Vererbung

- was ist Vererbung?
  - neben der Datenkapselung und dem in enger Beziehung zur Vererbung stehenden Polymorphismus eine der drei charakteristischen Eigenschaften der Objektorientierung
  - sie besteht darin, dass es bei der Deklaration von Klassen möglich ist, Eigenschaften bereits vorhandener Klassen zu übernehmen
  - wenn eine Klasse `B` Eigenschaften der Klasse `A` übernimmt, spricht man von Vererbung, d.h. `B` erbt Eigenschaften von `A`
  - Vererbung ist also eine Beziehung zwischen Klassen

- es sind verschiedene Terminologien üblich, um die Vererbungsbeziehung zwischen zwei Klassen `A` und `B` auszudrücken, zum Beispiel
  - `A` ist Superklasse, `B` ist Subklasse
  - `A` ist Oberklasse, `B` ist Unterklasse
  - `A` ist Elternklasse, `B` ist Kindklasse
  - `A` ist Basisklasse, `B` ist abgeleitete Klasse
- wir bevorzugen die zuletzt genannte

- **Achtung:** eine Klasse kann zugleich Basisklasse und abgeleitete Klasse sein
  - wenn die Klasse `B` von `A` erbt und die Klasse `C` von `B` erbt, dann ist `B` sowohl (von `A`) abgeleitete Klasse als auch Basisklasse (für `C`)
- in Java kann eine Klasse immer nur von einer anderen Klasse erben
  - so ist es auch in Smalltalk, Oberon und Ada
- in anderen objektorientierten Sprachen kann eine Klasse von mehreren anderen Klassen erben, d.h es ist Mehrfacherbung möglich
  - zum Beispiel in C++, Eiffel, Perl und Python

- welche Eigenschaften übernimmt eine abgeleitete Klasse von ihrer Basisklasse?
  - deren Attribute und Methoden, d.h. die Gestalt der Objekte der Basisklasse und deren Verhalten

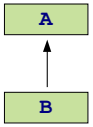
    Achtung: auf Attribute und Methoden, die in der Basisklasse `private` sind, kann aber auch die abgeleitete Klasse nicht direkt zugreifen!
    - die Datenkapselung der Basisklasse bleibt also bei Vererbung erhalten
  - die Konstruktoren der Basisklasse werden nicht übernommen
    - was deutlich macht, dass sie spezielle Methoden sind

- warum ist die Vererbung so wichtig?
  - durch die Möglichkeit zur Übernahme von Eigenschaften bereits vorhandener Klassen wird die Wiederverwendung von Software gefördert und durch die Programmiersprache unterstützt
- die Wiederverwendung von Code ist zwar auch bei prozeduralen Sprachen wie C möglich, wird aber durch die Sprachen nicht „erzwungen“
- bei richtiger Verwendung objektorientierter Sprachen kommt es dagegen durch Vererbung automatisch zur Wiederverwendung von Klassen

- ein Ziel guter objektorientierter Programmierung ist es daher, Klassen so zu gestalten, dass sie möglichst oft wiederverwendet werden
- besonders geeignet hierfür sind Klassen, die möglichst allgemein sind
  - je abstrakter das durch eine Klasse realisierte Konzept ist, desto größer ist die Wahrscheinlichkeit, dass sie zur Vererbung herangezogen wird
- sehr geeignet sind auch Klassen, die Ausschnitte der realen Welt modellieren, die sich nicht oder nur selten ändern
  - z. B. Komponenten grafischer Benutzeroberflächen

- gute objektorientierte Programmierung ist daher immer darauf bedacht, im konkreten Spezialfall das Allgemeine zu erkennen, das wiederverwendet werden kann
  - „weg von speziellen, hin zu allgemeinen Lösungen!“
- Abstraktion, d.h. der Übergang vom Speziellen zum Allgemeinen, ist ein zentraler Aspekt objektorientierter Programmierung
- das Gegenteil von Abstraktion ist Spezialisierung, d.h. der Übergang vom Allgemeinen zum Speziellen
  - sie wird durch Vererbung realisiert

- grafisch wird die Vererbungsbeziehung zwischen Klassen daher so dargestellt:

