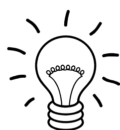
Kapitel 18 Jetzt kommt Bewegung ins Spiel

Lernziele:
In diesem Kapitel lernst du das Importieren von Java-Bibliotheksklassen.
Wiederholung:
Bei den Erweiterungen des BlueJ-Projekts wird die Mehrfachauswahl und ein Feld (Array) benötigt.

18.1 Mampfi bewegt sich in Blickrichtung

Für die Bewegung von mampfi gibt es mehrere Variationen. Im Moment muss für jeden Schritt eine der Cursortasten gedrückt werden. Eine andere Möglichkeit ist, dass sich Mampfi so lange bewegt, bis er an einem Hindernis, einer Mauer oder dem Rand des Spielfelds, anstößt. Durch einen Druck auf die Cursortasten wird die Blick- und damit die Bewegungsrichtung geändert.



Aufgabe 18.1
Kommen in der zweiten Variante auch Aufrufe der Methoden *NachNordenGehen*, *NachOstenGehen* usw. zum Einsatz? Werden zusätzliche Hilfsmittel benötigt? Welche Klasse soll für die automatische Bewegung sorgen?

Die Bewegung soll nun automatisch erfolgen, ein Methodenaufruf „von außen“ per Mausklick wird nur bei einer Richtungsänderung benötigt. Für jeden Schritt von mampfi ist ein Methodenaufruf einer der Methoden *NachNordenGehen*, *NachOstenGehen* usw. nötig. Jedoch muss intern für den Aufruf gesorgt werden. Eine Möglichkeit ist, über die Systemuhr Impulse („Ticks“) zu erhalten, und bei jedem Tick Methoden aufzurufen. Die Spielsteuerung sollte entsprechend ihrem Namen das Spiel steuern, d.h. dort sollten die Methodenaufrufe für die Bewegung stattfinden. Das Backend bietet dazu folgendes an:

a) In der Klasse STEUERUNGSANZEIGE gibt es eine Methode *TickerStarten*. Als Eingabewert benötigt sie ein Zeitintervall in Millisekunden. Findet beispielsweise der Methodenaufruf
```
steuerungsanzeige.TickerStarten(500)
```
statt, dann wird ein Taktgeber (Ticker) gestartet, der alle 500 ms = 0,5 s ein „Signal“, einen Taktimpuls, verschickt.

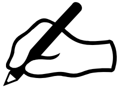
b) Wird nun in der Klasse SPIELSTEUERUNG eine Methode mit dem Methodenkopf
void Tick()
implementiert, so wird diese Methode bei jedem Taktimpuls ausgeführt. Füllt man den Rumpf der Methode *Tick* mit Methodenaufrufen (auch Zuweisungen sind möglich), so werden diese bei jedem Taktimpuls ausgeführt.

STEUERUNGSANZEIGE
int punkteStand int anzahlLeben int levelNummer String schriftFarbe String hintergrundFarbe
STEUERUNGSANZEIGE() void Anmelden(SPIELSTEUERUNG) void PunkteStandSetzen(int punkteStandNeu) void LebenSetzen(int lebenNeu) void levelSetzen(int levelNeu) void SchriftFarbeSetzen(String farbeNeu) void HintergrundFarbeSetzen(String farbeNeu) **void TickerStarten(int TickIntervall)** **void TickerAnhalten()**

Abbildung 1: Methoden *TickerStarten* und *TickerAnhalten* der Klasse STEUERUNGSANZEIGE

 © brichzin@tcs.ifi.lmu.de, 2007

Aufgabe 18.2



Im Folgenden ist der **Auszug** der Quelltexte zweier **verschiedener** Klassen SPIELSTEUERUNG abgebildet, die unterschiedliche Einsatzmöglichkeiten der Methode *Tick* zeigen.
Beschreibe knapp in Worten, was jeweils durch die Methode *Tick* erreicht wird.
Wenn du unsicher bist, kannst du deine Antwort mit der Lösung im Anhang vergleichen.

