Von Hamstern und Robotern (Teil B)

Homepage: http://www.heeb.ch/java/microworlds/
Feedback bitte an: feedback@heeb.ch

Release	Datum	Beschreibung
Version 0.0.1	15.01.05	Inhaltskonzept.
Version 0.1.0	09.03.05	Bereits früher erstelltes und als zu speziell ausgesondertes Thema zu Konstruktuktoren (Thema 1) und Gegenseitigen Nachrichtenaustausch (Thema 2) integriert
Version 0.1.1	20.04.05	Erklärung zu Eclipse, einbinden externer Jar-Archive Zu Thema 2, die Möglichkeit der inneren Klassen angefügt.
Version 0.2.0	12.05.05	Thema 6: Java-Projekte realisieren eingefügt. Thema 7: Dateien speichern, teilweise.

Teil B: Java 2 mit dem Hamstermodell

In diesem Teil wird es darum gehen, die Besonderheiten der Programmiersprache Java mithilfe des Hamstermodels noch besser kennen zu lernen.

Inhaltskonzept dieses Teils

- Konstruktoren
- Innere Klassen
- Pakete
- Spezielle Kontrollstrukturen: switch, continue, break, exception handling (Ausnahmebehandlung)
- Rechnen mit «Java 2»
- Daten speichern und auslesen
- Multi-Tasking

Kapitel 1: Spezielles in «Java 2»

Thema 1: Die Initialisierung eines Objektes

Werte initialisieren mit Konstruktoren

Bisher haben wir die Anfangswerte (Initialwerte) mit der Nachricht init(reihe,spalte,richtung,körnerZahl) übermittelt. Beim Standardhamster können wir die Initialwerte auch gleich beim Instantiieren setzen:

```
Hamster willi = new Hamster(3,2,1,10);
```

Wir wollen lernen, auch einem Spezialhamster (abgeleiteten Hamster) Initialwerte mitzugeben. Dies geschieht mit einem sogenannten Konstruktor. Konstruktoren haben **Ähnlichkeiten mit Methoden**. Sie weisen aber wichtige Besonderheiten auf.

Aufbau von Konstruktoren

Beispiel:

```
class DummerHamster extends Hamster
{
  DummerHamster()
  {
    super();
    init(2,2,1,0);
  }
}
```

Konstruktoren **heissen** gleich wie die **Klasse**.
Sie kommen im Gegensatz zu Methoden **ohne das Wort void** vor dem Konstruktor, der ja gleich heisst wie die Klasse, aus.

Sie werden nicht vererbt, rufen dafür in der ersten Zeile einen Konstruktor der Oberklasse auf.
Dies mit **super()**

Statt den parameterlosen Konstruktor von Hamster, hätten Sie auch den Konstruktor mit den Initialwerten verwenden können:

```
class DummerHamster extends Hamster
{
  DummerHamster()
  {
    super(2,2,1,0);
  }
}
```

super(2,2,1,0) ruft hier den gleichen Konstruktor auf , der bei **new Hamster(2,2,1,0)** aufgerufen wird.

Verwenden Sie keinen eigenen Konstruktor, so wird automatisch folgender Konstruktor verwendet:

```
class DummerHamster extends Hamster
{
  DummerHamster()
  {
    super();
  }
}
```

Aus diesem Grund konnten Sie auch bisher auf die Definition eines Konstruktors verzichten.

Flexibler Exemplare erzeugen mit mehreren Konstruktoren

Sicher fragen Sie sich, ob man nicht flexibel Meister und Hund in einem Schritt irgendwo im Feld platzieren könnte. Etwa mit folgendem Code:

```
void main()
{
  Meister m = new Meister(3,5,1);
}
```

Werte für Reihe, Spalte und Richtung sollen im Konstruktor bestimmt (übergeben) werden.

Damit sollen Meister und Hund an der Position (5,5) mit Richung 1 (Ost) instantiiert sein.
Dazu müssen wir für Meister einen zweiten Konstruktor definieren:

```
class Meister extends Hamster
{
  Hamster hund

  Meister()
  {
    super(5,5,1,0);
    hund = new Hamster(5,5,1,0);
  }

  Meister(int r, int s, int rtg)
  {
    super(r,s,rtg,0);
    hund = new Hamster(r,s,rtg,0);
  }
}
```

Da drei ganzzahlige Werte übergeben wurden, wird dieser Konstruktor aufgerufen. Der Name **r** steht jetzt fix für den 1. übergebenen Wert (3), **s** für den 2. Wert (5) und **rtg** für den 3. Wert (1).
Die Typangabe **int** für Ganzzahl ist zwingend erforderlich.

Hamster besitzt keinen Konstruktor mit 3 Werten, deshalb müssen wir denjenigen mit 4 Werten verwenden.

Wir können auch den Hund zuerst instantiieren und die Referenz (die Visitenkarte des Hundes) als Argument für den Konstruktor verwenden. Der Code für die Instantiierung von Hund und Meister kann dann etwa wie folgt aussehen:

```
void main()
{
  Hamster bari;
  bari = new Hamster(4,6,2,0);
  Meister andrea;
  andrea = new Meister(bari);
}
```

Wir können auch bari direkt instantiieren und die Referenz dem Konstruktor als Argument übergeben:

```
void main()
{
  Meister andrea;
  andrea = new Meister(new Hamster(4,6,2,0));
}
```

Damit dieser Code funktioniert, müssen wir im Meister einen Konstruktor definieren, der mit einer Referenz (Visitenkarte) eines Hamsters etwas anfangen kann.
Da ein Konstruktor der oben definierten Art bereits in der Klasse Hamster definiert ist, hätten wir den Konstruktor auch wie folgt definieren können.

```
class Meister extends Hamster
{
  Hamster hund

  Meister()
  {
    super(5,5,1,0);
    hund = new Hamster(5,5,1,0);
  }

  Meister(Hamster h)
  {
    super(h);
    hund = h;
  }
}
```

Unter h ist die Referenz, Visitenkarte des Hundes gespeichert.

Ruft den gleichen Konstrukor auf, wie der Aufruf:
Hamster anna, beate;
anna = new Hamster(1,1,1,0);
beate = **new Hamster(anna)**;

Um das Thema der verschiedenen Konstruktoren abzuschliessen, wollen wir noch einen Konstruktor definieren, der als Argument nur die Richtung entgegennimmt:

```
class Meister extends Hamster
{
  Hamster hund

  Meister(int r, int s, int rtg)
  {
    super(r,s,rtg,0);
    hund = new Hamster(5,5,1,0);
  }

  Meister(Hamster h)
  {
    super(h.getReihe(),h.getSpalte(),h.getRichtung(),0);
    hund = h;
  }

  Meister(int rtg)
  {
    this(5,5,rtg);
  }

  public void vor()
  {
    super.vor();
    hund.vor();
  }
}
```

Ist das Argument ein Hamster, wird dieser Konstuktor ausgeführt.

Ist das Argument eine Ganzzahl, wird dieser Konstuktor ausgeführt.

this ruft einen anderen Konstruktor der gleichen Klasse auf. Es steht anstelle von super(..) in der 1. Zeile des Konstruktors. Im Beispiel hier wird der erste der drei Konstruktoren aufgerufen.

Aufgabe zu Konstruktoren

Richten Sie die Klasse Meister vollständig ein. Instantiieren Sie Meister und Hund von der Methode void main() aus und lassen Sie die beiden ein wenig laufen.

Definitionen und Initialisierungsblock

Gibt es eine Möglichkeit, Anweisungen vor dem aufruf des Konstruktors der Oberklasse («super(…)») ausführen zu lassen? – Ja, die gibt es. Dies ist entweder gleich bei der Initialisierung dcr Instanzvariablen oder im sogenannten Initialisierungsblock möglich.

Beispiel bei der Initialisierung:

```
class Meister extends Hamster
{
  Hamster hund = new Hamster();

  Meister(int r, int s)
  {
    super(r,s,0,0);
    hund.init(r,s,0,0);
  }
}
```

Definitionen werden zu aller erst ausgeführt.

Daher ist hund hier bereits als Referenz auf ein Hamsterobjekt definiert. Wir können mit der Refernz dashalb arbeiten.

Beispiel mit Hilfe eines Initialisierungsblockes::

```
class Meister extends Hamster
{
  Hamster hund = null;

  {
    hund = new Hamster();
    hund.init(5,5,0,0);
  }

  Meister(int r, int s)
  {
    super(hund.getReihe()+r,hund.getSpalte()+s,0,0);
  }

}
```

Beginn des Initialiserungsblockes. Der Initailiserungsblock wird zusammen mit den Definitionen der Felder (Instanzvariablen), von oben nach untern ausgeführt.

Ende des Initialisierungsblockes.

Der Konstruktor kann bereits auf den initialisierten «hund» Bezug nehmen.

