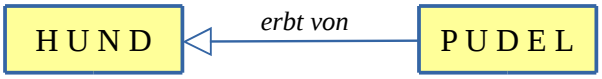
Vererbung – Teil 2

Wiederholung

Vererbung stellt man ***im Klassen-Diagramm*** durch einen **durchgezogenen Pfeil mit einem Dreieck als Spitze** dar.



Hiermit wird dargestellt, dass die Klasse *PUDEL* die Klasse *HUND* erweitert.
Man sagt auch, dass die Klasse *PUDEL* von der Klasse *HUND* erbt.

Die Klasse, von der geerbt wird (hier *HUND*), nennt man ***Superklasse***, die erbende Klasse (hier *PUDEL*) nennt man ***Subklasse***.

Man sagt auch: *PUDEL* ist eine ***Spezialisierung*** von *HUND* und meint damit, dass ein Pudel-Objekt nun durch die Vererbung auch ein spezielles Hund-Objekt ist. Anders herum betrachtet spricht man von einer ***Generalisierung*** und meint damit, dass ein Hund-Objekt ein allgemeineres Pudel-Objekt darstellt.

Deshalb nennt man die Vererbung auch eine ***„ist-ein“-Beziehung***.
Ein Pudel ist ein Hund.

Die Subklasse **erbt alle Attribute und Methoden** der Superklasse. Aus der Subklasse heraus **kann aber man NUR auf öffentliche Attribute und Methoden zugreifen!**

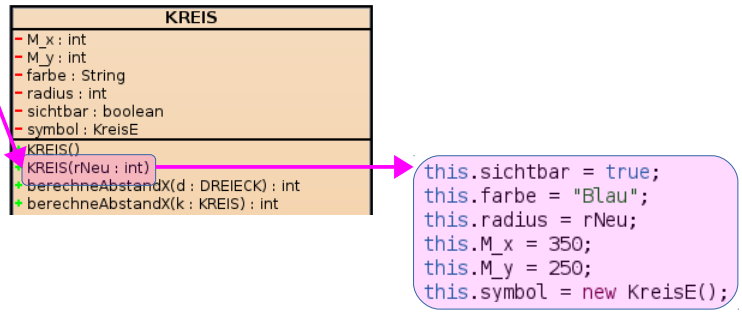
Eine Subklasse kann geerbte Methoden ***überschreiben***. Das bedeutet, dass man die Funktion dieser Methode neu gestaltet.

Beispiel 1:

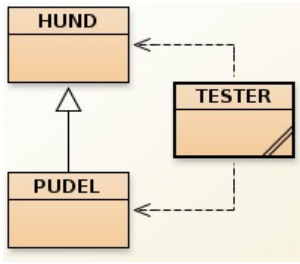
```
class BALL extends KREIS {

    BALL(int rNeu) {
        super(rNeu);

    }
}
```

*Klasse BALL **erbt von Klasse KREIS***

*Konstruktor-Methode geerbten **Superkonstruktor aufrufen***

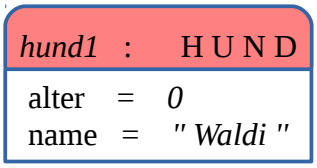


Entdeckungsreise ins Land des Sonderbaren ...



Wir schreiben nun zwei Klassen ***HUND*** und ***PUDEL***, wobei ***PUDEL*** von ***HUND*** erbt.
Der Reihe nach machen wir dann mit verschiedenen Objekten dieser Klassen Experimente und ergänzen immer wieder Code in den beiden Klassen.
Dabei werden wir einige interessante Erkenntnisse gewinnen …

1. Schreibe zunächst die **Klasse *HUND*** und initialisiere die Attribute wie in der Objekt-Karte beschrieben.




```
public class HUND {
      _____________________________
      _____________________________
      public HUND() {
            _____________________________________
            _____________________________________
      }
```

Außerdem soll es eine Methode `setzeAlter(...)` geben:

```
      public void setzeAlter( ____________________ ) {
            _____________________________________
      }
}
```

Erzeuge ein Objekt der Klasse ***HUND***, betrachte es im Objekt-Inspektor und teste die Methode `setzeAlter(...)`.

