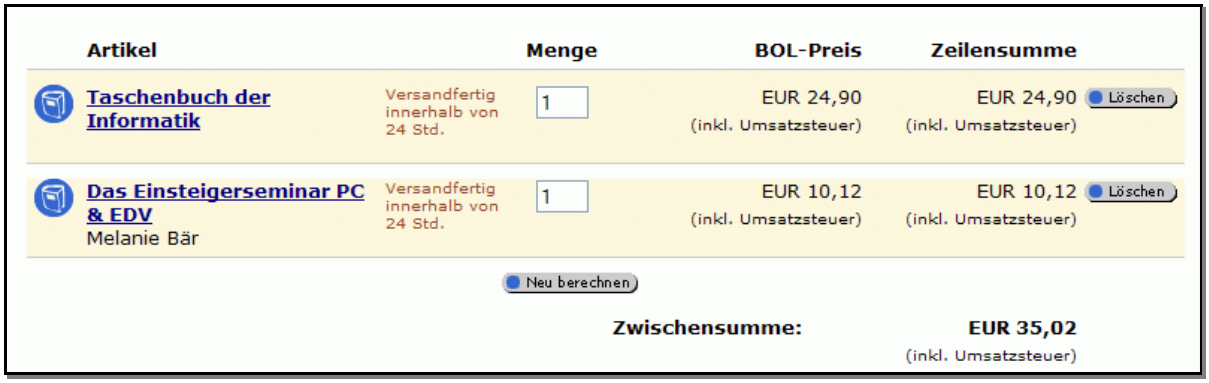


Abb. 1: Diese Webseite von www.bol.de berechnet automatisch die gesamte Kaufsumme.

Wir programmieren interaktive Webseiten

Viele Webseiten können nicht nur Eingaben entgegennehmen, sondern diese auch verarbeiten. So kann das Bestellformular in Abb. 1 z.B. den Gesamtpreis berechnen. Derartige Datenverarbeitungen kann die Seitenbeschreibungsprache HTML nicht leisten. Dazu ist eine Programmiersprache erforderlich. Besonders gut geeignet ist die Programmiersprache **JavaScript**. Sie arbeitet ausgezeichnet mit HTML zusammen, sie ist eine moderne Programmiersprache und sie ist in den gängigen Browsern bereits eingebaut. Diesem Umstand ist es zu verdanken, dass Webseiten wie die in Abb. 1 überhaupt funktionieren.

Die Programmiersprache JavaScript ist dir allerdings nicht ganz unbekannt. Du kennst sie bereits von der Igelsteuerung; die dort benutzten Anweisungen entstammten direkt der Programmiersprache JavaScript oder waren mit ihrer Hilfe hergestellt worden. Hattest du beim Igel die Anweisungen noch in einen speziellen Kommandobereich eingegeben, so wollen wir uns jetzt damit beschäftigen, wie sich diese Anweisungen direkt in den HTML-Quelltext einbauen lassen, so dass z. B. Berechnungen wie in Abb. 1 auf einen einfachen Knopfdruck hin ausgeführt werden können.

Textfelder für Ein- und Ausgabe

Zur Ein- und Ausgabe von Zahlen und Texten stellt HTML **Textfelder** zur Verfügung. Textfelder können aber nicht allein, sondern nur als Bestandteile eines Formulars auftauchen. Wenn du ein Textfeld mithilfe von Frontpage Express herstellst, wird ein solches Formular automatisch erzeugt. In Abb. 2 erkennt man zwei Textfelder, eines für den Nettopreis, das andere für die Mehrwertsteuer. Es ist eingerahmt von einem gestrichelten Rechteck, welches das **Formular** anzeigt. Unser Ziel ist es, dass die Webseite zu jedem eingegebenen Nettopreis auf Knopfdruck die zugehörige Mehrwertsteuer anzeigt.

