# Java von Kopf bis Fuß

Eine abwechslungsreiche Entdeckungsreise
durch die objektorientierte Programmierung

# Leben und Sterben eines Objekts


**Objekte werden geboren und Objekte sterben.** Sie als Programmierer/-in sind für den Lebenszyklus eines Objekts verantwortlich. Sie entscheiden, wie es **konstruiert** wird. Sie entscheiden, wenn es **zerstört** wird. Nur, dass Sie das Objekt nicht selbst *zerstören*, Sie *geben* es einfach *auf*. Aber sobald Sie es aufgegeben haben, kann der herzlose **Garbage Collector (gc)** es pulverisieren und den Speicher, den das Objekt belegte, wieder anderweitig nutzen. Wenn Sie Java schreiben, werden Sie Objekte erstellen. Früher oder später müssen Sie einige von ihnen gehen lassen, oder Sie riskieren, dass der Arbeitsspeicher knapp wird. In diesem Kapitel sehen wir uns an, wie Objekte erstellt werden, wo sie leben, während sie leben, und wie man sie effizient gehen lässt. Das heißt, wir sprechen über den Heap, den Stack, den Geltungsbereich (Scope), Konstruktoren, Superklassenkonstruktoren, Nullreferenzen und mehr. Warnung: Dieses Kapitel enthält Material über den Tod von Objekten, den Sie als Lesende möglicherweise beunruhigend finden. Am besten gehen Sie keine emotionale Bindung ein.

# Der Stack und der Heap: wo das Leben spielt

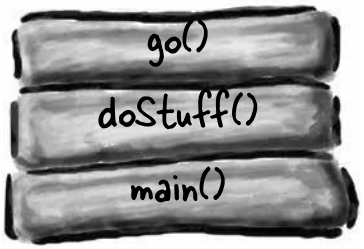
Bevor wir verstehen können, was bei der Erstellung von Objekten tatsächlich passiert, müssen wir etwas Abstand nehmen. Wir müssen mehr darüber lernen, wo die Dinge in Java leben (und wie lange). Das heißt, wir müssen zwei Speicherbereiche genauer betrachten: den Stack und den Heap. Wenn die JVM startet, erhält sie einen Teil des Arbeitsspeichers vom zugrunde liegenden OS zugewiesen und verwendet diesen, um Ihr Java-Programm auszuführen. Wie *viel* Speicher das ist und ob Sie ihn verändern können, hängt davon ab, welche Version der JVM Sie verwenden (und auf welcher Plattform). Aber normalerweise haben Sie auf diese Dinge *keinen* Einfluss. Und bei guter Programmierung sollte Sie das auch nicht weiter kümmern (mehr dazu später).

In Java sind uns (den Programmierern) die Speicherbereiche wichtig, in denen die Objekte leben (der Heap), und die, in denen die Methodenaufrufe und lokalen Variablen leben (der Stack).

Wir wissen, dass alle *Objekte* auf dem Garbage Collectible Heap leben. Was wir noch nicht wissen, ist, wo die *Variablen* wohnen. Und das hängt von der *Art* der Variablen ab. Mit »Art« meinen wir nicht den *Typ* (also elementar oder Objekt). Die zwei *Arten* von Variablen, deren Leben uns hier wichtig ist, sind *Instanz*variablen und *lokale* Variablen. Lokale Variablen werden auch als *Stack*-Variablen bezeichnet, was bereits ein deutlicher Hinweis auf ihren Lebensraum ist.

### Der Stack

Wo Methodenaufrufe und lokale Variablen wohnen.




### Der Heap

Wo ***ALLE*** Objekte wohnen.


## Instanzvariablen

**Instanzvariablen werden innerhalb einer *Klasse*, aber *nicht* innerhalb einer Methode deklariert**. Sie stehen für »Felder«, die jedes Objekt besitzt (und die mit unterschiedlichen Werten für jede Instanz der Klasse gefüllt werden können). Instanzvariablen wohnen innerhalb des Objekts, zu dem sie gehören.

```
public class Duck {

   int size;

}
```

Jedes Duck-Objekt hat eine Instanzvariable namens >>size<<.

## Lokale Variablen

**Lokale Variablen werden in einer *Methode* deklariert, auch die Methodenparameter.** Sie haben eine begrenzte Lebenszeit und leben nur so lange, wie sich die Methode auf dem Stack befindet (also solange die Methode die schließende geschweifte Klammer noch nicht erreicht hat).

```
public void foo(int x) {
  int i = x + 3;
  boolean b = true;
}
```

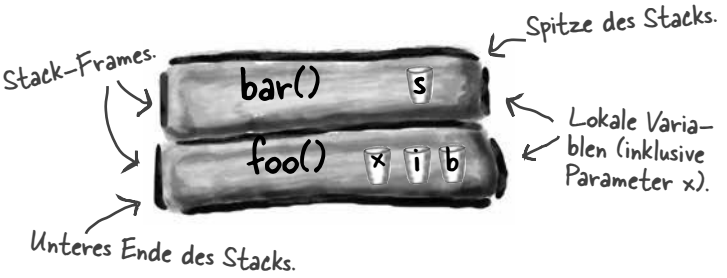
Der Parameter x und die Variablen i und b sind alles lokale Variablen.

# Methoden werden gestapelt

Rufen Sie eine Methode auf, landet diese oben auf einem Call-Stack (Aufrufstapel). Das neue Ding, in das die Methode geschoben wird, ist der Stack-*Frame*. Er enthält den Zustand der Methode inklusive der gerade ausgeführten Codezeile und den Werten der lokalen Variablen.

An der *Spitze* des Stacks befindet sich immer die aktuell ausgeführte Methode des Stacks. (Im Moment nehmen wir an, es gäbe nur einen Stack. In Kapitel 14 werden wir weitere Stacks hinzufügen.) Eine Methode bleibt auf dem Stack, bis ihre schließende geschweifte Klammer erreicht ist (was heißt, die Methode hat ihre Arbeit getan). Wenn die Methode foo() die Methode bar() aufruft, wird bar() auf dem Stack oberhalb von foo() eingefügt.

Ein Call-Stack mit zwei Methoden




**Die Methode an der Spitze des Stacks ist immer die aktuell ausgeführte Methode.**

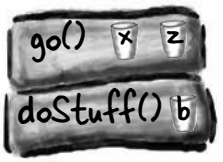
```
public void doStuff() {
  boolean b = true;
  go(4);
}

public void go(int x) {
  int z = x + 24;
  crazy();
  // mehr Code ...
}

public void crazy() {
  char c = 'a';
}
```

## Ein Stack-Szenario

Der Code auf der linken Seite ist nur ein Schnipsel mit drei Methoden (der Rest interessiert uns hier nicht). Die erste Methode (doStuff()) ruft die zweite Methode (go()) auf, und die zweite Methode ruft die dritte (crazy()) auf. Jede Methode deklariert in ihrem Körper eine lokale Variable (*b*, *z* und *c*). Außerdem deklariert go() eine Parametervariable (was bedeutet, dass go() zwei lokale Variablen, *x* und *z*, besitzt).

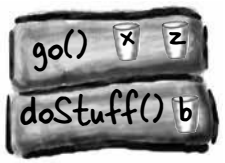
1. Code aus einer anderen Klasse ruft **doStuff()** auf, und **doStuff()** landet im Stack-Frame an der Spitze des Stacks. Die boolesche Variable namens »**b**« landet im Stack-Frame von **doStuff()**.


2. **doStuff()** ruft **go()** auf, und **go()** wird an die Spitze des Stacks geschoben. Die Variablen »**x**« und »**z**« befinden sich im Stack-Frame von **go()**.




3. **go()** ruft **crazy()** auf. Jetzt befindet sich **crazy()** an der Spitze des Stacks mit der Variablen »**c**« im entsprechenden Frame.

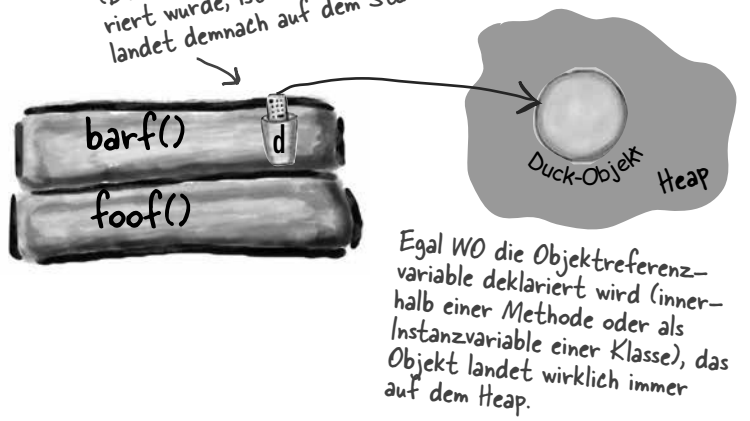

4. **crazy()** wird beendet, und ihr Stack-Frame wird vom Stack entfernt. Die Ausführung geht zurück zur Methode **go()** und macht auf der Zeile weiter, die auf den Aufruf von **crazy()** folgt.



# Was ist mit lokalen Variablen, die Objekte sind?

Wie Sie wissen, enthält eine nicht elementare Variable eine *Referenz* auf ein Objekt, nicht das Objekt selbst. Sie wissen bereits, dass Objekte auf dem Heap leben. Dabei spielt es keine Rolle, wo sie deklariert oder erstellt wurden. ***Ist eine lokale Variable eine Referenz auf ein Objekt, landet nur die Variable (Referenz/Fernbedienung) auf dem Stack.***