Ein Initialisierungsblock ist dann sinnvoll, wenn die Initialisierung der Instanzvariablen einheitlich ist und nicht in einem Befehl erfolgen kann. Der Block wird immer vor dem Konstruktor ausgeführt, selbst dann, wenn er

unterhalb des Konstruktors in der Klasse stehen würde. Wichtig: Der Block kann sich nicht innerhalb einer Methode oder eines Konstruktors befinden.

Merken Sie sich

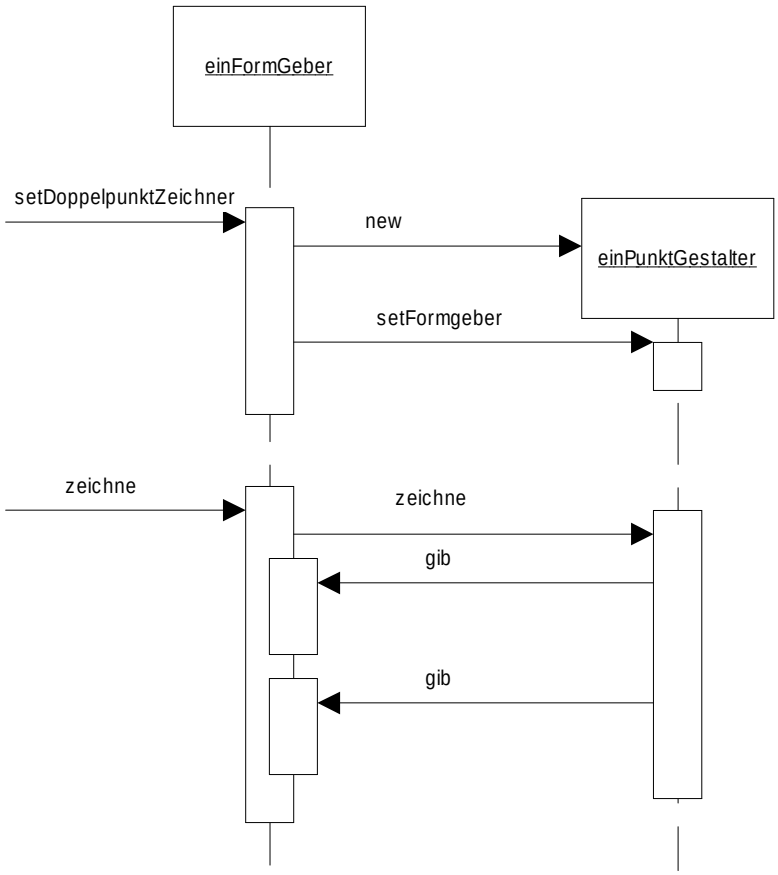
• Konstruktoren werden nicht vererbt , die erste Zeile des Konstruktors ruft daher einen Konstruktor der Oberklasse auf `super();` oder z. B. `super(3,4,0,10);`
• Konstruktoren heissen gleich wie die Klasse (Gross- und Kleinschreibung beachten!)
• Es gibt immer mindestns einen Konstruktor, wenn Sie nichts scheiben ist dies der Standardkonstruktor `«Klassenname»(){ super(); }`
• Die erste Zeile eines Konstruktore enthält den aufruf eines Konstruktors der Oberklasse: `super(…);`
Oder einen vorhandenen der gleichen Klasse: `this(…)`
• Instanzvariablen können Sie auch innerhalb eines Initialisierungsblocks Werte zuweisen, wenn ein Befehl bei der Deklaration der Instanzvariablen nicht ausreicht..

Übungsvorschlag

Schreiben Sie bestehende Lösungen um unter Verwendung von Konstruktoren in den Unterklassen von Hamster um, so dass die «main»-Methode keine Aufrufe von «init(…)» mehr enthält.

Thema 2: Gegenseitig Nachrichten austauschen

Wir kommen bei diesem Thema auf die Hamster zurück, die gemeinsam ein Haus zeichnen. Wir haben damals gesehen, dass es am einfachsten ist, wenn ein Hamster die Form vorgibt und der andere sich um die Gestaltung eines Punktes kümmert. Dabei haben wir die beiden Hamster miteinander laufen lassen. Jetzt betrachen wir eine andere Möglichkeit. Der Hamster, der die Punkte gestaltet soll selbst gar nicht laufen und Körner ablegen (zeichnen). Er soll nur den Form gebenden Hamster richtig steuern, dass er die gewünschte Strategie für das Zeichnen eines Punktes anwendet.
Beispiel: Der Hamster Maurer sendet dem DoppelpunktZeichner die Nachricht «zeichne». Daraufhin übermittelt der DoppelpunktZeichner innerhalb seiner Methode «zeichne» zweimal die Nachricht «gib» zurück. Dies bewirkt das Zeichnen eines Doppelkorn-Punktes durch den Maurer.



Damit dieser Vorgang funktionieren kann, muss der DoppelpunktZeichner die Visitenkarte (Referenz) des Maurers gespeichert haben. Das Einrichten eines entsprechenden Speicherplatzes kennen Sie schon:

```
class Zeichner extends Hamster
{
  Hamster formGeber;
}
```

Aber wie kommt die Visitenkarte (Refernz) des Formgebers in diesen Speicherplatz? Dazu verwenden wir ein Argument vom Typ Hamster.

Visitenkarten übermitteln mit Argumenten

Nebst dem Namen einer Methode kann eine Nachricht als Beilage (Argumente) Referenzen (Visitenkarten von Hamstern) enthalten. Damit dies funktioniert, müssen folgende Vorkehren getroffen werden:
1. Der Empfänger der Nachricht muss nebst dem Methodennamen auch die Beilage kennen. Eine Beilage (Argument) deklariert man wie folgt:

```
class Zeichner extends Hamster
{
  Hamster formGeber;

  void setFormGeber(Hamster fg)
  {
    formGeber = fg;
  }
}
```

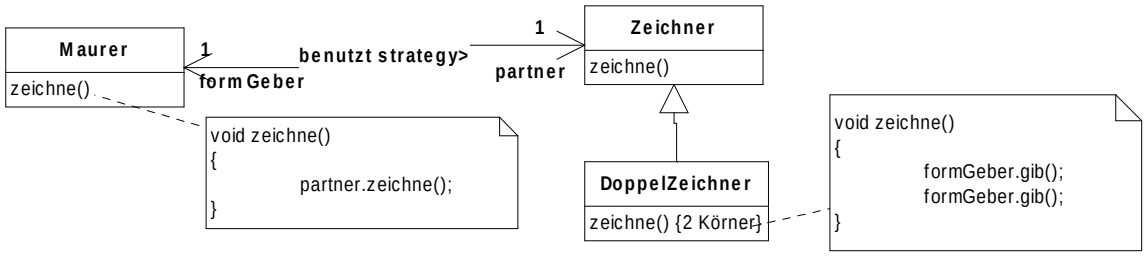
Zwischen den Klammern nennt man zuerst den Hamster-Typ (hier ist es ein ganz normaler Hamster). Anschliessend nach einem Leerschlag folgt ein Name für die Visitenkarte (Referenz). Standardmässig wählt man eine Abkürzung.
Innerhalb der Methode weist man den Namen für die Visitenkarte (das sogenannte Argument oder den Parameternamen) der Klassenvariable (formGeber) zu. So wird er intern im Zeichner gespeichert.
Mehrer Argumente sind übrigens auch möglich. Man würde diese von einander durch Komma abtrennen.
2. Der Sender muss seine Visitenkarte mit der Nachricht mitsenden. Dies sieht wie folgt aus:

```
class Maurer extends Hamster
{
  Zeichner partner;

  void setDoppelpunktZeichner
  {
    partner = new DoppelpunktZeichner();
    partner.setFormGeber(this);
  }
}
```

Mit «**this»** gibt ein Hamster seine eigene Visitenkarte ab. Es mag befremden, dass die eigene Visitenkarte «this» und nicht «me» oder «my_self» heisst. Daran müssen Sie sich einfach gewöhnen.

Gegenseitiger Nachrichtenaustausch (Klassendiagramm)



Im Klassendiagramm kann der gegenseitige Nachrichtenaustausch durch **Pfeile in beiden Richtungen der Assoziation** angezeigt werden.

Alternative: Innere Klassen

Eine interessante Alternative zur Lösung des gegenseitigen Nachrichtenaustausches sind innere Klassen.
Was sind innere Klassen? – Innere Klassen sind vollwertige Klassen, die vollwertige Objekte enthalten. Sie bieten alle Möglichkeiten der normalen Klassen. Ja: Sie können selbst sogar innere Klassen enthalten. Das einzige was innere Klassen nicht können, sie enthalten selbst keine statischen Elemente (static members).

Wie definiert man eine innere Klasse?

