5. Verzweigung

5.1. Bedingung

Im Leben müssen oft Entscheidungen getroffen werden - manche Entscheidungen sind nicht weltbewegend, andere hingegen können lebenswichtig sein. Eine Entscheidung beim Autofahren könnte etwa sein: “wenn auf dem Zebrastreifen vorne kein Mensch über die Straße geht oder gehen möchte, kann ich Gas geben, ansonsten muss ich abbremsen und anhalten”.

Solche Entscheidungen spielen auch in der Informatik eine wichtige Rolle, im konkreten Fall etwa bei selbstfahrenden Autos. Anhand bestimmter Bedingungen oder Einflüsse von außen wollen wir daher Computerprogrammen auch Entscheidungsmöglichkeiten geben, um verschiedenste Problemstellungen abbilden zu können. Solche Entscheidungen werden in Programmen “Verzweigungen” genannt, da sie alternative Programmabläufe bieten, ähnlich wie bei einer Weggabelung.

Eine Bedingung ist eine Abfrage oder auch eine Aussage, anhand derer danach eine Entscheidung getroffen wird: Trifft die Bedingung zu (die Frage wird mit “ja” beantwortet oder die Aussage ist wahr), dann werden bestimmte Aktionen ausgeführt. Trifft die Bedingung jedoch nicht zu, dann wird entweder nichts unternommen oder alternative Aktionen werden ausgeführt. Im obigen Beispiel wäre etwa die Bedingung “Überquert jemand vorn am Zebrastreifen die Straße bzw. möchte sie überqueren?” Wenn dies zutrifft, dann führe ich die Aktion “abbremsen und anhalten” aus. Trifft diese Bedingung nicht zu, dann kann ich Gas geben und weiterfahren.

Hinweis: Bedingungen können trotz unterschiedlicher Formulierung das Gleiche bedeuten: Zum Beispiel kann die Bedingung: “*Wenn Alice maximal so alt ist wie Bob*” auch umformuliert werden zu “*Wenn Alice nicht älter ist als Bob*”. Beide Bedingungen haben eine andere Formulierung, aber die gleiche Bedeutung,.

5.2. Vergleichsoperatoren

Um Bedingungen in Processing auszudrücken, verwenden wir Vergleichsoperatoren. Wie der Name bereits sagt, sind Vergleichsoperatoren dazu da, um Vergleiche zu formulieren. Wahrscheinlich kennen Sie Vergleichsoperatoren bereits aus der Mathematik. Die Bedeutung der Vergleichsoperatoren sind der aus der Mathematik gleich.
Ihre Schreibweise in Processing ist ein wenig anders und in der folgenden Tabelle zusammengefasst:

Bezeichnung	Vergleichsoperator Schreibweise
kleiner	<
kleiner gleich	<=
größer	>
größer gleich	>=
gleich	==
ungleich	!=

Das Ergebnis eines Vergleichs ist in Processing und vielen weiteren Programmiersprachen vom Datentyp `boolean` (siehe Kapitel 4) und hat daher entweder den Wert **true** (wahr) oder den Wert **false** (falsch).

Will man zum Beispiel herausfinden, ob Bob älter ist als Alice, könnte das Programm wie folgt aussehen:

```
void setup(){
  int ageAlice = 17;
  int ageBob = 19;

  boolean bobIsOlder = ageAlice < ageBob;
  println(bobIsOlder);
}
```

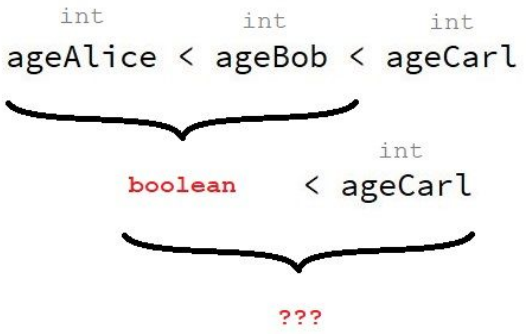
Nach dem Ausführen des Programms wird in der Konsole `true` ausgegeben, weil 19 größer ist als 17.

