Die Entscheidung, oder die Geschichte von *if* und *switch*

Jobst-Hartmut Lüddecke

8. April 2013

Zusammenfassung

Nach den Schleifen in der 3. Lektion – als Grundstruktur für Wiederholungen – kommt nun in der 4. Lektion die Grundstruktur der Fallunterscheidung. Ohne diese Grundstrukturen kann man keine sinnvollen Programme schreiben und sie müssen unbedingt vollständig verstanden und angewandt werden können. Hier bitte kein *Mut zur Lücke*, sondern ggf. unbedingt nacharbeiten! Zur Beruhigung der Nerven gibt es noch mögliche Lösungen der Schleifen-Aufgaben. Dabei führen aber *viele Wege nach Rom* und es gibt nicht *die einzig richtige Lösung*, sondern es geht auch immer etwas anders.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Entscheidungen

Das zweite wichtige Steuerungsmittel, neben den Schleifen für Wiederholungen, ist in allen Programmiersprachen die Entscheidung durch eine *Fallunterscheidung* anhand einer *Bedingung*. Diese *Bedingung* haben wir schon bei den Schleifen kennengelernt um die Wiederholungen zu steuern.

Hier wollen wir nun mit einer *Bedingung* eine *Programmverzweigung* steuern. Dies kann man sich wie eine *Weiche* bei der Bahn vorstellen, wobei man je nach Erfüllung der Bedingung das *linke oder das rechte Gleis benutzt*. Das Programm kann also jetzt unterschiedlich reagieren, je nachdem, ob die *Bedingung* erfüllt – und damit *true* ist – oder ob die *Bedingung* nicht erfüllt – und damit *false* – ist. Genau dies leistet die *if*-Abfrage. Ein Sonderfall ist die *switch*-Abfrage, die noch ein paar *Abzweigungen* mehr, wie eine *Gleisharfe* auf einem *Güterbahnhof* besitzt.[1]

1.1 if

Die *if*-Abfrage ist also das *entweder-oder* der Programmierung. Im Englischen direkt das *if-else* und auch so in fast allen Programmiersprachen implementiert.[2]

Für die, die es immer noch nicht begriffen haben: Die Begriffe in den *spitzen Klammern* sind *Platzhalter* zur *Erklärung* und *kein richtiger Code* in *Javascript*. Es ist also für das Verständnis eine *abstrakte Ebene*. Also bitte beim Abtippen das *Gehirn einschalten*. ☺

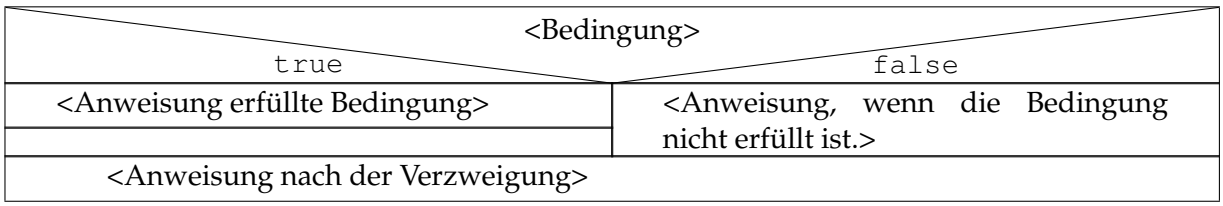

Abbildung 1: Einfache IF-Anweisung

```
if (<Bedingung>){
    <Anweisung für die erfüllte Bedingung>
    }
else{
    <Anweisung, wenn die Bedingung nicht erfüllt ist>
    }
<Anweisung nach der Verzweigung>
```

Abbildung 2: Prinzip der *if*-Abfrage in Javascript

[1] Zur Illustration siehe auch http://de.wikipedia.org/wiki/Gleisharfe und ...Maschen_Rangierbahnhof
[2] Siehe Beispiel Struktogramm Abbildung 1.1 auf Seite 2 und das Listing als Abbildung 2 auf Seite 2.

Jetzt konkret: *If* ist in *Javascript* als fest implementierte *Funktion* zu betrachten. Wie bei jeder Funktion in *Javascript* folgt dem *Namen der Funktion* ein *rundes* und ein *geschweiftes Klammernpaar*. In das *runde Klammernpaar* kommt als *Übergabeparameter* die *Bedingung* (die beliebig komplex werden kann) und in das *geschweifte Klammernpaar* der *Funktionsinhalt*, also die Aktionen, die durchgeführt werden sollen, wenn die **Bedingung erfüllt** ist.

