4.4.1 Vererbung

Erinnerung an KarelJ:

- Eine Klasse kann eine andere erweitern.
- Die erstere Klasse kann dann die Funktionalität der zweiten Klasse quasi ”gratis“ mitbenutzen.
- Dass eine Klasse eine Erweiterung darstellt, wird mit Schlüsselwort `extends` angezeigt:

```
public class MeinApplet extends Applet ...
```

- To extend = erweitern

Vererbung von Datenkomponenten

- *Terminologie*:
  - ◊ Ein ” Urstamm“ von Erweiterungen heißt meist *Basisklasse*.
  - ◊ Von einer ”Erweiterung“ einer Basisklasse sagen wir, sie sei von der Basisklasse *abgeleitet*.
- Wenn eine Klasse von einer anderen (Basis-)Klasse abgeleitet wird, dann ”erbt“ sie deren Datenkomponenten.
  - Methoden werden im Prinzip genauso vererbt (kommt noch).
- Eine abgeleitete Klasse darf natürlich ihrerseits als Basisklasse für weitere abgeleitete Klassen fungieren.

→ Die Klassen in der Java–Standardbibliothek sind in vielen, durchaus langen, sich verzweigenden Ketten von gegenseitigen Vererbungsbeziehungen organisiert.

Beispiel

```
public class BasisKlasse {
   public int n1;
}

class AbgeleiteteKlasse extends BasisKlasse {
   public int n2;
}

...

BasisKlasse x1 = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();

System.out.println ( x1.n1 );
System.out.println ( x2.n2 );
System.out.println ( x2.n1 );
```

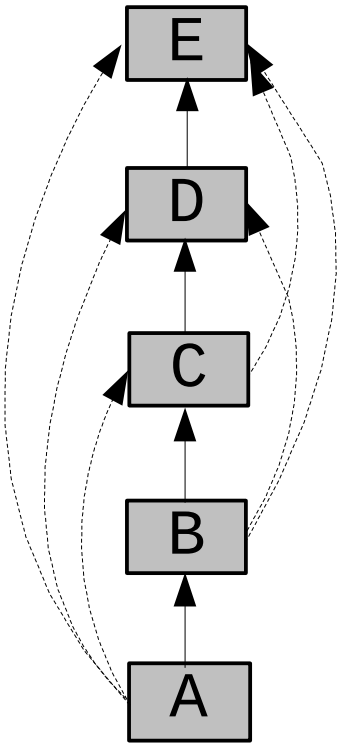
`AbgeleiteteKlasse` erbt die Komponente n1 von `Basisklasse`

die Komponente n1 kann daher auch von Objekten der abgeleiteten Klasse verwendet werden.

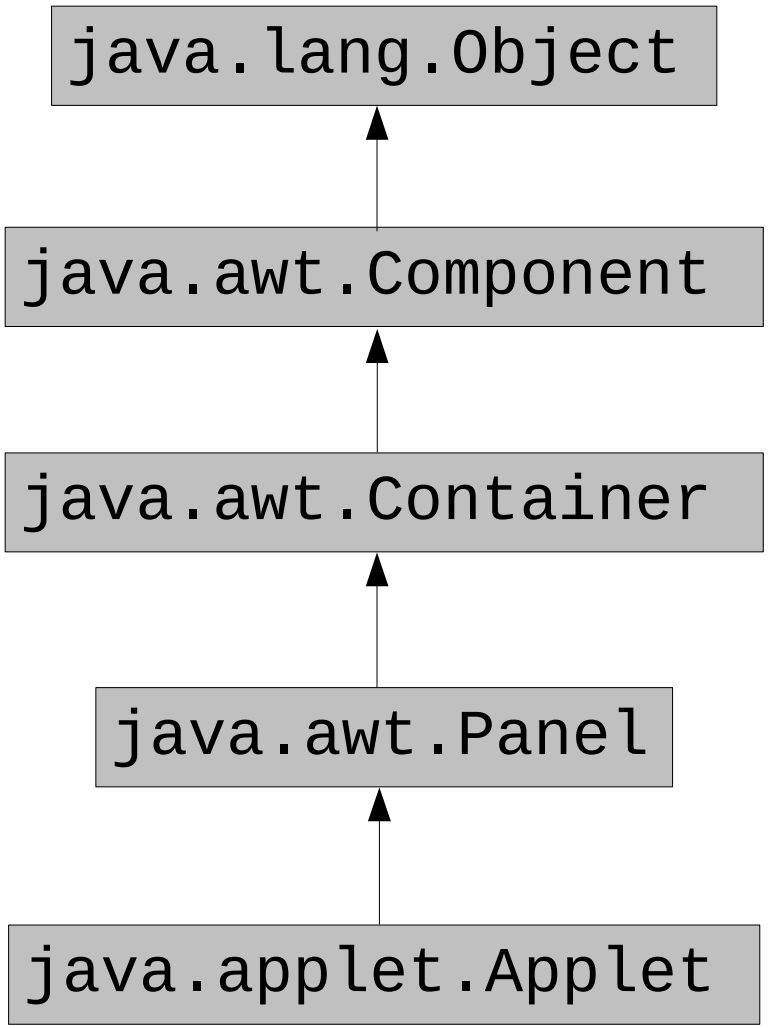
Ketten von Vererbungsbeziehungen

- Damit ist gemeint, dass eine Klasse A von einer Klasse B abgeleitet ist, B wiederum von einer Klasse C, C von einer Klasse D usw.
- Wir sagen, A ist von B, B von C usw. *direkt* abgeleitet.
- Daraus ergeben sich auch *indirekte* Vererbungsbeziehungen:
  - A ist von C,
  - A ist von D,
  - A ist von E,
  - B ist von D,
  - B ist von E,
  - C ist von E

indirekt abgeleitet.



Konkretes Beispiel aus der Java Standard-Bibliothek



Vererbung von Methoden

```
public class BasisKlasse
{
   public void f1 () {
      System.out.println ( "f1" );
    }
}

class AbgeleiteteKlasse extends BasisKlasse
{
   public void f2 () {
      System.out.println ( "f2" );
   }
}

BasisKlasse x1 = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();

x1.f1 ();
x2.f2 ();
x2.f1 ();
```

Die Methode `f1` wurde von der Basis-Klasse ererbt und kann daher auch von Objekten der abgeleiteten Klasse verwendet werden.

Beispiel

- In der Klasse `java.applet.Applet` ist alles implementiert, was man braucht, um ein Fenster zu öffnen und zu verwalten.
- Wenn man eine eigene Klasse von `java.applet.Applet` ableitet, dann erbt die eigene Klasse alle diese Funktionalität von `Applet` automatisch.
- Man braucht sich also um solche wirklich (wirklich!) lästigen technischen Details nicht zu kümmern.
- Statt dessen kann man sich voll und ganz auf die Logik der eigenen Applet–Klasse kümmern:
  - Bisher hat uns bei der Logik von Applets nur die Methode `paint` interessiert.
  - Andere Methoden wie `init` und `repaint` werden auch geerbt, können aber ebenfalls interessant sein.

Regeln zur Vererbung von Methoden

- Jede Methode der Basisklasse ist automatisch auch eine Methode der abgeleiteten Klasse:
  - ◊ exakt dieselbe Signatur (natürlich abgesehen vom Klassennamen),
  - ◊ exakt dieselbe Implementation.
- Einzige Ausnahme: Konstruktoren werden nicht in diesem Sinne vererbt.

→ Sind meist zu eng von ihrer Logik her mit ”ihrer“ Klasse verbunden.

4.4.2 Überschreiben von Methoden

- Wenn eine Methode in der Basisklasse vorhanden ist, ist sie auf jeden Fall auch in der abgeleiteten Klasse vorhanden.

- Es gibt aber zwei Möglichkeiten, wie sie in der abgeleiteten Klasse vorhanden sein kann:
  - ◊ von der Basisklasse vererbt (wie bisher betrachtet);
  - ◊ in der abgeleiteten Klasse *überschrieben*.

