Alle Studiengänge melden sich für die Schein-/ Bachelorklausur Teil 1 über die Lehrstuhlwebseite an.

Anmeldeschluß ist Fr. 08.12.2006 um 17:00 Uhr.

http://media.informatik.rwth-aachen.de/programmierung_ws0607.html

Nach der Anmeldung und nach der Korrektur der 6. Übung erhalten Sie eine Bestätigungsemail.

Rückblick

Wieviel h/Woche Mindestaufwand außer Vorlesung & Übung?
Unterschied Methoden überladen vs. überschreiben ?

Was sind abstrakte Klassen ?

Was sind abstrakte Methoden ?

Woher stammt das Innere eines jeden Objektes?
Was ist & kann die Klasse **Object**?

Eigenschaften von ArrayList<Object>?

Wie und weshalb castet man Referenzvariablen?

Was sind Interfaces und wofür brauchen wir sie?

Interfaces: abstrakte Klassen

```
public interface KannTanken {

   // alle Interface-Methoden sind öffentlich und abstrakt
   public abstract void tanken ();

}
```

```
public class VerkehrsFlugzeug extends Flugzeug implements KannTanken {
   // ...
}
```

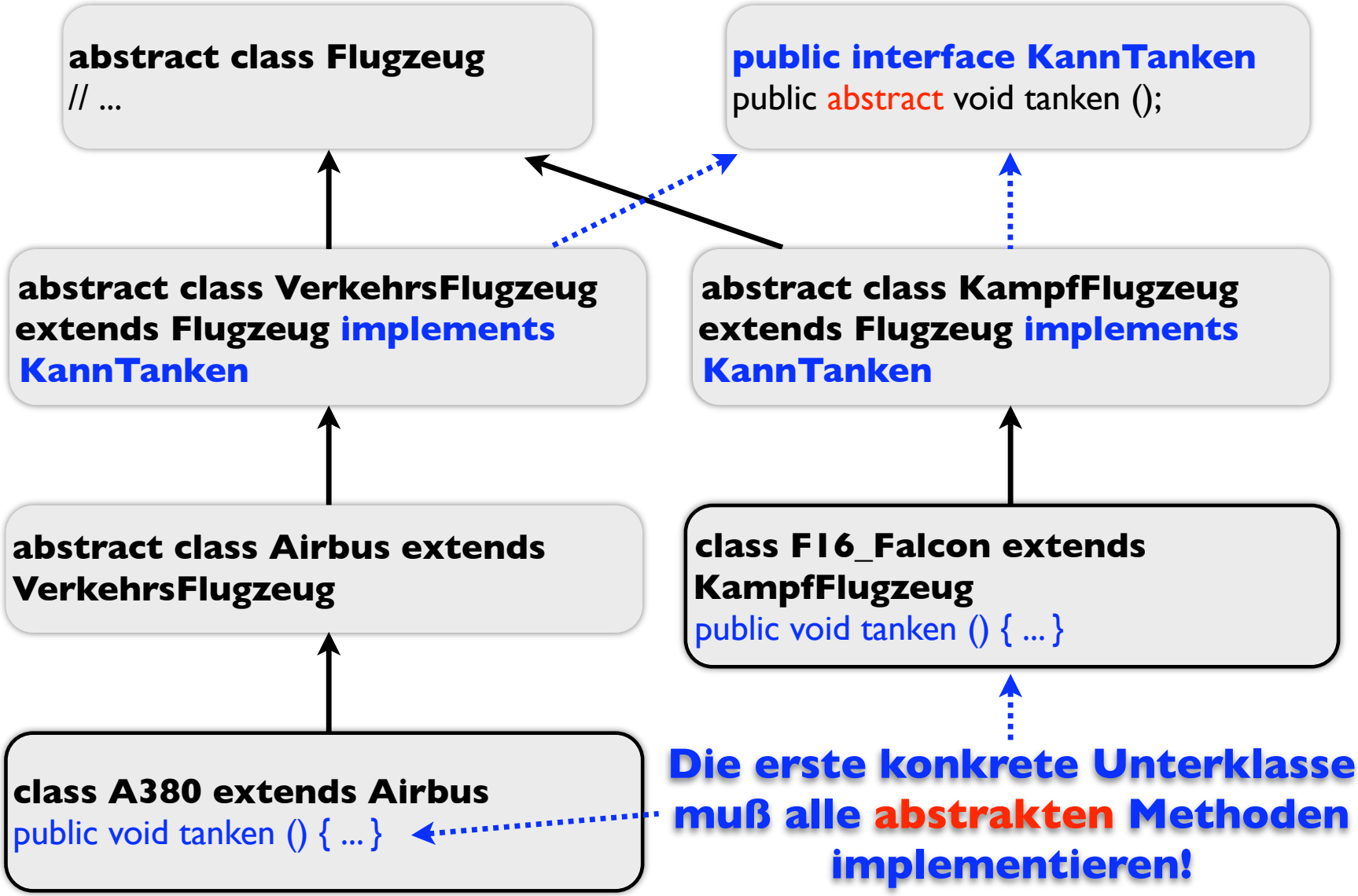
```
public class KampfFlugzeug extends Flugzeug implements KannTanken {
   // ...
}
```

Interfaces: abstrakte Klassen

```
public interface KannÜberschallFliegen {

   public abstract void mitÜberschallFliegen ();    // ohne Rumpf

}
```

```
public class KampfFlugzeug extends Flugzeug implements KannTanken, KannÜberschallFliegen {
   //...
}
```

Eine Klasse kann mehrere Interfaces implementieren. Sie hat aber nur eine Superklasse.


abstract class Flugzeug
// ...
public interface KannTanken
public abstract void tanken ();
abstract class VerkehrsFlugzeug extends Flugzeug implements KannTanken
abstract class KampfFlugzeug extends Flugzeug implements KannTanken
abstract class Airbus extends VerkehrsFlugzeug
class F16_Falcon extends KampfFlugzeug
public void tanken () { ... }
class A380 extends Airbus
public void tanken () { ... }
Die erste konkrete Unterklasse muß alle abstrakten Methoden implementieren!

Interfaces und Polymorphie

```
class Werkstatt {
    // Motor warten und Flugzeug tanken
    public void flugzeugTanken (KannTanken einFlieger) {
        einFlieger.tanken ();
    }
}
```

```
ArrayList<KannTanken> tankbareFlugzeuge = new ArrayList<KannTanken> ();
tankbareFlugzeuge.add (new A380 ());
tankbareFlugzeuge.add (new F16_Falcon ());

WerkStatt pitStop = new Werkstatt ();
for (KannTanken einFlugObjekt :  tankbareFlugzeuge ) {
    pitStop.flugzeugTanken (einFlugObjekt);
}
```

Jetzt sind Sie wieder an der Reihe!

