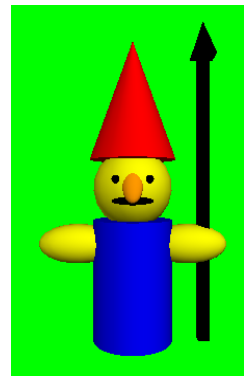


VPython am Beispiel eines Zwergs

- ☞ Ausführliche Dokumentation unter <http://www.vpython.org>
- ☞ Installation von VPython (siehe separate Anleitung)
 - zuerst Python (z. B. Version 3.2) installieren
 - dann dazu passendes VPython (z. B. Version 5.74) installieren



Erzeugen eines einfachen Zwergs

Ein einfacher Zwerg besteht aus einem Zylinder, einer Kugel und einem Kegel. Diese können in vpython mit den Klassen **cylinder()** (cylinder, engl. heißt Zylinder), **sphere()** (sphere, engl. heißt Kugel) und **cone()** (cone, engl. heißt Kegel) erzeugt werden.

Um das Visual Modul nutzen zu können, muss es zunächst eingebunden werden. Dies geschieht mit der Zeile

from visual import *

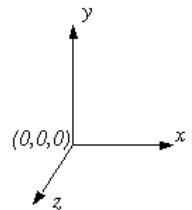
d. h. alle Klassen und Methoden des Moduls visual werden eingebunden.

Es wäre auch möglich, nur einzelne Methoden einzubinden (sinnvoll, wenn nur diese eine Methode benötigt wird). Z. B.

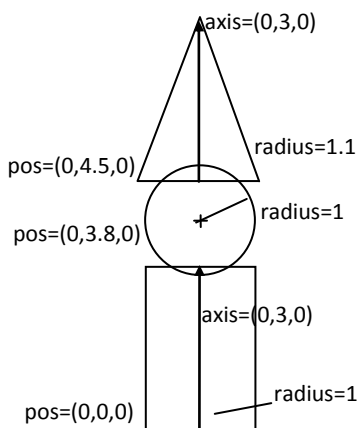
from random import choice oder z. B. **from random import random**

fügt nur die Methode "choice" bzw. im zweiten Beispiel die Methode "random" des Moduls "random" hinzu. Im zweiten Beispiel heißen die Methode und das Modul zufälligerweise gleich.

Im Visual-Modul gibt es 3 Achsen, wobei die z-Achse sozusagen aus dem Monitor raus zum Benutzer hin zeigt. Siehe Abbildung



Objektkarten der 3 "Zwergobjekte":



koerper:cylinder
pos = (0,0,0) radius = 1 axis = (0,3,0)

kopf:sphere
pos = (0,3,8,0) radius = 1.1

hut:cone
pos = (0,4,5,0) radius = 1.1 axis = (0,3,0)

Das **Objekt koerper** ist von der **Klasse cylinder** und hat die angegebenen Attributwerte.

Um in Python ein solches Objekt zu erzeugen, werden folgende Zeilen benötigt:

```
koerper = cylinder()  
koerper.pos = (0,0,0)  
koerper.radius = 1  
koerper.axis = (0,3,0)
```

Die zweite und dritte Zeile könnte weggelassen werden, da pos = (0,0,0) und radius = 1 die Standardbelegung ist. Default bei axis ist (1,0,0), d. h. das Objekt zeigt standardmäßig immer in Richtung der x-Achse.

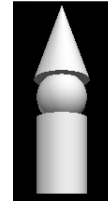
Alternativ können die Attributwerte auch beim Erzeugen des Objekts gesetzt werden:

```
koerper = cylinder(pos=(0,0,0), radius=1, axis=(0,3,0))
```

Die Reihenfolge der Attribute ist dabei egal. Weitere Attributwerte können analog gesetzt werden.

Anstatt den x-, y- und z-Wert in pos anzugeben, kann auf diese Werte auch einzeln zugegriffen werden:

```
koerper.x = 0          # x-Position ist Null, bewirkt das Gleiche wie koerper.pos.x = 0
koerper.y = 0
usw.
```



Aufgabe01 (Zwerg erzeugen):

Erzeuge einen Zwerg, der aus oben angegebenen Objekten besteht.