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

   public void barf() {
      Duck d = new Duck();
   }
}
```




## Es gibt keine Dummen Fragen

**F: Also noch einmal: WARUM lernen wir diesen ganzen Stack/Heap-Kram? Wie hilft mir das weiter? Muss ich das wirklich alles wissen?**

A: Die Kenntnis der Grundlagen von Java-Stack und -Heap ist entscheidend für das Verständnis von Variablen-Geltungsbereichen (Scope), Problemen bei der Objekterstellung, Speicherverwaltung, Threads und dem Exception-Handling. Mit Threads und Exception-Handling werden wir uns in späteren Kapiteln befassen. Sie müssen übrigens nicht wissen, *wie* Stack und Heap in einer bestimmten JVM und/oder Plattform implementiert sind. Alles, was Sie über den Stack und den Heap wissen müssen, finden Sie auf dieser und der vorherigen Seite. Wenn Sie diese Seiten wirklich verstanden haben, werden alle anderen Themen, die von Ihrem Verständnis der hier erklärten Dinge abhängen, viel, viel, viel einfacher werden. Also noch mal: Eines Tages werden Sie uns SO dankbar sein, dass wir Sie mit Stacks und Heaps regelrecht gemästet haben.

### PUNKT FÜR PUNKT

- Java hat zwei Speicherbereiche, die uns wichtig sind: den Stack und den Heap.
- Instanzvariablen werden innerhalb einer Klasse, aber außerhalb von Methoden deklariert.
- Lokale Variablen werden innerhalb von Methoden oder Methodenparametern deklariert.
- Alle lokalen Variablen leben auf dem Stack in dem Frame, der zu der Methode gehört, in der die Variablen deklariert werden.
- Objektreferenzvariablen funktionieren genau wie elementare Variablen – wird die Referenz als lokale Variable deklariert, landet sie auf dem Stack.
- Alle Objekte leben auf dem Heap, unabhängig davon, ob die Referenz eine lokale oder eine Instanzvariable ist.

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

Wenn Sie new CellPhone() schreiben, muss Java auf dem Heap Platz für das neue Handy schaffen. Aber *wie viel* Platz? Genug für das Objekt, also genug, um alle Instanzvariablen des Objekts aufnehmen zu können. Richtig gelesen, Instanzvariablen leben auf dem Heap *innerhalb* des Objekts, zu dem sie gehören.

Erinnern Sie sich, dass die *Werte* einer Objektinstanz innerhalb des Objekts leben. Sind alle Instanzvariablen elementar, schafft Java auf Basis der elementaren Typen Platz für die Instanzvariablen. Ein int benötigt 32 Bits, ein long 64 Bits etc. Dabei ist Java der Wert innerhalb der elementaren Variablen egal. Die Bitgröße einer int-Variablen ist immer gleich (32 Bits), egal ob der Wert 32.000.000 lautet oder nur 32.

Was aber, wenn die Instanzvariablen *Objekte* sind? Was, wenn CellPhone EINE Antenna HAT, also wenn ein CellPhone eine Referenzvariable des Typs Antenna besitzt?

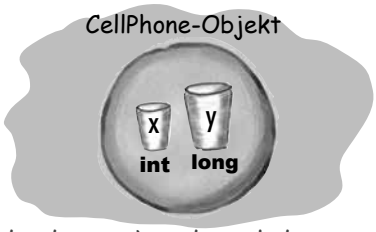
Hat das neue Objekt Instanzvariablen, die Objektreferenzen und keine elementaren Typen sind, lautet die eigentliche Frage: Braucht das Objekt Platz für alle anderen Objekte, die es enthält? Die Antwort lautet: *Nicht ganz*. Wie auch immer, Java muss Platz für die *Werte* der Instanzvariablen schaffen. Aber vergessen Sie nicht, dass eine Referenzvariable nicht das *Objekt selbst* ist, sondern nur eine *Fernbedienung* für das Objekt. Enthält CellPhone also eine Instanzvariable, die als nicht elementarer Typ Antenna deklariert wurde, schafft Java innerhalb des CellPhone-Objekts nur für die *Fernbedienung* des Antenna-Objekts Platz (d. h. für die Referenzvariable), aber nicht für das Antenna-*Objekt*.

Dann stellt sich natürlich die Frage, wann dem Antenna-*Objekt* Platz auf dem Heap zugewiesen wird. Dafür müssen wir zuerst herausfinden, *wann* das Antenna-Objekt erstellt wurde. Und das hängt von der Deklaration der Instanzvariablen ab. Wird die Instanzvariable deklariert, ohne ihr ein Objekt zuzuweisen, wird nur Platz für die Instanzvariable (Fernbedienung) geschaffen.

```
private Antenna ant;
```

Auf dem Heap wird erst dann ein Antenna-Objekt angelegt, wenn oder sofern der Referenzvariablen tatsächlich ein neues Antenna-Objekt zugewiesen wird.

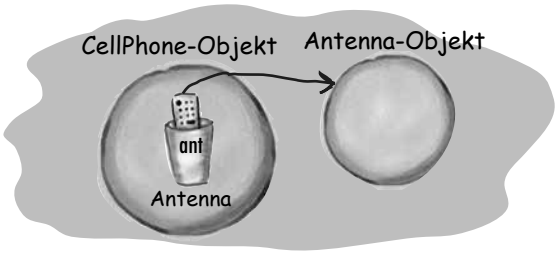
```
private Antenna ant = new Antenna();
```




Objekt mit zwei elementaren Instanzvariablen. Der Platz für die Variablen befindet sich im Objekt.


Objekt mit einer nicht elementaren Instanzvariablen, einer Referenz auf ein Antenna-Objekt, aber kein tatsächliches Objekt. Das bekommen Sie, wenn Sie die Variable deklarieren, aber nicht mit einem tatsächlichen Antenna-Objekt initialisieren.

```
public class CellPhone {
  private Antenna ant;
}
```




Objekt mit einer nicht elementaren Instanzvariablen. Der Variablen Antenna wird ein neues Antenna-Objekt zugewiesen.

```
public class CellPhone {
  private Antenna ant = new Antenna();
}
```

# Das Wunder der Objekterstellung

Nachdem Sie wissen, wo Variablen und Objekte leben, können wir eintauchen in die wunderbare Welt der Objekterstellung. Erinnern Sie sich an die drei Schritte der Objektdeklaration: Deklarieren Sie eine Referenzvariable, erstellen Sie ein Objekt und weisen Sie das Objekt der Referenz zu.

Bisher geschah das in Schritt 2 beschriebene »Wunder der Objektgeburt«, die für uns ein großes Rätsel blieb. Machen Sie sich bereit für die Tatsachen des Lebens der Objekte. *Hoffentlich sind Sie nicht zimperlich.*

**Sehen wir uns noch einmal die drei Schritte der Objektdeklaration, -erstellung und -zuweisung an:**

Erstellen Sie eine neue Referenzvariable eines Klassen- oder Interface-Typs.

**1** Deklarieren Sie eine Referenzvariable.

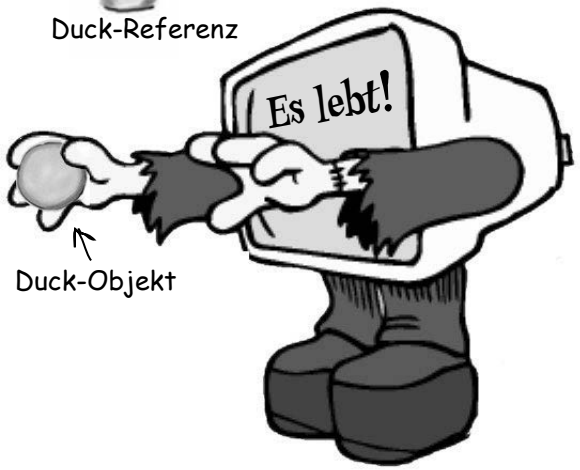
```
Duck myDuck = new Duck();
```


Duck-Referenz

Hier geschieht ein Wunder.

**2** Erstellen Sie ein Objekt.

```
Duck myDuck = new Duck();
```

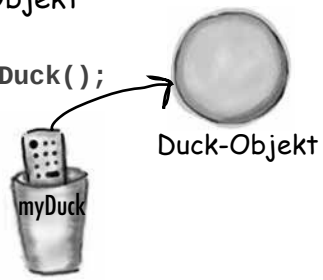



Duck-Objekt

Weisen Sie der Referenz das neue Objekt zu.

**3** Verbinden Sie das Objekt und die Referenz.

```
Duck myDuck = new Duck();
```




Duck-Referenz

Duck-Objekt

**Rufen wir etwa eine Methode namens Duck() auf? Denn genau so sieht es aus.**

```
Duck myDuck = new Duck();
```

Wegen der runden Klammern sieht es so aus, als riefen wir eine Methode namens Duck() auf

**Nein.**

**Wir rufen den Duck-*Konstruktor* auf.**

Ein Konstruktor sieht einer Methode sehr *ähnlich*, er ist aber keine. Der enthaltene Code wird ausgeführt, wenn Sie **new** sagen. Anders gesagt, *dies ist der Code, der ausgeführt wird, wenn Sie ein Objekt instanziieren.*

Die einzige Möglichkeit, einen Konstruktor aufzurufen, besteht in dem Schlüsselwort **new**, gefolgt vom Namen der gewünschten Klasse. Die JVM findet diese Klasse und ruft ihren Konstruktor auf. (Na gut, technisch gesehen ist das nicht der *einzige Weg*, einen Konstruktor aufzurufen. Aber es ist die einzige Möglichkeit, das von *außerhalb* eines Konstruktors zu tun. Sie *können* einen Konstruktor auch innerhalb eines anderen Konstruktors aufrufen, allerdings ist das mit bestimmten Einschränkungen verbunden, wie wir später in diesem Kapitel sehen werden.)

**Ein Konstruktor enthält den Code, der bei der Instanziierung eines Objekts ausgeführt wird, also der Code, der läuft, wenn Sie new auf einem Klassentyp aufrufen.**

**Jede Klasse, die Sie schreiben, besitzt einen Konstruktor, selbst wenn Sie ihn nicht selbst schreiben.**

**Wenn wir ihn nicht schreiben, wer macht das dann?**

Sie können den Konstruktor für Ihre Klasse selbst schreiben (und das machen wir auch gleich). Tun Sie das nicht, ***erzeugt der Compiler einen Konstruktor für Sie***.

So sieht der Standardkonstruktor des Compilers aus:

```
public  Duck() {

}
```

**Fehlt da was? Wo ist der Unterschied zu anderen Methoden?**

Sein Name entspricht dem der Klasse. Das ist obligatorisch.

Wo ist der Rückgabetyp? Wäre dies eine normale Methode, müssen Sie zwischen »public« und »Duck()« einen Rückgabetyp angeben.

```
public  Duck() {
   // Konstruktorcode steht hier
}
```

# Ein Duck-Objekt konstruieren

Die wichtigste Eigenschaft eines Konstruktors ist, dass er ausgeführt wird, *bevor* das Objekt der Referenz zugewiesen werden kann. Sie haben also die Möglichkeit, einzuschreiten und Dinge zu tun, um das Objekt für seine Verwendung vorzubereiten. Das heißt, bevor jemand die Fernbedienung für ein Objekt benutzen kann, hat das Objekt die Gelegenheit, bei seiner eigenen Erstellung zu helfen. In unserem Duck-Konstruktor passiert nicht viel Sinnvolles. Wir zeigen hier nur die Abfolge der Ereignisse.


```
public class Duck {

   public Duck() {
      System.out.println("Quack");
   }
}
```

Konstruktorcode.

**Der Konstruktor gibt Ihnen die Möglichkeit, in der Mitte von new einzugreifen.**

```
public class UseADuck {

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

Dies ruft den Duck-Konstruktor auf.

```
Datei Bearbeiten Fenster Hilfe Quak
% java UseADuck
Quack
```

## Spitzen Sie Ihren Bleistift

→ Das schaffen Sie selbst.

Mit einem Konstruktor können Sie mitten in den Schritt der Objekterstellung, in die Mitte von **new**, hineinspringen. Fallen Ihnen Situationen ein, in denen das nützlich sein könnte? Welche der Aktionen auf der rechten Seite könnten in einem Konstruktor für eine Car-(Auto-)Klasse nützlich sein, wenn das Car-Objekt Teil eines Rennspiels sein soll? Markieren Sie alle Punkte, für die Ihnen ein Szenario eingefallen ist.

- ■ Einen Zähler inkrementieren, um mitzuverfolgen, wie viele Objekte dieses Klassentyps erzeugt wurden.
- ■ Einen bestimmten Laufzeitzustand zuweisen (Daten darüber, was JETZT gerade passiert).
- ■ Den wichtigen Instanzvariablen Werte zuweisen.
- ■ Eine Referenz auf das Objekt, das das neue Objekt *erstellt*, erhalten und speichern.
- ■ Das Objekt einer ArrayList hinzufügen.
- ■ HAT-EIN-Objekte erstellen.
- ■ _____ (Ihre eigenen Ideen)

## Den Zustand einer neuen Duck initialisieren

Meistens werden Konstruktoren verwendet, um den Zustand eines Objekts zu initialisieren, das heißt, um Werte für die Instanzvariablen des Objekts zu erzeugen und diese zuzuweisen.

```
public Duck() {
   size = 34;
}
```

Das ist alles schön und gut, solange der *Entwickler* der Klasse weiß, wie groß die Ente sein sollte. Was machen wir aber, wenn ein Programmierer, der das Duck-Objekt *benutzt,* selbst über die Größe einer bestimmten Ente entscheiden soll?

Angenommen, Duck besitzt eine Instanzvariable namens size (Größe) und der Programmierer soll Ihre Duck-Klasse benutzen können, um die Größe des neuen Duck-Objekts selbst festzulegen. Wie könnten Sie das ermöglichen?