Lesen Sie zu Interfaces und Polymorphie
Kapitel 8.

Konstruktoren & Garbage Collection

Stack-Speicher

springen ()

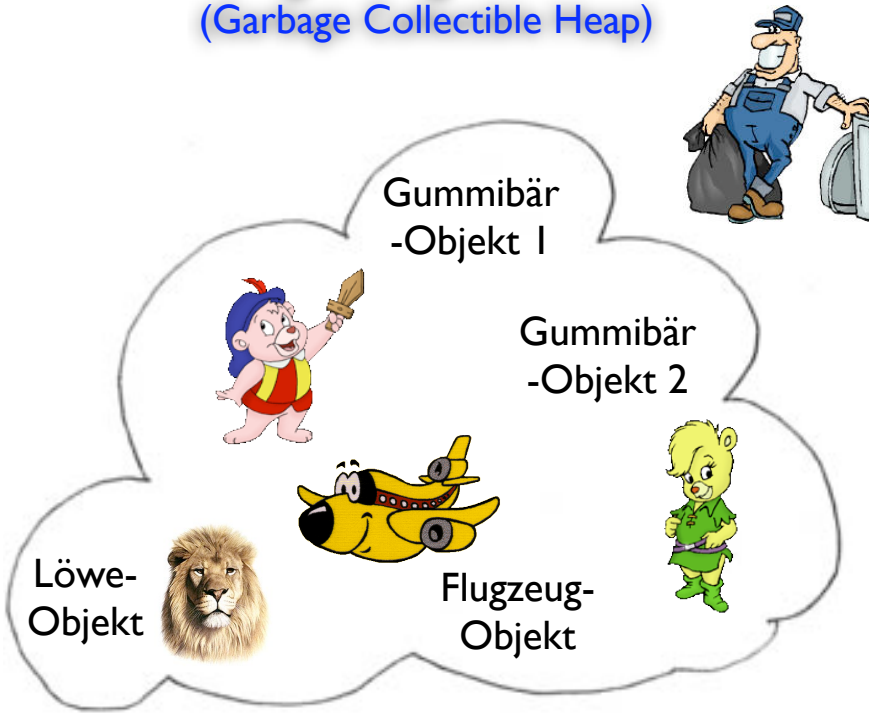
saftTrinken ()

main ()

Hier leben alle **Methodenaufrufe** und **lokale Variablen**.

Heap-Speicher

(Garbage Collectible Heap)



Hier leben alle **Objekte** und **Instanzvariablen**.

Instanzvariablen vs. lokale Variablen

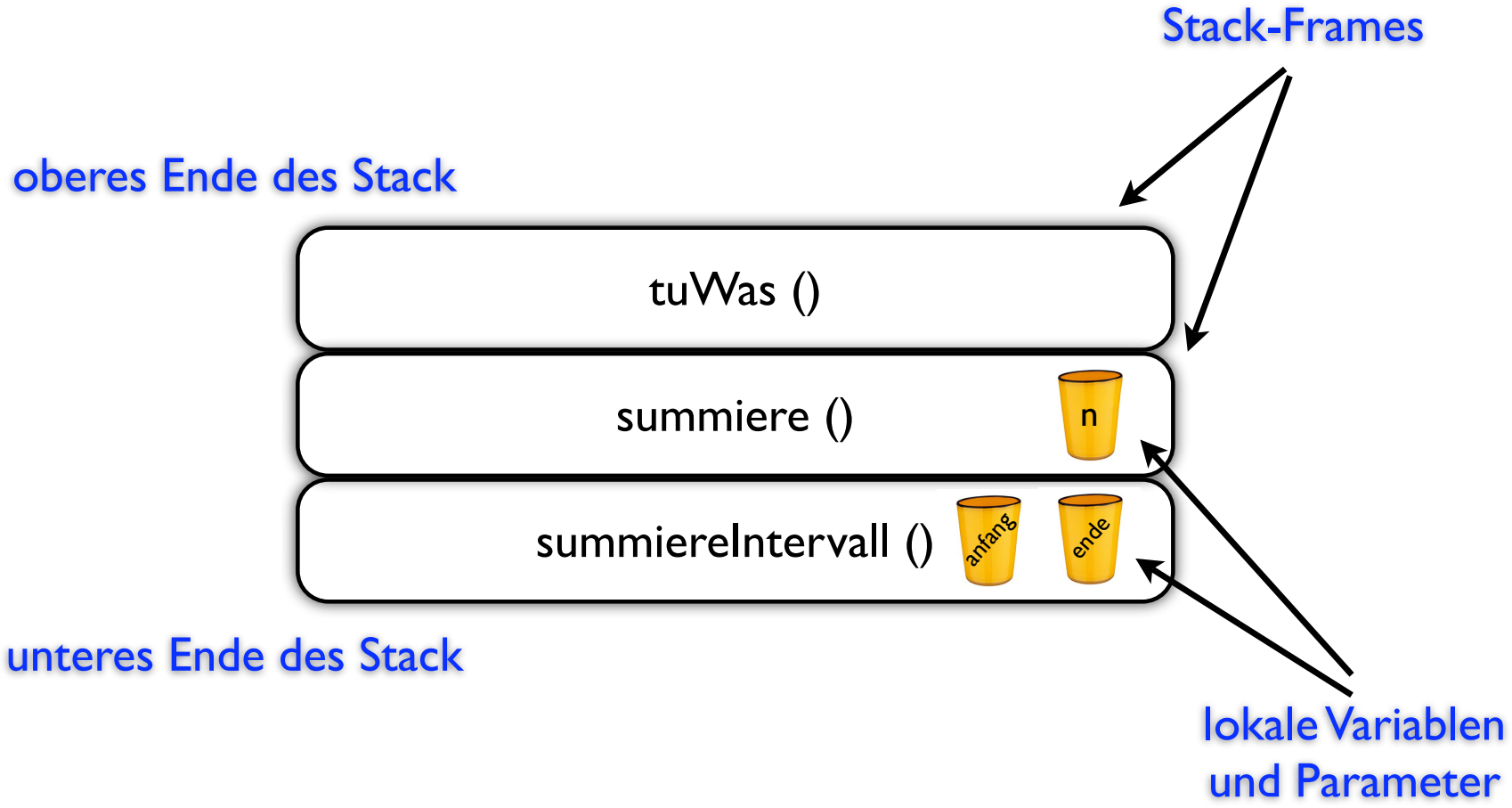
```
class Gummibär {
   String name;      // Instanzvariable
   ...
}
```

Instanzvariablen werden in einer Klasse deklariert und leben innerhalb von Objekten.

```
public static void main (String[] args) {
   int i;    // i und args sind lokale Variablen
   ...
}
```

Lokale Variablen werden in einer Methode deklariert und leben, solange die Methode ausgeführt wird (d.h., auf dem Stack ist).

