Leben und Sterben eines Objekts

Objekte werden geboren und Objekte sterben. *Sie* herrschen über das Leben eines Objekts. *Sie* entscheiden, wie und wann ein Objekt **konstruiert** wird. *Sie* entscheiden, wann es **zerstört** wird. Nur *zerstören* Sie das Objekt nicht wirklich selbst, Sie *geben* es einfach auf. Aber ist es einmal aufgegeben, kann es der gnadenlose **Garbage Collector (GC)** pulverisieren und den Speicher wieder anfordern, den das Objekt belegt hat. Wenn Sie Java schreiben, erzeugen Sie Objekte. Früher oder später werden Sie einige von ihnen loslassen müssen, oder Sie riskieren, dass Ihnen das RAM ausgeht. In diesem Kapitel sehen wir uns an, wie Objekte erzeugt werden, wie sie leben, während sie leben, und wie Sie sie effektiv bewahren oder aufgeben. Das bedeutet, dass wir über den Heap, den Stack, Geltungsbereiche, Konstruktoren, Super-Konstruktoren, null-Referenzen und anderes reden werden. Warnung: Dieses Kapitel enthält Szenen zum Tod von Objekten, die für manche verstörend sein können! Besser, Sie bauen keine zu engen Bindungen auf.

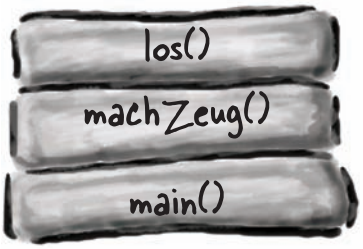
Der Stack und der Heap: wo das Leben spielt

Bevor Sie verstehen können, was wirklich passiert, wenn Sie ein Objekt erzeugen, müssen wir ein paar Schritte zurückgehen. Wir brauchen noch etwas mehr Informationen dazu, wo (und wie lange) die Dinge in Java leben. Das bedeutet, dass wir noch etwas mehr über den Stack und den Heap erfahren müssen. In Java interessieren wir (Programmierer) uns für zwei Speicherbereiche – den, in dem die Objekte leben (der Heap), und den, in dem die Methodenaufrufe und lokale Variablen leben (der Stack). Wenn eine JVM gestartet wird, wird ihr vom zu Grunde liegenden Betriebssystem ein Speicherbereich zugewiesen, den sie verwendet, um Ihr Java-Programm auszuführen. Wie *viel* Speicher das ist und ob Sie diese Menge steuern können oder nicht, ist von der JVM-Version (und der Plattform) abhängig, mit der Sie arbeiten. Aber in der Regel *müssen* Sie dazu nichts sagen. Und wenn Sie Ihr Programm ordentlich schreiben, brauchen Sie sich darüber keine Sorgen zu machen (mehr dazu gibt's etwas später).

Wir wissen, dass alle *Objekte* auf dem Garbage Collectible Heap leben, aber wir haben uns noch nicht angesehen, wo *Variablen* leben. Dies hängt davon ab, was für eine *Art* Variable sie ist. Mit »Art« meinen wir hier nicht den *Typ* (d.h. elementar oder Referenztyp). Die zwei *Arten* von Variablen, die uns jetzt interessieren, sind *Instanzvariablen* und *lokale Variablen*. Lokale Variablen werden auch als *Stack*-Variablen bezeichnet – und das ist ja schon ein freundlicher Hinweis darauf, wo sie leben.

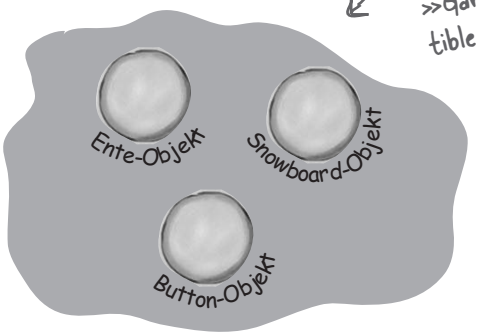
Der Stack

Wo Methodenaufrufe und lokale Variablen leben



Der Heap

Wo ***ALLE*** Objekte leben



Instanzvariablen

Instanzvariablen werden in einer *Klasse*, aber nicht in einer *Methode* deklariert. Sie repräsentieren die »Felder«, die die einzelnen Objekte haben (und die bei jeder Instanz der Klasse jeweils mit unterschiedlichen Werten gefüllt sein können). Instanzvariablen leben innerhalb des Objekts, zu dem sie gehören.

```
public class Ente {

    int größe;

}
```

Jede Ente hat eine Instanzvariable »größe«.

Lokale Variablen

Lokale Variablen (einschließlich Methodenparametern) werden in einer *Methode* deklariert. Sie sind temporär und leben nur so lange, wie die Methode auf dem Stack ist (d.h., solange die Methode noch nicht an der schließenden geschweiften Klammer angekommen ist).

```
public void foo(int x) {

    int i = x + 3;

    boolean b = true;

}
```

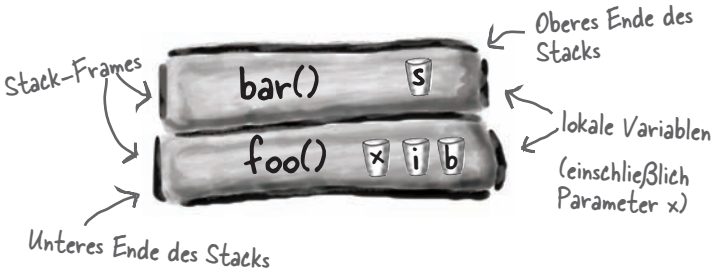
Der Parameter x und die Variablen i und b sind lokale Variablen.

Methoden werden gestapelt

Wenn Sie eine Methode aufrufen, kommt die Methode oben auf den Aufruf-Stack. Das neue Ding, das tatsächlich auf den Stack geschoben wird, ist der Stack-*Frame*, und er enthält den Status der Methode. Dieser hält fest, welche Zeile gerade ausgeführt wird und welche Werte alle lokalen Variablen haben.

Die Methode, die sich *oben* auf dem Stack befindet, ist die auf diesem Stack gerade ablaufende Methode (im Augenblick gehen wir mal davon aus, dass es nur einen Stack gibt, aber in Kapitel 15 werden wir mehr hinzufügen). Eine Methode bleibt so lange auf dem Stack, bis die schließende geschweifte Klammer der Methode erreicht wird (die sagt, dass die Methode fertig ist). Wenn die Methode *foo()* die Methode *bar()* aufruft, wird die Methode *bar()* auf die Methode *foo()* gestapelt.

Ein Aufruf-Stack mit zwei Methoden



Die oberste Methode auf dem Stack ist immer die, die aktuell ausgeführt wird.

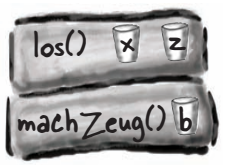
```
public void machZeug() {
   boolean b = true;
   los(4);
}
public void los(int x) {
   int z = x + 24;
   verrückt();
   // tun Sie so, als käme hier noch mehr
}
public void verrückt() {
   char c = 'a';
}
```

Ein Stack-Szenario

Der Code auf der linken Seite ist ein Schnipsel mit drei Methoden. (Wie der Rest der Klasse aussieht, interessiert uns nicht.) Die erste Methode (*machZeug()*) ruft die zweite Methode (*los()*) auf, und die zweite Methode ruft die dritte (*verrückt()*) auf. Im Body jeder Methode wird eine lokale Variable deklariert. Die Methode *los()* deklariert außerdem eine Parametervariable (was bedeutet, dass *los()* zwei lokale Variablen hat).

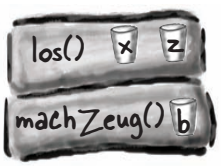
1. Code von einer anderen Klasse ruft **machZeug()** auf, und **machZeug()** kommt in einen Stack-Frame oben auf den Stack. Die boolean-Variable namens **b** kommt in den Stack-Frame für **machZeug()**.

2. **machZeug()** ruft **los()** auf. **los()** wird oben auf den Stack *geschoben*. Die Variablen **x** und **z** befinden sich im Stack-Frame für **los()**.



3. **los()** ruft **verrückt()** auf. Jetzt ist **verrückt()** oben auf dem Stack, und im entsprechenden Frame befindet sich die Variable **c**.

4. **verrückt()** ist abgeschlossen, und ihr Stack-Frame wird vom Stack *geschoben*. Die Ausführung kehrt zur Methode **los()** zurück und setzt an der Zeile wieder ein, die auf den Aufruf von **verrückt()** folgt.



Was ist mit lokalen Variablen, die Objekte sind?

Denken Sie daran, dass eine Variable eines nicht-elementaren Typs eine *Referenz* auf ein Objekt hält, nicht das Objekt selbst. Sie wissen bereits, wo Objekte leben – auf dem Heap. Es spielt keine Rolle, wo sie deklariert oder erzeugt werden. ***Wenn die lokale Variable eine Referenz auf ein Objekt ist, kommt nur die Variable (die Referenz/Fernsteuerung) auf den Stack.***

Das Objekt selbst kommt trotzdem auf den Heap.

```
public class StackRef {
   public void foof() {
      barf();
   }

   public void barf() {
      Ente e = new Ente(24);
   }
}
```

Es gibt keine Dummen Fragen

F: Noch mal. WARUM lernen wir diese Stack/Heap-Geschichte? Was bringt mir das? Muss ich das wirklich wissen?

A: Dass Sie die Java-Grundlagen zu Stack und Heap kennen, ist wichtig, wenn Sie Dinge wie Geltungsbereiche von Variablen, Probleme der Objekterzeugung, Speicherverwaltung, Threads und Exception-Handling verstehen wollen. Threads und Exception-Handling werden wir in späteren Kapiteln behandeln. Aber die anderen werden Sie in diesem Kapitel kennen lernen. Sie müssen nicht wissen, *wie* Stack und Heap in einer bestimmten JVM und/oder auf einer bestimmten Plattform implementiert sind. Alles, was Sie zu Stack und Heap wissen müssen, steht auf dieser und der vorangegangenen Seite. Wenn Sie diese beiden Seiten verstanden haben, kommen Sie mit all den Themen, die von diesem Stoff abhängen, viel, viel leichter klar. Noch mal: Eines Tages werden Sie uns dafür dankbar sein, dass wir Sie mit Stacks und Heaps gefüttert haben.

Punkt für Punkt

- Java hat zwei Speicherbereiche, die uns interessieren: den Stack und den Heap.
- Instanzvariablen werden in einer Klasse, aber außerhalb aller Methoden deklariert.
- Lokale Variablen werden in einer Methode oder als Methodenparameter deklariert.
- Alle lokalen Variablen leben in dem Stack-Frame, der zu der Methode gehört, in der die Variablen deklariert wurden.
- Objektreferenzvariablen funktionieren genau so wie Variablen eines elementaren Typs – wenn die Referenz als lokale Variable deklariert wird, kommt sie auf den Stack.
- Alle Objekte leben auf dem Heap, egal ob die Referenz eine lokale Variable oder eine Instanzvariable ist.

Wenn lokale Variablen auf dem Stack leben, wo leben dann Instanzvariablen?

Wenn Sie new Handy() sagen, muss Java auf dem Heap Platz für das neue Handy schaffen. Aber wie *viel* Platz? Genug Platz für das Objekt! Und das heißt genug Platz, um alle Instanzvariablen des Objekts zu beherbergen. Sie haben richtig gehört: Instanzvariablen leben auf dem Heap, und zwar innerhalb der Objekte, zu denen sie gehören.

Denken Sie daran, dass die *Werte* der Instanzvariablen eines Objekts innerhalb dieses Objekts leben. Wenn alle Instanzvariablen elementare Variablen sind, erzeugt Java den Platz für die Instanzvariablen auf Basis der elementaren Typen. Ein int braucht 32 Bits, ein long 64 Bits usw. Um den Wert in elementaren Variablen kümmert sich Java nicht. Die Bit-Größe eines ints bleibt immer gleich (32 Bits), egal ob der Wert 32.000.000 oder 32 ist.

Aber was ist, wenn die Instanzvariablen *Objekte* sind? Was ist, wenn Handy eine Antenne-Instanzvariable hat?

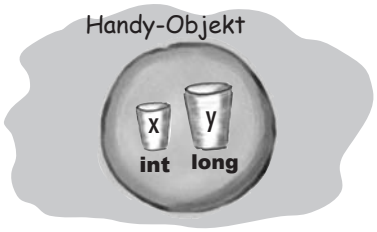
Hat das neue Objekt Instanzvariablen, die Objektreferenzen und keine elementaren Variablen sind, ist die eigentliche Frage, ob das Objekt wirklich Platz für die kompletten Objekte braucht, die es referenziert. Die Antwort ist: *nicht ganz*. Egal, worum es sich im Einzelnen handelt, Java muss Platz für die *Werte* der Instanzvariablen schaffen. Aber denken Sie daran, dass eine Referenzvariable nicht das ganze *Objekt* ist, sondern nur eine *Fernsteuerung* für das Objekt. Wenn Handy eine Instanzvariable mit dem nicht elementaren Typ Antenne hat, macht Java im Handy-Objekt nur Platz für die *Fernsteuerung* (d.h. Referenzvariable) für Antenne, aber nicht für das Antenne-*Objekt* selbst.