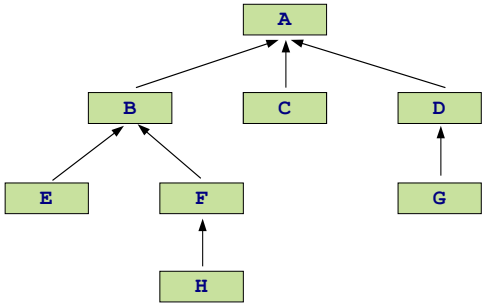


ist Basisklasse, die B verallgemeinert

ist abgeleitete Klasse, d.h. eine Spezialisierung von A

Achtung: bei der Unified Modeling Language (UML) wird die Pfeilspitze nicht gefüllt

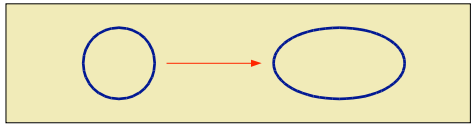
- wenn viele Klassen beteiligt sind, entstehen Klassenhierarchien



- **Fazit:** im Zentrum objektorientierter Programmierung steht nicht nur die Gestaltung einzelner Klassen, sondern darüber hinaus die Gestaltung von Klassenhierarchien

- der sinnvolle Einsatz von Vererbung – insbesondere die Gestaltung komplexer Klassenhierarchien – ist nicht ganz einfach
  - wann soll eine Klasse `B` von einer Klasse `A` abgeleitet werden ?
- zur Beantwortung dieser Frage wird häufig die Ist-ein-Beziehung herangezogen („is a relation“)
  - wenn jedes Objekt der Klasse `B` auch ein Objekt der Klasse `A` ist, sollte `B` von `A` abgeleitet werden
    - manchmal wird die Ist-ein-Beziehung sogar mit der Vererbungsbeziehung gleichgesetzt

- dass es nicht so einfach ist, zeigt folgende – als Kreis-Ellipse-Problem bekannte – Überlegung:
  - offenbar ist jeder Kreis eine Ellipse



  - wären die Ist-ein-Beziehung und die Vererbungsbeziehung identisch, müsste eine zu gestaltende Klasse `Kreis` von einer Klasse `Ellipse` abgeleitet werden

  - andererseits haben Ellipsen zwei Brennpunkte, weshalb die Klasse `Ellipse` sicher zwei private Attribute `brennpunkt1` und `brennpunkt2` haben wird
  - damit auf diese Attribute zugegriffen werden kann, wird die Klasse `Ellipse` Methoden der folgenden Art haben, die `public` sind:

```
setzeBrennpunkt1
setzeBrennpunkt2
liefereBrennpunkt1
liefereBrennpunkt2
```


  - würde die Klasse `Kreis` von der Klasse `Ellipse` abgeleitet werden, würde sie Eigenschaften von `Ellipse` übernehmen
    - insbesondere würde sie die genannten Methoden zur Behandlung der Brennpunkte erben
  - dann hätte aber jedes Objekt vom Typ `Kreis` Methoden, die für Kreise nicht sinnvoll sind !
    - ihr Code müsste sicherstellen, dass die beiden Brennpunkte eines Kreises stets gleich sind und mit dem Mittelpunkt des Kreises übereinstimmen
    - womit sie „unnatürlich“ kompliziert wären
  - **Fazit:** obwohl es zwischen `Kreis` und `Ellipse` eine Ist-ein-Beziehung gibt, sollte `Kreis` nicht von `Ellipse` abgeleitet werden !

- wann genau soll man Vererbung einsetzen?
  - hierauf gibt es keine allgemeingültige Antwort
- wir empfehlen, folgende Fragen zu stellen:
  - soll jedes Objekt der Klasse `B` alle öffentlich zugänglichen Eigenschaften der Objekte der Klasse `A` haben (Gestalt und Verhalten)?
  - soll jedes Objekt der Klasse `B` uneingeschränkt in allen Situationen eingesetzt werden können, in denen ein Objekt der Klasse `A` eingesetzt werden kann?
- ist die Antwort auf eine der Fragen negativ, sollte `B` nicht von `A` abgeleitet werden

