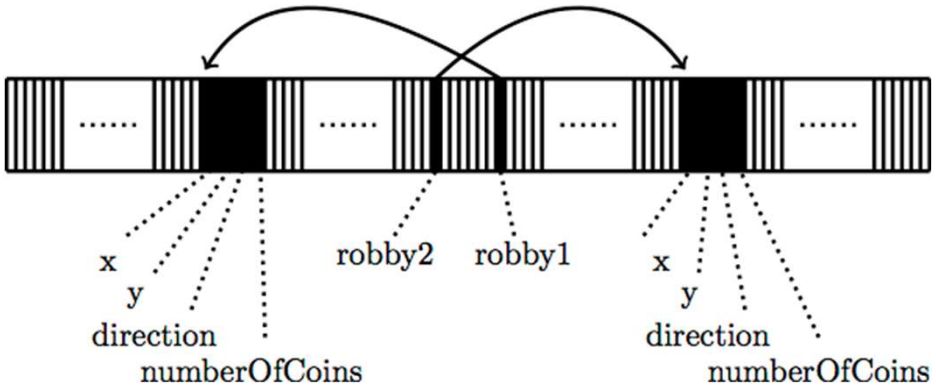

```
public class X {
  public final int i = 1;
  public double d;
  public final char c = ´z´;
}
```

```
X a = new X();
a.i = 1;
a.d = 3.14159;
a.c = ´z´;
```

Das final betrifft genau diese beiden schreibenden Zugriffe. Der Sinn dahinter ist, dass es immer wieder Werte in einem Programm gibt, die garantiert nicht im Programm geändert werden sollen, zum Beispiel die Anzahl der Jahreszeiten oder Naturkonstanten oder mathematische Konstanten wie die Kreiszahl. Um diese Werte vor unabsichtlichem Überschreiben zu schützen, werden sie sinnvollerweise final definiert, dann schützt der Compiler sie: Jeder Versuch, schreibend auf ein final-Attribut bei einem Objekt der Klasse X zuzugreifen, führt zu einer Fehlermeldung beim Kompilieren und zum Abbruch des Übersetzungsvorgangs.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT

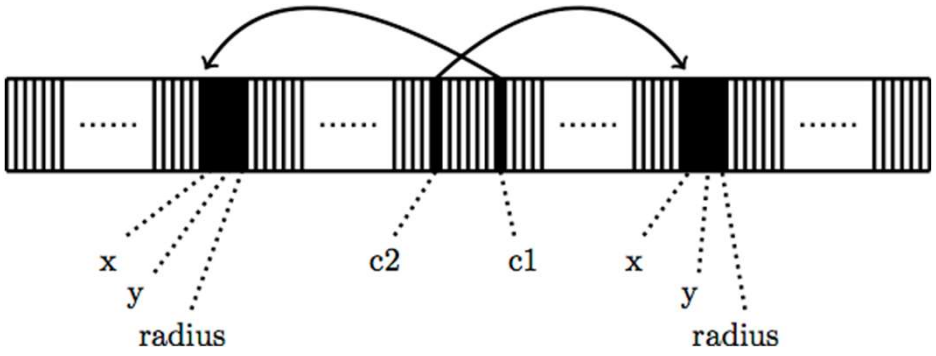



```
Robot robby1 = new Robot ( ..... );
Robot robby2 = new Robot ( ..... );
```

Wenn wir mehrere Referenzen und Objekte einrichten, dann sieht das Bild so aus: Die beiden Paare aus Objekt und Referenz haben nichts miteinander zu tun, sie sind unabhängig voneinander irgendwo im Computerspeicher abgelegt – natürlich an verschiedenen, sich nicht überlappenden Stellen.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT

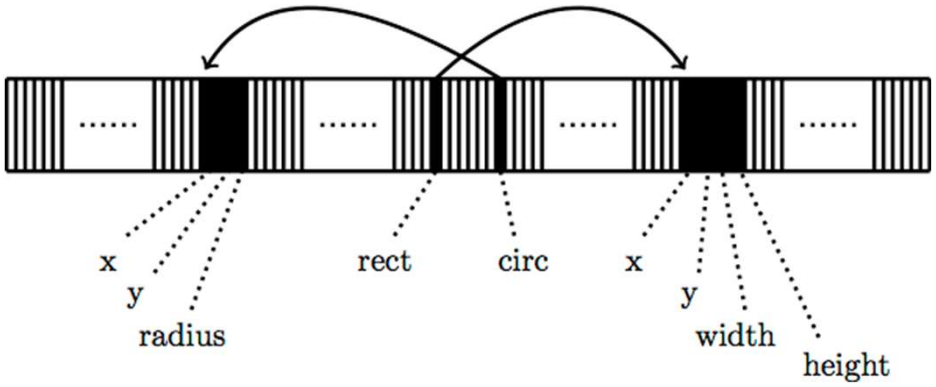



```
Circle c1 = new Circle ( ..... );
Circle c2 = new Circle ( ..... );
```

Noch einmal derselbe Fall mit Klasse Circle aus dem geometrischen Fallbeispiel ...

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT

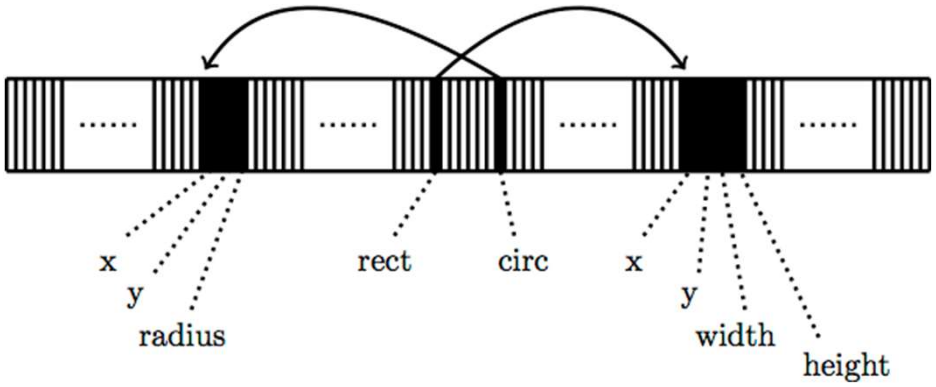



```
Circle circ = new Circle ( ..... );
Rectangle rect = new Rectangle ( ..... );
```

... oder auch mit zwei verschiedenen Klassen.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT

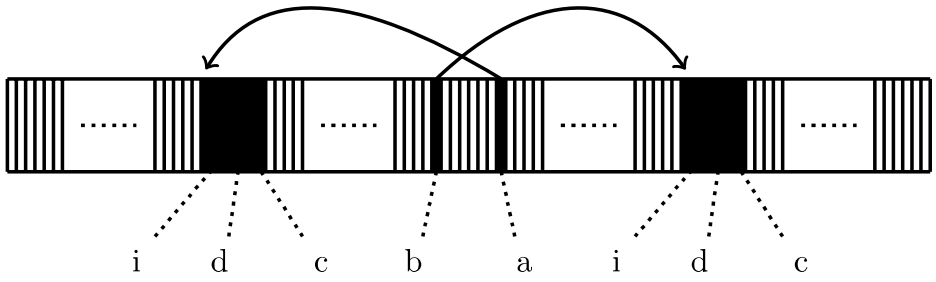



```
GeomShape2D circ = new Circle ( ..... );
GeomShape2D rect = new Rectangle ( ..... );
```

Exakt dasselbe Bild ergibt sich im Computerspeicher, wenn die Referenzen nicht vom selben Typ sind wie die Objekte.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT



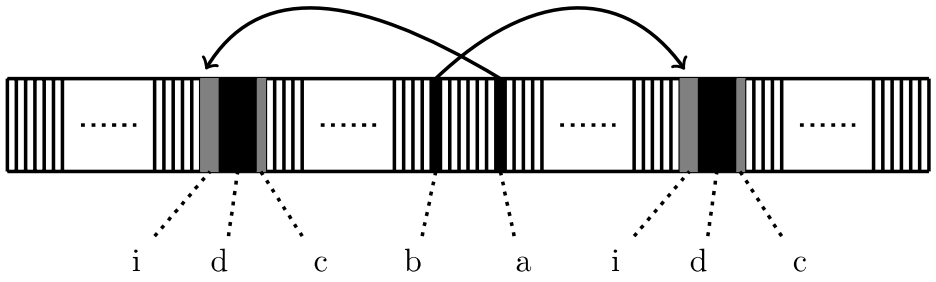

```
public class X {
   public int i;
   public double d;
   public char c;
}
```

```
X a = new X();
X b = new X();
```

Und schließlich wieder ein rein illustratives Beispiel zur Gegenüberstellung.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class X {
   public int i;
   public double d;
   public char c;
}
```

```
X a = new X();
X b = new X();
```

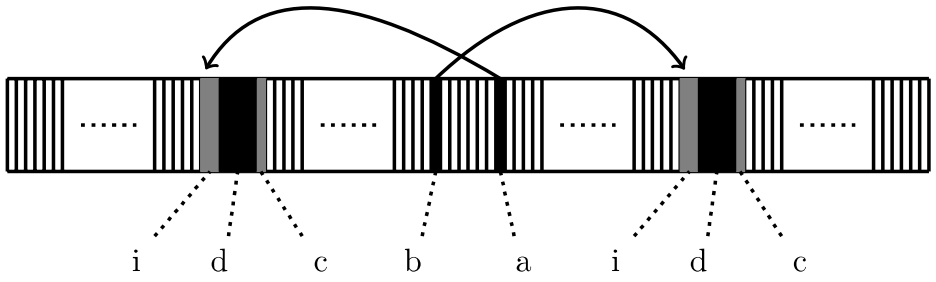
```
a.d = 2.71;
b.d = 3.14;
```

Jedes einzelne Attribut, i, d und c, bekommt bei der Übersetzung der Klasse X einen für alle Objekte der Klasse X gleichen, festen Abstand, englisch Offset, zur Anfangsadresse des Objekts, natürlich so, dass die Speicherbereiche der drei Attribute sich nicht überlappen.

Wenn im Java-Quelltext über eine Variable wie a nun auf ein Attribut von X wie etwa d zugegriffen wird, also a.d, dann wird dies vom Compiler in Code übersetzt, der die in a momentan gespeicherte Adresse hernimmt, den Offset für d draufaddiert und dann auf die Daten an dieser Adresse zugreift. Analog natürlich bei der Variablen b.

Zwei Objekte und zwei Variable

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class X {
   public int i;
   public double d;
   public char c;
}
```

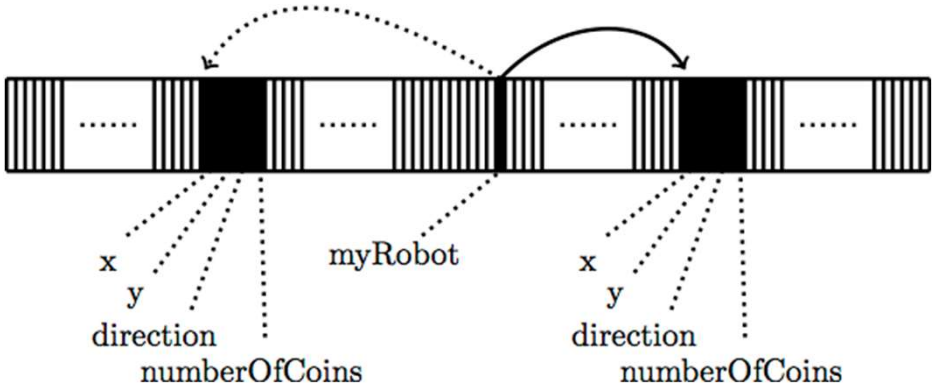
```
X a = new X();
X b = new X();
```

```
b.d = a.d;
```

Selbstverständlich sind auch Zuweisungen vom Attribut des einen Objektes zum Attribut des anderen Objektes möglich. In diesem kleinen Beispiel wird auf das Attribut d von a lesend und auf das Attribut d von b schreibend zugegriffen.

Referenz „verbiegen“

TECHNISCHE UNIVERSITÄT DARMSTADT

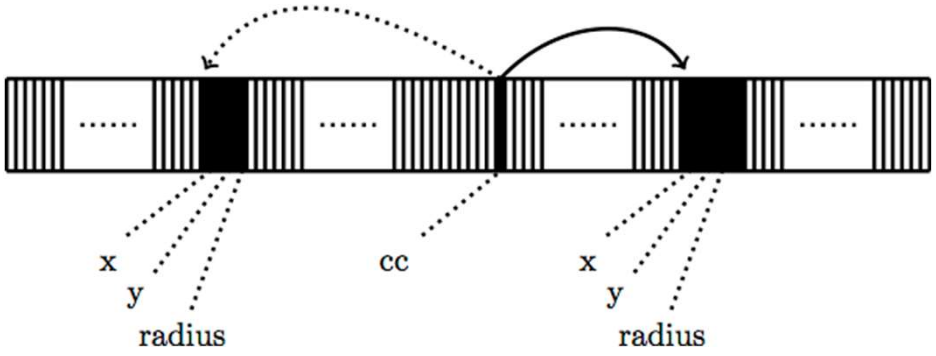



```
Robot myRobot = new Robot ( ..... );
myRobot = new Robot ( ..... );
```

Wenn zweimal ein Objekt erzeugt und beide Male die Adresse derselben Referenz zugewiesen wird, dann sieht die Situation so wie hier gezeigt aus: Der gestrichelte Pfeil deutet auf das erste erzeugte Objekt, auf das myRobot nach der ersten Zeile unten verwies; der durchgezogene Pfeil gibt die Situation nach der zweiten Zeile unten wieder. Das erste Objekt ist vom Programm aus nicht mehr ansprechbar, da es keine Referenz mehr darauf gibt. Etwaige wichtige Daten, die nur im ersten Objekt und sonst nirgendwo sind, sind also verloren.

Referenz „verbiegen“

TECHNISCHE UNIVERSITÄT DARMSTADT

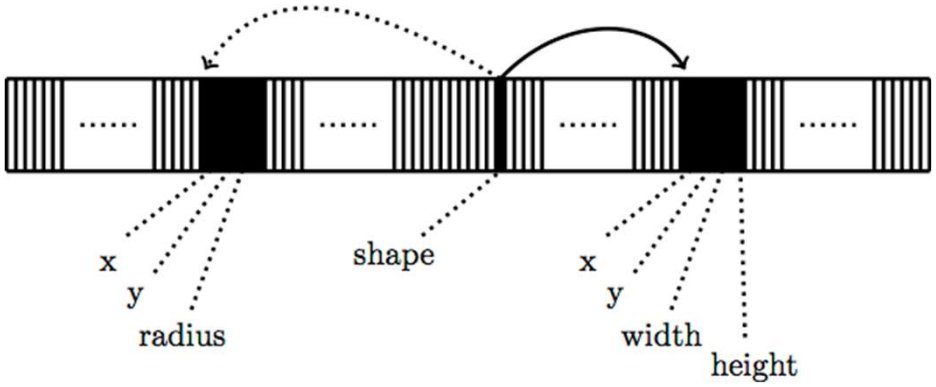



```
Circle cc = new Circle ( ..... );
cc = new Circle ( ..... );
```

Natürlich sieht das bei anderen Klassen genauso aus.

Referenz „verbiegen“

TECHNISCHE UNIVERSITÄT DARMSTADT

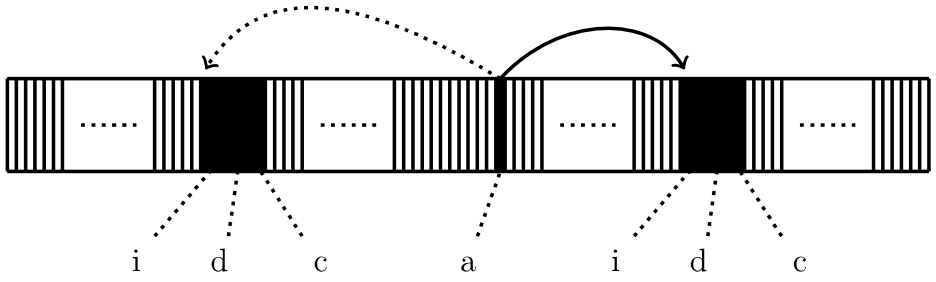



```
GeomShape2D shape = new Circle ( ..... );
shape = new Rectangle ( ..... );
```

Und auch bei unterschiedlichen Typen der Objekte sieht die Situation im Prinzip gleich aus.