Folgt gleich nach der *geschweiften Klammer zu* des *if* ein *else*, auch wieder mit einem *geschweiftem Klammerpaar*, dann stehen in diesem *geschweiftem Klammerpaar* nach dem *else* die Aktionen, die im Falle der **nicht erfüllten Bedingung** des *if* ablaufen sollen. Lässt man das *else* weg, dann erfolgt bei *nicht erfüllter Bedingung* eine *Leeranweisung*, also nichts.

Danach laufen *beide Gleise wieder zusammen*, d.h. die *Fallunterscheidungen* sind abgearbeitet und es geht normal im Programm weiter, es geht also auf dem *Hauptgleis* weiter.

Achtung: Machen Sie **kein Semikolon** nach der *geschweiften Klammer zu* des *if* direkt vor dem *else*, denn sonst gilt die if-Anweisung als abgeschlossen und das Programm kommt **nie in den else-Zeig**.

Natürlich lassen sich die *if*-Abfragen auch *rekursiv*[3] schachteln und für unterschiedliche Bedingungen *sequentiell*[4] wiederholen.

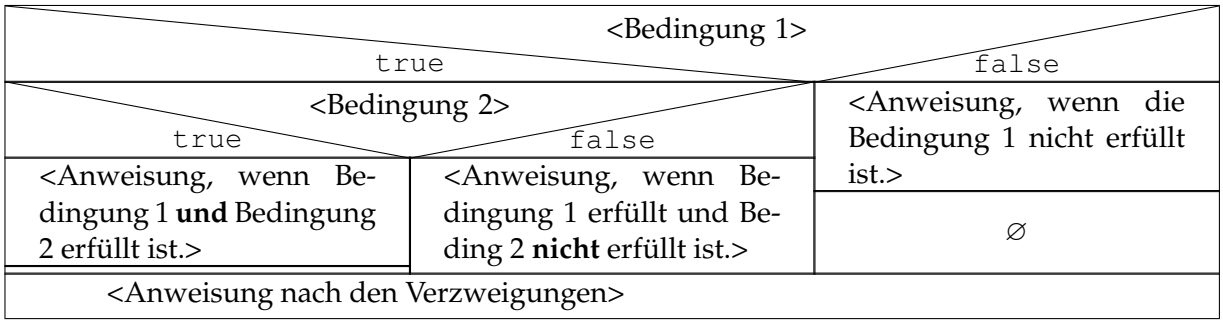

Abbildung 3: geschachtelte IF-Anweisung

```
if (<Bedingung1>){
    if (<Bedingung2>){
        <Anweisung, wenn Bedingung 1 und Bedingung2
        erfüllt ist>
        }
    else{
        <Anweisung, wenn Bedingung1 erfüllt ist und
        Bedingung2 nicht erfüllt ist>
        }
    }
else{
    <Anweisung, wenn Bedingung1 nicht erfüllt ist>
    }
<Anweisung nach den Verzweigungen>
```

Abbildung 4: Prinzip einer geschachtelten *if*-Abfrage in Javascript

Bei der *geschachtelten if*-Abfrage[5] erreicht man so eine Art logische *und*-Verknüpfung. D.h. im Beispiel erreicht man den Zweig der *erfüllten Bedingung2* nur, wenn vorher die *erfüllte Bedingung1* vorhanden ist.

[3] ineinander, oft gebrauchter Begriff in der Informatik, sollte man mal gelesen haben.
[4] nacheinander, auch so ein Begriff, den man kennen sollte. Jedes Fachgebiet hat seine *Vokabeln*.
[5] siehe Struktogramm Abbildung 1.1 auf Seite 3 und Listing Abbildung 4 auf Seite 3

Es gibt dann noch eine Möglichkeit für die *erfüllte Bedingung1* und die *nicht erfüllte Bedingung2*. Bei der *nicht erfüllten Bedingung1*, wird etwas ganz anderes gemacht und erst gar nicht nach *Bedingung2* gefragt.