a)

```
class SPIELSTEUERUNG
{
  ...
  private MAMPFI  mampfi;
  private STEUERUNGSANZEIGE anzeige;
  ...

  public SPIELSTEUERUNG()
  {
       ...
        anzeige = new STEUERUNGSANZEIGE();
        ...
        anzeige.TickerStarten(1000);
        ...
  }

  ...

   /**  Die Methode Tick wird bei jedem Tick (Taktimpuls) ausgef&uuml;hrt,
   *     sofern der Ticker (Taktgeber) &uuml;ber den Methodenaufruf
   *     steuerungsanzeige.TickerStarten(int Taktintervall)
   *     gestartet wurde.
   */
   */
  public void Tick()
  {
        mampfi.NachOstenGehen();
  }


}
```

 © brichzin@tcs.ifi.lmu.de, 2007

b)
```
class SPIELSTEUERUNG
{
   ...
   private int richtungsauswahl; //Attribut zum Festlegen der Blidckrichtung
   private MAMPFI  mampfi;
   private STEUERUNGSANZEIGE anzeige;
   ...

   public SPIELSTEUERUNG()
   {
         // Attribute initialisieren
         ...
         richtungsauswahl = 0;
        ...
         anzeige = new STEUERUNGSANZEIGE();
         ...
         anzeige.TickerStarten(2000);
         ...
   }

  public void Tick()
   {
   richtungsauswahl = richtungsauswahl +1;
         if( richtungsauswahl == 4)
         {
               richtungsauswahl = 0;
         }

         switch (richtungsauswahl)
         {
               case 0:     mampfi.NachNordenBlicken();
                           break;
               case 1:     mampfi.NachOstenBlicken();
                           break;
               case 2:     mampfi.NachSuedenBlicken();
                           break;
               case 3:     mampfi.NachWestenBlicken();
                           break;
         }
    }
}
```

Nutze nun die Methode *Tick* für unser Projekt. Die Folgende Aufgabe gibt eine Hilfestellung.

Aufgabe 18.3
Führe in der Klasse SPIELSTEUERUNG folgende beiden Änderungen durch:

a) Über den Druck der Taste „S“ soll der Ticker gestartet werden. Experimentiere selbst im Laufe dieser Aufgabe, welches Taktintervall eine für dich passende Mampfi-Geschwindigkeit festlegt.
b) Im Rumpf der oben beschriebenen Methode Tick werden abhängig von der Blickrichtung die Methoden *NachNordenGehen*, *NachOstenGehen* usw aufgerufen. Damit die Punkte bei gefressenen Krümeln hochgezählt werden, muss auch die Methode *SpielzugAuswerten* aufgerufen werden. Das Auswerten des Spielzugs innerhalb der Methode *AufTasteReagieren* kann dann gelöscht werden.
c) Beim Testen wird dir auffallen, dass sich beim Drücken der Cursor-Tasten mampfi ruckartig bewegt. Suche die Ursache und verbessere entsprechend dein Programm.

Solltest du Tipps benötigen, findest du sie am Ende dieses Kapitels.

 © brichzin@tcs.ifi.lmu.de, 2007

18.2 Wo sind die Monster?

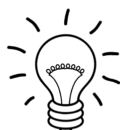
Im letzten Kapitel haben wurde die Klasse MONSTER erstellt. Wenn man nun ein Objekt der Klasse Spielsteuerung erzeugt, dann ist jedoch kein Monster zu sehen.

Aufgabe 18.4
Ergänze die Klasse SPIELSTEUERUNG so, dass beim Erzeugen eines Objekts neben Mampfi auch Monster im Labyrinth sind. Die Startposition von den Monstern kannst du beliebig wählen, musst jedoch sicherstellen, dass diese nicht auf einer Mauer liegt.

Solltest du Tipps benötigen, findest du sie am Ende dieses Kapitels.

18.3 Auch die Monster bewegen sich

Natürlich sollen sich auch die Monster bewegen. Aber nach welcher Strategie?



Aufgabe 18.5
Überlege dir verschiedene Strategien, nach denen sich die Monster bewegen. Achte dabei auf unterschiedlich „intelligente“ Bewegungskriterien.

Unter den sehr vielen unterschiedliche Möglichkeiten wird hier eine sehr einfache, zufallsgesteuerte Bewegung vorgestellt. Gerne ist es jedem überlassen eine intelligentere Bewegung der Monster zu programmieren. Dies wird jedoch im Skript nicht ausgeführt, sondern soll eine Herausforderung sein, die Schwierigkeit des Spiels zu steigern.