Eine Option wäre sicher, der Klasse eine Setter-Methode (setSize()) hinzuzufügen. Aber dadurch hat Duck möglicherweise kurzzeitig keine Größe* und zwingt den Benutzer von Duck, zwei Anweisungen zu schreiben – eine für die Erstellung der Ente (des Duck-Objekts) und eine für den Aufruf von setSize(). Der unten stehende Code setzt eine Setter-Methode ein, um die Startgröße der neuen Ente festzulegen.

```
public class Duck {
  int size;                          ← Instanzvariable

  public Duck() {
    System.out.println("Quack");     ← Konstruktor
  }

  public void setSize(int newSize) {
    size = newSize;                  ← Setter-Methode
  }
}
```

```
public class UseADuck {

  public static void main(String[] args) {
    Duck d = new Duck();

    d.setSize(42);   ← Das ist nicht gut! Die Ente (das Duck-Objekt) ist an diesem Punkt schon lebendig, hat aber noch keine Größe (size)!* Und dann verlassen Sie sich auch noch darauf, dass der Benutzer WEISS, dass für die Duck-Erstellung zwei Schritte nötig sind: erstens der Aufruf des Konstruktors und zweitens der Aufruf des Setters.
  }
}
```

* Instanzvariablen haben einen Standardwert: 0 oder 0.0 für numerische Elementartypen, false für den Typ boolean und null für Referenzen.

### Es gibt keine Dummen Fragen

**F: Warum muss man einen Konstruktor schreiben, wenn der Compiler das für uns erledigen kann?**

A: Wenn Sie Code brauchen, der Ihnen bei der Initialisierung Ihres Objekts hilft und für die Benutzung vorbereitet, müssen Sie Ihren eigenen Konstruktor schreiben. Vielleicht benötigen Sie bestimmte Benutzereingaben, bevor das Objekt fertiggestellt werden kann. Es kann aber auch sein, dass Sie einen eigenen Konstruktor brauchen, ohne hierfür eigenen Code schreiben zu müssen. Dieses Szenario hat mit dem Superklassenkonstruktor zu tun, auf den wir bald zu sprechen kommen.

**F: Wie kann man einen Konstruktor von einer Methode unterscheiden? Kann man auch eine Methode haben, die den gleichen Namen wie die Klasse hat?**

A: In Java ist es tatsächlich möglich, dass eine Methode den gleichen Namen wie die Klasse hat. Das macht die Methode aber nicht zu einem Konstruktor. Der Unterschied zwischen Methode und Konstruktor ist der Rückgabetyp. Methoden *müssen* einen Rückgabetyp haben, während Konstruktoren *keinen* Rückgabetyp *haben dürfen*.

```
public Duck() { }          Konstruktor

public void Duck() { }     Methode
           ↑ Rückgabetyp
```

Aber selbst wenn der Compiler diese Methoden erlaubt, **tun Sie das nicht!** Es entspricht nicht den Namenskonventionen (Methodennamen beginnen mit einem Kleinbuchstaben). Davon abgesehen ist es einfach super verwirrend.

**F: Werden Konstruktoren vererbt? Wenn Sie keinen Konstruktor bereitstellen, die Superklasse aber schon, erhalten Sie dann den Konstruktor der Superklasse anstelle des Compiler-Standards?**

A: Nein. Konstruktoren werden nicht vererbt. Aber das werden wir uns ein paar Seiten weiter ganz genau ansehen.

# Den Konstruktor benutzen, um wichtige Zustände von Duck zu initialisieren*


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

Der beste Ort für Initialisierungscode ist der Konstruktor. Dazu müssen Sie nur einen Konstruktor erstellen, der Argumente übernehmen kann.

Dem Duck-Konstruktor einen int-Parameter hinzufügen.

```
public class Duck {
  int size;

  public Duck(int duckSize) {
    System.out.println("Quack");

    size = duckSize;

    System.out.println("size is " + size);
  }
}
```

Den Argumentwert benutzen, um der Instanzvariablen size einen Wert zuzuweisen. Wir hätten hier auch die Methode setSize() aufrufen können.

```
public class UseADuck {

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

Einen Wert an den Konstruktor übergeben.

Diesmal haben wir nur eine Anweisung. Wir erstellen eine neue Ente (Duck-Objekt) und legen ihre Größe (size) in einer Anweisung fest.

```
Datei Bearbeiten Fenster Hilfe Schnatter
% java UseADuck
Quack
size is 42
```

* Was nicht heißen soll, dass nicht alle Duck-Zustände wichtig sind.

# Die Erstellung einer Ente erleichtern

## Stellen Sie sicher, dass Sie einen Konstruktor haben, der keine Argumente erwartet

Was passiert, wenn der Duck-Konstruktor ein Argument erwartet? Überlegen Sie mal. Auf der vorigen Seite gibt es nur *einen* Duck-Konstruktor, der ein int-Argument für den Wert von *size* des Duck-Objekts übernimmt. Das muss kein großes Problem sein, aber es kann die Erstellung eines Duck-Objekts erschweren, besonders wenn der Programmierer nicht *weiß*, welche Größe (size) die Ente haben soll. Wäre es nicht hilfreich, die Duck-Objekte mit einem Standardwert für size zu versehen? Auf diese Weise könnten Nutzer, denen keine passende Größe einfällt, trotzdem eine Ente erstellen, die funktioniert.

***Nehmen wir mal an, Sie wollten, dass Duck-Benutzer ZWEI Optionen für die Erstellung einer Ente haben sollen – eine, bei der Sie einen Wert für size (als Konstruktorargument) angeben, und eine, bei der die Angabe nicht nötig ist und stattdessen ein Standardwert für size verwendet wird.***

Das lässt sich sauber mit nur einem Konstruktor erledigen. Wie Sie wissen, *müssen* Sie beim Aufruf einer Methode oder eines Konstruktors ein passendes Argument übergeben, wenn die Methode (oder der Konstruktor – es gelten die gleichen Regeln) einen Parameter hat. Sie können nicht einfach sagen: »Wenn jemand dem Konstruktor nichts übergibt, dann benutze den Standardwert für size«, weil der Code erst gar nicht kompiliert würde, wenn Sie beim Konstruktoraufruf kein int-Argument übergeben. Sie *könnten* etwas Plumpes tun wie das hier:

```
public class Duck {
  int size;

  public Duck(int newSize) {
    if (newSize == 0) {
      size = 27;
    } else {
      size = newSize;
    }
  }
}
```

Ist der Parameterwert null, gib dem neuen Duck-Objekt den Standardwert für size (27), ansonsten verwende den tatsächlich übergebenen Parameterwert (newSize) als Wert für size. KEINE sehr gute Lösung.

Das heißt aber auch, dass der Programmierer, der das neue Duck-Objekt erstellt, *wissen muss*, dass die Übergabe von »0« das Protokoll für die Verwendung des Standardwerts für size ist. Ziemlich hässlich. Und was, wenn ein anderer Programmierer das nicht weiß? Oder wenn wir tatsächlich eine Ente mit der Größe null brauchen? (Wir nehmen an, dass Enten der Größe null erlaubt sind. Falls nicht, versehen Sie den Konstruktor mit dem entsprechenden Validierungscode, um das zu verhindern.) Der springende Punkt ist, dass eventuell nicht immer zwischen einem echten »Ich möchte 0 als Wert für size benutzen«-Konstruktorargument und einem »Ich schicke dir eine 0, damit ich die Standardgröße erhalte, wie auch immer sie lautet«-Konstruktorargument unterschieden werden kann.

**Eigentlich brauchen Sie ZWEI Möglichkeiten, ein neues Duck-Objekt zu erstellen:**

```
public class Duck2 {
  int size;

  public Duck2() {
    // Standardwert verwenden:
    size = 27;
  }

  public Duck2(int duckSize) {
    // duckSize verwenden
    size = duckSize;
  }
}
```

**Ein Duck-Objekt erstellen, für das die Größe (size) bekannt ist:**

```
Duck2 d = new Duck2(15);
```

**Ein Duck-Objekt erstellen, für das die Größe (size) *nicht* bekannt ist:**

```
Duck2 d2 = new Duck2();
```

Für diese zwei Möglichkeiten, ein Duck-Objekt zu erstellen, brauchen wir also *zwei* Konstruktoren. Einer übernimmt einen int, der andere nicht. ***Wenn Ihre Klasse mehr als einen Konstruktor enthält, heißt das, Sie haben überladene Konstruktoren.***

## Erstellt der Compiler nicht immer einen argumentlosen Konstruktor für Sie? Nein!

Wenn Sie *nur* einen Konstruktor mit Argumenten schreiben, könnten Sie denken, dass der Compiler erkennt, dass Sie keinen argumentlosen Konstruktor haben, und diesen für Sie anlegt. Aber so funktioniert das nicht. Der Compiler mischt sich in die Erstellung des Konstruktors *nur ein, wenn Sie den Konstruktor überhaupt nicht erwähnen.*


**Schreiben Sie dagegen einen Konstruktor, der ein Argument erwartet, und zusätzlich noch einen argumentlosen Konstruktor benötigen, müssen Sie diesen ebenfalls selbst schreiben!**

Sobald Sie EINEN Konstruktor bereitstellen, hält sich der Compiler raus und sagt: »Na gut, alles klar. Sieht aus, als hätten Sie beim Konstruktor jetzt den Hut auf.«

**Enthält eine Klasse mehr als einen Konstruktor, MÜSSEN die Konstruktoren verschiedene Argumentlisten haben.**

Die Argumentliste enthält die Reihenfolge und Typen der Argumente. Solange die Listen sich unterscheiden, können Sie mehr als einen Konstruktor verwenden. Das funktioniert auch mit Methoden, aber darum kümmern wir uns in einem anderen Kapitel.

## Überladene Konstruktoren heißt, Ihre Klasse kann mehr als einen Konstruktor enthalten.

### Damit das kompiliert wird, muss jeder Konstruktor eine *andere* Argumentliste haben.

Die unten stehende Klasse ist gültig, weil alle fünf Konstruktoren verschiedene Argumentlisten haben. Würden zwei Konstruktoren jeweils nur ein int erwarten, würde die Klasse nicht kompiliert. Wie Sie die Parametervariablen nennen, spielt dabei keine Rolle. Wichtig sind der Variablen*typ* (int, Dog etc.) und die *Reihenfolge*. Sie *können* zwei Konstruktoren mit identischen Typen haben, ***solange die Reihenfolge sich unterscheidet***. Ein Konstruktor, der einen String gefolgt von einem int übernimmt, ist *nicht* das Gleiche wie einer, der einen int gefolgt von einem String übernimmt.

Fünf verschiedene Konstruktoren bedeuten fünf unterschiedliche Wege, einen neuen Pilz (Mushroom) zu erstellen.