Aber wann erhält dann das Antenne-*Objekt* seinen Platz auf dem Heap? Dazu müssen wir erst herausfinden, *wann* das Antenne-Objekt selbst erzeugt wird. Das ist von der Deklaration der Instanzvariablen abhängig. Wenn die Instanzvariable deklariert, ihr aber kein Objekt zugewiesen wird, wird nur der Platz für die Referenzvariable (die Fernsteuerung) geschaffen.

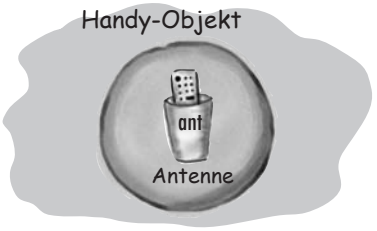
```
private Antenne ant;
```

Auf dem Heap wird kein Antenne-Objekt erzeugt, bis der Referenzvariablen ein neues Antenne-Objekt zugewiesen wird.

```
private Antenne ant = new Antenne();
```

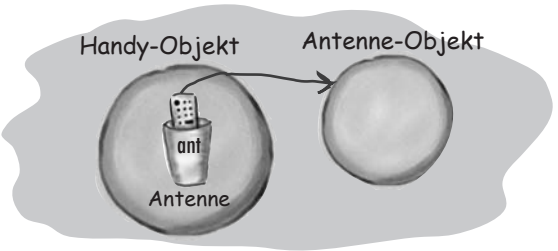


Ein Objekt mit zwei elementaren Instanzvariablen. Im Objekt ist der Platz für die beiden Variablen vorhanden.



Ein Objekt mit einer nicht elementaren Instanzvariablen – einer Referenz auf ein Antenne-Objekt, aber keinem tatsächlichen Antenne-Objekt. Das erhalten Sie, wenn Sie die Variable deklarieren, aber nicht mit einem tatsächlichen Antenne-Objekt initialisieren.

```
public class Handy {
  private Antenne ant;
}
```



Ein Objekt mit einer nicht elementaren Instanzvariablen, und der Antenne-Variablen wird ein neues Antenne-Objekt zugewiesen.

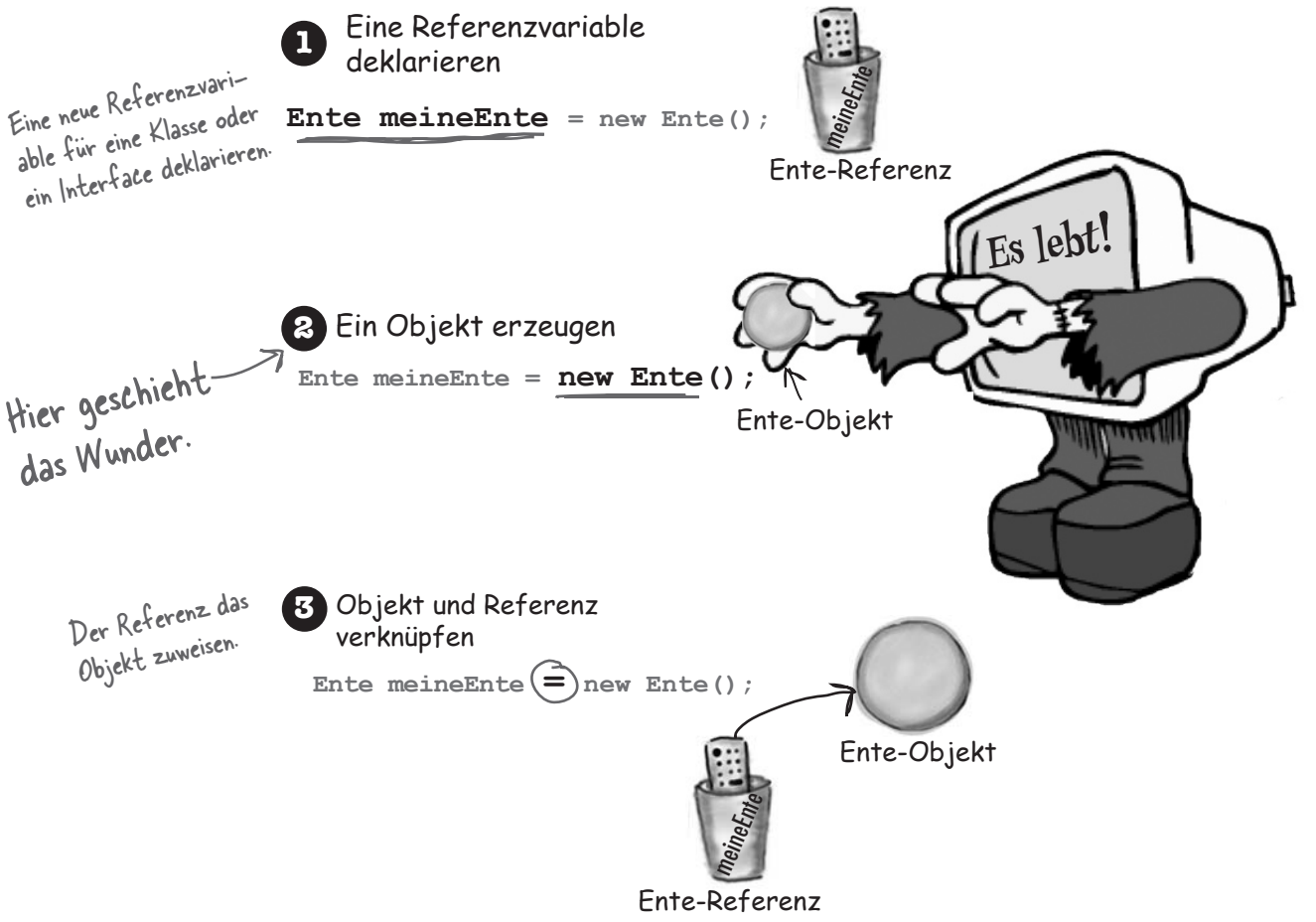
```
public class Handy {
  private Antenne ant = new Antenne();
}
```

Das Wunder der Objekterzeugung

Jetzt, da Sie wissen, wo Variablen und Objekte leben, können wir in die mysteriöse Welt der Objekterzeugung eintauchen. Erinnern Sie sich an die drei Schritte der Objekterzeugung: die Deklaration einer Referenzvariablen, das Erzeugen eines Objekts und die Zuweisung des Objekts zu der Referenz.

Aber bis jetzt blieb Schritt zwei – in dem das Wunder geschieht und das neue Objekt »geboren« wird – ein großes Mysterium. Bereiten Sie sich darauf vor, alles zum Leben von Objekten zu erfahren. *Hoffentlich sind Sie kein Weichei.*

Sehen wir uns die drei Schritte, Objektdeklaration, -erzeugung und -zuweisung, an:



Rufen wir eine Methode namens Ente() auf? So *sieht* es nämlich *aus*.

```
Ente meineEnte = new Ente();
```

Es sieht aus, als würden wir eine Methode namens Ente aufrufen, da sind schließlich auch die Klammern.

Nein.

Wir rufen den Ente-*Konstruktor* auf.

Ein Konstruktor *sieht* einer Methode sehr ähnlich, ist aber keine Methode. Er enthält den Code, der ausgeführt wird, wenn Sie **new** sagen. Anders gesagt: *den Code, der ausgeführt wird, wenn Sie ein Objekt instantiieren.*

Einen Konstruktor kann man nur mit dem Schlüsselwort **new** plus dem Namen der Klasse aufrufen. Die JVM sucht die Klasse und ruft den Konstruktor in der Klasse auf. (Okay, technisch gesehen ist das nicht die *einzige* Möglichkeit, einen Konstruktor aufzurufen. Aber es ist die einzige Möglichkeit, das *außerhalb* eines Konstruktors zu tun. Sie *können* einen Konstruktor aus einem anderen Konstruktor aufrufen. Da gibt es allerdings Einschränkungen. Aber dazu werden wir später in diesem Kapitel kommen.)

Ein Konstruktor enthält den Code, der ausgeführt wird, wenn Sie ein Objekt instantiieren. Anders gesagt, den Code, der ausgeführt wird, wenn Sie auf dieser Klasse new sagen.

Jede Klasse, die Sie erstellen, hat einen Konstruktor – selbst wenn Sie selbst keinen geschrieben haben.

Aber wo ist dieser Konstruktor?

Wenn wir ihn nicht geschrieben haben, wer hat ihn dann geschrieben?

Sie können einen Konstruktor für Ihre Klasse schreiben (und wir werden das gleich tun). Aber wenn Sie es nicht tun, ***schreibt der Compiler einen für Sie!***

So sieht der Default-Konstruktor des Compilers aus:

```
public  Ente() {

}
```

Fehlt hier irgendwas? Worin unterscheidet sich das von einer Methode?

Der Name ist mit dem der Klasse identisch. Das ist erforderlich.

```
public  Ente() {
   // hier kommt der Konstruktorcode hin
}
```

Wo ist der Rückgabetyp? Wenn das eine Methode wäre, bräuchte man zwischen »public« und »Ente()« einen Rückgabetyp.

Eine Ente konstruieren

Das Schlüssel-Feature eines Konstruktors ist, dass er ausgeführt wird, *bevor* das Objekt einer Referenz zugewiesen werden kann. Das bedeutet, dass Sie eine Möglichkeit erhalten, einzuschreiten und Dinge zu tun, die das Objekt für den Gebrauch vorbereiten. Das Objekt hat also eine Möglichkeit, sich selbst aufzubauen, bevor jemand die Fernsteuerung für ein Objekt verwenden kann. In unserem Ente-Konstruktor machen wir nichts Nützliches. Aber er veranschaulicht die Abfolge der Ereignisse.


```
public class Ente {

   public Ente() {
      System.out.println("Quak");
   }
}
```

Konstruktorcode.

Der Konstruktor gibt Ihnen eine Möglichkeit, mitten in `new` einzugreifen.

```
public class EnteVerwenden {

   public static void main (String[] args) {
      Ente e = new Ente();
   }
}
```

Ruft den Ente-Konstruktor auf.

```
Datei Bearbeiten Fenster Hilfe Quak
% java EnteVerwenden
Quak
```

Spitzen Sie Ihren Bleistift

Ein Konstruktor ermöglicht Ihnen, mitten in den Schritt der Objekterzeugung einzuspringen – mitten in **`new`**. Können Sie sich Umstände vorstellen, unter denen das nützlich sein könnte? Welche Vorbereitungen könnten im Konstruktor einer Klasse Wagen nützlich sein, wenn Wagen Teil eines RennSpiels ist? Haken Sie die ab, zu denen Ihnen ein Szenario eingefallen ist.

- ☐ Einen Zähler inkrementieren, um nachzuhalten, wie viele Objekte dieser Klasse erzeugt wurden.
- ☐ Laufzeitspezifische Daten zuweisen (Daten zu dem, was JETZT passiert).
- ☐ Den wichtigen Instanzvariablen des Objekts Werte zuweisen.
- ☐ Eine Referenz auf das Objekt, das das neue Objekt *erzeugt*, abrufen und speichern.
- ☐ Das Objekt einer ArrayList hinzufügen.
- ☐ HAS-A-Objekte erzeugen.
- ☐ _____

(Ihre eigenen Vorschläge)

Den Zustand einer neuen Ente initialisieren

Die meisten Menschen verwenden Konstruktoren, um den Zustand eines Objekts zu initialisieren, das heißt, um Werte für die Instanzvariablen des Objekts zu erzeugen und diese zuzuweisen.

```
public Ente() {
    größe = 34;
}
```

Das ist wunderbar, wenn der *Entwickler* der Klasse Ente weiß, wie groß das Ente-Objekt sein soll. Aber was ist, wenn der Programmierer, der die Ente *verwendet*, entscheiden soll, wie groß eine bestimmte Ente ist?

Stellen Sie sich vor, die Ente hat eine Instanzvariable größe und Sie möchten, dass der Programmierer, der die Klasse Ente verwendet, die Größe der neuen Ente festlegt. Wie könnten Sie das tun?