2. Schreibe nun die Klasse ***PUDEL***, welche von ***HUND*** erbt.

```
public class PUDEL ____________________

      public PUDEL() {
            ________________________
      }
```

Erzeuge nun ein Objekt der Klasse B und betrachte es im Objekt-Inspektor.
Führe bei offenem Objekt-Inspektor die Methode `setzeAlter(...)` aus und prüfe, was geschieht.

_____
_____

*Schreibe nun in der Klasse **PUDEL** die Methode `nenneAlter()`:*

```
public _____________________________ {
      ___________________________________
}
```

Was geschieht beim Übersetzen? Was bedeutet diese Fehlermeldung?

_____

_____

Folge:

_____

Überschreibe nun die Methode `setzeAlter(...)`:

```
@Override
public void setzeAlter(int alter) {
      if ( alter > this.alter ) {      // fehlerhafte Zeile
            this.alter = alter;        // fehlerhafte Zeile
      _____________________________    // richtige Zeile
            _____________________      // richtige Zeile
      }
}
```

Wieso klappt auch das zunächst nicht? Was bedeuten die Compiler-Meldungen?

_____

_____

Schreibe in der Klasse **HUND** eine Methode, die dieses Problem löst und verwende diese neue Methode dann in der Klasse **PUDEL**.

_____

_____

_____

*Erzeuge ein Objekt der Klasse **PUDEL** und teste die Methode `setzeAlter(...)`.*

3. Deklariere nun in der Klasse **PUDEL** ein zusätzliches Attribut `dressiert` vom Typ `boolean` und initialisiere es mit `false`.

```
public class PUDEL extends HUND {
      ________________________

       public PUDEL() {
            ...
            ____________________
      }
}
```

Schreibe außerdem die zugehörigen Setze- und Nenne-Methoden:

```
public _____________________________ {
      _____________________________
}
public _____________________________ {
      _____________________________
}
```

Was ist bei diesem eigenen Attribut anders als vorher bei den geerbten Attributen?

4. Schreibe nun eine Klasse *TESTER*, welche über drei Referenz-Attribute verfügt:

```
public class TESTER {
      private HUND hund_1;
      private PUDEL hund_2;
      private HUND hund_3;

      public TESTER() {
            this.hund_1 = new HUND();
            this.hund_2 = new PUDEL();
            this.hund_3 = new PUDEL();
      }
}
```

Sieh dir die dritte Initialisierung genau an. Beachte den Typ.
Was fällt auf? Wird das beim Übersetzen beanstandet?

Versuche mithilfe der Begriffe Spezialisierung / Generalisierung zu begründen, wieso das kein Fehler ist.

5. Versuche nun folgende Fragen erst **mündlich** zu beantworten, **bevor** du den entsprechenden Code in einer Methode `test()` in der Klasse `TESTER` ausprobierst und die Objekte `hund_1`, `hund_2` und `hund_3` im Objekt-Inspektor genau betrachtest. **Nach dem Test** füllst du die Zeilen **schriftlich** aus.