```
public class Mushroom {

  public Mushroom(int size) { }

  public Mushroom( ) { }

  public Mushroom(boolean isMagic) { }

  public Mushroom(boolean isMagic, int size) { }

  public Mushroom(int size, boolean isMagic) { }

}
```

Wenn Sie die Größe (size) kennen, aber nicht wissen, ob der Pilz magisch (isMagic) ist.

Wenn Sie überhaupt nichts wissen.

Wenn Sie wissen, dass der Pilz magisch ist, aber die Größe nicht kennen.

Wenn Sie wissen, dass der Pilz magisch ist (isMagic), UND Sie auch die Größe (size) kennen.

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

Woher soll der Compiler wissen, dass es sich um unterschiedliche Dinge handelt, wenn die Argumente den gleichen Typ haben?

### PUNKT FÜR PUNKT

- Instanzvariablen leben innerhalb des Objekts, zu dem sie gehören, auf dem Heap.
- Ist die Instanzvariable eine Referenz auf ein Objekt, befinden sich die Referenz und das Objekt, auf das sie sich bezieht, auf dem Heap.
- Ein Konstruktor ist der Code, der ausgeführt wird, wenn Sie **new** auf einem Klassentyp aufrufen.
- Ein Konstruktor muss den gleichen Namen haben wie die Klasse und darf *keinen* Rückgabetyp besitzen.
- Sie können einen Konstruktor benutzen, um den Zustand (die Instanzvariablen) des zu konstruierenden Objekts zu initialisieren.
- Wenn Sie Ihre Klasse nicht mit einem Konstruktor versehen, fügt der Compiler einen Standardkonstruktor für Sie ein.
- Der Standardkonstruktor ist grundsätzlich argumentlos.
- Wenn Sie Ihre Klasse selbst mit einem Konstruktor versehen, erstellt der Compiler keinen Standardkonstruktor.
- Wenn Sie einen argumentlosen Konstruktor brauchen und die Klasse bereits einen Konstruktor mit Argumenten enthält, müssen Sie den argumentlosen Konstruktor selbst erstellen.
- Stellen Sie nach Möglichkeit immer einen argumentlosen Konstruktor zur Verfügung. So erleichtern Sie Programmierern die Erstellung eines funktionierenden Objekts. Geben Sie Standardwerte an.
- Überladene Konstruktoren bedeuten, dass Ihre Klasse mehr als einen Konstruktor enthält.
- Überladene Konstruktoren müssen unterschiedliche Argumentlisten haben.
- Sie können keine zwei Konstruktoren mit gleichen Argumentlisten haben. Eine Argumentliste enthält die Reihenfolge und die Typen der Argumente.
- Instanzvariablen wird ein Standardwert zugewiesen, auch wenn Sie das nicht explizit tun. Die Standardwerte sind 0/0.0/false für Elementartypen und null für Referenzen.

## Spitzen Sie Ihren Bleistift → Das schaffen Sie selbst.

Verbinden Sie die Aufrufe von **new Duck()** mit dem Konstruktor, der aufgerufen wird, wenn das Duck-Objekt instanziiert wird. Um Ihnen auf die Sprünge zu helfen, haben wir den einfachen Fall schon für Sie erledigt.

```
public class TestDuck {

  public static void main(String[] args) {
    int weight = 8;
    float density = 2.3F;
    String name = "Donald";
    long[] feathers = {1, 2, 3, 4, 5, 6};
    boolean canFly = true;
    int airspeed = 22;

    Duck[] d = new Duck[7];

    d[0] = new Duck();

    d[1] = new Duck(density, weight);

    d[2] = new Duck(name, feathers);

    d[3] = new Duck(canFly);

    d[4] = new Duck(3.3F, airspeed);

    d[5] = new Duck(false);

    d[6] = new Duck(airspeed, density);
  }
}
```

```
class Duck {
  private int kilos = 6;
  private float floatability = 2.1F;
  private String name = "Generic";
  private long[] feathers = {1, 2, 3,
                             4, 5, 6, 7};
  private boolean canFly = true;
  private int maxSpeed = 25;

  public Duck() {
    System.out.println("type 1 duck");
  }

  public Duck(boolean fly) {
    canFly = fly;
    System.out.println("type 2 duck");
  }

  public Duck(String n, long[] f) {
    name = n;
    feathers = f;
    System.out.println("type 3 duck");
  }

  public Duck(int w, float f) {
    kilos = w;
    floatability = f;
    System.out.println("type 4 duck");
  }

  public Duck(float density, int max) {
    floatability = density;
    maxSpeed = max;
    System.out.println("type 5 duck");
  }
}
```

## Es gibt keine Dummen Fragen

**F: Vorhin haben Sie gesagt, dass es sinnvoll ist, einen argumentlosen Konstruktor bereitzustellen, damit wir bei seinem Aufruf Standardwerte für die »fehlenden« Argumente festlegen können. Aber kann es nicht auch sein, dass einem einfach keine Standardwerte einfallen? Gibt es Fälle, in denen eine Klasse *keinen* argumentlosen Konstruktor enthalten sollte?**

A: In manchen Fällen hat die Verwendung eines argumentlosen Konstruktors keinen Sinn. Beispiele finden sich etwa in der Java-API – einige Klassen besitzen keinen argumentlosen Konstruktor. Die Klasse Color repräsentiert beispielsweise eine Farbe. Color-Objekte können eingesetzt werden, um die Farbe einer Bildschirmschrift oder eines GUI-Buttons festzulegen oder zu ändern. Wenn Sie eine Color-Instanz erstellen, steht diese für eine bestimmte Farbe (zum Beispiel Neidgrün, Blue-Screen-of-Death-Blau, Schamrot etc.).
Wenn Sie ein Color-Objekt erstellen, müssen Sie grundsätzlich einen Farbwert angeben.

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

(Wir verwenden hier drei ints, um einen RGB-Wert auszudrücken. In Kapitel 15 werden wir näher auf Color eingehen.) Was würden Sie sonst bekommen? Die Java-API-Programmierer *hätten* entscheiden können, dass Sie beim Aufruf eines argumentlosen Color-Konstruktors standardmäßig einen schönen Malvenfarbton erhalten. Aber der gute Geschmack hat gesiegt.
Wenn Sie versuchen, eine Farbe (ein Color-Objekt) ohne Argument zu erstellen:

```
Color c = new Color();
```

dann bekommt der Compiler einen Anfall, weil er keinen passenden argumentlosen Konstruktor in der Color-Klasse finden kann.

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

# Kurzer Rückblick. Vier Dinge, die Sie sich zu Konstruktoren merken sollten!

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

```
Duck d = new Duck();
```

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

```
public Duck(int size) { }
```

3. Wenn Sie selbst keinen Konstruktor für Ihre Klasse anlegen, fügt der Compiler einen Standardkonstruktor hinzu. Der Standardkonstruktor ist grundsätzlich argumentlos.

```
public Duck() { }
```

4. Ihre Klasse kann mehr als einen Konstruktor enthalten, solange sich die Argumentlisten unterscheiden. Mehr als ein Konstruktor in derselben Klasse bedeutet, dass Sie überladene Konstruktoren haben.

```
public Duck() { }

public Duck(int size) { }

public Duck(String name) { }

public Duck(String name, int size) { }
```

* Man hat herausgefunden, dass die Kopfnuss-Übungen die Neuronengröße um 42 % steigern kann – wenn Sie alle Übungen machen. Und Sie kennen den Spruch: »Dicke Neuronen ...«

## KOPF-NUSS

**Was ist mit Superklassen?**

**Sollte bei der Erstellung eines Dog-Objekts nicht auch der Canine-Konstruktor ausgeführt werden?**

**Sollten abstrakte Superklassen überhaupt einen Konstruktor haben?**

Mit diesen Fragen werden wir uns auf den folgenden Seiten beschäftigen. Halten Sie also inne und denken Sie über die Auswirkungen von Konstruktoren und Superklassen nach.*

## Es gibt keine Dummen Fragen

**F: Müssen Konstruktoren öffentlich (public) sein?**

A: Nein. Konstruktoren können als **`public`**, **`protected`**, **`private`** oder *default* (das heißt ganz ohne Zugriffsmodifier) sein. Weitere Informationen zum *default*-Zugriff finden Sie in Anhang B.

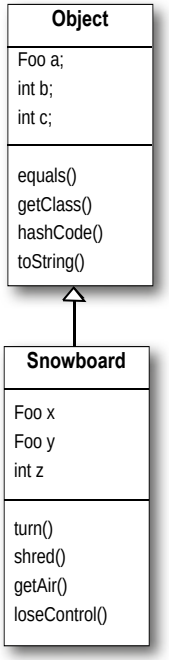
**F: Wie kann ein als private markierter Konstruktor überhaupt nützlich sein? Niemand kann ihn jemals aufrufen, also kann auch niemand jemals ein Objekt erstellen!**

A: Das ist nicht ganz richtig. Die Kennzeichnung als *private* bedeutet nicht, dass *niemand* darauf zugreifen kann. Es heißt nur, dass *niemand außerhalb der Klasse* darauf zugreifen kann. Das kommt Ihnen sicher wie eine Zwickmühle vor. Nur Code aus der Klasse mit dem privaten Konstruktor kann ein Objekt dieser Klasse erstellen. Aber wie wollen Sie jemals Code aus der Klasse ausführen, ohne vorher ein Objekt zu erstellen? Wie wollen Sie jemals an etwas in der Klasse herankommen? *Geduld, kleiner Grashüpfer*. Darum kümmern wir uns im folgenden Kapitel.

# Moment! Wir haben noch GAR NICHT über Superklassen und Vererbung gesprochen und wie das alles mit Konstruktoren zusammenpasst.

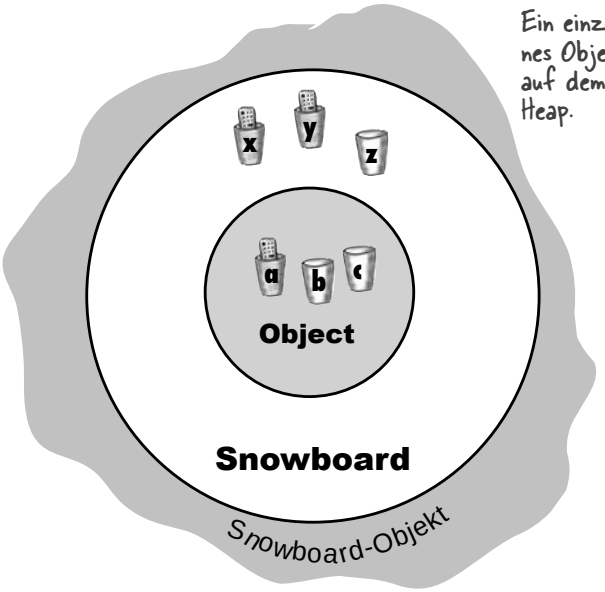
Und hier fängt der Spaß an. Im vorigen Kapitel haben wir uns angesehen, wie das Snowboard-Objekt einen inneren Kern umgibt, der für den Object-Teil der Snowboard-Klasse steht. Der springende Punkt hierbei war, dass jedes Objekt nicht nur seine *selbst* deklarierten Instanzvariablen enthält, sondern auch *alles aus seinen Superklassen* (was mindestens Object bedeutet, weil jede Klasse Object erweitert).

Wenn ein Objekt erstellt wird (weil jemand **new** gesagt hat; es gibt ***keine andere Möglichkeit***, ein Objekt zu erstellen, es sei denn, jemand ruft irgendwo **new** auf dem Klassentyp auf), dann erhält das Objekt Platz für *alle* Instanzvariablen – den gesamten Vererbungsbaum hinauf. Denken Sie einen Moment darüber nach. Eine Superklasse könnte Setter-Methoden haben, die eine private Variable verkapseln. Aber auch diese Variable muss *irgendwo* leben. Wenn ein Objekt erstellt wird, ist es fast so, als würden *mehrerer* Objekte erscheinen – das per **new** erstellte Objekt und ein Objekt für jede seiner Superklassen. Konzeptuell ist es aber deutlich sinnvoller, sich das vorzustellen, als besäße das erstellte Objekt verschiedene *Schichten*, die für die darüberliegenden Klassen stehen, wie unten im Bild dargestellt.




Object besitzt eine Instanzvariable, die von Zugriffsmethoden verkapselt wird. Diese Instanzvariablen werden erzeugt, wenn eine beliebige Unterklasse instanziiert wird. (Dies sind nicht die ECHTEN Objektvariablen, was uns hier aber egal ist, weil sie ohnehin gekapselt sind.)

Snowboard hat außerdem eigene Instanzvariablen. Um ein Snowboard-Objekt zu erstellen, brauchen wir also Platz für die Instanzvariablen beider Klassen.




Ein einzelnes Objekt auf dem Heap.

Auf dem Heap befindet sich nur EIN Objekt. Ein Snowboard-Objekt. Aber das enthält sowohl die eigenen Snowboard-Anteile als auch seine Object-Teile. Alle Instanzvariablen beider Klassen müssen hier vorhanden sein.

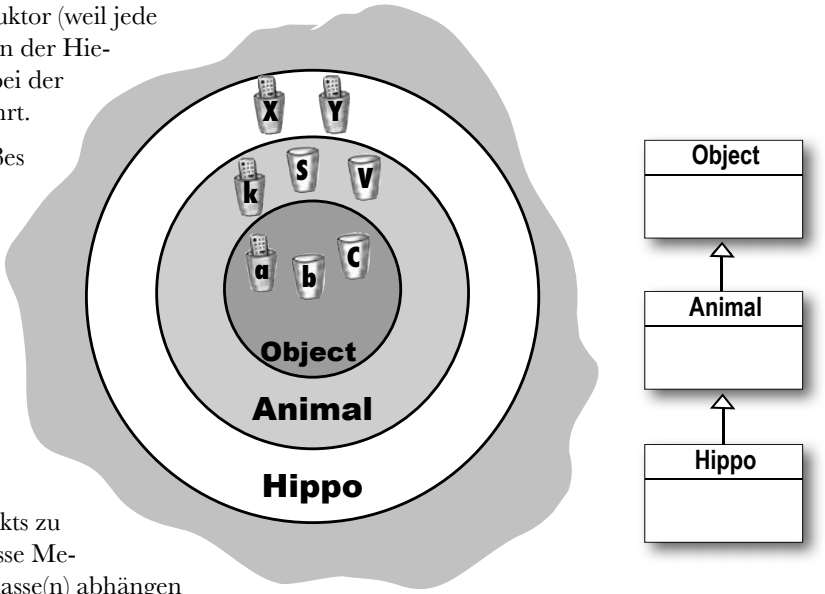
# Die Rolle der Superklassen-konstruktoren im Leben eines Objekts

***Um ein neues Objekt zu erstellen, müssen alle Konstruktoren im Vererbungsbaum des Objekts ausgeführt werden.***

Lassen Sie sich das mal auf der Zunge zergehen.

Das heißt, jede Superklasse hat einen Konstruktor (weil jede Klasse einen Konstruktor besitzt), und jeder in der Hierarchie oberhalb liegende Konstruktor wird bei der Erstellung eines Unterklassenobjekts ausgeführt.

Die Anweisung **new** ist also ein ziemlich großes Ding. Sie startet die ganze Konstruktor-Kettenreaktion. Ja, und selbst abstrakte Klassen besitzen Konstruktoren. Zwar können Sie auf abstrakten Klassen niemals **new** aufrufen, aber auch eine abstrakte Klasse ist immer noch eine Superklasse. Das heißt, ihr Konstruktor wird ausgeführt, sobald jemand eine Instanz einer konkreten Unterklasse erstellt.




Ein einzelnes Hippo-Objekt auf dem Heap.

**Ein neues Hippo-Objekt IST auch EIN Animal und IST EIN Object. Wollen Sie ein Hippo erstellen, müssen Sie auch die Animal- und Object-Anteile von Hippo erstellen.**

**Das alles passiert in einem Prozess, der als Konstruktorverkettung (Constructor Chaining) bezeichnet wird.**

Die Superklassenkonstruktoren werden ausgeführt, um die Superklassenanteile des Objekts zu erstellen. Wie Sie wissen, kann eine Unterklasse Methoden erben, die vom Zustand ihrer Superklasse(n) abhängen (soll heißen, die Werte der Instanzvariablen der Superklasse). Damit ein Objekt vollständig erstellt werden kann, müssen alle Superklassenanteile seiner selbst auch vollständig erstellt werden, und das ist der Grund, warum der Superklassenkonstruktor ausgeführt werden *muss*. Alle Instanzvariablen aller Klassen im Vererbungsbaum müssen deklariert und initialisiert sein. Selbst wenn Animal Instanzvariablen besitzt, die Hippo nicht erbt (etwa wenn die Variablen privat sind), hängt Hippo von den Animal-Methoden ab, die diese Variablen *benutzen*.

Wird ein Konstruktor ausgeführt, ruft er sofort seinen Superklassenkonstruktor auf, und zwar die ganze Kette entlang bis hin zum Konstruktor von Object.

Auf den folgenden Seiten werden Sie lernen, wie Superklassenkonstruktoren aufgerufen werden und wie Sie diese Aufrufe selbst durchführen können. Sie werden außerdem erfahren, was zu tun ist, wenn der Superklassenkonstruktor Argumente erwartet!

# Ein Hippo zu erstellen, heißt auch, die Animal- und Object-Teile zu erstellen ...