Gut, Sie könnten der Klasse eine setGröße()-Setter-Methode geben. Aber dann hätte die Ente vorübergehend keine Größe*. Außerdem müsste der Ente-Nutzer *zwei* Anweisungen schreiben – eine, die die Ente erzeugt, und eine, die die Methode getGröße() aufruft. Der folgende Code nutzt eine Setter-Methode, um die Anfangsgröße der neuen Ente zu setzen.

```
public class Ente {
   int größe;                                  ← Instanzvariable

   public Ente() {
      System.out.println("Quak");              ← Konstruktor
   }

   public void setGröße(int neueGröße) {       ← Setter-Methode
      größe = neueGröße;
   }
}
```

```
public class EnteVerwenden{

   public static void main (String[] args){
      Ente e = new Ente();
      e.setGröße(42);
   }
}
```

Aber hier ist etwas nicht in Ordnung. Die Ente lebt an diesem Punkt des Codes schon, hat aber noch keine Größe!* Und dann verlassen Sie sich darauf, dass der Ente-Nutzer WEISS, dass die Erzeugung einer Ente ein zweischrittiger Vorgang ist: ein Aufruf des Konstruktors und ein Aufruf des Setters.

*Instanzvariablen haben einen Default-Wert. 0 oder 0.0 bei elementaren numerischen Typen, false bei Booleschen Werten und null bei Referenzen.

Es gibt keine Dummen Fragen

F: Warum muss man einen Konstruktor schreiben, wenn der Compiler das für einen erledigt?

A: Wenn Sie Code brauchen, der Sie dabei unterstützt, Ihr Objekt zu initialisieren und zur Verwendung vorzubereiten, müssen Sie Ihren eigenen Konstruktor schreiben. Vielleicht brauchen Sie beispielsweise bestimmte Benutzereingaben, bevor Sie die Vorbereitung Ihres Objekts abschließen können. Es gibt einen anderen Grund, aus dem Sie einen Konstruktor schreiben müssen, auch wenn Sie selbst keinen Konstruktorcode benötigen. Der hat mit dem Konstruktor der Superklasse zu tun, und wir werden in ein paar Minuten darüber reden.

F: Wie kann man einen Konstruktor von einer Methode unterscheiden? Kann man auch Methoden mit dem gleichen Namen wie die Klasse haben?

A: Java erlaubt Ihnen, eine Methode mit dem gleichen Namen wie Ihre Klasse zu erstellen. Das gibt Ihnen aber noch keinen Konstruktor. Das, was eine Methode von einem Konstruktor unterscheidet, ist der Rückgabetyp. Methoden *müssen* einen Rückgabetyp haben, aber Konstruktoren *dürfen* keinen Rückgabetyp haben.

F: Werden Konstruktoren vererbt? Erhält man den Superklassenkonstruktor statt des Default-Konstruktors, wenn man keinen Konstruktor anbietet, aber die Superklasse einen bereitstellt?

A: Nein. Konstruktoren werden nicht vererbt. Wir werden uns das in ein paar Seiten anschauen.

Den Konstruktor verwenden, um die wichtigen Zustände von Ente zu initialisieren*

Sollte ein Objekt nicht verwendet werden, bis ein oder mehrere Teile seines Zustands (Instanzvariablen) initialisiert worden sind, sollten Sie niemandem ein Ente-Objekt in die Hand geben, bis Sie diese Initialisierung abgeschlossen haben! In der Regel ist es zu gefährlich, jemanden ein neues Ente-Objekt machen zu lassen – und sich eine Referenz darauf zu verschaffen –, das erst wirklich verwendungsfähig ist, wenn jemand vorbeikommt und die Methode *setGröße()* aufruft. Woher soll der Ente-Nutzer überhaupt *wissen*, dass er, nachdem er die neue Ente gemacht hat, die Setter-Methode aufrufen muss?

Der beste Ort für Initialisierungscode ist der Konstruktor. Dazu müssen Sie nur einen Konstruktor mit Argumenten erzeugen.

```
public class Ente {
   int größe;

   public Ente(int entenGröße) {
      System.out.println("Quak");

      größe = entenGröße;

      System.out.println("die Größe ist " + größe);
   }
}
```

Dem Ente-Konstruktor einen int-Parameter hinzufügen.

Den Argumentwert verwenden, um die Instanzvariable größe zu setzen.

```
public class EnteVerwenden {

   public static void main (String[] args) {
      Ente e = new Ente(42);
   }
}
```

Dieses Mal haben wir nur eine Anweisung. Wir machen die neue Ente und setzen ihre Größe in einer einzigen Anweisung.

Dem Konstruktor einen Wert übergeben.

```
Datei Bearbeiten Fenster Hilfe Tüt
% java EnteVerwenden
Quak
die Größe ist 42
```

*Was nicht heißen soll, dass andere Entenzustände unbedeutend wären.

Das Erzeugen einer Ente erleichtern

Achten Sie darauf, dass Sie einen Konstruktor haben, der kein Argument erwartet

Was passiert, wenn der Ente-Konstruktor ein Argument erwartet? Denken Sie darüber nach. Auf der vorangegangenen Seite gab es nur *einen* Ente-Konstruktor – und der erwartet ein int-Argument für die *größe* der Ente. Das muss nicht unbedingt ein großes Problem sein, erschwert es Programmierern aber, eine neues Ente-Objekt zu erzeugen, insbesondere wenn der Programmierer noch nicht *weiß*, welche Größe eine Ente haben soll. Wäre es nicht nützlich, wenn man eine Default-Größe für eine Ente hätte, damit der Benutzer auch dann eine funktionierende Ente erzeugen kann, wenn er noch keine geeignete Größe kennt?

Stellen Sie sich vor, Sie möchten Ihren Ente-Benutzern ZWEI Möglichkeiten geben, eine Ente zu erzeugen – eine, bei der sie die Größe der Ente (als Konstruktor-Argument) angeben, und eine, bei der sie keine Größe angeben und deswegen Ihre Default-Entengröße erhalten.

Mit einem einzigen Konstruktor können Sie das nicht sauber lösen. Denken Sie daran: Wenn eine Methode (oder ein Konstruktor – da gelten die gleichen Regeln) einen Parameter hat, *müssen* Sie ein geeignetes Argument übergeben, wenn Sie diese Methode oder diesen Konstruktor aufrufen. Sie können nicht einfach sagen: »Wenn jemand dem Konstruktor nichts übergibt, dann verwende die Default-Größe.« Das würde sich nicht einmal kompilieren lassen, wenn Sie beim Aufruf des Konstruktors kein int-Argument übergeben. Sie *könnten* irgendwas Umständliches wie das hier tun:

```
public class Ente {
   int größe;

   public Ente(int neueGröße) {
      if (neueGröße == 0) {
         größe = 27;
      } else {
         größe = neueGröße;
      }
   }
}
```

Wenn der Wert des Parameters 0 ist, der neuen Ente eine Default-Größe geben, andernfalls den Parameterwert für die Größe verwenden. KEINE sonderlich gute Lösung.

Aber das bedeutet, dass der Programmierer, der ein neues Ente-Objekt erzeugt, *wissen* muss, dass »0 übergeben« das Protokoll dafür ist, den Default-Wert zu erhalten. Ziemlich hässlich. Was ist, wenn der andere Programmierer das nicht weiß? Oder wenn er tatsächlich eine Ente mit der Größe 0 machen *möchte*? (Angenommen, eine Ente mit Größe 0 ist erlaubt. Wenn Sie keine Ente mit Größe 0 haben möchten, können Sie in den Konstruktor Validierungscode stecken, der das verhindert.) Der Punkt ist, dass es vielleicht nicht immer möglich ist, zwischen einem richtigen »ich möchte 0 als größe«-Konstruktorargument und einem »ich sende 0, damit du mir die Default-Größe lieferst, egal welche das ist«-Konstruktorargument zu unterscheiden.

Eigentlich brauchen Sie ZWEI Möglichkeiten, eine neue Ente zu erzeugen:

```
public class Ente2 {
   int größe;

   public Ente2() {
      // Default-Größe liefern
      größe = 27;
   }

   public Ente2(int entenGröße)
   {
      // entenGröße-Parameter
      // verwenden
      größe = entenGröße;
   }
}
```

Machen Sie so eine Ente, wenn Sie die Größe kennen:

```
Ente2 e = new Ente2(15);
```

Und so machen Sie eine Ente, wenn Sie die Größe *nicht* kennen:

```
Ente2 e2 = new Ente2();
```

Für die Zwei-Möglichkeiten-eine-Ente-zu-erzeugen-Idee brauchen Sie *zwei* Konstruktoren. Einen, der ein int annimmt, und einen, der das nicht tut. ***Wenn Sie in einer Klasse mehrere Konstruktoren haben, heißt das, dass Sie überladene Konstruktoren haben.***

Macht der Compiler nicht immer einen Default-Konstruktor für mich?

Nein!

Vielleicht erwarten Sie, dass der Compiler sieht, dass Sie keinen argumentlosen Konstruktor haben, und einen für Sie einfügt, wenn Sie *nur* einen Konstruktor mit Argumenten schreiben. Aber so funktioniert das nicht. Der Compiler kümmert sich um die Konstruktorerstellung nur dann, *wenn Sie überhaupt nichts zu Konstruktoren sagen.*

Wenn Sie einen Konstruktor schreiben, der Argumente erwartet, und *trotzdem* noch einen Konstruktor möchten, der keine Argumente erwartet, dann müssen Sie diesen Konstruktor selbst schreiben!

Sobald Sie einen Konstruktor anbieten, IRGENDEINEN Konstruktor, zieht sich der Compiler zurück und sagt: »Okay Kumpel, scheint, du bist jetzt für Konstruktoren verantwortlich.«

Wenn Sie in einer Klasse mehrere Konstruktoren haben, MÜSSEN diese Konstruktoren unterschiedliche Argumentlisten haben.

Die Argumentliste umfasst die Anzahl und die Typen der Argumente. Unterscheiden sich diese, können Sie mehrere Konstruktoren haben. Das können Sie auch mit Methoden machen, wie wir schon in Kapitel 7 gesehen haben.

Überladene Konstruktoren bedeuten, dass Sie in Ihrer Klasse mehrere Konstruktoren haben.

Damit sich das kompilieren lässt, müssen die verschiedenen Konstruktoren jeweils *unterschiedliche* Argumentlisten haben!

Die Klasse unten ist legal, weil alle vier Konstruktoren unterschiedliche Argumentlisten haben. Bei zwei Konstruktoren, die beide ein int annehmen, ließe sich die Klasse beispielsweise nicht kompilieren. Welche Namen Sie den Parametern geben, zählt nicht. Es sind *Variablentyp* (int, Hund usw.) und *Reihenfolge*, die entscheiden. Sie *können* zwei Konstruktoren haben, die die gleichen Typen erwarten, ***wenn die Reihenfolge anders ist***. Ein Konstruktor, der einen String und dann ein int erwartet, ist *nicht* das Gleiche wie ein Konstruktor, der ein int und dann einen String erwartet.

Vier verschiedene Konstruktoren bieten vier verschiedene Wege, neue Pilze zu machen.

```
public class Pilz {

  public Pilz(int größe) { }

  public Pilz( ) { }

  public Pilz(boolean isGiftig) { }

  public Pilz(boolean isGiftig, int größe) { }

  public Pilz(int größe, boolean isGiftig) { }
}
```

Wenn Sie die Größe kennen, aber nicht wissen, ob der Pilz giftig ist.

Wenn Sie gar nichts wissen.

Wenn Sie wissen, ob er giftig ist oder nicht, aber die Größe nicht kennen.

Diese beiden haben die gleichen Argumente, aber in unterschiedlicher Reihenfolge, also ist das in Ordnung.

Wenn Sie wissen, ob er giftig ist oder nicht, UND auch die Größe kennen.

Punkt für Punkt

- Instanzvariablen leben auf dem Heap in dem Objekt, zu dem sie gehören.
- Wenn die Instanzvariable eine Referenz auf ein Objekt ist, befinden sich die Referenz UND das Objekt, auf das sie verweist, auf dem Heap.
- Ein Konstruktor ist der Code, der ausgeführt wird, wenn Sie auf einer Klasse **`new`** sagen.
- Ein Konstruktor muss den gleichen Namen wie die Klasse und *darf* keinen Rückgabetyp haben.
- Sie können einen Konstruktor verwenden, um den Zustand (d.h. die Instanzvariablen) des Objekts zu initialisieren, das konstruiert wird.
- Wenn Sie keinen Konstruktor in Ihre Klasse stecken, fügt der Compiler einen Default-Konstruktor ein.
- Der Default-Konstruktor ist immer ein argumentloser Konstruktor.
- Wenn Sie in Ihrer Klasse einen Konstruktor – einen beliebigen Konstruktor – angeben, fügt der Compiler keinen Default-Konstruktor ein.
- Wenn Sie einen argumentlosen Konstruktor brauchen und bereits einen Konstruktor mit Argumenten geschrieben haben, müssen Sie auch den argumentlosen Konstruktor selbst schreiben.
- Geben Sie immer einen argumentlosen Konstruktor an, wenn das möglich ist. Das erleichtert es Programmierern, ein Arbeitsobjekt zu erstellen. Geben Sie Default-Werte vor.
- Überladene Konstruktoren bedeuten, dass Sie in Ihrer Klasse mehrere Konstruktoren haben.
- Überladene Konstruktoren müssen unterschiedliche Argumentlisten haben.
- Sie dürfen keine Konstruktoren mit den gleichen Argumentlisten haben. Zur Argumentliste zählen die Reihenfolge und/oder der Typ der Argumente.
- Instanzvariablen wird ein Default-Wert zugewiesen, auch wenn Sie selbst keinen expliziten Wert angeben. Die Default-Werte sind 0/0.0/false für elementare Typen und null für Referenzen.