Darstellung:

Mit der rechten Maustaste kann die Szene gedreht werden.
Mit beiden Maustasten gleichzeitig kann gezoomt werden

Veränderung der Fenster- /Kameraeigenschaften:

Für eine ansprechende Optik kann es sinnvoll sein, einige Änderungen am eigentlichen Fenster/Kamera vorzunehmen.

„scene“ ist der Standardname des ersten visual-Fensters, das automatisch erzeugt wird.

Es gibt zahlreiche Möglichkeiten die Fenstereigenschaften einzustellen bzw. die Kamera zu steuern (siehe dazu online-Dokumentation).

Hier einige Beispiele:

Fenstereigenschaften:

scene.title	Fenstertitel
scene.background	Farbe Hintergrund
scene.height	Fensterhoehe
scene.width	Fensterbreite
scene.fullscreen	Boolean-Wert, gibt Vollbildmodus an
scene.stereodepth	fuer 3D-Effekte sollte der Wert 2 sein
scene.stereo	für die Ansicht mit Anaglyphen-Brillen , z.B. „redcyan“ (oder redblue, yellowblue) einstellen
scene.ambient	Farbe des Umgebungslichts (default 0.2, erhöht man diesen Wert, wird die Szene heller, sonst dunkler)
scene.lights	Liste von "Lampen", die die Szene beleuchten, siehe dazu online-Dokumentation
scene.cursor.visible	Boolean-Wert, gibt an, ob Cursor sichtbar ist

Kamerasteuerung:

scene.center	Position, auf die die Kamera schaut, default (0,0,0)
scene.forward	Vektor, in dessen Richtung die Kamera schaut, default (0,0,-1)
scene.userzoom	Boolean-Wert, gibt an, ob der Benutzer zoomen darf
scene.userspin	Boolean-Wert, gibt an, ob der Benutzer rotieren darf
scene.autoscale	Boolean-Wert, sorgt für automatische Skalierung

Farben:

Die Farben können als rgb-Werte angegeben werden. Dabei erwartet vpython jeweils Werte zwischen 0 und 1.

Beispiel:

```
koerper = sphere(pos=(0,0,0), radius = 1, color = (0,0,1))
```

erzeugt im Ursprung des KO-Systems einen blauen Zylinder namens koerper mit Radius 1.

Manche Farben sind schon vordefiniert in der Klasse `color` enthalten, die wichtigsten:

```
color.red, color.yellow, color.black, color.green, color.orange, color.white, color.blue, color.cyan, color.magenta
```

Um leuchtende Farben zu erzeugen kann die Methode

`rgb_to_hsv(c)`

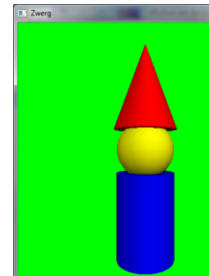
verwendet werden. `c` stellt dabei einen rgb-Wert, z. B. (1,1,0) dar.

Beispiel:

```
c = (1,1,0)
neue_farbe = color.rgb_to_hsv(c)
```

Aufgabe02 (Titel und Hintergrundfarbe):

Nenne den Fenstertitel `Zwerg` und setze die Hintergrundfarbe auf `green`. Färbe zudem den Körper blau, den Kopf gelb und den Hut rot. Setzt das Bildzentrum auf den Mittelpunkt des Kopfes.



Gruppierung der Objekte

Bisher besteht der Zwerg aus drei voneinander unabhängigen Objekten. Um nur ein einziges Objekt `Zwerg` zu erhalten, müssen diese gruppiert werden.