```
public class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
  public Hippo() {
    System.out.println("Making a Hippo");
  }
}
```

```
public class TestHippo {
  public static void main(String[] args) {
    System.out.println("Starting...");
    Hippo h = new Hippo();
  }
}
```

Anhand der Hierarchie im oben gezeigten Code können wir den Prozess der Erzeugung eines neuen Hippo-Objekts Schritt für Schritt nachvollziehen.

**Spitzen Sie Ihren Bleistift**

Wie lautet die richtige Ausgabe? Finden Sie anhand des Codes auf der linken Seite heraus, welche Ausgaben bei der Ausführung von TestHippo erzeugt werden. A oder B?

(Die Antwort finden Sie unten auf dieser Seite.)

A

```
Datei Bearbeiten Fenster Hilfe Fluchen
% java TestHippo
Starting...
Making an Animal
Making a Hippo
```

B

```
Datei Bearbeiten Fenster Hilfe Fluchen
% java TestHippo
Starting...
Making a Hippo
Making an Animal
```

1. Der Code einer anderen Klasse ruft **`new Hippo()`** auf, und der **Hippo()**-Konstruktor wird in den Stack-Frame an der Spitze des Stacks eingefügt.


2. **Hippo()** ruft den Superklassenkonstruktor auf, wodurch der **Animal()**-Konstruktor an die Spitze des Stacks geschoben wird.


3. **Animal()** ruft den Superklassenkonstruktor auf, was den **Object()**-Konstruktor an die Spitze des Stacks setzt, weil Object die Superklasse von Animal ist.


4. **Object()** wird beendet, und sein Stack-Frame wird vom Stack entfernt. Die Ausführung kehrt zum **Animal()**-Konstruktor zurück und macht auf der Zeile weiter, die auf den Aufruf von **Animal()** an seinen Superklassenkonstruktor folgt.


Die erste Ausgabe (A) ist richtig. Der Hippo()-Konstruktor wird zuerst aufgerufen, aber es ist der Animal()-Konstruktor, der zuerst fertig ist.

# *Wie* ruft man einen Superklassenkonstruktor auf?

Sie könnten denken, dass Sie irgendwo im Duck-Konstruktor Animal() aufrufen, wenn Duck Animal erweitert. Aber so funktioniert das nicht:

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    Animal();
    size = newSize;
  }
}
```

SCHLECHT! → `Animal();` ← NEIN! Das ist ungültig.

Die einzige Möglichkeit, einen Superklassenkonstruktor aufzurufen, besteht in der Verwendung von ***super()***. Richtig gelesen – ***super()*** ruft den ***Superklassenkonstruktor*** auf.

Wer hätte das gedacht?

```
public class Duck extends Animal {
  int size;

  public Duck(int newSize) {
    super();
    size = newSize;
  }
}
```

`super();` ← Rufen Sie einfach super() auf.

Ein Aufruf von super() schiebt den Superklassenkonstruktor an die Spitze des Stacks. Und was, glauben Sie, tut dieser Superklassenkonstruktor? *Er ruft seinen Superklassenkonstruktor auf*. Und so geht das weiter, bis sich der Object-Konstruktor an oberster Stelle des Stacks befindet. Sobald Object() fertig ist, wird er wieder vom Stack entfernt, und das nächste Ding auf dem Stack (der Unterklassenkonstruktor, der Object() aufgerufen hat) befindet sich jetzt an der Spitze des Stacks. *Auch dieser* Konstruktor wird beendet, und so geht es weiter, bis sich der ursprüngliche Konstruktor an der Spitze des Stacks befindet, wo *dieser* schließlich beendet wird.

## Und wie haben wir das vorher ohne den Aufruf von super() geschafft?

Das kriegen Sie bestimmt heraus.

### Unser guter Freund, der Compiler, fügt einen Aufruf von super() ein, wenn Sie das nicht selbst tun.

Der Compiler ist also auf *zwei* Arten an der Konstruktion beteiligt:

1. **Wenn Sie *keinen* Konstruktor bereitstellen,**

fügt der Compiler einen für Sie ein, wie hier:

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

2. ***Wenn* Sie einen Konstruktor bereitstellen, aber keinen Aufruf von super() einfügen,**

fügt der Compiler einen Aufruf an super() in jeden Ihrer überladenen Konstruktoren ein.*

```
super();
```

So sieht das immer aus. Der vom Compiler eingefügte Aufruf von super() ist immer argumentlos. Hat die Superklasse überladene Konstruktoren, wird nur der argumentlose Konstruktor aufgerufen.

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

# Kann ein Kind vor seinen Eltern existieren?


Um herauszufinden, wer zuerst existieren muss, stellen Sie sich Superklassen am besten als Eltern und Unterklassen als Kinder vor. ***Der Superklassenteil eines Objekts muss vollständig ausgeformt (komplett aufgebaut) sein, bevor der Unterklassenteil konstruiert werden kann.***

Vergessen Sie nicht, dass ein Objekt von Dingen abhängen kann, die es von seiner Superklasse erbt. Daher ist es wichtig, dass diese ererbten Dinge fertiggestellt sind. Daran führt kein Weg vorbei. Der Superklassenkonstruktor muss vor seinem Unterklassenkonstruktor beendet werden.

Wenn Sie sich noch einmal die Abfolge von Stacks auf Seite 254 ansehen, können Sie erkennen, dass der Hippo-Konstruktor zwar *zuerst* aufgerufen wird (er ist das erste Ding auf dem Stack), aber das *letzte*, das beendet wird! Jeder Unterklassenkonstruktor ruft seinen eigenen Superklassenkonstruktor auf, bis sich die Konstruktion von Object an der Spitze des Stacks befindet. Wenn der Object-Konstruktor fertig ist, springen wir im Stack zurück zum Animal-Konstruktor. Und erst wenn der Animal-Konstruktor mit seiner Arbeit fertig ist, kommen wir wieder zurück, um den Rest des Hippo-Konstruktors abzuarbeiten. Daher gilt:

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

Mögliche Konstruktoren für die Klasse Boop

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

Diese Konstruktoren sind okay, weil der Programmierer die Aufrufe von super() ausdrücklich als erste Anweisung geschrieben hat.

☑
```
public Boop(int i) {
    super();
    size = i;
}
```

☑
```
public Boop() {
}
```

Diese Konstruktoren sind okay, weil der Compiler den Aufruf von super() als erste Anweisung einfügen wird.

☑
```
public Boop(int i) {
    size = i;
}
```

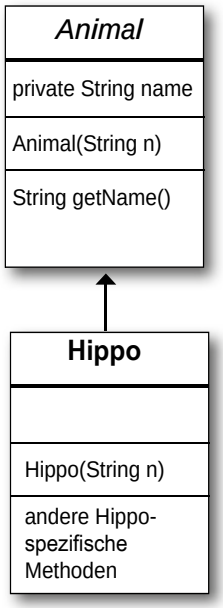
⊘
```
public Boop(int i) {
    size = i;
    super();
}
```

SCHLECHT!! Das wird nicht kompiliert! Sie können den Aufruf von super() nicht ausdrücklich unter eine andere Anweisung schreiben.

* Zu dieser Regel gibt es eine Ausnahme, die Sie auf Seite 258 kennenlernen werden.