Spitzen Sie Ihren Bleistift

Ordnen Sie den `new Ente()`-Aufrufen die Konstruktoren zu, die ausgeführt werden, wenn die Ente instantiiert wird. Den leichten haben wir erledigt, um Ihnen auf die Sprünge zu helfen.

```
public class EnteTesten {

  public static void main(String[] args){

    int gewicht = 8;
    float dichte = 2.3F;
    String name = "Donald";
    long[] federn = {1,2,3,4,5,6};
    boolean kannFliegen = true;
    int luftGeschwindigkeit = 22;

    Ente[] e = new Ente[7];

    e[0] = new Ente();

    e[1] = new Ente(dichte, gewicht);

    e[2] = new Ente(name, federn);

    e[3] = new Ente(kannFliegen);

    e[4] = new Ente(3.3F, luftGeschwindigkeit);

    e[5] = new Ente(false);

    e[6] = new Ente(luftGeschwindigkeit, dichte);
  }
}
```

```
class Ente {

  int kilo = 6;
  float treibFähigkeit = 2.1F;
  String name = "Generisch";
  long[] federn = {1,2,3,4,5,6,7};
  boolean kannFliegen = true;
  int maxGeschwindigkeit = 25;

  public Ente() {
    System.out.println("Ententyp 1");
  }

  public Ente(boolean fliegen) {
    kannFliegen = fliegen;
    System.out.println("Ententyp 2");
  }

  public Ente(String n, long[] f) {
    name = n;
    federn = f;
    System.out.println("Ententyp 3");
  }

  public Ente(int g, float t) {
    kilo = g;
    treibFähigkeit = t;
    System.out.println("Ententyp 4");
  }

  public Ente(float dichte, int max) {
    treibFähigkeit = dichte;
    maxGeschwindigkeit = max;
    System.out.println("Ententyp 5");
  }
}
```

F: Weiter oben haben Sie gesagt, dass es gut sei, wenn man argumentlose Konstruktoren bereitstellt, damit wir Default-Werte für die »fehlenden Argumente« angeben können, wenn jemand den argumentlosen Konstruktor aufruft. Aber ist es manchmal nicht unmöglich, geeignete Default-Werte vorzugeben? Kann es nicht auch sein, dass es manchmal besser ist, wenn man in einer Klasse keinen argumentlosen Konstruktor hat?

A: Das ist richtig. Manchmal ist ein argumentloser Konstruktor sinnlos. Das sehen Sie auch in der Java-API – manche Klassen haben keinen argumentlosen Konstruktor. Die Klasse Color repräsentiert beispielsweise eine Farbe. Color-Objekte werden unter anderem dazu verwendet, die Farbe einer Bildschirmschrift oder eines GUI-Buttons zu ändern. Wenn Sie eine Color-Instanz erzeugen, steht diese Instanz für eine bestimmte Farbe (das Übliche, Tod-durch-Schokoladenbraun, Bluescreen-des-Todes-Blau, Skandalöses Rot usw.). Wenn Sie ein Color-Objekt machen, müssen Sie auf irgendeine Weise eine Farbe angeben.

```
Color f = new Color(3,45,200);
```

(Wir verwenden hier drei ints für die RGB-Werte. Wie man Color verwendet, betrachten wir später in den Swing-Kapiteln.) Denn was würden Sie erhalten, wenn Sie keine Farbe angeben? Die Programmierer der Java-API *hätten* entscheiden können, dass Sie einen wundervollen Lavendelton erhalten, wenn Sie einen argumentlosen Konstruktor aufrufen. Aber der gute Geschmack hat gesiegt. Wenn Sie versuchen, mit

```
Color f = new Color();
```

ein Color-Objekt zu erzeugen, ohne eine Farbe anzugeben, tillt der Compiler aus, weil er in der Klasse Color keinen passenden argumentlosen Konstruktor finden kann.

```
Datei Bearbeiten Fenster Hilfe Kopfeinschalten
cannot resolve symbol
:constructor Color()
location: class java.awt.Color
Color c = new Color();
          ^
1 error
```

Kurzrückblick: Vier Dinge, die man sich zu Konstruktoren merken sollte

1. Ein Konstruktor ist der Code, der ausgeführt wird, wenn jemand auf einem Klassentyp **new** sagt.

```
Ente d = new Ente();
```

2. Ein Konstruktor muss den gleichen Namen wie die Klasse und darf ***keinen*** Rückgabetyp haben.

```
public Ente(int größe) { }
```

3. Wenn Sie Ihrer Klasse keinen Konstruktor geben, steckt der Compiler einen Default-Konstruktor in die Klasse. Der Default-Konstruktor ist immer ein argumentloser Konstruktor.

```
public Ente() { }
```

4. Sie können in einer Klasse mehrere Konstruktoren haben, wenn sich deren Argumentlisten unterscheiden. Hat man in einer Klasse mehrere Konstruktoren, heißt das, dass man überladene Konstruktoren hat.

```
public Ente() { }

public Ente(int größe) { }

public Ente(String name) { }

public Ente(String name, int größe) {
}
```

Es hat sich gezeigt, dass das ganze Gehirnjogging eine 42%ige Zunahme der Neuronenmasse bewirkt. Und Sie kennen ja den Spruch »Dicke Neuronen …«

KOPF-NUSS

Was ist mit Superklassen?

Wenn man einen Hund erzeugt, sollte dann auch der Konstruktor von Hundeartig aufgerufen werden?

Und wenn die Superklasse abstrakt ist, sollte sie dann überhaupt einen Konstruktor haben?

Das werden wir uns auf den nächsten paar Seiten ansehen. Machen Sie jetzt also Halt und denken Sie über die Implikationen von Konstruktoren und Superklassen nach.

Es gibt keine Dummen Fragen

F: Müssen Konstruktoren öffentlich sein?

A: Nein. Konstruktoren können **public**, **private** oder *Default* (d.h. ohne Zugriffsmodifizierer) sein. Diesen *Default*-Zugriff sehen wir uns in Anhang B ausführlicher an.

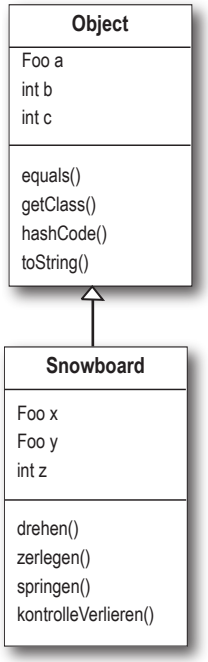
F: Welchen Zweck kann ein privater Konstruktor überhaupt haben? Niemand könnte ihn aufrufen. Also könnte niemand neue Objekte erzeugen!

A: Das stimmt nicht so ganz. Dass etwas als **private** markiert ist, heißt nicht, dass *niemand* darauf zugreifen kann. Das heißt nur, dass *niemand von außerhalb der Klasse* darauf zugreifen kann. Ich wette, Sie denken jetzt: »Da beißt sich der Hund doch in den Schwanz!« Nur Code aus der *gleichen* Klasse wie die Klasse-mit-dem-privaten-Konstruktor kann ein neues Objekt dieser Klasse machen. Wie aber kann man überhaupt Code dieser Klasse ausführen, wenn man nicht erst einmal ein Objekt macht? Wie kommt man jemals an irgendetwas in der Klasse? *Geduld, Grashüpfer* … dazu kommen wir im nächsten Kapitel.

Einen Augenblick ... wir haben im Zusammenhang mit Konstruktoren ÜBERHAUPT noch nicht über Superklassen und Vererbung gesprochen.

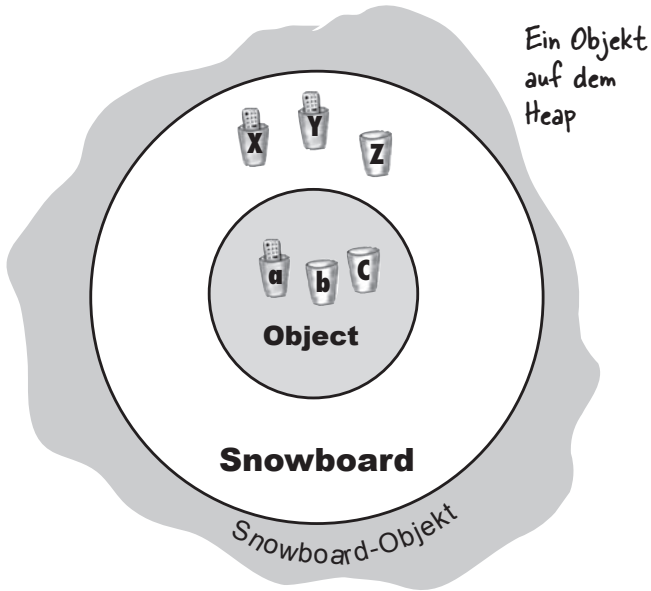
Hier beginnt das Vergnügen. Erinnern Sie sich an das letzte Kapitel? An den Teil, in dem wir uns angesehen haben, wie das Snowboard-Objekt einen inneren Kern umhüllt, der den Object-Teil der Klasse Snowboard repräsentiert? Die wichtigste Sache dort war, dass jedes Objekt nicht nur seine *eigenen* deklarierten Instanzvariablen enthält, sondern auch *alles von seinen Superklassen* (und das ist zumindest die Klasse Object, weil *jede* Klasse Object erweitert).

Wenn ein Objekt erzeugt wird (weil jemand **new** gesagt hat, es gibt ***keinen anderen Weg,*** ein Objekt zu erzeugen, als dass jemand irgendwo **new** auf dem Typ der Klasse sagt), erhält das Objekt Platz für *all* diese Instanzvariablen aus dem gesamten Vererbungsbaum. Denken Sie einen Augenblick darüber nach ... eine Superklasse könnte Setter-Methoden haben, die eine private Variable kapseln. Aber diese Variable muss *irgendwo* leben. Wenn ein Objekt erzeugt wird, ist das fast, als materialisierten sich *mehrere* Objekte – das Objekt der Klasse, auf der new gesagt wird, und ein Objekt für jede Superklasse. Theoretisch sollte man sich das allerdings besser vorstellen wie das Bild unten: Das ist so, als hätte das erzeugte Objekt *Schichten*, die die einzelnen Superklassen repräsentieren.



Object hat Instanzvariablen, die mit Zugriffsmethoden gekapselt werden. Diese Instanzvariablen werden erzeugt, wenn eine Unterklasse instantiiert wird. (Es sind keine ECHTEN Object-Variablen, aber das ist uns egal, da sie ordentlich gekapselt sind.)

Snowboard hat außerdem eigene Instanzvariablen. Wenn wir ein Snowboard-Objekt machen, brauchen wir also Platz für die Instanzvariablen beider Klassen.



Auf dem Heap gibt es nur EIN Objekt. Ein Snowboard-Objekt. Aber es enthält gleichzeitig die Snowboard-Teile seiner selbst und die Object-Teile seiner selbst. Hier müssen sich alle Instanzvariablen beider Klassen befinden.

Die Rolle des Superklassen-Konstruktors im Leben eines Objekts.

Alle Konstruktoren im Vererbungsbaum eines Objekts müssen ausgeführt werden, wenn Sie ein neues Objekt erzeugen.

Lassen Sie das sacken.

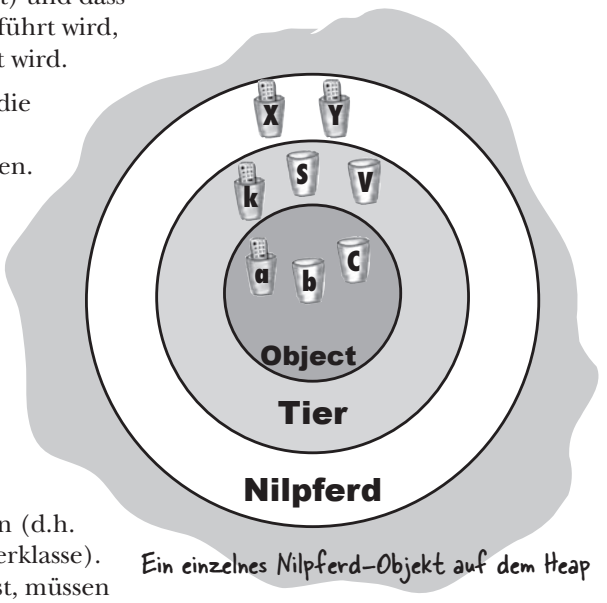
Es bedeutet, dass jede Superklasse einen Konstruktor hat (weil jede Klasse einen Konstruktor hat) und dass jeder Konstruktor in der Hierarchie ausgeführt wird, wenn ein Objekt einer Unterklasse erzeugt wird.

new zu sagen ist eine große Sache. Es löst die ganze Konstruktor-Kettenreaktion aus. Ja, auch abstrakte Klassen haben Konstruktoren. Obwohl man auf einer abstrakten Klasse nie new sagen kann, bleibt eine abstrakte Klasse eine Superklasse. Und deswegen wird ihr Konstruktor ausgeführt, wenn jemand eine Instanz einer konkreten Unterklasse erzeugt.