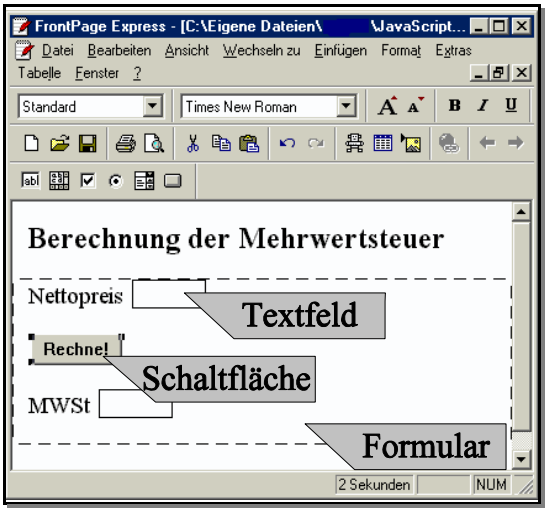


Abb. 2: Ein Formular in Frontpage Express

Der zugehörige Abschnitt des HTML-Quelltexts lautet:

```
<form name="mwstform">
<p>Nettopreis <inputtype="text" size="20" name="nettopreis"></p>
<p><input type="button" name="rechnen" value="Rechne!"></p>
<p>MWSt <input type="text" size="20" name="mwst"></p>
</form>
```

Ein Formular wird durch die Tags <form> und </form> definiert. Zwischen diesen Tags stehen die Tags der Textfelder und Schaltflächen sowie Texte, welche zu diesem Formular gehören sollen. Wenn Frontpage Express ein solches Formular erzeugt, fügt es automatisch das Attribut `method=″post″` beim Formular-Tag ein. Dieses Attribut muss entfernt werden, da sonst der Browser versucht, die Inhalte des Textfeldes über das Netz zu verschicken.

Wenn wir mit JavaScript auf dieses Formular Bezug nehmen wollen, muss es einen Namen erhalten; wir haben es hier "mwstform" genannt. Diese Änderungen kannst du auch mit Frontpage Express vornehmen, indem du im Kontextmenü (Zu Erinnerung: das erreichst du immer über die rechte Maustaste!) `Formulareigenschaften` wählst. Es erscheint dann ein Fenster, in welchem die gewünschten Änderungen vorgenommen werden können. Wenn man sich einmal an den Umgang mit den Tags gewöhnt hat, ist es aber einfacher und übersichtlicher, diese Änderungen in einem Editor wie z. B. Notepad vorzunehmen.

Innerhalb des <form>-Tags finden wir <input>-Tags. Diese Tags kennzeichnen offensichtlich nicht nur Textfelder, sondern auch Schaltflächen. Die wichtigsten Attribute sind in der folgenden Tabelle zusammen gefasst:

Attribut	**Bedeutung**
name = ...	Kennzeichnet den Namen
type = ...	Der Attributwert "text" erzeugt ein Textfeld, der Attributwert "button" erzeugt eine Schaltfläche.
value = ...	Bei einem Textfeld kennzeichnet dies den Inhalt, bei einer Schaltfläche die Beschriftung.
size = ...	Breite des Objektes

Mit Frontpage Express können diese Attribute verändert werden, indem man das Objekt doppelt anklickt. Wenn du dich erst einmal mit dem <input>-Tag vertraut gemacht hast, wirst du wahrscheinlich auch hier einen einfachen Editor bevorzugen.

Nachdem du das Dokument nun abspeicherst und mit einem Browser geöffnet hast, funktionieren schon einige Teile: Du kannst in den Textfelder bereits Eingaben machen, und du kannst auch schon die Schaltfläche drücken - aber natürlich passiert dadurch noch gar nichts. Dazu muss dem Browser erst noch mitgeteilt werden, was er denn nun machen soll, wenn diese Schaltfläche angeklickt wird. Und hier kommt endlich JavaScript ins Spiel. Wie das geht, davon handelt der nächste Abschnitt.

Aufgaben

1. Erstelle mit Frontpage Express ein Formular wie in Abb. 2. Füge noch ein Textfeld für den Bruttopreis hinzu. Achte darauf, dass die Attribute des Formulars und der Input-Tags korrekt gesetzt sind. Kontrolliere das Ergebnis mit einem Browser. Betrachte auch den Quelltext.