# Superklassenkonstruktoren mit Argumenten

Was ist, wenn der Superklassenkonstruktor Argumente erwartet? Können Sie etwas an den super()-Aufruf übergeben? Aber sicher. Wäre das nicht möglich, könnten Klassen ohne argumentlose Konstruktoren gar nicht erweitert werden. Stellen Sie sich beispielsweise vor, alle Tiere (Animals) hätten einen Namen. Die Klasse Animal besitzt eine getName()-Methode, die den Wert der Instanzvariablen name zurückgibt. Die Instanzvariable ist als private markiert, aber die Unterklasse (in diesem Fall Hippo) erbt die getName()-Methode. Und damit haben wir ein Problem: Hippo hat (durch Vererbung) eine getName()-Methode, aber die Instanzvariable name fehlt. Hippo muss sich auf den Animal-Teil seiner selbst verlassen, um die Instanzvariable name zu verwalten und ihren Wert zurückzugeben, wenn jemand getName() auf einem Hippo-Objekt aufruft. Aber ... wie erhält der Animal-Teil den Namen? Die einzige Referenz, die Hippo auf seinen Animal-Teil hat, ist über super(). Über diesen Weg kann Hippo seinen Namen also an seinen Animal-Teil weitergeben, damit dieser ihn in seiner privaten Instanzvariablen name speichern kann.




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

  public String getName() {
    return name;
  }

  public Animal(String theName) {
    name = theName;
  }
}
```

Alle Animal-Objekte (inklusive der Unterklassen) haben einen Namen.

Eine Getter-Methode, die Hippo erbt.

Der Konstruktor, der den Namen übernimmt und ihn der Instanzvariablen name zuweist.

```
public class Hippo extends Animal {
  public Hippo(String name) {
    super(name);
  }
}
```

Der Hippo-Konstruktor übernimmt den Namen.

Der Name wird im Stack aufwärts an den Animal-Konstruktor weitergereicht.

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

Ein Hippo erstellen und den Namen »Buffy« an den Hippo-Konstruktor übergeben. Danach Hippos geerbte getName()-Methode aufrufen.


```
Datei Bearbeiten Fenster Hilfe Verstecken
%java MakeHippo
Buffy
```

# Einen überladenen Konstruktor aus einem anderen aufrufen

Was können Sie tun, wenn Sie überladene Konstruktoren brauchen, die aber, abgesehen davon, dass sie unterschiedliche Argumenttypen entgegennehmen, alle das Gleiche tun? *Duplizierten* Code in den Konstruktoren wollen Sie natürlich vermeiden (schwer zu warten etc.), also würden Sie am liebsten den Großteil des Konstruktorcodes (inklusive des Aufrufs von super()) in nur *einen* der überladenen Konstruktoren auslagern. Dann soll der jeweils zuerst aufgerufene Konstruktor Den Einzig Wahren Konstruktor aufrufen, der dann die Konstruktion zu Ende bringt. Das ist leicht. Sagen Sie einfach *this()*. Oder *this(einString)*. Oder *this(27, x)*. Anders gesagt: Stellen Sie sich das Schlüsselwort *this* einfach als Referenz **auf das aktuelle Objekt** vor.

Sie können *this()* nur innerhalb eines Konstruktors verwenden. Außerdem muss es die erste Anweisung im Konstruktor sein!

Aber das ist ein Problem, oder? Schließlich haben wir zuvor gesagt, dass super() die erste Anweisung im Konstruktor sein muss. Und das heißt, Sie müssen sich entscheiden.

**Verwenden 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 im Konstruktor sein.**

**Ein Konstruktor kann einen Aufruf von super() ODER einen von this() enthalten, aber niemals beide!**

### Jeder Konstruktor kann einen Aufruf von super() oder einen von this() haben, aber niemals beide!

Die Auswahl hängt davon ab, welche Werte Sie haben, welche Sie setzen müssen und welche Konstruktoren in dieser Klasse oder der Superklasse zur Verfügung gestellt werden.

```
import java.awt.Color;

class Mini extends Car {
  private Color color;

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

  public Mini(Color c) {
    super("Mini");
    color = c;
    // mehr Initialisierung
  }

  public Mini(int size) {
    this(Color.RED);
    super(size);
  }
}
```

Der argumentlose Konstruktor legt eine Standardfarbe (Color) fest und ruft den überladenen Einzig Wahren Konstruktor auf (den, der super() aufruft).

Dies ist Der Einzig Wahre Konstruktor, der in Wirklichkeit die Initialisierung des Objekts durchführt (inklusive des Aufrufs von super()).

Das wird nicht funktionieren! super() und this() können nicht im gleichen Konstruktor verwendet werden, weil beide die erste Anweisung 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

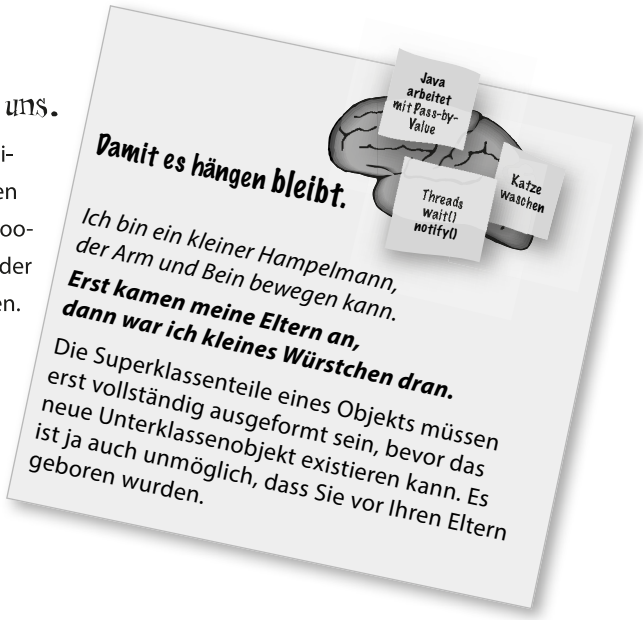
→ Das schaffen Sie ohne uns.

Einige Konstruktoren in der Klasse SonOfBoo können nicht kompiliert werden. Versuchen Sie, herauszufinden, welche Konstruktoren ungültig sind. Verbinden Sie die Compiler-Fehler mit den SonOfBoo-Konstruktoren, die sie ausgelöst haben, indem Sie eine Linie von der Fehlermeldung zum jeweiligen »schlechten« Konstruktor zeichnen.

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

---

```
class SonOfBoo extends Boo {
  public SonOfBoo() {
    super("boo");
  }

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

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

  public SonOfBoo(int i, String s) {
  }

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

  public SonOfBoo(int i, int j) {
    super("man", j);
  }

  public SonOfBoo(int i, int x, int y) {
    super(i, "star");
  }
}
```




```
Datei Bearbeiten Fenster Hilfe Rumpel
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.
lang.String)
```

```
Datei Bearbeiten Fenster Hilfe Pumpel
%javac SonOfBoo.java
cannot resolve symbol
symbol  : constructor Boo
(int,java.lang.String)
```

```
Datei Bearbeiten Fenster Hilfe KrachStraße
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

# Jetzt wissen wir, wie ein Objekt geboren wird. Aber wie lange lebt ein Objekt?

Das Leben eines *Objekts* hängt komplett vom Leben der Referenzen ab, die auf es verweisen. Solange die Referenz als »lebendig« gilt, lebt auch das Objekt auf dem Heap noch. Stirbt die Referenz (was das bedeutet, werden wir gleich sehen), muss auch das Objekt dran glauben.

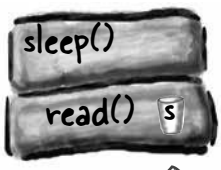
### Wenn die Lebensdauer von Objekten vom Leben ihrer Referenzen abhängt, wie lange lebt dann eine *Variable*?

Das kommt darauf an, ob es eine *lokale* oder eine *Instanz*variable ist. Der unten stehende Code zeigt das Leben einer lokalen Variablen. In diesem Beispiel hat sie einen elementaren Typ. Dabei spielt es für die Lebensdauer keine Rolle, ob es sich um eine elementare oder eine Referenzvariable handelt.

```
public class TestLifeOne {

  public void read() {
    int s = 42;
    sleep();
  }

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

Der Geltungsbereich von »s« ist auf die read()-Methode beschränkt. Sie kann also nirgendwo anders verwendet werden.

SCHLECHT!! Kein gültiger Einsatz von »s«!




sleep() kann die Variable »s« nicht sehen. Da sie sich nicht im gleichen Stack-Frame wie sleep() befindet, hat sleep() auch keine Ahnung, wer oder was »s« überhaupt ist.

Die Variable »s« lebt, aber ihr Geltungsbereich (Scope) ist auf die read()-Methode beschränkt. Wird sleep() beendet und read() befindet sich wieder an der Spitze des Stacks, kann read() »s« immer noch sehen. Wenn read() fertig ist und vom Stack entfernt wird, stirbt auch »s« und sieht sich die digitalen Radieschen von unten an.

## 1 Eine lokale Variable lebt nur innerhalb der Methode, in der sie deklariert wurde.

```
public void read() {
  int s = 42;
  // 's' kann nur innerhalb
  // dieser Methode benutzt
  // werden.
  // Endet die Methode,
  // verschwindet 's'
  // vollständig.
}
```

Die Variable »**s**« kann *nur* innerhalb der *read()*-Methode benutzt werden. Anders gesagt, ***der Geltungsbereich der Variablen ist auf ihre eigene Methode beschränkt***. Kein anderer Code in der Klasse (oder in anderen Klassen) kann »**s**« sehen.

## 2 Alle Instanzvariablen leben so lange wie ihr Objekt. Ist das Objekt noch am Leben, sind es auch seine Instanzvariablen.

```
public class Life {
  int size;