- Was heißt eine Methode überschreiben:

  Eine Methode mit exakt identischer Signatur(!) wie in der Basisklasse wird in der abgeleiteten Klasse noch einmal definiert.

Überschreiben vs. Überladen

- Überschrieben werden Methoden nur, wenn in der abgeleiteten Klasse die exakt gleiche Signatur verwendet wird.
- Es ist durchaus auch möglich, den Namen einer Methode, die in der Basisklasse schon definiert wurde, in der abgeleiteten Klasse für eine weitere Methode *mit anderer Signatur* zu verwenden.
- Das ist dann aber keine Vererbung, sondern nur ein Fall von Überladung:
  - Eben eine neue Methode,
  - die halt ”zufällig“ genauso heißt.
- *Einfache allgemeine Regel*:

  Wenn zwei Methoden gleichen Namens unterschiedliche Signatur haben, handelt es sich *immer* um Überladung.

  → Die beiden Parameterlisten müssen wie immer unterschiedlich sein.

Beispiel

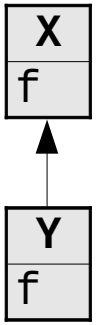
```
public class BasisKlasse
{
    public void f ( int i ) {
        System.out.println ( "1" );
    }
}

public class AbgeleiteteKlasse extends BasisKlasse
{
    public void f ( int j ) {
        System.out.println ( "2" );
    }

    public void f ( double d ) {
        System.out.println ( "3" );
    }
}

BasisKlasse x = new BasisKlasse();
AbgeleiteteKlasse y = new AbgeleiteteKlasse();

x.f (1);      // -> "1"
y.f (1);      // -> "2"
y.f (3.14);   // -> "3"
x.f (3.14);   // Fehler: BasisKlasse.f(double) existiert nicht!
```

Originalmethode

Überschreibung

Überlagerung

Schlüsselwort **super**



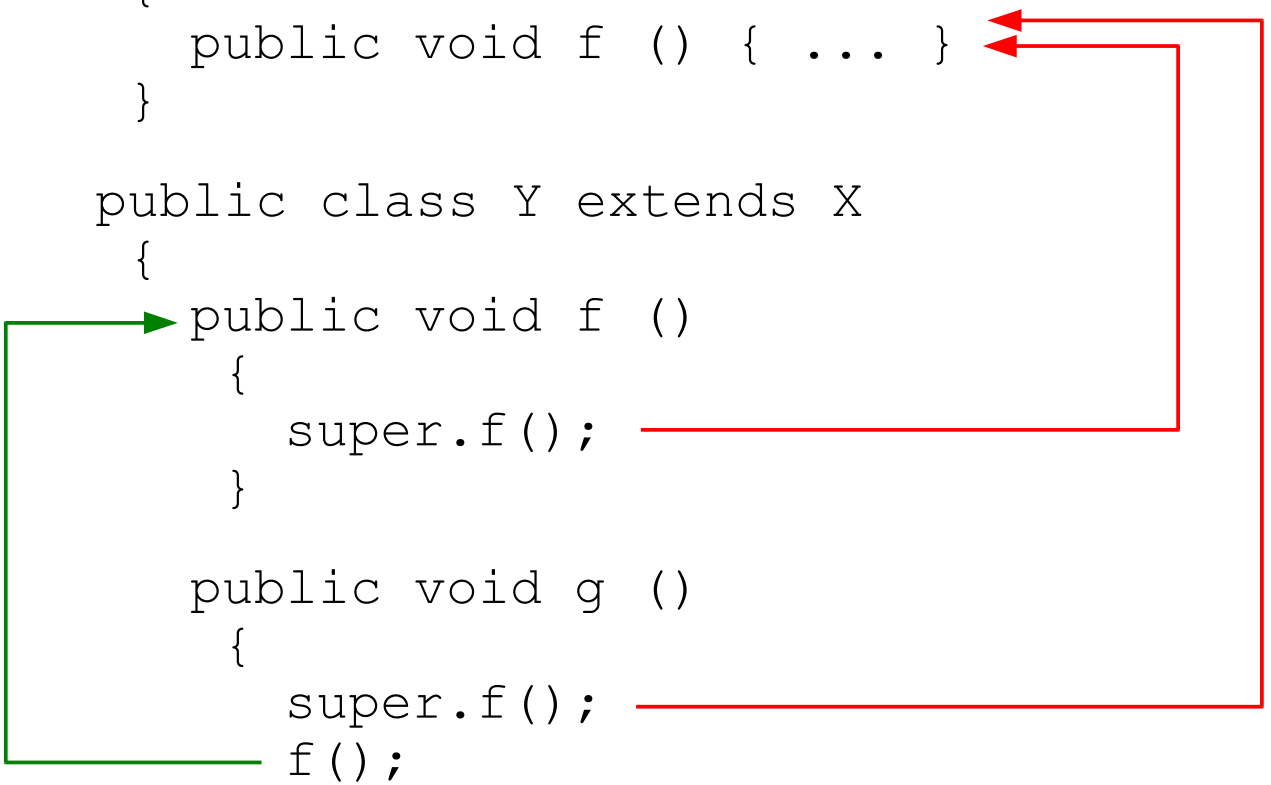
- Betrachte wieder die allgemeine Situation, dass eine Methode `f` in einer Basis-Klasse `X` definiert und in einer von `X` abgeleiteten Klasse `Y` überschrieben ist.
- In dieser Methode von `Y` oder auch in anderen Methoden von `Y` möchte man aber vielleicht auch gerne `X.f` benutzen.
- Aber durch die Überschreibung ist in den Methoden von `Y` zunächst einmal nur `Y.f` ansprechbar.

  → Ähnlich wie bei `this`
- Mit Schlüsselwort `super` kann man die Methoden der Basisklasse in den Methoden der abgeleiteten Klasse aufrufen.

  → Egal ob in der abgeleiteten Klasse überschrieben oder nicht.
- Die Syntax ist wie bei `this`.
- Geschachtelte Aufrufe von Methoden indirekter Basisklassen über `super.super`, `super.super.super` usw. sind nicht erlaubt.

Beispiel

```
public class X
  {
    public void f () { ... }
  }

public class Y extends X
  {
    public void f ()
      {
        super.f();
      }

    public void g ()
      {
        super.f();
        f();
      }
  }
```



Konstruktoren der Basisklasse

- *Erinnerung*: Konstruktoren werden nicht vererbt.
- Trotzdem möchte man im Konstruktor der abgeleiteten Klasse oft auch den Konstruktor der Basisklasse aufrufen, um den Anteil der Basisklasse am Gesamtobjekt mit einem dafür schon vorgesehenen Konstruktor zu initialisieren.
- Mit Schlüsselwort `super` kann man einen Konstruktor der Basisklasse in einem Konstruktor der abgeleiteten Klasse aufrufen.
- Verwendung analog zur Verwendung von `this`

Verwendung von **super**

- Nur die allererste Zeile in einem Kontruktor darf der Aufruf eines anderen Konstruktors mit `this` oder `super` sein.

  → Insbesondere darf es in jedem Konstruktor nur einen einzigen Aufruf eines anderen Konstruktors geben.

- Auch hier wird dafür gesorgt, dass immer ein Konstruktor aufgerufen werden muss:

  ◊ Wenn die direkte Basisklasse keinen Konstruktor mit leerer Parameterliste hat (weder selbst geschrieben noch vom Compiler hinzugedacht), dann muss ein Konstruktor mit `super` aufgerufen werden (mit entprechenden Parametern).

  ◊ Wenn die direkte Basisklasse einer Klasse `X` einen Konstruktor mit leerer Parameterliste hat und die erste Zeile eines Konstruktors von `X` kein Konstruktoraufruf ist, wird der leere Konstruktor der Basisklasse implizit am Anfang aufgerufen.