Die Normalform der inneren Klasse ist die sogenannte Member-Klasse. Member (zu deutsch Element) ist ein Oberbegriff für alle Elemente der Klasse (Instanzvariablen, Instanzmethoden und eben Member-Klassen). Man definiert sie so, wie dies folgendes Beispiel zeigt. Die Klasse Handwerker aoll als Basisklasse für Maurer, Dachdecker und Zimmermann dienen.

```
class Handwerker extends Hamster
{
  private class Zeichner extends Object
  {
    protected void zeichne()
    { gib();
    }
  }

  private class DoppelZeichner extends Zeichner
  {
    protected void zeichne()
    { gib(); gib();
    }
  }

  private Zeichner partner;

  public void setZeichner()
  { partner = new Zeichner();
  }

  public void setDoppelZeichner()
  { partner = new DoppelZeichner();
  }

  public void zeichne()
  { partner.zeichne();
  }
}
```

Eine innere Member-Klassen wird wie jede Klasse definiert. Jedoch dort, wo alle Members (z. B.Instanzvariablen) definiert werden.

Eine innere Klasse kann sich von anderen inneren Klassen ableiten.

Da Doppelzeichner die Methode «gib()» nicht kennt, schickt er diese Nachricht automatisch seiner äusseren Klasse.

«partner» ist eine Referenz auf die Instanz der inneren Klasse. Die Nachricht wird ganz normal zugestellt.

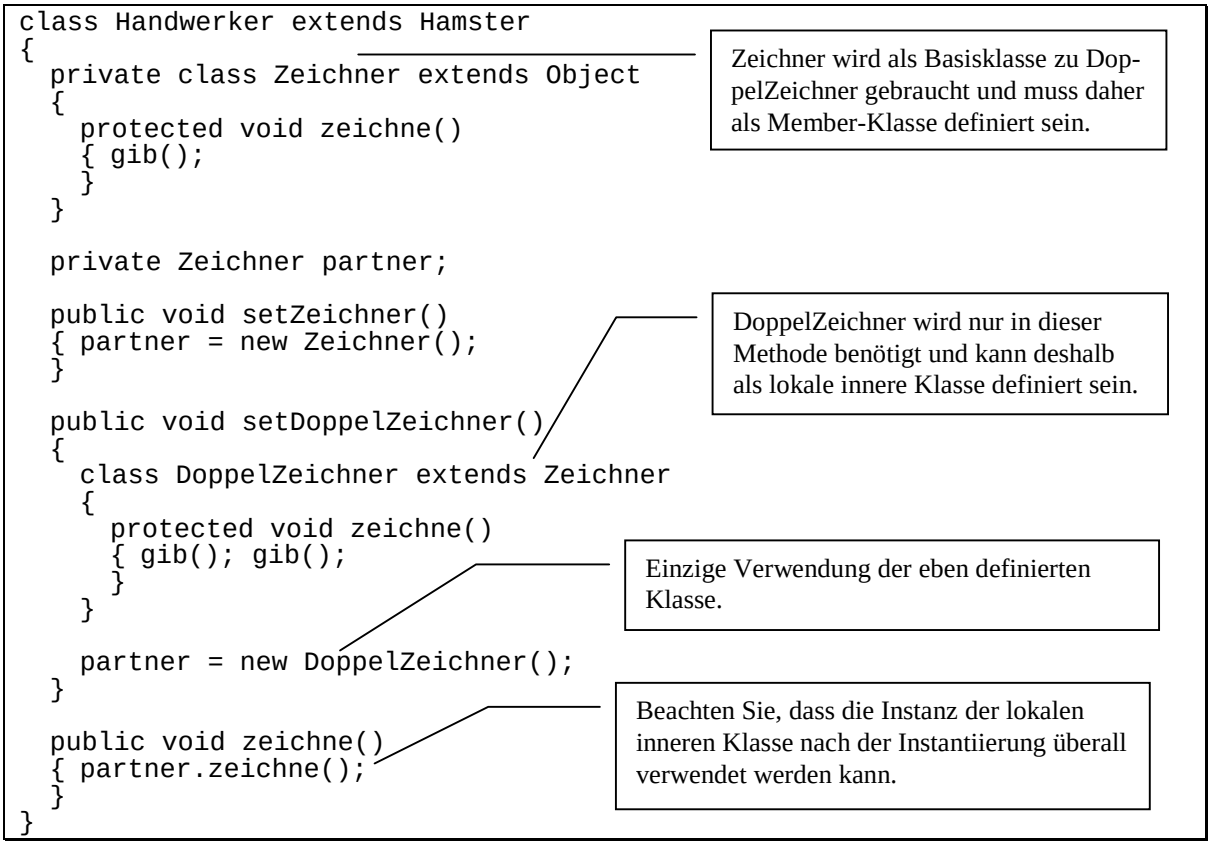
Wie Sie sehen, sieht alles ganz natürlich aus. Es gibt nur zwei Auffälligkeiten:
1. Dass die Methode «gib()» einer Instanz von Zeichner oder DoppelZeichner diese Nachricht an die Instanz des Handwerkers senden, der sie instantiiert hat.
2. Dass die inneren Klassen private sein können. Darauf komme ich anschliessend noch.
Wichtig ist zuerst einmal, dass sich der Handwerker nicht von Hamster ableitet. Deshalb kennt er die Methode «gib()» nicht selbst. Er würde sonst zuerst einmal seine eigene Methode «gib()» ausführen. Als Instanz einer inneren Klasse schickt er diese Nachricht daher automatisch an die Instanz seiner äusseren Klasse.

Bedeutung von private bei Instanzen von inneren Klassen

Instanzen der inneren und der äusseren Klassen können gegenseitig auf private Elemente (Members) unbeschränkt zugreifen. Sie können sich die Instanz der inneren Klasse wie einen Chef vorstellen, der in einem zentralen Raum sitzt. Rund herum sind die Büros der Angestellten. Zu diesen steht die Türe offen und sie können direkt miteinander uneingeschränkt kommunizieren. Die Mitarbeiter untereinander haben aber keinen speziellen Kontakt. Deshalb muss «zeichne()» protected sein, damit es in Doppelzeichner überschrieben werden kann. Zwischen den inneren Klassen oder Instanzen von inneren Klassen gelten die ganz normalen Zugriffsregeln. Innere Klassen haben also eine Doppelfunktion: Einfache Möglichkeit, der Instanz der äusseren Klasse Nachrichten zu schicken oder auf deren Instanzvariablen zuzugreifen einerseits. Möglichkeit zum Verstecken von inneren Klassen in einer äusseren Klasse. Letzteres hat natürlich auch Nachteile: Sie können ausserhalb des Handwerkers keine Unterklassen von Zeichner bilden.

Lokale innere Klassen

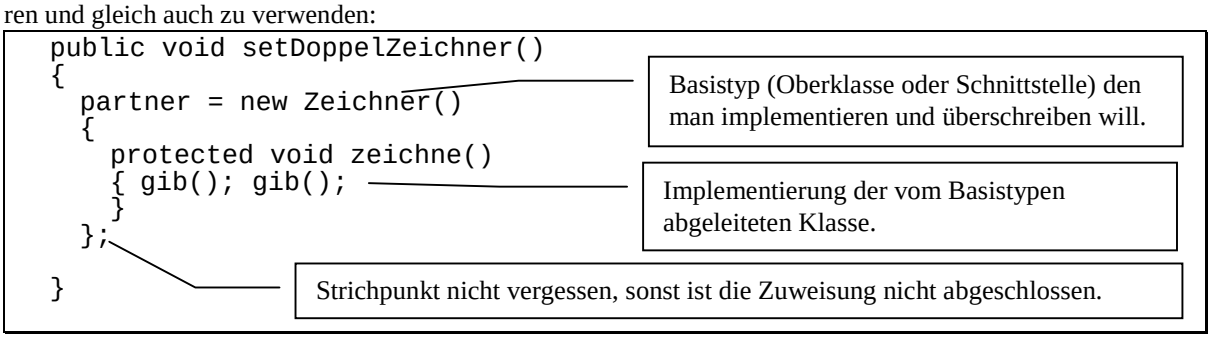
Ihnen ist bekannt, dass es nebst den Instanzvariablen als Element (engl. Member) der Klasse auch lokale Variablen als Element einer Methode oder eines Blockes innerhalb einer Methode gibt. So wie Member-Klassen Typendefinitionen für die Klasse zum internen Gebrauch sind, so kann man lokale innere Klassen definieren, wenn die Typangabe nur in einer Methode benötigt wird.
Dazu zurück zu unserem Beispiel:



Wichtig ist, dass Sie unterscheiden zwischen der Verwendung der Klasse und dem Zugriff auf die Objekte. Die Verwendung der Klasse geschieht bei der Instantiierung. Der Zugriff auf die instantiierten Objekte ist anschliessend überall und immer möglich vorausgesetzt, man verfüge über eine Referenz auf das Objekt und es bestehe keine Zugriffsbeschränkung wie «private» oder «protected».