Referenz „verbiegen“

TECHNISCHE UNIVERSITÄT DARMSTADT



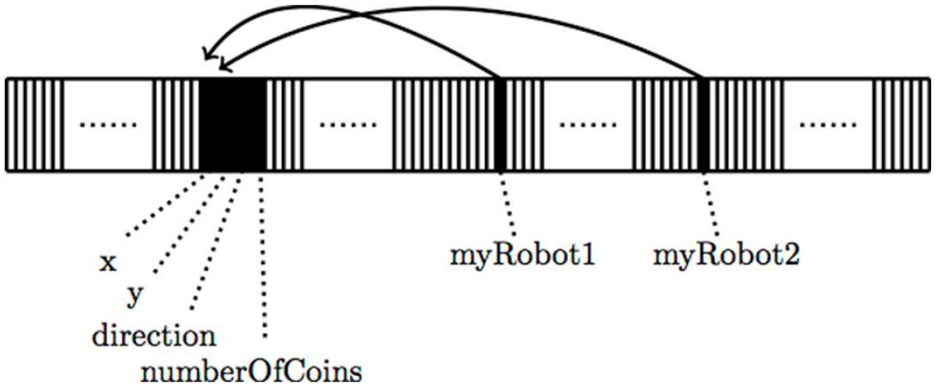

```
public class X {
  public int i;
  public double d;
  public char c;
}
```

```
X a = new X();
a = new X();
```

Schließlich wieder dasselbe Schema anhand einer illustrativen Klasse.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT

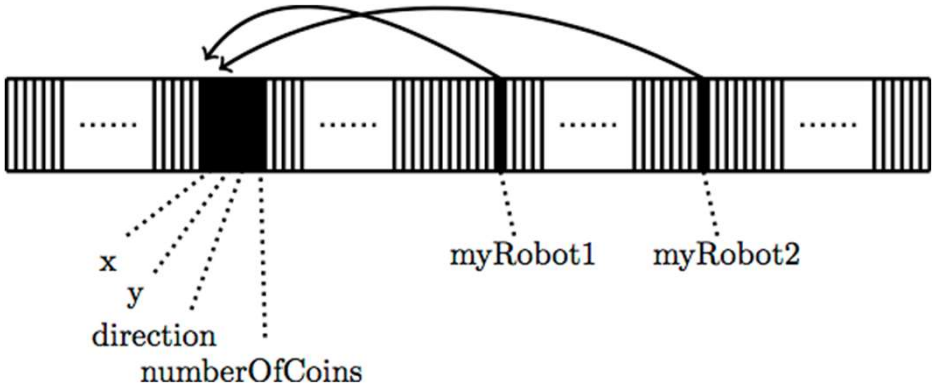



```
Robot myRobot1 = new Robot ( ..... );
Robot myRobot2 = myRobot1;
```

So sieht dann die Situation aus, wenn man nur ein Objekt erzeugt, dessen Adresse dann aber einer zweiten Referenz zugewiesen wird. Jetzt verweisen eben beide Referenzen auf dasselbe Objekt.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT

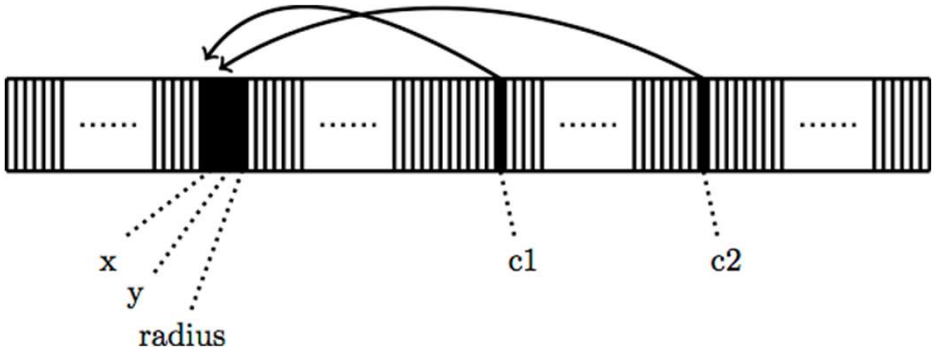



```
myRobot1.move();        // <=> myRobot2.move();
myRobot2.turnLeft();    // <=> myRobot1.turnLeft();
```

Da beide Referenzen auf dasselbe Objekt verweisen, ist es natürlich egal, welche der beiden Referenzen man verwendet, um etwas mit dem Objekt zu machen, also um lesend oder schreibend auf ein Attribut zuzugreifen oder um eine Methode aufzurufen.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT

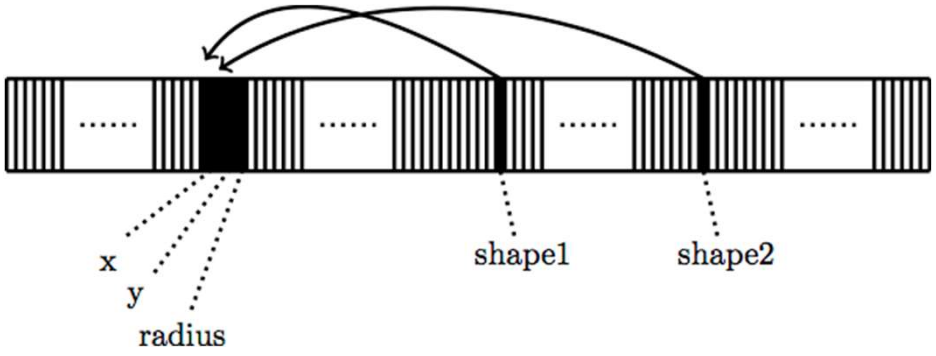



```
Circle c1 = new Circle ( ..... );
Circle c2 = c1;
```

Natürlich ist die Situation analog für jede andere Klasse, so etwa auch wieder für die Klassen aus dem geometrischen Fallbeispiel.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT

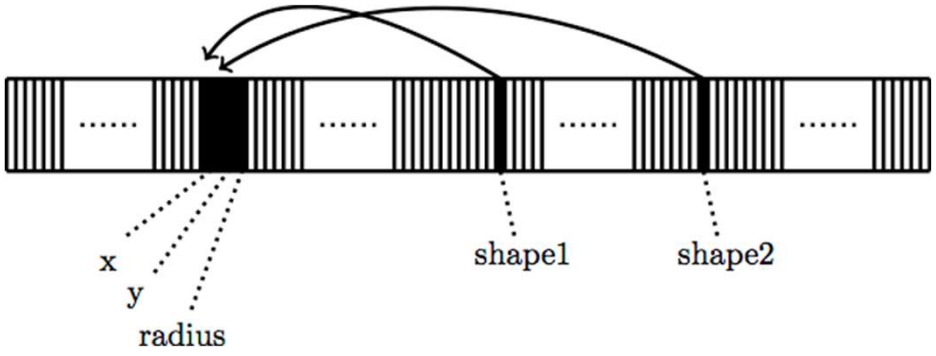



```
GeomShape2D shape1 = new Circle ( ..... );
GeomShape2D shape2 = shape1;
```

Auch hier wieder derselbe Fall, aber mit unterschiedlichen Klassen von Referenz und Objekt.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT

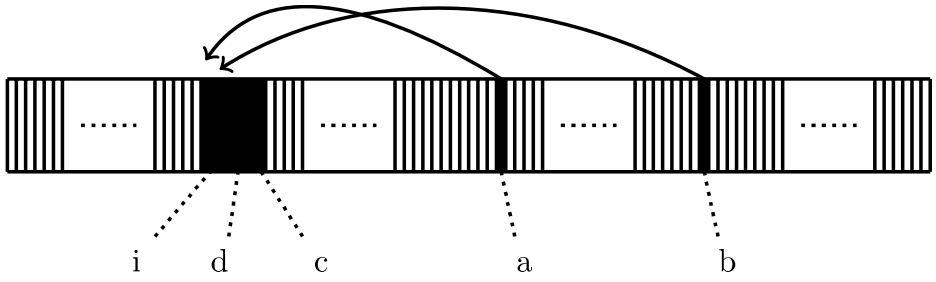



```
shape1.scale ( 3 );     //  <=> shape2.scale ( 3 );
shape2.paint();         //  <=> shape1.paint();
```

Auch hier ist es wieder völlig egal, über welche der beiden Referenzen auf das Objekt zugegriffen wird.

Zwei Referenzen auf ein Objekt

TECHNISCHE UNIVERSITÄT DARMSTADT



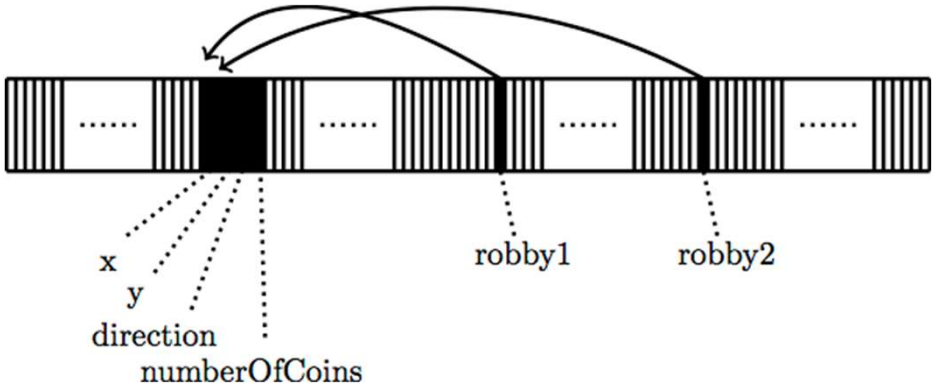

```
public class X {
  public int i;
  public double d;
  public char c;
}
```

```
X a = new X();
X b = a;
a.d = 3.14;    //  <=> b.d = 3.14;
int n = b.i;   //  <=> int n = a.i;
```

Auch beim direkten lesenden und schreibenden Zugriff auf Attribute ist es natürlich egal, über welche Referenz diese Zugriffe jeweils erfolgen, wenn beide Referenzen auf dasselbe Objekt verweisen.

Test auf Gleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT

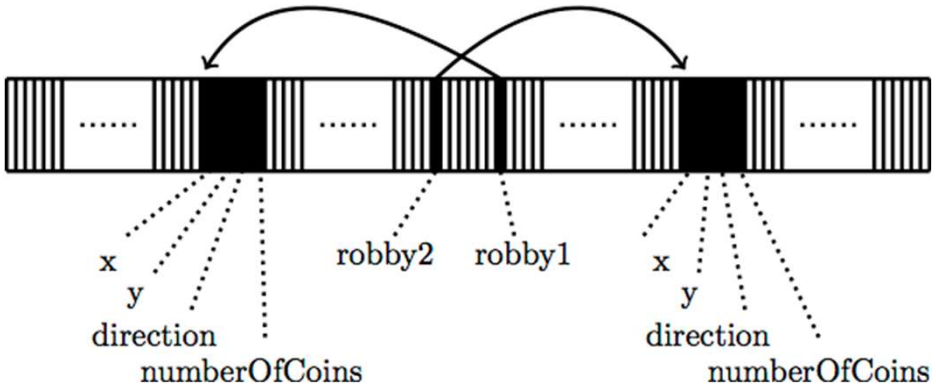



```
Robot robby1 = new Robot ( ..... );        if ( robby1 == robby2 )
Robot robby2 = robby1;                          // => true
```

Was kommt beim Test auf Gleichheit heraus, wenn wir den Gleichheitsoperator auf zwei Referenzen anwenden? Im Prinzip gäbe es ja zwei Möglichkeiten: Entweder werden die beiden Adressen miteinander verglichen oder statt dessen die Inhalte der beiden Objekte. In beiden Fällen käme in diesem kleinen Beispiel true heraus, daher ist dieses Beispiel noch nicht aussagekräftig für unsere Frage.

Test auf Gleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT

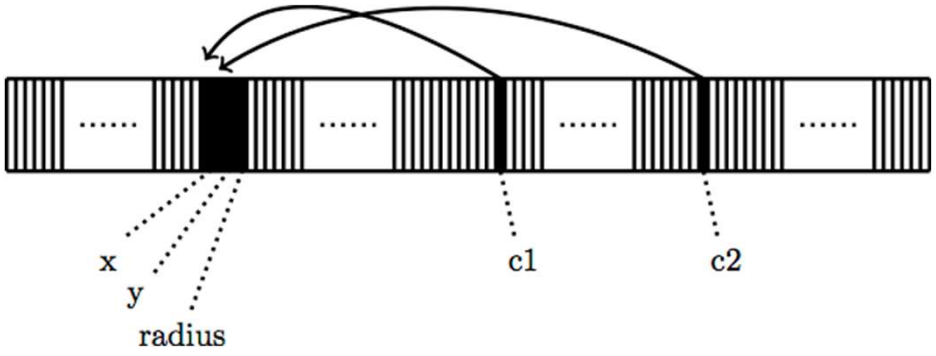



```
Robot robby1 = new Robot ( 1, 2, UP, 0 );     if ( robby1 == robby2 )
Robot robby2 = new Robot ( 1, 2, UP, 0 );       // => false
```

Aber wenn wir das Beispiel ein wenig modifizieren, wird die Antwort klar: Der Gleichheitsoperator vergleicht die Adressen, also die Inhalte der *Referenzen*, *nicht* die Inhalte der *Objekte*. Denn in dieser Modifikation haben die beiden Objekte in allen Attributen jeweils denselben Wert, aber die Adressen sind natürlich unterschiedlich, daher kommt false heraus.

Test auf Gleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT

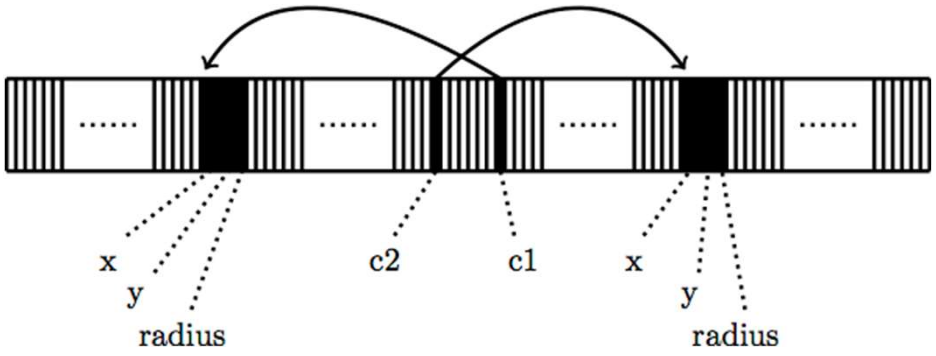



```
Circle c1 = new Circle ( ..... );          if ( c1 == c2 )
Circle c2 = c1;                             //  true
```

Bei anderen Referenztypen sieht das natürlich völlig identisch aus, wieder zum Beispiel geometrische Klassen.

Test auf Gleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT




```
Circle c1 = new Circle ( 1, 2, 3 );
Circle c2 = new Circle ( 1, 2, 3 );
```

```
if ( c1 == c2 )
  //  false
```

Auch hier zeigt wieder die kleine Modifikation des Beispiels, dass die *Adressen* der Objekte, nicht ihre *Inhalte* auf Gleichheit getestet werden.

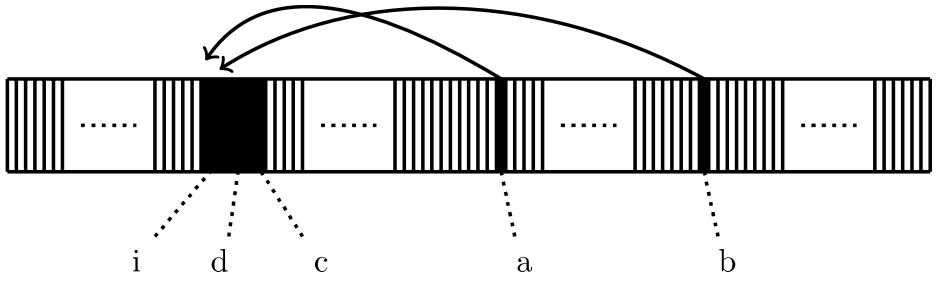
TECHNISCHE UNIVERSITÄT DARMSTADT

Begriffsbildung:
Objektidentität
vs.
Wertgleichheit

Für den Unterschied zwischen dem Fall, dass zwei Referenzen auf dasselbe Objekt verweisen, und dem Fall, dass sie auf verschiedene Objekte mit denselben Attributwerten verweisen, gibt es eine grundlegende Begriffsbildung in der Informatik.

Objektidentität vs. Wertgleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT



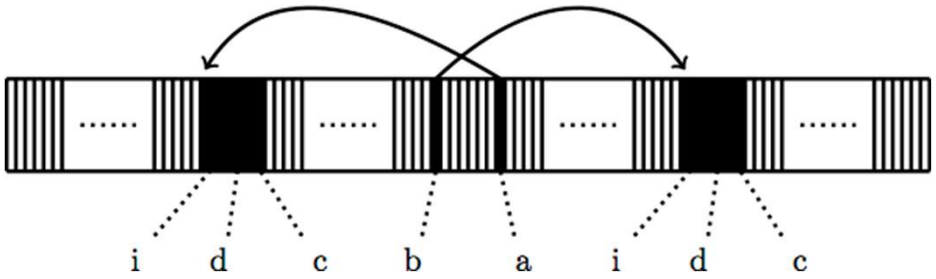

```
X a = new X();
X b = a;
```

***Objektidentität*: a und b verweisen auf dasselbe Objekt**

Dieser Fall, den wir in diesem Kapitel schon öfters in verschiedensten Variationen gesehen haben, ist der Fall von Objektidentität.