Beispiel

```
public class Auto {
   public Auto() { ... }
 }

public class Kombi extends Auto {
   public Kombi()
    {
      super(); // OK!
      ...
    }
 }

public class Cabrio extends Auto {
   public Cabrio()
    {
      // irgendwelche Variablen werden
      // initialisiert
      ...
      super(); // -> Fehler!
    }
 }
```

4.4.3 Polymorphie

- Eine Variable vom Typ der Basisklasse kann also auch auf ein Objekt vom Typ einer davon abgeleiteten Klasse verweisen:

```
AbgeleiteteKlasse x = new AbgeleiteteKlasse();
BasisKlasse y = x;
```

- Diese beiden Anweisungen lassen sich selbstverständlich auch zu einer Anweisung zusammenfassen:

```
BasisKlasse x = new AbgeleiteteKlasse();
```

- Das macht Sinn, da Objekte der abgeleiteten Klasse alle Fähigkeiten haben, die die Objekte der Basisklasse haben.

- Das heißt: alle Methodenaufrufe, die für Objekte der Basis-Klasse definiert sind, sind auch für Objekte der abgeleiteten Klasse definiert.

Beispiel

```
public class BasisKlasse {
    public void f () {
       System.out.println ( "Basisklasse" );
     }
  }

 public class AbgeleiteteKlasse extends BasisKlasse {
    public void f () {
       System.out.println ( "Abgeleitete Klasse" );
     }
  }
 ....

 BasisKlasse x1          = new BasisKlasse();
 AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();
 BasisKlasse x3          = new AbgeleiteteKlasse();

 x1.f (); // -> "BasisKlasse.f()"
 x2.f (); // -> "AbgeleiteteKlasse.f()"
 x3.f (); // -> "AbgeleiteteKlasse.f()" !!!
```

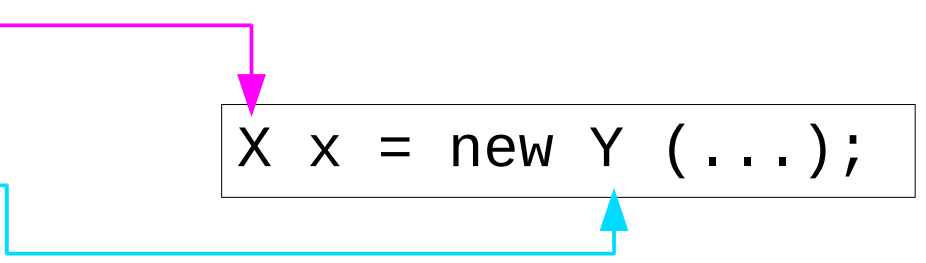
Statischer (Formaler) und Dynamischer (Aktualer) Typ

- Wegen solcher Möglichkeiten der Form

```
X x = new Y (...);
```

  hat das Objekt, auf das die Referenz x zeigt, also nicht unbedingt denselben Typ bei der Deklaration angegeben.

- Bei einer Variablen wie x, die von einem Klassentyp ist, müssen wir daher sorgfältig zwischen zwei Typen unterscheiden:
  - ◊ *statischer (formaler)* Typ: der Typ der *Variablen* (also X)
  - ◊ *dynamischer (aktualer)* Typ: dem Typ des *Objekts* (also Y).



- Bei einer Zeile der Form

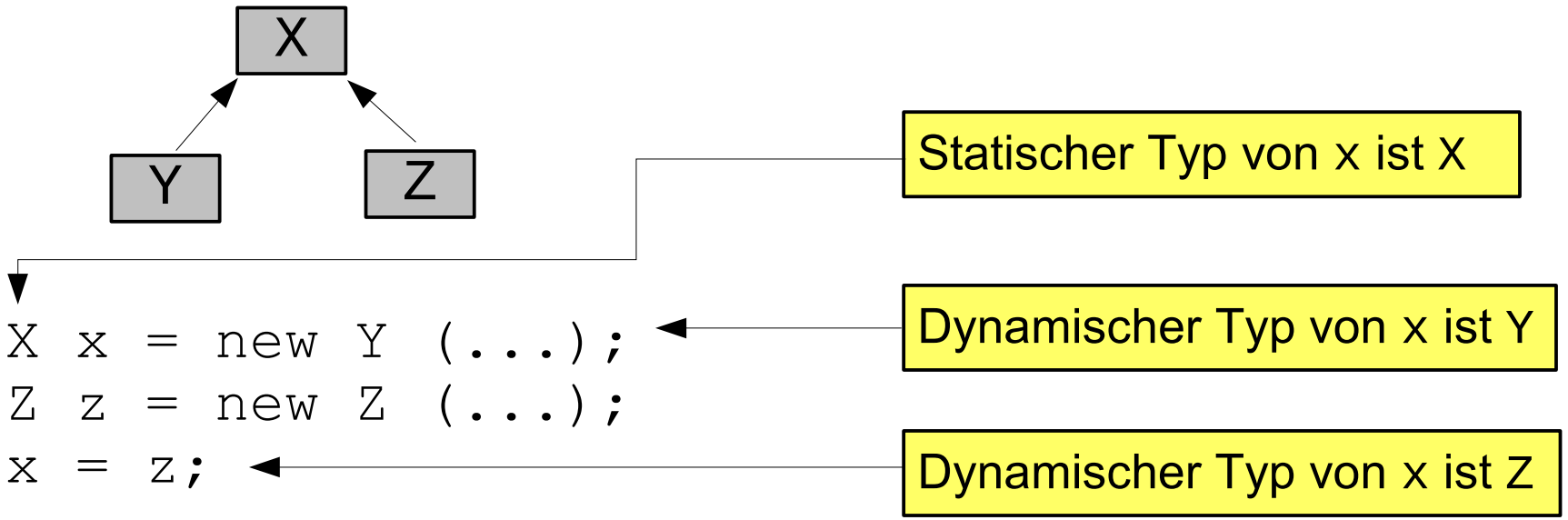
```
X x = new X (...);
```

  sind dynamischer Typ und statischer Typ von x identisch.

Statisch und Dynamisch

- Der statische Typ einer Variablen bleibt immer gleich.
- Der dynamische Typ kann sich jederzeit ändern.

→ Einfach durch Zuweisung eines neuen Objekts an die Variable wird der Typ des neuen Objekts der neue aktuale Typ:



Beziehung Statischer und Dynamischer Typ

- Es gibt nur zwei Möglichkeiten für eine Variable von einem Klassentyp:
  - Der statische und der dynamische Typ sind identisch.
  - Der dynamische Typ eine direkte oder indirekte Unterklasse von statischen Typ
- *Warum* diese Einschränkung:
  - Der statische Typ fungiert als ”Fassade“, hinter dem Objekte von allen möglichen dynamischen Typen stecken können.
  - Dazu ist es natürlich notwendig, dass der dynamische Typ alles kann, was der statische Typ ”verspricht“.
  - Das ist am einfachsten und elegantesten eben durch diese Einschränkung gewährleistet.

Rollenverteilung

- Welche Methoden mit einer Variablen eines Klassentyps aufgerufen werden dürfen, hängt von seinem *statischen* Typ ab.
- Falls eine Methode für Basisklasse und abgeleitete Klasse(n) implementiert ist:

  → Dann wird die Implementation des dynamischen Typs aufgerufen.

- Falls eine Methode nur für die Basisklasse, aber nicht für die abgeleitete Klasse implementiert ist:

  Dann wird die Implementation der Basisklasse auf die abgeleitete Klasse vererbt.

  → Dann ist die Implementation der Basisklasse in jedem Fall auch die Implementation des dynamischen Typs.