5.3. Logische Operatoren

Will man nun das Beispiel um eine Person, Carl, erweitern und wissen, ob Bob vom Alter her zwischen Alice und Carl liegt, dann ist es (auf die Mathematik zurückgreifend) naheliegend, folgende Bedingung zu schreiben:

```
ageAlice < ageBob < ageCarl
```

Wären wir in der Mathematik, so wäre diese Bedingung korrekt. Leider ist das in Processing und so gut wie jeder anderen Programmiersprache nicht zulässig. Der Grund dafür liegt in einer "Mischung" von verschiedenen Datentypen, da diese Bedingung eigentlich aus zwei Teilbedingungen besteht, nämlich `ageAlice < ageBob` und `ageBob < ageCarl`.
Die untenstehende Grafik soll anhand der jeweiligen resultierenden Datentypen verdeutlichen, warum diese Bedingung für Programmiersprachen keinen Sinn ergibt.

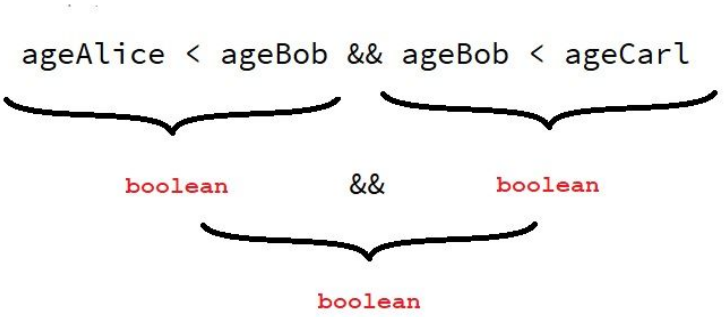


In Processing wird die Bedingung von links nach rechts abgearbeitet und daher zunächst die erste Teilbedingung (ageAlice < ageBob) evaluiert. Dabei werden zwei int-Zahlenwerte miteinander verglichen. Das Ergebnis dieses Vergleichs ist vom Datentyp `boolean`. Nun erfolgt die Evaluierung der zweiten Teilbedingung (Abbildung oben, zweite Zeile). Dabei wird das Ergebnis der ersten Teilbedingung mit dem Rest der Bedingung verglichen. Das bedeutet, es würde ein boolean Wert mit einem int Wert verglichen werden. Dies ist nicht möglich, weshalb es in Processing zu einem Fehler führt.

Die Lösung zu diesem Problem ist, die Bedingung auf zwei Bedingungen aufzuspalten und diese zwei Bedingungen miteinander zu verknüpfen. Für das konkrete Beispiel gilt: *Wenn `ageAlice < ageBob` wahr ist UND `ageBob < ageCarl` wahr ist,* dann ist die ganze Bedingung erfüllt. In Processing schreibt man dies folgendermaßen:

```
ageAlice < ageBob && ageBob < ageCarl
```

Der verwendete Operator (&&) nennt sich der UND-Operator. Logische Operatoren haben als Operanden immer Werte vom Datentyp `boolean`. Das Ergebnis einer solchen Operation ist wiederum ein `boolean`.



Es stehen vier logische Operatoren zur Verfügung, die im weiteren im Detail erklärt werden:

Name	Logischer Operator
NOT	!
AND	&&
OR	||
XOR	^

5.3.1. Operator mit einem Operanden

Der einzige logische Operator mit nur einem Operanden ist die **Negation**. Sie wird verwendet, um eine Aussage zu negieren, also zu verneinen bzw. um abzufragen, ob etwas nicht zu trifft. Die Negation wird in Processing mit einem Rufzeichen (`!`) ausgedrückt und wird vor die Aussage gesetzt. Angenommen Sie wollen folgende Aussage formulieren: “Alice ist nicht älter als Bob”. Dann müssten Sie zunächst die Teilaussage “Alice ist älter als Bob” formulieren und im zweiten Schritt negieren, was in Processing folgendermaßen aussehen würde:

```
boolean answer = !(alice > bob);
```

Im Grunde fügen wird also der normalen Aussage ein “nicht” vorne hinzu. Dadurch ergeben sich folgende Antworten für die Aussage `(alice > bob)`:

`(alice > bob)`	`!(alice > bob)`
true	false
false	true

War die Antwort `true`, dann bewirkt das Negieren, dass als neue Antwort `false` geliefert wird (“nicht wahr” entspricht “falsch”). War hingegen die Antwort `false`, dann bewirkt das Negieren, dass als neue Antwort `true` geliefert wird (d.h. “nicht falsch” entspricht “wahr”).