- **Hinweis:** die Forderung, dass die zweite Frage positiv beantwortet werden muss, wird als Liskovsches Substitutionsprinzip bezeichnet
  - es wurde erstmals 1987 von Barbara Liskov auf der Conference on Object Oriented Programming Systems Languages and Applications in Orlando, FL, formuliert

Quelle: http://en.wikipedia.org/wiki/File:Barbara_Liskov.PNG

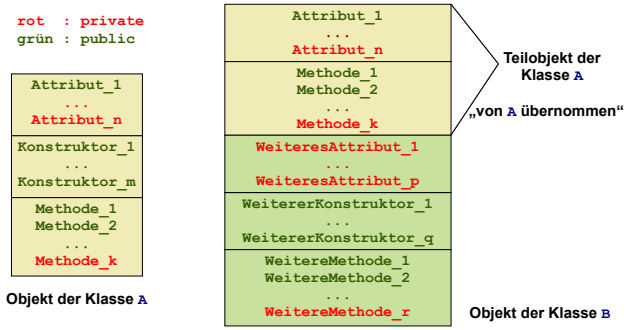
Barbara Liskov ist Professorin am Massachusetts Institute of Technology (MIT).

2008 wurde sie mit der ACM Turing Award ausgezeichnet:

„For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

sie ist die erste Frau, die in den USA an einem Computer Science Department promoviert wurde (Stanford, 1968).

- wenn eine Klasse `B` von einer Klasse `A` abgeleitet wurde, stellen wir uns die Objekte so vor:



- diese Veranschaulichung zeigt:
  - Objekte der Klasse abgeleiteten Klasse `B` haben alle Attribute und Methoden, die Objekte der Basisklasse `A` haben
    - weil auf diejenigen, die `public` sind, jederzeit von außen zugegriffen werden kann, muss die erste der beiden Fragen positiv beantwortet werden können
  - Objekte der abgeleiteten Klasse `B` haben alle Fähigkeiten, die Objekte der Basisklasse `A` haben, darüber hinaus aber (in der Regel) noch mehr
    - daher muss auch die zweite Frage positiv beantwortet werden können

- wie ist die Vererbungsbeziehung zu deklarieren?
  - mit Hilfe des Schlüsselwortes `extends`

```
public class A
{
  // Deklaration von Attributen, Konstruktoren und Methoden
}
```

`B` wird von `A` abgeleitet

```
public class B extends A
{
  // Deklaration weiterer Attribute, Konstruktoren und Methoden
}
```

  - weil es in Java keine Mehrfacherbung gibt, darf hinter `extends` immer nur eine (Basis-)Klasse angegeben werden

- damit eine Klasse **B** von einer Klasse **A** erben kann, muss es möglich sein, auf **A** zuzugreifen
  - d.h. **A** muss geladen werden können und **public** sein oder demselben Package angehören wie **B**
  - außerdem darf **A** nicht als **final** deklariert worden sein
- der Klassenmodifizierer **final** legt fest, dass von Klassen nicht geerbt werden kann

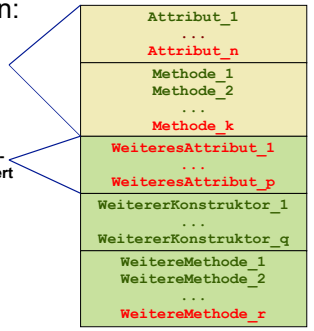
```
public final class A            // von A erben ist nicht erlaubt
{
  // Deklaration von Attributen, Konstruktoren und Methoden
}
```


- **Achtung:** eine Klasse als **final** zu deklarieren ist eine restriktive Maßnahme, die nicht ohne Grund eingesetzt werden soll
  - sie wird zum Beispiel benutzt, wenn die Methoden einer Klasse als so „endgültig“ angesehen werden, dass sie auf keinen Fall verändert werden dürfen
  - die Klassen **Math** und **String** der Java-Klassenbibliothek sind **final**

- wie werden die Objekte abgeleiteter Klassen erzeugt?
  - syntaktisch gibt es keinen Unterschied zum Erzeugen von Objekten nicht abgeleiteter Klassen
  - mit Hilfe des **new**-Operators wird ein Konstruktor der abgeleiteten Klasse aktiviert
  - intern ist der Konstruktionsprozess aber komplizierter, da Objekte abgeleiteter Klassen eine komplexere Struktur haben
  - und dies hat Auswirkungen auf die Gestaltung der Konstruktoren abgeleiteter Klassen