Die Super-Konstruktoren werden ausgeführt, um die Superklassenteile des Objekts aufzubauen. Denken Sie daran, dass eine Unterklasse Methoden erben kann, die vom Zustand der Superklasse abhängen (d.h. vom Wert von Instanzvariablen in der Superklasse). Damit ein Objekt vollständig ausgeformt ist, müssen auch seine Superklassenteile vollständig ausgeformt sein. Und deswegen *muss* der Superklassenkonstruktor ausgeführt werden. Alle Instanzvariablen aller Klassen im Vererbungsbaum müssen deklariert und initialisiert werden. Auch wenn die Klasse Tier Instanzvariablen hat, die Nilpferd nicht erbt (weil die Variablen beispielsweise private sind), ist das Nilpferd trotzdem noch von den Tier-Methoden abhängig, die diese Variablen *nutzen*.

Wird ein Konstruktor ausgeführt, ruft er sofort seine Superklassenkonstruktoren auf und durchläuft dabei die ganze Hierarchie bis zum Konstruktor von Object.

Auf den nächsten paar Seiten werden Sie erfahren, wie Superklassenkonstruktoren aufgerufen werden und wie Sie sie selbst aufrufen können. Sie erfahren auch, was Sie tun müssen, wenn Ihr Superklassenkonstruktor Argumente hat!



Ein einzelnes Nilpferd-Objekt auf dem Heap

Ein neues Nilpferd-Objekt ist ein Tier und ist ein Object. Wenn Sie ein Nilpferd machen möchten, müssen Sie auch die Tier- und die Object-Teile von Nilpferd machen.

All das passiert in einem Prozess, der als Konstruktorverkettung bezeichnet wird.

Ein neues Nilpferd zu machen heißt auch, die Tier- und Object-Teile zu machen ...

```
public class Tier {
   public Tier() {
      System.out.println("Mache ein Tier");
   }
}
```

```
public class Nilpferd extends Tier {
   public Nilpferd() {
      System.out.println("Mache ein Nilpferd");
   }
}
```

```
public class NilpferdTesten {
   public static void main (String[] args) {
      System.out.println("Starte ...");
      Nilpferd n = new Nilpferd();
   }
}
```

Spitzen Sie Ihren Bleistift

Wie sieht die Ausgabe aus? Den Code auf der linken Seite vorausgesetzt, was wird ausgegeben, wenn Sie NilpferdTesten ausführen? A oder B?

(Die Antwort finden Sie unten.)

A

```
Datei Bearbeiten Fenster Hilfe Fluchen
% java NilpferdTesten
Starte ...
Mache ein Tier
Mache ein Nilpferd
```

B

```
Datei Bearbeiten Fenster Hilfe Fluchen
% java NilpferdTesten
Starte ...
Mache ein Nilpferd
Mache ein Tier
```

1. Code von einer anderen Klasse sagt `new Nilpferd()`, und der **Nilpferd()**-Konstruktor kommt in einen Stack-Frame oben auf den Stack.

2. **Nilpferd()** ruft den Superklassenkonstruktor auf, und der **Tier()**-Konstruktor wird oben auf den Stack geschoben.

3. **Tier()** ruft den Superklassenkonstruktor auf, und der **Object()**-Konstruktor wird oben auf den Stack geschoben, da Object die Superklasse von Tier ist.

4. **Object()** wird abgeschlossen, und der entsprechende Stack-Frame wird vom Stack *gezogen*. Die Ausführung kehrt in den **Tier()**-Konstruktor zurück und fährt mit der Zeile fort, die in Tier() auf den Aufruf des Superklassenkonstruktors folgt.

Die erste Ausgabe, A. Der Nilpferd()-Konstruktor wird zuerst aufgerufen, aber der Tier-Konstruktor wird zuerst vollständig abgearbeitet.

Wie ruft man einen Superklassenkonstruktor auf?

Sie glauben vielleicht, dass man beispielsweise irgendwo in einem Ente-Konstruktor Tier() aufruft, wenn Ente Tier erweitert. Aber so funktioniert das nicht:

```
public class Ente extends Tier {
   int größe;

   public Ente(int neueGröße) {
      Tier();
      größe = neueGröße;
   }
}
```

SCHLECHT!

NEIN! Das ist nicht zulässig!

Einen Superkonstruktor kann man nur mit ***super()*** aufrufen. Das ist richtig – ***super()*** ruft den ***Superkonstruktor*** auf.

Was für ein seltsamer Zufall!

```
public class Ente extends Tier {
   int größe;

   public Ente(int neueGröße) {
      super();
      größe = neueGröße;
   }
}
```

Sagen Sie einfach super()!

Wenn Sie in Ihrem Konstruktor *super()* aufrufen, kommt der Superklassenkonstruktor oben auf den Stack. Und was meinen Sie, macht der Superklassenkonstruktor? *Er ruft seinen Superklassenkonstruktor auf.* Und so geht das weiter, bis oben auf dem Stack der Object-Konstruktor liegt. Wenn *Object()* fertig ist, wird er vom Stack genommen, und es befindet sich das oben, was auf dem Stack als Nächstes kommt (der Unterklassenkonstruktor, der *Object()* aufgerufen hat). *Dieser* Konstruktor wird beendet und so geht das weiter, bis der ursprüngliche Konstruktor wieder oben auf dem Stack liegt, wo *er* jetzt abgearbeitet werden kann.

Und wie kommt es, dass wir davongekommen sind, ohne das zu tun?

Wahrscheinlich haben Sie sich das schon gedacht.

Unser guter Freund, der Compiler, schiebt den Aufruf von *super()* ein, wenn Sie das nicht tun.

Der Compiler ist am Konstruktormachen also auf *zweierlei* Weise beteiligt:

1 Wenn Sie *keinen* Konstruktor angeben:

Der Compiler fügt einen ein, der so aussieht:

```
public KlassenName() {
      super();
}
```

2 Wenn Sie einen Konstruktor *angeben,* aber den Aufruf von super() *nicht* einfügen:

Der Compiler fügt den Aufruf von super() in jeden Ihrer überladenen Konstruktoren ein.*

Der vom Compiler eingefügte Aufruf sieht so aus:

```
super();
```

Er sieht immer so aus. Der vom Compiler eingefügte Aufruf von *super()* ist immer ein argumentloser Aufruf. Wenn die Superklasse überladene Konstruktoren hat, wird nur der argumentlose Konstruktor aufgerufen.

*Es sei denn, der Konstruktor ruft einen anderen überladenen Konstruktor auf (das werden Sie in ein paar Seiten sehen).

Kann das Kind vor den Eltern leben?

Wenn Sie sich eine Superklasse als Eltern des Unterklassenkindes vorstellen, können Sie herausfinden, wer zuerst bestehen muss. ***Die Superklassenteile eines Objekts müssen vollständig ausgeformt (vollständig aufgebaut) sein, bevor die Unterklassenteile konstruiert werden können.*** Denken Sie daran, dass das Unterklassenobjekt von Dingen abhängig sein könnte, die es von seiner Superklasse erbt. Deswegen ist es wichtig, dass diese geerbten Dinge zuerst fertig gestellt werden. Daran führt kein Weg vorbei. Der Superklassenkonstruktor muss vor dem Unterklassenkonstruktor fertig sein.

Schauen Sie sich die Stack-Abfolge auf Seite 252 noch einmal an. Dann sehen Sie, dass der Nilpferd-Konstruktor zwar *zuerst* aufgerufen wird (er ist das erste Ding auf dem Stack), aber *zuletzt* fertig ist! Jeder Unterklassenkonstruktor ruft sofort den Konstruktor seiner eigenen Superklasse auf, bis sich der Konstruktor von Object oben auf dem Stack befindet. Ist der Konstruktor von Object fertig, gehen wir den Stack wieder nach unten zum Konstruktor von Tier. Erst nachdem der Konstruktor von Tier abgeschlossen ist, kommen wir endlich zurück, um den Rest des Nilpferd-Konstruktors zu erledigen. Deswegen:

Der Aufruf von super() muss die *erste* Anweisung in jedem Konstruktor sein!*

Mögliche Konstruktoren für die Klasse Blub

```
☑ public Blub() {
      super();
  }
```

Die beiden sind okay, weil der Programmierer den Aufruf von super() explizit als erste Anweisung angegeben hat.

```
☑ public Blub(int i) {
      super();
      größe = i;
  }
```

```
☑ public Blub() {
  }
```

Diese beiden sind okay, weil der Compiler den Aufruf von super() als erste Anweisung einfügt.

```
☑ public Blub(int i) {
      größe = i;
  }
```

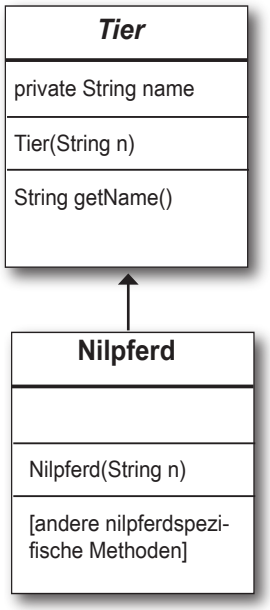
```
⊘ public Blub(int i) {
      größe = i;
      super();
  }
```

NEIN!! Das lässt sich nicht kompilieren! Sie können den Aufruf von super() nicht explizit nach etwas anderem angeben.

*Es gibt eine Ausnahme zu dieser Regel. Sie werden sie auf Seite 256 kennen lernen.

Superklassenkonstruktoren mit Argumenten

Was ist, wenn der Superklassenkonstruktor Argumente hat? Kann man mit dem Aufruf von *super()* etwas übergeben? Natürlich. Wenn man das nicht könnte, könnte man Klassen, die keinen argumentlosen Konstruktor haben, überhaupt nicht erweitern. Stellen Sie sich folgendes Szenario vor: Alle Tiere haben einen Namen. In der Klasse Tier gibt es eine *getName()*-Methode, die den Wert der Instanzvariablen *name* zurückliefert. Die Instanzvariable ist als private markiert, aber die Unterklasse (in diesem Fall Nilpferd) erbt die Methode *getName()*. Das Problem ist also folgendes: Nilpferd hat (durch Vererbung) eine *getName()*-Methode, aber keine Instanzvariable *name*. Nilpferd ist davon abhängig, dass sein Tier-Teil die Instanzvariable name festhält und ihren Wert zurückliefert, wenn jemand *getName()* auf einem Nilpferd-Objekt aufruft. Aber ... wie erhält der Tier-Teil den Namen? Die einzige Referenz, die Nilpferd auf seinen Tier-Teil hat, ist über *super()*. Das muss also der Ort sein, über den Nilpferd den Namen des Nilpferds an seinen Tier-Teil schickt, damit der Tier-Teil ihn in der privaten Instanzvariablen *name* speichern kann.




```
public abstract class Tier {
   private String name;

   public String getName() {
      return name;
   }

   public Tier(String derName) {
      name = derName;
   }
}
```

Alle Tiere (einschließlich der Unterklassen) haben einen Namen.

Eine Getter-Methode, die Nilpferd erbt.

Der Konstruktor, der den Namen entgegennimmt und der Instanzvariablen name zuweist.

```
public class Nilpferd extends Tier {

   public Nilpferd(String name) {
      super(name);
   }
}
```

Der Nilpferd-Konstruktor nimmt einen Namen.

und sendet ihn den Stack hinauf zum Tier-Konstruktor.


```
public class NilpferdMachen{
   public static void main(String[] args) {
      Nilpferd h = new Nilpferd("Kurt");
      System.out.println(h.getName());
   }
}
```

Ein Nilpferd machen und dabei den Namen »Kurt« an den Nilpferd-Konstruktor übergeben.

Dann die geerbte getName()-Methode von Nilpferd aufrufen.

```
Datei Bearbeiten Fenster Hilfe Verstecken
%java NilpferdMachen
Kurt
```

Einen überladenen Konstruktor aus einem anderen Konstruktor aufrufen

Was ist, wenn man überladene Konstruktoren hat, die zwar mit unterschiedlichen Argumenttypen umgehen können, sonst aber genau das Gleiche machen? Sie wissen, dass Sie *doppelten Code* vermeiden wollen, der in jedem der Konstruktoren sitzt (grausam bei der Wartung usw.). Also würden Sie gern den Großteil des Konstruktorcodes (einschließlich des super()-Aufrufs) in einen *einzigen* der überladenen Konstruktoren stecken. Und jeder Konstruktor, egal welcher zuerst aufgerufen wird, soll dann **Den Richtigen Konstruktor** aufrufen und ihn die Aufbauarbeit erledigen lassen. Das ist leicht: Sagen Sie einfach *this()*. Oder *this(einString)*. Oder *this(27, x)*. Mit anderen Worten: Stellen Sie sich einfach vor, dass das Schlüsselwort *this* eine Referenz auf **das aktuelle Objekt** ist.

this() können Sie nur in einem Konstruktor sagen, und es muss die erste Anweisung im Konstruktor sein!