Ein Fall der Wiederholung ist es, der Reihe nach[6] verschiedene Bedingungen für verschiedene Fälle der Reihe nach abzufragen und so nach den verschiedenen Fällen seine Aktionen zu sortieren.[7]

Dafür ist es nicht notwendig, dass immer ein *else*-Zweig zur *if*-Abfrage existiert.

Achtung: Machen Sie *kein Semikolon* zwischen der *geschweiften Klammer zu* des *if* und dem folgenden *else*. Dann ist die *if-Anweisung* **abgeschlossen** und das Programm kommt nie in den *else-Zweig* und Sie suchen stundenlang den Fehler.

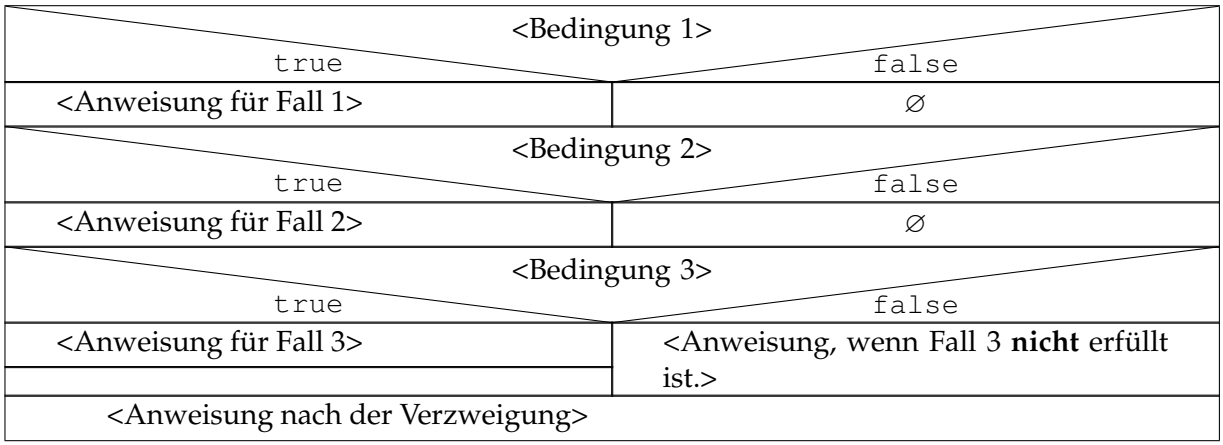

Abbildung 5: sequentielle IF-Anweisung

```
if (<Bedingung1>){
    <Anweisung für den Fall1>
    }
if (<Bedingung2>){
    <Anweisung für den Fall2>
    }
if (<Bedingung3>){
    <Anweisung für den Fall3>
    }
else{
    <Anweisung, wenn Fall3 nicht erfüllt ist>
    }
```

Abbildung 6: Prinzip einer wiederholten *if*-Abfrage in Javascript

[6] eben *sequentiell*

[7] Siehe Struktogramm Abbildung 1.1 auf Seite 4 und Listing Abbildung 6 auf Seite 4.

1.2 switch

Switch ist hier der Umschalter, der mehrere Schalterstellungen und nicht nur *an* oder *aus* besitzt.

Wenn wir wieder die Bahn und ihre Gleise als Schaubild heranziehen, haben wir jetzt die *Gleisharfe* eines *Rangier-Bahnhofs*.

Damit lassen sich wiederholte *if*-Abfragen für verschiedene Fälle ersetzen (siehe Struktogramm Abbildung 1.2 auf Seite 5 und Listing Abbildung 8 auf Seite 5).