Namenlose innere Klassen

Im oben stehenden Beispiel ist es eigentlich unnötig, dass die Klasse DoppelZeichner einen Namen trägt. Tatsächlich ist es erlaubt, eine solche nur einmal verwendete Klassendefinition wie folgt in einem Schritt zu definieren und gleich auch zu verwenden:



Wie Sie sehen, ist die namenlose innere Klasse einfach eine abgekürzte Form der lokalen inneren Klasse.

Namenlose innere Klassen und Konstruktoren

Namenlose innere Klassen haben keinen Konstruktor. Sie rufen direkt den Konstruktor der Oberklasse auf. Wenn der Basistyp eine Schnittstelle ist, ist die Oberklasse «Object». Verwenden Sie Argumente beim Konstruktor, so wird der Konstruktor mit der Oberklasse mit der entsprechenden Signatur aufgerufen. Sie sehen: Namenlose innere Klassen sind für bequeme Programmierer sehr praktisch, wenn man sie mal versteht.
Was tut man aber, wenn man trotz allem gewisse Initialisierungen ausführen will? Beispiel: Ich möchte eine Klasse MehrfachZeichner ableiten, die in einer Instanzvariable festhält, wie viele Körner zu legen sind.

```
public void setMehrfachZeichner(final int anz)
{
  partner = new Zeichner()
  {
    private int anzahl = anz;

    protected void zeichne()
    {
      for(int i=0;i<anzahl;i++)
        gib();
    }
  };

}
```

.als «final» nicht neu definierbares Argument

Da «anz» als «final» definiert wurde, kann ich in der lokalen Klasse darauf zugreifen

Anzahl funktioniert wie irgend eine andere Instanzvariable auch.

Was eine einfache Initialisierung wie im Beispiel nicht ausreicht, kann man auf den Initialisierungsblock zurückgreifen (Siehe Definitionen und Initialisierungsblock, S. 5). Dieser ist auch bei inneren Klassen möglich.

Gut strukturierter Code bei anonymen inneren Klassen

Innere Klassen sind in Java beliebt bei der Implementierung von Benutzeroberflächen. Einige Entwickler verwenden in diesem Zusammenhang einen absolut unübersichtlichen Code. Hier ein beliebtes Beispiel:

```
setWindowListener( new WindowAdapter()
{
  public void windowClosing(WindowEvent event)
  {
    System.exit(0);
  }
}
);
```

Anonyme innere Klasse wird instantiiert und die Referenz gleich als Argument verwendet.

Ende der anonymen inneren Klassendefinition.

Schliessende Klammer des Methodenaufrufs

Störend an oben stehenden Beispiel ist vor allem das optische Auseinanderfallen der verschiedenen zusammengehörenden Element.
Das Gleiche hätte man einiges übersichtlicher codieren können:

```
ActionListener al;
al = new WindowAdapter()
{
  public void windowClosing(WindowEvent event)
  {
    System.exit(0);
  }
};
setWindowListener(al);
```

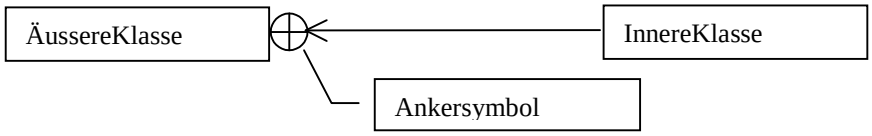
Anonyme innere Klasse wird instantiiert und die Referenz der lokalen Variable «al» zugewiesen.

Ende der anonymen inneren Klassendefinition.

Methodenaufruf mit der Referenz auf die anonyme innere Klasse

Innere Klassen und UML-Klassendiagramme

Innere Klassen sind eine Möglichkeit eine gefundene Lösung eleganter zu implementieren. Da die UML-Klassendiagramme in der Regel gedacht sind, eine Lösung unabhängig von der Implementation festzuhalten /zu spezifizieren) ignoriert man in der Regel ganz einfach, um schlussendlich mit oder ohne innere Klassen gearbeitet wird.
Will man doch einmal klar festhalten, dass eine Klasse eine innere Klasse ist, so kann man das Ankersymbol verwenden.



Bei anonymen Klassen empfehle ich einen Klassennamen zu verwenden, der auf den Basistyp hinweist. Zum Beispiel «NamenloserZeichner», «ZeichnerImplementation» oder «AbleitungVonZeichner».. Möglich ist auch den Namen zu wählen, den man genommen hätte, wenn man einen Namen hätte kreieren müssen.

Innere Klassen als Elemente der Klasse (static)

Wir haben gesehen, dass innere Klassen zwei Vorteile bieten: 1. Einfacher Nachrichtenaustausch mit der Instanz der äusseren Klassen; 2. Verstecken der inneren Klasse in der äusseren Klasse. Es gibt Fälle, wo man den ersten Vorteil nicht braucht, sondern nur eine Klasse in einer andern unterbringen will. Zum Beispiel weil die innere Klasse nur mit der Klasse (als statischem Objekt) zusammenarbeitet. In diesem Fall kann man die innere Klasse als static definieren. Oder: Die Klasse innerhalb einer Klassenmethode (static method) als anonyme innere Klasse definieren.

Spezialfragen zu inneren Klassen

Können innere Klassen statische Elemente (Members) haben, so wie das bei normalen Klassen der Fall ist?

Innere Klassen definieren nie und niemals statische Objekte. Innere Klassen haben deshalb keine static members (statischen Elemente). Sie müssen deshalb immer instantiiert werden, um sie in irgend einer Weise zu benutzen.

Kann man mit innere Klassen ausserhalb der Klasse arbeiten?

Ja. Angenommen, die innere Klasse Zeichner im Handwerker wäre sichtabr (nicht private), dann könnten Sie die innere Klasse von ausserhalb wie folgt instantiieren:

```
Handwerker hw = new Handwerker();
hw.init(2,2,1,1000);

Handwerker.Zeichner z;
z = hw.new Zeichner();

z.gib();
```

Der Typ der inneren Klasse: «ÄussereKlasse».«InnereKlasse»

Nachricht an Instanz der äusseren Klasse, seine innere Klasse zu instantiieren.

Sie sehen, die Typenbezeichnung der Inneren Klasse lehnt sich an die Schreibweise von Paketen an (Paketbezeichnung, dann ein Punkt, dann die Klassenbezeichnung, z. B. java.util.Observable). In gewisser Weise, können Sie eine äussere Klasse als Paket betrachten.

Die Instantiierung der inneren Klasse geht natürlich nicht ohne die Instanz der äusseren Klasse, sind doch die beiden automatisch miteinander assoziiert. Die von Java gewählte Lösung, dass man mittels Nachricht die Instanz der äusseren Klasse auffordert, eine neue Instanz der inneren Klasse zu erzeugen, leuchtet deshalb ein.

Der geübte Programmierer wird aber eine solch umständliche Codierung vermeiden. Er wird, falls er die Typenbezeichnung Zeichner ausserhalb des Handwerkers braucht, eine Klasse oder Schnittstelle «Zeichner» ausserhalb des Handwerkers anlegen:

```
interface Zeichner
{
  void zeichne();
}
```

Innerhalb des Handwerkers wird er zudem Fabrikmethoden für die benötigten Instanzen vom Basistyp Zeichner anlegen. Hier ein Beispiel, das anonyme innere Klassen verwendet:

```
class Handwerker extends Hamster
{
  Zeichner getZeichner()
  { return new Zeichner()
    {
      public void zeichne()
      { gib();
      }
    };
  }

  Zeichner getMehrfachzeichern(final int anz)
  {
    return new Zeichner()
    {
      anzahl = anz;

      public void zeichne()
      { for(int i=0;i<anzahl;i++)
          gib();
      }
    };
  }
}
```

Anonyme innere Klasse, die die Schnittstelle «Zeichner» implementiert.

Implementation der Methoden der Schnittstelle. Diese sind zwingend alle public.

Ende des Klassenkörpers gefolgt vom Strichpunkt für das Ende von «return»

Die bereits bekannte Nutzung eines Argumentes oder einer lokalen Variable, welche final (nicht neudefinierbar sein müssen)

Wie löst man Namenskonflikte?
Würde es die Methode «gib()» beim Zeichner selbst geben, so wäre noch nichts verloren. Wir können die eigene Methode umgehen, indem wir den Namen der Klasse, einen Punkt und das Codewort «this» voranstellen:

```
Handwerker.this.gib();
```

Sie erinnern sich vielleicht daran, dass man einen Namenskonflikt zwischen einer lokalen Variablen und einer Instanzvariablen durch Voranstellen von «this» lösen kann. Dies ist natürlich auch bei Instanzen der inneren Klasse der Fall. Angenommen wir hätten in zeichne eine lokale Variable «partner», eine Instanzvariable «partner» und es gebe die bereits vorhandene Instanzvariable «partner» in Handwerker. Dann würde

`partner`	sich auf die lokale Variable beziehen
`this.partner`	auf die Instanzvariable von Zeichner (Instanz der inneren Klasse)
`Handwerker.this.partner`	auf die Instanzvariable von Handwerker (Instanz der äusseren Klasse).