2. Verändere im Dokument von Aufgabe 1 das size-Attribut. In welcher Einheit wird hier die Breite angegeben?

Ein Knopfdruck löst Anweisungen aus

Die Berechnung der Mehrwertsteuer übernimmt eine JavaScript-Funktion. Was soll diese Funktion leisten? Nun, sie soll

1. den Wert aus dem Textfeld `nettopreis` holen und in einer Variablen `np` abspeichern,
2. diesen Wert mit 0,16 multiplizieren und das Ergebnis in einer Variablem `ms` speichern,
3. Den Wert der Variablen `ms` im Textfeld `mwst` anzeigen.

Zunächst muss geklärt werden, wie man von JavaScript aus an den Inhalt des Textfeldes `nettopreis` gelangt. Um diese Information zu finden, muss als Erstes das Formular genannt werden, in welchem sich unser Textfeld befindet. Ein solches Formular können wir uns – wie bei den Variablen – als einen Behälter vorstellen. Allerdings befinden sich in diesem Behälter nicht nur ein einziger Wert, sondern gleich mehrere Objekte, nämlich unsere drei Input-Objekte: zwei Textfelder und eine Schaltfläche. Aber auch diese Input-Objekte kann man wieder als Behälter ansehen, in denen sich auch wieder mehrere Angaben finden: der Typ (Textfeld oder Schaltfläche), der Inhalt bzw. die Beschriftung (value) sowie die Breite (size).

Um also an den Inhalt des Textfeldes `nettopreis` zu gelangen, müssen der Reihe nach mehrere Behälter geöffnet werden. Man fügt nun einfach die Etikettenangaben, d. h. die Namen dieser Behälter jeweils mit einem Punkt zusammen; dabei beginnt man links mit der äußersten Behälter und arbeitet sich der Reihe nach bis zum innersten durch (Abb. 3). Den Inhalt des Textfeldes `nettopreis` vom Formular `mwstform` erhält man also durch:

```
mwstform.nettopreis.value
```

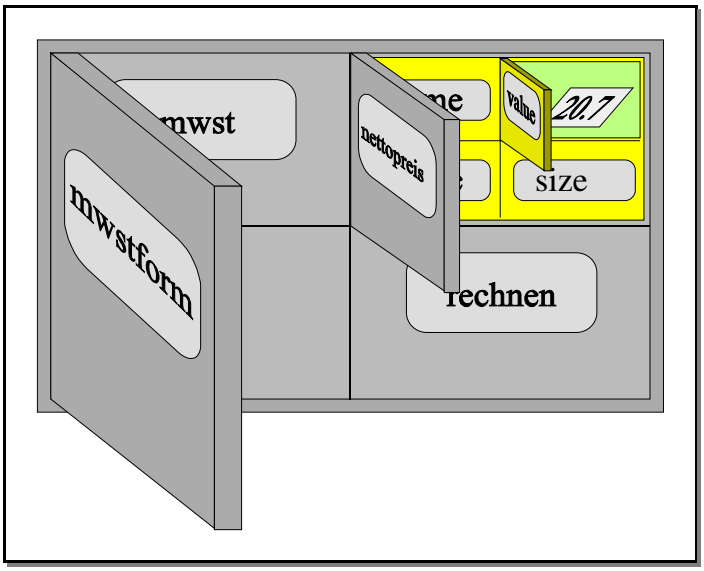


Abb. 3: Das Objekt `mwstform`

Alle Behälter, die in dem Behälter von `mwstform` liegen, werden **Eigenschaften** von `mwstform` genannt. Das Textfeld `nettopreis` ist demnach eine Eigenschaft von `mwstform`. Aber auch unser Textfeld `nettopreis` ist selbst wieder ein Behälter, welcher verschiedene Eigenschaften besitzt. Derartige Behälter bezeichnet man als **Objekte**. Objekte haben also verschiedene Eigenschaften, welche selbst auch wieder Objekte sein können.