Der Stack-Speicher



Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);    // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```

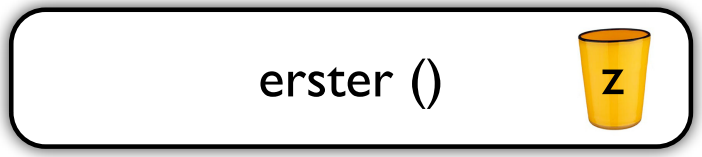
Wie sieht der Stack aus, wenn erster() aufgerufen wird?

Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);    // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```



Stack-Speicher

Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);   // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```

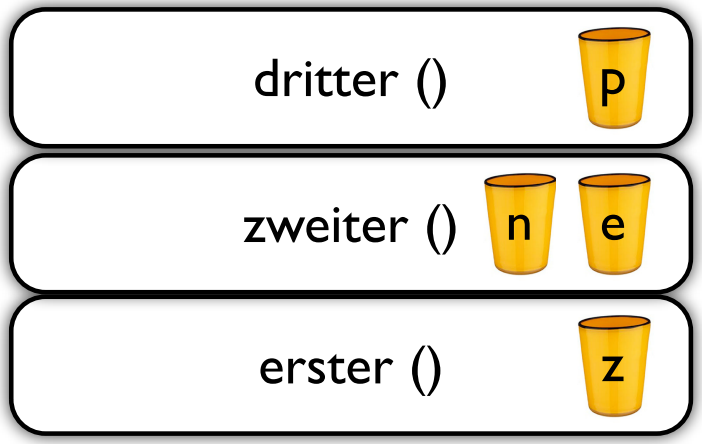
Stack-Speicher

Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);   // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```



Stack-Speicher

Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);   // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```

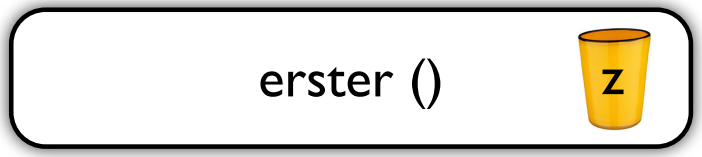
Stack-Speicher

Lokale Variablen auf dem Stack

```
public void erster () {
   int z = 13;                  // lokale Variable
   zweiter (z);
}
```

```
public int zweiter (int n) {  // lok. Par.
   int e =  dritter (n * 2);    // lok. Var.
   ...
}
```

```
public int dritter (int p) {   // lok. Par.
   return p++;
   ...
}
```

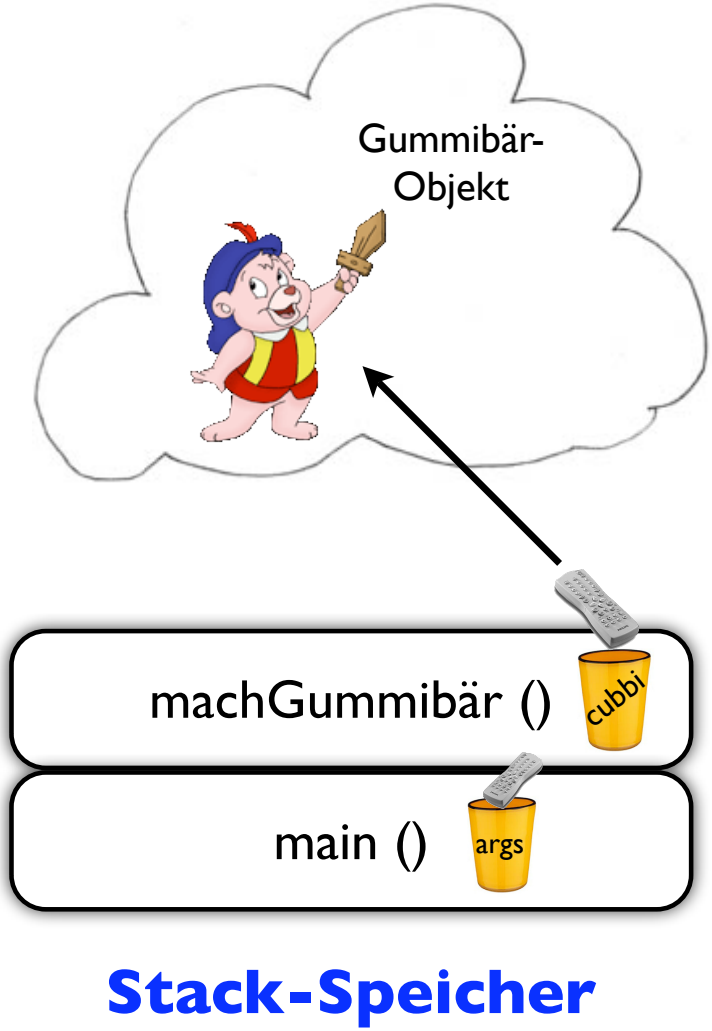


Stack-Speicher

Lokale Referenz-Variablen leben auf dem Stack

```
class GummibärTest {
   public static void main (String[] args) {
      machGummibär ();
   }

   public void machGummibär () {
      Gummibär cubbi = new Gummibär ();
   }
}
```

cubbi und args sind lokale Referenz-Variablen und leben auf dem Stack!