   - *Wie reagiert `hund_1` auf Aufrufe der Methode `setzeAlter(...)` bei negativen Werten des Übergabe-Parameters?*

     _____

     _____

   - *Wie reagiert `hund_2` auf Aufrufe der Methode `setzeAlter(...)` bei negativen Werten des Übergabe-Parameters?*

     _____

     _____

   - *Wie reagiert `hund_3` auf Aufrufe der Methode `setzeAlter(...)` bei negativen Werten des Übergabe-Parameters?*

     _____

     _____

   - *Wie reagiert `hund_1` auf Aufrufe der Methode `nenneDressiert()` ?*

     _____

     _____

   - *Wie reagiert `hund_2` auf Aufrufe der Methode `nenneDressiert()` ?*

     _____

     _____

   - *Wie reagiert `hund_3` auf Aufrufe der Methode `nenneDressiert()` ?*

     _____

     _____

Überschreiben von Methoden genauer betrachtet

Bisher hatten wir die Funktion einer geerbten Methode immer gänzlich neu gestaltet. Diese Vorgehensweise nennt man ***überschreiben einer Methode***. Man kann aber auch erst den geerbten Code aufrufen und diesen anschließend um weitere Funktionalität ergänzen. Bei dieser Vorgehensweise spricht man dann von ***erweiternden einer Methode***.

Beispiel 1:

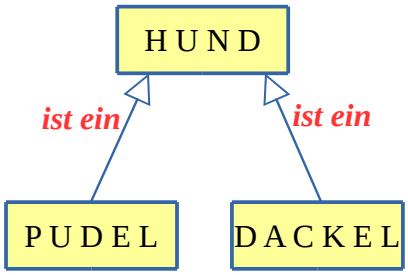
```
@Override
void tuwas(...) {
      super.tuWas(...);
      // tu noch was anderes;
}
```

Zuerst den geerbten Code aufrufen. Erfolgt dieser Aufruf nicht, so wird die Methode völlig neu gestaltet.

Polymorphie – Vielgestaltigkeit

Du hast in der HUND-Übung gesehen, dass einem Attribut des Supertyps *(Klasse HUND)* auch ein Objekt des Subtyps *(Klasse PUDEL)* zugewiesen werden kann.

Es ist aber ganz offensichtlich, dass jeder Dackel oder Pudel auch allgemein betrachtet ein Hund ist.



Unter dem Begriff ***Polymorphie*** versteht man, dass ein Objekt der Subklasse immer auch als ein Objekt der Superklasse angesehen werden kann.
So ist z.B. auch jeder Mensch ein Säugetier.

Da jedes Objekt der Unterklasse auch als ein Objekt der Oberklasse betrachtet werden kann, nennt man die Vererbung auch ***„ist-ein“-Beziehung***.
Jedes Objekt der Unterklasse ist eben auch ein Objekt der Oberklasse.

Gibt es nun Methoden in der Superklasse, die in einer Subklasse überschreiben worden sind, so wird bei einem Objekt immer die Variante ausgeführt, welche in der Vererbung am weitesten unten anzutreffen ist.

Bspl.: `private HUND hund_3 = new PUDEL();`
`setzeAlter(...)` wird hier immer in der Variante der Klasse Pudel ausgeführt.

Diese Tatsache fasst man unter dem Begriff ***dynamische Bindung*** zusammen und meint damit, dass im Zweifelsfall immer die speziellere Variante einer Methode ausgeführt wird.

Verfügt allerdings die Unterklasse über eine neue Methode, die in der Oberklasse noch nicht enthalten ist, so kann diese Methode auch nur ausgeführt werden, wenn der Typ dazu passt.

Bspl.: `nenneDressiert()` kann bei *hund_1* und *hund_3* vom Typ *HUND* nicht ausgeführt werden, da deren Typ *HUND* und nicht *PUDEL* ist, auch wenn bei *hund_3* sogar explizit ein *PUDEL* erzeugt wurde *(der Typ ist dennoch HUND)*.