Beispiel

```
public class BasisKlasse {
   public void f1 ()
    {
      System.out.println ( "B.f1" );
    }
 }

public class AbgeleiteteKlasse extends BasisKlasse {
   public void f1 () {      // Ueberschrieben
      System.out.println ( "A.f1" );
    }
   public void f2 () {      // Neu in abgeleiteter Klasse
      System.out.println ( "A.f2" );
    }
 }
...

BasisKlasse x = new AbgeleiteteKlasse();

x.f1();
x.f2();
```

Verboten: `f2` ist zwar für diesen speziellen dynamischen Typ (`AbgeleiteteKlasse`) definiert, aber nicht für den statischen Typ (`BasisKlasse`)

Anderes Beispiel

- Wo immer eigentlich ein `Applet`–Objekt erwartet wird (zum Beispiel auch im Interpreter `appletviewer`)
- kann statt dessen auch ein Objekt einer davon abgeleiteten Klasse verwendet werden,
- und soweit `Applet`–Methoden wie `paint` oder `init` oder `repaint` in der abgeleiteten Klasse implementiert sind (mit identischer Signatur),
- werden diese Implementationen verwendet.
- `appletviewer` und co. bekommen also Variablen vom statischen Typ **`Applet`**,
- dahinter kann aber auch ein Objekt vom dynamischen Typ **`MeinApplet`** stehen (`MeinApplet` wurde von `Applet` abgeleitet).

Kann man der Compiler nicht den dynamischen Typ erkennen?

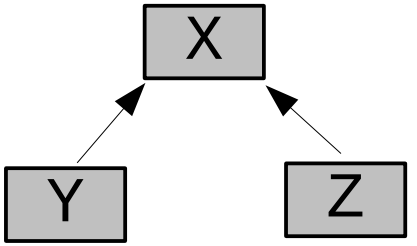
- Warum dürfen denn eigentlich nur die Methoden aufgerufen werden, die für den *statischen* Typ schon definiert sind?
- Der Compiler kann doch den dynamischen Typ ”erkennen“ und dementsprechend auch alle Methoden erlauben, die erst für den *dynamischen* Typ definiert sind, oder?

Einfache Antwort: Nein!

- Es stimmt leider nicht immer, dass der Compiler den dynamischen Typ einer Variablen erkennen kann.
- Der dynamische Typ einer Variablen kann von Informationen abhängen, die erst zur Laufzeit des Programms zur Verfügung stehen (und sich auch von Programmlauf zu Programmlauf ändern können).

Beispiel

```
public class X { ... }

public class Y extends X { ... }

public class Z extends X { ... }
...

boolean b;

...

X x;
if ( b )
   x = new Y ();
else
   x = new Z ();
```



Der Wert von b kann von verschieden Faktoren abhängen, z.B. von Benutzereingaben

Der dynamische Typ von x kann daher von Programmlauf zu Programmlauf variieren und daher prinzipiell nicht vom Compiler erkannt werden.

Down-Cast

- *Erinnerung*:
  - ◊ Unsichere Konversionen zwischen eingebauten Typen sind möglich.
  - ◊ Man muss den Zieltyp dann in runden Klammern vor den zu konvertierenden Ausdruck schreiben.
    z.B. `float pi = (float) 3.14159;`
- Man kann auch von einer Basisklasse auf eine abgeleitete Klasse ”herunter“ konvertieren (engl. *down–cast*).
- Dies ist ebenfalls eine unsichere Konversion:
  - ◊ Wenn der dynamische Typ des Objekts gleich dem Zieltyp oder vom Zieltyp abgeleitet ist, geht alles gut.
  - ◊ Wenn nicht: Programmabsturz, da das Objekt nicht die Anforderungen erfüllt, die der Zieltyp haben muß.
- Da diese Art von Konversion generell unsicher ist, muss man eben wieder den Zieltyp in runden Klammern vor die zu konvertierende Variable schreiben.

Beispiel

```
public class BasisKlasse { ... }

public class AbgeleiteteKlasse1 extends BasisKlasse
 {
   public void f () { ... }
 }

public class AbgeleiteteKlasse2 extends BasisKlasse { ... }

...

BasisKlasse x = new AbgeleiteteKlasse1 ();
x.f();

AbgeleiteteKlasse1 y = (AbgeleiteteKlasse1) x;
y.f();

AbgeleiteteKlasse2 z = (AbgeleiteteKlasse2) x;
```

Statischer Typ: `BasisKlasse`
Dynamischer Typ: `AbgeleiteteKlasse1`

nicht o.k., da x vom Typ `BasisKlasse`

Down-Cast

o.k., da y vom Typ `Abgeleitetelasse1`

Fehler: Down-Cast auf anderen Typ nicht möglich

Erläuterungen

- Der Quelltext lässt sich nach Entfernen von `x.f();` problemlos kompilieren.
- Aber beim Laufenlassen stürzt das Programm in der letzten Zeile, in der z eingerichtet wird, ab mit der Fehlermeldung:

```
java.lang.ClassCastException
```

- Die Zeile, in der y eingerichtet wird, ist absolut korrekt:
  - ◊ Die Variable y verweist hinterher auf dasselbe Objekt wie x.
  - ◊ Der dynamische und der statische Typ von y sind dann gleich.

Allgemeine Regel für Down-Casts

Ein Down–Cast

```
AbgeleiteteKlasse x = (AbgeleiteteKlasse) y;
```

ist genau dann ok, wenn der dynamische Typ von `y`

- entweder `AbgeleiteteKlasse` selbst oder
- direkt oder indirekt von `AbgeleiteteKlasse` abgeleitet

ist.

Schlüsselwort `instanceof`

- Bevor man einen unsicheren Down–Cast macht — und dadurch einen Programmabsturz riskiert — kann man in Java abprüfen, ob der Down–Cast ok ist.
- Dafür gibt es in Java das Schlüsselwort `instanceof`.
- *Gebrauch*: Der logische Ausdruck

  ```
  x instanceof Y
  ```

  liefert `true` genau dann, wenn der dynamische Typ von `x` entweder gleich `Y` ist oder direkt oder indirekt von `Y` abgeleitet ist.
- Mit anderen Worten: genau dann, wenn der Down–Cast ok ist.

```
Y y;
if ( x instanceof Y )
    y = (Y)x;
else
    System.out.println ( "Falscher Typ!" );
```

Was ist nun Polymorphie?

- Name:
  - ”Poly“ ≈ viel
  - ”Morphe“ ≈ Gestalt
- *Idee*:
  - Eine Variable der Basisklasse (z.B. `Applet`) fungiert als eine einheitliche Fassade.
  - Dahinter können sich Objekte von unterschiedlichen (abgeleiteten) Klassentypen verbergen (zum Beispiel `MeinApplet1, MeinApplet2,` etc.).
  - Dieselbe Methode kann nun in den verschiedenen abgeleiteten Klassen unterschiedlich implementiert sein (z.B. `MeinApplet1.paint, MeinApplet2.paint,` etc.).

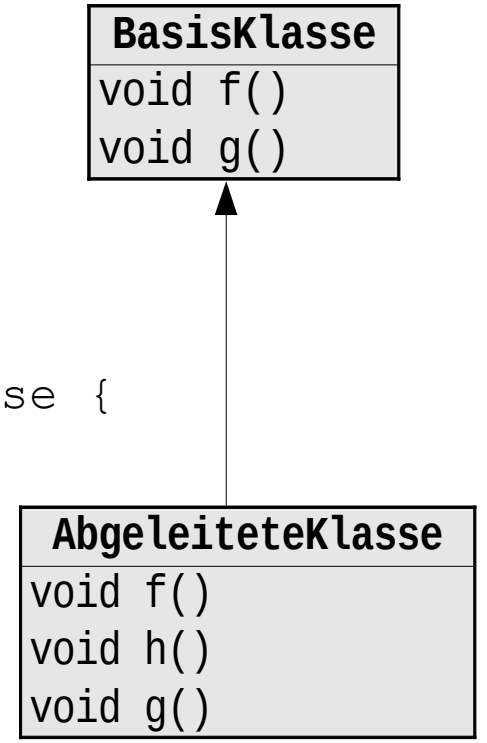
  → Ein Methodenaufruf vermittels dieser Fassade kann je nach Typ des dahinterstehenden Objekts völlig unterschiedliche Effekte erzeugen.