Aber das ist ein Problem, oder? Eben haben wir gesagt, dass super() die erste Anweisung im Konstruktor sein muss. Das bedeutet, dass Sie die Wahl haben.

Jeder Konstruktor kann einen Aufruf von super() oder von this() enthalten, aber niemals beides gleichzeitig!

Nutzen Sie this( ), um einen Konstruktor aus einem anderen überladenen Konstruktor in der gleichen Klasse aufzurufen.

Der Aufruf von this( ) kann nur in einem Konstruktor verwendet werden und muss die erste Anweisung in einem Konstruktor sein.

Ein Konstruktor kann entweder einen Aufruf von super( ) ODER einen Aufruf von this( ) haben, aber nie beides!

```
class Mini extends Auto {

    Color farbe;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color f) {
        super("Mini");
        farbe = f;
        // mehr Initialisierungscode
    }

    public Mini(int größe) {
        this(Color.Red);
        super(größe);
    }
}
```

Der argumentlose Konstruktor gibt eine Default-Farbe an und ruft den überladenen richtigen Konstruktor auf (den, der super() aufruft).

Das ist Der Richtige Konstruktor, der Die Richtige Arbeit der Initialisierung des Objekts leistet (einschließlich des Aufrufs von super()).

Funktioniert nicht!! super() und this() können nicht im gleichen Konstruktor stehen, weil beide jeweils die erste Anweisung im Konstruktor sein müssen!

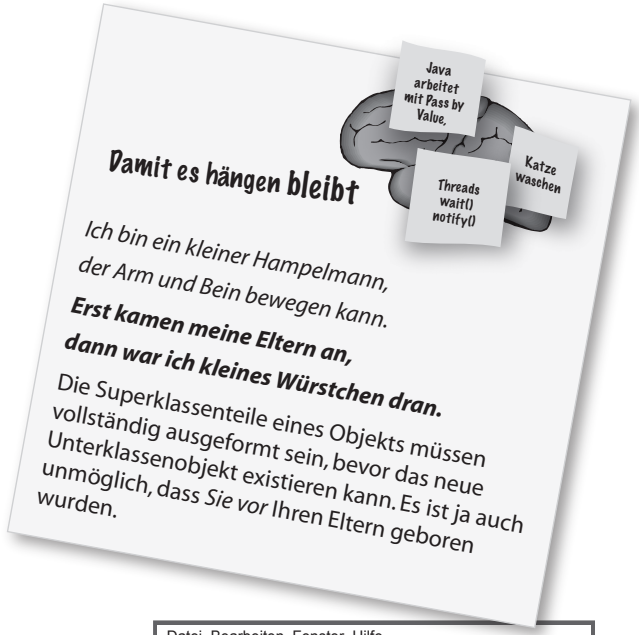
```
Datei Bearbeiten Fenster Hilfe Fahren
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor
      super();
      ^
```

Spitzen Sie Ihren Bleistift

Einige Konstruktoren in der Klasse SohnVonBuh lassen sich nicht kompilieren. Können Sie herausfinden, welche der Konstruktoren nicht zulässig sind? Ordnen Sie die Compiler-Fehler den SohnVonBuh-Konstruktoren zu, die sie verursacht haben, indem Sie eine Linie vom Compiler-Fehler zu dem entsprechenden »schlechten« Konstruktor ziehen.

```
public class Buh {
   public Buh(int i) { }
   public Buh(String s) { }
   public Buh(String s, int i) { }
}
```

```
class SohnVonBuh extends Buh {

   public SohnVonBuh() {
      super("Buh");
   }

   public SohnVonBuh(int i) {
      super("Fred");
   }

   public SohnVonBuh(String s) {
      super(42);
   }

   public SohnVonBuh(int i, String s) {
   }

   public SohnVonBuh(String a, String b, String c) {
      super(a,b);
   }

   public SohnVonBuh(int i, int j) {
      super("Mann", j);
   }

   public SohnVonBuh(int i, int x, int y) {
      super(i, "Stern");
   }
}
```



Datei Bearbeiten Fenster Hilfe

```
%javac SohnVonBuh.java

cannot resolve symbol

symbol : constructor Buh
(java.lang.String,java.la
ng.String)
```

Datei Bearbeiten Fenster Hilfe BlaBlaBla

```
%javac SohnVonBuh.java

cannot resolve symbol

symbol : constructor Buh
(int,java.lang.String)
```

Datei Bearbeiten Fenster Hilfe IchHöreNichtZu

```
%javac SohnVonBuh.java

cannot resolve symbol

symbol:constructor Buh()
```

Wir wissen jetzt, wie ein Objekt geboren wird. Aber wie lange *lebt* ein Objekt?

Das Leben eines *Objekts* ist vollkommen vom Leben der Referenzen abhängig, die es referenzieren. Wenn die Referenz als »lebend« betrachtet wird, ist das Objekt auf dem Heap noch am Leben. Wenn die Referenz stirbt (und was das heißt, werden wir uns gleich ansehen), stirbt auch das Objekt.

Wenn das Leben eines Objekts also vom Leben der Referenzvariablen abhängt, wie lange lebt dann eine *Variable*?

Das hängt davon ab, ob die Variable eine *lokale* Variable oder eine *Instanzvariable* ist. Der folgende Code zeigt das Leben einer lokalen Variablen. In diesem Beispiel handelt es sich um eine Variable für einen elementaren Typ, aber die Lebensdauer einer Variablen ist bei elementaren Variablen und Referenzvariablen gleich.

```
public class LebenstestEins {

    public void lesen() {
        int s = 42;
        schlafen();
    }

    public void schlafen() {
        s = 7;
    }
}
```

»s« ist nur in der Methode lesen() gültig und kann deswegen nirgendwo sonst verwendet werden.

SCHLECHT!! Hier ist die Verwendung von »s« nicht zulässig!

schlafen() kann die Variable »s« nicht sehen. Da sie sich nicht im Stack-Frame von schlafen() befindet, weiß schlafen() nichts über diese Variable.

Die Variable »s« lebt, aber sie ist nur in der Methode lesen() gültig. Wenn schlafen() fertig ist und lesen() sich oben auf dem Stack befindet und wieder ausgeführt wird, kann lesen() »s« immer noch sehen. Wenn lesen() fertig ist und vom Stack genommen wird, ist »s« tot und sieht sich die digitalen Radieschen von unten an.

1 Eine lokale Variable lebt nur in der Methode, die die Variable deklariert hat.

```
public void lesen() {
    int s = 42;
    // 's' kann nur innerhalb
    // dieser Methode verwendet
    // werden. Wenn diese Methode
    // endet, verschwindet 's'
    // vollständig.
}
```

Die Variable »**s**« kann *nur* in der Methode *lesen()* verwendet werden. Anders gesagt: ***Der Geltungsbereich dieser Variablen ist auf diese Methode beschränkt***. Kein anderer Code in der Klasse (oder in einer anderen Klasse) kann »**s**« sehen.

2 Eine Instanzvariable lebt so lange wie das Objekt. Solange das Objekt lebt, leben seine Instanzvariablen.

```
public class Leben {
    int größe;

    public void setGröße(int s) {
        größe = s;
        // 's' verschwindet am
        // Ende dieser Methode,
        // aber 'größe' kann
        // überall in dieser
        // Klasse verwendet
        // werden.
    }
}
```

Die Variable »**s**« (diesmal ein Methodenparameter) ist nur innerhalb der Methode setGröße() in Geltung. Aber die Instanzvariable »größe« ist, unabhängig vom Leben der *Methode,* so lange in Geltung, wie das *Objekt* lebt.

Der Unterschied zwischen **Leben** und **Geltung** bei lokalen Variablen:

Leben

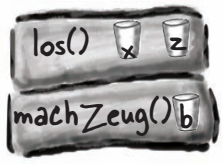
Eine lokale Variable *lebt*, solange ihr Stack-Frame auf dem Stack ist. Mit anderen Worten: *bis die Methode abgeschlossen ist.*

Geltung

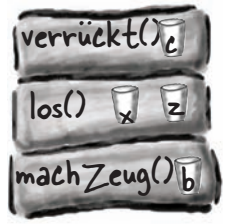
Eine lokale Variable ist nur innerhalb der Methode in *Geltung*, in der die Variable deklariert wurde. Wenn diese Methode eine andere Methode aufruft, ist die Variable zwar immer noch am Leben, aber nicht mehr in Geltung, bis die Methode wieder aufgenommen wird. ***Sie können eine Variable nur innerhalb ihres Geltungsbereichs verwenden.***

```
public void machZeug() {
   boolean b = true;
   los(4);
}
public void los(int x) {
   int z = x + 24;
   verrückt();
   // stellen Sie sich hier mehr Code vor
}
public void verrückt() {
   char c = 'a';
}
```

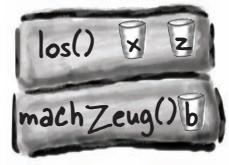
1 ***machZeug()*** **kommt auf den Stack. Die Variable »b« lebt und ist gültig.**



2 ***los()*** **kommt oben auf den Stack. »x« und »z« leben und sind gültig, und »b« lebt, ist aber *nicht* gültig.**



3 ***verrückt()*** **wird auf den Stack geschoben, und jetzt lebt »c« und ist gültig. Die anderen drei Variablen leben, sind aber nicht in Geltung.**



4 ***verrückt()*** **wird abgeschlossen und vom Stack genommen, »c« ist also nicht mehr gültig *und tot*. Wenn los() wieder aufgenommen wird, wo es verlassen wurde, sind »x« und »z« wieder lebendig und in Geltung. Die Variable »b« lebt immer noch, ist aber nicht in Geltung (bis los() fertig ist).**

Solange eine lokale Variable lebt, bleibt ihr Zustand bestehen. Solange die Methode machZeug() auf dem Stack ist, behält die Variable »b« ihren Wert. Aber die Variable »b« kann nur verwendet werden, solange sich der Stack-Frame von machZeug() oben auf dem Stack befindet. Das heißt, dass Sie eine lokale Variable *nur* verwenden können, solange die Methode dieser lokalen Variablen tatsächlich ausgeführt wird (und nicht einfach nur darauf wartet, dass über ihr liegende Stack-Frames fertig sind).

Was ist mit Referenzvariablen?

Für elementare Variablen und Referenzvariablen gelten die gleichen Regeln. Eine Referenzvariable kann nur innerhalb ihres Geltungsbereichs verwendet werden. Das heißt, Sie können die Fernsteuerung für ein Objekt nur verwenden, wenn Sie eine Referenzvariable haben, die in Geltung ist. Die *eigentliche* Frage ist:

»Wie beeinflusst das *Variablenleben* das *Objektleben*?«

Ein Objekt lebt, solange es lebende Referenzen darauf gibt. Wenn eine Referenzvariable nicht mehr in Geltung, aber immer noch am Leben ist, bleibt das Objekt auf dem Heap, das sie *referenziert*, am Leben. Und dann müssen Sie fragen: »Was passiert, wenn der Stack-Frame, der die Referenz hält, am Ende der Methode vom Stack genommen wird?«

War das die *einzige* Referenz auf das Objekt, wurde dieses Objekt auf dem Heap aufgegeben. Die Referenzvariable hat sich mit dem Stack-Frame aufgelöst, und das aufgegebene Objekt ist jetzt *offiziell* GC-Futter. Entscheidend ist, dass man weiß, wann ein Objekt zur ***Garbage Collection ausgewählt werden darf***.

Wenn ein Objekt zur Garbage Collection (GC) ausgewählt werden darf, müssen Sie sich nicht mehr darum kümmern, den Speicher zurückzufordern, den das Objekt benötigt hat. Steht Ihrem Programm nur noch wenig Speicherplatz zur Verfügung, zerstört die GC einige oder alle zur Garbage Collection auswählbaren Objekte und verhindert so, dass Ihnen das RAM ausgeht. Ihnen kann immer noch der Speicher ausgehen, aber *nicht* bevor alle zur Garbage Collection auswählbaren Objekte zur Kippe gefahren wurden. Ihr Job ist es sicherzustellen, dass Sie Objekte aufgeben (und so zur Garbage Collection auswählbar machen), wenn Sie sie nicht mehr benötigen, damit die GC etwas hat, das sie zurückfordern kann. Wenn Sie an Ihren Objekten kleben, kann die GC Ihnen nicht helfen. Dann riskieren Sie, dass Ihre Programme einen grausamen Tod sterben.