5.3.2. Operatoren mit zwei Operanden

AND-Operator (deutsch: UND-Operator)
Den Logischen AND-Operator schreibt man mit zwei Und-Zeichen (`&&`). Um den Operator zu verwenden, schreibt man ihn einfach zwischen zwei Aussagen

```
boolean answer = (alice > bob) && (alice > carl);
```

Mit dem Operator wollen Sie wissen, ob **sowohl** die linke Aussage **als auch** die rechte Aussage wahr ist. **Nur wenn beide Teilaussagen wahr sind, erhalten Sie als Antwort** `true`. Falls nur eine der beiden Aussagen oder gar beide Aussagen falsch sind, dann erhalten Sie als Antwort `false`.
Die Klammern sind hier nicht notwendig, sie verbessern aber die Lesbarkeit. Man kann natürlich auch noch mehr Aussagen anhängen z.B.

```
boolean answer = (alice > bob) && (alice > carl) && (bob > carl);
```

Hier erhalten Sie als Antwort nur dann `true`, wenn **jede** der Teilaussagen wahr ist.
Ansonsten erhalten sie `false`.

OR-Operator (deutsch: ODER-Operator)
Weiters gibt es den Logischen OR-Operator. Mit dem Operator setzen Sie die Abfrage um, ob zumindest eine der Aussagen wahr ist. Sie erhalten `true`, wenn mindestens eine der Aussagen wahr ist. Wenn gar keine der Aussagen wahr ist, dann erhalten Sie als Antwort `false`.

Der OR-Operator wird in Processing mit zwei senkrechten Strichen (`||`) dargestellt (auf einer deutschen Tastatur gibt man ihn mit der Tastenkombination “AltGr + <” (WIN, Linux) oder “Option/alt+ 7” (MAC) ein). Die Syntax ist die Gleiche wie für den AND-Operator - er wird zwischen die jeweiligen Aussagen geschrieben.

Zum Beispiel:
Wollen wir abfragen, ob Alice älter als Bob ist oder Alice älter als Carl ist, schreiben wir in Processing

```
boolean answer= (alice > bob) || (alice > carl);
```

XOR-Operator (deutsch: EXKLUSIVER ODER-Operator)
Es gibt vielleicht Situationen, wo Sie wissen wollen, ob nur genau eine der beiden Aussagen zutrifft. Ein Beispiel aus der Medizin: 2 Medikamente stehen für dieselbe Behandlung zur

Verfügung. Es darf nur eines davon, aber keinesfalls beide eingesetzt werden, da sich die beiden Medikamente nicht miteinander vertragen.

Solche **entweder...oder…** Operationen werden mit dem EXCLUSIVE-OR (XOR) Operator umgesetzt. Der XOR-Operator wird mit einem Zirkumflex - einem “Dacherl” - (^) dargestellt (auf einer deutschen Tastatur links neben der 1 zu finden). XOR wird auch zwischen zwei Aussagen geschrieben.

Das Beispiel mit den zwei Medikamenten, von welchem nur entweder das erste oder das zweite verwendet werden darf, um eine sichere Anwendung zu gewährleisten, wird in Processing wie folgt abgebildet:

```
boolean safeUse = useMedicineA ^ useMedicineB;
```

`useMedicineA` und `useMedicineB` sind dabei zwei Variablen vom Typ `boolean`, welche `true` beinhalten, wenn das Medikament verwendet wird bzw. `false`, wenn das Medikament nicht eingesetzt wird. Falls nur die linke Aussage wahr ist oder nur die rechte Aussage wahr ist, dann liefert die Operation `true` - nur eines der beiden Medikamente wird eingesetzt und es kommt zu keiner Wechselwirkung, d.h. der Einsatz ist sicher. Falls beide Aussagen falsch sind oder beide wahr sind dann erhalten Sie `false` als Antwort, d.h. der Einsatz ist nicht sicher, denn es besteht entweder eine Wechselwirkung aufgrund des Einsatzes beider Medikamente oder gar keine Wirkung, da keines der beiden Medikamente eingesetzt wird.

Wahrheitstabelle für Operatoren mit zwei Operanden

Die folgende Tabelle fasst das oben Erklärte nochmal kompakt zusammen indem sie die einzelnen Ausdrücke auswertet und deren Ergebnis, `true` oder `false`, aufschlüsselt. Diese Tabelle nennt man auch **Wahrheitstabelle**. Für die Aussagen `a` und `b` werden alle möglichen Kombinationen aufgelistet und die jeweiligen Resultate bei Verknüpfung mit bestimmten Operatoren in der Tabelle angezeigt.