Man spricht in diesem Zusammenhang davon, dass die lokale Variable die gleichnamige Variable der inneren und äusseren Klasse verdecke. Elemente (Members) der inneren Klasse verdecken wiederum gleichnamige Elemente der äusseren Klasse.

Gibt es innere Klassen in inneren Klassen?
Ja das Konzept funktioniert in beliebiger Verschachtelung von Klassen. Zur Referenzierung von Klassen können Sie einfach den Namen der inneren Klasse beliebiger Stufe nehmen.

```
InnereKlasse.this.eineInstanzvariable;
```

Oder den vollständigen Namen der inneren Klasse, z. B. bei Namenskonflikten:

```
AeussereKlasse.InnereKlasse.this.eineInstanzvariable;
```

Aber übertreiben Sie es nicht mit inneren Klassen. Der Code wird nicht unbedingt verständlicher und bezüglich Speicherplatz des kompilierten Programmes oder Performance bestehen keine Vorteile.

Verdecken (hiding) und Überschreiben (overriding)

Bitte halten Sie hiding (verdecken) und oberriding (Überschreiben) auseinander. Verdeckte Elemente stehen immer zur Verfügung. Wir brauchen bloss den Kniff zu kennen, wie man das verdeckte Element aufdeckt. Bei verdeckten Instanzvariablen einer Oberklasse zum Beispiel, können wir einfach in Klammern den Typ der entsprechenden Oberklasse in Klammern voranstellen.
Beim overriding (wörtlich Niederreiten, Niedermachen) ist die überschriebene Methode hingegen ausgemerzt worden. Nur innerhalb von Methoden können wir mittels dem Codewort super auf überschriebene Methoden der direkten Oberklasse (dort implementierte oder geerbte Methoden) zugreifen. Ein super.super, oder irgend einen andern Trick, gibt es nicht.
Überschreiben hat grosse Bedeutng für die Polymorphie. Diese entsteht ja dadurch, dass die Nachrichten, die ein Basistyp versteht verschieden implementiert sein können. Und da kommt nebst der Möglichkeit, eine abstrakte Methode in diversen implementierenden Klassen unterschiedlich zu definieren dem Überschreiben der Implementierung in der Oberklasse eine wichtig Bedeutung zu.
Verdecken hat dagegen grundsätzlich nichts mit Polymorphie zu tun.

Merken Sie sich

- Dass es möglich ist Nachrichten in beide Richtungen zu schicken
- Die Bedeutung von **«this»** als **Referenz auf sich selbst** `add(this);`
- Dass eine Instanz der inneren Klasse automatisch Nachrichten an die Instanz seiner äusseren K lasse sendet.
- Dass äussere und innere Klasse gegenseitig auf private Elemente (Members) zugreifen können. Jedoch innere Klassen untereinander nicht.
- Dass innere Klassen keine statischen Elemente (static members) haben, aber selbst als static member zum statischen Objekt (der Klasse) gehören können, statt zur Instanz der äusseren Klasse.
- Dass lokale innere Klassen auch schnell, schnell ohne Angabe eines Klassennamens in einem Aufwisch instantiiert:
- und implementiert werden können
- Aber man das Befehlsende (Strichpunkt) nach dem Klassenkörper auf keinen Fall vergessen darf:

```
Zeichner z = new Zeichner()
{    void zeichne()
     {gib(); }
};
```

- Dass lokale innere Klassen auf Argumente und lokale Variablen zugreiffen können, wenn diese «final» sind.

Thema 3: Spezielle Kontrollstrukturen

Continue und Break

Case-Anweisung

Thema 4: Exceptionhandling

Thema 5: Pakete und Zugriffsoperatoren

Thema 6: Java Projekte realisieren

Um Projekte in Java zu realisieren oder auf bestehende Projekte zurückzugreifen, muss man ein wenig über den technischen Aufbau einer Java Applikation Bescheid wissen. Ich beschränke mich auf «Java 2» und lasse das Thema «Java 2 Enterprise Edition» zur Seite.
Jede Applikation besteht aus ein oder mehreren Dateien mit ausführbarem Code. Jede Programmiersprache kennt Regeln, wie man aus den Dateien mit dem Quellcode zu den Dateien mit dem ausführbaren Code galngt.

Entwicklungsumgebungen und Projekte

Will man ein Java Projekt realisieren, so wählt man zuerst eine Entwicklungsumgebung, die man für geeignet hält. Man startet dann die Entwicklungsumgebung und legt zuerst ein neues Projekt an. Hier zwei sehr unterschiedliche Produkte, die beid in Java programmiert sind im Vergleich:

Entwicklungsumgebung	**Eclipse**	**BlueJ**
Urheber	Grosse open-source Gemeinde	3 Universitäten
URL	http://www.eclipse.org	http://www.bluej.org
Grösse auf Festplatte	90 MB	4 MB
Neues Projekt anlegen	File – New – Project…	Project – New Project…
Darstellung	Zahlreiche Fenster bieten eine umfassende Ansicht aller Elemente der Pakete und Klassen.	Klassen und Instanzen (Objekte) werden übersichtlich und einfach als Icons UML-like dargestellt.
Programm starten Methoden aufrufen	Kontextmenü zur Klasse mit Programmeinstieg – Run – Java Application	Vom Kontextmenü zu Klassen und Instanzen aus lassen sich alle Methoden und Konstruktoren aufrufen.
Ausführbare Datei erzeugen	File – Export… (im Dialog Export mit 5 Fenstern) JAR-File, Klassen auswählen, Exportdatei angeben, Main-Class angeben, etc..	Project – Create jar File… (im Dialog Create jar File) Main-class auswählen Anschliessend Dateinamen angeben.
Klassenpfad definieren	Kontextmenü zum Projekt – Properties… (im Dialog Properties) Java Build Path, Add External Jar…	Tools – Preferences… (im Dialog Preferences) Register Libraries, Add…
Inkompatibilitäten		Läuft unter Windows weniger stabil

Theoretisch könnte man auch alles im Java Developer Kit (JDK) erledigen und einen beliebigen Editoren verwenden. JDK ist aber Kommando Zeilen orientiert und verlangt deshalb viel mehr Einarbeitungszeit.

Eine Klasse ausführen

Eiinfache Programme in Java bestehen aus Klassendateien, die sich zusammen in einem Ordner befinden. Klassendateien haben die Endung «class».
Beispiel für die Applikation Roboworld

```
Main.class
MainFrame.class
```

Beim Start von Java wird an eine bestimmte Klasse die Nachricht «main(: String[])» geschickt. Wichtig: diese Nachricht richtet sich an die Klasse (das statische Objekt, das in der Klasse definiert ist) und das Argument ist immer ein Array vom Typ String (eine Liste von Strings).
Die entsprechende Methode ist deshalb wie folgt zu definieren:

```
public static void main( String[] args)
{
  //hier steht der Code
}
```

beliebiger Name für das Argument

Man startet das Programm manuell, indem man den Namen der Klasse, die die Methode «main(: String[])» enthält als Argument angibt. Zum Beispiel unter Windows/DOS:

```
c:/irgendwo> java c:/projekte/Main
```

Befindet man sich bereits im richtigen Ordner, so geht es auch Kürzer

```
c:/projekte> java Main
```

Viele Entwicklungsumgebungen helfen diesen Aufruf elegant aus der Entwicklungsumgebung auszufürhen, ohne in die Betriebssystemumgebung zu wechseln. BluJ erlaubt sogar die Ausführung jeder Methode und jedes Konstruktors mit Hilfe des Kontextmenüs.
Nach dem Namen der Klasse kann man Argumente eingeben, die dann den String-Array bilden:

```
c:/projekte> java Main Arg0 1 zwei
```

Der übergebene Array besteht dann aus:

```
args[0] = "Arg0"
args[1] = "1"
args[2] = "zwei"
```

Man kann in Java einen solchen String-Array wie folgt schreiben:

```
{"Arg0","2","zwei"}
```

In «BlueJ» sind allfällige Argumente in dieser Schreibweise einzugeben.

Ablauf eines Java Programmes

Ein «Java 2»-Programm startet, wie bereits gesagt, mit der statischen Methode «main(:String[])». Einfachste Programme spulen diese Methode ab und sind dann zu ende.
Im Normalfall erzeugt die Methode «main(:String[])» mindestens ein Fenster. Dieses Fenster wartet dann in einem eigenen Thread auf Benutzereingaben, während der Thread «main» beendet wird. Ein solches Programm mit Fenstern endet im Normalfall durch Aufruf der Klassenmethode «exit(: int)» der Klasse System. Als Wert für das Argument übergibt man 0, wenn das Programm normal beendet wird. Beispiel:

```
System.exit(0);
```

Ein Programm kann auch Threads ohne Bildschirmoberfläche erzeugen. Es kann auch einen sogeanannten daemon Thread starten, dessen Aufgabe es ist im Hintergrund auf Ereignisse zu warten. In diesem Fall ist das Programm zu ende, wenn alle Threads (mit Ausnahme der daemon threads) abgelaufen sind.