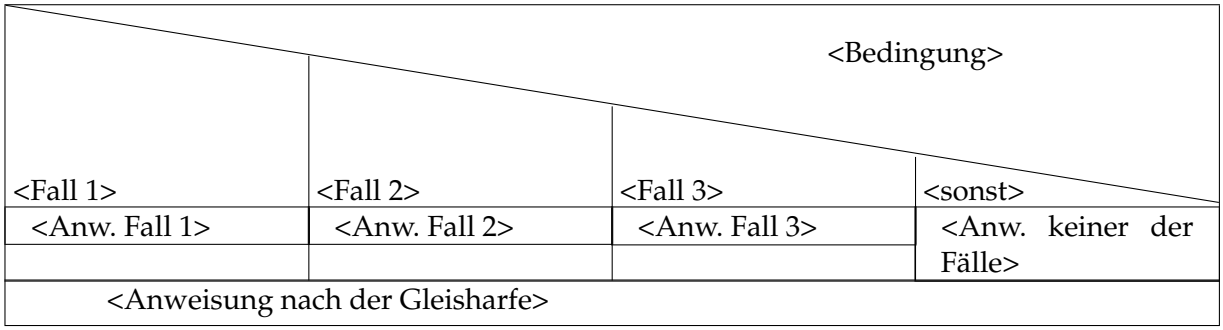

Abbildung 7: Struktogramm einer *switch*-Abfrage

```
switch(<Bedingung>){
    case 1: <Anweisung für den Fall1>;
        break;
    case 2: <Anweisung für den Fall2>;
        break;
    case 3: <Anweisung für den Fall3>;
        break;
    default : <Anweisung, wenn die Fälle 1 bis 3
                nicht erfüllt sind>
    }
```

Abbildung 8: Prinzip einer *switch*-Abfrage in Javascript

2 Aufgaben

- erweitern Sie ihre *Tabelle* dahingehend, dass alle durch 5, ohne Rest, teilbaren Werte (Sie erinnern sich hoffentlich an den Operator *Modulo*) im **Fettdruck** ausgegeben werden. Zeichnen Sie vorher ein Struktogramm und informieren Sie sich dazu bei *Stefan Münz* wie man eine *Modulo*-Rechnung macht. Zu verwenden ist eine *if*-Abfrage.
- erweitern Sie ihre *Tabelle* (nicht die Modula-Aufgabe) dahingehend, dass alle, glatt durch 10 teilbaren Werte als *römische Zahlen* ausgegeben werden. Dafür drängt sich eigentlich ein *switch* mit einer Textausgabe auf.
- Wer in der Stunde nicht fertig wird, macht es zuhause fertig, und zwar mit *Dokumentation*.[8]

[8] das übt und macht *Klausur-fest*.

3 mögliche Lösungen der Schleifenaufgaben

Viele Wege führen nach Rom und so gibt es in der Programmierung nicht nur *eine* Lösung, sondern *mehrere* mögliche Lösungswege, die alle ihre Vor- und Nachteile haben und sehr unterschiedlich lang und *elegant* sein können. Mit Hilfe von grafischen *Werkzeugen* wie *Struktogramme* und *Hierarchiediagramme*, wird dann aber sehr schnell deutlich, ob es ein *Spaghetti-Programm* und sehr umständlich programmiert ist, oder ob eine klare und einleuchtende Struktur vorhanden ist. Daher noch einmal die Empfehlung: *Nehmen Sie Papier und Bleistift und skizzieren Sie eine Struktur ihrer Lösung, bevor sie anfangen zu programmieren!*

3.1 Tabelle

In meiner Lösung der *Tabellen-Aufgabe*[9] habe ich mich für 2 geschachtelte *For*-Schleifen entschieden. Die erste *For*-Schleife *ackert* die 10 Zeilen ab. Innerhalb jeder Zeile *klappert* eine andere *For*-Schleife die 10 Spalten von 1 bis 10 ab. Die Variable *Zahl* wird mit dem Wert *Null* initialisiert und bei jeder Ausgabe in einer Spalte inkrementiert[10]. Da 10 Zeilen mit je 10 Spalten 100 Werte sind und der erste Wert eine 0 ist, ist der letzte Wert zwangsläufig 99.[11]

Abbildung 9: Struktogramm der *Tabelle*

```
<!DOCTYPE html>
<html lang="de" xml:lang="de">
<head>
<title>Tabelle mit for-Schleifen mit Javascript</title>
<script language="javascript" src="tabelle.js">
</script>
</head>
<body onload=zaehler()>

</body>
</html>
```

Abbildung 10: *Tabelle* mit 2 Zählschleifen für Zeilen und Spalten

[9] siehe Struktogramm Abbildung 9 auf Seite 6 und Listing Abbildung 11 auf Seite 7
[10] jeweils den Wert um 1 erhöht — haben wir schon mal gehabt, *Vokabeln lernen*
[11] 0 bis 99 sind genau 100 Zahlen, wer es nicht glaubt, zähle nach.

```
function zaehler(){
    var zeile;
    var spalte;
    var zahl=0;
    var schluss=10;
    document.write("hier geht es los: ");
    document.write("<br>");
    for( zeile=1; zeile<=schluss; zeile++ ){
        document.write("<br>");
        for(spalte=1;spalte<=schluss;spalte++){
            document.write(zahl);
            document.write(",");
            zahl++;
            }
        }
}
```

Abbildung 11: tabelle.js

3.2 Tabellen-Werk

Kochrezept: Man nehme die gelöste Aufgabe *Tabelle* und erweitere diese um eine zusätzliche, äußere Schleife für mehrere Tabellen.

