4 Kontrollstrukturen

- if-Bedingung
- while-Loop
- for-Schleife
- Operatoren

4.0 Kontrollstrukturen

Ich möchte nun gern zuerst auf die „Bedingungen" und Schleifen, die ich im zweiten Kapitel angesprochen habe zu sprechen kommen und das Wissen vertiefen. Wir erinnern uns, dass die Bedingung (z.B. if-Anweisung) den Programmfluss steuern kann. Die Bedingung steht zwischen den runden Klammern. Die weiteren Anweisungen zwischen den Geschweiften. Anfänger machen hier gern den Fehler, dass sie zu Anfang ein Semikolon hinter der runden Klammer schreiben. Das führt zu einem semantischen Fehler. D.h. Es gibt keine Fehlerausgabe im Output Fenster. Für den Kompiler verhält es sich so als ob hinter der Bedingung keine Anweisung erfolgt und die eigentliche Anweisung immer ausführt. Denn sie wird ja durch keine Bedingung gesteuert.

Sehr oft benutzte Bedingungen sind sind z.B. Prüfen von:

Gleichheit	– *(x == y)*
Nicht Gleich	- *(x != y)*
Kleiner als	- *(x < y)*
Grösser als	- *(x < y)*
Kleiner oder gleich	- *(x <= y)*
Kleiner oder gleich	- *(x <= y)*

4.1 if – Bedingung

Die if-Anweisung haben wir auch schon bereits im Kapitel zwei kennen gelernt. Deshalb gehe ich hier gleich etwas weiter.
Ihre generisch Struktur der if-Bedingung:

```
if(Bedingung)
{
   Anweisungen...
}
```

Bisher waren unsere Abfragen jedoch sehr einfach. Wir haben immer nur eine Bedingung abgefragt. Es können aber auch Situationen entstehen an denen der Programmfluss von zwei oder sogar drei Bedingungen die gleichzeitig erfüllt sein müssen gesteuert wird. Wenn ich z.B. feststellen möchte, ob eine Variable in einem bestimmten Wertebereich liegt, trifft dies genau zu. Dazu haben wir die logischen Operatoren zur Hand. Damit lassen sich logisch verknüpfte Ausdrücke schreiben. Processing besitzt zwei: Das logische ODER || . Das logische UND &&. Der Code der prüfen soll, ob eine Variable zwischen zwei Wertebereichen liegt könnte so aussehen:

Code:

```
int x = 10;

if(x >= 5 && x <= 10 )
{
  println("x liegt im Bereich von 5 - 10");
}
```

Wenn ich prüfen möchte ob einer oder mehrere von drei verschiedenen Zuständen wahr sind, damit das Programm weiter machen kann könnte ich folgenden Code schreiben.

Code:

```
boolean x = true;
boolean y = false;
boolean z = false;

if(x == true || y == true || z == true)
{
  println("Logische oder Bedingung triff zu");
}
```

Diese Beispiele bieten jeweils nur eine Alternative. Nämlich wenn die Bedingung zutrifft. Das optionale Schlüsselwort *else* veranlasst eine zweiseitige Alternative. Da es auch passieren kann, dass eine Bedingung mal nicht zutrifft und man das auch wissen möchte. Z.B. bei der Prüfung von Geschlechtern. Wenn ich eine Frau bin dann werde ich in Zukunft mit „Frau" angesprochen ansonsten mit „Herr". Hierzu schreiben wir ein kleines Programm welches zwei Alternativen anbietet.

Code:

```
String name       = "Hollig";
String geschlecht = "Frau";

if(geschlecht == "Frau")
{
  println("Hallo Frau "+name);
}
else
{
   println("Hallo Herr "+name);
}
```

Hier sehen wir, dass die zwei unabhängigen Anweisungsblöcke durch das *else* verbunden werden können.

```
if( Bedingung )
{
   Anweisungen...
}
else
{
   Anweisungen...
}
```

Oder noch etwas komplexer und verschachtelter

```
if( Bedingung )
{
   if( Bedingung )
  {
    Anweisungen...
  }
}
else
{
   if( Bedingung )
  {
    Anweisungen...
  }
  else
  {
    Anweisungen...
  }
}
```

Da die if-Anweisung zur Programmführung häufig benutzt wird um einen bestimmten Wert zu überprüfen, kann man sie mit else */ if* schachteln.
Wenn eine Variable einem bestimmten Wert entspricht, wird eine Anweisung ausgeführt,sonst wird eine andere Variable mit einem Wert getestet und so weiter. So können komplexe Abfrage erstellt werden.
Dazu der generische Code:

```
if( Bedingung )
{
   Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
} etc...
```

Dazu ein keines Codebeispiel. Es Wert einer Variable soll geprüft in einer Aufzählung werden . Je nach bestimmten Wert soll das Programm immer andere Kommandos ausführen.
Der Code dazu könnte so aussehen:

Code:

```
int zahl = 3;

if(zahl == 0)
{
  println("Zahl ist: 0");
}else if(zahl == 1)
{
  println("Zahl ist: 1");
}else if(zahl == 2)
{
  println("Zahl ist: 2");
}else if(zahl == 3)
{
  println("Zahl ist: 3");
}else if(zahl == 4)
{
  println("Zahl ist: 4");
}else if(zahl == 5)
{
  println("Zahl ist: 5");
}else if(zahl < 5)
{
  println("Zahl ist grösser als 5");
}
```

4.2 while – Schleife

Eine weitere Kontrollanweisung ist die while Scheife. Im Kapitel 2 haben wir auch diese schon mal ein wenig kennen gelernt. Von der Struktur her ist die while-Schleife der if-Bedingung ähnlich. Nur dass die while-Schleife so oft ausgeführt wird wie ihre Bedingung zutrifft. Im Gegensatz dazu wird das if nur einmal ausgeführt. D.h. beim ersten Durchlauf wird in der Schleife die Bedingung geprüft. Wenn diese zutrifft, werden die Anweisungen (Schleifen-Körper) zwischen den geschweiften Klammern ausgeführt. Danach beginnt, anders wie beim if, das Programm wieder die Bedingung zu Prüfen. Wenn diese wieder zutrifft werden die Anweisungen, oder Schleifenkörper ausgeführt. Das passiert so lange bis die Bedingung eben nicht mehr zutrifft. Erst dann geht das Programm aus der Schleife raus und bearbeitet das nächste Kommando. Bei Programmieren von while-Schleifen sollte man unbedingt darauf achten dass keine Endlosschleifen produziert werden.
Die generische Struktur der while-Schleife:

```
while(Bedingung)
{
   Anweisungen
}
```

Loop Terminologie:

Initialisierung
Die Angabe oder Ausdruck, die eine Variable oder mehrere definiert, welche in der Loop-Bedingung getestet werden.

Test des Ausdrucks
Die Bedingung, die zutreffen muss, damit die Subanweisungen im Loopkörper ausgeführt werden.

Update
Die Anweisung die die Variable modifiziert, welche in der Bedingung getestet wird. Eine typische Anwendung ist z.B. ein Zähler der die Schleifendurchgänge hoch oder runter zählt.

Verschachtelte Loops
Ein Loop der einen zweiten Loop beinhaltet. Das Erlaubt eine Iteration durch z.B. Zwei- oder Mehr- dimensionale Daten. Bei einer Tabelle ist der erste Loop für die Zeilen, der Andere für Iterierung der Zellen zuständig.

Iterator oder Index-Variable
Eine Variable die sich bei jedem Durchlauf des Lopps erhöht oder vermindert. Oft benutzt bei Zählern die Sequenziell durch Loops laufen. Herkömmlich heissen diese Zähler i, j, k ....

Loop body (Schleifen-Körper)
Beinhaltet den Block von Anweisungen die ausgeführt werden, wenn die Loop-Bedingung zutrifft. Der Schleifen-Körper kann gar nicht oder mehrere tausend Mal ausgefürt werden.

Loop Heder (Schleifen-Kopf)
Der Teil des Loops, der das Loop Schlüsselwort beinhaltet(*while, for*) und auf die dazugehörige Bedingung testet.

Endlosschleifen
Ein Loop der unendlich oft ausgeführt wird, weil der Wert seines Ausdrucks niemals falsch ist. D.h. Die Bedingung trifft immer zu.

Als kleine Übung zu diesem Thema wollen wir lauter Dreiecke der Breite nach zeichnen, bis das Applet in der komplette Breite ausgefüllt ist. Um ein Dreieck zeichnen zu können, verwenden wir das Freiform-Zeichenwerkzeug *beginnShape(TRIANGLES)*. Dazu iterieren wir jede x-Koordinate der drei Punkte , die es braucht ein Dreieck zu zeichnen. Die verschieden Zeichenwerkzeuge werde ich in einem gesonderten Kapitel erklären.
Und hier der dazugehörige Code:

Code:

```
size(600,80);             // Appletgrösse
int a = 0;                // Zähler oder x-Koordinate
while(a < width)          // Schleifen-Kopf und Bedingung
{
  beginShape(TRIANGLES); // Beginne Triangle
  vertex(a, 75);          // Erster  Vertexpunkt (x,y)
  a+=10;                  // Zähler wird um 10 pixel iteriert
  vertex(a, 20);          // Zweiter Vertexpunkt (x,y)
  a+=10;                  // Zähler wird um 10 pixel iteriert
  vertex(a, 75);          // Dritter Vertexpunkt (x,y)
  endShape();             // Schliesse Triangle
}
```

4.3 for-Schleife

Die for-Schleife ist mit der while-Schleife sinnverwandt. Der Unterschied ist, dass die for-Schleife den Iterator im Schleifenkopf definiert und in- oder dekrementiert. Die Syntax der Bedingung bleibt dieselbe. for-Schleifen werden oft genutzt um Aufzählungen abzuarbeiten.

