Datenkapselung und Vererbung

In diesem Script werden zwei Themen behandelt. Zum einen **Datenkapselung** - d.h. Daten vor Zugriff schützen - und zum anderen **Vererbungsmechanismen** - d.h. wir wollen Instanzen und Methoden vererben.

Wie funktioniert beides und mit was fangen wir damit an?

Bevor wir dazu kommen, müssen wir noch eine Konstruktion aus einem der letzten Scripts nochmals genauer anschauen.

Sie erinnern sich sicher, dass wir in den Programmen Autofahren1 und Autofahren2 zwei Objekte angelegt hatten, denen dann im Anschluss Eigenschaften zugewiesen wurden. Es waren die beiden Objekte ***mein Auto*** und ***schumachersAuto*.**

```
public class Autofahren2 {
     public static void main(String args[]) {
      Auto2  meinAuto, schumachersAuto;
      schumachersAuto = new Auto2();
      meinAuto =  new Auto2();
      ...
      schumachersAuto.lenken(15);
      ...
      meinAuto.beschleunigen(4.3, 8.0);
      ...
    }
}
```

In den Zeilen 4 und 5 wird mit dem Schlüsselwort new jeweils eine neue Instanz definiert.

Es wird damit Speicherplatz für die später noch festzulegenden Eigenschaften reserviert. Das geschieht in Zusammenhang mit dem Konstruktor Auto2().

Konstruktoren sind nun Methoden, die den Namen der Klasse als Bezeichner tragen. Sie werden bei der Instanzierung eines Objekts der Klasse ausgeführt.

Jede Klasse in Java hat implizit einen solchen Standard-Konstruktor wie hier Auto2(). Dieser ist bereits durch die Definition der Klasse gegeben.

Solche Konstruktoren dienen zum Festlegen von Startwerten für Instanzen.

Jede Klasse hat also automatisch einen Konstruktor mit dem selben Namen wie die Klasse selbst.

Hier kommt ein weiteres Konzept zum Tragen. Das der sogenannten ***Überladung***. Das bedeutet, Sie können mehrere Konstruktoren programmieren - alle mit der selben Bezeichnung.

Diese Konstruktoren müssen sich lediglich in ihren Parametern unterscheiden. Man sagt, sie unterscheiden sich in der ***Signatur***.

Hier ist ein Beispiel:

```
class Auto3{
...
public Auto3(String eigentuemer, String farbe) {
   eigentuemer = eigentuemer;
      farbe = farbe;
      geschwindigkeit = 0.0;
   richtung = 0;
}
public Auto3(Auto3 auto) {
   this(auto.eigentuemer, auto.farbe);
   geschwindigkeit  = auto.geschwindigkeit;
   richtung = auto.richtung;
}
...
}
```

Der erste Konstruktor erlaubt beim Aufruf sofort die Übergabe der Werte für den Eigentümer und die Farbe des Autos.

Beim zweiten Konstruktor wird sogar zugelassen, dass als Parameter eine Instanz der aktuellen Klasse übergeben wird.

Diese Möglichkeit braucht man zum Beispiel, wenn sich jemand ein zweites Auto kauft. Man will dann alle Daten zur Person - allgemeiner alle Daten, die sich nicht ändern - nicht nochmals eingegeben wollen.

Sie sehen in Zeile 2 der zweiten Definition des Konstruktors Auto3 das neue Schlüsselwort ***this***. Zum Aufruf des als erstes definierten Konstruktors mit den beiden Parametern Eigentümer und Farbe hier, muss es verwendet werden. Damit werden alle dort gesetzten Daten übernommen.

Der Befehl "eigentuemer = eigentuemer;" ist durch die folgende Konvention bedingt.

Links des Gleichheitszeichens dieser Anweisung wird die Zeichenkette "eigentuemer" als Instanzvariable aufgefasst.

Rechts vom Gleichheitszeichen steht die Zeichenkette "eigentuemer" für den entsprechenden Parameter, den wir aus verständlichen Gründen genau so bezeichnet haben.