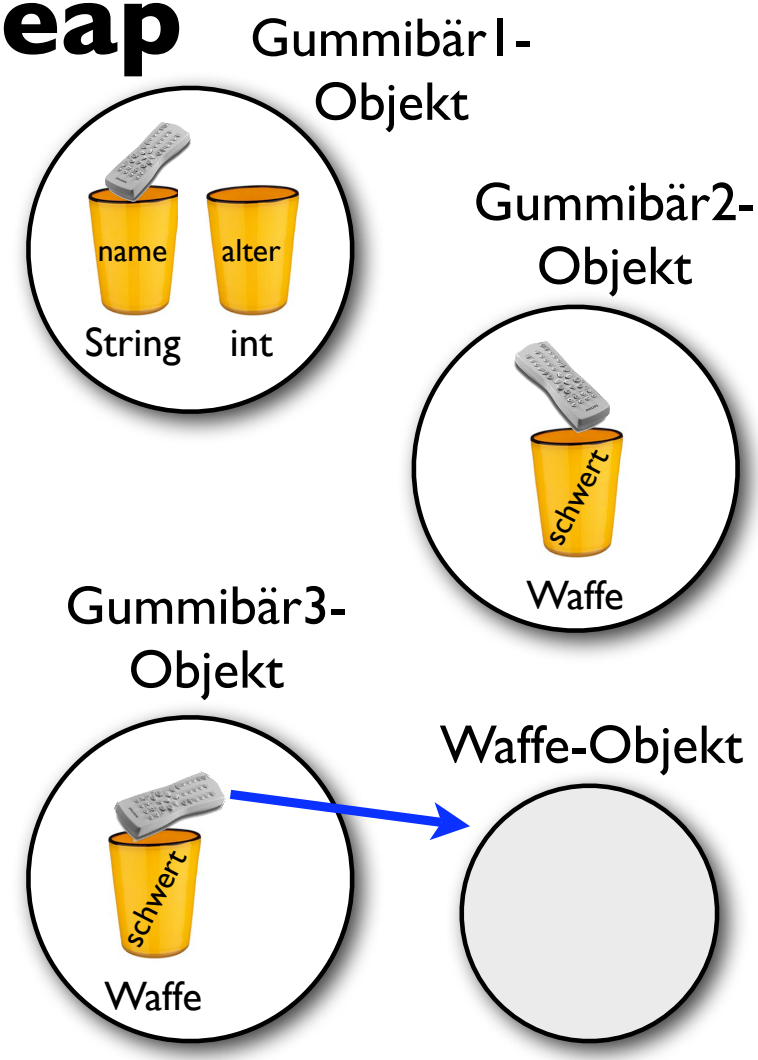


Instanzvariablen leben in **Objekten** auf dem **Heap**

```
class Gummibär1 {
   String name;        // Instanzvariable
   int alter;          // elementare Inst.var.
}
```

```
class Gummibär2 {
   Waffe schwert;  // Referenzvariable
                   // als Instanzvariable
}
```

```
class Gummibär3 {
   // Referenzvariable und neues Objekt
   Waffe schwert = new Waffe ();
}
```



Konstruktoren erzeugen Objekte

```
Gummibär cubbi = new Gummibär ();
```

Referenzvariable deklarieren

Variable mit Objekt verknüpfen

Objekt erzeugen

Konstruktoraufruf

```
public Gummibär () {  // Default-Konstruktor
    // Konstruktoren tragen den Namen der Klasse
    // und haben keinen Rückgabetyp !!!
}
```

Jede Klasse hat einen Konstruktor!

Den Default-Konstruktor erstellt der Compiler (wenn wir es nicht machen).

Gummibären konstruieren

```
public class Gummibär {
    private String name;
    private int alter;

    public Gummibär () {     // Der Default-Konstruktor sollte
        name = "Nobody";     // niemals fehlen, wenn er Sinn macht!
        alter = 100;         // (hier macht er wenig Sinn)
    }

    public Gummibär (String n, int a) { // überladener Konstruktor
        name = n;                       // mit Parametern
        alter = a;
    }

    public void setName () {...}        // Setter-Methoden
    public void setAlter () {...}
}
```

```
Gummibär einGummibär  = new Gummibär ();    // (sinnloser?) Default-Konstruktor
Gummibär cubbi = new Gummibär ("Cubbi", 7);  // überladener Konstruktor
```

Geht das?


```
public class Gummibär {
    private String name;
    private int alter;

    public Gummibär (String n, int a) {
        name = n;  alter = a;
    }

    public Gummibär (String einName, int einAlter) {
        name = einName; alter = einAlter;
    }

    public Gummibär (int a, String n) {
        name = n;  alter = a;
    }
}
```

// nur Parameternamen ändern
// geht nicht !

// ok, überladener Konstruktor,
// Parametertypen mit
// unterschiedlicher Reihenfolge

Mehr zu Konstruktoren

```
public class Gummibär {
    private String name;
    private int alter;

    // Wo ist der Default-Konstruktor ???

    public Gummibär (String n, int a) { // Konstruktor mit Parametern
        name = n;
        alter = a;
    }

    // Setter-Methoden...
}
```

```
Gummibär einGummibär  = new Gummibär ();      // Compiliert nicht !!!
Gummibär cubbi = new Gummibär ("Cubbi", 7);   // überladener Konstruktor
```

Der Compiler erstellt keinen Default-Konstruktor, wenn wir mindestens einen Konstruktor mit Parametern schreiben!

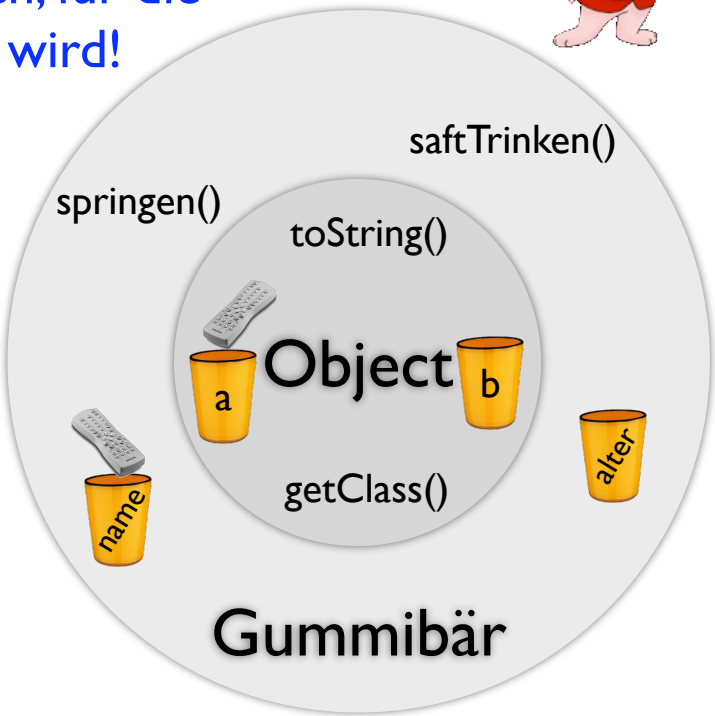
Instanzvariablen im inneren Object

```
class Object
  private Foo a;
  private int b;

  public toString () {...}
  public getClass () {...}
  ...
```

IS-A

```
class Gummibär
  private String name;
  private int alter;

  public springen () {...}
  public saftTrinken () {...}
  ...
```

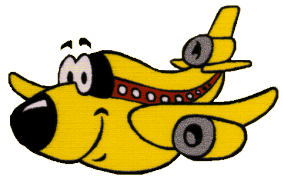
Die ultimative Superklasse hat auch Instanzvariablen, für die Speicher reserviert wird!