Die ersten zwei Spalten repräsentieren dabei die Resultate für die einzelnen Aussagen `a` und `b`. Die Spalten drei bis fünf sind die Antworten, die Sie erhalten, wenn Sie die Logischen Operatoren anwenden auf `a` und `b`.

a	b	a && b AND	a || b OR	a ^ b XOR
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

zum Beispiel:

```
boolean a = true;
boolean b = false;
boolean answer= a || b;
println(answer);
```

Sie erhalten als Antwort `true`, denn `true || false` ist `true`. Die Variablen `a` und `b` können natürlich auch ausgewertete Ergebnisse sein und sinnvoller benannt werden:

```
void setup(){
  int ageAlice = 17;
  int ageBob = 19;
  int ageCarl = 25;

  boolean isYoungerB = ageAlice < ageBob;
  boolean isOlderC = ageAlice > ageCarl;
  boolean answer= isYoungerB || isOlderC;
  println(answer);
}
```

5.4. Optionale Codeausführung: If-Anweisung

Mithilfe von Vergleichsoperatoren und Logischen Operatoren ist es möglich komplexe Aussagen zu formulieren. Diese Aussagen können nun als Bedingungen verwendet werden, um in einem Computerprogramm optionale Code-Ausführungen zu ermöglichen, das heißt: Nur falls die Bedingung zutrifft, sollen bestimmte Anweisungen ausgeführt werden.

Dafür gibt es die **If-Anweisung**. Mit der `if`-Anweisung können Probleme der Form "Wenn … dann … " dargestellt werden. Ein Beispiel wäre:

"Wenn vor dir kein Hindernis ist, dann fahre gerade aus".

Die If-Anweisung ist folgendermaßen aufgebaut:

Allgemein:

```
if (Bedingung) {
  //optionaler Code
}
```

Bsp:

```
if (kein Hindernis vor dir) {
   //Anweisungen für 'fahre geradeaus'
}
```

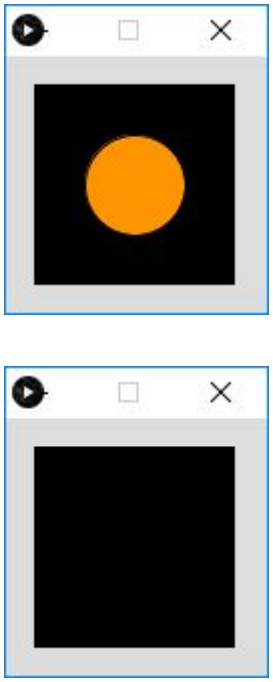
Die `if`-Anweisung wird mit dem Schlüsselwort `if` eingeleitet. In runden Klammern folgt die Bedingung, unter der der optionale Code ausgeführt werden soll. Danach werden in geschwungene Klammern der optionale Code hingeschrieben. Dieser wird nur ausgeführt, wenn die Bedingung zutrifft, also die Aussage wahr ist. Ist die Bedingung nicht erfüllt, wird der Code nicht ausgeführt.

Beispiel: Auf der Suche nach der ältesten Person wurde Bob nach seinem Alter gefragt. Da Bob die erste Person ist, wird sein Alter als das älteste Alter gesetzt. Danach wird Alice nach ihrem Alter gefragt. Ist Alice älter als Bob? Falls ja, soll das Maximalalter auf das Alter von Alice gesetzt werden und die Ausgabe "Alice ist älter als Bob" erfolgen. Danach wird das Maximalalter ausgegeben.

```
void setup() {

  int bob = 25;
  int maximumAge = bob;
  int alice = 19;

  if (alice > bob) {
     maximumAge = alice;
     println("Alice ist älter als Bob.");
  }

  println("Maximales Alter: " + maximumAge);
}
```

Da in diesem Beispiel Alice nicht älter als Bob ist, folgt nur die Ausgabe des maximalen Alters. Wird das Alter von Bob auf zum Beispiel 18 gesetzt, dann würde der Code innerhalb des If ausgeführt werden.