- Objekte abgeleiteter Klassen werden in drei Schritten erzeugt, die stets in folgender Reihenfolge ausgeführt werden:

1. **zuerst wird das Teilobjekt der Basisklasse mit einem Konstruktor der Basisklasse erzeugt**
2. **danach werden die Attribute der abgeleiteten Klasse angelegt und initialisiert**
3. **anschließend werden die (weiteren) Anweisungen des Konstruktors der abgeleiteten Klasse ausgeführt**



- jeder Konstruktor einer abgeleiteten Klasse aktiviert also zuerst einen Konstruktor der Basisklasse
  - dabei gibt es zwei Möglichkeiten, die sich gegenseitig ausschließen:
    - der Konstruktor der Basisklasse wird automatisch aktiviert (implizite Aktivierung)
    - der Konstruktor der Basisklasse wird durch eine Anweisung des Konstruktors der abgeleiteten Klasse aktiviert (explizite Aktivierung)
  - der erste Fall kann nur dann eintreten, wenn die Basisklasse einen Konstruktor ohne Parameter hat
    - da nur dieser automatisch aktiviert werden kann

- zur Illustration der impliziten Aktivierung betrachten wir das folgende, auf das Wesentliche reduzierte Beispiel:

```
/* Achtung: der Einfacheit halber völlig untypische Klasse, die
            weder Attribute noch Methoden hat! */

public class A
{
  public A()
  {
    System.out.println("Klasse A: Konstruktor ohne Parameter");
  }

  public A(String info)
  {
    System.out.println("Klasse A: " + info);
  }
}
```



```
/* Achtung: der Einfacheit halber völlig untypische Klasse, die
            weder Attribute noch Methoden hat! */

public class B extends A
{
  public B()
  {
    System.out.println("Klasse B: Konstruktor ohne Parameter");
  }

  public B(String info)
  {
    System.out.println("Klasse B: " + info);
  }
}
```

diese Anweisung:

```
B testObjekt = new B();
```

führt zu dieser Ausgabe:

```
Klasse A: Konstruktor ohne Parameter
Klasse B: Konstruktor ohne Parameter
```


- die Ausgabe zeigt, dass der parameterlose Konstruktor von **B** zuerst automatisch den parameterlosen Konstruktor von **A** ausführt, bevor er seine eigene Anweisung ausführt
- dasselbe passiert, wenn der mit einem Parameter versehene Konstruktor von **B** aktiviert wird:

diese Anweisung:

```
B testObjekt = new B("Konstruktor mit Parameter");
```

führt zu dieser Ausgabe:

```
Klasse A: Konstruktor ohne Parameter
Klasse B: Konstruktor mit Parameter
```

- **Achtung:** hätte **A** keinen parameterlosen Konstruktor, würde es beim Übersetzen der Anweisungen zu einem Fehler kommen

- zur (häufigeren) expliziten Aktivierung eines Konstruktors der Basisklasse gibt es zwei Möglichkeiten
  - sie kann mit Hilfe von **super** oder **this** erfolgen
- hierbei kann **super** aufgefasst werden als ein allgemeingültiger Platzhalter für den Namen der Basisklasse

`super();`	aktiviert den Konstruktor	`BasisKlasse();`
`super(wert);`	aktiviert den Konstruktor	`BasisKlasse(wert);`
`super(wert1, wert2);`	aktiviert den Konstruktor	`BasisKlasse(wert1, wert2);`

- zur Illustration verändern wir unsere Klasse **B** wie folgt:

```
/* Achtung: der Einfacheit halber völlig untypische Klasse, die
            weder Attribute noch Methoden hat! */

public class B extends A
{
  public B()
  {
    System.out.println("Klasse B: Konstruktor ohne Parameter");
  }

  public B(String info)
  {
    super(info);
    System.out.println("Klasse B: " + info);
  }
}
```


die Anweisung:

```
B testObjekt = new B("Konstruktor mit Parameter");
```

führt jetzt zu dieser Ausgabe:

```
Klasse A: Konstruktor mit Parameter
Klasse B: Konstruktor mit Parameter
```