Um das etwas durchsichtiger zu gestalten, empfiehlt sich auch hier die Verwendung von ***this***, das steht dann für den aktuell zu definierenden Konstruktor, und das Programm schaut dann so aus:

```
class Auto3{
...
public Auto3(String eigentuemer, String farbe) {
   this.eigentuemer = eigentuemer;
      this.farbe = farbe;
      this.geschwindigkeit = 0.0;
   this.richtung = 0;
}
public Auto3(Auto3 auto) {
   this(auto.eigentuemer, auto.farbe);
   this.geschwindigkeit  = auto.geschwindigkeit;
   this.richtung = auto.richtung;
}
...
}
```

Wir wollen aber nun noch einen anderen Aspekt in unseren Programmen anschauen.

Das Programm Autofahren1 hat wie folgt ausgesehen:

```
public class Autofahren1 {
     public static void main(String args[]) {
         Auto1  meinAuto, schumachersAuto;

         meinAuto = new Auto1();
         schumachersAuto = new Auto1();

         meinAuto.eigentuemer="JG";
         meinAuto.farbe="schwarz";
         meinAuto.marke="Fiat";
         ...
         schumachersAuto.marke="Ferrari";
         schumachersAuto.geschwindigkeit ="350";
      ...
    }
}
```

Wir haben da in einer anderen Klasse - nämlich **Autofahren1** - direkt auf die Instanzvariablen zugegriffen.

Das verstösst nun eigentlich gegen das Prinzip der **Datenkapselung** oder auf englisch ***Information Hiding***. Dieses Prinzip besagt, dass Daten so weit wie nur möglich gegenüber einem Zugriff von aussen abgeschottet werden sollen.

Wir müssen hier die Informationen einerseits ***geheim*** halten, wollen aber gerade die Zustände eines Autos ändern!

Und das wiederum bedeutet, dass wir *den Zugriff* auf die Daten ja eigentlich behalten müssen.

Nur sollte das ganz konsequent über entsprechend zu definierende Methoden geschehen.

Das heisst wir erlauben nicht mehr wie bei den Klassen **Auto1** und **Auto2**, dass von anderen Klassen den Instanzvariablen Werte zugewiesen werden dürfen. Für den Zugriff und das Setzen von Werten werden zusätzlich Methoden bereitgestellt, die genau das leisten.

Man hat über die Jahre gelernt, dass so ein etwas engeres Korsett im Endeffekt mehr Vorteile als Nachteile bringt.

Wenn Eingriffe von aussen nur durch genau definierte Methoden stattfinden, dann bietet das weniger Fehlerquellen in großen Projekten der Anwendungsentwicklung!

Da arbeiten viele Teams an Teilproblemen und von daher ist so eine strenge Konvention zur Gestaltung der Schnittstellen von Vorteil.

Es gibt auch noch weitere Vorteile. Wenn eine Klasse als sogenannte Black-Box verwendet wird, dann kann sich das Innenleben auch verändern, ohne dass die Gefahr der Fehleranfälligkeit beim Austausch einer solchen Black-Box entsteht.

Man spricht von einer Black Box, wenn wir über die internen Strukturen nichts wissen, sondern nur über die Schnittstellen nach außen.

Die Änderung des Innenlebens einer Klasse kann mehrere Gründe haben. Nehmen Sie z.B. Verfahren der Kryptographie zur Verschlüsselung von Texten und Daten, die in einer Klasse

implementiert sind und die im grösseren Kontext von sicheren Übertragungen von Nachrichten und Geldtransaktionen im Internet benutzt wird.

In der Mathematik und Informatik forscht man nun ständig an neuen und besseren Verschlüsselungsverfahren.

Will man nun ein besseres Verfahren implementieren, dann ist durch die Technik des Information Hiding sichergestellt, dass es beim Austausch der Klasse mit dem alten Verfahren mit der Klasse des neuen Verfahrens keine Probleme geben wird!

Syntaktisch wird man deshalb besser die Instanzvariablen mit dem Modifikator ***private*** deklarieren.

Veränderungen dieser Zustandswerte geschehen dann durch zusätzliche Methoden, deren Bezeichnungen oft mit ***set*** für das Setzen von neuen Werten beziehungsweise mit **get** für das Auslesen der aktuellen Werte beginnen.