Gummibär cubbi = new Gummibär ();

ein einzelnes Gummibär-Objekt auf dem Heap

Superklassen-Konstruktoren und Konstruktorverkettung



class Object
private Foo a;
private int b;

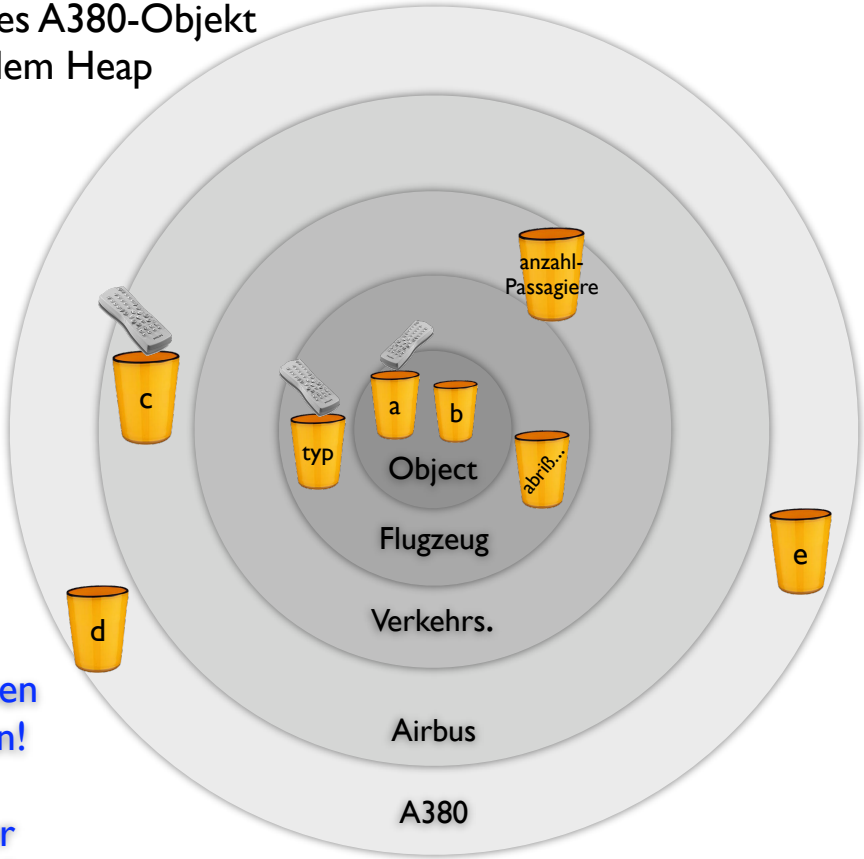
abstract class Flugzeug
private String typ;
private float abrißGeschwindigkeit;

abstract class VerkehrsFlugzeug
private int anzahlPassagiere;
...

abstract class Airbus
String c;

class A380
int d, e;

ein einzelnes A380-Objekt
auf dem Heap

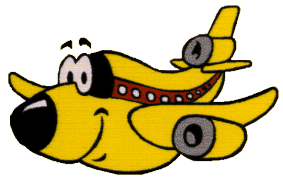


Auch abstrakte Klassen haben Konstruktoren!

Was passiert bei der Erzeugung eines A380-Objektes?

```
A380 eineA380 = new A380 ();
```

Superklassen-Konstruktoren und Konstruktorverkettung



class Object
private Foo a;
private int b;

abstract class Flugzeug
private String typ;
private float abrißGeschwindigkeit;

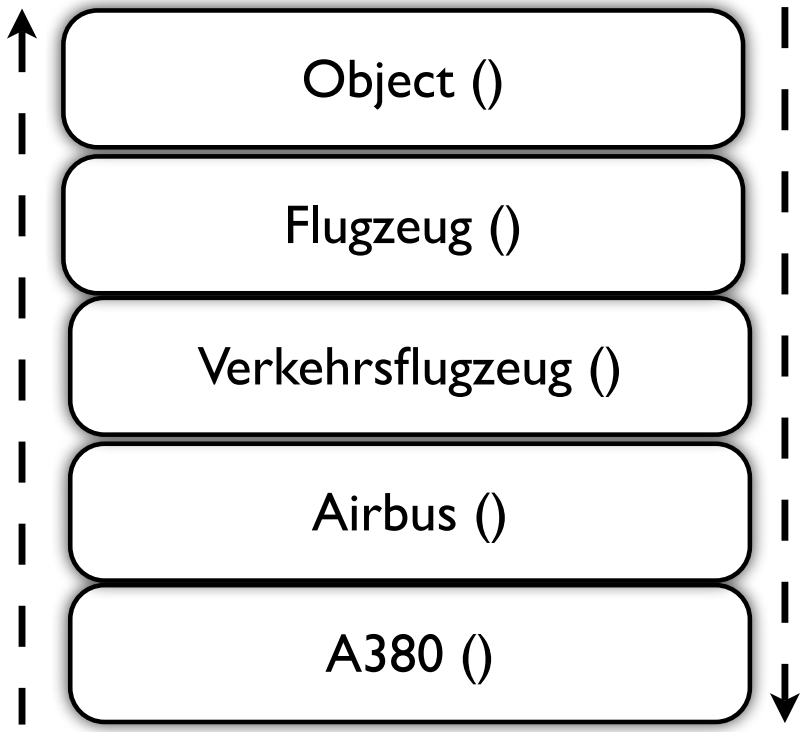
abstract class VerkehrsFlugzeug
private int anzahlPassagiere;
...

abstract class Airbus
String c;

class A380
int d, e;

Auch abstrakte Klassen haben Konstruktoren!

Was passiert bei der Erzeugung eines A380-Objektes?