Bewegungsstrategie
Die Monster sollen sich zufallsgesteuert bewegen. Damit sie sich nicht in kleinen Kreisen bewegen, soll die Bewegungsrichtung bevorzugt in Blickrichtung erfolgen.
Dazu wird eine Zufallszahl aus der Menge {0, 1, 2, ..., 6} generiert.

- Ist die Zufallszahl 0, so soll das Monster nach Osten gehen.
- Ist die Zufallszahl 1, so soll das Monster nach Süden gehen.
- Ist die Zufallszahl 2, so soll das Monster nach Westen gehen.
- Ist die Zufallszahl 3, so soll das Monster nach Norden gehen.
- Ist die Zufallszahl 4, 5 oder 6, so soll das Monster in die aktuelle Blickrichtung gehen.

 © brichzin@tcs.ifi.lmu.de, 2007

Die Klasse Random

Man muss nicht immer das Rad neu erfinden: Aufgaben, deren Lösung für viele Nutzer von Bedeutung sind wie mathematische Funktionen, das Erzeugen von Zufallszahlen, Zeichnen von geometrischen Figuren, Elemente graphischer Benutzeroberflächen wurden bereits in Java programmiert. Diese Klassen wurden gesammelt und können können von jedem verwendet werden. Man nennt diese Klassen Bibliotheksklassen. Sie sind unter http://java.sun.com/javase/6/docs/api/ dokumentiert.

Eine Bibliotheksklasse ist `java.util.Random`. Sie ermöglicht es Zufallszahlen zu erzeugen. Ein Auszug des Klassendiagramms mit Erläuterungen zeigt Abbildung 2:

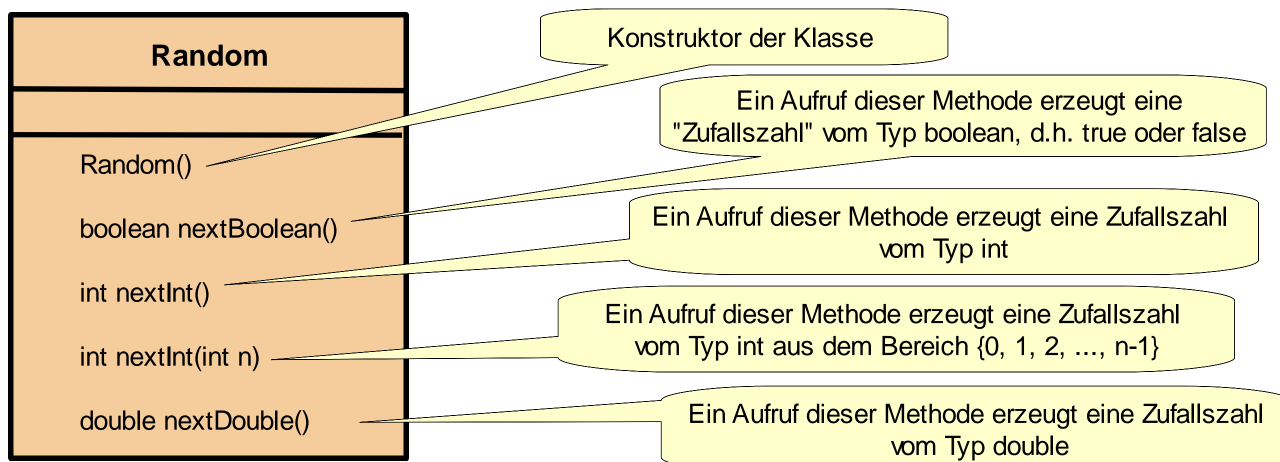


Abbildung 2: Wichtige Methoden der Klasse Random

Hinweis:

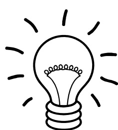
Beachte die Bibliotheksklassen haben eine andere Namenskonvention, sie werden nicht ausschließlich mit Großbuchstaben geschrieben, sondern nur der erste Buchstabe ist groß.

Um eine Bibliotheksklasse in eine eigene Klasse, in unserem Fall die Klasse SPIELSTEUERUNG, einzubinden, schreibt man vor dem Klassenkopf eine `import`-Anweisung

```
import java.util.Random;

class SPIELSTEUERUNG
{
    ...
}
```

Umsetzen der Bewegungsstrategie in der Klasse SPIELSTEUERUNG



Aufgabe 18.6

Überlege dir wie die oben beschriebene, einfache Bewegungsstrategie mit Hilfe eines Objekts der Klasse Random in der Klasse SPIELSTEUERUNG umgesetzt werden kann.