(Teil-)Applikationen in Paketen organisieren

Teilapplikationen organisiert der erfahrene Entwickler in Paketen. Bei der Strukturierung der Pakete in Teilapplikationen verwendet man Domainnamen als erstes Strukturelement. Roboworld wurde z. B. in zwei Pakete strukturiert: «ch.heeb.helpmenu» und «ch.heeb.roboworld».
Bei jeder Klasse gibt man als erstes an, zu welchem Paket die Klasse gehört:

```
package ch.heeb.roboworld;
```

Die Klassendateien werden dann automatisch in den entsprechenden Unterordner verschoben. Der Aufruf des Programmes erfolgt anschliessend über:

```
c:/irgendwo> java c:/projekte/ch.heeb.roboworld.Main
```

Befindet man sich bereits im richtigen Ordner, so geht es auch kürzer

```
c:/projekte> java ch.heeb.roboworld.Main
```

Beachten Sie, dass der Klassenname mit vollständigem Pfad wie folgt lautet:

```
c:/projekte/ch/heeb/roboworld/Main.class
```

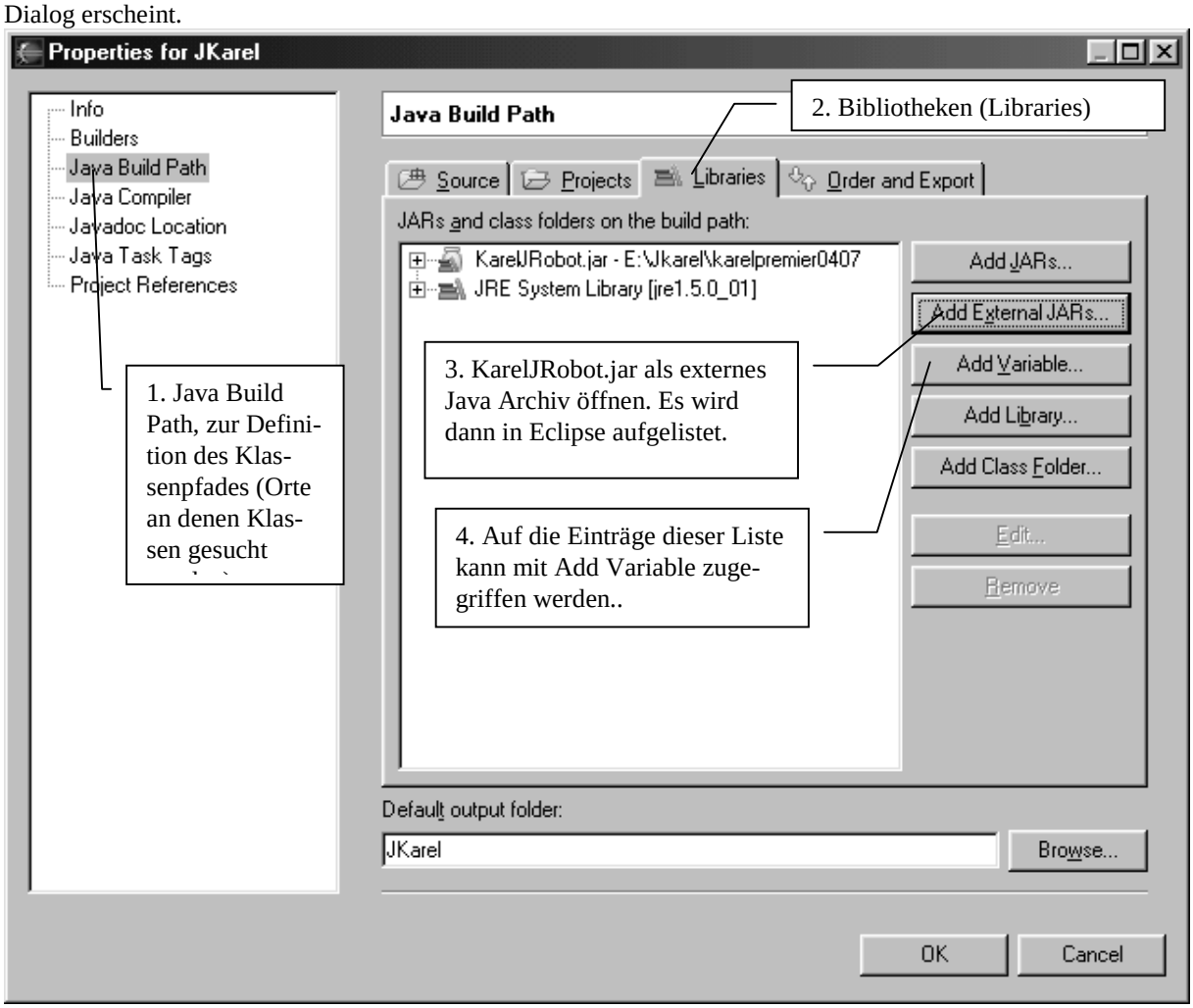
Doppelklickbare Applikationen in Archivdateien speichern

Eine Entwicklungsumgebung erlaubt es, eine Applikation in eine Java Archiv-Datei (Extension «.jar») zu exportieren. Wichtig ist, dabei die Klasse anzugeben, welche den Programmeinstieg («main(: String[])») enthält. Diese Angabe wird dann innerhalb des Archivs in einer sogenannten Manifest-Datei eingetragen. Eine Java Archiv lässt sich ganz normal entzippen.

Bestehende Archive verwenden

Die Verwendung bestehender Archivdateien, die Teilapplikationen oder Frameworks (Basisfunktionen) zur Verfügung stellen, ist nicht immer einfach. Für BlueJ wählen Sie :im Menü «Tools» den Befehl «Preferences…». Im Dialog «Preferences» wählen Sie das Register «Libraries» aus und klicken auf den Button «Add». Ein Öffnen-Dialog mit dem Titel «Select directory or jar/zip file» erscheint. Sie öffnen das gewünschte Archiv

und gelangen zurück zum Dialog «Preferences». Mit «OK» bestätigen und anschliessend das Programm neu starten.
In Eclipse wählen Sie zuerst das Projekt aus und dann im Menü «File» den Befehl «Properties…». Folgender Dialog erscheint.



Nehmen Sie die gewünschten Einstellungen vor. Praktischerweise bleiben externe Archive einmal verwendet in einer Variablenliste und sind mit «Add Variable…» schnell eingebaut.

Dateien mit Quellcode und Klassendateien

Dateien mit dem Quellcode haben die Endung «.java». Grundsätzlich können beliebig viele Klassen in einer Quellcode-Datei definiert sein. Aber: Sobald eine Klasse öffentlich ist, muss sie zwingend in einer Datei mit dem gleichen Namen und der Endung «.java» untergebracht sein. Gross- und Kleinschreibung ist dabei auch unter Windows oder MacOS 9 zwingend zu beachten.
Beispiele:

Enthaltene Elemente	**Quellcode**	**Klassendateien**
Eine öffentliche Klasse oder Schnittstelle mit Namen UrRobot	UrRobot.java	UrRobot.class
Klasse mit der öffentlichen Klasse A und den nicht öffentlichen Klassen B und C	A.java	A.class B.class C.class
Klasse mit den nicht öffentlichen Klassen A, B und C	Beliebig.java	A.class B.class C.class
Öffentliche Klasse A mit den inneren Klassen B und C	A.java	A.class A$B.class A$C.class

Enthaltene Elemente	Quellcode	Klassendateien
Nicht öffentliche Klasse A mit den inneren Klassen B und C und einer anonymen inneren Klasse	Beliebig.java	A.class A$B.class A$C.class a$1.class
Klasse A im Package	c:/projekte/A.java oder c:/projekte/ch/heeb/roboworld/A.java	c:/projekte/ch/heeb/roboworld/A.java

Übungen

Installieren Sie BlueJ und/oder Eclipse. Erstellen Sie ein neues Projekt. Nehmen Sie das Java Archiv «KarelJ-Robot.jar» in den Klassenpfad auf. Erzeugen Sie ein Paket «karel». Definierren Sie analgo dem Muster in der Datei «StairClimber.java» ein eigenes Roboterprogrämmchen. Ordnen Sie die Klasse dem Paket «karel» zu. Führen Sie das Roboter-Programmchen aus.
Erstellen Sie ein ausführbares Java-Archiv. Verlassen Sie die Entwicklungsumgebung. Starten Sie das Archiv per Doppelklick.