Wenn von einem Objekt nur bekannt ist, dass es allgemein ein Hund ist, so kann man nicht pauschal davon ausgehen, dass es über Methoden z.B. eines Zirkus-Hundes verfügt. Hierzu müsste man erst sicher stellen, dass es auch wirklich ein Zirkus-Hund ist.
Das könnte man durch einen ***Typ-Cast*** versuchen: `(PUDEL)hund_3` *analog* `(int)1.24`

Anwendung – Kamel-Simulation

(Nach einer Idee von Christian Wendl)

Normale Kamele haben die Eigenschaften *Alter*, *Gewicht*, *Name* und *Darstellung* (für die Darstellung wählen wir ein WECHELSBILD). Durch Züchtung werden Kamele für verschiedene Aufgaben spezialisiert. Man erhält zum Beispiel ein Transportkamel, welches besser dafür geeignet ist, eine große Last über weite Strecken zu transportieren. Transportkamele zeichnen sich zudem durch ihre *maximale Traglast* aus. Eine weitere Spezialisierung des normalen Kamels ist das Rennkamel. Dieses ist besonders schnell, jedoch hat es weniger Ausdauer und somit eine geringere Reichweite. Ein Rennkamel zeichnet sich zudem durch seine *Höchstgeschwindigkeit* aus.

KAMEL	
- alter :	int
- gewicht :	int
- name :	String
- darstellung:	WECHSELBILD
. . .	
. . .	

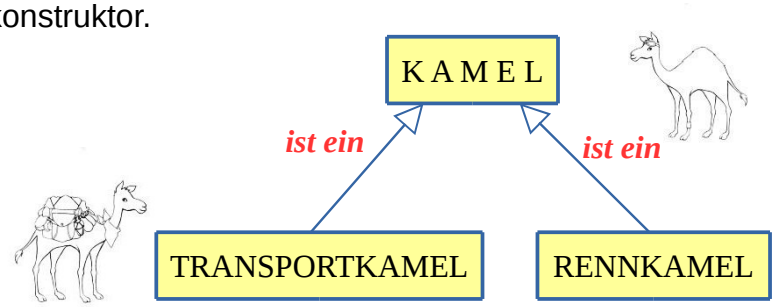
RENNKAMEL	
- alter :	int
- gewicht :	int
- name :	String
- darstellung:	WECHSELBILD
- **maxGeschw:**	**int**
. . .	
. . .	

TRANSPORTKAMEL	
- alter :	int
- gewicht :	int
- name :	String
- darstellung:	WECHSELBILD
-maxLast :	**int**
. . .	
. . .	

1. Schreibe eine Klasse ***KAMEL*** und vereine darin alle Attribute, über die jeder Kameltyp verfügt. Verwende für alle Attribute den Modifikator `protected`.
   (Solche Attribute können in Subklassen direkt bearbeitet werden, dennoch sind die Attribute nicht öffentlich.)
   Schreibe außerdem einen Konstruktor, der alle Attribute durch Übergabe-Parameter initialisiert.

   Erzeuge ein Objekt und betrachte es genau im Objekt-Inspektor.

2. Schreibe nun zwei Unterklassen ***RENNKAMEL*** und ***TRANSPORTKAMEL***. Jede Klasse soll ihre speziellen neuen Attribute deklarieren. Der Konstruktor soll alle Attribute (die geerbten und die neuen) durch Übergabe-Parameter initialisieren. Nutze dabei zuerst den Superkonstruktor.



Erzeuge Objekte von jeder Klasse und betrachte sie genau im Objekt-Inspektor.

3. Schreibe in der Klasse KAMEL eine Methode ***informationAusgeben()***, welche mithilfe von vier `System.out.println("...")` – Befehlen folgende Ausgabe auf der Konsole bewirkt:

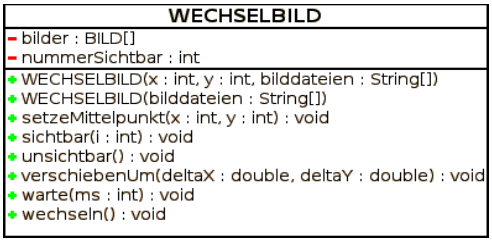
   eine Leerzeile ausgeben
   Ich heiße …
   Ich bin … Jahre alt
   Ich wiege … kg

   Überschreibe diese Methode nun (erweiternd) in den beiden Subklassen. Rufe zuerst den geerbten Code auf und ergänze dann um eine weitere Ausgabe-Zeile:

   Ich kann bis zu … km/h schnell laufen
   bzw. *Ich kann bis zu … kg tragen*

   Erzeuge Objekte von jeder Klasse und teste jeweils die Ausgabe der neuen Methode.