Objektidentität vs. Wertgleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT




```
X a = new X();
X b = new X();
.........
```

```
b.i = a.i;
a.d = b.d;
b.c = a.c;
```

Wertgleichheit: a und b verweisen auf zwei Objekte mit demselben Inhalt

In diesem Beispiel hingegen haben wir nach Ausführung der drei Anweisungen links zwei wertgleiche, aber nicht identische Objekte.

Die Frage, ob Objektidentität dann auch Wertgleichheit beinhaltet oder nicht, also ob es bei Wertgleichheit wirklich *zwei* Objekte sein müssen, ist spitzfindig, aber zur Vermeidung von Missverständnissen sicher berechtigt. Eine Normierung der Antwort ist dem Autor dieser Folien nicht bekannt. Wann immer man von Wertgleichheit spricht, sollte man sicherheitshalber erwähnen, ob zwei Verweise auf dasselbe Objekt mitgemeint sind oder nicht.

Deep (<-> Shallow) Copy

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int i;
   public double d;
   public char c;
}

public class Y {
   public X x;
}

public class Z {
   public X x;
   public Y y;
}
```

```
Z z1 = new Z();

z1.x = new X();
z1.x.i = 1;
z1.x.d = 3.14;
z1.x.c = ´a´;

z1.y = new Y();
z1.y.x = new X();
z1.y.x.i = 2;
z1.,y.x.d = 2.71;
z1.y.x.c = ´b´;
```

```
Z z2 = new Z();

z2.x = new X();
z2.x.i = z1.x.i;
z2.x.d = z1.x.d;
z2.x.c =  z1.x.c;

z2.y = new Y();
z2.y.x = new X();
z2.y.x.i = z1.y.x.i;
z2.,y.x.d = z1.y.x.d;
z2.y.x.c = z1.y.x.c;
```

So wie wir Wertgleichheit auf der vorangegangenen Folie betrachtet hatten, ist das aber nur die halbe Wahrheit. Wenn ein Objekt Attribute von Referenztypen hat, dann müssen diese entweder beide den symbolischen Wert null haben, oder beide müssen wiederum auf wertgleiche Objekte verweisen.

Im Beispiel auf dieser Folie gehen wir sogar noch einen Schritt weiter: Ein Objekt verweist auf ein zweites und dieses wiederum auf ein drittes. An allen parallelen Stellen muss Wertgleichheit bestehen, damit z1 und z2 als wertgleich angesehen werden können.

Was Sie auf dieser Folie sehen, nennt man ein *deep copy* von z1 nach z2 im Gegensatz zu einer *shallow copy*, bei der nur die Adressen in z.x und z.y kopiert werden. In der Regel wird man eine Deep Copy haben wollen, also zwei Gebilde, die wirklich völlig separat voneinander, aber auch in der Tiefe wertgleich sind.

TECHNISCHE UNIVERSITÄT DARMSTADT

Vererbung von Attributen

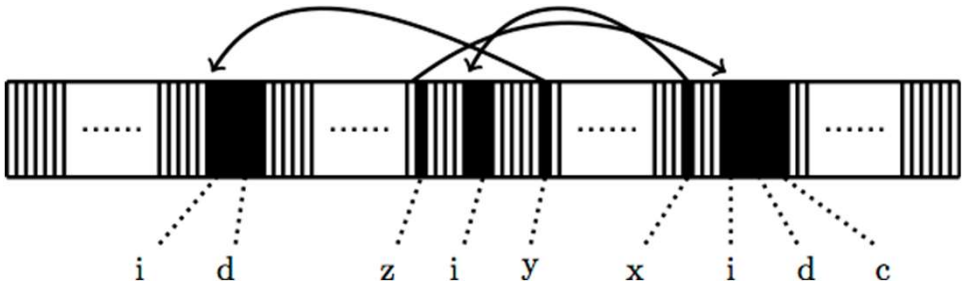
Oracle Java Tutorials: inheritance

Wenn eine Klasse von einer anderen abgeleitet wird, erbt sie ja auch deren Attribute. Diese sind dann in jedem Objekt der abgeleiteten Klasse zusätzlich vorhanden.

***Erinnerung*: Klassen wie SymmTurner, PacmanRobot und so weiter hatten die Attribute von Robot geerbt.**

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class X { public int i; }
public class Y extends X { public double d; }
public class Z extends Y { public char c; }
```

```
X x = new X();
Y y = new Y();
Z z = new Z();
```

So kann man sich das schematisch in einem rein illustrativen Beispiel vorstellen. Aus Platzgründen sind die drei Klassen jeweils als Ganzes in einer Zeile definiert, was natürlich keine gute Praxis ist.

Das Objekt von Klasse X hat nur das Attribut i, das Objekt von Klasse Y hat i und d, und das Objekt von Klasse Z hat i, d und c.

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int i;
  protected double d;
  private char c;
  public void m1() {
    i = 1;
    d = 3.14;
    c = ´z´;
  }
}
```

```
public class Y {
  public void m2() {
    X a = new X();
    a.i = 1;
    a.d = 3.14;
    a.c = ´z´;
  }
}
```

Wir haben schon hin und wieder public und private bei Attributen von Klassen gesehen: public bedeutet, dass man überall im Java-Quelltext, also auch in anderen Klassen beliebig auf das Attribut zugreifen kann. Bei private hingegen kann außerhalb der Klasse überhaupt nicht zugegriffen werden.

In Kapitel 01f, Abschnitt „Zugriffsrechte und Packages“, hatten wir auch schon eine dritte Variante namens protected eingeführt.

Was wir hier zur Vereinfachung nicht noch einmal betrachten, ist die vierte Variante, in der kein Schlüsselwort an dieser Stelle steht, was ja bedeutet, dass das Attribut im gesamten Package, aber nirgendwo sonst sichtbar ist. Das ist wie private, nur nicht bezogen auf die eigene Klasse, sondern auf das eigene Package.

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int i;
  protected double d;
  private char c;
  public void m1() {
    i = 1;
    d = 3.14;
    c = ´z´;
  }
}
```

```
public class Y {
   public void m2() {
      X a = new X();
      a.i = 1;
      a.d = 3.14;
      a.c = ´z´;
   }
}
```

Innerhalb der Klasse X selbst kann auf jedes Attribut zugegriffen werden, egal ob public, protected oder private.

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int i;
  protected double d;
  private char c;
  public void m1() {
    i = 1;
    d = 3.14;
    c = ´z´;
  }
}
```

```
public class Y {
  public void m2() {
    X a = new X();
    a.i = 1;
    a.d = 3.14;
    a.c = ´z´;
  }
}
```

Hier sehen Sie, was beim Zugriff aus einer anderen Klasse Y, die mit X nichts zu tun hat, möglich ist und was nicht. Auf ein Attribut, das public deklariert ist, kann auch hier zugegriffen werden. Bei Zugriff auf ein Attribut, das protected oder private ist, wird der Compiler hingegen eine Fehlermeldung ausgeben und den Quelltext nicht übersetzen.

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int i;
  protected double d;
  private char c;
  public void m1() {
    i = 1;
    d = 3.14;
    c = ´z´;
  }
}
```

```
public class Y extends X {
   public void m2() {
      X a = new X();
      a.i = 1;
      a.d = 3.14;
      a.c = ´z´;
   }
}
```

Anders verhält es sich, wenn die Klasse Y von X abgeleitet ist. Dabei ist es egal, ob Y so wie hier *direkt* oder statt dessen *indirekt* – also über weitere Zwischenklassen – von X abgeleitet ist.

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int i;
  protected double d;
  private char c;
  public void m1() {
    i = 1;
    d = 3.14;
    c = ´z´;
  }
}
```

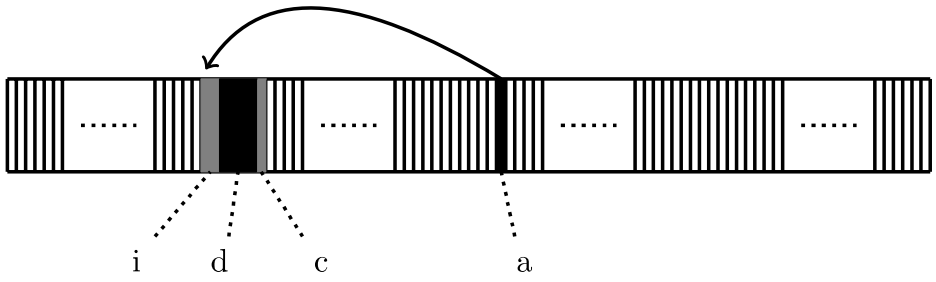
```
public class Y extends X {
  public void m2() {
    X a = new X();
    a.i = 1;
    a.d = 3.14;
    a.c = ´z´;
  }
}
```

Der Unterschied ist, dass auf Attribute, die protected deklariert sind, in einer abgeleiteten Klasse ebenfalls zugegriffen werden kann. Genau das ist der Sinn von protected.

***Erinnerung*: Bei protected haben zusätzlich auch alle Klassen im selben Package Zugriff.**

Vererbung von Attributen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class X {
  public int i;
  public double d;
}
```

```
public class Y extends X {
   public char c;
}
```

```
X a = new Y();
```

Sie sehen, dass sich das Bild im Speicher überhaupt nicht geändert hat. Ob Variable und Objekt vom selben Typ oder von unterschiedlichen Typen sind, ist auf dieser Betrachtungsebene egal. Das ist der tiefere Grund dafür, warum Objekt und Variable zwei verschiedene Entitäten sind, denn sonst wäre das nicht möglich, dass sich ein Y hinter einem X „verbirgt“.

***Vorgriff*: Später in diesem Kapitel, im Abschnitt zum statischen und dynamischen Typ, werden wir sehen, dass es *doch* einen Unterschied gibt, nämlich dass eine Variable der Klasse X nur Zugriff auf die Attribute und Methoden ermöglicht, die schon in X definiert sind; das heißt, *die* Attribute und Methoden, die erst in Y erstmals definiert werden, können über eine Variable der Klasse X *nicht* angesprochen werden. im konkreten Beispiel kann das Attribut c nicht über die Variable a angesprochen werden.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Klasse java.lang.String

Oracle Java Tutorials: Strings

Wir kommen zu einer einzelnen Klasse, die besonders wichtig ist und wahrscheinlich in so ziemlich jedem ernsthaften Java-Programm in der Welt verwendet wird. Da sie so wichtig ist, findet sie sich in dem Package, von dem Sie schon aus Kapitel 03a wissen, das es als einziges vom Compiler automatisch importiert wird, also auch wenn man es nicht explizit mit Schlüsselwort import selbst importiert.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

Erinnerung Zeichenliterale (char):

‘a‘ ‘A‘ ‘9‘ ‘!‘

‘ \n‘ ‘ \b‘ ‘ \t‘

‘ \‘‘ ‘ \“‘ ‘ \ \‘

‘ \u03A9‘

Erinnerung **aus Kapitel 01b, Abschnitt „Allgemein: Primitive Datentypen“: So sehen Literale vom primitiven Datentyp char aus.**

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

String-Literale:

“Hello, World!“

“\‘Hell\u03A9,\tWorld\\\‘“

➔ ‘HellΩ, World\‘

String hingegen ist kein primitiver Datentyp, sondern eine vordefinierte Klasse, aber dennoch sind Literale vom Typ String in Java eingebaut. Klasse String nimmt in Java daher eine Sonderrolle ein – keine Klasse wie jede andere.

Strings sind in der Informatik generell Zeichenketten. String-Literale sind in Java daher Sequenzen von char-Literalen, nur dass nicht mehr jedes einzelne char-Literal in einzelne Hochkommas gesetzt wird, sondern der ganze String in doppelte Hochkommas.

Auf dieser Folie haben wir zwei Beispiele für String-Literale, ein einfaches und ein etwas komplizierteres. Unten nach dem Pfeil sehen Sie, wie das zweite String-Literal aussehen würde, wenn man es beispielsweise auf dem Bildschirm ausgeben würde.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

String-Literale:

“Hello, World!“

“\‘Hell\u03A9,\tWorld\\\“

➔ ‘HellΩ, World\‘

Strings können tatsächlich alles enthalten, was auch char-Literal sein kann, zum Beispiel einzelne Hochkommas, natürlich mit Backslash davor, wie auf der letzten Folie bei char-Literalen noch einmal kurz gesehen.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

String-Literale:

“Hello, World!“

“\‘Hell\u03A9,\tWorld\\\‘“

➔ ‘HellΩ, World\‘

Genauso können String-Literale auch Backslashes enthalten, wie eben nochmals gesehen als doppelte Backslashes geschrieben.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

String-Literale:

“Hello, World!“

“\‘Hell\u03A9,\tWorld\\\‘“

➔ ‘HellΩ, World\‘

Auch innerhalb eines String-Literals können Zeichen durch ihre Unicode-Nummer angesprochen werden. Da jede Unicode-Nummer aus exakt sechs Zeichen besteht, ist immer eindeutig, was noch zur Unicode-Nummer gehört und was danach nicht mehr dazu gehört.

Jetzt sehen wir, für welches Zeichen die Unicode-Nummer steht, die wir eben schon bei Zeichenliteralen als Beispiel hatten.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

String-Literale:

“Hello, World!“

“\‘Hell\u03A9,\tWorld\\\‘“

➔ ‘HellΩ, World\‘

Hier sehen Sie den Tabulatorschritt. Verschiedene Ausgabeprogramme stellen den Tabulatorschritt unterschiedlich weit dar beziehungsweise das ist auch oft konfigurierbar.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

Char- vs. String-Literale:

‘A‘ “A“

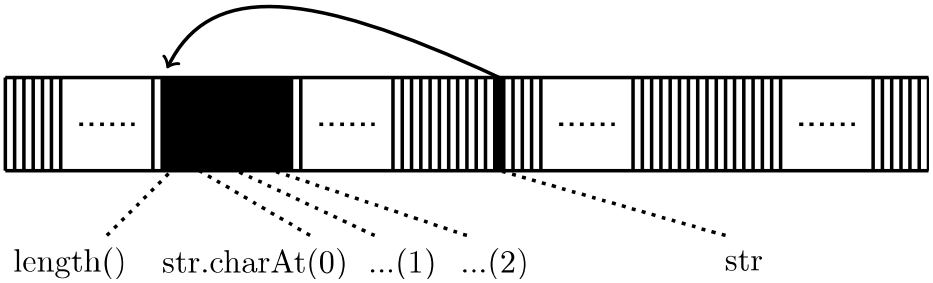
““

Es ist wichtig, zwischen einem char-Literal und einem String-Literal der Länge 1 zu unterscheiden. Links sehen Sie ein char-Literal, rechts sehen Sie ein String-Literal, das nur ein Zeichen umfasst, also Länge 1 hat.