Vererbung

Wir stellen uns vor, dass wir schliesslich alle wesentlichen Aspekte in der Klasse Auto implementiert haben. Eine neue Klasse *Familienauto* soll nun sicher all diese Aspekte realisieren.

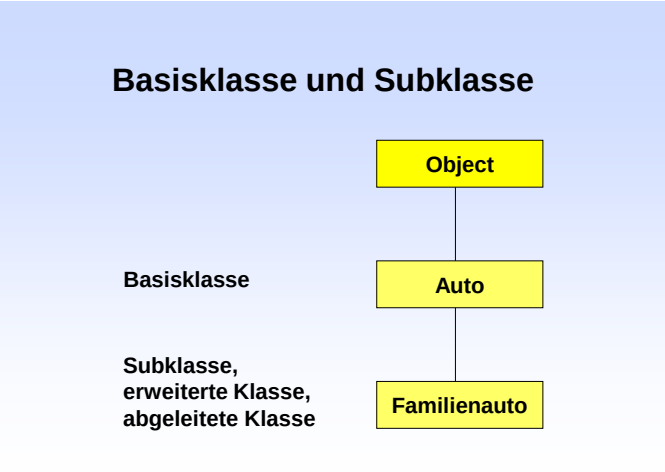
Da würde es doch nicht besonders interessant sein, wenn wir all die vielen Eigenschaften wieder neu implementieren müssten!

Das Konzept der Vererbung ist zentrale Eigenschaft von Klassen

- Klassen können Eigenschaften anderer Klassen *erben*.
- Klassen können durch Vererbung *weiterverwendet* und
- zusätzlich *verfeinert* werden.

Das zentrale Konzept der Vererbung erlaubt es Eigenschaften und Methoden aus einer Klasse - die Basisklasse - in einer neuen Klasse - die Subklasse - zu verwenden und durch Definition von neuen Methoden diese Verfeinern.

Der Ausgangspunkt aller Klasse ist in Java die Klasse *Object*.



Die Klasse Auto erbt also bereits Basismethoden aus der Klasse Object.

Die Subklasse - oder auch erweiterte Klasse - Familienauto erbt alle Eigenschaften und Methoden aus der Klasse Auto.

Weiter können in ihr neue Eigenschaften und Methoden implementiert werden.

Ich denke in diesem Fall an eingebaute Kindersitze, einen großen Kofferraum für Kinderwägen etc.

Diese Vererbung geschieht beim Implementieren im Programmcode durch die Verwendung des Schlüsselwortes ***extends.***

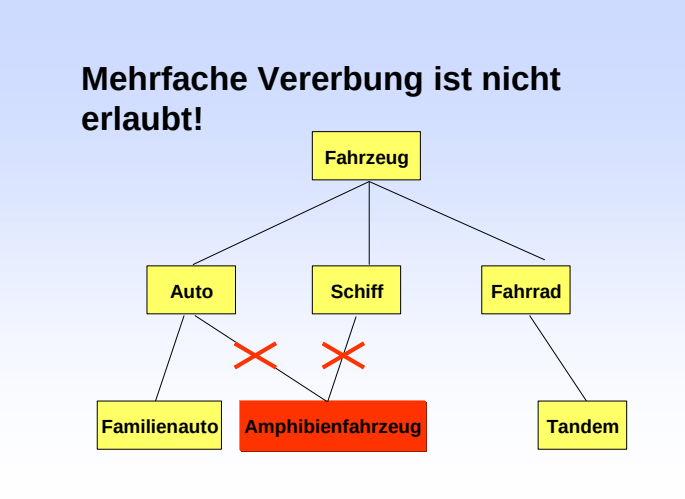
```
class Familienauto extends Auto {


   private boolean kindersitz;
  public boolean isKindersitz() {
      return this.kindersitz;
   }
   public void setKindersitz(boolean ks) {
      this.kindersitz = ks;
   }
}
```

Im Programmcode sehen Sie übrigens auch jetzt die konsequente Umsetzung des Prinzips der Datenkapselung durch die Methoden **isKindersitz** und **setKindersitz**

Mit dem Schlüsselwort „**extend“** haben wir uns das erneute Hinschreiben des **kompletten Codes** für die Eigenschaften und Methoden aus der Basisklasse sozusagen erspart!