 © brichzin@tcs.ifi.lmu.de, 2007

Neben dem Importieren der Bibliotheksklasse Random (siehe oben) müssen noch folgende Punkte in der Klasse SPIELSTEUERUNG geändert werden, um die oben beschriebene Bewegungsstrategie zu realisieren.

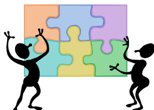
- Es muss ein Objekt der Klasse Random erzeugt und einem Referenzattribut zugewiesen werden. So ist es möglich über Methodenaufrufe dieses Objekts Zufallszahlen zu erhalten.
  In unserem konkreten Fall wird der Methodenaufruf
  ZufallszahlGenerator.nextInt(7)
  benötigt, da dieser eine Zufallszahl aus der Menge {0, 1, 2, ..., 6} erzeugt.
- Die Methode *Tick* muss um einen Abschnitt ergänzt werden der für die Bewegung der Monster sorgt.

SPIELSTEUERUNG
int punkteStand int leben int level int punktewertNormalerKruemel int punktewertPowerkruemel STEUERUNGSANZEIGE steuerungsanzeige LABYRINTH aktLabyrinth MAMPFI mampfi MONSTER[] monsterliste **Random ZufallszahlGenerator**
SPIELSTEUERUNG() void SpielzugAuswerten() void AufTasteReagieren(int taste) **void Tick()**

Abbildung 3: Zusammenfasung der Neuerungen in der Klasse SPIELSTEUERUNG

Aufgabe 18.7
Ergänze die Klasse SPIELSTEUERUNG entsprechend den formulierten Überlegungen und teste dann.
Solltest du Tipps benötigen, findest du sie am Ende dieses Kapitels.

18.4 Zusammenfassung



Aufgabe 18.8
In den Kapiteln wurde gezeigt, wie man eine Java-Bibliothek in eine eigene Klasse einbinden und nutzen kann. Fasse die Vorgehensweise knapp zusammen.

© brichzin@tcs.ifi.lmu.de, 2007

Anhang A: Tipps zu den Aufgaben

Lösung zu Aufgabe 18.2:
a)
Mampfi geht jede Sekunde einen Schritt nach Osten.

Hinweise:
- Sobald Mampfi vor einer Mauer steht bzw. am rechten Spielfeldrand wird zwar weiterhin die Methode *Tick* und damit auch die Methode mampfi.*NachOstenGehen* jede Sekunde aufgerufen, jedoch ändert sich dadurch nichts. Wir haben die Methode *NachOstenGehen* so programmiert, dass Mampfi nicht aus dem Spielfeld hinaus bzw. auf eine Mauer gehen kann.
- Der Aufruf erfolgt erfolgt jede Sekunde, da der Eingabewert bei folgendem Methodenaufruf 1000 ms = 1 s ist.

```
anzeige.TickerStarten(1000);
```

b)
Mampfi dreht sich im Uhrzeigersinn alle zwei Sekunde um 90° .

Erklärung:
Das Attribut richtungsauswahl wird in dem Konstruktor mit dem Wert 0 initialisiert und mit jedem Aufruf der Methode Tick um eins erhöht. Sobald jedoch der Wert 4 erreicht ist, wird der Wert von dem Das Attribut richtungsauswahl auf 0 zurückgesetzt. So hat richtungsauswahl beim Eintritt in die switch-Anweisung abwechselnd den Wert 0, 1, 2 oder 3 und zwar genau in dieser Reihenfolge. So blickt mit Hilfe der Methodenaufrufe in der Switch-Anweisung Mampfi nach Osten, dann nach Süden, dann nach Westen, dann nach Norden, dann wieder nach Osten usw.