  public void setSize(int s) {
    size = s;
    // 's' verschwindet am Ende
    // dieser Methode, aber
    // 'size' kann überall in
    // der Klasse benutzt werden.
  }
}
```

Der Geltungsbereich der Variablen »s« (diesmal ein Methodenparameter) ist auf die setSize()-Methode beschränkt. Die Instanzvariable size ist dagegen während des Lebens des *Objekts* gültig und nicht während des Lebens der *Methode*.

## Der Unterschied zwischen **Leben** und **Geltungsbereich** für lokale Variablen:

### Leben

Eine lokale Variable ist so lange *lebendig*, wie sich ihr Stack-Frame auf dem Stack befindet, das heißt, *bis die Methode beendet wird*.

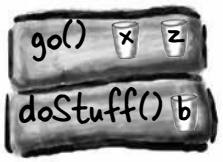
### Geltungsbereich

Der Geltungsbereich einer lokalen Variablen ist auf die Methode beschränkt, in der sie deklariert wurde. Ruft eine Methode eine andere auf, ist die Variable zwar lebendig, befindet sich aber außerhalb des aktuellen Geltungsbereichs, bis die aufrufende Methode weiterläuft. ***Sie können eine Variable nur innerhalb ihres Geltungsbereichs verwenden.***

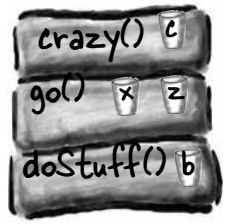
```
1  public void doStuff() {
     boolean b = true;
     go(4);
   }
2  public void go(int x) {
     int z = x + 24;
     crazy();
4    // mehr Code hier
   }
   public void crazy() {
3    char c = 'a';
   }
```

Sehen wir uns an, was auf dem Stack passiert, wenn jemand die doStuff()-Methode aufruft.

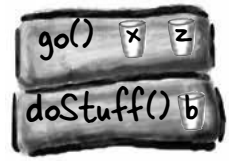

**1** ***doStuff()*** **landet auf dem Stack. Die Variable »b« ist lebendig und befindet sich im aktuellen Geltungsbereich.**




**2** ***go()*** **rutscht an die Spitze des Stacks. »x« und »z« sind lebendig und im aktuellen Geltungsbereich, »b« ist lebendig, aber** ***nicht*** **im aktuellen Geltungsbereich.**




**3** ***crazy()*** **wird auf den Stack geschoben. Jetzt ist »c« lebendig und im Geltungsbereich. Die anderen drei Variablen sind lebendig, befinden sich aber außerhalb des aktuellen Geltungsbereichs.**




**4** ***crazy()*** **wird beendet und verschwindet vom Stack. »c« befindet sich also außerhalb des aktuellen Geltungsbereichs und** ***ist tot****. Wenn go() weitermacht, wo es pausiert hat, sind »x« und »z« beide lebendig und befinden sich wieder im aktuellen Geltungsbereich. Die Variable »b« ist ebenfalls noch lebendig, aber nicht mehr im aktuellen Geltungsbereich (bis go() beendet wird).**

Solange eine lokale Variable lebendig ist, behält sie ihren Zustand. Solange sich beispielsweise die Methode doStuff() auf dem Stack befindet, behält die Variable »b« ihren Wert. Allerdings kann »b« nur benutzt werden, solange sich der Stack-Frame von doStuff() an der Spitze des Stacks befindet. Das heißt, Sie können lokale Variablen *nur* benutzen, während ihre Methode tatsächlich ausgeführt wird (sie also nicht nur darauf wartet, dass die über ihr liegenden Stack-Frames fertig sind).

# Was ist mit Referenzvariablen?

Die Regeln sind für elementare und Referenzvariablen gleich. Eine Referenzvariable kann nur benutzt werden, wenn sie sich im aktuellen Geltungsbereich befindet. Das heißt, Sie können die Fernbedienung eines Objekts nur verwenden, wenn sich eine Referenzvariable im Geltungsbereich befindet. Die *wahre* Frage lautet aber:

## »Wie wirkt sich das Leben der *Variablen* auf das Leben des *Objekts* aus?«

Ein Objekt ist so lange lebendig, wie lebendige Referenzen auf es verweisen. Verlässt eine Referenzvariable den Geltungsbereich, ist aber noch lebendig, ist auch das Objekt, auf das sie *verweist,* auf dem Heap noch am Leben. Und dann müssen Sie fragen: »Was passiert, wenn der Stack-Frame, der die Referenz enthält, am Ende der Methode vom Stack entfernt wird?«

War dies die einzige lebendige Referenz auf das Objekt, wird das Objekt auf dem Heap nun aufgegeben. Die Referenzvariable wurde mit dem Stack-Frame ins Nirwana geschickt. Das aufgegebene Objekt ist jetzt *offiziell* am Ende und damit ***ein Fall für den Garbage Collector***.

Sobald ein Objekt für die Garbage Collection (GC) qualifiziert ist, müssen Sie sich keine Gedanken mehr darum machen, dass der vom Objekt verwendete Arbeitsspeicher wieder freigegeben wird. Sollte Ihr Programm Speicherprobleme bekommen, wird der GC einige oder alle infrage kommenden Objekte zerstören, damit immer genug RAM verfügbar ist. Zwar kann es trotzdem noch zu Speicherproblemen kommen, aber *nicht,* bevor alle qualifizierten Objekte auf der Müllkippe gelandet sind. Sie müssen nur dafür sorgen, dass der GC auch etwas zum Einsammeln hat. Wenn Sie zu sehr an Ihren Objekten hängen, kann auch der GC Ihnen nicht helfen, und Sie riskieren, dass Ihr Programm schmerzvoll an einem Speicherverlust stirbt.

**Das Leben eines Objekts hat keinen Wert, keine Bedeutung, keinen Sinn, solange es keine Referenz darauf gibt.**

**Wenn Sie das Objekt nicht erreichen, können Sie es auch nicht auffordern, etwas zu tun, und es ist nur eine riesige fette Bitverschwendung.**

**Aber sobald ein Objekt nicht mehr erreichbar ist, wird der Garbage Collector das herausfinden. Früher oder später muss das Objekt dran glauben.**

**Ein Objekt qualifiziert sich für die GC, wenn seine letzte lebendige Referenz verschwindet.**

**Drei Wege, eine Objektreferenz loszuwerden:**

1. Die Referenz verlässt den Geltungsbereich für immer.

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

Die Referenz >>z<< stirbt am Ende der Methode.

2. Der Referenz wird ein anderes Objekt zugewiesen.

```
Life z = new Life();
z = new Life();
```

Das erste Objekt ist aufgegeben, wenn z auf ein neues Objekt >>reprogrammiert<< wurde.

3. Die Referenz wird explizit auf null gesetzt.

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

Das erste Objekt ist aufgegeben, wenn z >>deprogrammiert<< wurde.

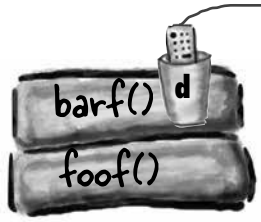
## Objekt-Killer Nr. 1

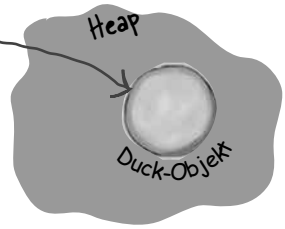
**Die Referenz verlässt für immer den Geltungsbereich.**

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

   public void barf() {
      Duck d = new Duck();
   }
}
```


**1** *foof()* **wird auf den Stack geschoben; es werden keine Variablen deklariert.**

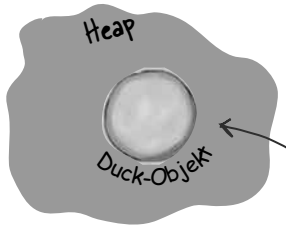

**2** *barf()* **wird auf den Stack geschoben, wo sie eine Referenzvariable deklariert und ein neues Objekt erstellt, das dieser Referenz zugewiesen wird. Das Objekt wird auf dem Heap angelegt, und die Referenz ist lebendig und im Geltungsbereich.**




Die neue Ente (das Duck-Objekt) landet auf dem Heap. Solange barf() ausgeführt wird, die Referenz »d« lebendig ist und sich im aktuellen Geltungsbereich befindet, ist auch die Ente am Leben.

**3** *barf()* **wird beendet und vom Stack entfernt. Ihr Frame löst sich in Wohlgefallen auf, also ist auch »d« nun tot. Die Ausführung kehrt zu** *foof()* ***zurück*****, aber** *foof()* **kann »d« nicht benutzen.**

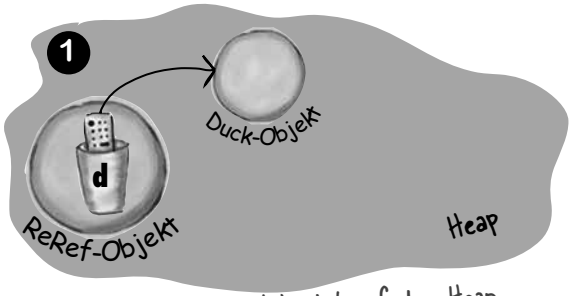




Oje! Die Variable »d« verschwand, als der Stack-Frame von barf() vom Stack gefegt wurde. Die Ente wurde aufgegeben und ist GC-Futter.

## Objekt-Killer Nr. 2

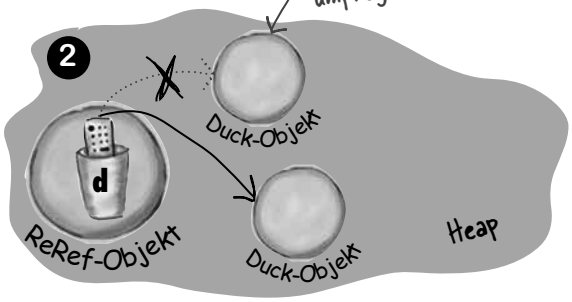
**Weisen Sie der Referenz ein anderes Objekt zu.**

```
public class ReRef {
  Duck d = new Duck();

  public void go() {
    d = new Duck();
  }
}
```




Das neue Duck-Objekt landet auf dem Heap und wird von >>d<< referenziert. Da >>d<< eine Instanzvariable ist, lebt das Duck-Objekt so lange, wie das ReRef-Objekt, das es instanziiert hat, am Leben bleibt. Es sei denn ...

Wenn jemand die Methode go() aufruft, wird dieses Duck-Objekt aufgegeben. Seine einzige Referenz wurde auf ein anderes Duck-Objekt umprogrammiert.



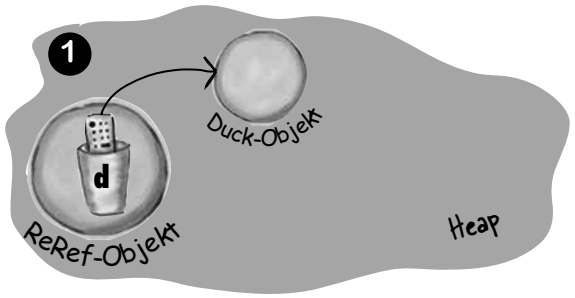

>>d<< wird ein neues Duck-Objekt zugewiesen, wodurch das ursprüngliche (erste) Duck-Objekt aufgegeben wird. Das erste Duck-Objekt ist damit so gut wie tot.

Alter! Du musstest doch nur die Referenz zurücksetzen. Aber vielleicht gab es damals einfach noch keine Speicherverwaltung.

## Objekt-Killer Nr. 3

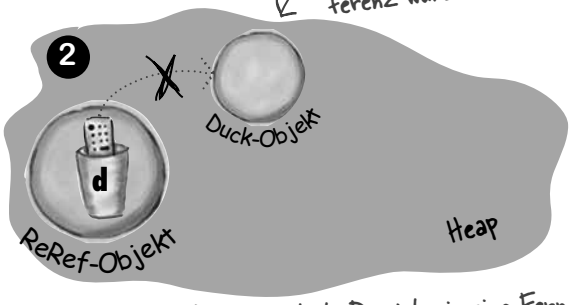
**Die Referenz explizit auf null setzen.**

```
public class ReRef {
  Duck d = new Duck();

  public void go() {
    d = null;
  }
}
```




Das neue Duck-Objekt landet auf dem Heap und wird von »d« referenziert. Da »d« eine Instanzvariable ist, lebt das Duck-Objekt so lange wie das ReRef-Objekt, das es instanziiert hat. Es sei denn ...

Dieses Duck-Objekt wurde aufgegeben. Seine einzige Referenz wurde auf null gesetzt.




»d« wurde auf null gesetzt. Das ist wie eine Fernbedienung, die nicht programmiert ist. Solange »d« nicht reprogrammiert ist (ihr also noch kein neues Objekt zugewiesen wurde), dürfen Sie nicht einmal den Punktoperator darauf anwenden.

### Die Bedeutung von `null`

Wenn Sie eine Referenz auf **`null`** setzen, dann programmieren Sie die Fernbedienung neu. Das heißt, Sie haben zwar eine Fernbedienung, aber keinen Fernseher, den Sie damit steuern können. Eine Nullreferenz enthält Bits, die für »null« stehen. (Wir wissen nicht, welche Bits das sind. Es ist uns auch egal, solange die JVM damit klarkommt.)

Wenn Sie in der wahren Welt die Knöpfe einer nicht programmierten Fernbedienung drücken, passiert schlicht gar nichts. In Java können Sie auf einer Nullreferenz keine Knöpfe drücken (den Punktoperator verwenden), weil die JVM weiß (das ist ein Laufzeitproblem, kein Compiler-Fehler), dass Sie ein Bellen erwarten, es aber keinen Hund gibt.

**Wenn Sie den Punktoperator auf einer Nullreferenz verwenden, erhalten Sie zur Laufzeit eine NullPointerException.** Alles Wissenswerte zu Exceptions finden Sie in Kapitel 13.