4. Erzeuge eine Klasse ***WUESTE***, welche von *SPIEL* erbt.
   Die Klasse soll ein Referenz-Attribut namens `kamele` vom Typ KAMEL-Array haben.



   Der **Konstruktor** soll der Reihe nach folgendes erledigen:

   - ► Zuerst den Superkonstruktor aufrufen und ein Spielfeld der Breite 800, Höhe 600 und ohne Punktestände und ohne Maus erzeugen.
   - ► Als Hintergrundbild nimmst du das Bild `wueste.jpg`.
   - ► Dann soll das Array mit der Länge drei initialisiert werden.
   - ► Nun wird in je einem eigenen Befehl (ohne Schleife) jedem Speicherplatz des Arrays ein neues Kamel bzw. Rennkamel bzw. Transportkamel zugewiesen.
   - ► Zum Schluss soll das Array mit einer Schleife durchlaufen werden. Im Rumpf der Schleife soll jedes Kamel seine Information ausgeben.

   Bemerkung: Du wirst an dieser Stelle noch keine Kamele sehen.
   Auf der Konsole sollten sie sich dennoch bereits vorstellen.

   Erzeuge ein Objekt der Klasse WUESTE.

   Erscheint das Hintergrund-Bild?

   Stellen sich deine drei Kamele ordentlich im Konsolen-Fenster vor?

Nun kümmern wir uns um die ***Kamel-Darstellung***.
Hierzu steht dir die Klasse ***WECHSELBILD*** zur Verfügung. Dabei handelt es sich vereinfacht gesagt um eine Folge von Bildern, die gegeneinander ausgetauscht werden können. So können z.B. unsere Kamele die Beine bewegen, wenn sie später laufen sollen …

Wechselbild erzeugen

Bevor du ein Wechselbild erzeugen kannst,musst du ein **String-Array** anlegen. Nenne es z.B. *dateinamen*. Die Länge des Arrays entspricht der Anzahl der Bild-Dateien. Speichere deine **Bilddateien (GIF, JPG, PNG) im BlueJ-Projekt-Ordner**. Speichere den Namen der Bilddateien nun in den Zellen des String-Arrays. Dieses String-Array wird im Konstruktor von WECHSELBILD als (dritter) Parameter übergeben.

unsichtbar()

Die Darstellung des Bildes im Grafik-Fenster wird gelöscht. Das Objekt ist aber noch da und kann auch (unsichtbar) bewegt werden.

sichtbar(int i)

Das Bild mit Index i im String-Array wird angezeigt. Nur sinnvoll, wenn du das Wechselbild-Objekt vorher unsichtbar gemacht hast.

wechseln()

Löscht die gerade angezeigte Bilddatei vom Grafik-Fenster und zeigt stattdessen das nächste Bild im String-Array an. Ist das letzte Bild im Array erreicht, so kommt automatisch wieder das erste Bild usw.

warte(int ms)

Veranlasst, dass der darauf folgende Befehl zeitverzögert ausgeführt wird. Du übergibst die Verzögerung in Millisekunden.

Für jedes Kamel stehen zunächst zwei **Bild-Dateien** zur Verfügung,

z.B. ***kamel_n_1.gif*** und ***kamel_n_2.gif***

*(**n**: normales Kamel, **r**: Rennkamel, **t**:Transportkamel).*

→ Deklariere in der Klasse ***KAMEL*** ein ***String-Array*** names ***dateinamen***.
Verwende als Modifikator ***protected***.
(Darin speichern wir die namen der benötigten Bilder für das Kamel.)
Zusätzlich benötigt die Klasse *KAMEL* nun auch noch ein **Referenz-Attribut** namens ***darstellung*** vom **Typ *WECHSELBILD***.