Dazu erstellt man zunächst eine Gruppe `zwerg`. In Python benötigt man hierfür die Klasse `frame()`.

```
zwerg = frame()           #eine Gruppe namens zwerg wird erzeugt
```

Um der Gruppe `zwerg` mitzuteilen, welche Objekte sie enthält, muss das Attribut `'frame'` des jeweiligen Objekts den Attributwert des Gruppennamens, hier ist dies `'zwerg'`, erhalten.

Beispiel:

```
zwerg = frame()
koerper = cylinder(frame = zwerg, pos = ....)
kopf = sphere(frame = zwerg, pos = ....)
....
```

Aufgabe03 (Gruppierung und Verschieben):

a) Gruppiere die vorher erstellten Objekte des Zwerges.

Lass anschließend deinen Zwerg 10 Schritte in x-Richtung wandern. Ein Schritt soll dabei 0.3 Einheiten lang sein.

Hinweise:

Um die Bewegung sehen zu können, setzen wir die Animationsfrequenz (also Anzahl der Animationen pro Sekunde) innerhalb der Schleife auf 30. Verwende hierfür die Methode

```
rate(30)
```

Um den Zwerg wandern zu lassen, verändere den x-Wert der Zwergenposition sukzessive. Verwende also folgende Zeile:

```
zwerg.x = zwerg.x + 0.3           # alternativ: zwerg.x += 0.3
```

b) Aufgrund der automatischen Skalierung sieht der Zwerg gedreht aus. Verwende daher
`scene.autoscale = False`

Material

Jeder Körper kann aus einem bestimmten Material bestehen. Attribut material, Attributwert z. B.

materials.chrome	das Material ist Chrom,
materials.marble	Marmor
materials.wood	Holz
materials.rough	
materials.plastic	
materials.earth	
materials.BlueMarble	
materials.emissive	sieht aus, als würde es leuchten

(und noch weitere - siehe Dokumentation)

Aufgabe04 (Material):

Erzeuge eine Kugel und gib ihr den Materialwert earth bzw. auch BlueMarble

Aufgabe05 (Kampfwerg):

Öffne die Datei *Aufgabe05_Kampfwerg.py* und erstelle alle fehlenden Objekte.
Gruppiere diese anschließend und rücke den Kamera-Blickpunkt auf die Kopfmitte.



eigene Klassen erzeugen

Bevor auf das Erzeugen neuer Klassen eingegangen wird, soll der Begriff der Vererbung näher erläutert werden:

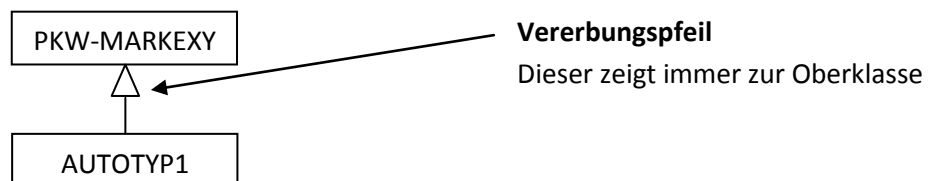
Definition **Vererbung**:

Klassen können von anderen Klassen *abgeleitet* werden (Vererbung).
Dabei erbt die Klasse (**Unterklasse**) die Attribute und Methoden von der *vererbenden* Klasse (**Oberklasse**). D. h. sie besitzt die gleichen Attribute und Methoden. Auf diese kann damit zugegriffen werden, ohne sie neu definieren zu müssen.
Die Unterklasse kann jedoch auch noch weitere zusätzliche Attribute bzw. Methoden besitzen. Methoden können auch überschrieben werden. Dies nennt man dann **Polymorphismus**. Dazu verwendet man den gleichen Methodennamen wie in der Oberklasse. Der zugehörige Codeblock, der nach Aufruf der Methode ausgeführt wird, unterscheidet sich jedoch.

Beispiel aus dem Alltag:

Oberklasse sei die Klasse PKW-MARKEY, eine mögliche Unterklasse AUTOTYP1.