# Kamingespräche

Heute Abend: **Eine Instanzvariable und eine lokale Variable reden über Leben und Tod (und bleiben dabei erstaunlich höflich)**

**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 Menschen 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.

Unter lokalen Variablen wird der Ausdruck »temporäre Variable« als abwertend angesehen. Wir bevorzugen »lokal«, »Stack«, »automatisch« oder »geltungsbereichsbeschränkt«.

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 gestopft ... und wenn dann die Methode, zu der wir gehören, eine andere Methode aufruft, wird uns ein weiterer Frame aufgeladen. Und wenn *diese* Methode ihrerseits wieder eine *andere* Methode aufruft ... und so weiter. Manchmal müssen wir ewig 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 im Tiefschlaf – was die Leute in Science-Fiction-Filmen immer machen, wenn sie große Entfernungen überbrücken müssen. Na ja, eigentlich eher so eine Art Scheintod. Wir hängen einfach in der Warteschleife.

## Instanzvariable

Ich habe mal einen Dokumentarfilm darüber gesehen. Scheint ein ziemlich grausames Ende zu sein. 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 ziemlich luxuriös sein kann. Viele von uns haben deshalb Schuldgefühle, besonders 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.

Die lassen uns *trinken*?

## Lokale Variable

Solange unser Frame noch da ist, sind wir und die Werte, die wir enthalten, 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, desto 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, als entfernte man einen Fleck. 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 lange wie das Objekt, das euch deklariert hat, oder? Nehmen wir beispielsweise an, es gibt ein Hund-Objekt mit einer Halsband-Instanzvariablen. 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, solange wir die Zeit dazu haben. Carpe RAM und alles.


# SEIEN Sie der Garbage Collector

Welche der Codezeilen auf der rechten Seite würde, wenn sie an Punkt A eingefügt würde, dafür sorgen, dass genau ein zusätzliches Objekt für die Garbage Collection freigegeben würde? (Nehmen Sie an, dass Punkt A – // weitere Methoden aufrufen – sehr lange ausgeführt wird, damit der GC Zeit hat, seinen Job zu tun.)

Antworten auf Seite 272.

```
public class GC {
  public static GC doStuff() {
    GC newGC = new GC();
    doStuff2(newGC);
    return newGC;
  }

  public static void main(String[] args) {
    GC gc1;
    GC gc2 = new GC();
    GC gc3 = new GC();
    GC gc4 = gc3;
    gc1 = doStuff();

    A

    // weitere Methoden aufrufen
  }

  public static void doStuff2(GC copyGC) {
    GC localGC = copyGC;
  }
}
```

1 `copyGC = 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 erstellt. Ihre Herausforderung besteht darin, das »beliebteste« Objekt zu finden, also das Objekt, auf das die meisten Referenzvariablen verweisen. Danach notieren Sie, wie viele Referenzen es insgesamt auf das Objekt sind und welche! Wir beginnen, indem wir eines der neuen Objekte und seine Referenzvariable zeigen.

Viel Glück!

→ Antworten auf Seite 272.

```
class Bees {
  Honey[] beeHoney;
}

class Raccoon {
  Kit rk;
  Honey rh;
}

class Kit {
  Honey honey;
}

class Bear {
  Honey hunny;
}

public class Honey {
  public static void main(String[] args) {
    Honey honeyPot = new Honey();
    Honey[] ha = {honeyPot, honeyPot, honeyPot, honeyPot};
    Bees bees = new Bees();
    bees.beeHoney = ha;
    Bear[] bears = new Bear[5];
    for (int i = 0; i < 5; i++) {
      bears[i] = new Bear();
      bears[i].hunny = honeyPot;
    }
    Kit kit = new Kit();
    kit.honey = honeyPot;
    Raccoon raccoon = new Raccoon();
    raccoon.rh = honeyPot;
    raccoon.rk = kit;
    kit = null;
  } //  Ende von main
}
```

Dies ist ein Racoon-(Waschbär-)Objekt!

Und hier ist seine Referenzvariable, raccoon.

## Kurzkrimi

Sarah klang verzweifelt. »Wir haben die Simulation jetzt viermal laufen lassen, und die Temperatur des Hauptmoduls bewegt sich ständig vom Sollwert in Richtung Kälte«, sagte sie. »Wir haben erst letzte Woche den neuen Temperatur-Bot installiert. Die ausgelesenen Werte der Radiator-Bots, die den Wohnbereich herunterkühlen sollen, scheinen im grünen Bereich zu sein. Daher haben wir unsere Analyse auf die Wärmespeicher-(Retention-)Bots konzentriert, die beim Heizen der Quartiere helfen sollen.« Tom seufzte. Zuerst schien es, als könnte die Nanotechnologie ihnen enorme Zeitvorteile verschaffen. Aber jetzt, nur fünf Wochen vor dem Start, bestanden einige der wichtigsten Lebenserhaltungssysteme des Orbiters den simulierten Stresstest immer noch nicht.

»Welche Verhältnisquotienten simuliert ihr?«, fragte Tom.

»Wenn ich deine Frage richtig verstehe, 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-Radiator-Bot-SimUnits in einem 2:1-Verhältnis mit den V2-Radiator-SimUnits testen«, fuhr Sarah fort. »Insgesamt soll das Verhältnis von Wärmespeicher-Bots zu Radiator-Bots 4:3 betragen.«

»Wie ist der Energieverbrauch, Sarah?«, fragte Tom. 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 Radiator-Bots von dem der Wärmespeicher-Bots zu trennen.« »Insgesamt soll der Energieverbrauch«, ergänzte Sarah, »im Verhältnis 3:2 stehen, wobei die Radiatoren mehr Energie aus dem Wireless Grid ziehen dürfen.«

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

```
import java.util.ArrayList;

class V2Radiator {
  V2Radiator(ArrayList<SimUnit> list) {
    for (int x = 0; x < 5; x++) {
      list.add(new SimUnit("V2Radiator"));
    }
  }
}

class V3Radiator extends V2Radiator {
  V3Radiator(ArrayList<SimUnit> lglist) {
    super(lglist);
    for (int g = 0; g < 10; g++) {
      lglist.add(new SimUnit("V3Radiator"));
    }
  }
}

class RetentionBot {
  RetentionBot(ArrayList<SimUnit> rlist) {
    rlist.add(new SimUnit("Retention"));
  }
}
```

## Kurzkrimi, Fortsetzung ...

```
import java.util.ArrayList;

public class TestLifeSupportSim {
  public static void main(String[] args) {
    ArrayList<SimUnit> aList = new ArrayList<SimUnit>();
    V2Radiator v2 = new V2Radiator(aList);
    V3Radiator v3 = new V3Radiator(aList);
    for (int z = 0; z < 20; z++) {
      RetentionBot ret = new RetentionBot(aList);
    }
  }
}

class SimUnit {
  String botType;

  SimUnit(String type) {
    botType = type;
  }

  int powerUse() {
    if ("Retention".equals(botType)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Tom warf einen kurzen Blick auf den Code, und es erschien ein kleines Lächeln auf seinen 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 Tom? Wie konnte er die Abweichung der Energieverbrauchswerte raten? Und welche wenigen Codezeilen würden Sie dem Programm hinzufügen, um den Fehler zu beheben?***

→ Antworten auf Seite 273.

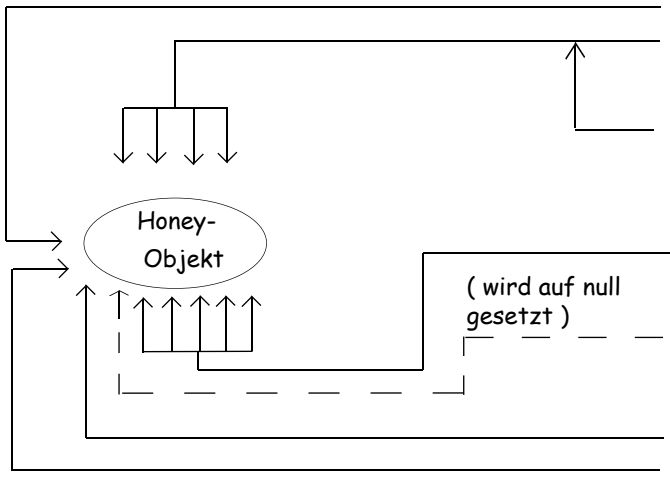
## Lösungen zu den Übungen

### Seien Sie der Garbage Collector
(von Seite 268)

| | | |
|---|---|---|
| 1 | `copyGC = null;` | Nein. Diese Zeile versucht, auf eine Variable außerhalb des Geltungsbereichs zuzugreifen. |
| 2 | `gc2 = null;` | Okay. gc2 war die einzige Referenzvariable, die sich auf das Objekt bezog. |
| 3 | `newGC = gc3;` | Nein. Noch eine Variable außerhalb des Geltungsbereichs. |
| 4 | `gc1 = null;` | Okay. gc1 enthält die einzige Referenzvariable, weil newGC sich außerhalb des Geltungsbereichs befindet. |
| 5 | `newGC = null;` | Nein. newGC ist außerhalb des Geltungsbereichs. |
| 6 | `gc4 = null;` | Nein. gc3 bezieht sich noch auf das Objekt. |
| 7 | `gc3 = gc2;` | Nein. gc4 bezieht sich noch auf das Objekt. |
| 8 | `gc1 = gc4;` | Okay. Die einzige Referenz auf das Objekt erhält eine neue Zuweisung. |
| 9 | `gc3 = null;` | Nein. gc4 bezieht sich noch auf das Objekt. |

### Beliebte Objekte
(von Seite 269)

Wahrscheinlich war es nicht besonders schwer, herauszufinden, dass das Honey-Objekt, das zuerst von der Variablen honeyPot referenziert wurde, das bei Weitem »beliebteste« Objekt in dieser Klasse ist. Vermutlich war es aber nicht ganz so leicht zu erkennen, dass alle Variablen aus dem Code, die auf das Honey-Objekt zeigen, ***dasselbe Objekt*** referenzieren! Es gibt insgesamt zwölf aktive Referenzen auf das Objekt, bevor die main( )-Methode beendet wird. Die Variable *kit.honeyPot* ist eine Zeit lang gültig, aber *kit* wird am Ende auf null gesetzt. Da *raccoon.rk* immer noch auf das Kit-Objekt verweist, referenziert *raccoon.kit.honeyPot* das Objekt immer noch (obwohl es nie explizit deklariert wurde).




```
public class Honey {
  public static void main(String[] args) {
    Honey honeyPot = new Honey();
    Honey[] ha = {honeyPot, honeyPot,
                  honeyPot, honeyPot};
    Bees bees = new Bees();
    bees.beeHoney = ha;
    Bear[] bears = new Bear[5];
    for (int i = 0; i < 5; i++) {
      bears[i] = new Bear();
      bears[i].hunny = honeyPot;
    }
    Kit kit = new Kit();
    kit.honey = honeyPot;
    Raccoon raccoon = new Raccoon();

    raccoon.rh = honeyPot;
    raccoon.rk = kit;
    kit = null;
  } //  Ende von main}
```

## Kurzkrimi, Lösung (von den Seiten 270/271)

Tom bemerkte, dass der Konstruktor für die Klasse V2Radiator eine ArrayList erwartete. Das bedeutete, dass jedes Mal, wenn der *V3*Radiator-Konstruktor aufgerufen wurde, im super()-Aufruf an den *V2*Radiator-Konstruktor eine ArrayList übergeben wurde. Das hieß, dass jedes Mal fünf zusätzliche V2Radiator-SimUnits erzeugt wurden. Wenn Tom recht hatte, würde der Gesamtenergieverbrauch 120 betragen und nicht die 100, die Sarahs angenommene Verhältnisquotienten erwarten ließen.

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