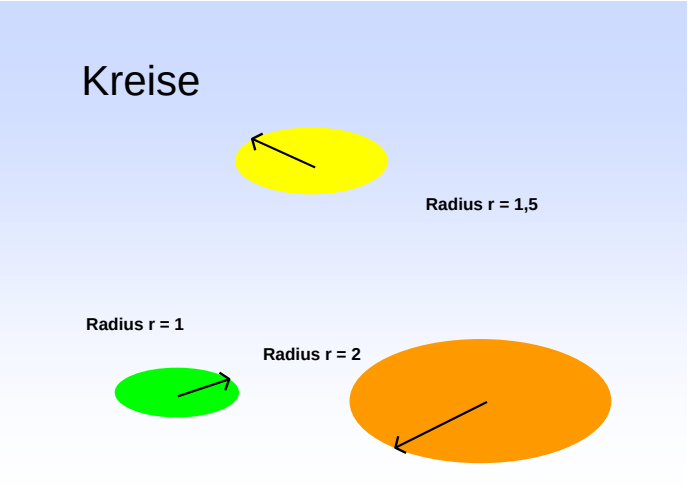
Mit dieser Idee kann man nun ganze *Hierarchien* von Klassen in Java entwickeln, beispielsweise sehen Sie auf der nächsten Folie eine ganze Hierarchie von Fahrzeugen.



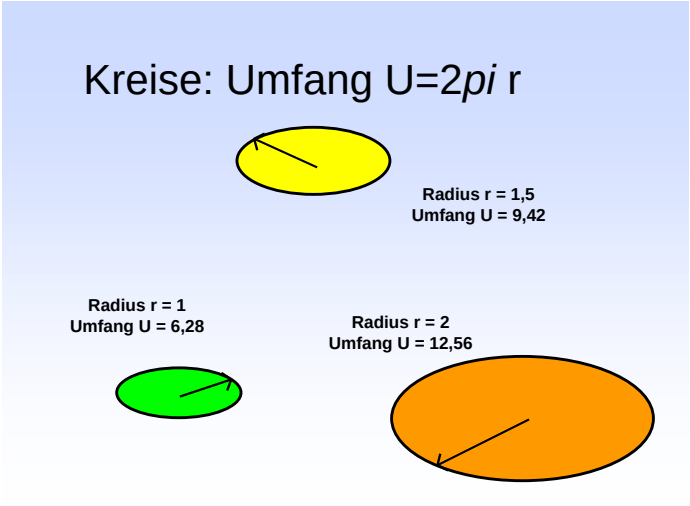
Man muss allerdings noch beachten, dass immer nur von einer Klasse geerbt werden kann!

Amphibienfahrzeuge, die sowohl von der Klasse Auto als auch von der Klasse Schiff erben könnten, sind so in Java nicht erlaubt! Zur weiteren Vertiefung der Theorie der Vererbungsmechanismen in Java werden wir beispielhaft dazu Kreise und Zylinder programmieren.

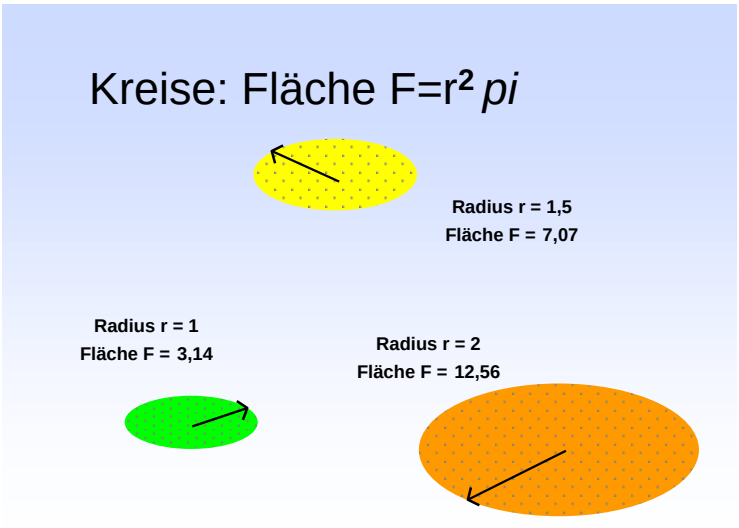
Und als kleine Erinnerung und Einführung müssen wir uns zunächst an deren wichtigste mathematische Eigenschaften erinnern.