Der erste Ansatz wäre eine zusätzliche *For*-Schleife für 10 Tabellen, ähnlich der *Zeilen*-Schleife im Tabellen-Beispiel. Damit ist das Tabellenwerk wirklich *kinderleicht* zu programmieren.

Als kleine *Gemeinheit* war aber eine *while*-Schleife für die äußere Schleife als Pflicht gefordert!

Schließlich sollen Sie nicht nur eine Schleifenform *beherrschen*. ☺

Jetzt könnte man sich das Beispiel der *Zählschleife mit while* aus der letzten Stunde ansehen und hätte ein direktes Äquivalent zur *For*-Schleife und die Aufgabe eigentlich *schon im Kasten*.

Es geht aber noch einfacher! Wozu noch eine zusätzliche Variable *Tabellen* mitschleppen und diese weiterzählen und prüfen, wenn wir doch schon die Abbruchbedingung vorgegeben bekommen haben? Wir sollen das *Tabellen-Werk* beenden, wenn die auszugebende *Zahl* den Wert *999* erreicht hat, bzw. wir wollen solange Zahlen ausgeben, solange die *Zahl* einen **Wert kleiner als 1000** hat. Diese Bedingung lässt sich doch ganz einfach mit *while* formulieren.[12]

[12] siehe Struktogramm Abbildung 12 auf Seite 8 und Listing Abbildung 13 auf Seite 8.

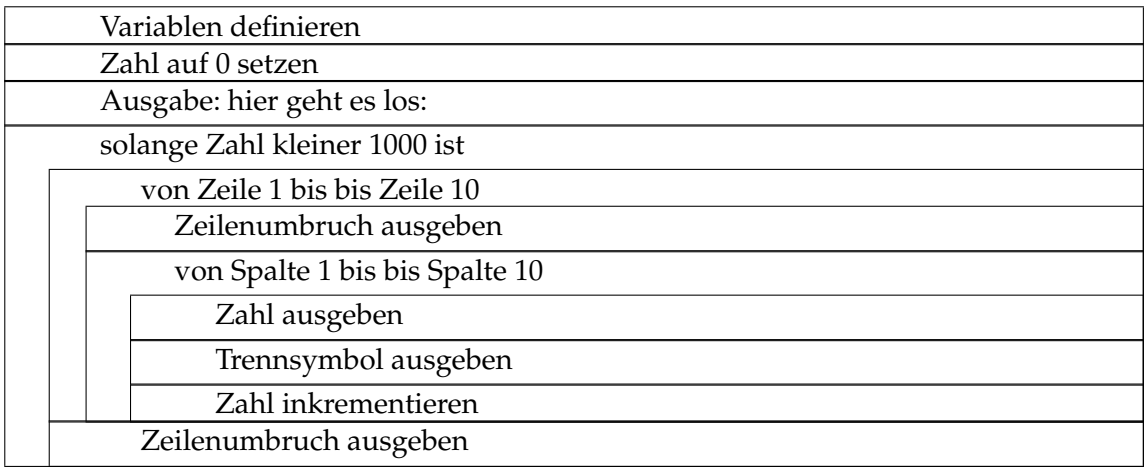

Abbildung 12: Struktogramm des *Tabellen-Werkes*

```
function zaehler(){
    var zeile;
    var spalte;
    var zahl=0;
    var schluss=10;
    document.write("hier geht es los: ");
    document.write("<br>");
    while(zahl<1000){
        for( zeile=1; zeile<=schluss; zeile++ ){
            document.write("<br>");
            for(spalte=1;spalte<=schluss;spalte++){
                document.write(zahl);
                document.write(",");
                zahl++;
                }
            }
        document.write("<br>");
        }
}
```

Abbildung 13: *Tabelle-Werk* mit 2 Zählschleifen für Zeilen und Spalten und einer *while*-Schleife für den letzten Tabelleneintrag