Beispiel: Ein animiertes graphisches Beispiel zeigt einen orange blinkenden Kreis (oder Ampel).



```
int x = 0;

void setup() {
  size(100, 100);
  fill(255, 150, 0);
}

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    ellipse(50, 50, 50, 50);
  }

  x++;
}
```

Eine globale Variable `x` wird hoch gezählt im **`draw`**`()` und im Bereich zwischen 0 und 119 gehalten. Falls `x < 60` gilt, dann wird ein orangener Kreis gezeichnet. Da der **`draw`**`()` Bereich etwa 60 mal die Sekunde ausgeführt wird, entsprechen 120 Durchgänge etwa 2 Sekunden. Durch die Bedingung erscheint in der ersten Sekunde der Kreis und in der zweiten Sekunde wird er nicht gezeichnet.

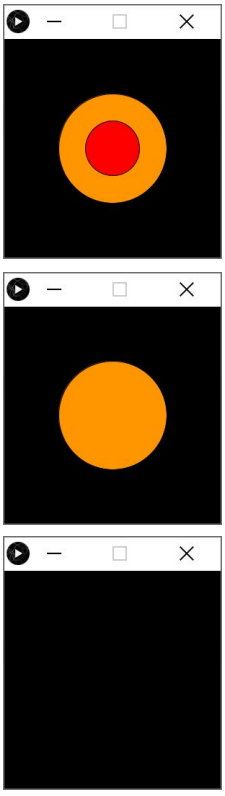
5.4.1. Verschachtelte If-Anweisungen

`if`-Anweisungen lassen sich natürlich auch verschachteln, das heißt, in einem `if`-Block kann noch ein `if`-Block stehen. Das macht dann Sinn, wenn man die zweite Anweisung nur dann ausführen möchte, wenn die erste bereits ausgeführt wurde. Im Beispiel mit dem Auto könnte das der folgende Fall sein:

"Wenn vor dir kein Hindernis ist, dann fahre gerade aus.
Wenn die Geschwindigkeit unter 10 km/h ist, beschleunige"

In dem Beispiel wollen wir nur dann beschleunigen, wenn wir bereits geradeaus fahren und zu langsam fahren. Damit verhindern wir, dass wir beschleunigen, wenn wir beispielsweise in der Kurve abbiegen.

Beispiel: Das animierte Ampel Beispiel wurde erweitert. Nun wird für eine kurze Zeit zusätzlich noch ein kleinerer rote Kreis gezeichnet. Der rote Kreis erscheint allerdings erst eine halbe Sekunde später auf als der orangene und kommt nie alleine vor.



```
int x = 0;

void setup() {
  size(100, 100);
}

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    fill(255, 150, 0);
    ellipse(50, 50, 50, 50);
    if (x > 30) {
      fill(255, 0, 0);
      ellipse (50, 50, 25, 25);
    }
  }

  x++;
}
```

Verschachtelte `if`-Anweisungen sind sehr brauchbar. Manchmal fordern sie aber einen heraus zu verstehen, welche Bedingungen nun wirklich gelten. Sehen Sie sich das folgende Codebeispiel an, der unter bestimmten Bedingungen ein oder zwei Kreise zeichnen soll:

```
void setup() {
  size(100, 100);
}

void draw() {
  int number = 8;

  if (number > 5) {
    ellipse(width/2, height/2, 100, 100);
    if (number < 3) {
      ellipse(width/2, height/2, 40, 40);
    }
  }
}
```

Ist es möglich, mit diesem Code einen zweiten Kreis zu zeichnen, wenn Ihnen nur erlaubt ist, den Initialisierungswert von `number` zu verändern? Es klingt für viele sehr einleuchtend, dass eine Zahl, die größer als 5 ist, nicht kleiner als 3 sein kann. Aber oft übersehen viele beim Programmieren diese Dinge. Sehen Sie sich das kleine Programm genauer an. Wenn die Zahl in `number` größer als 5 ist, dann wird ein Kreis gezeichnet. Danach wird überprüft, ob `number` kleiner als 3 ist. Dieser Fall kann nie eintreten, da `number` inzwischen nicht verändert wurde. Denn innerhalb vom ersten `if`-Block können wir garantieren, dass `number` größer als 5 ist, ansonsten hätten wir ja nie den `if`-Block betreten.