Auch leere Strings mit genau null Zeichen, also mit Länge 0, sind möglich. Das sieht dann so aus wie unten.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT



String str = new String (“aZ1“);

Wie der Name schon sagt, enthält ein Objekt der Klasse java.lang.String eben einen String, also eine Zeichenkette. Wir können uns Strings als eine Form von Arrays von Zeichen vorstellen, und tatsächlich sind Strings mit Sicherheit intern analog zu Array von char realisiert. Aber Arrays sind eingebaut in die Sprache Java, String ist hingegen eine zwar in java.lang vordefinierte, aber ansonsten ganz normale Klasse, so dass alle Operationen auf Strings durch Methoden realisiert sein müssen.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );

int n = str.length();                    //    n == 11

char c = str.charAt ( 6 );               //    c == ´W´

int m = str.indexOf ( ´o´ );             //    m == 4

boolean b =
    str.matches ( “He.+rld“ );           //    b == true
```

Wir schauen uns nur ein paar ausgewählte Methoden der Klasse String beispielhaft an.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );
int n = str.length();                    //  n == 11
char c = str.charAt ( 6 );               //  c == ´W´
int m = str.indexOf ( ´o´ );             //  m == 4
boolean b =
    str.matches ( “He.+rld“ );           //  b == true
```

Eine Methode namens length für die Länge des Strings, die keine Parameter hat und int zurückliefert.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );

int n = str.length();                    //    n == 11

char c = str.charAt ( 6 );               //    c == ´W´

int m = str.indexOf ( ´o´ );             //    m == 4

boolean b =
    str.matches ( “He.+rld“ );           //    b == true
```

Eine Methode namens charAt, um auf das Element an einem gegebenen Index zuzugreifen. Wie bei Arrays, ist auch bei Strings der erste Index gleich 0, so dass ein String der Länge 11 einen Indexbereich von 0 bis 10 hat.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );
int n = str.length();                    //   n == 11
char c = str.charAt ( 6 );               //   c == ´W´
int m = str.indexOf ( ´o´ );             //   m == 4
boolean b =
    str.matches ( “He.+rld“ );           //   b == true
```

Die Klasse String bietet einige komfortable Möglichkeiten, um mit Zeichenketten umzugehen. Wir zeigen hier nur beispielhaft zwei davon. Die erste ist eine Methode, die für ein gegebenes Zeichen den ersten Index zurückliefert, an dem dieses Zeichen im String vorkommt. Das Zeichen klein-o kommt an den Indizes 4 und 7 vor, Rückgabewert ist also 4.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );
int n = str.length();                    //   n == 11
char c = str.charAt ( 6 );               //   c == ´W´
int m = str.indexOf ( ´o´ );             //   m == 4
boolean b =
    str.matches ( “He.+rld“ );           //   b == true
```

Diese **Methode bekommt als Parameter einen sogenannten regulären Ausdruck, englisch Regular Expression. Ein Regular Expression passt auf einer endlich oder je nachdem sogar unendlich großen Menge von unterschiedlichen Zeichenketten. Die Methode matches liefert genau dann true zurück, wenn die Zeichenkette, auf die str verweist, zu dieser Menge gehört.**

Aber Achtung**: Verschiedene Programme und Programmiersprachen verwenden Regular Expressions in leicht unterschiedlicher Weise, so dass das hier Gesagte auf die Methode matches der Klasse String in Java zutrifft, aber keine Garantie dafür, dass das auch exakt so aussieht, wenn Ihnen der Begriff Regular Expression in einem anderen Kontext begegnet.**

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str = new String ( “Hello World“ );

int n = str.length();                    //    n == 11

char c = str.charAt ( 6 );               //    c == ´W´

int m = str.indexOf ( ´o´ );             //    m == 4

boolean b =
    str.matches ( “He.+rld“ );           //    b == true
```

Der Punkt meint in einem Regular Expression bei Java-Strings nicht das Interpunktionszeichen, sondern steht als Platzhalter für ein beliebiges Zeichen. Das Plus sagt, dass das Zeichen vor dem Plus ein oder mehrmals vorkommen darf. Das Zeichen vor dem Plus ist aber durch den Punkt beliebig, das heißt, dieser Regular Expression passt auf alle Zeichenketten, die mit groß-H und klein-e beginnen, mit r-l-d enden und mindestens ein Zeichen dazwischen haben.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Hello World“ );
String str2 = “Hello World“;

String str3 = str1.concat ( str2 );
String str4 = str1 + str2;
```

In Java sind Strings zwar als Klasse realisiert, nicht als primitiver Datentyp, aber trotzdem wird in Java die zentrale Rolle von Strings durch spezielle Sprachkonstrukte berücksichtigt. Hier die beiden wohl wichtigsten Beispiele.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Hello World“ );
String str2 = “Hello World“;

String str3 = str1.concat ( str2 );
String str4 = str1 + str2;
```

Diese beiden Ausdrücke sind absolut austauschbar, das heißt, speziell bei Klasse String gibt es eine intuitive Kurzform für die Einrichtung eines String-Objekts.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Hello World“ );
String str2 = “Hello World“;

String str3 = str1.concat ( str2 );
String str4 = str1 + str2;
```

Nun das zweite Beispiel: Die Methode concat von Klasse String liefert als String die Konkatenation aus dem String, mit dem die Methode aufgerufen wird, und dem String-Parameter. Konkret wird der Parameter an den anderen String angehängt.

Der Begriff *Konkatenation* mit zugehörigem Verb *konkatenieren* wird allgemein in der Informatik verwendet, wenn wie hier zwei Sequenzen beliebiger Art zu einer zusammengefügt werden, indem die zweite an die erste angehängt wird.

Wichtig einzusehen ist bei Methode concat von Klasse String, dass der String, auf den str3 hinterher verweist, ein separates, durch Methode concat neu eingerichtetes Objekt ist, das zwar dieselben Zeichen enthält wie die beiden Objekte, auf die str1 und str2 verweisen, aber ansonsten haben die drei Objekte nichts miteinander zu tun.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str1 = new String ( “Hello World“ );
String str2 = “Hello World“;

String str3 = str1.concat ( str2 );
String str4 = str1 + str2;
```

Auch für die spezielle, sehr häufige Situation, dass zwei Strings konkateniert werden sollen, gibt es eine intuitive Kurzform. Die farblich unterlegte Zeile ist äquivalent zum Aufruf von concat in der Zeile darüber.

Klasse String

TECHNISCHE UNIVERSITÄT DARMSTADT

```
String str5 = str1 + str2 + str3;

String str6 = str1 + 123;

String str7 = 3.14 + str2;

String str8 = “Hello “ + 1138 + ´T´ + “HX“;
```

Das funktioniert auch mit mehr als zwei Strings, und es funktioniert auch nicht nur allein mit Strings, sondern auch in Kombination mit Werten von primitiven Datentypen. Sobald auch nur ein String in der Additionskette vorkommt, wandelt der Compiler alle Werte von primitiven Datentypen in Strings um und konkateniert alle diese Strings.

Weitere Details der Klasse String schauen wir uns hier nicht an. Nach dem, was Sie in diesem Abschnitt gesehen haben, sollten alle weiteren Methoden der Klasse String selbsterklärend für Sie sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Finale Klassen

Oracle Java Tutorials:
Writing Final Classes and Methods

Dieses einfach zu verstehende Thema ist schnell mit einem einzigen kurzen Beispiel erklärt. Wir nehmen die Klasse String aus dem letzten Abschnitt.

Finale Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public final class String { ..... }
```

```
public class MyString extends String { ..... }
```

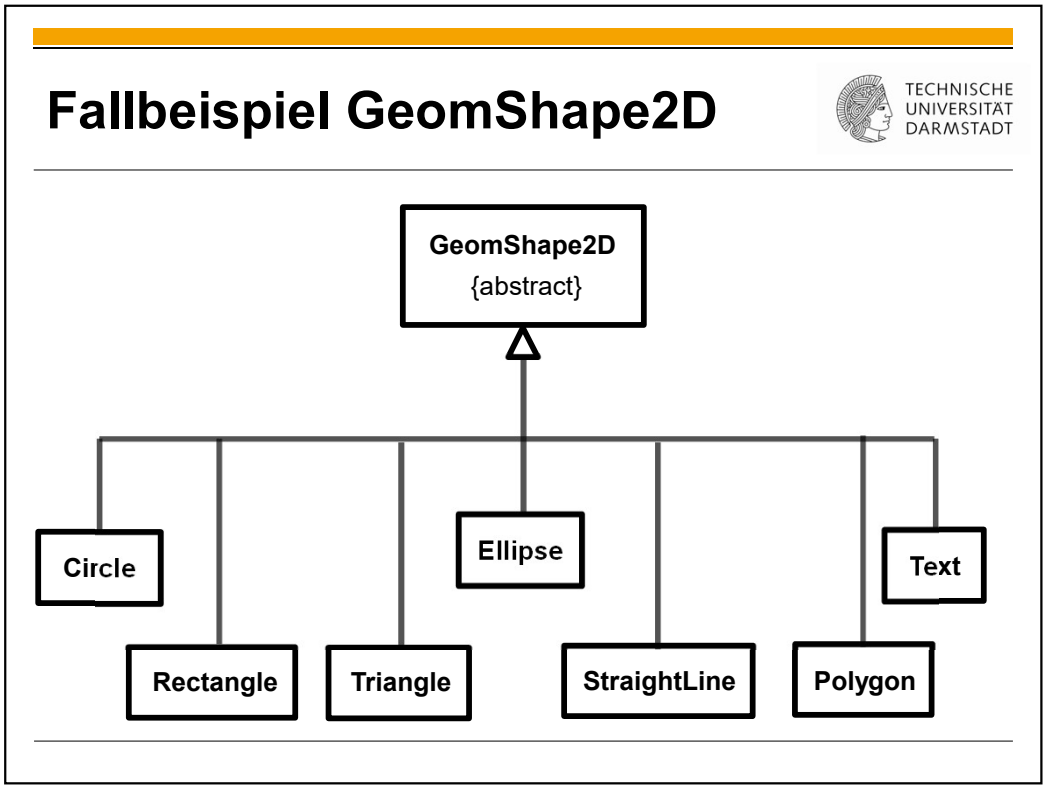
Und das ist auch schon das ganze Beispiel. Bei Design und Implementation solcher Klassen wie String kam die Erkenntnis auf, dass manche Klassen besser nicht sein sollten, weil man sich beim Versuch, eine Klasse wie hier MyString davon abzuleiten, höchstwahrscheinlich in den Besonderheiten der Basisklasse verheddern wird. Um solche Fehler von vornherein auszuschließen, kann man eine Klasse wie hier gezeigt als final deklarieren, dann geht der Versuch, eine Klasse davon abzuleiten, nicht mehr durch den Compiler.

TECHNISCHE UNIVERSITÄT DARMSTADT

Abstrakte Klassen

Oracle Java Tutorials: Abstract Methods and Classes

Bis jetzt haben wir immer nur Methoden gesehen, die implementiert waren. Es ist aber auch möglich, Methoden in einer Klasse erst einmal ohne Implementation zu definieren und erst in abgeleiteten Klassen auch zu *implementieren*.



***Erinnerung*: Klasse GeomShape2D in Kapitel 02 war abstrakt, weil die Methode paint abstrakt war.**

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

Hier ist ein einfaches, rein illustratives Beispiel mit drei Klassen, X, Y und Z.

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

So wie hier in Klasse X sieht es aus, wenn Methoden ohne Implementation definiert werden. Anstelle der Anweisungen in geschweiften Klammern steht nur ein Semikolon. Bei einer solchen Methode muss man allerdings das Schlüsselwort abstract davor schreiben, um anzugeben, dass die Methode nicht implementiert, sondern eben abstrakt ist.

Auch der Klasse selbst muss das Schlüsselwort abstract vorangestellt werden, wenn auch nur eine einzige Methode der Klasse abstrakt ist. Lässt man das abstract auch nur an einer der hier farblich markierten Stellen aus, bricht der Compiler die Übersetzung mit einer entsprechenden Fehlermeldung ab.

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

In dieser von X abgeleiteten Klasse ist die implementierte Methode m1 von X ererbt und *eine* der beiden abstrakten Methoden von X wird implementiert. Die Methode m3 ist auch in Klasse Y nicht implementiert. Damit hat auch Y qua Vererbung eine abstrakte Methode und muss als ganze Klasse ebenfalls als abstrakt deklariert werden.

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

In der Klasse Z werden die implementierten Methoden m1 und m2 von X beziehungsweise Y ererbt, und m3 wird hier ebenfalls implementiert. Damit hat Z keine unimplementierten Methoden und muss daher auch nicht abstract deklariert werden.

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

In den ersten drei Zeilen wird versucht, Objekte vom Typ X oder Y einzurichten. Das ist aber nicht erlaubt, da X und Y abstrakt sind. Wäre es erlaubt, hätten wir es plötzlich mit Objekten zu tun, für die nicht alle Methoden implementiert sind. Daher ist es sinnvoll, dass der Compiler die Übersetzung bei jeder Zeile dieser Art mit einer Fehlermeldung abbricht.

Abstrakte Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public void m1() { ..... }
   abstract public void m2();
   abstract public void m3();
}
abstract public class Y
                    extends X {
   public void m2() { ..... }
}
```

```
public class Z extends Y {
  public void m3() { ..... }
}

X a = new X();
Y b = new Y();
X c = new Y();
Z d = new Z();
Y e = new Z();
X f = new Z();
```

In diesen Deklarationen hingegen wird jeweils versucht, ein Objekt von Klasse Z einzurichten. Da die Klasse Z nicht abstrakt ist, spricht nichts dagegen, und daher akzeptiert der Compiler diese drei Zeilen klaglos und übersetzt sie.

TECHNISCHE UNIVERSITÄT DARMSTADT

Interfaces

Oracle Java Tutorials: Interfaces
(erste vier Abschnitte:
Defining, Implementing, Using, Evolving)

***Erinnerung*: Kapitel 01g, Interface PacmanStrategy nebst implementierenden Klassen PacmanStrategy1 und PacmanStrategy2.**

Was wir dort gesehen haben, ist natürlich nur ein kleiner Teil von dem, was es zu Interfaces zu sagen gibt. Wir werden Interfaces hier auch noch nicht vollständig behandeln, in Kapitel 04c sehen wir noch einiges mehr zu Interfaces.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
   int m1 ();
   void m2 ( int n );
   String m3 ( double d, char c );
}

            public class X implements Intf {
               public int m1 () { ..... }
               public void m2 ( int n ) { ..... }
               public String m3 ( double d, char c ) { ..... }
            }
```

Hier sehen wir ein rein illustratives Interface und eine dieses Interface implementierende Klasse.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
  int m1 ();
  void m2 ( int n );
  String m3 ( double d, char c );
}

        public class X implements Intf {
          public int m1 () { ..... }
          public void m2 ( int n ) { ..... }
          public String m3 ( double d, char c ) { ..... }
        }
```

Anstelle des Schlüsselwortes class steht das Schlüsselwort interface.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
  int m1 ();
  void m2 ( int n );
  String m3 ( double d, char c );
}
```

```
public class X implements Intf {
  public int m1 () { ..... }
  public void m2 ( int n ) { ..... }
  public String m3 ( double d, char c ) { ... }
}
```

Keine einzige Methode ist implementiert.

Und alle Methoden in einem Interface sind automatisch public, daher kann man das public bei den Methoden in einem Interface auch weglassen. Man dürfte es aber auch hinschreiben.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
  int m1 ();
  void m2 ( int n );
  String m3 ( double d, char c );
}

        public class X implements Intf {
          public int m1 () { ..... }
          public void m2 ( int n ) { ..... }
          public String m3 ( double d, char c ) { ..... }
        }
```

Statt extends steht hier das Schlüsselwort implements, wenn eine Klasse ein Interface implementiert.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
   int m1 ();
   void m2 ( int n );
   String m3 ( double d, char c );
}

        public class X implements Intf {
          public int m1 () { ..... }
          public void m2 ( int n ) { ..... }
          public String m3 ( double d, char c ) { ..... }
        }
```

Da die Methoden im Interface grundsätzlich public sind, müssen sie bei der Implementation ebenfalls public sein.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf {
   int m1 ();
   void m2 ( int n );
   String m3 ( double d, char c );
}

            public class X implements Intf {
               public int m1 () { ..... }
               public void m2 ( int n ) { ..... }
               public String m3 ( double d, char c ) { ..... }
            }
```

Die im Interface definierten Methoden können dann in der implementierenden Klasse auch implementiert werden. In diesem Beispiel sind alle drei Methoden des Interface in der Klasse implementiert.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

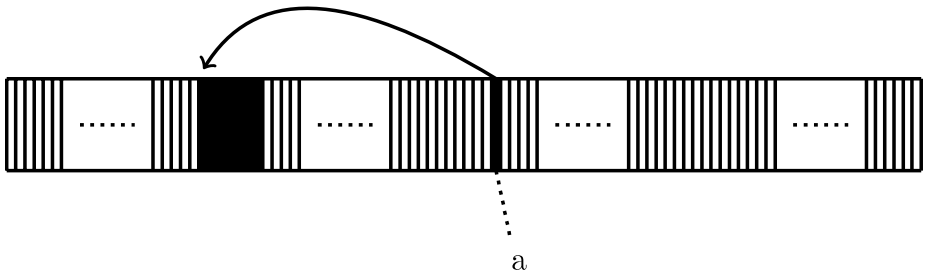
```
public interface Intf {
  int m1 ();
  void m2 ( int n );
  String m3 ( double d, char c );
}
```

```
abstract public class X implements Intf {
   public int m1 () { ..... }
   public String m3 ( double d, char c ) { ..... }
}
```

Sind eine oder mehr Methoden aus dem Interface *nicht* implementiert, sind diese Methoden und damit die ganze Klasse abstrakt. Hier sehen Sie eine kleine Variation des Beispiels von der letzten Folie, bei der die Methode m2 aus dem Interface *nicht* implementiert ist.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

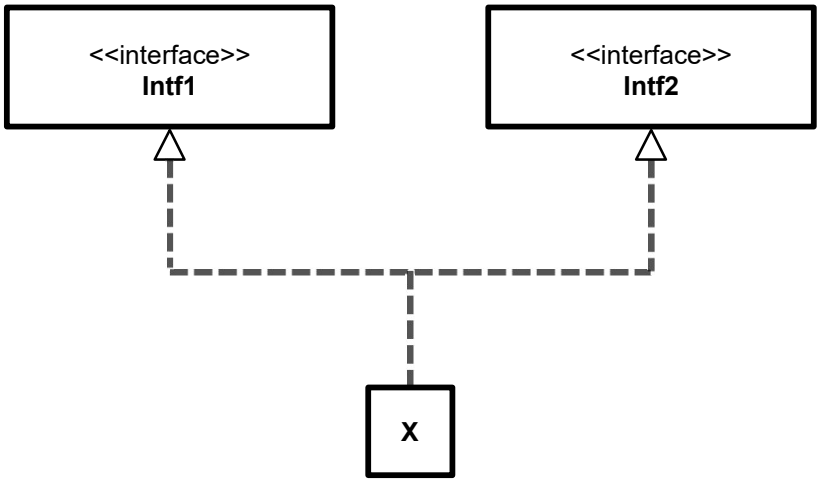



```
Intf a = new X();
int n = a.m1();
a.m2 ( 123 );
String str = a.m3 ( 3.14, ´z´);
```

