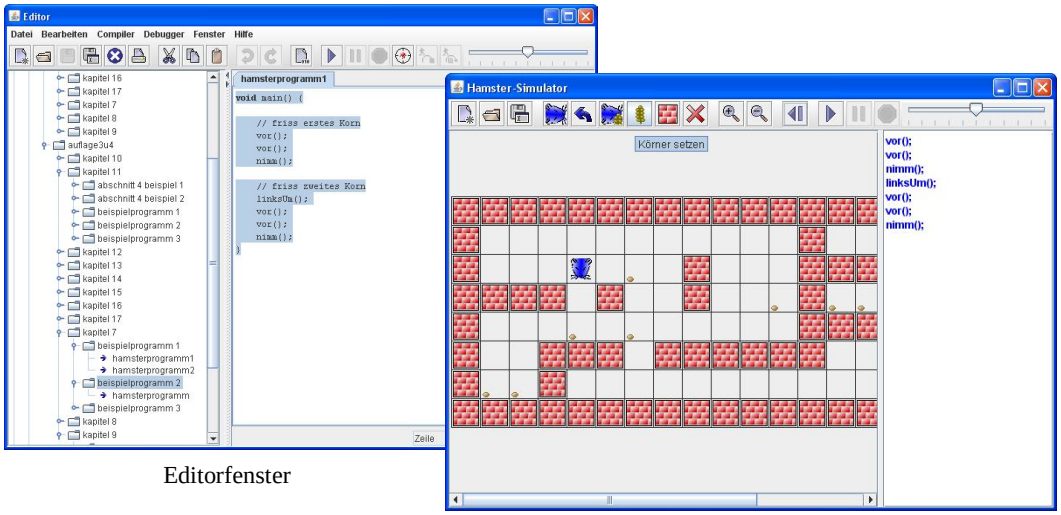
Einführung in die Programmierung mit dem Hamster-Simulator

Der Hamster-Simulator

Der Hamster-Simulator ist ein Programm, mit dem Schüler und Studenten Grundlagen der Programmierung spielerisch lernen können. Der Simulator wurde an der Universität Oldenburg entwickelt. Man kann ihn auf der Seite http://www.java-hamster-modell.de herunterladen. Der Hamster ist in der Programmiersprache Java geschrieben, deshalb braucht man zum Betrieb des Hamster-Simulators eine Java-Laufzeitumgebung. Einen Downloadlink für Java gibt es auf der Hamster-Simulator-Homepage.
Java und damit auch der Hamster-Simulator haben die gleiche Syntax (gleiche Schreibweisen) wie C++ oder C#. Deshalb können wir den Hamster als Einführung in die Programmierung verwenden, ohne nachher alles umlernen zu müssen.

Der Hamster-Simulator besteht aus zwei Fenstern, dem Editorfenster und das eigentliche Simulatorfenster. Im Editorfenster können die Programme erstellt werden, die der Hamster auf dem Territorium im Simulatorfenster durchführt:



Editorfenster

Simulatorfenster

Die Vorgehensweise bei der Arbeit mit dem Hamster-Simulator ist immer gleich. Man erstellt ein Programm und ein Territorium, kompiliert das Programm und führt es aus. Das Programm wird dabei im Editor immer zwischen den geschweiften Klammern in der Main-Methode erzeugt:

```
void main(){
  // hier wird das Programm erstellt
}
```


Aktionen des Hamsters:

Der Hamster kann die folgenden Aktionen ausführen:

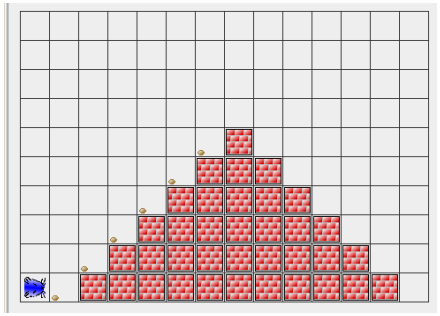
`vor();`	Der Hamster geht einen Schritt nach vorne.
`linksUm();`	Der Hamster dreht sich um 90° nach links.
`nimm();`	Der Hamster nimmt ein Korn auf.
`gib();`	Der Hamster legt ein Korn auf das Territorium.