Mit diesen Kenntnissen fällt es nun nicht mehr schwer, unsere JavaScript-Funktion zu schreiben; beachte dabei aber, dass in JavaScript 0.16 statt 0,16 geschrieben wird.

```
function mwstberechnen()
{
  var np;
  var ms;
  np = mwstform.nettopreis.value;
  ms = np*0.16;
  mwstform.mwst.value = ms;
}
```

Würden wir diese Funktion einfach in unser HTML-Dokument einfügen, würde der Browser sie lediglich als Text auf dem Bildschirm anzeigen, aber nicht ausführen können. Damit der Browser sie nun als JavaScript-Anweisungen auffasst, wird die Funktion in **<script>**-Tags gesetzt:

```
<script language = "JavaScript">

function mwstberechnen()
{
  ...
}

</script>
```

Solche <script>-Tags können praktisch überall im HTML-Dokument eingebaut werden. Man setzt sie jedoch üblicherweise an das Ende des HEAD-Teils oder an den Anfang des BODY-Teils.

Ein letzter Schritt bleibt noch übrig: Wir müssen dem Broser noch mitteilen, dass er die Funktion `mwstberechnen` ausführen soll, wenn unsere Schaltfläche angeklickt wird. Dazu fügen wir in den <input>-Tag dieser Schaltfläche ein weiteres Attribut ein:

```
onclick = "mwstberechnen()"
```

Nun ist unsere Website zur Mehrwertsteuerberechnung fertig. In Abb. 4 ist noch einmal der gesamte Quelltext angegeben; dabei wurde auf die Schriftformatierung verzichtet.

```
<html>

<head>
<title>Berechnung der Mehrwertsteuer</title>
</head>

<body>

<script language = "JavaScript">

function mwstberechnen()
{
  var nb; var ms;
  nb = mwstform.nettopreis.value;
  ms = nb*0.16;
  mwstform.mwst.value = ms;
}

</script>

<p>Berechnung der Mehrwertsteuer</p>

<form name="mwstform">
    <p>Nettopreis <input type="text" size="20"
          name="nettopreis"></p>
     <p><input type="button"  value="Rechne!"
          onclick="mwstberechnen()"></p>
    <p>MWSt <input type="text" size="20" name="mwst"></p>
</form>
</body>
</html>
```

Abb. 4: Das fertige HTML-Dokument für die Mehrwertsteuer-Berechnung

Aufgaben

1. Das Ansprechen von Objekteigenschaften ähnelt einer Adressenangabe. Nenne Gemeinsamkeiten und Unterschiede. Wie würde deine Adresse in der Objektsprache lauten?

2. Wie spricht man die Beschriftung der Schaltfläche mit dem Namen „pressme“ in dem Formular „testform“ an?

3. Das Attribut language = „JavaScript“ im <script>-Tag lässt darauf schließen, dass es auch noch andere Sprachen gibt, mit denen Browser zusammen arbeiten. Versuche, mithilfe des Internets eine weitere derartige Skriptsprache ausfindig zu machen.

4. Schreibe eine Webseite wie die in Abb. 2, welche zu einem Nettopreis sowohl die Mehrwertsteuer als auch den Bruttopreis berechnet.

5. Was geschieht, wenn man im Quelltext von Aufgabe 4 das </script>-Tag vergisst? Teste aus!

Das EVA-Prinzip

Wie schon in unserem Beispiel mit dem Mehrwertsteuerrechner kann man Programmieraufgaben oft in drei Schritten lösen:

Eingabe : Die Daten des Benutzers werden aus dem Formular abgeholt und Variablen zugewiesen.

Verarbeitung : In einem oder mehreren Schritten werden die erforderlichen Berechnungen ausgeführt.

Ausgabe : Die Ergebnisse werden die vorgesehenen Felder des Formulars geschrieben.