→ Im **Konstruktor** von *KAMEL* fügst du nun folgendes hinzu:
- ► Initialisiere das String-Array mit der Länge 2.
- ► Speichere in jedem Speicherplatz des Arrays je einen Bild-Namen.
- ► Weise der Darstellung ein neues Wechselbild zu. *(bei x=100, y=200)*

*(Konstruktor **s. Klassen-Karte von WECHSELBILD**)*

Erzeuge ein Objekt der Klasse KAMEL.
Es sollte sich jetzt in einem schwarzen Fenster zeigen.

Werden die speziellen Kamele nun auch schon grafisch dargestellt? Begründe!

---

5. Die Spezialisierungen des normalen Kamels sollen natürlich auch anders aussehen. Deshalb musst du **in jeder Subklasse von KAMEL** nun **gleich nach dem Aufruf des Superkonstruktors** die geerbte Darstellung beseitigen und durch eine spezielle andere ersetzen. Gehe dazu folgendermaßen vor:

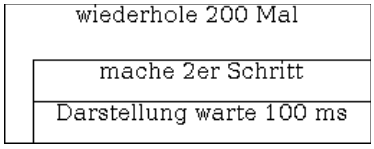
- ► Mache gleich nach dem Superkonstruktor-Aufruf die geerbte Darstellung unsichtbar. ***(s. Klassen-Karte von WECHSELBILD)***
- ► Speichere in den Zellen des geerbten String-Arrays die neuen Datei-Namen.
- ► Weise der Darstellung eine neues Wechselbild-Objekt zu. *(bei x=100, y=350 bzw. 500)*

Erzeuge ein Objekt der Klasse WUESTE.
Nun solltest du drei verschiedene Kamele sehen.

6. Die Kamele sollen über die Wüstenlandschaft laufen können.
   Hierzu schreiben wir in der **Klasse KAMEL** zuerst eine Methode `schritt(int n)`, welche ein Kamel um n Pixel verschiebt und dabei das Kamel-Bild wechselt, so dass es aussieht, als ob sich die Füße des Kamels bewegen.
   In einer weiteren Methode `laufen()` wird dann `schritt(...)` wiederholt innerhalb einer Schleife aufgerufen.

   → Schreibe in der Klasse KAMEL eine **`Methode schritt(int n)`**, welche folgendes leistet: *(s. Klassen-Karte von WECHSELBILD)*
   - ► Das Referenz-Attribut `darstellung` um n Pixel nach rechts verschieben.
   - ► Das angezeigte Bild des Referenz-Attributs `darstellung` wechseln.

   → Schreibe in der Klasse KAMEL eine **Methode `laufen()`**, welche folgendes leistet: *(s. Klassen-Karte von WECHSELBILD)*



   Erzeuge ein Objekt der Klasse KAMEL und teste die Methode `laufen()`.

   Können nun auch schon Objekte der Subklassen laufen? Begründe deine Antwort.

   _____

7. Natürlich laufen Rennkamele schneller und Transportkamele weiter als normale Kamele.

   Durch welche Programmier-Technik kannst du das realisieren?

   _____

   Betrachte die Methode `laufen()` noch einmal ganz genau!
   Was musst du ändern, damit das Kamel mehr / weniger einzelne Schritte macht?

   _____

   Was musst du ändern, damit das Kamel kleinere / größere Schritte macht?

   _____

   Was musst du ändern, damit das Kamel mehr / weniger Schritte pro Zeiteinheit macht?

   _____

   Überschreibe nun die Methode `laufen()` in den Subklassen entsprechend.