4.4.4 Interne Realisierung

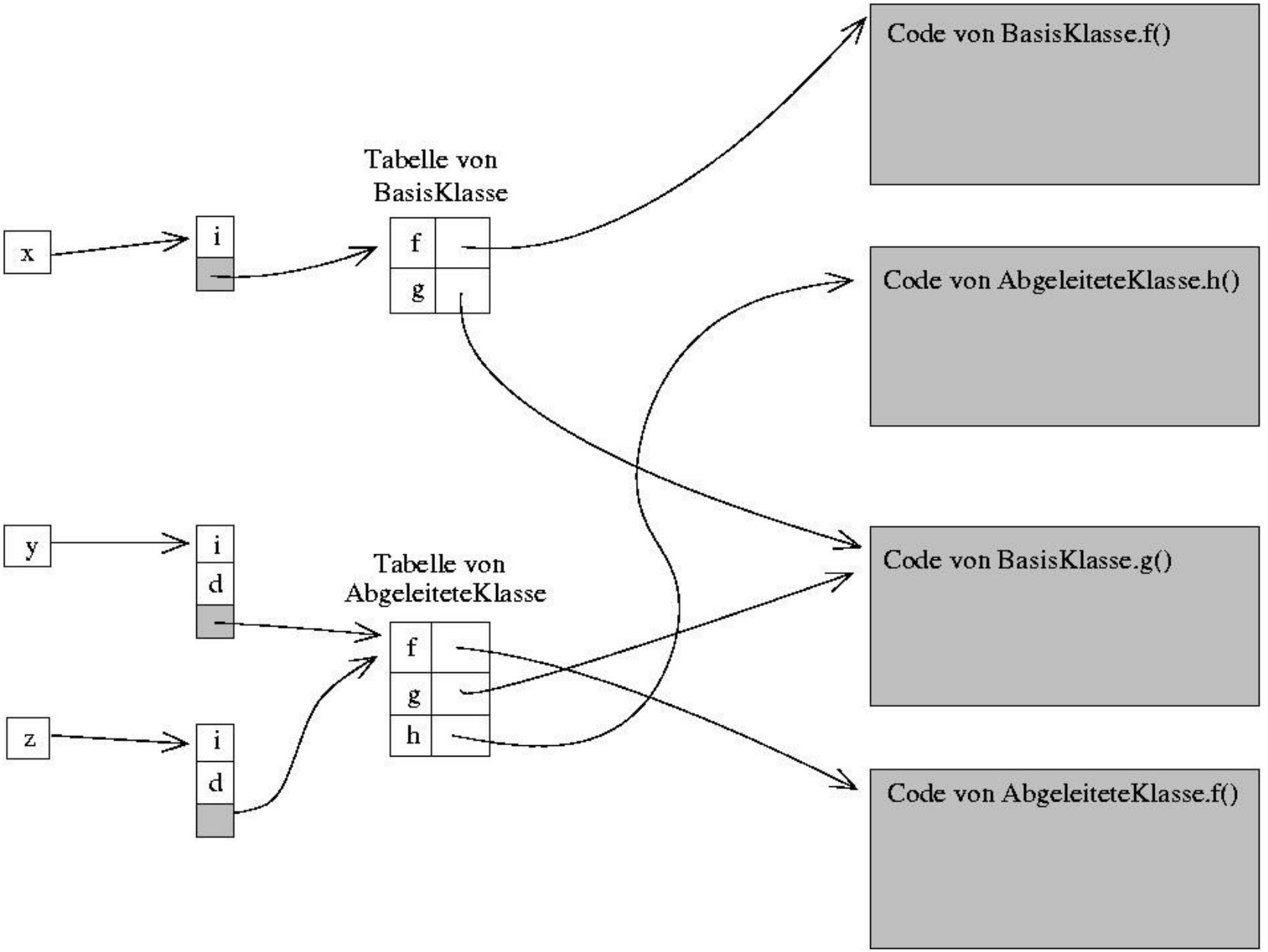
- Jedes Objekt einer Klasse enthält eine weitere, ”unsichtbare“ Datenkomponente.
- In dieser Datenkomponente ist kodiert, von welchem Typ das Objekt ist.
- Zu jeder Klasse wird separat irgendwo im Speicher einmal eine Tabelle angelegt, in der zu jeder *Objekt*methode der Klasse die Startadresse gespeichert ist.
- Falls eine Objektmethode in der Klasse nicht selbst implementiert ist, sondern von der Basisklasse vererbt wird, dann wird die Startadresse der ererbten Methode dort gespeichert.

Beispiel

```
public class BasisKlasse {
    public int i;

    public void f () {
        System.out.println ( "1" );
      }

    public void g () {
        System.out.println ( "2" );
      }
  }

public class AbgeleiteteKlasse extends BasisKlasse {
    public double d;

     public void f () {
         System.out.println ( "3" );
       }

     public void h () {
         System.out.println ( "4" );
       }
  }
 ...

BasisKlasse        x = new BasisKlasse();
BasisKlasse        y = new AbgeleiteteKlasse();
AbgeleiteteKlasse  z = new AbgeleiteteKlasse();
```



Schematische Darstellung



Aufruf einer Methode

Jeder Aufruf einer Objektmethode im Java–Quelltext wird vom Compiler in Code übersetzt, der

- zuerst die zusätzliche Datenkomponente mit der Typinformation liest,
- damit die Speicheradresse der zu diesem Klassentyp zugehörigen Tabelle von Methodenadressen heraussucht,
- in dieser Tabelle dann den Eintrag der aufgerufenen Methode nachschlägt und
- einen Methodenaufruf mit der dort gefundenen Startadresse ausführt.

→ Erst im letzten dieser vier Schritte werden die Argumente des und die Rücksprungadresse auf den Run-Time-Stack gelegt

Veranstalter:
Fachbereich Informatik
FG Metamodellierung

TUD Programming Contest 2005

Vorrunde 17.6.
Testlauf 24.6.

Wettbewerb

25.6.

Studierende aller Fachbereiche können teilnehmen!

KOM

danet GmbH

SOFTWARE AG
THE XML COMPANY

Information & Anmeldung

http://tud-pc.informatik.tu-darmstadt.de

4.4.5 Vererbung und Zugriffsrechte

- Eigentlich können die Methoden einer Klasse alle Datenkomponenten und Methoden dieser Klasse benutzen.

- *Nun eine Ausnahme*: Die Methoden einer abgeleiteten Klasse dürfen `private`–deklarierte Datenkomponenten und Methoden, die sie von einer Basisklasse ererbt haben, nicht verwenden.

- Mit `private` wird also genauer gesagt der Zugriff erlaubt
  - ◊ nur für die eine Klasse, in der die Datenkomponente bzw. Methode eingeführt wurde,
  - ◊ und nicht auch für alle davon abgeleiteten Klassen, in denen dieselbe Datenkomponente bzw. Methode (ererbt) ebenfalls vorkommt.

Zugriffsrechte `public` und `private`

(→Folie 467)

- Eine Datenkomponente oder Methode
  - ◊ darf in den Methoden jeder beliebigen Klasse angesprochen werden, wenn sie durch `public` gekennzeichnet ist,
  - ◊ nur in den Methoden derselben Klasse, wenn sie durch `private` gekennzeichnet ist, und
  - ◊ in den Methoden aller Klassen desselben Packages, wenn sie gar nicht gekennzeichnet ist.

Beispiel

```
public class MeineKlasse
 {
   public  int n1;
   private int n2;

   public  void f1 () { ... }
   private void f2 () { ... }
 }
```

```
// Verwendung in einer anderen Klasse:
MeineKlasse meinObjekt = new MeineKlasse();

meinObjekt.n1 = 1;    // Erlaubt   (n1 ist public)
meinObjekt.n2 = 1;    // Verboten  (n2 ist private)
meinObjekt.f1();      // Erlaubt   (f1 ist public)
meinObjekt.f2();      // Verboten  (f2 ist private)
```

Zugriffsrecht `protected`

Wird eine Datenkomponente oder Methode einer Klasse `X`

- nicht `public` oder `private`,
- sondern `protected` deklariert,

dann kann sie verwendet werden

- nicht nur von den Methoden von `X` selbst,
- sondern auch von den Methoden aller direkt und indirekt von `X` abgeleiteten Klassen,
- aber nicht von den Methoden irgendeiner anderen Klasse.