Diese Art der Vorgehensweise wird **EVA-Prinzip** genannt. Bei der Eingabe und der Ausgabe muss man besonders auf eine korrekte Adressierung der Objekte achten, bei der Verarbeitung auf die richtigen Terme. Diese ergeben sich aus der jeweiligen Aufgabenstellung. Die folgende Tabelle zeigt dir eine Auswahl der Zeichen und Funktionen, die in JavaScript benutzt werden können.

Operation	Zeichen	Beispiel	Ergebnis
Addition	+	3 + x	7
Subtraktion	-	7 - 12	- 5
Negation	-	- (3 - 9)	6
Multiplikation	*	3 * 7	21
Division	/	21 / 3	7
Wurzel	Math.sqrt	Math.sqrt(9)	3
Potenz	Math.pow	Math.pow(2,3)	8
Runden	Math.round	Math.round(5.7)	6

Bei der Addition können allerdings manchmal Probleme auftreten; in diesen Fällen wird das Pluszeichen von JavaScript nicht richtig gedeutet. Warum das so ist, wirst du in einem späteren Kapitel erfahren. Mit einem einfachen Trick kannst du diese Schwierigkeiten aber umgehen:

Schreibe statt `a+b` immer `a*1+b*1`.

Das EVA-Prinzip sollte auch bei der optischen Gestaltung eines Formulars berücksichtigt werden: Unter einer Überschrift sollten zunächst die Eingabefelder mit entsprechenden Erläuterungen stehen. Darunter schließen sich die Schaltflächen an, welche die Bearbeitung auslösen. Erst danach folgen die Ausgabefelder (s. Abb. 2).

Aufgaben

1. Larissa hat von ihrer älteren Schwester einen alten quaderförmigen Schrank erhalten. Er ist 80 cm breit, 60 cm hoch und 40 cm tief. Sie möchte den Schrank schwarz lackieren. Um die richtige Menge an Farbe zu kaufen, will Larissa die zu streichende Fläche berechnen (ohne Rückseite und Unterseite). Schreibe dazu ein Programm, welches für diesen und auch für andere quaderförmige Schränke die Rechnung durchführt.

2. Schreibe ein Programm, welches aus der Anzahl der Jahre, dem Zinssatz (in %) und dem Kapital den Zins a) ohne Zinseszins b) mit Zinseszins berechnet.

3. Banken benötigen für die Zinsberechnung die Anzahl der Tage zwischen zwei Kontobewegungen. Man gibt ein:
   von Tag - Monat - Jahr
   bis Tag - Monat - Jahr (in einzelnen Eingabefeldern)
   Ausgabe soll die Zahl der Tage sein, wobei (laut Handelsgesetz) jeder Monat mit 30 und das Jahr mit 360 Tagen gerechnet wird.

4. Schreibe für den rechts abgebildeten Rechenknecht den nötigen HTML-Code und die JavaScript-Funktion `loesen()`. Teste ihn für verschiedene Koeffizienten aus. Wieso ergeben sich manchmal Probleme?

5. Berücksichtigt das Formular in Abb. 5 das EVA-Prinzip? Begründe deine Antwort.

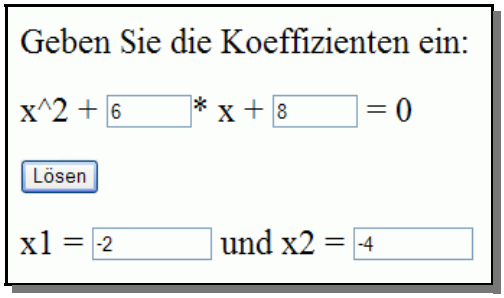


Abb. 5: Zu Aufgabe 4

Debuggen

Unter Debuggen versteht man die Beseitigung von Programmierfehlern. Vielleicht ist dir bei der bisherigen Arbeit schon einmal passiert, dass der Browser nicht deinen Anweisungen folgen wollte und dies mit einer Meldung wie in Abb. 6 anzeigte. Dann liegt in deinem Programm ein Fehler vor. Auch dem besten Programmierer unterlaufen hin und wieder Fehler. Er weiß aber, wie man sie vermeiden und wie man sie gegebenenfalls auch ausfindig machen kann.