Für die Programmierung in Java gibt es einige Regeln:

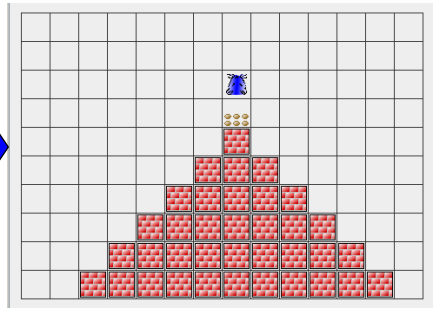
- Jeder Befehl muss mit einem Strichpunkt ; enden.
- Auf die Groß- und Kleinschreibung kommt es an.
- Java erwartet nicht einen Befehl pro Zeile. Trotzdem sollte man seinen Quelltext sinnvoll strukturieren. Eine solche Möglichkeit ist im Beispiel dargestellt. Wie so oft gilt also auch hier: *Weniger* (Ausdrücke pro Zeile) *ist mehr* (Übersichtlichkeit). Nach einer geschweiften Klammer { sollte um 2 oder 4 Stellen nach rechts eingerückt werden.

Aufgabe 1:

- Lege im Ordner „Eigene Dateien“ einen Unterordner „Hamsterprogramme“. Starte den Hamster-Simulator. Platziere das Editor- und Simulatorfenster so, dass sie sich nicht überlappen (u. U. Größen anpassen!).
- Erzeuge das unten bei der Anfangssituation abgebildete Territorium (14 x 10) und speichere es unter dem Namen „berg“.
- Schreibe ein Programm (Typ: „Imperatives Programm“), so dass die unten bei der Endsituation abgebildete Situation am Programmende entsteht. Speichere das Programm ebenfalls unter dem Namen „berg“. Die beiden Dateien müssen zwar nicht gleich benannt sein, aber es ist hilfreich.
- Kompiliere und teste das Programm.



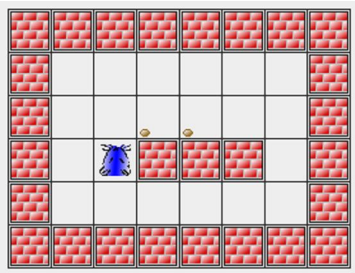
Anfangssituation



Endsituation

Aufgabe 2:

Der Hamster hat zu Beginn 4 Körner im Maul und soll die Körner in den vier Ecken des rechts abgebildeten Territoriums ablegen.



Sensoren des Hamsters

Wenn in Aufgabe 1 oder 2 nur ein Korn nicht da liegt, wo es sein sollte, bricht der Hamster-Simulator mit einer Fehlermeldung ab. Auch wenn man den Hamster auf ein Territorium loslässt, auf dem vorher nicht bekannt ist, wie viele Körner vorhanden sind, gibt es Probleme.
⇒ Der Hamster kann also bisher nicht auf seine Umgebung reagieren.
Im Hamster-Simulator wird dieses Problem durch die Sensoren des Hamsters gelöst. Der Hamster besitzt folgende Sensoren, die alle `true` (wahr) oder `false` (falsch) zurückliefern:

`vornFrei()`	Prüft, ob der Hamster *nicht* vor einer Wand steht.
`kornDa()`	Prüft, ob auf dem Feld, auf dem der Hamster gerade steht, mindestens ein Korn enthalten ist.
`maulLeer()`	Prüft, ob der Hamster ein Korn im Maul hat.

Auf die Informationen, die die Sensoren liefern, muss reagieren werden. Dies erfolgt beim Hamster durch die Auswahlanweisung (bzw. Testanweisung) `if`:

Beispiel in Java:	**Beispiel in Umgangssprache übersetzt**

`if(KornDa())`
`{`
`  nimm();`
`  vor();`
`}`
	Wenn ein Korn auf dem Feld vorhanden ist, mache folgendes:
	- Nimm das Korn auf
	- Gehe einen Schritt vorwärts.