8. Natürlich brauchen auch Kamele regelmäßig ihren Schlaf. Damit du diese Aufgabe ganz auskosten kannst, brauchst du einen Kopfhörer z.B. vom Handy, iPod, …

Deklariere in der Klasse KAMEL ein Referenz-Attribut vom Typ *BILD (nicht WECHSELBILD)* und nenne es `schlafBild`. Wähle als Modifikator `protected`.
Initialisiere es im Konstruktor an der Stelle (100|200) mit der Grafikdatei `kamel_n_s.gif`.
Setze dieses Bild gleich im nächsten Befehl unsichtbar.
(s. Klassen-Karte von BILD)

BILD
+ BILD(x : int, y : int, name : String) + beinhaltetPunkt(x : int, y : int) : boolean + cos_Drehwinkel() : double + drehenUm(winkelAenderung : int) : void + nenneMx() : int + nenneMy() : int + nenneSichtbar() : boolean + nenneWinkel() : int + setzeDrehwinkel(neuerDrehwinkel : int) : void + setzeMittelpunkt(x : int, y : int) : void + setzeSichtbar(sichtbarNeu : boolean) : void + sin_Drehwinkel() : double + verschiebenUm(deltaX : double, deltaY : double) : void

Deklariere außerdem ein Referenz-Attribut vom Typ *SOUND* und nenne es `sound`.
Denke an den Modifikator `protected`.
Initialisiere es im Konstruktor.

SOUND
- is : InputStream - player : Player
+ SOUND() + play(soundfile : String) : void

play(String sounddatei)
Ein Sound-Objekt verfügt über die Methode zum Abspielen einer mp3-Datei. Diese musst du im BlueJ-Projekt-Ordner speichern und der Methode den Dateinamen als Parameter übergeben.

Schreibe eine Methode `schlafen()`,
welche folgendes leistet:

(s. Klassen-Karten von BILD und WECHSELBILD)

Darstellung unsichtbar
Schlafbild sichtbar
Sound play schlafen.mp3
Schlafbild unsichtbar
Darstellung sichtbar mit Bild 0

Erzeuge ein Objekt der Klasse KAMEL.

Stecke deine Kopfhörer an und teste die Methode `schlafen()`.

Ändert das Kamel sein Aussehen? Hörst du das Kamel atmen?

9. Rennkamele sind völlig überzüchtet und haben deshalb genetisch bedingt eine schiefe Nasenscheidewand, wodurch sie beim Schlafen fürchterlich schnarchen.

Überschreibe die Methode in der Subklasse RENNKAMEL entsprechend.

10. Nun wollen wir alle Kamele gemeinsam bewegen oder schlafen schicken.

In der ***Klasse WUESTE*** ist eine **Methode `laufenLassen()`** vorgegeben. Darin befinden sich bereits einige Zeilen an JAVA-Code, den du allerdings erst in der Qualifikationsphase in Informatik verstehen lernen wirst.
Wenn du den drei Kamelen nacheinander sagst, dass sie laufen sollen, dann wird sich das zweite Kamel erst in Bewegung setzen, wenn das erste Kamel fertig ist mit laufen usw. Der vorgegeben Code veranlasst, dass die Kamele gleichzeitig laufen werden. Ersetze also nur die entsprechenden Kommentare im Rumpf der Methode durch entsprechende Methodenaufrufe.

WICHTIG:
Lasse bei der Verwendung des Kamel-Arrays in den Methoden `laufenLassen()` und `schlafenLassen()` das `this.` weg !!! Sonst bekommst du für dich unverständliche Fehlermeldungen ...

Außerdem gibt es eine vorbereitete ***Methode `schlafenLassen()`***, in der du analog vorgehst.

Erzeuge ein Objekt der Klasse WUESTE und teste die neuen Methoden.

Wiederhole nun das theoretische Wissen zu

- ***Vererbung,***
- ***Überschreiben von Methoden,***
- ***Polymorphie,***
- ***dynamischer Bindung***