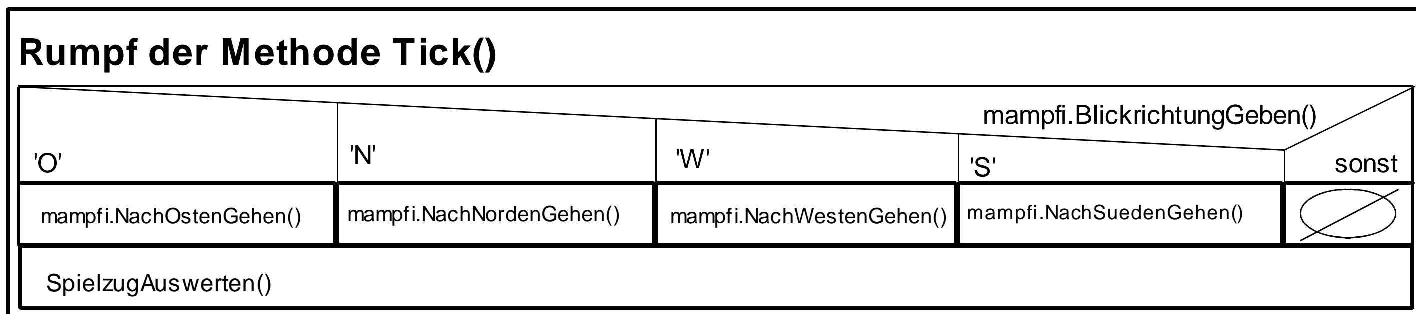
Tipps zu Aufgabe 18.3
zu a)
In der Methode *AufTasteReagieren* muss ein Fall (case) ergänzt werden. Die Nummern für die Tasten findest du im Anhang zu Kapitel 15.

zu b)
Für den Rumpf der methode *Tick* ist eine Mehrfachauswahl mit der Blickrichtung von Mampfi als Auswahlkriterium sinnvoll.
Da die Spielsteuerung die Blickrichtung von Mampfi nicht kennt, muss sie Mampfi entsprechend dem Sequenzdiagramm in der Abbildung rechts fragen.

zu c)
Die Cursor-Tasten haben nun eine andere Funktion. Da die Bewegung automatisch über den Ticker durchgeführt wird, muss beim Drücken der Cursor-Tasten nur die Blickrichtung von mampfi geändert werden. Das Gehen in die betreffende Richtung und die Spielauswertung ist in der Methode *AufTasteReagieren* falsch.

 © brichzin@tcs.ifi.lmu.de, 2007

Noch mehr Tipps zu Aufgabe 18.3 b)



Tipps zu Aufgabe 18.4

Wie Mampfi von der Spielsteueurng (im Konstruktor) erzeugt wird, so können auch Monster dort erzeugt werden. Natürlich müssen zuvor entsprechende Referenzattribute deklariert werden (siehe erweitertes Klassendiagramm unten links).

Hinweis:
Man wird spätestens in Kapitel 18.3 feststellen, dass es viel geschickter ist nicht vier einzelne Monster zu deklarieren, sondern ein Feld der Länge 4, indem die 4 Monster verwaltet werden (siehe erweitertes Klassendiagramm unten rechts). Solltest du dir unsicher sein im Umgang mit Feldern, so siehe in deinen Aufzeichnungen bzw. in Kapitel 6 und 9 nach.

SPIELSTEUERUNG
int punkteStand int leben int level int punktewertNormalerKruemel int punktewertPowerkruemel STEUERUNGSANZEIGE steuerungsanzeige LABYRINTH aktLabyrinth MAMPFI mampfi **MONSTER monster1** **MONSTER monster2** **MONSTER monster3** **MONSTER monster4**
SPIELSTEURUNG() void SpielzugAuswerten() void AufTasteReagieren(int taste) void Tick()

SPIELSTEUERUNG
int punkteStand int leben int level int punktewertNormalerKruemel int punktewertPowerkruemel STEUERUNGSANZEIGE steuerungsanzeige LABYRINTH aktLabyrinth MAMPFI mampfi **MONSTER[] monsterliste**
SPIELSTEURUNG() void SpielzugAuswerten() void AufTasteReagieren(int taste) void Tick()