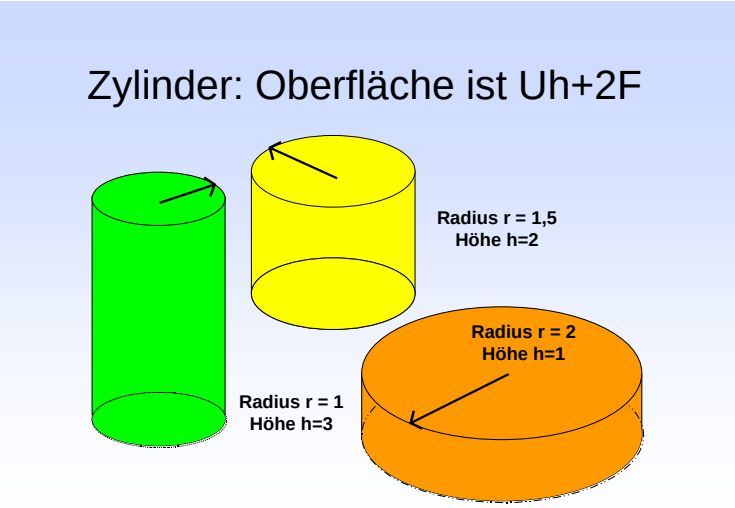
Auf dieser Folie sehen Sie 3 Kreise mit verschiedenen Radien. Ein Kreis ist durch die Lage seines Mittelpunktes sowie durch die Länge seines Radius gegeben. Unsere Kreise hier haben die Radien 1, 1,5 und 2.



Bei der Berechnung von Umfang und Fläche eines Kreises kommt die Kreiszahl *pi* ins Spiel. Zur Berechnung des Umfangs U ist das Doppelte der Kreiszahl mit dem Radius zu multiplizieren.



Bei der Berechnung der Fläche ist der Radius zu quadrieren und mit der Kreiszahl *pi* zu multiplizieren. Das haben wir hier für die drei Kreise gemacht. Da die Dezimalentwicklung der transzendenten Zahl ***pi*** nicht abbricht, sind unsere Ergebnisse hier natürlich immer gerundet.



Wenn sich ein Kreis in die dritte Dimension weiter entwickelt, dann kann er zum Zylinder werden. Mit der zusätzlich zum Mittelpunkt und zum Radius festgelegten Höhe h sind die wesentlichen Kenngrößen eines Zylinders schon festgelegt.

Man könnte also die drei-dimensionalen Zylinder als Erweiterung der zwei-dimensionalen Kreise mit zusätzlicher Angabe einer Höhe auffassen - und die Kreise damit als Spezialfall von Zylindern mit einer Höhe 0.

Wenn wir also in Java die Klasse Zylinder implementieren wollen, dann können wir von der Klasse: „Kreise“ erben! Zusätzlich wird als weitere Instanzvariable die Höhe eines Zylinders eingeführt, da es unnatürlich für einen Kreise wäre, wenn er immer einen Wert Höhe = 0 mit sich rumschleppen müssten.

Wir erben natürlich auch die Methoden zur Berechnung des Umfangs und der Fläche.

Der Umfang eines Zylinders ist ja wohl keine sinnvolle Grösse. Die Fläche eines Zylinders wiederum könnte dann aber die Oberfläche bedeuten. Genau so haben wir das im folgenden Programm implementiert:

```
class Kreis {
   ...
   private double radius;
   public void setRadius ( double r ) { radius = r }
   public double flaeche() {return 3.14*r*r}
   public Kreis ( doublr r ) { radius = r }
}
class Zylinder extends Kreis {
   private double hoehe;
   ...
   public double flaeche() {
      double fl , umf;
      umf = super.umfang();
      fl = super.flaeche();
      return 2*fl + hoehe*umf;
   }
}
```

Zur Berechnung der beiden Deckelflächen können wir die geerbte Flächenberechnung für die Kreise nutzen. Die Oberfläche des Zylinders ergibt sich aus zweimal der Kreisfläche für die beiden Deckel sowie der Fläche des Mantels. Dafür können wir dann sogar auch noch die geerbte Methode

für den Umfang des Kreises nutzen. Die Fläche des Mantels ist ja gerade dieser Umfang mal die Höhe.