Diese Verschachtelung von `if` macht nur dann Sinn, wenn die Variable `number` verändert wird. Zum Beispiel so:

```
void setup() {
  size(100, 100);
}

void draw() {
  int number = 8;

  if (number > 5) {
    ellipse(width/2, height/2, 100, 100);
    number = number / 4;
    if (number < 3) {
       ellipse(width/2, height/2, 40, 40);
    }
  }
}
```

5.5. Alternative Codeausführung: If-Else-Anweisung

Oft ist es auch sinnvoll nicht nur optionalen Code zu haben, sondern auch einen alternativen Code. Um alternativen Code auszuführen gibt es die **If-Else-Anweisung**. Sie ist eine Erweiterung der `if`-Anweisung. Mit ihr können Probleme der Form "Wenn … dann … sonst …" dargestellt werden. Der "sonst"-Teil stellt die Alternative dar. Das Beispiel von vorhin sollte erweitert werden zu:

"Wenn vor dir keine Wand ist, dann fahre gerade aus, sonst bleibe stehen".

Die If-Else-Anweisung ist folgendermaßen aufgebaut:

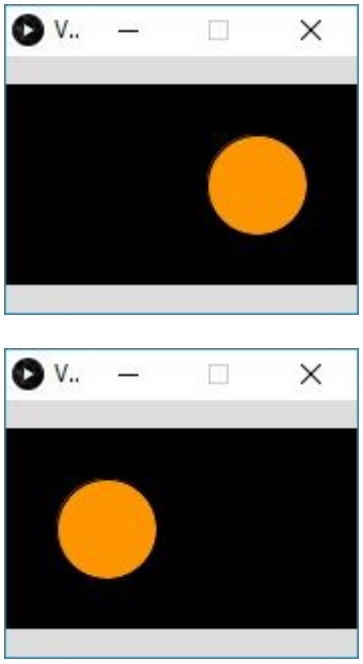
```
if (Bedingung) {
  // optionaler Code
} else {
  // Alternativer Code
}
```

Die `if`-Verzweigung wird um einen else-Teil, welcher mit dem Schlüsselwort `else` eingeleitet wird, erweitert. In geschwungener Klammer folgen dann die alternativen Anweisungen. Der `else`-Teil hat keine Bedingung, sondern wird immer dann ausgeführt, wenn die Bedingung im `if`-Teil nicht zutrifft. Durch die `if-else`-Anweisung wird somit auf jeden Fall einer der beiden Code-Blöcke ausgeführt. Im Beispiel oben fährt also das Auto entweder weiter oder es bleibt stehen.

Beispiel: Der Altersvergleich zwischen Alice und Bob kann um eine Ausgabe erweitert werden. Statt nichts auszugeben, falls Alice älter ist, kann alternativ ausgegeben werden, dass Alice nicht älter ist, um ein besseres Feedback ermöglichen.

```
void setup() {
  int bob = 25;
  int alice = 19;
  int maximumAge = bob;

  if (alice > bob) {
     maximumAge = alice;
     println("Alice ist älter als Bob");
  } else {
     println("Alice ist nicht älter als Bob.");
  }

  println("Maximales Alter: " + maximumAge);
}
```

Beispiel: Auch die blinkende Ampel kann erweitert werden. Statt eines Kreises sollen nun zwei Kreise abwechselnd für jeweils eine Sekunde angezeigt werden.



```
int x = 0;

void setup() {
  size(175, 100);
  fill(255, 150, 0);
}

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    ellipse(50, 50, 50, 50);
  } else {
    ellipse(125, 50, 50, 50);
  }
  x++;
}
```

5.5.1. Verschachtelung mehrerer If und Else Blöcke

In jedem `if`- oder `else`-Block können wiederum beliebig viele `if`-Blöcke, mit oder auch ohne `else`-Block, folgen. Aus der `if`- und `if-else`-Verzweigung können durch eine bestimmte Verschachtelung auch **mehrere Optionen/Alternativen** repräsentiert werden. Das Beispiel mit dem Auto kann erweitert werden:

"Wenn vor dir keine Wand ist, dann fahre gerade aus, sonst:
Wenn links von dir keine Wand ist, dann fahre nach links, sonst:
wenn rechts von dir keine Wand ist, dann fahre nach rechts, sonst bleib stehen."