- sie zeigt, dass der mit dem Parameter versehene Konstruktor von **B** wegen der expliziten Aktivierung mittels **super** zuerst den mit dem Parameter versehenen Konstruktor von **A** ausführt, bevor er seine eigene Anweisung ausführt
- **Achtung:** die explizite Aktivierung eines Konstruktors der Basisklasse mit **super** muss immer die erste Anweisung des Konstruktors der abgeleiteten Klasse sein

- das Schlüsselwort **this** kann in diesem Zusammenhang analog aufgefasst werden als ein allgemeingültiger Platzhalter für den Namen der abgeleiteten Klasse

  Achtung: so verwendet hat das Schlüsselwort **this** eine andere Bedeutung als bisher !

`this();`	aktiviert den Konstruktor	`AbgeleiteteKlasse();`
`this(wert);`	aktiviert den Konstruktor	`AbgeleiteteKlasse(wert);`
`this(w1, w2);`	aktiviert den Konstruktor	`AbgeleiteteKlasse(w1, w2);`

- durch **this** wird also immer ein anderer Konstruktor der abgeleiteten Klasse aktiviert
- dieser wiederum aktiviert zuerst (implizit oder explizit) einen Konstruktor der Basisklasse

- zur Illustration erhält **B** einen weiteren Konstruktor

```
public class B extends A
{
  public B()
  {
    System.out.println("Klasse B: Konstruktor ohne Parameter");
  }

  public B(String info)
  {
    super(info);
    System.out.println("Klasse B: " + info);
  }

  public B(String info, int zeile)
  {
    this("Zeile " + zeile + " ---> " + info);
    System.out.println("Klasse B: " + info);
  }
}
```


die Anweisung:

```
B testObjekt = new B("Aufmerksamkeit erforderlich", 5);
```

führt jetzt zu dieser Ausgabe:

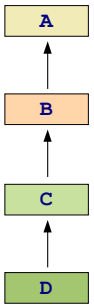
```
Klasse A: Zeile 5 ---> Aufmerksamkeit erforderlich
Klasse B: Zeile 5 ---> Aufmerksamkeit erforderlich
Klasse B: Aufmerksamkeit erforderlich
```

- sie zeigt, dass der mit zwei Parametern versehene Konstruktor von **B** wegen der expliziten Aktivierung mittels **this** zuerst den mit einem Parameter versehenen Konstruktor von **B** ausführt, der wiederum zuerst mit **super** den mit einem Parameter versehenen Konstruktor von **A** aktiviert, bevor er seine eigene Anweisung ausführt

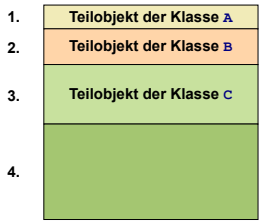
- **Achtung:** die explizite Aktivierung eines Konstruktors der Basisklasse mit **this** muss immer die erste Anweisung des Konstruktors der abgeleiteten Klasse sein

- wenn die Basisklasse einer abgeleiteten Klasse selbst eine abgeleitete Klasse ist, muss auch ihr aktivierter Konstruktor zuerst einen Konstruktor ihrer Basisklasse aktivieren (implizit oder explizit mit **super** oder **this**), usw.
  - dadurch kann die Erzeugung von Objekten abgeleiteter Klassen beliebig komplex werden

bei dieser Klassenhierarchie



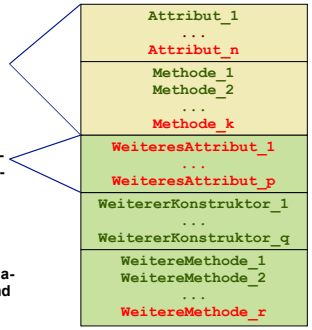
wird ein Objekt der Klasse D in dieser Reihenfolge konstruiert:



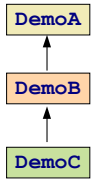
Objekt der Klasse D

- in den nachfolgenden Schritten der Konstruktion eines Objektes einer abgeleiteten Klasse können bereits fertige Teile benutzt werden