Überblick Zugriffsrechte

Methode bzw. Datenkomponente...

...deklariert als

...sichtbar in	`public`	`protected`	Standard	`private`
Selbe Klasse	ja	ja	ja	ja
Selbes Paket	ja	ja	ja	nein
Unterklasse	ja	ja	nein	nein
Beliebige Klasse	ja	nein	nein	nein

Beispiel

```
public class A {
       private   int i;
       protected int j;
       public    int k;
}

public class B {
    public void f () {
       A x = new A();
       System.out.print(x.i);      // Verboten (private)
       System.out.print(x.j);      // Verboten (protected, B keine Unterklasse)
       System.out.print(x.k);      // Erlaubt  (public)
    }
}

public class C extends A {
    public void f () {
       System.out.print(i);        // Verboten (private)
       System.out.print(j);        // Erlaubt  (protected, C ist Unterklasse)
       System.out.print(k);        // Erlaubt  (public)
    }
}
```

4.4.6 Abstrakte Methoden und Klassen

- Manchmal macht es überhaupt keinen Sinn, eine Methode für eine Basisklasse tatsächlich zu implementieren.
- Für solche Fälle gibt es in Java die Möglichkeit,
  - ◊ eine Methode in einer Klasse einzuführen
  - ◊ ohne sie zu implementieren.
- *Syntaktische Unterschiede*:
  - ◊ Direkt vor dem Rückgabetyp das Schlüsselwort `abstract`.
  - ◊ Der Methodenrumpf `{...}` wird einfach durch ein Semikolon hinter der Parameterliste der Methode ersetzt.
- Eine solche Methode nennt man *abstrakt*.
- Wenn eine Klasse mindestens eine abstrakte Methode hat, nennt man die Klasse ebenfalls *abstrakt*.

→ `abstract` muss auch vor `class` geschrieben werden.

Beispiel

```
public abstract class X {
   public void f () {
      System.out.println ( "Methode f in X" );
    }

   public abstract void g ();
 }

public class Y extends X {
   public void g () {
      System.out.println ( "Methode g in Y" );
    }
 }

public abstract class Z extends X {
   public void h () {
      System.out.println ( "Methode h in Z" );
   }
}
```

daher muß auch die Klasse als `abstract` deklariert werden

Methode g wird in der Basis-Klasse X nicht implementiert, daher als `abstract` deklariert

in der abgeleiteten Klasse Y wird g definiert, daher ist Y nicht abstrakt

in der abgeleiteten Klasse Z wird g auch nicht definiert, daher ist Z ebenfalls abstrakt

Abstrakte Klassen

- Von einer abstrakten Klasse X kann man
  - ◊ durchaus Variable definieren,
  - ◊ aber keine Objekte anlegen!
- Es wäre auch fatal, wenn das ginge:
  - ◊ Dann könnte X der dynamische Typ einer Variable vom Klassentyp werden.
  - ◊ Dann könnten die Methodenimplementationen von X aufgerufen werden.
  - ◊ Die existieren aber gar nicht alle!
- *Glücklicherweise*:
  - ◊ Wenn es keinen Sinn macht, alle Methoden einer Klasse zu implementieren,
  - ◊ dann macht es typischerweise auch keinen Sinn, ein Objekt dieser Klasse zu erzeugen.

Beispiel

```
abstract public class X
 {
  X () { ... }

  abstract public void f ();

  public void g () { ... }
 }
...

X x = new X();
x.f();
```

Fehlermeldung des Compilers:
X ist abstrakt!

Beispiel

- Ein Programm soll verschiedene geometrische Figuren verwalten: Kreise, Rechtecke, Quadrate, ...
- Jede dieser Objektarten muss natürlich anders gespeichert werden:
  - ◊ Kreis: Radius
  - ◊ Rechteck: lange und kurze Seite
  - ◊ Quadrat: Seitenlänge
  - ◊ ...

  → Sollte jeweils eine eigene Klasse werden.

- Aber alle diese Objektarten haben gemeinsame Eigenschaften
  - insbesondere kann man für jede den Umfang und die Fläche berechnen
- Weiters kann es auch günstig sein, Arrays von geometrischen Objekten zu definieren.

  → d.h. wir brauchen eine Basisklasse für alle diese Klassen.

Beispiel (Fs.)

- Jede dieser Klassen soll Methoden `flaeche` und `umfang` bekommen, die entsprechend definiert sind

  → Auch die Basisklasse, damit man zum Beispiel in einer Schleife für alle Elemente eines Arrays mit ”gemischten“ geometrischen Objekten Fläche und Inhalt für jedes Objekt berechnen kann.

- Natürlich muss jede dieser Methodenimplementationen an die internen Daten der jeweiligen Klasse angepasst werden.

- Daher macht es überhaupt keinen Sinn, die Methoden für die Basisklasse zu implementieren.

Beispiel-Code

```
public abstract class Figur {
   public abstract double umfang ();
   public abstract double flaeche ();
 }

public class Kreis extends Figur {


   protected double r;

   public Kreis ( double radius ) {
      this.r = radius;
    }

   public double flaeche () {
      return Math.PI * r * r;
    }

   public double umfang () {
      return 2 * Math.PI * r;
    }
 }
```

jede geometrische `Figur` muss eine `umfang` und eine `flaeche` Methode haben

ein Kreis ist eine geometrische Figur

definiert durch seinen Radius

Konstruktor für Kreise

konkrete Implementierung von `flaeche` für `Kreis`-Objekte

konkrete Implementierung von `umfang` für `Kreis`-Objekte

Beispiel-Code (Fs.)

```
public class Quadrat extends Figur {
   protected double a;

   public Quadrat ( double seite ) { a = seite; }

   public double flaeche () { return a * a; }

   public double umfang () { return 4 * a; }
}


public class Rechteck extends Figur {
   protected double l, b;

   public Rechteck ( double laenge, double breite ) {
      l = laenge;
      b = breite;
   }

   public double flaeche () { return l * b; }

   public double umfang () { return 2 * (l + b); }
}
```

konkrete Implementierungen von `flaeche` und `umfang` für `Rechteck` und `Quadrat`

Verwendung des Beispiel-Codes

Die abstrakte Klasse kann verwendet werden, um einen Array von (unterschiedlichen) geometrischen Figuren zu definieren:

```
Figur[] f = new Figur[3];

f[0] = new Kreis    ( 2.0 );
f[1] = new Rechteck( 1.0, 3.0 );
f[2] = new Quadrat ( 2.0);

double gesamtflaeche = 0;
double gesamtumfang = 0;
for ( int i=0; i < f.length; i++ ) {
   gesamtflaeche += f[i].flaeche();
   gesamtumfang  += f[i].umfang();
}
```

definiert ein Array von Figuren

die einzelnen Elemente des Arrays sind verschiedene Figuren

Berechnung von Gesamtfläche und Gesamtumfang aller Figuren des Arrays

Verboten ist aber:

```
f[0] = new Figur();
```

→ `Figur` ist eben abstrakt!

4.4.7 Interfaces und Mehrfachvererbung

- Bis jetzt haben wir
  - ◊ immer nur eine einzige Klasse hinter `extends` geschrieben,
  - ◊ das heißt, immer nur von einer einzelnen Basisklasse direkt abgeleitet.
- In vielen objektorientierten Programmiersprachen kann man eine Klasse von mehreren Basisklassen zugleich direkt ableiten.
- Aus verschiedenen (guten!) Gründen ist in Java direkte Vererbung von mehreren Klassen zugleich nicht möglich.

  → Es darf also tatsächlich auch nur der Name einer einzigen Klasse hinter `extends` stehen.
- Oft ist es aber wünschenswert, dass ein und dasselbe Objekt hinter verschiedenen ”Fassaden“ verwendet werden kann.

Lösung in Java: Interfaces

- Die Syntax von Klassen- und Interface–Deklarationen sind im Großen und Ganzen identisch.

- *Wesentliche Unterschiede*:

  ◊ Das Schlüsselwort `interface` ersetzt das Schlüsselwort `class`.