Das Verhältnis zwischen einem Interface und einer das Interface implementierenden Klasse ist im Grunde völlig analog zum Verhältnis zwischen Basisklasse und abgeleiteter Klasse. Die Realisierung im Speicher ist identisch.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT



Der Unterschied ist, dass jede Klasse nur von maximal *einer* anderen Klasse abgeleitet sein, aber beliebig viele Interfaces implementieren kann.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf1 {
   int m1 ();
   void m2 ( int n );
}

public interface Intf2 {
   int m1 ();
   double m3();
}
```

```
public class X
     implements Intf1, Intf2 {
   public int m1 () { ..... }
   public void m2 ( int n ) {
      .....
   }
   public double m3 () { ..... }
}
```

Hier sehen Sie das in Java.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf1 {
   int m1 ();
   void m2 ( int n );
}

public interface Intf2 {
   int m1 ();
   double m3();
}
```

```
public class X
    implements Intf1, Intf2 {
   public int m1 () { ..... }
   public void m2 ( int n ) {
      .....
   }
   public double m3 () { ..... }
}
```

Die Klasse X rechts implementiert beide Interfaces, die links definiert sind. Die Interfaces, die eine Klasse implementiert, folgen alle auf das Schlüsselwort implements, durch Kommas voneinander getrennt.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf1 {
  int m1 ();
  void m2 ( int n );
}

public interface Intf2 {
  int m1 ();
  double m3();
}
```

```
public class X
    implements Intf1, Intf2 {
  public int m1 () { ..... }
  public void m2 ( int n ) {
    .....
  }
  public double m3 () { ..... }
}
```

Wenn Klasse X nicht abstrakt sein soll, dann müssen natürlich die Methoden aus beiden Interfaces in X implementiert werden, so wie hier die Methode m2 aus dem ersten und die Methode m3 aus dem zweiten Interface.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf1 {
  int m1 ();
  void m2 ( int n );
}

public interface Intf2 {
  int m1 ();
  double m3();
}
```

```
public class X
    implements Intf1, Intf2 {
  public int m1 () { ..... }
  public void m2 ( int n ) {
     .....
  }
  public double m3 () { ..... }
}
```

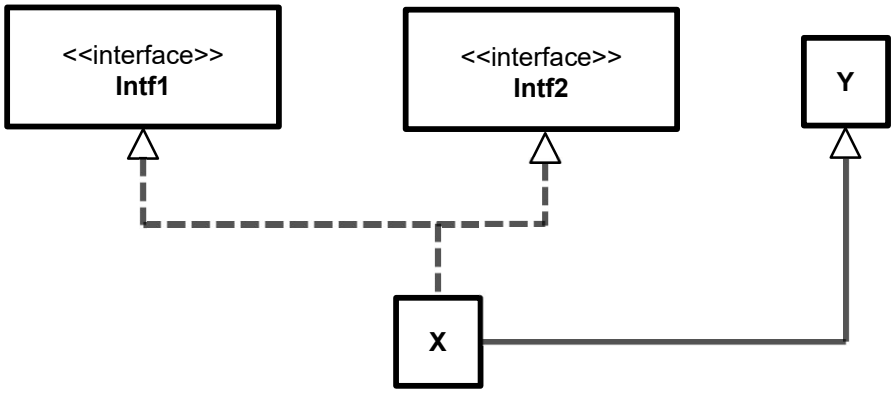
Dieselbe Methode kann auch in beiden Interfaces vorkommen und wird dann von einer einzigen Methode in der implementierenden Klasse X implementiert. Aber dann müssen die beiden Definitionen der Methode in beiden Interfaces in allen Details exakt übereinstimmen, sonst wird der Compiler mit einer Fehlermeldung abbrechen. Das wäre beispielsweise der Fall, wenn nur die Rückgabetypen unterschiedlich sind.

Wären auch die Parameterlisten der beiden Definitionen der Methode m1 in Intf1 und Intf2 unterschiedlich, dann wären es zwei verschiedene Methoden desselben Namens, und in X wäre m1 überladen, das ist dann kein Problem.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

public class X extends Y implements Intf1, Intf2



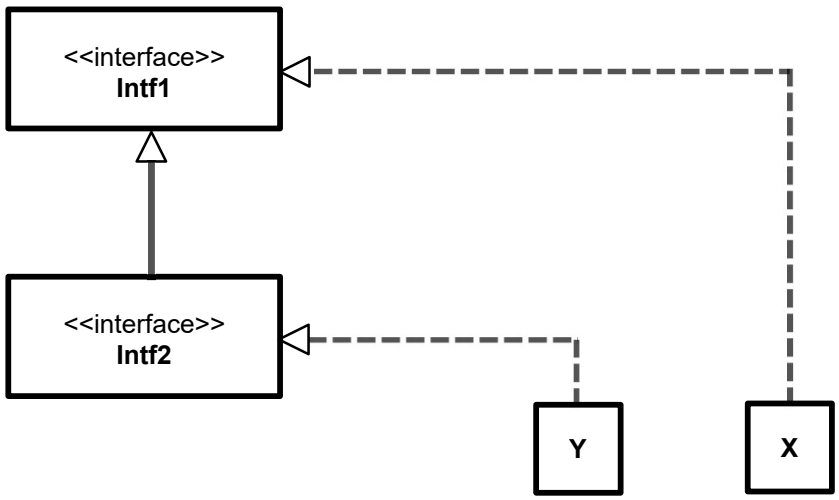
Natürlich kann eine Klasse so wie hier von einer anderen Klasse erben und zugleich beliebig viele Interfaces implementieren. Oben sehen Sie den Kopf der Klasse, unten das UML-Klassendiagramm.

***Nebenbemerkung*: Aufgrund der sehr problematischen Erfahrungen mit anderen Programmiersprachen wie C++ haben die Entwickler von Java sich von vornherein gegen Mehrfachvererbung entschieden, also gegen die Möglichkeit, eine Klasse von mehreren anderen zugleich abzuleiten. Aber man hat sich dann dafür entschieden, einen unproblematischen Fall, für den man sehr häufig Mehrfachvererbung haben möchte, doch in Form von Interfaces zu realisieren. Solange die Basisklasse – beziehungsweise in Java das Interface – keine Methodenimplementationen enthält, tauchen die Probleme nicht auf, deretwegen man sich bei der Entwicklung von Java gegen Mehrfachvererbung entschieden hat.**

Mehr zur Problematik von Mehrfachvererbung lernen Sie in Vorlesungen zu Compilerbau kennen.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT



Ein Interface kann ein anderes Interface erweitern so wie auf der linken Seite dieser Folie. Jedes dieser beiden Interfaces wird durch eine Klasse implementiert. Auf der nächsten Folie sehen wir das in Java.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Intf1 {
  void m1();
}

public interface Intf2
                extends Intf1 {
  void m2();
}
```

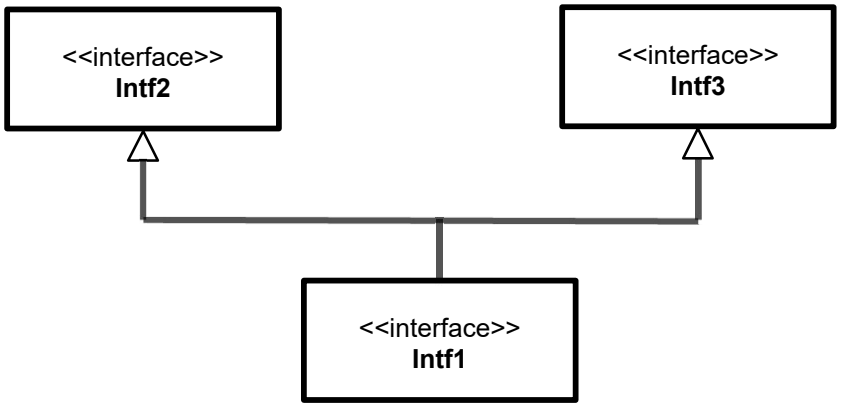
```
public class X implements Intf1 {
  public void m1() { ..... }
}

public class Y implements Intf2 {
  public void m1() { ..... }
  public void m2() { ..... }
}
```

Links sehen Sie, dass die Erweiterung bei Interfaces genauso wie bei Klassen mit „extends“ definiert wird. Bei Interface Intf1 und Klasse X sehen Sie nichts Neues. Das Interface Intf2 hingegen ist neu. Es hat zwei Methoden: m1 von Intf1 und m2. Eine Klasse wie Y, die Intf2 implementiert und nicht abstrakt sein soll, muss daher diese beiden Methoden implementieren.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

public interface Intf1 extends Intf2, Intf3



Auch mit extends auf Interfaces ist Mehrfachvererbung möglich, das heißt, ein Interface kann mehrere Interfaces erweitern.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }

public class Mammal
            extends Animal {
   .....
   int maxWalkingSpeed();
   .....
}
```

```
public class Bird
            extends Animal {
   .....
   int maxFlightHeight();
   .....
}

public class Fish
            extends Animal {
   .....
   int maxDivingDepth();
   .....
}
```

Wir schauen uns noch ein (hoffentlich) motivierendes Anwendungsbeispiel an.

So wie hier würde man vielleicht im ersten Ansatz die Hierarchie der Tierarten modellieren. Dagegen ist auch generell nichts einzuwenden, ...

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }

public class Mammal
        extends Animal {
  .....
  int maxWalkingSpeed();
  .....
}
```

```
public class Bird
        extends Animal {
  .....
  int maxFlightHeight();
  .....
}

public class Fish
        extends Animal {
  .....
  int maxDivingDepth();
  .....
}
```

... das Problem steckt aber im Detail: Wir gehen hier davon aus, dass Säugetiere laufen, Vögel fliegen und Fische schwimmen. Wie Sie wissen, stimmt das aber hinten und vorne nicht. Ein Klassendesign wie auf dieser Folie ist daher nicht tragfähig für reale Anwendungen.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Walking {
   .........
   int maxWalkingSpeed();
   .........
}

public interface Flying {
   .........
   int maxFlightHeight();
   .........
}
```

```
public interface Swimming {
   .........
   int maxDivingDepth();
   .........
}

public interface Amphibian
   extends Walking, Swimming
   { }
```

Ein tragfähiges Design erhält man, wenn man für die verschiedenen Fortbewegungsarten jeweils ein Interface definiert, in dem die Methoden – natürlich noch ohne Implementation – zusammengefasst sind, die für die drei Fortbewegungsarten relevant sind, hier jeweils nur eine Methode ausformuliert.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface Walking {
   .....
   int maxWalkingSpeed();
   .....
}
```

```
public interface Swimming {
   .....
   int maxDivingDepth();
   .....
}
```

```
public interface Flying {
   .....
   int maxFlightHeight();
   .....
}
```

```
public interface Amphibian
   extends Walking, Swimming
   { }
```

Grob gesprochen ist *Amphibie* ein Sammelbegriff für alle Tierarten, die laufen und schwimmen können. Daher liegt auch der Begriff Amphibie quer zur Hierarchie der Arten und wird ebenfalls in ein Interface ausgelagert. Hier können wir uns die Mehrfachvererbung zunutze machen, um Amphibien explizit als das zu definieren, was sie sind: Läufer, die zugleich schwimmen. Da das schon das vollständige Wesen von Amphibien ist, gibt es im Rumpf nichts Zusätzliches zu definieren, der Rumpf bleibt hier also leer.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }

public class Mammal extends Animal { ..... }     //  no walking
public class Bird extends Animal { ..... }       //  no flying
public class Fish extends Animal { ..... }       //  no swimming

public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
public class Whale extends Mammal implements Swimming { ..... }
public class Frog1 extends Animal implements Amphibian { ..... }
public class Frog2 extends Animal implements Walking, Swimming { ..... }
public class Drone implements Flying { ..... }
```

So etwa könnte dann ein tragfähiges Klassendesign aussehen.

***Nebenbemerkung*: Anspruch auf hundertprozentige biologische Richtigkeit und Eignung wird nicht erhoben.**

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }

public class Mammal extends Animal { ..... }     //  no walking
public class Bird extends Animal { ..... }       //  no flying
public class Fish extends Animal { ..... }       //  no swimming

public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
public class Whale extends Mammal implements Swimming { ..... }
public class Frog1 extends Animal implements Amphibian { ..... }
public class Frog2 extends Animal implements Walking, Swimming { ..... }
public class Drone implements Flying { ..... }
```

Die Klassen für Säugetiere, Vögel und Fische werden wie bisher von der Klasse für Tiere allgemein abgeleitet, jetzt aber *ohne* Funktionalität für die Fortbewegung.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }
public class Mammal extends Animal { ..... }     //  no walking
public class Bird extends Animal { ..... }       //  no flying
public class Fish extends Animal { ..... }       //  no swimming
public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
public class Whale extends Mammal implements Swimming { ..... }
public class Frog1 extends Animal implements Amphibian { ..... }
public class Frog2 extends Animal implements Walking, Swimming { ..... }
public class Drone implements Flying { ..... }
```

An verschiedenen Stellen in der biologischen Hierarchie aller Arten weiß man, dass alle Arten, die darunter fallen, eine bestimmte Fortbewegungsart haben. An diesen Stellen wird das jeweils passende Interface implementiert, und damit steht die jeweilige Fortbewegungsart für alle darunter fallenden Arten zur Verfügung.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }
public class Mammal extends Animal { ..... }     //  no walking
public class Bird extends Animal { ..... }       //  no flying
public class Fish extends Animal { ..... }       //  no swimming
public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
public class Whale extends Mammal implements Swimming { ..... }
public class Frog1 extends Animal implements Amphibian { ..... }
public class Frog2 extends Animal implements Walking, Swimming { ..... }
public class Drone implements Flying { ..... }
```

Bei Amphibien haben wir die Wahl, ob wir das Interface für Amphibien oder die beiden Interfaces für Laufen und Schwimmen implementieren, besser ist natürlich ersteres.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Animal { ..... }
public class Mammal extends Animal { ..... }     //  no walking
public class Bird extends Animal { ..... }        //  no flying
public class Fish extends Animal { ..... }        //  no swimming
public class PasserineBird extends Bird implements Flying { ..... }
public class Chiropter extends Mammal implements Flying { ..... }
public class Whale extends Mammal implements Swimming { ..... }
public class Frog1 extends Animal implements Amphibian { ..... }
public class Frog2 extends Animal implements Walking, Swimming { ..... }
public class Drone implements Flying { ..... }
```

Und wie Sie sehen, ist dieses Design sogar flexibel genug, um über Biologie hinauszugreifen.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FlightHandler {
   .........
   public void letItFly ( Flying flying ) {
      .........
      int maxFlightHeight = flying.maxFlightHeight();
      .........
   }
   .........
}
```

Auf dieser Folie sehen Sie noch einen wesentlichen Vorteil: Alles, was fliegen kann, kann unterschiedslos behandelt werden in einem Kontext, in dem es nur auf die Flugfähigkeit ankommt und von allem anderen abstrahiert wird. Eine solche Methode braucht nur einmal implementiert zu werden und kann auf flugfähige Vögel, Fledermäuse, Drohnen und so weiter gleichermaßen angewendet werden.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class FlightHandler {
   .........
   public void letThemAllFly ( Flying[ ] flyings ) {
      .........
      for ( Flying flying : flyings ) {
         int maxFlightHeight = flying.maxFlightHeight();
         .........
      }
      .........
   }
   .........
}
```

Kleine Variation des Beispiels, rein zur Illustration: ein Array von Flugtieren und -objekten. Die Komponenten des Arrays können alle Klassen, die Flying implementieren, in beliebiger Mischung enthalten.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
   double apply ( double x );
}

public class QuadraticFunction implements DoubleToDoubleFunction {
   private double summand;
   private double factorQ;
  private double factorL;
   public QuadraticFunction ( double factorQ, double factorL, double summand ) {
      this.factorQ = factorQ;
      this.factorL = factorL;
      this.summand = summand;
  }
   public double apply ( double x ) {
       return x * x * factorQ + x * factorL + summand;
   }
}
```

Noch ein letztes Beispiel für ein Interface und eine beispielhafte Klasse, die dieses Interface implementiert.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
   double apply ( double x );
}

public class QuadraticFunction implements DoubleToDoubleFunction {
    private double summand;
    private double factorQ;
   private double factorL;
    public QuadraticFunction ( double factorQ, double factorL, double summand ) {
       this.factorQ = factorQ;
       this.factorL = factorL;
       this.summand = summand;
   }
    public double apply ( double x ) {
        return x * x * factorQ + x * factorL + summand;
    }
}
```

Das Interface DoubleToDoubleFunction repräsentiert reellwertige mathematische Funktionen mit einem reellwertigen Argument. Die repräsentierte Funktion wird durch Methode apply aufgerufen.

Die Klasse unten repräsentiert allgemeine quadratische Funktionen, also das Quadrat des Arguments multipliziert mit einem festen Faktor, plus das Argument ohne Quadrierung mit einem zweiten festen Faktor, dann noch ein fester Summand draufaddiert. Selbstverständlich können wir alle reellwertigen Funktionen mit reellwertigem Argument, die Sie in der Mathematik kennengelernt haben, als Klassen repräsentieren, die das Interface DoubleToDoubleFunction implementieren, zum Beispiel auch trigonometrische Funktionen oder Exponentialfunktionen.