Die generische Struktur der for-Schleife:

```
for(Initialisierung; Bedingung; Iteration)
{
   Anweisungen...
}
```

Die for-Schleife platziert also die Schlüssel-Komponenten, die es für eine Schleife braucht ordentlich in den Schleifen-Kopf. Sie werden durch ein Semikolon getrennt. Vor der ersten Iteration der Schleife, wird die Initialisierung des Zählers nur einmal vorgenommen! Wenn die Bedingung zutifft wird der Zähler iteriert und der Schleifen-Körper ausgeführt. Die schleife stoppt sobald ihre Bedingung falsch ist. Auch hier ist es möglich mehrere Schleifen ineinander zu verschachteln.

Damit man den Loop besser versteht und unterscheiden kann, hier zwei equivalente Beispiele mit der for- und while-Schleife.

Code:

```
// --- for-Schleife ---
for(int i = 0; i < 10; i++)
{
  println("Durchlauf for-Schleife: "+i);
}

// --- while-Schleife ---
int i = 0;
while(i < 10)
{
  println("Durchlauf while-Schleife: "+i);
  i++;
}
```

Um Bedingungen und Schleifen kontrollieren zu können brauchen wir zu unseren Operanden auch Operatoren. Die ich gerne auf der nächsten Seite aufführen möchte.

4.4 Operatoren

Ein Operator ist ein Symbol oder Schlüsselwort, dass Daten manipuliert, kombiniert und transformiert. Einige der Operatoren sind auch dem Nicht-Programmierer bestens bekannt, wie z.B:

- Das Multiplizieren (4*3). Der Operand(4) wird mit dem (*) Operator und einem anderen Datenwert (Operand 3) im Ausdruck multipliziert.
- Wir finden auch den = Operator bei jeder Zuweisung von Variablen-Werten.
- Um Bedingungen zu schreiben brauchen wir die Vergleichsoperatoren (<, >, ==,...) somit können Werte und Variablen miteinander verglichen werden.
- Und nicht zu Letzt die logischen Operatoren(&&, ||), die Verknüpfungen von Variablen oder Bedingungen ermöglichen.

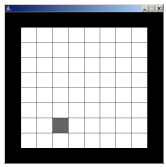
Viele haben wir in den letzten Skripten schon benutzt.

Der Vollständigkeitshalber möchte ich hier auf die verfügbaren Operatoren eingehen, die uns Processing bietet.

Operator	Beschreibung
.	Zugang zur Objekt Eigenschaft
[]	Zugang zum Array Element
()	Parenthese
function()	Funktionsaufruf
x++	Postfix increment (zählt x um 1 rauf)
y--	Postfix decrement(zählt y um 1 runter)
!	Logisches NICHT (NOT)
new	Neue Instanz
void	Keine bestimmte Rückgabe
*	Multiplikator
/	Division
%	Modulo
+	Addition
-	Subtraktion
<	Kleiner als
<=	Kleiner oder gleich als
>	Grösser als
>=	Grösser oder gleich als
==	Gleichheit
!=	Nicht-Gleichheit
&&	Logisches UND
||	Logisches ODER
=	Zuweisung
+=	Addition und Zuweisung
-=	Substraktion und Zuweisung
*=	Multiplikation und Zuweisung
/=	Division und Zuweisung

Übung zum Kapitel:

Als Trainingsübung habe ich die letzte Übung zum Kapitel 3 vom Konzept her erweitert. Es soll nun ein Raster gezeichnet werden das am Rand schwarz eingefärbt wird. Ein Quadrat soll sich per Zufall über das Raster bewegen ohne jemals über den Rand gezeichnet zu werden.



Code:

```
int quadrat = 50;                // Grösse des einzelen Quadrats
int zeilen;                      // Anzahl der Quadrate in x Richtung
int zellen;                      // Anzahl der Quadrate in y Richtung

float xpos;                      // x Postion des Quadrats
float ypos;                      // y Postion des Quadrats

void setup()                     // Aufruf bei Initialisierung
{
   framerate(5);                 // Setzen der Abspielgeschwindigkeit
   size(500,500);                // Setzten der Appletgroesse
   zeilen = width/quadrat;       // Zeilen Berechnung
   zellen = height/quadrat;      // Zellen Bereichnung
}


void draw()                      // Aufruf bei jeder Frame
{
  background(255);               // Setzten des Hintergrunds

  /************************************************************
  Malen des Rasters
  ************************************************************/
  for(int i = 0; i < zeilen; i++)
  {
    for(int j = 0; j < zellen; j++)
    {
      // Bedingung für den schwarzen Rahmen
      if(i == 0 || i == zellen-1 || j == 0 || j == zeilen-1 )
      {
         fill(0);
         rect((j*quadrat),(i*quadrat),quadrat,quadrat);
      }else{
         fill(255);
         rect((j*quadrat),(i*quadrat),quadrat,quadrat);
      }
    }
  }
  /************************************************************
  Setzten und Zeichnen der Zufallskoordinaten des Quadrats
  ************************************************************/
  xpos = (int)random(1,zellen-1);  // Wird passend konvertiert(int)
  ypos = (int)random(1,zeilen-1);  // sonst passt's nicht ins Raster

  fill(100);                       // Füllfarbe des Quadrats
  rect((quadrat*xpos),(quadrat*ypos),quadrat,quadrat);

}
```