Der Aufbau sieht folgendermaßen aus:

```
if (Bedingung 1) {
  // Option 1 Code
} else {
  if (Bedingung 2){
    // Option 2 Code
  } else {
    if (Bedingung 3) {
      // Option 3 Code
    } else {
      // Alternativer Code
    }
  }
}
```

Falls die erste Bedingung nicht zutrifft, wird der alternative Teil des ersten `if`s ausgeführt. Dieser wiederum besteht selbst nur aus einem Paar von `if-else`-Block. Falls die zweite Bedingung nicht erfüllt ist, wird wiederum die dritte Bedingung überprüft. Dieses Muster kann so oft wiederholt werden, wie man will. Treffen alle Bedingungen nicht zu, kann ein alternativer Code in einem letzten `else`-Block hinzugefügt werden. Da der Code bei großen Mengen von Optionen bzw. Alternativen sehr unübersichtlich wird, schreibt man diese Art von Verschachtelung gerne auch anders:

```
if (Bedingung 1) {
  // Option 1 Code
} else if (Bedingung 2){
  // Option 2 Code
} else if (Bedingung 3) {
  // Option 3 Code
}
...
else {
  // Alternativer Code
}
```

Um die Leserlichkeit zu bewahren, werden die geschwungene Klammern für die `else`-Blöcke weggelassen und der innere `if`-Block direkt an das `else` angehängt. Der restliche Code wird entsprechend eingerückt. Es handelt sich hier um keine neue Anweisung oder Konstrukt. Der Code ist aus dem vorigen verschachtelnden Code hergeleitet.

Beispiel: Das Beispiel mit dem Altersvergleich kann noch einmal erweitert werden. Denn “nicht älter” sein, kann zwei Dinge bedeuten. Entweder Alice ist jünger oder Alice und Bob sind gleich alt.

```
void setup() {
  int bob = 25;
  int alice = 19;
  int maximumAge = bob;

  if (alice > bob) {
    maximumAge = alice;
    println("Alice ist älter als Bob");
  } else if (alice == bob) {
    maximumAge = alice;
    println("Alice und Bob sind gleich alt");
  } else {
    maximumAge = bob;
    println("Alice ist jünger als Bob");
  }
  println("Maximales Alter: " + maximumAge);
}
```

5.5.2. Nie erreichter Programmcode

Mit Verzweigungen kann man also elegant steuern, welche Codeteile ausgeführt werden. Es muss aber auch die Reihenfolge, in welcher die Abfragen passieren, beachtet werden: Führen Sie folgenden Code in Processing aus. Idealerweise sollte das Programm die Farben und Positionen der gezeichneten Ellipsen ändern. Von schwarz auf rot über gelb zu grün und wieder von vorne. Allerdings wird nur ein grüner Kreis angezeigt. Versuchen Sie zu erklären, warum jenes Bild erzeugt wird.

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  } else if (number < 300) {
    fill( yellow );
    ellipse(width/2, 300, 100, 100);
  } else if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  } else if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  }
}
```

Erklärung:
Das Problem ist, dass in der ganzen `if`, `else if` Abfolge maximal nur ein Block ausgeführt wird, egal wie viele `else if` Blöcke angehängt werden. Entweder trifft einer der Aussagen zu, der jeweilige Block wird ausgeführt und der Rest der Blöcke wird übersprungen oder es trifft keine einzige Aussage zu und nichts wird ausgeführt. In diesem Fall wird zuerst die Aussage `number < 400` überprüft und die Aussage ist wahr. Wegen modulo 400 wird `number` immer kleiner 400 sein. Deshalb wird ein grüner Kreis gezeichnet und der Rest der Abfragen wird übersprungen.

Um dieses Problem zu lösen muss die Reihenfolge der Abfragen und Blöcke abgeändert werden.

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  } else if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  } else if (number < 300) {
    fill(yellow);
    ellipse(width/2, 300, 100, 100);
  } else if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  }
}
```

5.5.3. Mehrere unabhängige if vs. if- else if -else Anweisungen

Es gibt Situationen, in denen man nur eine Möglichkeit bzw. Alternative haben möchte (zum Beispiel beim Altersvergleich), manchmal ist es aber wünschenswert, dass mehrere Blöcke ausgeführt werden. In einer Abfolge von `if-else if-else` wird immer nur ein Block ausgeführt. Wenn mehrere Blöcke ausgeführt werden sollen, müssen die zusammenhängenden `if- else if-else` Blöcke getrennt werden. Folgender Code ist eine Möglichkeit, den Code aus dem vorigen Abschnitt zu trennen:

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  }
  if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  }
  if (number < 300) {
    fill(yellow);
    ellipse(width/2, 300, 100, 100);
  }
  if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  }
}
```

Nun werden - je nachdem welchen Wert `number` hat, mehrere Kreise gezeichnet. Durch die Aufteilung in mehrere `if`-Blöcke wird jede einzelne Aussage überprüft und keine Abfrage übersprungen. Jedes `if` ist daher unabhängig von den anderen `if`. Es können jetzt alle Blöcke ausgeführt werden, sofern die jeweilige Bedingung zutrifft. Testen Sie den Code und versuchen Sie die Unterschiede nachzuvollziehen!

5.5.4. Sichtbarkeit von Variablen / Lokale Variablen Wiederholung

Die Sichtbarkeit der Variablen ist auf den Bereich der geschwungenen Klammern begrenzt. Innerhalb dieser ist die Variable nach ihrer Deklaration verwendbar. Außerhalb der geschwungenen Klammern existiert die Variable nicht. Auch in diesem Abschnitt spielt die Sichtbarkeit der Variable eine Rolle. Da die `if`-Anweisung (und die Alternative Anweisungen `else if` und `else`) geschwungene Klammern verwendet, können Variablen, die innerhalb der `if`-Anweisung deklariert wurden, außerhalb nicht verwendet werden. Folgenden Code z.B. lässt Processing nicht zu:

```
void setup() {
  size(100, 100);
}

void draw() {
  int number = 10;

  if (number > 10) {
    int pos = 0;
  } else {
    int pos = 50;
  }

  ellipse(pos, pos, 100, 100);
}
```

Sie erhalten den Fehler “The variable ‘pos’ does not exist” bzw. “pos cannot be resolved to a variable” wenn Sie auf *Run* klicken. Um die Variable `pos` außerhalb der `if-else` Anweisung verwenden zu können (wie hier als Parameter in `ellipse(pos, pos, 100, 100);`), muss sie auch außerhalb der `if-else` Anweisung deklariert sein:

```
void setup() {
  size(100, 100);
}

void draw() {
  int number = 10;
  int pos;

  if (number > 10) {
    pos = 0;
  } else {
    pos = 50;
  }
  ellipse(pos, pos, 100, 100);
}
```

5.6. Abfrage von Maus und Tastatur

Jetzt, da Sie wissen, wie Sie Processing dazu bringen Entscheidungen zu treffen, können wir noch mehr Interaktivität in das Spiel integrieren, wie etwa mittels Tastatur- und Mauseingaben.

In Processing kann abgefragt werden, ob eine Maustaste geklickt wurde und wenn ja, welche (links oder rechts). Auch die Abfrage von Tastatureingaben. Um diese Funktionalität zu nutzen, gibt es in Processing die folgenden internen (globalen) Variablen:

- `mousePressed`
- `mouseButton`
- `keyPressed`
- `key`

`mousePressed` und `keyPressed` sind Variablen vom Datentyp `boolean`. Sie haben also den Wert `true` oder `false`, je nachdem ob eine Taste gedrückt wurde oder nicht. Die Variablen `key` und `mouseButton` halten Informationen darüber, welche Tastaturtasten bzw. Maustasten gedrückt wurden.

Anstelle langwieriger Erklärungen - spielen Sie einfach mal mit dem vorgegebenen Code und finden Sie heraus, wie genau diese Variablen in Processing verwendet werden und wie mit ihnen eine Interaktion gestaltet werden kann. Laden Sie sich dazu die Datei *pacman5.pde* herunter. Starten Sie das Programm und versuchen Sie mit den Tasten `a` und `d` den PacMan zu steuern. (Achten Sie darauf, dass das Sketch Fenster fokussiert ist. Klicken ins Sketch Fenster, um das zu gewährleisten). Schauen Sie sich im Programm folgenden Codeabschnitt an:

```
if (keyPressed) {
  if (key == 'd') {
    pacManHorizontal = 1;
    pacManVertical = 0;
  }
  if (key == 'a') {
    pacManHorizontal = -1;
    pacManVertical = 0;
  }
}

pacManCenterX = (pacManCenterX + pacManHorizontal);
pacManCenterY = (pacManCenterY + pacManVertical);
```

Ändern Sie die Werte von `pacManHorizontal` und `pacManVertical` (erhöhen oder vertauschen Sie z.B. die Werte). Ändern Sie auch die Abfragen. Beobachten Sie, wie sich das Programm nach den Veränderungen verhält.