Das Leben eines Objekts hat keinen Wert, keine Bedeutung, keinen Sinn, wenn keiner mehr eine Referenz darauf hält.

Wenn Sie nicht zum Objekt gelangen können, können Sie es nicht bitten, etwas für Sie zu tun. Dann ist es nichts anderes als fetter Haufen Bitmüll.

Aber wenn ein Objekt unerreichbar ist, findet der Garbage Collector das heraus. Früher oder später wird dieses Objekt entsorgt.

Ein Objekt ist zur GC auswählbar, wenn die letzte lebende Referenz darauf weg ist.

Die drei Möglichkeiten, Referenzen auf ein Objekt loszuwerden:

1. Eine Referenz verliert dauerhaft ihre Geltung.

```
void go() {
  Leben z = new Leben();
}
```

Die Referenz z stirbt am Ende der Methode.

2. Der Referenz wird ein anderes Objekt zugewiesen.

```
Leben z = new Leben();
z = new Leben();
```

Das erste Objekt wird aufgegeben, wenn z auf ein neues Objekt >>umprogrammiert<< wird.

3. Die Referenz wird explizit auf null gesetzt.

```
Leben z = new Leben();
z = null;
```

Das erste Objekt wird aufgegeben, wenn die Programmierung von z >>aufgehoben wird<<.

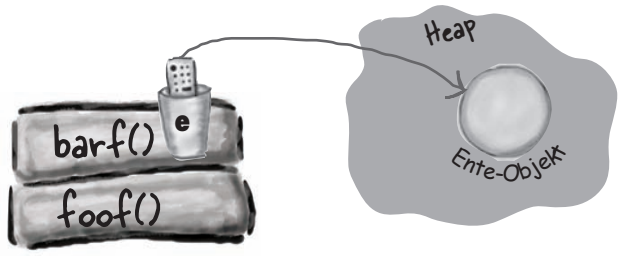
Objekt-Killer Nr. 1

Die Referenz verliert dauerhaft ihre Geltung.

```
public class StackRef {
   public void foof() {
      barf();
   }
   public void barf() {
      Ente e = new Ente();
   }
}
```

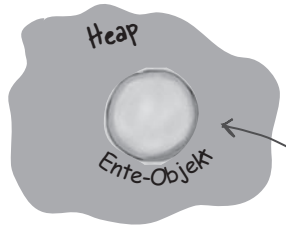
1 ***foof()*** **wird auf den Stack geschoben. Keine Variablen sind deklariert.**

2 ***barf()*** **wird auf den Stack geschoben. Dort deklariert es eine Referenzvariable und erzeugt ein neues Objekt, das dieser Referenz zugewiesen wird. Das Objekt wird auf dem Heap erzeugt, und die Referenz lebt und ist in Geltung.**



Die neue Ente kommt auf den Heap, und solange barf() läuft, lebt die Referenz »e« und ist gültig. Die Ente wird also als lebend betrachtet.

3 ***barf()*** **ist beendet und wird vom Stack genommen. Der Stack-Frame wird aufgelöst, und »e« ist jetzt tot und weg. Die Ausführung kehrt zu *foof()* zurück, aber *foof()* kann »e« nicht verwenden.**

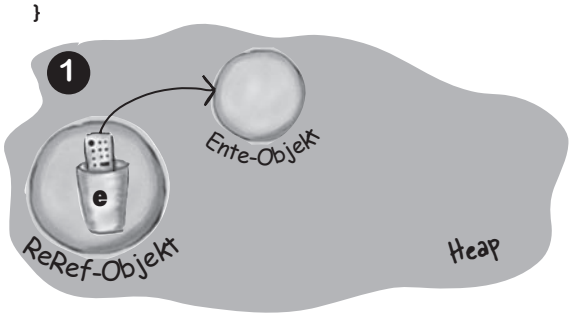


Oje! Als der Stack-Frame für barf() vom Stack geblasen wurde, ging die Variable »e«, und damit wurde die Ente aufgegeben. Garbage Collector-Futter.

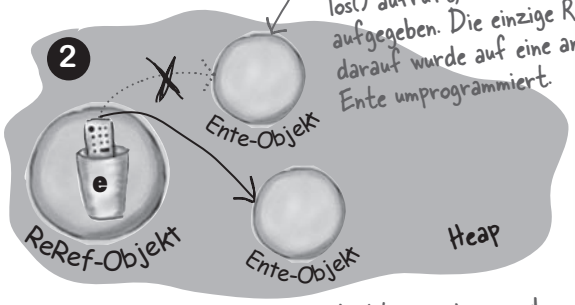
Objekt-Killer Nr. 2

Der Referenz ein anderes Objekt zuweisen.

```
public class ReRef {

    Ente e = new Ente();

    public void los() {
      e = new Ente();
    }
}
```



Die neue Ente kommt auf den Heap und wird von >>e<< referenziert. Da >>e<< eine Instanzvariable ist, lebt die Ente, solange das ReRef-Objekt lebt, das sie instantiiert hatte. Es sei denn ...



Wenn jemand die Methode los() aufruft, wird diese Ente aufgegeben. Die einzige Referenz darauf wurde auf eine andere Ente umprogrammiert.

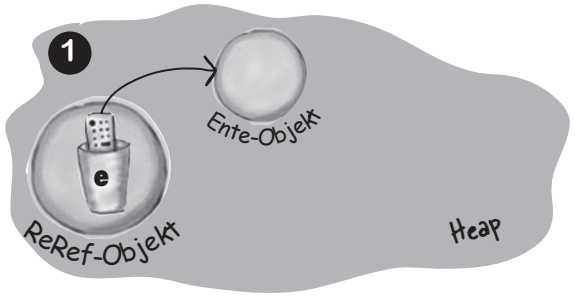
>>e<< erhält ein neues Ente-Objekt zugewiesen und gibt damit das ursprüngliche (erste) Ente-Objekt auf. Die erste Ente ist jetzt so gut wie tot.

Mensch, du hättest doch nur die Referenz neu setzen müssen. Vermutlich kannten die damals noch keine Speicherverwaltung.

Objekt-Killer Nr. 3

Die Referenz explizit auf null setzen

```
public class ReRef {

    Ente e = new Ente();

    public void los() {
      e = null;
    }
}
```



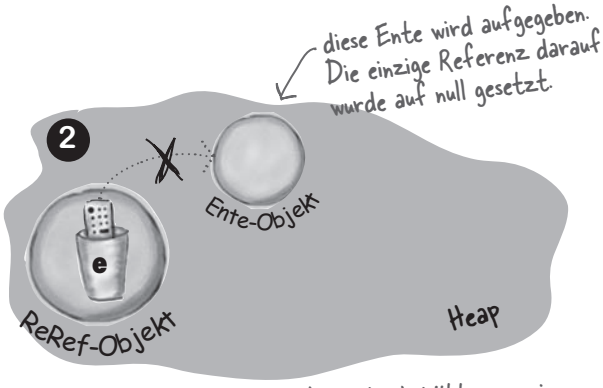
Die neue Ente kommt auf den Heap und wird von >>e<< referenziert. Da >>e<< eine Instanzvariable ist, lebt die Ente, solange das ReRef-Objekt lebt, das sie instantiiert hatte. Es sei denn ...

Die Bedeutung von `null`

Wenn Sie eine Referenz auf **`null`** setzen, heben Sie die Programmierung der Fernsteuerung auf. Mit anderen Worten: Sie haben eine Fernsteuerung, aber am anderen Ende gibt es keinen Fernseher. Eine null-Referenz hat Bits, die »null« repräsentieren (*wir* wissen nicht, was diese Bits sind, und kümmern uns auch nicht darum, das überlassen wir der JVM).

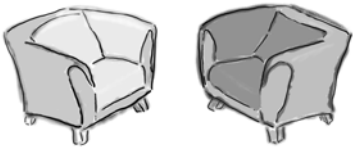
Wenn Sie im richtigen Leben eine nicht programmierte Fernsteuerung haben, passiert nichts, wenn Sie auf eine der Tasten drücken. Aber in Java können Sie auf einer null-Referenz nicht auf die Tasten drücken (d.h. den Punktoperator verwenden). Sie hätten vielleicht gern, dass etwas für Sie bellt, aber die JVM weiß (das ist ein Laufzeitproblem, kein Compiler-Fehler), dass leider kein Hund da ist, der das machen könnte!

Wenn Sie auf einer null-Referenz den Punktoperator verwenden, erhalten Sie zur Laufzeit eine NullPointerException. Im Kapitel *Riskantes Verhalten* werden Sie alles zu Exceptions lernen.



>>e<< wird auf null gesetzt. Das ist, als hätte man eine Fernsteuerung, die auf nichts programmiert ist. Sie dürfen auf >>e<< noch nicht einmal den Punktoperator verwenden, bis sie neu programmiert wird (ihr ein Objekt zugewiesen wird).

Kamingespräche



Heute Abend: **Eine Instanzvariable und eine lokale Variable sprechen (mit erstaunlicher Zurückhaltung) über Leben und Tod.**

Instanzvariable

Ich würde gern anfangen, weil ich für Programme meist wichtiger bin als eine lokale Variable. Ich soll Objekte unterstützen. In der Regel über die ganze Lebensdauer des Objekts hinweg. Was soll schon ein Objekt ohne *Zustand* sein? Und was ist ein Zustand? Werte, die mit ***Instanzvariablen*** festgehalten werden.

Nein. Versteh mich nicht falsch. Deine Bedeutung in Methoden ist mir vollkommen klar. Es ist nur, dass dein Leben so kurz ist. So vorübergehend. Deswegen nennt man euch Typen doch »temporäre Variablen«.

Entschuldige vielmals. Das verstehe ich vollkommen.

Darüber habe ich noch nie nachgedacht. Was macht ihr, wenn die anderen Methoden laufen und ihr darauf wartet, dass euer Frame wieder oben auf dem Stack liegt?

Lokale Variable

Ich kann deine Sichtweise nachvollziehen. Die Bedeutung von Objektzuständen und all dem ist mir natürlich vollkommen klar. Man sollte die Leute aber nicht in die Irre führen. Lokale Variablen sind *echt* wichtig. Um mal eine deiner Formulierungen zu verwenden: »Was soll schon ein Objekt ohne *Verhalten* sein?« Und was ist Verhalten? Algorithmen in Methoden. Und du kannst deine Bits darauf wetten, dass da ein paar *lokale Variablen* im Spiel sind, die dafür sorgen, dass diese Algorithmen funktionieren.

In der Lokale-Variablen-Gemeinschaft wird die Bezeichnung »temporäre Variablen« als abqualifizierend betrachtet. Wir ziehen »lokale Variable«, »Stack-Variable«, »automatische Variable« oder »geltungseingeschränkte Variable« vor.

Trotzdem ist es richtig, dass wir kein langes Leben haben und dass es auch kein besonders *gutes* Leben ist. Erst werden wir mit all den anderen lokalen Variablen in einen Stack-Frame gepresst ... und wenn dann die Methode, von der wir ein Teil sind, eine andere Methode aufruft, wird auf uns auch noch ein anderer Frame gepackt. Und wenn *diese* Methode ihrerseits wieder eine *andere* Methode aufruft ... und so weiter. Manchmal müssen wir Ewigkeiten warten, bis die Methoden, die auf dem Stack über uns liegen, fertig sind, damit unsere Methode wieder laufen kann.

Nichts. Gar nichts. Das ist, als wäre man in Tiefschlaf – was die Leute in Science-Fiction-Filmen immer machen, wenn sie große Entfernungen überbrücken müssen. Eigentlich eine Art ausgesetztes Leben. Wir sitzen einfach da in der Warteschleife. Solange unser Frame noch da ist, sind

Instanzvariable

Ich habe mal einen Aufklärungsfilm darüber gesehen. Sieht nach einem ziemlich grausamen Ende aus. Wenn die Methode endet, weil sie auf die schließende geschweifte Klammer stößt, wird der Frame im Wortsinn vom Stack *geblasen*! *Das* muss schon schmerzhaft sein.

Ich lebe mit den Objekten auf dem Heap. Na, nicht *mit* den Objekten, sondern eigentlich *in* einem Objekt. Dem Objekt, dessen Zustand ich festhalte. Ich muss zugeben, dass das Leben auf dem Heap eine ziemlich luxuriöse Angelegenheit sein kann. Viele von uns fühlen sich schuldig, insbesondere während der Feiertage.

Okay. Theoretisch ja. Wenn ich eine Instanzvariable von Halsband bin und das Halsband von der GC geschluckt wird, werden auch die Instanzvariablen von Halsband weggeworfen wie die Zeitung vom Vortag. Aber mir hat man erzählt, das würde fast nie passieren.

Sie lassen uns *trinken*?