***Nebenbemerkung*: Im Package java.util.function finden Sie einige Interfaces, die Funktionen im mathematischen Sinn ausdrücken und analog zu dem hier definierten konzipiert sind, ebenfalls mit Methode apply.**

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public double plotFunction ( DoubleToDoubleFunction fct,
                             double a, double b,
                             int numberOfPoints,
                             Plotter plotter ) {
   double stepWidth = ( b – a ) / ( numberOfPoints – 1 );
   for ( int i = 0; i < numberOfPoints; i++ ) {
      double x = a + i * stepWidth;
      plotter.plotPoint ( x, fct.apply(x) );
}
```

Hier sehen Sie ein Beispiel dafür, wie sinnvoll es ist, das allgemeine Konzept „reellwertige Funktion mit reellwertigem Argument“ durch ein Interface zu realisieren und dann im nächsten spezielle Funktionen dieses Typs durch Klassen realisieren, die dieses Interface implementieren.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public double plotFunction ( DoubleToDoubleFunction fct,
                             double a, double b,
                             int numberOfPoints,
                             Plotter plotter ) {
   double stepWidth = ( b – a ) / ( numberOfPoints – 1 );
   for ( int i = 0; i < numberOfPoints; i++ ) {
      double x = a + i * stepWidth;
      plotter.plotPoint ( x, fct.apply(x) );
}
```

Sie können in diese Methode ein Objekt einer beliebigen Klasse hineinstecken, die das Interface DoubleToDoubleFunction implementiert. Abstrakt gesehen, kann diese Methode also beliebige reellwertige Funktionen mit reellwertigen Parametern verarbeiten.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public double plotFunction ( DoubleToDoubleFunction fct,
                             double a, double b,
                             int numberOfPoints,
                             Plotter plotter ) {
   double stepWidth = ( b – a ) / ( numberOfPoints – 1 );
   for ( int i = 0; i < numberOfPoints; i++ ) {
      double x = a + i * stepWidth;
      plotter.plotPoint ( x, fct.apply(x) );
}
```

Wir werden hier nicht ins Detail gehen, sondern nehmen einfach an, wir haben eine Klasse Plotter geschrieben oder von irgendwoher importiert. Diese Klasse habe eine Methode plotPoint, die die beiden Koordinaten eines Punktes als Parameter bekommt und den Punkt dann auf irgendeiner Zeichenfläche visuell darstellt. Alles Weitere zu Klasse Plotter interessiert uns hier nicht, uns geht es hier um DoubleToDoubleFunction.

Jede Funktion, die durch eine von doubleToDoubleFunction implementierende Klasse realisiert ist, kann unterschiedslos durch Methode plotFunction „geplottet“ werden.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleToDoubleFctSum
                                    implements DoubleToDoubleFunction {
   private DoubleToDoubleFunction fct1;
   private DoubleToDoubleFunction fct2;
   public DoubleToDoubleFctSum ( DoubleToDoubleFunction fct1,
                                    DoubleToDoubleFunction fct2 ) {
      this.fct1 = fct1;
      this.fct2 = fct2;
   }
   public double apply ( double x ) {
      return fct1.apply(x) + fct2.apply(x);
   }
}
```

Nicht nur konkrete Funktionen, sondern auch Zusammensetzungen von Funktionen kann man als Klassen realisieren, die von Interface DoubleToDoubleFunction abgeleitet sind. Hier sehen Sie eine Klasse, die die Summe zweier beliebiger Funktionen realisiert.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleToDoubleFctSum
                                    implements DoubleToDoubleFunction {
  private DoubleToDoubleFunction fct1;
  private DoubleToDoubleFunction fct2;
  public DoubleToDoubleFctSum ( DoubleToDoubleFunction fct1,
                                  DoubleToDoubleFunction fct2 ) {
     this.fct1 = fct1;
     this.fct2 = fct2;
  }
  public double apply ( double x ) {
     return fct1.apply(x) + fct2.apply(x);
  }
}
```

Die beiden Funktionen sind Objekte von Klassen, die von Interface DoubleToDoubleFunction abgeleitet sind. Sie werden in private-Attributen gehalten und im Konstruktor initialisiert. Bei jedem Aufruf von Methode apply sind die beiden Funktionen in den private-Attributen vorhanden und können verwendet werden.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class DoubleToDoubleFctComposer
                                    implements DoubleToDoubleFunction {
   private DoubleToDoubleFunction fct1;
   private DoubleToDoubleFunction fct2;
   public DoubleToDoubleFctComposer ( DoubleToDoubleFunction fct1,
                                      DoubleToDoubleFunction fct2 ) {
      this.fct1 = fct1;
      this.fct2 = fct2;
   }
   public double apply ( double x ) {
      return fct1.apply ( fct2.apply(x) );
   }
}
```

Eine kleine Variation des letzten Beispiels: Diese Klasse realisiert nicht die *Summe* aus zwei Funktionen, sondern ihre *Komposition*. Im Gegensatz zur Summe ist die Komposition nicht kommutativ, das heißt, beim Konstruktoraufruf muss auf die korrekte Reihung der beiden Funktionen geachtet werden, die man komponieren möchte, sonst kommt die falsche Komposition heraus.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Um das volle Potential zu erahnen, verwenden wir diese Klassen nun noch in einem kleinen, rein illustrativen Beispiel.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Hier noch einmal die quadratische Funktion, wie wir sie soeben definiert haben.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Ein paar weitere Funktionen von Klassen, die analog zur quadratischen Funktion definiert sind und ebenfalls das Interface DoubleToDoubleFunction implementieren, zum Beispiel andere Monome wie die Kubikfunktion, Polynome, trigonometrische Funktionen, Exponentialfunktionen, Logarithmen und so weiter.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Mit Klassen wie DoubleToDoubleFctSum und DoubleToDoubleFctComposer können wir daraus dann komplexere Funktionen konstruieren.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Solche Klassen, die Funktionen aus anderen Funktionen zusammensetzen, kann man natürlich ebenfalls beliebig vielfältig entwickeln, zum Beispiel Produkt von zwei Funktionen. Die Klasse DoubleToDoubleFctProduct lässt sich offensichtlich völlig analog zu DoubleToDoubleFunctionSum realisieren.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Natürlich spricht weder etwas gegen mehrere Objekte einer solchen Klasse ...

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

... noch gegen die Verwendung desselben Objektes an verschiedenen Stellen. Die Objekte, auf die fct5 beziehungsweise fct7 verweisen, enthalten beide als Attribut einen Verweis auf das Objekt, auf das fct1 verweist.

Interfaces

TECHNISCHE UNIVERSITÄT DARMSTADT

```
DoubleToDoubleFunction fct1 = new QuadraticFunction ( ..... );
DoubleToDoubleFunction fct2 = new CubicFunction ( ..... );
DoubleToDoubleFunction fct3 = new LogarithmFunction ( ..... );
DoubleToDoubleFunction fct4 = new SineFunction ( ..... );

DoubleToDoubleFunction fct5 = new DoubleToDoubleFctSum ( fct1, fct2 );
DoubleToDoubleFunction fct6 = new DoubleToDoubleFctComposer ( fct3, fct4 );
DoubleToDoubleFunction fct7 = new DoubleToDoubleFctProduct ( fct1, fct6 );
DoubleToDoubleFunction fct8 = new DoubleToDoubleFctSum ( fct5, fct7 );

DoubleToDoubleFunction fct9 = new DoubleToDoubleFctDerivation ( fct8 );
```

Wenn das Design so allgemein wie hier ist, dann kann beispielsweise auch die mathematische Ableitung als Klasse eingebaut werden. Damit diese Klasse ihrem Namen gerecht wird, müsste sie beim Aufruf von Methode apply mit einem double-Wert x die Ableitung von fct8 an der Stelle x zumindest näherungsweise berechnen, also beispielsweise eine Folge von Differenzenquotienten an x mit abnehmender Intervallbreite bilden und den Grenzwert dieser Folge durch Extrapolation schätzen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Begriffsbildung:
Subtypen
und
statischer / dynamischer Typ

Wir brauchen noch ein paar grundlegende Begriffe, die aus dem bisher Gesagten leicht verständlich sein sollten.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Klassen X: alle direkt oder indirekt mit extends von X abgeleiteten Klassen
- von Interfaces I:
  - alle direkt oder indirekt mit extends von I abgeleiteten Interfaces
  - alle I oder ein von I abgeleitetes Interface implementierenden Klassen und deren Subtypen
- von Array von Referenztyp T: alle Arraytypen, deren Komponententyp ein Subtyp von T ist

Der erste einzuführende Begriff ist „Subtyp“. Dafür machen wir eine Fallunterscheidung. Enum-Typen sind hier wieder unter Klassen subsumiert, da sie ja spezielle Klassen sind, die von Klasse java.lang.Enum abgeleitet sind.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Klassen X: alle direkt oder indirekt mit extends von X abgeleiteten Klassen
- von Interfaces I:
  - alle direkt oder indirekt mit extends von I abgeleiteten Interfaces
  - alle I oder ein von I abgeleitetes Interface implementierenden Klassen und deren Subtypen
- von Array von Referenztyp T: alle Arraytypen, deren Komponententyp ein Subtyp von T ist

Jede Klasse Y, die von einer Basisklasse X mit extends direkt abgeleitet ist, ist ein Subtyp der Basisklasse X, ebenso alle von Y direkt mit extends abgeleiteten Klassen und so weiter. Allgemeiner gesprochen, werden alle direkt oder indirekt abgeleiteten Klassen Subtypen genannt, also auch wenn noch eine oder mehrere Klassen in der Vererbungshierarchie dazwischen stehen.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Klassen X: alle direkt oder indirekt mit extends von X abgeleiteten Klassen
- von Interfaces I:
  - alle direkt oder indirekt mit extends von I abgeleiteten Interfaces
  - alle I oder ein von I abgeleitetes Interface implementierenden Klassen und deren Subtypen
- von Array von Referenztyp T: alle Arraytypen, deren Komponententyp ein Subtyp von T ist

Dasselbe gilt völlig analog für Interfaces.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Klassen X: alle direkt oder indirekt mit extends von X abgeleiteten Klassen
- von Interfaces I:
  - alle direkt oder indirekt mit extends von I abgeleiteten Interfaces
  - alle I oder ein von I abgeleitetes Interface implementierenden Klassen und deren Subtypen
- von Array von Referenztyp T: alle Arraytypen, deren Komponententyp ein Subtyp von T ist

Interfaces haben aber auch Klassen als Subtypen, und zwar kann man man das so sagen: Jede Klasse, die durch eine Kette von „extends“ und „implements“ aus einem Interface abgeleitet ist, ist ein Subtyp dieses Interface.

***Nebenbemerkung*: Wie Sie sich leicht selbst überlegen können, kommt in einer solchen Kette maximal einmal „implements“ vor.**

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

- von Klassen X: alle direkt oder indirekt mit extends von X abgeleiteten Klassen
- von Interfaces I:
  - alle direkt oder indirekt mit extends von I abgeleiteten Interfaces
  - alle I oder ein von I abgeleitetes Interface implementierenden Klassen und deren Subtypen
- von Array von Referenztyp T: alle Arraytypen, deren Komponententyp ein Subtyp von T ist

Die Subtyp-Relation überträgt sich auch auf Arraytypen: Ist A ein Subtyp von B, dann ist „Array von A“ ein Subtyp von „Array von B“.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  .....
}

public class Y extends X {
  .....
}
```

```
public class A {
  X a;
  public X m ( X b ) {
    a = new Y();
    return new Y();
  }
}
```

Speziell für Klassen sehen wir uns die Verwendung von Subtypen zuerst an einem kleinen Beispiel an und sammeln die Erkenntnisse nach diesem Beispiel systematisch zusammen. Interfaces und Arraytypen sind analog, so dass wir uns beispielhaft auf Klassen konzentrieren können. Wir erinnern uns, dass auch Enum-Typen unter Klassen subsumiert sind.

Links sehen sie einfach wieder Beispielmaterial: eine Klasse Y abgeleitet von einer Klasse X. In der Klasse A rechts werden die drei Fälle illustriert.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   .........
}

public class Y extends X {
   .........
}
```

```
public class A {
  X a;
  public X m ( X b ) {
    a = new Y();
    return new Y();
  }
}
```

Hier wird einer Referenz ein Objekt von einem Subtyp zugewiesen.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   .....
}

public class Y extends X {
   .....
}
```

```
public class A {
   X a;
   public X m ( X b ) {
      a = new Y();
      return new Y();
   }
}
```

Auch bei Rückgabetyp X kann ein Objekt eines Subtyps von X wie etwa Y zurückgeliefert werden.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
  X a;
  public X m ( X b ) {
    a = new Y();
    return new Y();
  }
}
```

```
A a = new A();
X x1 = new Y();
Y y = new Y();
X x2 = a.m ( y );
```

Die Klasse A, die Sie auf der letzten Folie rechts gesehen haben, sehen Sie eins-zu-eins jetzt auf der linken Seite noch einmal.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
  X a;
  public X m ( X b ) {
    a = new Y();
    return new Y();
  }
}
```

```
A a = new A();
X x1 = new Y();
Y y = new Y();
X x2 = a.m ( y );
```

Ein zweites Beispiel einer Zuweisung mit einem Objekt eines Subtyps, diesmal an eine lokale Variable.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {
  X a;
  public X m ( X b ) {
    a = new Y();
    return new Y();
  }
}
```

```
A a = new A();
X x1 = new Y();
Y y = new Y();
X x2 = a.m ( y );
```

Und in der letzten Zeile noch ein Beispiel dafür, dass auch als Parameterwert ein Objekt eines Subtyps übergeben werden kann.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überall wo ein Referenztyp (der Supertyp) erwartet wird, kann ein Objekt eines Subtyps verwendet werden:

- **In einer Zuweisung an eine Variable/Konstante**
- **Als Parameterwert**
- **Als Rückgabewert**

Wir sind jetzt an dem angekündigten Punkt, dass wir die Erkenntnisse aus den Beispielen zusammensammeln können.

Wir haben gesehen: Einer Variablen von einem Referenztyp kann ein Objekt desselben Typs oder eines Subtyps zugewiesen werden. Dabei spielt es keine Rolle, ob dies in Form einer Initialisierung, einer Zuweisung, eines Parameterwertes bei einem Methodenaufruf oder einer Rückgabe passiert.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überall wo ein Referenztyp (der Supertyp) erwartet wird, kann ein Objekt eines Subtyps verwendet werden:

- **In einer Zuweisung an eine Variable/Konstante**
- **Als Parameterwert**
- **Als Rückgabewert**

Natürlich nur, wenn der Subtyp Objekte haben kann. Bei Interfaces und abstrakten Klassen als Subtyp geht das also nicht. Bei Enum-Typen kommen nur die in der Definition aufgezählten Konstanten in Frage, da keine weiteren Objekte von einem Enum-Typ erzeugt werden können.

Subtypen

TECHNISCHE UNIVERSITÄT DARMSTADT

Überall wo ein Referenztyp (der Supertyp) erwartet wird, kann ein Objekt eines Subtyps verwendet werden:

- **In einer Zuweisung an eine Variable/Konstante**
- **Als Parameterwert**
- **Als Rückgabewert**

Es gibt natürlich auch den komplementären Begriff: X ist genau dann ein Supertyp von Y, wenn Y ein Subtyp von X ist.