Merken Sie sich

- Es gibt einige gute, kostenlose Entwicklugnsumgebungen für «Java 2». Welche das Erstellen und Testen einer «Java 2»-Applikation unterstützen.
- Einsitegspunkt ist die Klassenmethoder: `public static void main(String[] args);`
- Applikationen ruft man wie folgt auf: `java NameDerKlasseMitMain Arg0 Arg1 Arg2`
- Es lassen sich doppelklickbare jar-Archive erstellen und ausführen: `Hamstersimulator.jar  Roboworld.jar`
- Java muss alle benötigten Klassen finden. Separat erstellte Archive sind daher im Klassenpfad zu verzeichnen.

Thema 7: Daten speichern und auf gespeicherte Daten zugreifen (nur mit Karel J Robot)

Mit «Java 2» können Sie Daten aus diversen Quellen (Dateien lokal oder im Internet, TCP/IP-Port, Arbeitsspeicher) herauslesen oder an den selben Ort hinschreiben (senden). Dieser Teil des Java-API bietet diverse zusätzliche Möglichkeiten. Von diesen ist die Übersetzung von und in unterschiedliche Zeichensätze die wichtigste.
Die hier benötigten Teile des API befinden sich in den Paketen «java.io» und «java.net». Stellen Sie deshalb oben alle Klassen die Sie schreiben die impotbefehle

```
import java.io.*;
import java.net.*;
```

Was wird gespeichert?

Der Computer behandelt Daten Bit und Byteweise. Beim Speichern und Lesen geht es um Byte (= 8 bit). Sie können Dateien, die geschrieben wurden oder die sie einlesen in einem Editoren einsehen oder auf den Bildschrim ausgeben. Die meisten Bytes werden dann von Ihrem Betriebssystem als lesbares Zeichen angezeigt (Zeichen 33 bis 255), als Steuerzeichen angezeigt (32 als Leerschlag, 9 als Tabulator, 12 als Zeilenvorschub, etc.) oder ignoriert. Die Bytes ab Nr. 127 werden von jedem Betriebssystem unterschiedlich interpretiert.
Es ist wichig, dass Sie zwischen den Bytes und den Zeichen, als die sie ein Betriebssystem interpretiert unterscheiden. Bytes sind einfach Folgen von 8 bits zwischen 0000 0000 (Dezimal 0) und 1111 1111 (Dezimal 255). Zeichen sind in Java vom Typ «char» (character). Es handelt sich dabei um ganze Zahlen zwischen 0 und 65 000 (einen 2 byte grosse Ganzzahl ohne Vorzeichen, engl.: 2 byte unsigned integer).
Beispiele

Java Zeichen			Ein-/Ausgabe als Byte	
Konstanten im Code		Wert (dezimal), der in Java Variablen gespeichert ist	Binäre Darstellung des Byte	Darstellung auf Bildschirm, Tastatur oder im Editor
Normal	Hexadezimal			
`'\t'`	`'\0009'`	9	0000 1001	Tabulator
`'\n'`	`'\000A'`	10	0000 1010	neue Zeile
`'\r'`	`'\000D'`	13	0000 1101	Return
`' '`	`'\0020'`	32	0010 0000	Leerschlag
`'A'`	`'\0041'`	65	0100 0001	A
`'¦'`	`'\00A6'`	166	1010 0110	¦

Die Zeichen ab 128 sind Betriebssystemabhängig. Insbesondere unter Windows stimmen Java-Konstante und Bildschirmausgabe oder Eingabe nicht überein, wenn Java im DOS-Fenster ausgeführt wird.

Im folgenden werden wir diverse Möglichkeiten kennenlernen, um unterschiedliche Daten (auch Zeichen) von Java in Bytes zu übersetzen und umgekehrt Bytes in Daten in Java.

Speicherung von Schrift im Computer

Über die Jahre haben sich verschiedene Standards für die Speicherung von Schrift und Zeichen im Computer herausgebildet

ascii

American Standard Code for Information Interchange definiert die ersten 128 Zeichen (7-Bit) einheitlich zum zweck des Datenaustauschs.

ANSI

Das American National Standards Institute definiert die ersten 256 Zeichen (8-Bit) einheitlich, wobei die ersten 128 dem ascii-Standard entsprechen. ANSI wird unter Windows, Unix und im Internet verwendet.

Unicode

Mindestens 2 Byte (32 Bit oder 36 Bit) Speicherung von Zeichen. Dies erlaubt die Berücksichtigung der verschiedenen weltweit verwendeten Schritsysteme (Kyrillisch, Griechisch, Arabisch, Hebräisch, Chinesisch, etc.). Die ersten 256 Zeichen entsprechen dem ANSI Standard. Die ersten rund 65 000 Zeichen (32 Bit), die auch von Java unterstützt werden decken die lebenden Sprachen ab. Die weiteren decken sämtliche Schriftsysteme ab.

utf-8

UTF-8 (Abk. für 8-bit Unicode Transformation Format) definiert einen Standard für den Austausch von Unicode Zeichen. Java unterstützt utf-8.

Fehlerbehandlung beim Speicherzugriff

Beim Zugriff auf Dateien kann es natürlich immer zu Fehlern kommen, die vom Betriebssystem herrühren: Eine Datei exisitiert nicht, eine Datei kann mangels Schreibzugriff nicht erstellt werden, eine Datei ist bereits geöffnet, etc.. Diese Fehler des Typs «java.io.IOException» sind Exceptions, die Sie zwingend abfangen oder weiterwerfen müssen.

```
try
{
  // hier kommen die Dateizugriffe
}
catch(IOException ioe)
{
  // hier der Code für den Fall von Problemen
}
```

Falls im folgenden Codeabschnitte gezeigt werden, so gehören diese in eine solche try-catch-Kontrollstuktur, auch wenn diese nicht immer aufgeführt ist. Falls Sie diese Kontrollstuktur vergessen erfolgt die Meldung:

```
Befehl sowie IOException must be caught
```

Wie bringe ich Daten rein ins Programm und raus in die Welt?

In Java brauche ich dazu ein Objekt, dass sich um eine Datenquelle oder eine Datensenke (Ort der Daten entgegennimmt) kümmert. Viele Objekte (zum Beispiel eine Datei) kommen gleichzeitig als Datenquelle und Datensenke in Frage.
Hier ein Überblick über die Klassen von Java, die sich um Datenquellen und Datensenken kümmern (Dargestellt als CRC-Karten):

java.io.File	
Datei oder Verzeichnis erstellen. Verfügbarkeit von Dateien prüfen. Eigenschaften von Dateien oder Verzeichnissen abfragen.	Dateisystem des Betriebssystems URI (Uniform Resource Identifiers), dies können URLs sein (Uniforme Ressource Locators z. B. http://www.heeb.ch) oder URN (Uniform Resource Names, z. B. info@heeb.ch)

java.io.FileInputStream	
Datei einlesen.	File Datei im Dateisystem des Computers

java.io.FileOutputStream	
Datei ausgeben.	File Datei im Dateisystem des Computers

java.io.RandomAccessFile	
Java Objekte und einfache Datentypen aus Datei lesen oder gleichzeitig lesen und schreiben..	File Datei im Dateisystem des Computers

Zu diesen vier Klassen sehen Sie ein einfaches Beispiel zuerst für Output und dann für Input. Das Beispiel zeigt nicht das RandomAccessFile, welches gleichzeitig das Lesen und das Schreiben erlaubt.
(innerhalb des try-Blockes):

```
File f = new File("Test.txt");
FileOutputStream fos = new FileOutputStream(f);
int i = 65;
fos.write(i);
i = 166;
fos.write(i);
fos.write('A');
fos.write('¦');
fos.write('\u0041');
fos.write('\u00A6');
fos.close();
```

Stellt Datei zum Schreiben zur Verfügung.

Byte wird als Ganzzahl vorgegeben.

Byte wird als normale Zeichenkonstante vorgegeben.

Byte wird als hexadezimale Zeichenkonstante vorgegeben.

```
FileOutputStream fos = new FileOutputStream("Test.txt");
fos.write('\u0041');
fos.write('\u00A6');
fos.close();
```

Alternative ohne ein Objekt vom Typ File.

Durch das Speichern, haben wir sichergestellt, dass eine Datei "Test.txt" im aktuellen Ordner existiert und können darauf zurückgreifen.

```
File f = new File("Test.txt");
FileInputStream fis = new FileInputStream(f);
int zch = fis.read();
while(zch != -1)
{
  // zch verarbeiten

  zch = fis.read();
}

fis.close();
```

Stellt Datei zum Lesen zur Verfügung.

read() gibt -1 zurück, wenn das Dateiende erreicht ist.

hier wird das gelesene Zeichen verarbeitet.

Nächstes Zeichen lesen.