Um im Programmcode klar zu stellen, dass einmal die Methode flaeche der Klasse Zylinder gemeint ist, zum anderen Mal die Methode Flaeche aus der Klasse Kreis, die die Subklasse Zylinder geerbt hat, wird sie als Methode der Basisklasse mittels des Schlüsselworts ***super*** genauer bezeichnet.

Die Basisklasse wird also mit **super** referenziert, eine Methode flaeche daraus dann mit super.flaeche.

Soll der Konstuktor der Basisklasse in einem Konstruktor verwendet werden, dann muss der Aufruf super(); in der ersten Zeile des Konstruktors der Subklasse stehen! Man beachte, dass super() nur in der Subklasse selbst und nicht für Instanzen der Basisklasse verwendet werden kann.

Um das nochmals zusammenzufassen: Wir haben in der Klasse Zylinder die Methode flaeche geerbt. Diese übernehmen wir aber nur mit ihrer Signatur und implementieren sie neu, benutzen aber dabei die von den Kreisen geerbte Methode flaeche zur Berechnung der Grund- und der Deckelfläche!

Übungsaufgabe:

Aufgabe1: Vererbung:

Entwickeln Sie eine spezielle Klasse (z.B. Person, Angestellter, Schüler) und nutzen dafür die Konzepte der Vererbung.

Aufgabe2: Überladung und Datenkapsel

```
class Auto3 {

  String marke;

  String eigentuemer;

  String farbe;

  double v;

  int richtung;
```

```
 public int lenken ( int w ){
  richtung = richtung + w;
  return richtung;
 }
 public double beschleunigen(double a, double t){
  v = v + a*t;
  return v;
 }
 public Auto3() {
 // Konstruktor 1
  System.out.println("Konstruktor1 Auto3");
  v = 0;
  richtung = 0;
 } // end Konstruktor 1

 public Auto3(String eigentuemer, String farbe) {
 // Konstruktor 2
  System.out.println("Konstruktor2 Auto3");
 } // end Konstruktor 2

 public Auto3(Auto3 auto) {
 // Konstruktor 3
  System.out.println("Konstruktor2 Auto3");
} // end Konstruktor 3
}
```

Ergänzen Sie die drei Konstruktoren mit den entsprechenden Initialisierungen. Bei Konstruktor 3 soll Konstruktor 2 verwendet werden. Wählen Sie sinnvolle Beispiele.

Verändern Sie die Klasse zu einer vollständigen **Datenkapsel** (Information Hiding). Dazu müssen Sie zusätzliche Methoden schreiben.

Aufgabe3: Unterschiedliche Signaturen bei Auto3

Erzeugen Sie drei verschiedene Objekte der Klasse Auto3. Es soll dabei anhand von Beispielen ersichtlich werden, wie die mit unterschiedlichen Signaturen ausgestatteten Konstruktoren verwendet werden.

Es soll ebenfalls durch Beispiele gezeigt werden, wie Eigenschaften der Klasse durch Methodenaufrufe verändert werden können.

Aufgabe 4: Klasse Autofahren4 (Anwendung)

Schreiben Sie eine vollständige Subklasse Familienauto mit eigenen weiteren Methoden. Als Basisklasse soll die Klasse Auto3 verwendet werden.

Schreiben Sie zwei Konstruktoren mit unterschiedlicher Signatur für diese Klasse.

Beachten Sie auch hier die Datenkapselung

Erzeugen Sie Objekte der Klasse Familienauto. Untersuchen Sie welche Methoden Ihnen zur Verfügung stehen. (Basisklasse und Subklasse)

Untersuchen Sie wann welche Konstruktoren aufgerufen werden. Zeigen Sie diesen Vorgang anhand geeigneter Beispiele.