***Nebenbemerkung*: Zur Vermeidung von Fallunterscheidungen werden die Begriffe Subtyp und Supertyp häufig so definiert, dass jeder Referenztyp auch sein eigener Subtyp und Supertyp ist. Dem folgen wir hier nicht.**

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y ();
a = new X ();
a = new Z ();
a = null;
a = new Y ();
```

Nun noch die letzte grundlegende Begriffsbildung in diesem Abschnitt: die Unterscheidung zwischen dem statischen und dem dynamischen Typ einer Referenz.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y ();
a = new X ();
a = new Z ();
a = null;
a = new Y ();
```

Der *statische* Typ einer Referenz ist der, mit dem sie definiert wird. In diesem Beispiel ist also X der statische Typ von a. Dieser Typ ist unveränderlich mit der Referenz verknüpft, daher der Begriff *statischer* Typ.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y ();
a = new X ();
a = new Z ();
a = null;
a = new Y ();
```

Der *dynamische* Typ einer Referenz ist der Typ des Objekts, auf das diese Referenz verweist. Der dynamische Typ muss immer entweder gleich dem statischen Typ oder ein Subtyp des statischen Typs sein.

Wie dieses kleine Beispiel schon zeigt, kann sich der dynamische Typ einer Variablen im Laufe der Zeit beliebig häufig ändern.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y ();
a = new X ();
a = new Z ();
a = null;
a = new Y ();
```

Nach Setzung der Variable auf null hat sie formal weiterhin einen dynamischen Typ, genannt Nulltyp (null type). Aber wir können uns das auch informell so vorstellen, dass die Variable gar keinen dynamischen Typ mehr hat, bis sie wieder auf ein Objekt gesetzt wird.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;
   public void m1() { ... }
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();

a.n1 = 1;

a.n2 = 2;

a.m1();

a.m2();
```

Wir schauen uns jetzt noch einmal zusammenfassend an, was der statische und was der dynamische Typ jeweils entscheidet.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;
   public void m1() { ... }
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();
a.n1 = 1;
a.n2 = 2;
a.m1();
a.m2();
```

Zugriff auf ein Attribut des statischen Typs X – sofern das Zugriffsrecht passt – ist kein Problem, das kennen wir ja auch schon.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;
   public void m1() { ... }
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();
a.n1 = 1;
a.n2 = 2;
a.m1();
a.m2();
```

Was hingegen *nicht* geht, ist der Zugriff auf ein Attribut, das im statischen Typ noch nicht definiert ist, auch wenn der dynamische Typ dieses Attribut besitzt.

Das ist auch sinnvoll, denn in unwesentlich komplexeren Situationen als hier kann der Compiler schon nicht mehr bei der Übersetzung überprüfen, ob der dynamische Typ von a wirklich Y ist, also ob das Objekt, auf das a verweist, wirklich dieses Attribut hat.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;
   public void m1() { ... }
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();

a.n1 = 1;

a.n2 = 2;

a.m1();

a.m2();
```

Die Methode m2 ist ebenfalls nicht im statischen Typ vorhanden und kann daher aus denselben Gründen nicht aufgerufen werden.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;
   public void m1() { ... }
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();

a.n1 = 1;

a.n2 = 2;

a.m1();

a.m2();
```

Die Methode m1 ist schon im statischen Typ eingeführt, kann also aufgerufen werden. Wir wissen schon, dass dann aber die Implementation für Y aufgerufen wird.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
   public int n1;
   abstract public void m1();
}

public class Y extends X {
   public int n2;
   public void m1() { ... }
   public void m2() { ... }
}
```

```
X a = new Y();

a.n1 = 1;

a.n2 = 2;

a.m1();

a.m2();
```

Das geht auch, wenn die Methode im statischen Typ nur *definiert*, aber nicht *implementiert* wurde, also abstrakt ist. Denn die nichtvorhandene Implementation der Methode in der Basisklasse kommt ja nicht ins Spiel. Sie kann *niemals* ins Spiel kommen, da von einer abstrakten Klasse kein Objekt eingerichtet werden kann und der Typ des Objekts die Implementation der Methode auswählt.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der statische Typ entscheidet:

  auf welche Attribute und Methoden zugegriffen werden darf

- Der dynamische Typ entscheidet:

  welche *Implementation* einer Methode aufgerufen wird

Hier haben wir noch einmal die Regeln zusammengefasst. Auf ein Attribut oder eine Methode kann genau dann über eine Referenz zugegriffen werden, wenn das Attribut beziehungsweise die Methode schon im statischen Typ vorhanden ist – dort definiert oder ererbt. Welche *Implementation* der Methode aufgerufen wird, das hängt von der Information im Objekt, also vom dynamischen Typ ab.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
GeomShape2D [ ] picture
      = new GeomShape2D [3];

picture[0] = new Circle ( ..... );
picture[1] = new Rectangle ( ..... );
picture[2] = new StraightLine ( ..... );
```

Wir hatten schon beim Thema abstrakte Klassen ein Beispiel gesehen, warum diese Regelung gut ist. Hier noch einmal die Folie zum Einrichten eines beispielhaften kleinen Bildes als ein Array einer gemeinsamen Basisklasse aller zweidimensionalen geometrischen Formen, und für jede Art von geometrischer Form gibt es eine eigene, davon abgeleitete Klasse.

Statischer / dynamischer Typ

TECHNISCHE UNIVERSITÄT DARMSTADT

```
for ( int i = 0; i < picture.length; i++ ) {
    picture[i].paint ( g );
}
```

Und so wurde das Bild auf eine Zeichenfläche gezeichnet. Der Komponententyp des Arrays ist von der Basisklasse. Hier können keine Methoden aufgerufen werden, die in der Basisklasse nicht schon *definiert* sind. Aber sie müssen in der Basisklasse noch nicht *implementiert* sein, denn die tatsächlich aufgerufene Implementation der Zeichenmethode ist die der abgeleiteten Klasse. Das ist auch sinnvoll; natürlich sollte das Zeichnen bei jedem geometrischen Objekt spezifisch für diese Objektart sein, und das ist eben die Implementation in der jeweiligen abgeleiteten Klasse.

Downcast

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A { ..... }
public class B1 extends A { ..... }
public class B2 extends A { ..... }
public class C extends B1 { ..... }
public class D extends C { ..... }

A a = new A();
A b = new B1();
A c = new B2();
A d = new C();
A e = new D();
```

```
B1 x;
if ( a instanceof B1 )
   x = (B1)a;
if ( b instanceof B1 )
   x = (B1)b;
if ( c instanceof B1 )
   x = (B1)c;
if ( d instanceof B1 )
   x = (B1)d;
if ( e instanceof B1 )
   x = (B1)e;
```

Erinnerung **an Kapitel 01b, Abschnitt zu Konversionen: Primitive Datentypen können ineinander konvertiert werden.**

Eine analoge Möglichkeit für Referenztypen sehen wir uns kurz an einem rein illustrativen Beispiel auf dieser Folie an.

Downcast

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A { ..... }
public class B1 extends A { ..... }
public class B2 extends A { ..... }
public class C extends B1 { ..... }
public class D extends C { ..... }

A a = new A();
A b = new B1();
A c = new B2();
A d = new C();
A e = new D();
```

```
B1 x;
if ( a instanceof B1 )
   x = (B1)a;
if ( b instanceof B1 )
   x = (B1)b;
if ( c instanceof B1 )
   x = (B1)c;
if ( d instanceof B1 )
   x = (B1)d;
if ( e instanceof B1 )
   x = (B1)e;
```

Der Operator instanceof ist ein binärer, boolescher Infix-Operator. Sein erster Operand ist eine Variable oder Konstante von einem Referenztyp. Hier ist es eine lokale Variable, es könnte aber auch ein Attribut eines Objektes oder eine Komponente eines Arrays sein, Hauptsache Referenztyp.

Der zweite Operand ist ein Referenztyp. Der Operator liefert genau dann true zurück, wenn der dynamische Typ des ersten Operanden gleich dem zweiten Operanden oder ein Subtyp des zweiten Operanden ist.

Im ersten und dritten Fall ist das Ergebnis also false, ansonsten true.

Downcast

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A { ..... }
public class B1 extends A { ..... }
public class B2 extends A { ..... }
public class C extends B1 { ..... }
public class D extends C { ..... }

A a = new A();
A b = new B1();
A c = new B2();
A d = new C();
A e = new D();
```

```
B1 x;
if ( a instanceof B1 )
  x = (B1)a;
if ( b instanceof B1 )
  x = (B1)b;
if ( c instanceof B1 )
  x = (B1)c;
if ( d instanceof B1 )
  x = (B1)d;
if ( e instanceof B1 )
  x = (B1)e;
```

Das ist jetzt die eigentliche Konvertierung, genannt *Downcast*, also Abwärtskonvertierung. Sie sehen, Man sollte einen Downcast eigentlich nie ohne vorherige Abfrage mit instanceof durchführen, denn in den Fällen, die hier rot unterlegt sind, würde der Ablauf des Programms vom Laufzeitsystem mit einer Fehlermeldung abgebrochen werden, da der dynamische Typ des ersten Operanden eben *nicht* so ist, dass der dynamische Typ von x gleich seinem statischen Typ B1 oder einem Subtyp von B1 wie etwa C und D wäre.

***Vorgriff*: In Kapitel 05 werden wir sehen, dass wir diesen Prozessabbruch auch verhindern können.**

Downcast

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A { ..... }

public class B extends A {
   public int n;          // nicht in A definiert
   public void m() {  // nicht in A definiert
       .....
}
```

```
A x = new B();
x.n = 1;
x.m();

B y = (B)x;
y.n = 1;
y.m();
```

Wozu braucht man den Downcast? Nun, manchmal ist es doch sinnvoll, auf Funktionalität zuzugreifen, die nicht im statischen, wohl aber im dynamischen Typ definiert ist. Das geht nur über eine Referenz vom dynamischen Typ. Daher wird der Compiler die rot unterlegten Zugriffe nicht durchgehen lassen. Die grün unterlegten Zugriffe sind hingegen kein Problem: Der Downcast ist der kritische Moment, der Zugriff nach gelungenem Downcast sind unkritisch.

***Nebenbemerkung*: Wir haben eben bei der letzten Folie zu Recht gesagt, dass man eigentlich nie den Downcast ohne vorherige Abfrage mit instanceof verwenden soll. Aber in diesem kleinen Beispiel ist das ausnahmsweise einmal unkritisch.**

Bei Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X { ..... }

X [ ] a = new Y [ 3 ];

X b = a [ 2 ];
```

Wie schon gesagt, setzt sich das Thema statischer / dynamischer Typ auch auf Arrays fort: Einer Arrayvariablen von einer Basisklasse kann man ein Arrayobjekt von einer abgeleiteten Klasse zuweisen. Der Zugriff auf eine Komponente erfolgt dann über den Typ X. Insbesondere kann man nicht auf die Funktionalität von Y zugreifen, die noch nicht in X definiert ist.

Damit schließen wir das Thema statischer / dynamischer Typ ab.

TECHNISCHE UNIVERSITÄT DARMSTADT

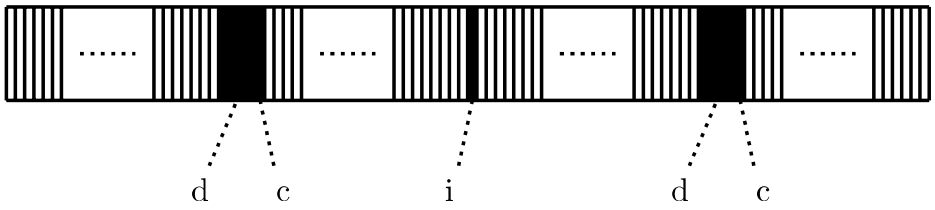
Klassenattribute

Oracle Java Tutorials:
Understanding Class Members
(Abschnitt Class Variables)

Attribute von Klassen kennen wir zur Genüge. Was wir bisher kennen gelernt haben, nennen wir in Zukunft genauer *Objektattribute*. Wir schauen uns kurz eine komplementäre Form von Attributen an, die man *Klassenattribute* nennt. Attribute von Klassen und Klassenattribute sind also nicht dasselbe!

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT




```
public class X {
   public static int i;
   public double d;
   public char c;
}
```

Steht bei der Deklaration eines Attributs das Schlüsselwort static vor dem Typnamen, dann wird für dieses Attribut nicht in jedem Objekt eine Kopie des Attributs angelegt, sondern nur eine einzige Speicherstelle für die ganze Klasse. Die normalen Attribute *ohne* static nennt man dementsprechend Objektattribute, die *mit* static nennt man Klassenattribute.

***Nebenbemerkung*: Versuchen Sie nicht, einen Sinn dahinter zu finden, dass das Schlüsselwort für Klassenattribute static heißt. Dahinter stehen pragmatische Gründe aus der Vorgeschichte lange vor Java, die hier zu weit führen würden.**

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();
X b = new X();

a.i = 123;
int n1 = b.i;        //  n1 == 123
b.i = 321;
int n2 = X.i;        //  n2 == 321
X.i = 213;
int n3 = a.i;        //  n3 == 213
```

Die Konsequenz daraus, dass i ein Klassenattribut ist, ist natürlich, dass jeder Zugriff auf das Attribut i zur selben Speicherstelle führt, egal über welches Objekt.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();
X b = new X();
a.i = 123;
int n1 = b.i;        //  n1 == 123
b.i = 321;
int n2 = X.i;        //  n2 == 321
X.i = 213;
int n3 = a.i;        //  n3 == 213
```

Um das zu sehen, richten wir wie gehabt zwei verschiedene Objekte der Klasse X ein und lassen die Variablen a und b auf je eines dieser Objekte verweisen.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();
X b = new X();
a.i = 123;
int n1 = b.i;        //   n1 == 123
b.i = 321;
int n2 = X.i;        //   n2 == 321
X.i = 213;
int n3 = a.i;        //   n3 == 213
```

Wenn man nun über eine der beiden Variablen, etwa a, dem Attribut i einen Wert zuweist und dann den Wert des Attributs über die andere Variable, b, abfragt, dann kommt natürlich der Wert heraus, der vorher über a zugewiesen wurde, denn a.i und b.i bezeichnen dieselbe, nur einmal eingerichtete Speicherstelle.

Genau das ist bei Objektattributen anders, da spricht man über zwei verschiedene Objekte mit demselben Attributnamen zwei verschiedene Speicherstellen an, die nichts miteinander zu tun haben.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();
X b = new X();

a.i = 123;
int n1 = b.i;        //  n1 == 123
b.i = 321;
int n2 = X.i;        //  n2 == 321
X.i = 213;
int n3 = a.i;        //  n3 == 213
```

Diese Schreibweise ist neu: Wenn das Attribut i ohnehin unabhängig von dem konkreten Objekt ist, dann ist es nur konsequent, wenn man es auch ohne eine Variable ansprechen kann, einfach nur mit dem Namen der Klasse, passend zum Begriff Klassenattribut.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

Math.PI

Math.E

Ein reales Beispiel: Die Klasse java.lang.Math enthält unter anderem diese beiden mathematischen Konstanten, die Kreiszahl und die Eulersche Zahl.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
   ..........
   public static final double PI = 3.14159;
   public static final double E = 2.71828;
   ...........
}
```

Wie werden solche konstanten Klassenattribute definiert: mit static als Klassenattribut und mit final als konstant. Wegen final müssen beide Attribute dann auch initialisiert werden; natürlich werden sie in java.lang.Math mit mehr Nachkommastellen initialisiert sein als auf dieser Folie.

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ..........
  public static final double PI = 3.14159;
  public static final double E = 2.71828;
  ...........
}
```

***Vorgriff*: Weitere Bestandteile der Klasse Math sind wie üblich ausgelassen; auf einige weitere Bestandteile werden wir in Kapitel 03c, Abschnitt zu Klassenmethoden, zu sprechen kommen.**

Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static java.lang.Math.PI;

.........

area = radius * radius * PI;
```

***Erinnerung*: Das Konstrukt „import static“ hatten wir schon in Kapitel 01e, Abschnitt „Erste eigene Enumeration“, gesehen.**

Hier sehen Sie jetzt den allgemeinen Fall: Klassenkonstanten lassen sich mit import static so importieren, dass der Name der Klassenkonstante ohne den Namen der Klasse, zu der die Klassenkonstante gehört, verwendet werden kann.

Die Konstanten in einer Enumeration sind ebenfalls Klassenkonstanten, daher funktioniert es auch bei ihnen. Was wir in Kapitel 01e gesehen hatten, war also „nur“ ein Spezialfall.

Variable und Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n1;                    //  Objektvariable
   public final int N2 = 1;          //  Objektkonstante
   public static int n3;             //  Klassenvariable
   public static final int N4 = 2;   //  Klassenkonstante
}
```

Zum Abschluss des Themas Klassenattribute noch eine systematische Begriffsbildung. Wir sprechen von Objektattributen und Klassenattributen, aber die mit final sind konstant und die ohne final sind variabel. Das schlägt sich in den vier konkretisierenden Begriffen nieder, die Sie in den Kommentaren auf dieser Folie jeweils mit Beispiel sehen.

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Damit sind wir ´zudem jetzt soweit, dass wir die Methoden verstehen können, mit denen man in einem Java-Programm typischerweise Informationen auf den Bildschirm schreibt.

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Wie Sie sehen, gibt es zunächst einmal zwei grundsätzlich unterschiedliche Möglichkeiten, print und print-l-n, die letztere Version wird meist print-line ausgesprochen. Der Unterschied ist, dass println einen Zeilenumbruch hinter die Ausgabe schreibt, das heißt, die darauf folgend nächste Ausgabe ist dann in der nächsten Zeile.

***Nebenbemerkung*: Ein weiterer Unterschied ist, dass die Ausgabe mit println garantiert sofort hinausgeschrieben wird, mit print nur bei entsprechender Konfiguration.**

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Wie Sie ferner sehen, gibt es für die verschiedenen Arten von Daten jeweils eine solche Methode, beide Methoden sind überladen. Es gibt noch ein paar weitere Methoden print beziehungsweise println, aber die auf dieser Folie sind die für uns relevanten.

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Die Klasse System ist vordefiniert in java.lang.

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Diese Klasse System hat unter anderem ein Klassenattribut namens out. Dieses ist von einer ebenfalls vordefinierten Klasse namens java.io.PrintStream (io für input/output).

***Vorgriff*: Klasse PrintStream schauen wir uns in Kapitel 08 noch einmal an.**

System.out.print(ln)

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

Diese Klasse PrintStream hat mehrere Objektmethoden namens print beziehungsweise println.