Stack-Speicher

```
A380 eineA380 = new A380 ();
```

super: Aufruf des Superklassen-Konstruktors

Object ()
Konstruktor

```
public abstract class Flugzeug {
    private String typ;
    private float abrißGeschwindigkeit;

    public Flugzeug () {  // Default-Konstruktor ruft Object-Konstruktor auf
        super ();             // fügt der Compiler ein, wenn wir es nicht machen
    }
    public Flugzeug (String marke, float geschwindigkeit) {   // überladener K.
        super ();             // fügt der Compiler ein, wenn wir es nicht machen
        typ = marke;  abrißGeschwindigkeit = geschwindigkeit;
    }
}
```

```
public abstract class VerkehrsFlugzeug extends Flugzeug {
    private int anzahlPassagiere;

    // Der Compiler fügt immer super() ein,  nie einen überladenen K. !!!
    public VerkehrsFlugzeug () {  // Default-Konstruktor ruft Flugzeug-K. auf
        super ();             // fügt der Compiler ein, wenn wir es nicht machen
    }
}
```

super: Aufruf des Superklassen-Konstruktors

```
public abstract class Flugzeug {
    private String typ;
    private float abrißGeschwindigkeit;

    public Flugzeug () {  // Default-Konstruktor ruft Object-Konstruktor auf
        super ();
    }
    public Flugzeug (String marke, float geschwindigkeit) {   // überladener K.
        typ = marke;  abrißGeschwindigkeit = geschwindigkeit;
        super ();          // geht nicht, super () muß die erste Anweisung sein !!!
    }
}
```

```
public abstract class VerkehrsFlugzeug extends Flugzeug {
    private int anzahlPassagiere;

    // Der Compiler fügt immer super() ein,  nie einen überladenen K. !!!
    public VerkehrsFlugzeug () {  // Default-Konstruktor ruft Flugzeug-K. auf
        super ();          // fügt der Compiler ein, wenn wir es nicht machen
    }
}
```

super: Aufruf des Superklassen-Konstruktors

```
public abstract class Flugzeug {
    private String typ;                          // private Instanzvariablen sind in den
    private float abrißGeschwindigkeit;  // Unterklassen nicht sichtbar

    public Flugzeug () {  // Default-Konstruktor ruft Object-Konstruktor auf
        super ();
    }
    public Flugzeug (String marke, float geschwindigkeit) {   // überladener K.
        super ();              // Object-Konstruktor aufrufen
        typ = marke;  abrißGeschwindigkeit = geschwindigkeit;
    }
}
```

```
public abstract class VerkehrsFlugzeug extends Flugzeug {
    private int anzahlPassagiere;
    // Default-Konstruktor wegen Platzmangel nicht gezeigt

    // Wir rufen den überladenen Konstruktor der Superklasse Flugzeug auf !!
    public VerkehrsFlugzeug (String marke, float geschwindigkeit) { // überladen
        super (marke, geschwindigkeit);   // private Instanzv. von Flugzeug setzen
    }
}
```

this: Konstruktor-Aufruf innerhalb der Klasse

```
public abstract class Airbus extends VerkehrsFlugzeug {
    public Airbus () {
        super ();     // Default-Konstruktor ruft Default-VerkehrsF.-Konstruktor
    }

    // Den überladenen Konstruktor der Superklasse Verkehrsflugzeug aufrufen
    public Airbus (String marke, float geschwindigkeit) { // überladen
        super (marke, geschwindigkeit);   // überladenen K. von VerkehrsF. aufr.
    }
}
```

```
public class A380 extends Airbus {
    // this ruft einen Konstruktor in der gleichen Klasse auf
    public A380 () {                  // Default-K. gibt Default-Werte vor
        this ("A380", 250.0f);   // den überladenen K. von A380 aufrufen
    }

    //  nur unser "richtiger Konstruktor" ruft den Superklassen-K. auf
    public A380 (String marke, float geschwindigkeit) {
        super (marke, geschwindigkeit);  // überladenen K. von Airbus aufrufen
    }
}
```

Beispielaufrufe

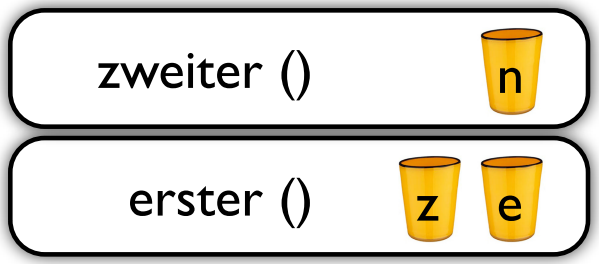
```
// ruft den überladenen Konstruktor mit this auf
// Typ A380, abrißGeschwindigkeit = 250.0
A380 eineA380 = new A380 ();

// Typ A380, abrißGeschwindigkeit = 265.0
A380 eineA380 = new A380 ("A380", 265.0f);

// Typ unbekannt, abrißGeschwindigkeit = 105.0
Flugzeug einFlieger = new Flugzeug ("unbekannt", 105.0);
```

Geltungsbereich von lokalen Variablen und Referenzvariablen

```
class Test {
   // lokale Variablen z und e sind...
   public void erster () {
      int z = 13;              // ... nur in dieser
      int e = zweiter (z);  // Methode sichtbar
   }

   // n ist nur in dieser Methode sichtbar
   public int zweiter (int n) {  // lokaler Par.
      z = z * 2;     // z ist hier nicht sichtbar !!!
      return n * 2;
   }
}
```

erster() kann **n** nicht sehen !

n ist nur in der Methode zweiter() gültig, d.h., in ihrem Geltungsbereich.



z und **e** sind nur in der Methode erster() gültig.

zweiter() kann **z** und **e** nicht sehen !

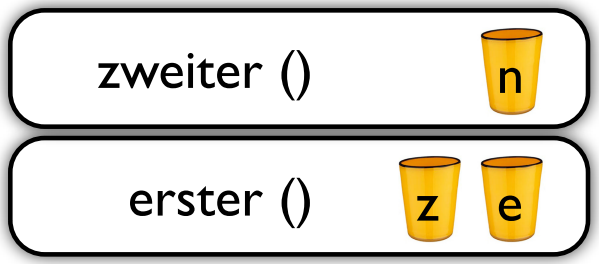
Stack-Speicher

Leben von lokalen Variablen und Referenzvariablen

```
class Test {
   // lokale Variablen z und e sind...
   public void erster () {
      int z = 13;              // ... nur in dieser
      int e = zweiter (z);  // Methode sichtbar
   }

   // n ist nur in dieser Methode sichtbar
   public int zweiter (int n) {  // lokaler Par.
      z = z * 2;     // z ist hier nicht sichtbar !!!
      return n * 2;
   }
}
```

erster() kann **n** nicht sehen !

n ist nur in der Methode zweiter() gültig. Sie lebt, solange zweiter() auf dem Stack ist.



z und **e** sind nur in der Methode erster() gültig. Sie leben, solange erster() auf dem Stack ist.

zweiter() kann **z** und **e** nicht sehen !