Allgemein kann man die Auswahlanweisung folgendermaßen formulieren:

```
WENN Bedingung  DANN { Anweisungen für "Ja" }
                SONST { Anweisungen für "Nein" }
```

Wenn die Bedingung WAHR (`true`) ist, dann wird die Anweisung für "Ja" ausgeführt sonst - d.h. wenn die Bedingung FALSCH (`false`) ist - wird die Anweisung für "Nein" ausgeführt.

Formulierung in Pseudo-Java:	**Konkretes Beispiel:**

`if(Bedingung)`
`{`
`  Anweisungen für "Ja";`
`}`
`else`
`{`
`  Anweisungen für "Nein";`
`}`
	`if(vornFrei())`
	`{`
	`  vor();`
	`}`
	`else`
	`{`
	`  linksUm();`
	`}`

Beim Hamster benötigt man häufig die Information, ob der Hamster noch Körner im Mund hat. Da es keinen Sensor gibt, der `maulVoll` lautet, muss man den Sensor `maulLeer()` negieren (verneinen, umkehren). Dies geschieht mit einem Ausrufezeichen „!“, das in Java für unser Wort „nicht“ steht:

```
if(!maulLeer()){
  gib();
}
```

Abgesehen von der Negation kann man die folgenden Operatoren (genannt relationale Operatoren) bei der Formulierung von Bedingungen verwenden:

`==`	ist gleich mit	`!=`	Ist ungleich mit
`<`	ist kleiner als	`>`	Ist größer als
`<=`	Ist kleiner oder gleich mit	`>=`	Ist größer oder gleich mit

Diese Operationen brauchen wir nicht so häufig beim Hamster, jedoch sehr häufig beim „richtigen Programmieren“.

Schleifen

Die bisherigen Hamsterprogramme waren trotz der Sensoren sehr eintönig. Sie waren praktisch nur für ein Territorium geeignet. Auch kamen viele Anweisungen mehrfach vor, was die Programme lang und unflexibel gemacht hat.
Die Lösung dieser Probleme liefern die Wiederholungsanweisungen, die man auch Schleifen nennt. Es gibt mehrere Schleifentypen. Wir beginnen mit dem Typ, der sehr stark an die Auswahlanweisung erinnert, aber natürlich viel mächtiger ist:

While-Schleifen mit dem Hamster

Der Aufbau einer While-Schleife ist:

Umgangssprache	**In Pseudo-Java:**

SOLANGE *Bedingung* FÜHRE AUS
{
Anweisung 1
Anweisung 2
....
Anweisung n
} (Anweisung 1 … Anweisung n: Schleifenrumpf)
	`while (Bedingung)`
	`{`
	`  Anweisung_1;`
	`  Anweisung_2;`
	`  // …`
	`  Anweisung_n;`
	`}`

Zu Beginn der Schleife wird eine Bedingung auf ihren Wahrheitsgehalt geprüft. Wenn diese Bedingung WAHR (`true`) ist, wird der Schleifenrumpf betreten und die Anweisung(en) wird (werden) ausgeführt. Ist die Ausführungsbedingung FALSCH (`false`), so wird der Schleifenrumpf nicht betreten und das Programm mit der dahinter folgenden Anweisung fortgesetzt. Es kann demnach sein, dass die while-Schleife kein einziges Mal durchlaufen wird.

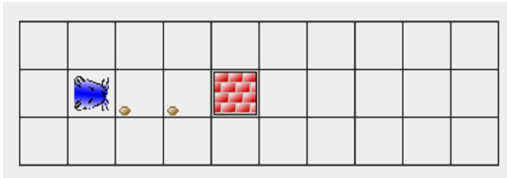
Wichtig: Man muss unbedingt darauf achten, dass die Schleife auch tatsächlich nach einer endlichen Anzahl von Durchläufen verlassen wird und keine Endlosschleife entsteht. Dies bedeutet, dass der Wahrheitswert der Ausführungsbedingung nach endlich vielen Durchläufen den Wert FALSCH annehmen muss. Erreicht wird das durch eine entsprechende Anweisung im Schleifenrumpf.

Beispiel:

```
void main() {

    if(kornDa())
    {
      nimm();
    }

    while(vornFrei())
    {
      vor();
      if(kornDa())
```



```
      {
        nimm();
      }
    }

    linksUm();
    vor();
}
```

In diesem Beispiel wird zunächst ein Korn aufgenommen, wenn es vorhanden. Danach läuft der Hamster solange vorwärts, bis er auf eine Wand trifft. Während dessen nimmt er ein Korn auf, wenn es vorhanden ist. An der Wand angekommen, macht er eine Linksdrehung und einen Schritt nach vorne.