***Nebenbemerkung*: Das können keine Klassenmethoden sein, denn jedes Objekt von Klasse PrintStream ist potentiell mit einem anderen Ausgabegerät verbunden, und die Ausgabe soll natürlich auf diesem Gerät erfolgen.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Attribute / Komponenten von Referenztypen

Bisher waren Attribute einer Klasse bei uns meist von primitiven Datentypen. Das muss natürlich nicht so sein. Bei Arrays hingegen haben wir schon gesehen, dass die Komponenten von einem Referenztyp sein können.

Attribute von Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class X {
   ..........
}

class Y {
   public X b;
}
```

```
Y a = new Y();

a.b = new X();
```

Wir richten wieder zwei verschiedene Klassen ein, X und Y.

Attribute von Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class X {
  ..........
}

class Y {
   public X b;
}
```

```
Y a = new Y();

a.b = new X();
```

Die Details der Klasse X interessieren uns nicht, weswegen wir diese Details wieder nur mit Punkten andeuten.

Attribute von Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class X {
   ..........
}

class Y {
   public X b;
}
```

```
Y a = new Y();

a.b = new X();
```

Das Einzige, was uns an Klasse Y interessiert, ist das Attribut b, das von Klasse X ist.

Attribute von Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

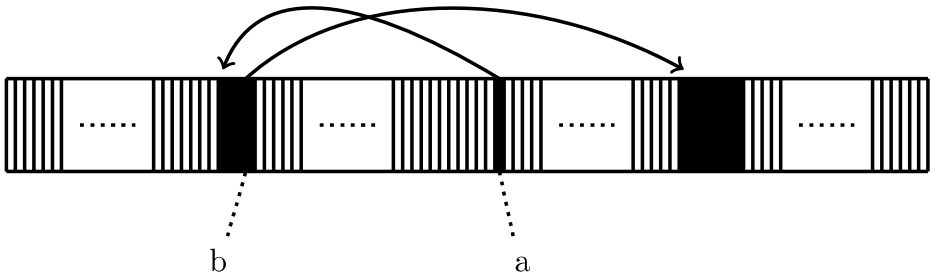
```
class X {
   ..........
}

class Y {
   public X b;
}
```

```
Y a = new Y();

a.b = new X();
```

Das Objekt von Klasse Y, auf das a verweist, verweist nun mittels Attribut b seinerseits auf ein Objekt, nämlich von Klasse X.

Attribute von Referenztypen

TECHNISCHE UNIVERSITÄT DARMSTADT

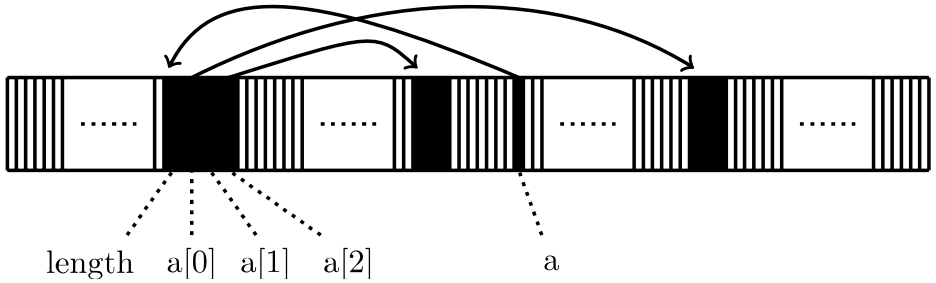



```
Y a = new Y();
a.b = new X();
```

Und so kann man sich das Ganze dann im Speicher vorstellen: Die Referenz a verweist wie gesagt auf ein Objekt von Klasse Y, und das Attribut a.b verweist auf ein Objekt von Klasse X.

Array von Klassentyp

TECHNISCHE UNIVERSITÄT DARMSTADT



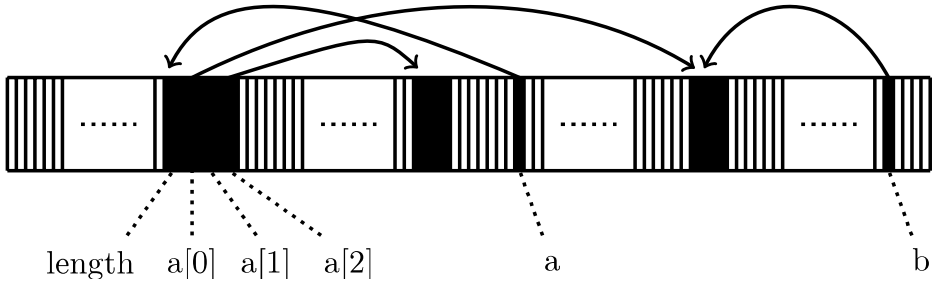

```
X [ ] a = new X [ 3 ];
a[0] = new X();
a[2] = new X();
```

Analog sieht es aus, wenn ein Array von Klasse X eingerichtet wird. Zwei der drei Komponenten verweisen auf neu eingerichtete Objekte.

***Vorgriff*: Arraykomponenten von einem Klassentyp werden bei Einrichtung des Arrays immer mit null initialisiert, daher hat die Komponente an Index 1 den Wert null. Implizite Initialisierungen sehen wir uns im Kapitel 03c, Abschnitt zu Variablendefinitionen, genauer an.**

Array von Klassentyp

TECHNISCHE UNIVERSITÄT DARMSTADT



for (X b : a)

Wir haben schon diese Kurzform für die for-Schleife beim Durchlauf durch ein Array gesehen: Name des Komponententyps, dann ein neu eingeführter Identifier, nach dem Doppelpunkt schließlich der Name des Arrays.

Die Konsequenz ist, dass die Komponenten von a nacheinander in den einzelnen Schleifendurchläufen in aufsteigender Reihenfolge der Indizes jeweils der Variablen b zugewiesen werden. Im obigen Bild ist die Situation im ersten Durchlauf dargestellt: b enthält denselben Wert wie die Komponente von a an Index 0, verweist also auf dasselbe Objekt.

Attribut von Arraytyp

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   int [ ] a;
   String b;
}
```

Und natürlich geht es auch umgekehrt: Attribute einer Klasse können Arrays sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Klasse java.lang.Object

Java Oracle Tutorial: Object as a Superclass

Was noch fehlt, ist die Klasse aller Klassen. Diese trägt den Namen Object und findet sich im zentralen Package java.lang, das ja automatisch in jedem Java-Quelltext vom Compiler importiert wird.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

public class X extends Y { ..... }

→ Y ist direkte Basisklasse von X

public class Z { ..... }

→ Object ist direkte Basisklasse von Z

⇒ Jede Klasse ist direkt oder indirekt abgeleitet von Object

Bevor wir uns die Klasse Object selbst kurz ansehen, klären wir zuerst, wie sie sich in unser bisheriges Bild von Java einfügt.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

public class X extends Y { ..... }

→ Y ist direkte Basisklasse von X

public class Z { ..... }

→ Object ist direkte Basisklasse von Z

⇒ Jede Klasse ist direkt oder indirekt abgeleitet von Object

Der erste Fall ist altbekannt: Mit extends wird dafür gesorgt, dass eine Klasse X direkt von einer anderen Klasse Y abgeleitet wird, beziehungsweise Y ist die Basisklasse von X.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

public class X extends Y { ..... }

→ Y ist direkte Basisklasse von X

public class Z { ..... }

→ Object ist direkte Basisklasse von Z

⇒ Jede Klasse ist direkt oder indirekt abgeleitet von Object

Wir hatten bisher nichts weiter zu dem Fall gesagt, dass *kein* extends im Kopf der Klassendefinition steht. Das holen wir jetzt nach: Falls nicht *explizit* eine Basisklasse mit extends definiert wird, dann wird *implizit* Klasse Object als Basisklasse genommen. Das gilt auch für die Klassen in der Java-Standardbibliothek und in jeder anderen Bibliothek von Java-Klassen.

Interfaces sind hier nicht im Spiel, ohne extends erweitert ein Interface kein anderes Interface – und Klassen erweitert ein Interface sowieso nie.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

public class X extends Y { ..... }

→ Y ist direkte Basisklasse von X

public class Z { ..... }

→ Object ist direkte Basisklasse von Z

⇒ Jede Klasse ist direkt oder indirekt abgeleitet von Object

Das heißt also, dass jede Klasse A außer Klasse Object von irgendeiner anderen Klasse B abgeleitet ist. Wenn B nicht gleich Object ist, ist B wiederum von einer anderen Klasse C abgeleitet. Wenn C nicht gleich Object ist, ist C von einer Klasse D abgeleitet, und so weiter. Diese Kette A, B, C, D, ... von Basisklassen ist natürlich endlich, sie kann aber nur bei Klasse Object enden.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Object {
    .....
    public boolean equals ( Object obj )  { ..... }
    .....
    public String toString ()  { ..... }
    .....
}
```

Wir schauen uns nur zwei Methoden von Klasse Object beispielhaft an.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Object {
    .....
    public boolean equals ( Object obj )  { ..... }
    .....
    public String toString ()  { ..... }
    .....
}
```

Methode equals hat die Aufgabe, den Test auf Wertgleichheit zu realisieren, also nicht nur dann true zurückzuliefern, wenn das Objekt hinter dem Parameter gleich dem Objekt ist, mit dem die Methode equals aufgerufen wird, sondern auch dann, wenn die Attribute in den beiden Objekten wertgleich sind.
***Erinnerung*: Abschnitt „Begriffsbildung: Objektidentität vs. Wertgleichheit“ weiter vorne in diesem Kapitel.**

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Object {
    .....
    public boolean equals ( Object obj )  { ..... }
    .....
    public String toString ()  { ..... }
    .....
}
```

Diese Methode soll eine Darstellung des Zustands des Objektes in Form eines Strings zurückliefern.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Object {
    .....
    public boolean equals ( Object obj )  { ..... }
    .....
    public String toString ()  { ..... }
    .....
}
```

Die Methoden von Klasse Object werden zwar an jede Klasse vererbt, tun aber nicht unbedingt das, was für die jeweilige Klasse sinnvoll ist. In professioneller Softwareentwicklung in Java wird man daher die Methoden von Klasse Object in jeder eigenen Klasse überschreiben, und zwar so, dass die überschreibenden Implementationen passend für die jeweilige Klasse sind.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i;
   private double d;
   private char c;
   .........
   public boolean equals ( Object obj ) {
      if ( ! obj instanceOf X )
         return false;
      X x = (X)obj;
      return i == x.i && d == x.d && c == x.c;
   }
   .........
}
```

Wir schauen uns das beispielhaft mit den beiden Methoden equals und toString an der einfachen selbstdefinierten Klasse X an, die wir schon öfters zur Illustration hergenommen haben. Zuerst Methode equals.

***Achtung*: In professioneller Softwareentwicklung müsste unbedingt noch eine weitere Methode namens hashCode überschrieben werden, die eng mit equals zusammenhängt. Da das Thema Hashfunktionen erst in der AuD drankommt, lassen wir diesen Schritt hier unprofessionellerweise aus.**

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i;
   private double d;
   private char c;
   .........
   public boolean equals ( Object obj ) {
      if ( ! obj instanceOf X )
         return false;
      X x = (X)obj;
      return i == x.i && d == x.d && c == x.c;
   }
   .........
}
```

Der statische Typ des Parameters ist Object, was soll er auch sonst sein? Wenn der dynamische Typ nicht X oder ein Subtyp von X ist, sind wir schon fertig, denn dann kann die Antwort nur false sein.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i;
   private double d;
   private char c;
   .........
   public boolean equals ( Object obj ) {
      if ( ! obj instanceOf X )
         return false;
      X x = (X)obj;
      return i == x.i && d == x.d && c == x.c;
   }
   .........
}
```

An dieser Stelle wissen wir, dass der dynamische Typ von Parameter obj entweder gleich X oder ein Subtyp von X ist. In diesem Fall können wir den Downcast anwenden und so an die Werte der drei Attribute herankommen.

Sie sehen also, dass Wertidentität hier durchaus gegeben ist, wenn der dynamische Typ von obj direkt oder indirekt abgeleitet von X ist, sofern nur die drei Attribute von X in beiden miteinander verglichenen Objekten wertgleich sind. Man kann darüber streiten, ob das sinnvoll ist, oder ob man Wertgleichheit nicht eher nur annimmt, wenn die beiden Objekte auch wirklich denselben Typ haben. Wir haben es hier jedenfalls beispielhaft so und nicht anders implementiert.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i;
   private double d;
   private char c;
   .........
   public String toString () {
      return “Class X: ( i = “ + x.i + “, d = “ + x.d + “, c = “ + x.c + “ )“;
   }
   .........
}
```

Nun ist noch die Methode toString für Klasse X zu überschreiben. Diese Methode hat die Aufgabe, ein Objekt dieser Klasse so textuell darzustellen, dass das Wesentliche leicht durch einen Menschen erfasst werden kann.

Klasse java.lang.Object

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private int i;
   private double d;
   private char c;
   .........
   public String toString () {
      return "Class X: ( i = " + x.i + ", d = " + x.d + ", c = " + x.c + " )";
   }
   .........
}
```

→ Class X: (i = 1, d = 3.14, c = z)

So könnte etwa eine sinnvolle textuelle Repräsentation aussehen: Sie enthält den Namen der Klasse und die Werte aller Attribute, letztere in einer vertrauten und daher hoffentlich sofort verständlichen Form: in Klammern und durch Kommas separiert.

***Erinnerung*: Addition bei Strings hatten wir weiter vorne in diesem Kapitel gesehen, im Abschnitt zur Klasse String.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Garbage Collector

Ein Objekt kann sozusagen verlorengehen, nämlich dadurch, dass seine Adresse in keiner Referenz mehr gespeichert ist. Das ist auch sehr häufig so gewünscht, nämlich immer wenn man ein Objekt nicht mehr braucht.

Das Problem ist, dass der Speicherplatz für das Objekt weiterhin reserviert bleibt. Dieses Problem und seine Lösung des Problems in Java sehen wir uns als nächstes an.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

```
while ( true )
    X x = new X();
```

Um die Brisanz des Problems klarer zu sehen, treiben wir es ins Extrem, indem wir immer neue Objekte in einer Endlosschleife einrichten und jeweils die Adresse sofort wieder verloren geht.

Es werden also ohne Ende neue Objekte eingerichtet, jeweils in dem Speicherplatz, der noch nicht von den bisher eingerichteten Objekten belegt ist. Da der Speicherplatz natürlich nur endlich groß ist, ist irgendwann kein Speicherplatz mehr da; das Programm bricht mit einem Fehler ab.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil des Laufzeitsystems
- Wird nach gewissen Regeln hin und wieder aufgerufen
  - Kann zwecks Laufzeitoptimierung konfiguriert werden
- Gibt alle Speicherplätze frei, die vom Programm aus nicht mehr erreichbar sind

Aber ganz so ist es zum Glück nicht. Genau dafür gibt es den *Garbage Collector*, zu deutsch Müllsammler.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil des Laufzeitsystems
- Wird nach gewissen Regeln hin und wieder aufgerufen
  - Kann zwecks Laufzeitoptimierung konfiguriert werden
- Gibt alle Speicherplätze frei, die vom Programm aus nicht mehr erreichbar sind

Der Garbage Collector ist ein Bestandteil des Laufzeitsystems von Java und leistet uns damit implizit seine Dienste, für die wir ihm dankbar sein müssen.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil des Laufzeitsystems
- Wird nach gewissen Regeln hin und wieder aufgerufen
  - ➢ Kann zwecks Laufzeitoptimierung konfiguriert werden
- Gibt alle Speicherplätze frei, die vom Programm aus nicht mehr erreichbar sind

Spätestens wenn der Speicherplatz knapp wird, wird der Garbage Collector durch das Laufzeitsystem zum Leben erweckt, vielleicht auch schon früher, damit es gar nicht erst soweit kommt.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil des Laufzeitsystems
- Wird nach gewissen Regeln hin und wieder aufgerufen
  - ➢ Kann zwecks Laufzeitoptimierung konfiguriert werden
- Gibt alle Speicherplätze frei, die vom Programm aus nicht mehr erreichbar sind

Wann genau das passiert, kann man beeinflussen. Das ist eine der wesentlichen Stellschrauben, um die Laufzeit eines Java-Programms zu verbessern. Mehr dazu im Kapitel zu effizienter Software.

Garbage Collector

TECHNISCHE UNIVERSITÄT DARMSTADT

- Teil des Laufzeitsystems
- Wird nach gewissen Regeln hin und wieder aufgerufen
  - ➢ Kann zwecks Laufzeitoptimierung konfiguriert werden
- Gibt alle Speicherplätze frei, die vom Programm aus nicht mehr erreichbar sind

Was bedeutet nun Müllsammeln konkret? Der Müll, der aufgesammelt wird, das sind genau die Objekte, um die es hier geht, nämlich die Objekte, deren Speicherplatz immer noch reserviert ist, obwohl sie vom Programm aus nicht mehr angesprochen werden können und daher absolut nutzlos geworden sind. Der freigegebene Platz kann für die Einrichtung neuer Objekte wiederverwendet werden.

Mehr brauchen wir hier über den Garbage Collector nicht zu wissen, und daher beenden wir das Thema und damit das ganze Kapitel.