1. **zuerst wird das Teilobjekt der Basisklasse mit einem Konstruktor der Basisklasse erzeugt**
2. **zur Initialisierung der Attribute der abgeleiteten Klasse können bereits Attribute und Methoden des Teilobjektes der Basisklasse benutzt werden, die `public` sind**
3. **die weiteren Anweisungen des Konstruktors können die im 2. Schritt initialisierten Attribute und die Attribute und Methoden des Teilobjektes der Basisklasse benutzen, die `public` sind**



- **Achtung:** bei falschem Gebrauch dieser Möglichkeit können beliebig verworrene Programme entstehen!
  - wir illustrieren das mit drei nur zu diesem Zweck (besonders schlecht) geschriebenen Klassen **`DemoA`**, **`DemoB`** und **`DemoC`**, die eine Klassenhierarchie bilden




```
public class DemoA
{
  private int a = -1;          // Initialisierung vor dem Konstruktor

  public DemoA(int wert)     // Ausgaben zum Nachvollziehen untypisch
  {
    System.out.println("DemoA: a ---> " +  a);// schon initialisiert
    setzeA(100);                 // Methode im Konstruktor benutzbar
    System.out.println("DemoA: a ---> " +  a);
    a = wert;
    System.out.println("DemoA: a ---> " +  a);
  }

  public void setzeA(int wert)
  {
    a = wert;
  }

  public int liefereA()
  {
    return a;
  }
}
```



```
public class DemoB extends DemoA
{
  private int b = -1000;

  public DemoB(int wert)    // Ausgaben zum Nachvollziehen untypisch
  {
    super(wert);
    System.out.println("DemoB: b ---> " +  b);
    setzeA(b + liefereA());            // Teilobjekt schon benutzbar
    b = wert;
    System.out.println("DemoB: b ---> " +  b);
    setzeB(wert + liefereA());              // Teilobjekt benutzbar
    System.out.println("DemoB: b ---> " +  b);
  }
  public void setzeB(int wert)
  {
    b = wert;
  }
  public int liefereB()
  {
    return b;
  }
}
```

```
public class DemoC extends DemoB
{
  private int c = liefereA() + liefereB();   // Teilobjekt benutzbar

  public DemoC(int wert)    // Ausgaben zum Nachvollziehen untypisch
  {
    super(wert);
    System.out.println("DemoC: c ---> " +  c);
    setzeA(c + liefereA());                  // Teilobjekt benutzbar
    setzeB(wert + liefereB());               // Teilobjekt benutzbar
    setzeC(liefereA() + liefereB());            // Methode benutzbar
    System.out.println("DemoC: c ---> " +  c);
    c = wert;
    System.out.println("DemoC: c ---> " +  c);
  }
  public void setzeC(int wert)
  {
    c = wert;
  }
  public int liefereC()
  {
    return c;
  }
}
```


welchen Wert haben `a`, `b` und `c` nach diesen Anweisungen?

```
DemoC testObjekt = new DemoC(1);
System.out.println();
System.out.println("a ---> " + testObjekt.liefereA());   // Wert?
System.out.println("b ---> " + testObjekt.liefereB());   // Wert?
System.out.println("c ---> " + testObjekt.liefereC());   // Wert?
```

auf der Konsole ausgegeben wird:

```
DemoA: a ---> -1
DemoA: a ---> 100
DemoA: a ---> 1
DemoB: b ---> -1000
DemoB: b ---> 1
DemoB: b ---> -998
DemoC: c ---> -1997
DemoC: c ---> -3993
DemoC: c ---> 1

a ---> -2996
b ---> -997
c ---> 1
```

während der Konstruktion des Objektes hat sich dessen Zustand mehrfach (auf dubiose Weise) verändert !

- **Fazit:** Konstruktoren sollen immer möglichst einfach gestaltet werden und ausschließlich zur Initialisierung von Attributen dienen

- bei der Deklaration der Methoden abgeleiteter Klassen gibt es zwei Möglichkeiten
  - es werden neue Methoden hinzugefügt, d.h. Methoden, die es in der Basisklasse nicht gibt
  - es werden Methoden der Basisklasse überschrieben
  - beides ist auch gleichzeitig möglich
- wir veranschaulichen das anhand einer bewusst einfach gehaltenen Klasse **`Ware`** und einer davon abgeleiteten Klasse **`ProduzierteWare`**