  ◊ Methoden werden nicht implementiert, sondern nach dem Methodenkopf steht nur noch ein Semikolon.

  → Wie bei abstrakten Methoden.

  ◊ Bei der Ableitung einer Klasse von einem Interface wird das Schlüsselwort `extends` durch das Schlüsselwort `implements` ersetzt.

  ◊ Alle Methoden müssen `public` sein

  ◊ Klassenmethoden sind nicht erlaubt.

  ◊ Nur konstante Datenkomponenten sind erlaubt.

Beispiel

- Wir wollen nun unsere `Figur`-Klasse erweitern, sodaß wir Figuren haben, die zeichenbar sind
  - zeichenbar heißt, daß es für jedes Objekt Methoden geben muss, die
    - die Position der Zeichnung bestimmen
    - die Farbe der Figur festlegen
    - die Figur tatsächlich zeichnen
- Mögliche Lösung:
  - Definieren einer abstrakten Klasse `ZeichenbareFigur`, die die entsprechenden Methoden definiert
  - und konkrete Unterklassen, wie `ZeichenbarerKreis`, `ZeichenbaresQuadrat`, etc., die diese Methoden definieren
- Problem:
  - Man müßte dann allerdings wieder die Methoden `umfang` und `flaeche` für jedes zeichenbare Objekt neu implementieren

Syntax von Interfaces

- Man sagt dann auch nicht, die Klasse ist vom Interface abgeleitet.
- Sondern man sagt, die Klasse *implementiert* das Interface.
  - Beispiel:

```
public class MeineKlasse implements MeinInterface
```

- Eine Klasse kann mehrere Interfaces zugleich implementieren.
  - Beispiel:

```
public class MeineKlasse
       implements MeinInterface1, MeinInterface2
```

- Aber sie darf weiterhin nur von maximal einer Klasse abgeleitet sein.
  - Beispiel:

```
public class MeineKlasse
             extends MeineBasisKlasse
             implements MeinInterface1, MeinInterface2
```

Beispiel

- Zeichenbare Objekte haben Methoden
  - zum Festlegen der Farbe
  - zum Festlegen der Position
  - und zum Zeichnen auf einem Graphik-Objekt.

```
public interface Zeichenbar {
   public void setColor (Color c);
   public void setPosition (double x, double y);
   public void zeichne (Graphics g);
 }
```

Beispiel-Code

```
public class ZeichenbaresQuadrat
        extends Quadrat
        implements Zeichenbar {


    protected Color c;
    protected int x, y;


    public ZeichenbaresQuadrat ( double seite ) { super(seite); }

    public void setColor (Color c) { this.c = c }
    public void setPosition (double x, double y) {
        this.x = (int) x;
        this.y = (int) y;
    }
    public void zeichne (Graphics g) {
        g.setColor(c);
        g.drawRect(x,y,(int) a,(int) a);
    }
}
```

ein `ZeichenbaresQuadrat` erbt alle Methoden von `Quadrat`

und implementiert die Methoden des Interfaces `Zeichenbar`

Die Datenkomponente a wird von `Quadrat` geerbt, die Komponenten `x`, `y`, und `c` werden in der Klasse definiert

Die Methoden `flaeche` und `umfang` werden von `Quadrat` geerbt, die Methoden des Interfaces müssen implementiert werden.

Interfaces und `instanceof`

- Interfaces werden im Sinne der Mehrfachvererbung wie normale Klassen behandelt
  - daher kann auch mit dem Operator instanceof überprüft werden, ob ein Objekt ein bestimmtes Interface implementiert
- Beispiel:

```
ZeichenbaresQuadrat q = new ZeichenbaresQuadrat(4.0);

if (q instanceof Figur) {
    System.out.println("Flaeche: " + q.flaeche());
}
else {
    System.out.println("Keine Figur!");
}

if (q instanceof Zeichenbar) {
   q.setPosition(1,1);
}
else {
  System.out.println("Objekt ist nicht zeichenbar.");
}
```

Für q liefern die Bedingungen in beiden `if`-Statements `true`

Interfaces als abstrakte Klassen

Analog zu Klassen können auch Interfaces als Container verwendet werden

```
Zeichenbar[] f = new Zeichenbar[3];
```

definiert ein Array von zeichenbaren Objekten

Wir nehmen an, `ZeichbaresRechteck` und `ZeichenbarerKreis` sind analog zu `ZeichenbaresQuadrat` definiert

```
f[0] = new ZeichenbarerKreis    ( 2.0 );
f[1] = new ZeichenbaresRechteck( 1.0, 3.0 );
f[2] = new ZeichenbaresQuadrat ( 2.0);

for ( int i=0; i < f.length; i++ ) {
   f[i].setColor(Color.blue);
}
```

Setze die Farbe für alle Objekte auf blau

Verboten ist natürlich auch:

```
f[0] = new Zeichenbar();
```

Abstrakte Klassen vs. Interfaces

- Abstrakte Klassen
  - können einige ihrer Methoden implementieren und weitervererben
    - die abstrakten Methoden müssen mit dem Schlüsselwort `abstract` gekennzeichnet werden
  - Können auch beliebige Datenkomponenten definieren und weitergeben
  - Ein Klasse kann nur von einer abstrakten Klasse erben
- Interfaces
  - Implementierung und Vererbung von Methoden nicht möglich
    - daher sind alle Methoden automatisch als `abstract` deklariert
    - muß nicht extra hingeschrieben werden (kann aber)
  - Können nur Konstanten weitervererben
  - eine Klasse kann viele Interfaces implementieren

Konstanten in Interfaces

- Interfaces können keine Datenkomponenten enthalten
  - macht auch keinen Sinn, da Interfaces nicht erzeugt werden könnten, sondern nur abstrakte Definitionen sind
- Interfaces können aber Klassen-Konstanten enthalten
  - also Komponenten, die `static` und `final` sind
  - diese werden dann von Klassen, die die Interfaces implementieren, geerbt.

```
interface A {
   static final char CONST = 'A';
}

class C implements A {
   void f () {
      System.out.println( CONST );
   }
}
```