Grafische Darstellung:

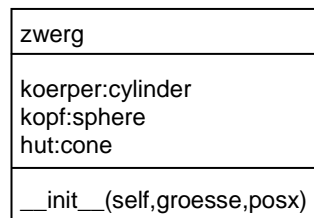
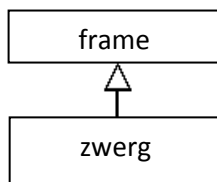


Jeder PKW hat eine bestimmte PS-Zahl, einen Hubraum, ein Gewicht usw. Diese Attribute können somit in der Oberklasse definiert werden und müssen in der Unterklasse AUTOTYP1 nicht noch einmal neu erstellt werden. Wir können jedoch trotzdem auch in der Unterklasse auf sie zugreifen. Zusätzlich kann in der Unterklasse beispielsweise das Attribut Farbe definiert werden. Auf die Farbe kann nun nur ein Objekt der Klasse AUTOTYP1 zugreifen, ein Objekt der Klasse PKW-MARKEY kennt die Farbe nicht.

Bei jedem PKW lässt sich ein Rückwärtsgang einlegen. Standardmäßig geschieht dies durch Verschieben des Schalthebels nach links-oben. Bei manchen PKWs liegt der Rückwärtsgang jedoch rechts unten. Es ist nun möglich, in der Oberklasse eine Methode rückwärtsgang_einlegen() zu

definieren und in dieser den Schalthebel nach links oben stellen zu lassen. Damit wird auch beim Aufruf dieser Methode in einer zugehörigen Unterklasse der Schalthebel nach links oben gestellt. Wird nun in der Unterklasse eine neue Methode gleichen Namens (rückwärtsgang_einlegen()) definiert und in dieser der Schalthebel nach rechts unten gestellt, so wird für jedes Objekt dieser Unterklasse beim Aufruf dieser Methode der Rückwärtsgang durch Stellung des Schalthebels nach rechts-unten eingelegt. Die Methode rückwärtsgang_einlegen() wurde überschrieben.

Zurück zu unserem Zwergenbeispiel:



Die erste Grafik gibt an, dass die Klasse 'zwerg' von der Klasse 'frame' erbt.

Die zweite Grafik zeigt eine Klassenkarte der Klasse 'zwerg'.

```
class zwerg(frame):
    def __init__(self, groesse = 1, posx = 0):
        frame.__init__(self)
        self.koerper= cylinder(frame = self, pos = (posx,0,0),radius = groesse,
                               axis = (0,3*groesse))
        self.kopf = sphere(frame = self,pos = (posx,groesse*3.8,0), radius = groesse,
                           color = color.yellow)
        self.hut = cone(frame = self,pos = (posx, groesse*4.5,0), radius = groesse*1.1,
                        axis = (0, groesse*3,0), color = color.red)
```

```
borin = zwerg(groesse = 1, posx = 0)
furgil = zwerg(groesse =0.75, posx = -3)
```

Möchte man viele gleiche Zwerge haben, so ist es sinnvoll, eine **eigene Klasse Zwerg** zu haben. Alle Werte, die man gleich beim Erzeugen setzen möchte, schreibt man in die Init-Methode (ähnlich wie beim Zylinder, als man die Attributwerte sofort beim Erzeugen des Objekts gesetzt hat - z. B. durch cylinder(pos=(0,0,0)). (Die __init__-Methode nennt man **Konstruktor**, siehe unten). Anschließend definiert man die zugehörigen Objekte.

Betrachten wir die einzelnen Zeilen des Beispiels näher:

Zeile **class zwerg(frame):**

Eine neue **Klasse** namens **zwerg** wird erstellt.

Da die enthaltenen Objekte eine Gruppe bilden sollen, muss die neue Klasse über die Attribute und Methoden der Gruppenbildung verfügen.

Dies geschieht, indem sie diese von der Klasse frame (nicht zu verwechseln mit dem weiter oben genannten Attribut frame) erbt.

Eine Vererbung wird in Python implementiert, indem man die Oberklasse einfach in Klammern zu dem neuen Klassennamen schreibt.

Es ist auch möglich, eine Klasse von mehreren Oberklassen gleichzeitig erben zu lassen, dies nennt man dann **Mehrfachvererbung**.