Etwas anders funktionieren Netzwerkzugriffe. Die Objekte der Klassen, die sich um Netzwerkzugriffe verfügen, besitzen eine Fabrik-Methode «getInputStream()» respektive «getOutputStream()» um einen InputStream (Datenquelle) oder OutputStream (Datensenke, Datenausgabe) zu erhalten.

java.net.URL	
Eigenschaften abfragen Stellt Ressource als Datenquelle («openStream()») zur Verfügung.	Alle verfügbaren Ressourcen im Internet

Beispiel
(Innerhalb von try-Block):

```
URL res = new URL("http://www.heeb.ch/index.htm");
InputStream fis = res.openStream();
int zch = fis.read();
int cnt = 0;
while(zch != -1)
{
  zch = fis.read();
  cnt++;
}
System.out.println(""+cnt);
fis.close();
```

InputStream zu einer Ressource im Internet.

Das Beispiel zählt einfach die Anzahl Bytes.

Resultatusgabe auf die Standardausgabe.

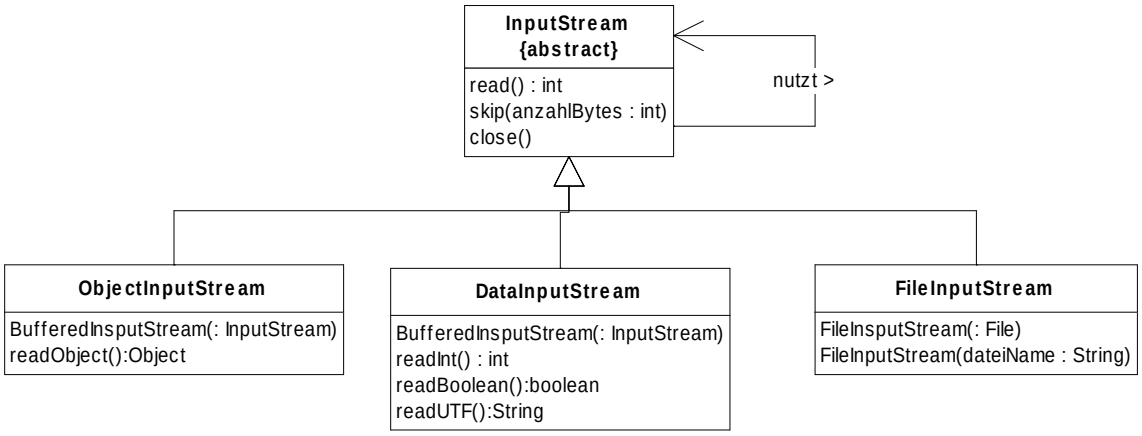
Über ein Socket sind Ein- und Ausgabe möglich. Auf diese Datensenke, respektive Datenquelle kommen wir im nächsten Kapitel zurück.

java.net.Socket	
Eigenschaften und Verfügbarkeit abfragen. Stellt einen IP-Port als Datenquelle («getInpotStream()») oder Datensenke («getOutputStream()») zur Verfügung.	Verfügbare Sockets. Ein Socket besteht aus der IP-Adresse und der Portnummer eines Computers.

Im Ergebnis brauchen wir immer einen InputStream oder OutputStream als Grundlage zur Nutzung von Datenquellen und Datensenken.

Zusatzfunktionen zur Ein- und Ausgabe

Es gibt zahlreiche Klassen, deren Instanzen bieten Zusatzfunktionen für die Ein- und Ausgabe. Das Grundprinzip zum Hinzufügen dieser Zusatzfunktionen folgt dem Designmuser Dekorator.



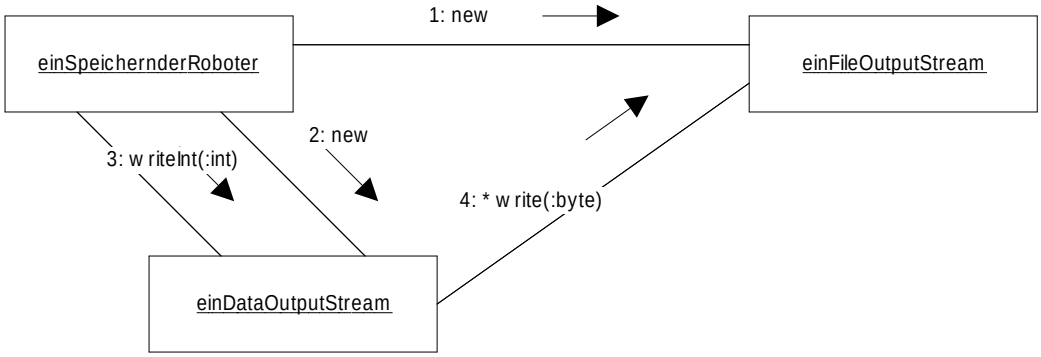
Ein erstes Objekt (zum Beispiel eine Instanz von FileInputStream) stellt eine Datenquelle zur Verfügung, mit der einfachen Möglichkeit, Byte für Byte Daten einzulesen. Weitere Klassen nutzen einen bestehenden InputStream und stellen zusätzliche Funktionalität zur Verfügung. Dank dem Designmuster Decorator können fast beliebig zusätzliche Funktionalitäten angehängt werden. Die Objekte bilden dann eine Art Kette. Einige Funktionalitäten beinhalten zusätzliche Operationen, zum Einlesen eines Objektes, von Zahlen oder Zeichenketten (String). Diese Klassen bilden den Schluss der Kette.
Eine Übersicht wichtiger Klassen

Funktion	**Klassenname** **setzt Speicherung voraus mit**	**Typische Operation**
Zeichenketten (String) oder **einfache Datentypen**, wie int, boolean, double einlesen	**Data**InputStream Voraussetzung: DataOutputStream	readUTF() : String readInt() : int readBoolean() : boolean
Objekte, assozierte Objekte und Informationen zu deren **Klassen** einlesen (sog. Objekt Deserialisierung)	**Object**InpoutStream Voraussetzung: ObjectOutputStream, keine Änderung an den Klassendefinitionen	readObject() : Object
Entzippen von **gezippten Dateien**	**java.util.zip.Inflater**InputStream	Typisches Beispiel einer zwischen-

	Voraussetzung: Speicherung mit java.util.zip.**Deflater**OuputStream	schaltbaren zusätzlichen Funktionalität ohne zusätzliche Operationen

Übung: Zeichenketten und Zahlen speichern und wieder einlesen

Programmieren Sie einen Roboter, der Zeichenketten oder Zahlen in eine Datei speichern. Anschliessend soll ein zweiter Roboter diese Zahlen aus der Datei herauslesen und etwas damit tun. Ihrer Phantasie sind dabei keine Grenzen gesetzt:
Der erste Roboter legt Zahlen (int) ab und der Zweite legt die entsprechende Anzahl Beeper oder geht entsprechend viele Schritte.
Der Speichervorgang folgt folgendem Sequenzdiagramm:



Die Schritte 1 bis 3 müssen Sie programmieren. Den Schritt 4 ist bereits in der Klasse «DataOutputStream» implementiert. Sie müssen also etwa folgenden Code verwenden (im try-Block):

```
OutputStream os = new FileOutputStream("Zahlen.dat");
DataOutputStream dos = new DataOutputStream(os);
dos.writeInt(5);
dos.close();
```

Aufgaben

Zeichnen Sie ein Kollaborations- oder Sequenzdiagramm für das Einlesen der Zahl.
Implementieren Sie die Methoden für das Auslesen und Einlesen der Zahl und Testen Sie Ihre Lösung.
Öffnen Sie die Datei «Zahlen.dat» mit einem Editor oder Textprogramm.

Zusatzaufgabe

Schalten Sie beim Speichern und wieder auslesen einen DeflaterOutputStream, respective InflaterInputStream dazwischen. Zeichnen Sie das zusätzliche Objekt im Kollaborations- oder Sequenzdiagramm ein.

Übung: Roboter speichern und wieder herstellen

Instantiieren Sie einen Roboter, speichern Sie den Roboter in eine Datei. Bewegen Sie den Roboter weg. Stellen Sie einen neuen Roboter aufgrund der gespeicherten Informationen wieder her.
Wichtig: Nicht jedes Objekt kann man speichern. Sie müssen die Speichermöglichkeit in der Klassendafiition wie folgt vorbereiten:

```
class BasisRoboter
{
  public BasisRoboter()
  {
    super();
  }
}
```

ein öffentlicher argumentloser Konstruktor muss in der Oberklasse definiert sein, java diesen automatisch aufruft.

Von dieser Klasse leitet man die Klasse mit den speicherbaren Instanzen ab.

```
class SpeicherbarerRoboter extends BasisRoboter implements Serializable
{


}
```

java.io.Serializable deklariert keine Methoden, ist aber notwendig.

Übung: Roboterpärchen speichern und wieder herstellen

Schon einiges kniffliger

Thema 8: Rechnen mit «Java 2»

Fliesskommazahlen (double, float)

Bit-Operationen

Präzises rechnen mit «Java 2»

Kapitel 2: Multi-Threding und verteilte Programme

Thema 9: Hamster arbeiten parallel

Thema 10: Hamster nehmen Rücksicht aufeinander

Thema 11: Hamster arbeiten im Netzwerk zusammen