Aufgabe 3: Geschachtelte Schleifen

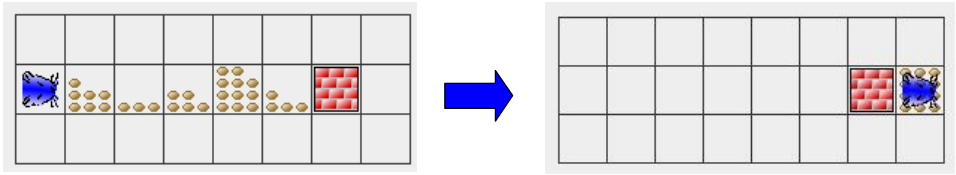
a) Was passiert, wenn das folgende Programm ausgeführt wird?

```
void main()
{
  while(vornFrei())
  {
    while(kornDa())
    {
       nimm();
    }
    vor();
  }
}
```

b) Baue ein Territorium, das vom folgenden Programm gelöst wird, und teste das Programm!

Aufgabe 4:

Schreibe ein Programm, das alle Körner hinter eine Mauer bringt, egal wie viele im Weg liegen. Verwende dazu eine `while`-Schleife.



Aufgabe 5:

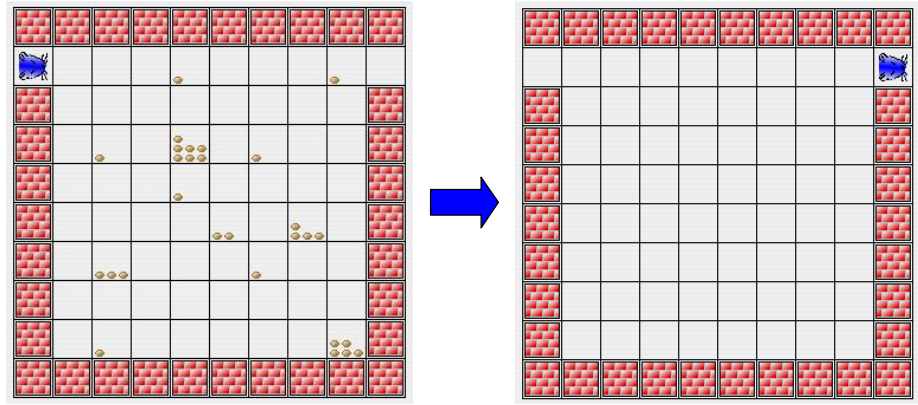
Der Hamster steht irgendwo in einem rechteckigen, geschlossenen Raum (mindestens 4x4 Kacheln) unbekannter Größe ohne innere Mauer. Er soll in irgendeine Ecke laufen und dort anhalten. Schreibe ein Hamster-Programm dafür.

Aufgabe 6:

Verändere das Programm aus Aufgabe 5, so dass der Hamster auf dem Weg zur Ecke alle Körner einsammelt, auf die er trifft.

Aufgabe 7: Der Allesfresser

Erzeuge folgendes Territorium (10x10), auf dem Du wahllos Körner verteilen kannst. Der Hamster soll alle finden, fressen und am Ende im Ausgang rechts oben sitzen.
Du kannst hierzu auch Teile des Programms aus Aufgabe 3 verwenden.



Aufgabe 8: Round Robin (Aufgabe für sehr Schnelle!)

Der Hamster soll zunächst auf einem Weg unbekannter Länge zu einer Mauer alle Körner fressen, die er findet.
Vor der Mauer beginnt nun die nächste Aufgabe: Der Hamster soll so oft um die Mauer tanzen, wie er Körner im Maul hat. Dabei soll er immer dann, wenn er gerade hinter der Mauer vorbeitanzt, ein Korn ablegen.
Teste Dein Programm mit unterschiedlichen Ausgangssituationen.

```
void main() {
    // sammle bis Mauer
    […]
    // in Ausgangsposition drehen
    […]
    // links herum tanzen
    […]
}
```