Die Zeile wird durch einen Doppelpunkt abgeschlossen. Das bedeutet, dass anschließend die eigentlichen "Inhalte" der Klasse folgen. Diese müssen - wie alle Code-Blöcke - eingerückt werden.

Zeile `def __init__(self, groesse = 1, posx = 0):`

Hierbei wird der **Konstruktor** erzeugt. (engl. to construct = errichten).

Das Schlüsselwort **def** bewirkt, dass eine neue Methode definiert wird.

Unsere neue Methode heißt in diesem Fall `__init__`.

In Python wird ein Konstruktor einer Klasse immer `__init__` genannt. Diese Methode wird als allererstes beim Erstellen eines neuen Objekts aufgerufen. Somit können dann die Attribute mit ihren Werten gesetzt werden.

Der Konstruktor benötigt immer den Parameter '**self**' (engl. self = selbst). Damit ist der Selbstbezug, also das Objekt der Klasse selbst gemeint. Dies ist notwendig, um Attribute mit Hilfe der Punktnotation zu setzen. Bei der Punktnotation wird zunächst der Objektname, dann das Attribut genannt (getrennt durch einen Punkt). Da bei der Klassenerstellung der Name eines zugehörigen Objekts nicht bekannt ist, nennt man dieses '**self**'.

Als nächstes können Werte genannt werden, die schon im Konstruktor gesetzt werden sollen. Hier z. B. `groesse`, die den vorbelegten Wert 1 erhält und `posx`, vorbelegt mit Null. Es könnten auch noch weitere Parameter angegeben werden, die mit vordefinierten Werten oder auch ohne Wert belegt sind. Wichtig ist dabei nur, dass **zunächst die noch nicht belegten, danach die vordefinierten Parameter erscheinen**. Diese Parameter sind jedoch nicht automatisch auch Attribute der Klasse. Möchte man Attribute mit diesen Werten haben, muss man sie innerhalb des Konstruktors definieren.

Z. B. `def __init__(self, groesse =1, posx =0):`

```
...
    self.groesse = groesse
    self.positionx = posx
...
```

Hier werden zwei Attribute namens '`groesse`' bzw. '`positionx`' erstellt. Das erste Attribut, `groesse`, hat den gleichen Namen wie der Parameter `groesse` des Konstruktoraufrufs. Beim zweiten Attribut, `positionx`, wurde ein anderer Namen gewählt. Auf diese Attribute können dann alle weiteren Methoden der Klasse zugreifen. Dies wäre nicht der Fall, würde man die Parameterwerte nicht Attributen zuweisen. Ebenso können später deklarierte Objekte dieser Klasse darauf zugreifen.

Der Doppelpunkt am Ende der Zeile gibt wieder an, dass nun die durch den Konstruktor auszuführenden Zeilen folgen (wieder eingerückt).

Zeile `frame.__init__(self)`

Die Klasse `zwerg` erbt von der Klasse `frame`. Alle Oberklassen müssen zunächst auch erstellt werden, sie benötigen also auch einen Konstruktor. Als Parameter muss dabei nur der Selbstbezug, also `self` angegeben werden, es wären jedoch auch hier weitere Parameter denkbar.

Da dies keine neue Methode der Klasse `zwerg` ist, sondern der Methodenaufruf des Konstruktors der Klasse `frame`, folgt am Ende dieser Zeile natürlich kein Doppelpunkt.

Zeilen `self.koerper= cylinder(frame = self, pos = (posx,0,0),radius = groesse, axis = (0,3*groesse))`
`self.kopf = sphere(frame = self,pos = (posx,groesse*3.8,0), radius = groesse,`
`color = color.yellow)`
`self.hut = cone(frame = self,pos = (posx, groesse*4.5,0), radius = groesse*1.1,`
`axis = (0, groesse*3,0), color = color.red)`

Die Attribute `koerper`, `kopf` und `hut` der neuen Klasse `zwerg` werden mit Werten versehen. Alle Attribute sind in diesem Fall Objekte von `visual`-Klassen. Deshalb beginnen die Zeilen jeweils mit `self....=cylinder(...)`. (bzw. `sphere(...)` oder `cone(...)`).