 © brichzin@tcs.ifi.lmu.de, 2007

Tipps zu Aufgabe 18.7

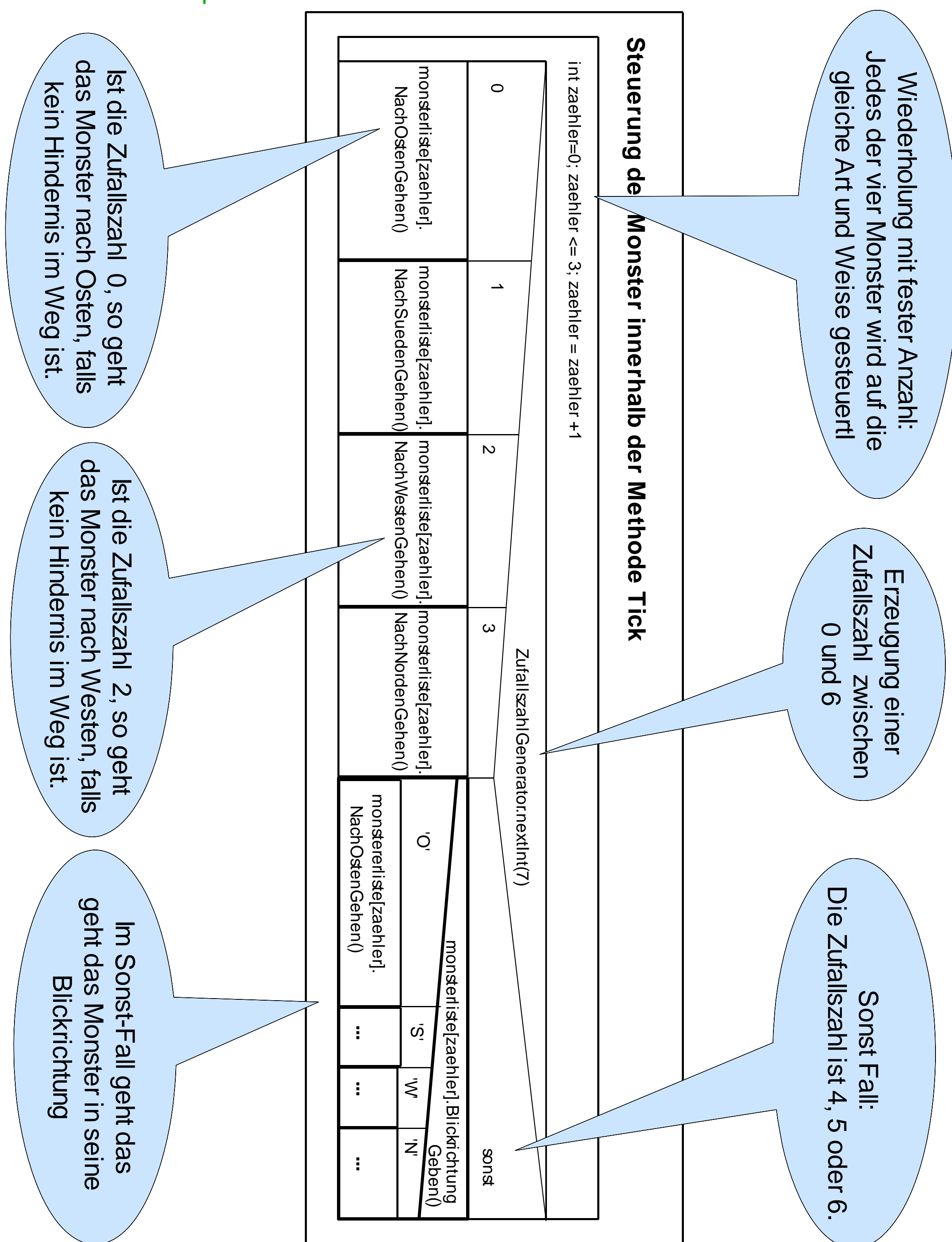
Die Deklaration eines Zufallszahlengenerators erfolgt über die Quelltestzeile

```
private Random zufallszahlGenerator;
```

Das Referenzattribut zufallszahlGenerator wird imKonstruktor wie folgt initialisiert:

```
zufallszahlGenerator = new Random();
```

Die Steuerung der Monster veranschaulicht das folgende Struktogramm. Es stellt nur einen Teil des Rumpfs der Methode *Tick* dar.



© brichzin@tcs.ifi.lmu.de, 2007

Anhang B: Farbwerte

Folgende Farbwerte sind innerhalb des Projekts Krümel & Monster möglich:
"blau", "dunkelblau"
"gelb", "dunkelgelb"
"gruen", dunkelgruen"
"rot", "dunkelrot"
"orange", "dunkelorange“
"pink", "dunkelpink"
"magenta", "dunkelmagenta"
"grau", "dunkelgrau "
"weiss", " schwarz", "zufall"

© brichzin@tcs.ifi.lmu.de, 2007