Stack-Speicher

Geltungsbereich von Instanzvariablen und Instanz-Referenzvariablen

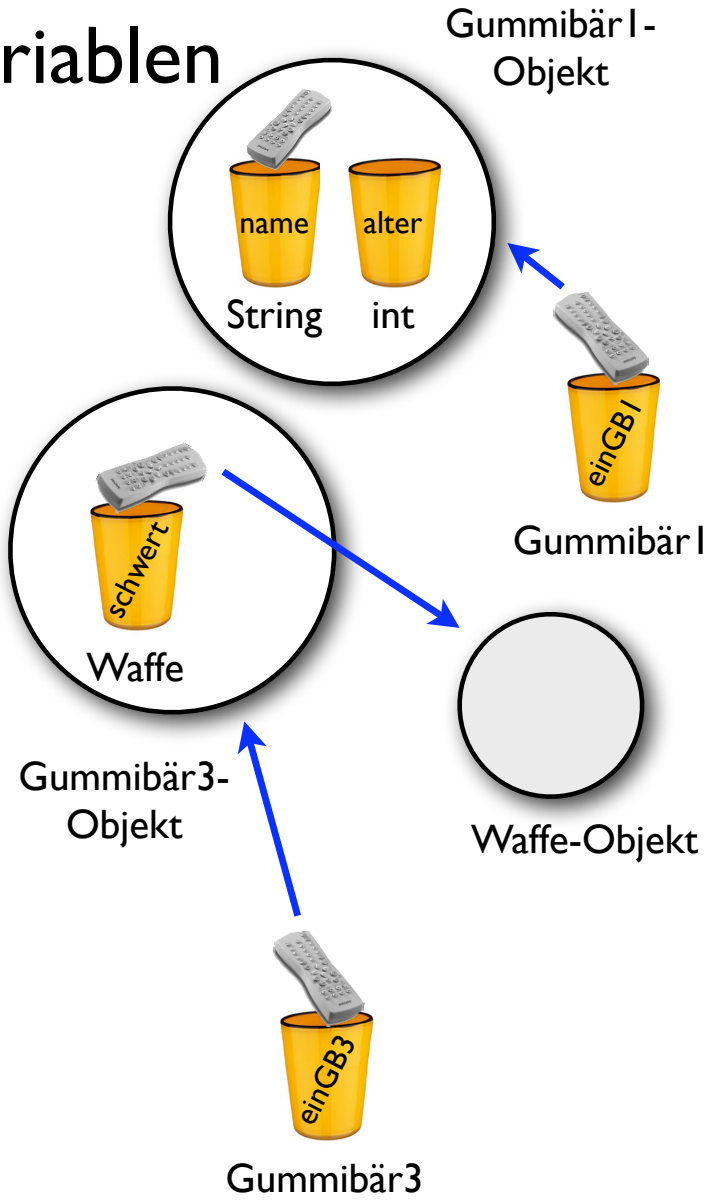
```
class Test {
   int ergebnis;  // Instanzvariablen und Instanz-Referenzvariablen sind
   Foo a;         // überall in der Klasse sichtbar

   public void erster () {
      int z = 13;    // lokale Variable z ist nur in dieser
      zweiter (z);  // Methode sichtbar
   }

   // zweiter () kann auf die Instanzvariable ergebnis zugreifen
   public void zweiter (int n) {  // lokaler Parameter
      ergebnis = n * 2;
   }
}
```

Leben von Instanz-Referenzvariablen

```
class Gummibär1 {
    String name;        // Instanzvariablen
    int alter;
}
```

```
class Gummibär3 {
    // Instanz-Referenzvariable und neues Objekt
    Waffe schwert = new Waffe ();
}
```

```
// irgendwo im Programm
Gummibär1 einGB1 = new Gummibär1 ();
Gummibär3 einGB3 = new Gummibär3 ();
```



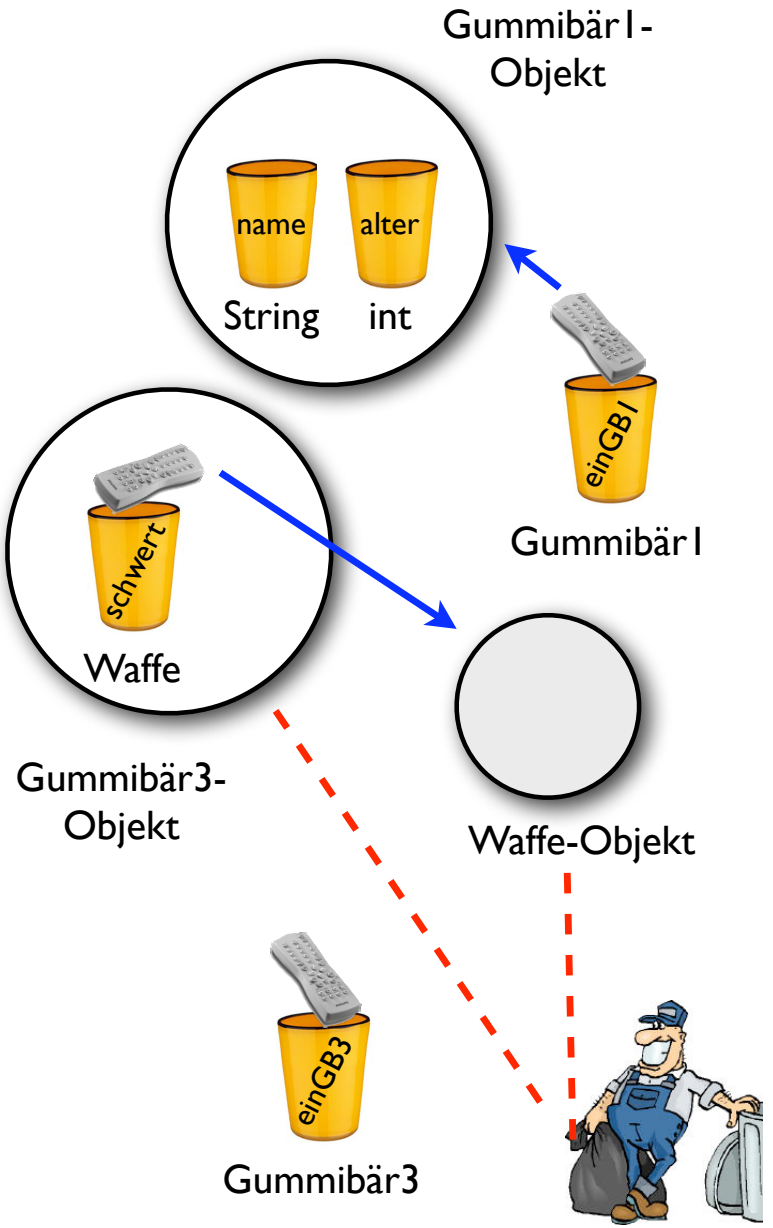
Instanzvariablen leben in Objekten.