Zunächst wird das Attribut frame der jeweiligen Visual-Klasse gesetzt. Alle 3D-Körper sollen zu einer Gruppe des neuen Objekts der Klasse zwerg gehören. Dies wird durch 'frame=self' erreicht.

Anschließend werden die weiteren Attributwerte analog oben (als der Zwerg noch keine Klasse war) gesetzt. Der einzige Unterschied dabei ist, dass die Position bzw. der Radius von posx bzw. groesse abhängig sind. Dadurch können unterschiedlich große Zwerge an verschiedenen Positionen der x-Achse erzeugt werden.

Zeilen `borin = zwerg(groesse = 1, posx = 0)`
`furgil = zwerg(groesse =0.75, posx = -3)`

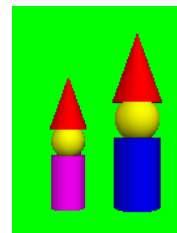
Diese beiden Zeilen sind nicht mehr eingerückt. Das heißt, sie gehören nicht mehr zur Klasse zwerg.

Durch diese Zeilen werden zwei Objekte der Klasse zwerg unterschiedlicher Größe und Position erzeugt.

Aufgabe06 (Klasse Zwerg):

Erzeuge analog obigem Beispiel eine Klasse Zwerg. Füge dem Konstruktor noch zusätzlich eine farbe, vorgelegt mit color.blue hinzu. Das soll die Farbe des Körpers sein.

Erzeuge mit dieser neuen Klasse anschließend zwei Zwerge, borin und furgil. furgil soll jedoch nicht blau, sondern magenta sein. Die übrigen Werte werden obigem Beispiel entnommen.



Einer Klasse weitere Methoden zufügen

Neben der Methode des Konstruktors kann eine Klasse natürlich noch weitere Methoden erhalten. Diese werden mit Hilfe des Schlüsselworts **def** erstellt und haben mindestens 'self' als Parameter.

Beispiel:

```
def essen(self):  
    ...
```

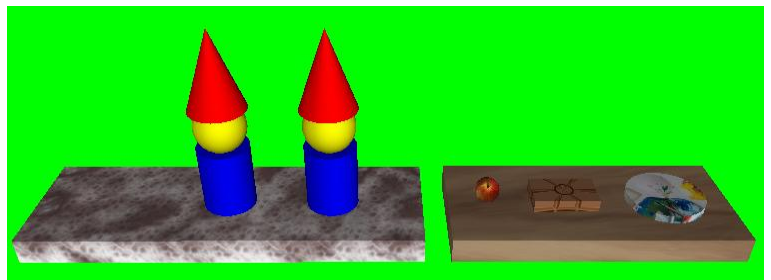
Aufgabe07 (Zwerg Methode abnehmen):

Öffne die Datei aufgabe07_zwerg_methode_abnehmen.py. Starte das Programm und versuche anhand der Implementierung und der Kommentare nachzuvollziehen, was das Programm bisher tut. (Die Mauseaktionen sind nicht ganz einfach zu verstehen, nimm sie also zunächst einmal als gegeben hin).

Eine Methode essen ist schon implementiert.

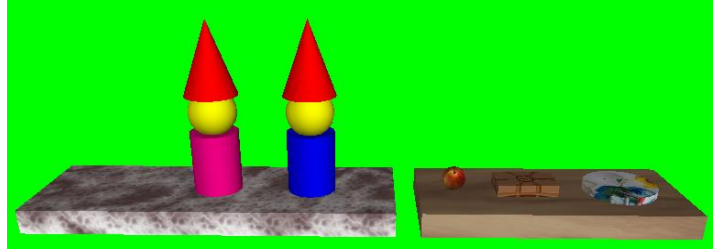
Schreibe an entsprechender Stelle (siehe Kommentare im Programm) eine Methode, die das Dickwerden des Zwerges wieder rückgängig macht.