Lokale Variable

wir und der Wert, den wir festhalten, sicher. Aber es ist eine zweifelhafte Freude, wenn unser Frame wieder läuft. Einerseits sind wir endlich wieder aktiv. Andererseits nagt die Uhr wieder an unseren kurzen Leben. Je länger unsere Methode läuft, um so näher kommen wir dem Ende der Methode. Und wir wissen *alle*, was dann passiert.

Das musst du mir nicht sagen. Normalerweise verwendet man den Begriff *entfernen*, wie in »der Frame wurde vom Stack entfernt«. Das klingt so sachlich, so nach Hausputz. Aber du hast den Film ja gesehen. Warum reden wir nicht ein bisschen über dich? Ich weiß, wie mein kleiner Stack-Frame aussieht, aber wo lebst *du* eigentlich?

Aber ihr lebt nicht *immer* so lang wie das Objekt, das euch deklariert hat, oder? Nehmen wir beispielsweise an, es gibt ein Hund-Objekt mit einer Halsband-Instanzvariable. Stell dir vor, *du* wärst eine Instanzvariable des *Halsband*-Objekts. Vielleicht eine Referenz auf eine Namensschild-Instanz oder so was. Und du sitzt da glücklich in dem *Halsband*-Objekt, das glücklich in dem *Hund*-Objekt sitzt. Aber ... was passiert, wenn der Hund ein neues Halsband will oder seine Halsband-Instanzvariable auf *null* setzt? Dann kann das Halsband-Objekt für die GC ausgewählt werden. Und wenn *du* eine Instanzvariable in Halsband bist und das ganze *Halsband* aufgegeben wird, was passiert dann mit *dir*?

Und das hast du geglaubt? Das erzählen die uns, damit wir motiviert und produktiv bleiben. Aber vergisst du da nicht etwas? Was ist, wenn du eine Instanzvariable in einem Objekt bist und dieses Objekt *nur* von einer *lokalen* Variablen referenziert wird? Wenn ich die einzige Referenz auf das Objekt bin, in dem du steckst, nehme ich dich mit, wenn ich gehe. Ob es dir gefällt oder nicht, unser Schicksal kann verknüpft sein. Deswegen schlage ich vor, dass wir den ganzen Kram jetzt vergessen und uns besaufen, so lange wir die Zeit dazu haben. Carpe RAM und alles.

Spielen Sie Garbage Collector

Welche der Codezeilen auf der rechten Seite würde bewirken, dass genau ein weiteres Objekt für den Garbage Collector auswählbar wird, wenn sie am Punkt A eingefügt wird? (Gehen Sie davon aus, dass die Ausführung des Punkts A (// weitere Methoden aufrufen) lange dauert und dem Garbage Collector Zeit gibt, seine Arbeit zu machen.)

```
public class GC {
  public static GC machZeug() {
    GC newGC = new GC();
    machZeug2(newGC);
    return newGC;
  }

  public static void main(String [] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = machZeug();
    A
    // weitere Methoden aufrufen
  }

  public static void machZeug2(GC gcKopie) {
    GC lokalesGC;
  }
}
```

1 `gcKopie = null;`

2 `gc2 = null;`

3 `newGC = gc3;`

4 `gc1 = null;`

5 `newGC = null;`

6 `gc4 = null;`

7 `gc3 = gc2;`

8 `gc1 = gc4;`

9 `gc3 = null;`

Beliebte Objekte

In diesem Codebeispiel werden mehrere neue Objekte erzeugt. Sie haben die Aufgabe herauszufinden, welches Objekt das »beliebteste« ist, d.h., auf welches die meisten Referenzvariablen verweisen. Geben Sie dann an, wie *viele* Referenzen es für dieses Objekt insgesamt gibt und welche das sind! Als Hinweis zeigen wir Ihnen eins der neuen Objekte und die Referenzvariable, die es referenziert.

Viel Glück!

```
class Bienen {
  Honig [] bienenHonig;
}

class Waschbär {
  Topf t;
  Honig waschbärHonig;
}

class Topf {
  Honig honig;
}

class Bär {
  Honig bärenHonig;
}

public class Honig {
  public static void main(String [] args) {
    Honig honigTopf = new Honig();
    Honig [] honig = {honigTopf, honigTopf, honigTopf, honigTopf};
    Bienen b1 = new Bienen();
    b1.bienenHonig = honig;
    Bär [] bären = new Bär[5];
    for (int x=0; x < 5; x++) {
      bären[x] = new Bär();
      bären[x].bärenHonig = honigTopf;
    }
    Topf t = new Topf();
    t.honig = honigTopf;
    Waschbär wb = new Waschbär();

    wb.waschbärHonig = honigTopf;
    wb.t = t;
    t = null;
  } // main-Ende
}
```

Hier ist ein neues Waschbär-Objekt!

Hier ist seine Referenzvariable »wb«.

Kurzkrimi

»Wir haben die Simulation viermal laufen lassen, aber die Temperatur des Hauptmoduls wandert immer vom Nominalwert abwärts«, sagte Sarah entnervt. »Letzte Woche haben wir unsere neuen Temperatur-Bots installiert. Die Messwerte der Kühler-Bots, die die Kühlung der Wohnquartiere unterstützen sollen, scheinen der Spec zu entsprechen. Deswegen haben wir unsere Analyse auf die Wärmespeicher-Bots konzentriert, die Bots, die das Heizen der Wohnquartiere unterstützen sollen.« Tim seufzte. Zunächst schien es, dass die Nano-Technologie ihnen einen Vorsprung im Zeitplan verschaffen würde. Aber jetzt, da nur noch fünf Wochen bis zum Start blieben, bestanden einige der wichtigen Lebenserhaltungssysteme der Raumstation den Simulationsspießrutenlauf immer noch nicht.

»Welche Verhältnisquotienten simuliert ihr?«, fragte Tim.

»Wenn ich richtig verstehe, worauf du hinauswillst, dann haben wir das bereits in Betracht gezogen«, erwiderte Sarah. »Mission Control akzeptiert die kritischen Systeme nicht, wenn wir im Test den Rahmen der Spezifikation verlassen. Wir müssen die V3-Kühler-Bot-Sim-Einheiten in einem 2:1-Verhältnis mit den V2-Kühler-SimEinheiten testen«, fuhr Sarah fort. »Insgesamt soll das Verhältnis von Wärmespeicher-Bots zu Kühler-Bots 4:3 betragen.«

»Wie ist der Energieverbrauch, Sarah?«, fragte Tim. Sarah machte eine Pause. »Das ist ein weiteres Problem. Der Energieverbrauch ist höher als vorhergesehen. Auch darauf haben wir ein Team angesetzt. Aber weil die Nanos drahtlos sind, ist es schwierig, den Energieverbrauch der Kühler-Bots von dem der Wärmespeicher-Bots zu trennen.« »Insgesamt soll der Energieverbrauch«, ergänzte Sarah, »im Verhältnis 3:2 stehen, wobei die Kühler mehr Energie aus dem Wireless Grid ziehen dürfen.«

»Okay, Sarah«, sagte Tim. »Werfen wir einen Blick auf die Initialisierungsteile des Simulationscodes. Wir müssen dieses Problem finden, und wir müssen es schnell finden!«

```
import java.util.*;
class V2Kühler {
  V2Kühler(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimEinheit("V2Kühler"));
    }
  }
}

class V3Kühler extends V2Kühler {
  V3Kühler(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimEinheit("V3Kühler"));
    }
  }
}

class WärmespeicherBot {
  WärmespeicherBot(ArrayList rlist) {
    rlist.add(new SimEinheit("Wärmespeicher"));
  }
}
```

Kurzkrimi (Fortsetzung)

```
public class TestLebenserhaltungsSim {
  public static void main(String [] args) {
    ArrayList aList = new ArrayList();
    V2Kühler v2 = new V2Kühler(aList);
    V3Kühler v3 = new V3Kühler(aList);
    for(int z=0; z<20; z++) {
      WärmespeicherBot speicher = new WärmespeicherBot(aList);
    }
  }
}

class SimEinheit {
  String botTyp;
  SimEinheit(String typ) {
    botTyp = typ;
  }
  int energieVerbrauch() {
    if ("Wärmespeicher".equals(botTyp)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Tim warf einen kurzen Blick auf den Code, und da erschien ein kleines Lächeln auf seine Lippen. »Ich glaube, ich habe das Problem gefunden, Sarah. Und ich wette, ich weiß auch, um wie viel Prozent deine Energieverbrauchswerte abweichen!«

Welchen Verdacht hatte Tim? Wie konnte er die Abweichung der Energieverbrauchswerte raten? Welche paar Codezeilen würden Sie dem Programm hinzufügen, um den Fehler zu beheben?

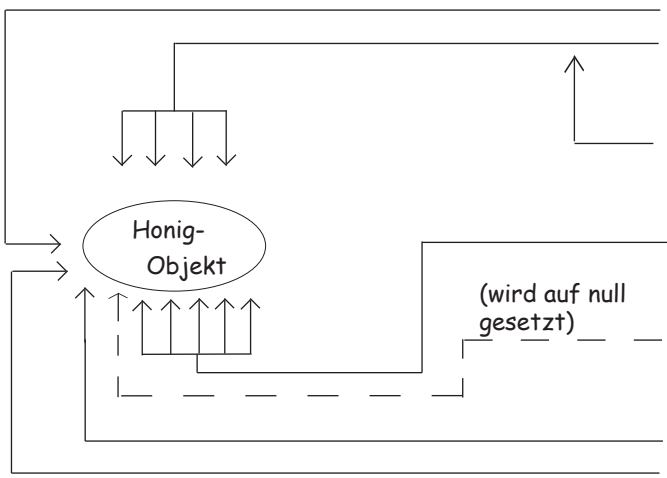
Lösungen zu den Übungen

G.C.

1	`gcKopie = null;`	Nein – diese Zeile versucht auf eine Variable zuzugreifen, die nicht mehr in Geltung ist.
2	`gc2 = null;`	Okay – gc2 war die einzige Referenzvariable, die das Objekt referenzierte.
3	`newGC = gc3;`	Nein – wieder eine Variable, die nicht mehr in Geltung ist.
4	`gc1 = null;`	Okay – gc1 hielt die einzige Referenz, weil newGC bereits tot ist.
5	`newGC = null;`	Nein – newGC ist nicht mehr in Geltung.
6	`gc4 = null;`	Nein – das Objekt wird immer noch von gc3 referenziert.
7	`gc3 = gc2;`	Nein – das Objekt wird immer noch von gc4 referenziert.
8	`gc1 = gc4;`	Okay – der einzigen Referenz auf das Objekt wird ein neues Objekt zugewiesen.
9	`gc3 = null;`	Nein – das Objekt wird immer noch von gc4 referenziert.

Beliebte Objekte

Wahrscheinlich war es nicht besonders schwer herauszufinden, dass das Honig-Objekt, das zuerst von der Variablen honigTopf referenziert wurde, das bei Weitem »beliebteste« Objekt in dieser Klasse ist. Aber es war vielleicht etwas schwerer zu erkennen, dass alle Variablen aus dem Code, die auf das Honig-Objekt zeigen, ***dasselbe Objekt referenzieren***! Es gibt insgesamt zwölf aktive Referenzen auf das Objekt, bevor die main( )-Methode fertig ist. Die Variable *t.honig* ist eine Zeit lang gültig, aber t wird am Ende auf null gesetzt. Da *wb.t* immer noch auf das Topf-Objekt verweist, referenziert *wb.t.honig* das Objekt immer noch (obwohl es nie explizit deklariert wurde)!




```
public class Honig {
  public static void main(String [] args) {
    Honig honigTopf = new Honig();
    Honig [] honig = {honigTopf, honigTopf,
                      honigTopf, honigTopf};
    Bienen b1 = new Bienen();
    b1.bienenHonig = honig;
    Bär [] bären = new Bär[5];
    for (int x=0; x < 5; x++) {
      bären[x] = new Bär();
      bären[x].bärenHonig = honigTopf;
    }
    Topf t = new Topf();
    t.honig = honigTopf;
    Waschbär wb = new Waschbär();

    wb.waschbärHonig = honigTopf;
    wb.t = t;
    t = null;
  } // main-Ende
}
```

Kurzkrimi-Lösung

Tim bemerkte, dass der Konstruktor für die Klasse V2Kühler eine ArrayList erwartete. Das bedeutete, dass jedes Mal, wenn der *V3*Kühler-Konstruktor aufgerufen wurde, im super()-Aufruf an den V2Kühler-Konstruktor eine ArrayList übergeben wurde. Das hieß, dass jedes Mal fünf zusätzliche V2Kühler-SimEinheiten erzeugt wurden. Wenn Tim Recht hatte, würde der Gesamtenergieverbrauch 120 betragen und nicht die 100, die Sarahs angenommene Verhältnisquotienten erwarten ließen.

Da alle Bot-Klassen SimEinheiten erstellen, hätte man das Problem leicht herausfinden können, indem man einen SimEinheit-Konstruktor geschrieben hätte, der jedes Mal eine Zeile ausgibt, wenn eine SimEinheit erzeugt wird!