Leben & Sterben von Instanz-Referenzvariablen

```
class Gummibär1 {
    String name;        // Instanzvariablen
    int alter;
}
```

```
class Gummibär3 {
    // Instanz-Referenzvariable und neues Objekt
    Waffe schwert = new Waffe ();
}
```

```
// irgendwo im Programm
einGB3 = null;
```



Instanzvariablen leben und sterben mit ihren Objekten.

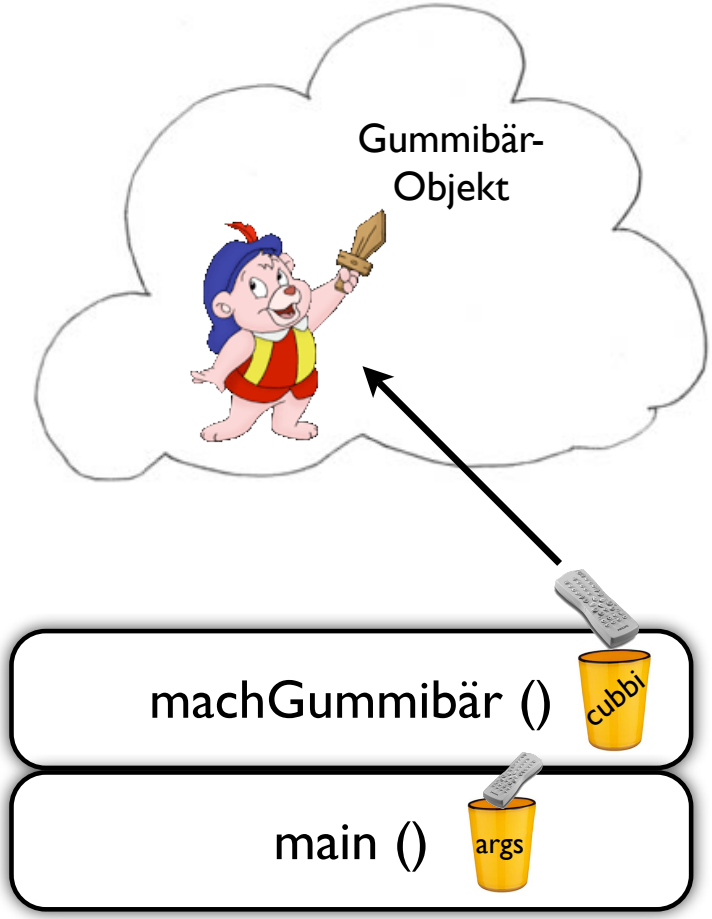
Leben und Sterben von Objekten

(oder auch "cubbi lebt nur kurz")

```
class GummibärTest {
   public static void main (String[] args) {
      machGummibär ();
   }

   public void machGummibär () {
      Gummibär cubbi = new Gummibär ();
   }
}
```

GummibärTest test = new GummibärTest ();

cubbi ist eine
lokale Referenzvariable!

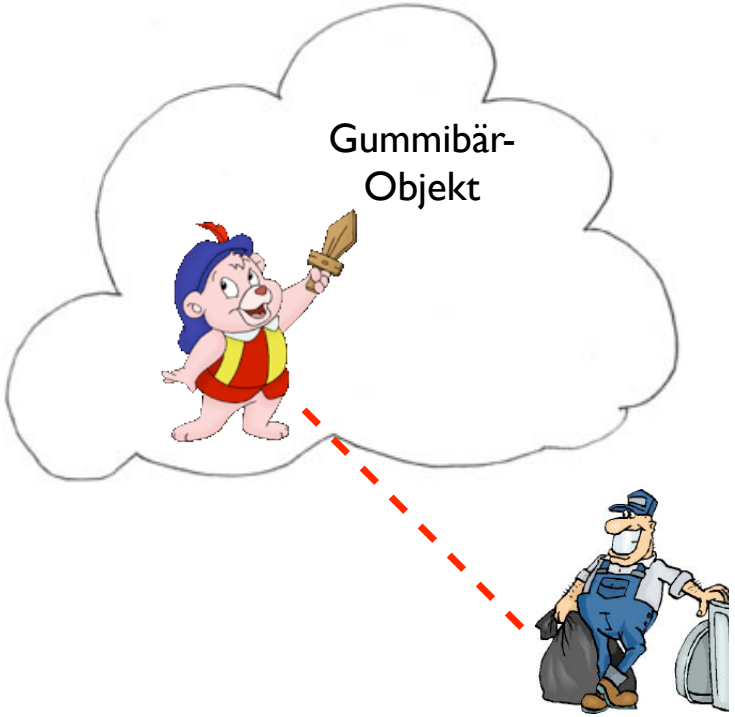


Leben und Sterben von Objekten

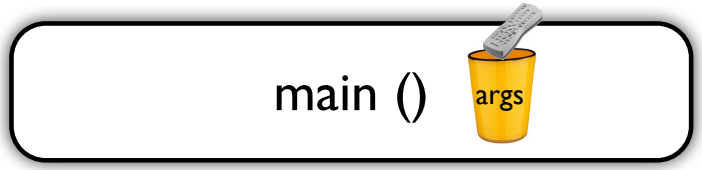
(oder auch "cubbi lebt nur kurz")

```
class GummibärTest {
   public static void main (String[] args) {
      machGummibär ();
   }

   public void machGummibär () {
      Gummibär cubbi = new Gummibär ();
   }
}
```

Heap-Speicher



cubbi "lebte" nur solange machGummibär() auf dem Stack war!



Stack-Speicher

Jetzt sind Sie wieder an der Reihe!

Lesen Sie zu Konstruktoren und Garbage Collection **Kapitel 9**.