Teste dein Programm.



Aufgabe08 (Zwerg Methode faerben):

Öffne die Datei
aufgabe08_zwerg_methode_faerben.py.
In dieser Datei ist u. a. auch die Lösung zu
vorangegangener Aufgabe zu finden.



Wird die Farbpalette auf einen Zwerg gezogen, so soll sich dessen Körper auf die vorgegebene Farbe hin verfärben. Die Farbe wird dabei zufällig ausgewählt. Diese zufällige Auswahl findet vor dem Methodenaufruf im Bereich der Mausektionen statt und ist schon implementiert. Es muss daher nur die eigentliche Methode faerben, die die Farbe als Parameter enthält, im Bereich der Klassendefinition des zwergs implementiert werden. An der entsprechenden Stelle ist ein entsprechender Kommentar zu finden.

Eine Klasse von einer selbst erstellten Klasse erben lassen

Öffne die Datei Aufgabe09_zwergenkind.py.
Hier wurde eine [weitere Klasse namens zwergenkind](#) erstellt. zwergenkind ist eine [Unterklasse](#) von zwerg, d. h. sie besitzt alle Attribute und Methoden von zwerg.
Die Klassendefinition und der Konstruktor lauten folgendermaßen:

```
class zwergenkind(zwerg):  
    def __init__(self, posx = 0, posy = 0, groesse = 1, farbe = color.blue):  
        zwerg.__init__(self, posx, posy, groesse, farbe)  
        self.kindgroesse=groesse  
        self.posy = posy
```

Zum Erzeugen eines Zwergenkindes müssen keine Anweisungen für das Erzeugen der 3D-Objekte erstellt werden. Diese Anweisungen wurden schon in der Klasse zwerg implementiert.

Der Konstruktor der neuen Klasse enthält in diesem Beispiel die gleichen Parameter wie die Oberklasse. Theoretisch wären jedoch auch noch weitere möglich.

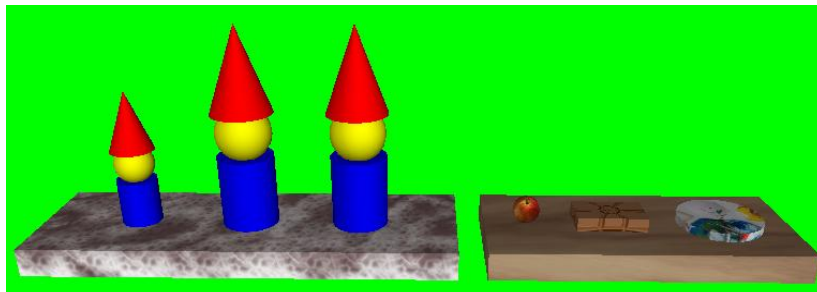
Die erste Anweisung im Konstruktor des zwergenkindes ist der Konstruktor des zwergs.

Die Klasse zwergenkind enthält die zusätzlichen Attribute kindgroesse und posy.

kindgroesse deshalb, um zu erkennen, ob das Zwergenkind schon ausgewachsen ist,
posy, um beim essen = wachsen die richtige vertikale Position gewährleisten zu können.

Die Methode essen ist schon implementiert. essen() wird analog zu der Klasse zwerg aufgerufen, wenn die Schokolade auf das Zwergenkind gezogen wird. Anstatt dick zu werden, wächst das Zwergenkind nun. Dies geschieht jedoch nur, wenn das Zwergenkind noch nicht ausgewachsen ist, das Attribut kindgroesse also kleiner als 1 (=Erwachsenengröße) ist.

Aufgabe09 zwergenkind:



Wird ein Apfel auf ein Zwergenkind gezogen, so wird die Methode abnehmen aufgerufen. In dieser soll dasselbe passieren wie beim "Schokoladeessen". Implementiere eine solche Methode abnehmen (die def-Anweisung ist schon vorgegeben).