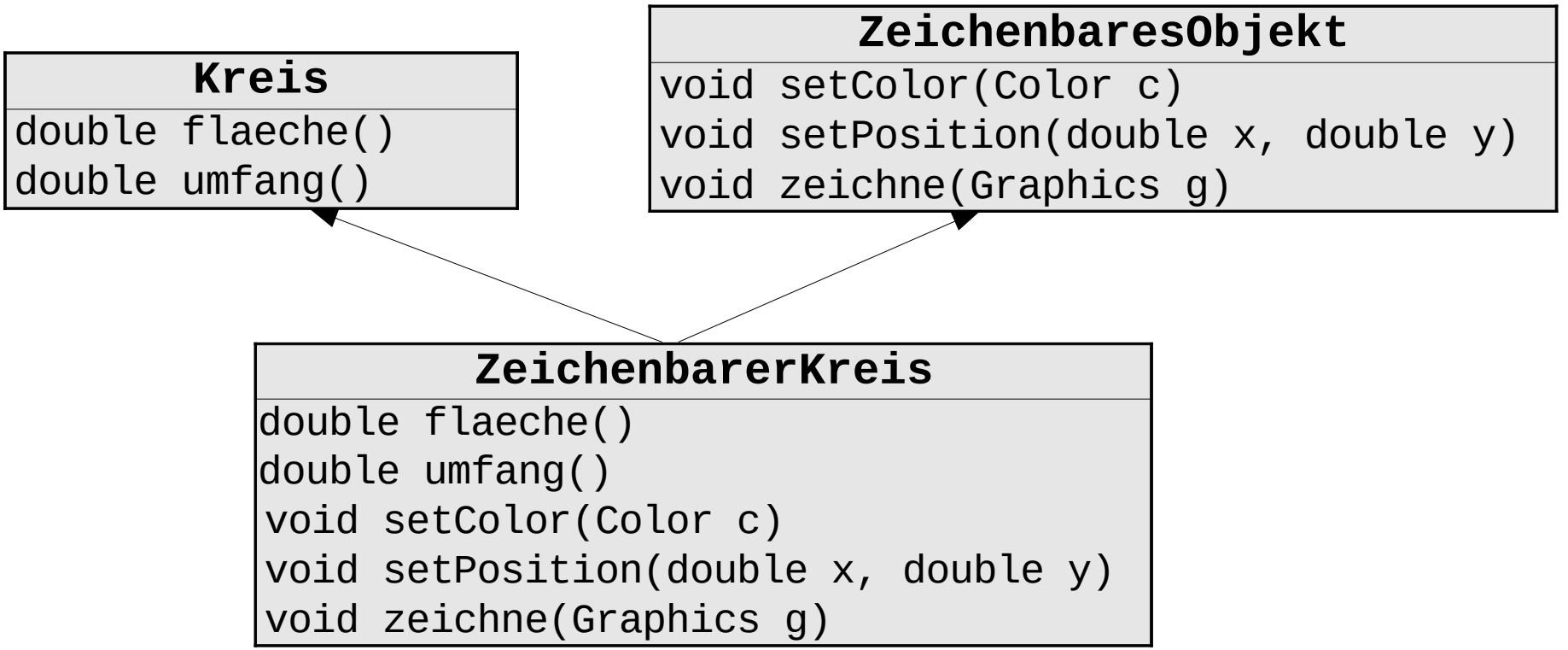
Mehrfache Interfaces

- Eine Klasse kann die Methoden mehrerer Interfaces implementieren
  - z.B. könnte es auch noch ein Klasse `Skalierbar` geben, die angibt, daß ein Objekt mit einem bestimmten Faktor vergrößert werden kann
- Die Definition eines skalierbaren, zeichenbaren Quadrats könnte dann so lauten:

```
public class ZeichenUndSkalierbaresQuadrat
             extends Quadrat
             implements Zeichenbar, Skalierbar {

  // die Methoden von Zeichenbar und Skalierbar
  // muessen hier implementiert werden

}
```

Mehrfachvererbung

- in anderen Programmiersprachen (z.B., C++) ist eine Mehrfachvererbung möglich
  - das heißt, das ein Objekt von mehreren Elternklassen erben kann



Interfaces vs. Mehrfachvererbung

- Interfaces können keine internen Datenkomponenten haben
  - daher können in Interfaces auch keine Methoden implementiert werden
  - obwohl das unter Umständen Sinn machen würde
    - z.B. könnte die Methode `setColor` könnte für alle Unterklassen von `Zeichenbar` gleich sein.

- Bei Mehrfach-Vererbung ist das möglich
  - eine neue Klasse kann Methoden und Datenkomponenten von mehreren Basis-Klassen erben
  - also könnte `ZeichenbarerKreis`
    - die Methoden `laenge` und `umfang` und die Datenkomponente `r` von `Kreis` erben
    - die Methoden `setColor` und `setPosition` und die Datenkomponenten `c`, `x`, `y`, von `Zeichenbar` erben
    - und nur die Methode `zeichne` tatsächlich neu implementieren

Vererbung von Interfaces

- Interfaces müssen nicht immer neu definiert werden
- sondern genauso wie Klassen bestehende Interfaces erweitern
  - Definition wie gehabt mittels `extends` in der Interface Deklaration

- Besonderheit:
  - Im Unterschied zu Klassen kann ein Interface auch von mehreren Interfaces erben!
  - Eine Klasse die solch ein Interface implementiert, muß auch alle Methoden der zugrundeliegenden Basis-Interfaces definieren

Beispiel

```
public interface InterfaceF {
    public void f ();
}

public interface InterfaceG {
    public void g();
}

public interface InterfaceFGH
                  extends InterfaceF, InterfaceG {
    public void h ();
}

public class MeineKlasse implements InterfaceFGH
   {
    public void f () { ... }
    public void g () { ... }
    public void h () { ... }
   }
```

`InterfaceFGH` erbt die Anforderungen von `InterfaceF` und `InterfaceG`

und stellt eine neue

Objekte, die `InterfaceFGH` implementieren, müssen daher Methoden `f`, `g`, und `h` implementieren.

4.4.8 Klasse `Object`

- *Regel*: Wenn eine Klasse
  - nicht mit `extends` *ex*plizit von einer anderen Klasse abgeleitet ist,
  - ist sie *im*plizit von der Klasse `java.lang.Object` abgeleitet.
- *Einzige Ausnahme*: natürlich `java.lang.Object` selbst.
- Die Klasse `java.lang.Object` enthält eine ganze Reihe von Methoden, zum Beispiel:
  - `boolean equals (Object obj)`
  - `String toString ()`

Vererbung von Methoden der Klasse `Object`

- Jede Methode von `java.lang.Object` ist gemäß einer allgemeinen Bedeutung implementiert, zum Beispiel
  - `java.lang.Object.equals`: identisch mit "==".
  - `String toString ()`: der Name der Klasse und eine technische Zusatzinformation wird in einem einzelnen String zusammengefasst.
- Wenn man diese Methoden
  - für eine selbstgebastelte Klasse haben will,
  - und auch genau mit dieser Semantik, dann muss man gar nichts tun: alles schon fertig.
- Will man hingegen eine dieser Methoden
  - für eine selbstgebastelte Klasse haben,
  - aber nicht mit genau dieser Semantik, dann kann man sie eben selbst noch einmal mit der erwünschten Semantik implementieren.

Beispiel

```
public class MeineKlasse
 {
    private int    i;
    private double d;
    private char   c;

   public boolean equals ( Object obj )
    {
      MeineKlasse x = (MeineKlasse) obj;
      return i == x.i && d == x.d && c == x.c;
    }

   public String toString () {
      return new String ( i + " " + d + " " + c );
    }
  }
```

explizites Type-Cast notwendig, da die Signatur der Methode ja eine Vergleich mit `Object` verlangt!

Zwei Objekte vom Typ `MeineKlasse` sollen als gleich betrachtet werden, wenn ihre Datenkomponenten `i`,d und c gleich sind.

Ein `MeineKlasse` Objekt soll durch Angabe der drei internen Komponenten beschrieben werden

Beispiel: `java.util.Vector`

- Die Klasse `java.util.Vector` realisiert im Prinzip ein Array, in dem man auch einfügen und löschen kann.
- Ist aber eine ganz normale Klasse.
- *Dilemma*:
  - ◊ In Methoden wie `add` und `elementAt` muss man den Elementen des Vektors ja irgendeinen Typ geben.
  - ◊ Aber eigentlich wollen wir ja wie bei Arrays Vektoren für alle möglichen Typen haben.
- *Lösung* in Java: Verwenden von `java.lang.Object`.
  - Objekte werden zwar mit ihrem dynamischen Typ abgespeichert
  - Deklaration der Methoden verwendet nur den statischen Typ `Object`
- Beispiel auf der nächsten Folie zeigt auch, dass Wrapper–Klassen wie `Integer` durchaus wichtig sind.

→ Werte von eingebauten Typen können nur in Objekten der jeweils zugehörigen Wrapper–Typen in einem `Vector` gespeichert werden.

Beispiel für Verwendung von `Object`

```
import java.util.*;
...

Vector v = new Vector();

for ( int i=0; i<4; i++ )
 {
   Integer x = new Integer ( i * i );
   v.add (x);
 }

for ( int i=0; i<v.size(); i++ )
 {
   Integer x = (Integer)(v.elementAt(i));
   System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "0149"

v.remove(2);

for ( int i=0; i<v.size(); i++ )
 {
   Integer x = (Integer)(v.elementAt(i));
   System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "019"
```

Importieren von `java.util.Vector`

definiert eine neues `Vector` Objekt (es muß keine Größe angegeben werden)

Die Zahlen `0, 1, 4, 9` werden nacheinander in den `Vector v` eingefügt

`elementAt` retourniert ein Objekt vom generischen Typ `Object`. Daher muß eine explizite Rück-Konvertierung zu `Integer` erfolgen!

Löschen des Elements mit dem Index 2