Abb. 6: Ein solches Fenster erscheint, wenn du das kleine Warndreieck in der Statuszeile anklickst.

Wesentliche Mittel zur **Vermeidung von Fehlern** hast du schon kennen gelernt:

- die Modularisierung; durch sie bleiben einzelnen Abschnitte überschaubar und auch überprüfbar;
- die Benutzung von sinnvollen Namen für Variablen und Funktionen; so können Missverständnisse vermieden werden;
- die Benutzung lokaler Variablen; so können unbeabsichtigte Beeinflussungen von Funktionen verhindert werden.

Wie aber findet man einen Fehler, wenn er sich nun einmal in das Programm eingeschlichen hat? Manche Fehler findet der Browser; das sind die **syntaktischen Fehler**. Bei diesen wird gegen die Regeln der Programmiersprache JavaScript verstoßen. Das kann z.B. eine fehlende geschweifte Klammer für einen Anweisungsblock sein oder der Aufruf einer Objekteigenschaft, welche es nicht gibt. In Abb. 6 wurde auf die Eigenschaft `mwdt` des Formulars `mwstform` zurückgegriffen – vermutlich ein Tippfehler; richtig müsste es hier natürlich `mwst` lauten.

Leider zeigt der Browser Fehler nicht immer so treffend mit der passenden Zeilennummer an wie in diesem Fall. Das liegt daran, dass er manchmal erst die Spätfolgen, nicht aber den eigentlichen Fehler entdeckt. Deswegen lohnt es sich immer, auch solche Zeilen einer eingehenden Untersuchung zu unterziehen, welche vor der beanstandeten Zeile stehen.

Schwieriger zu finden sind sogenannte **logische Fehler**. Hier läuft das Programm zwar ohne Fehlermeldung, es liefert aber nicht die gewünschten Ergebnisse; manchmal erscheint im Ausgabefeld statt einer Zahl die Anzeige `NaN` (Not a Number = keine Zahl) oder `undefined` (= undefiniert). Häufige Ursachen dafür sind fehlerhafte Terme, Probleme mit lokalen und globalen Variablen oder auch eine falsche Reihenfolge der Befehle. Derartige Fehler zu finden, ist schon etwas schwieriger. Folgende Vorgehensweisen haben sich in solchen Fällen zum **Lokalisieren von Fehlern** bewährt:

- Weniger wichtigen Anweisungen ein // voranstellen; sie werden dann nicht bearbeitet. Das bezeichnet man als **Auskommentieren**.
- Mit der Anweisung `alert(mwst)` kann der Inhalt von Zwischenergebnissen, hier der Variablen `mwst`, in einem **Meldungsfenster** angezeigt werden.
- Das Programm auf die häufigsten Fehler hin durchsuchen (Groß/Kleinschreibung, Schleifen mit globalen Variablen, Funktionsname ohne Klammern)

Aufgaben

1. „Debuggen“ heißt wörtlich übersetzt „Entwanzen“. Versuche herauszubekommen, woher diese Bezeichnung stammt.

2. Bei dem Programm in Abb. 7 gibt es drei Fehler. Finde und korrigiere sie.

```
function woSindDie Fehler(x);
{
    VAR z = 5*x;
    Textfeld.value = z;
}
```

Abb. 7: Wo sind die drei Fehler?

3. Versuche, die Anzeige `NaN` bzw. `undefined` in einem Anzeigefenster zu erzeugen.

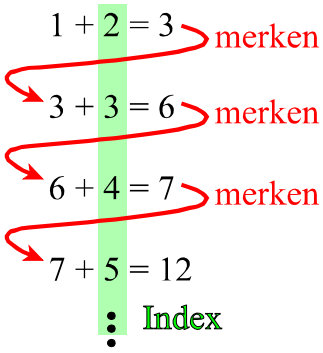
4. Worin liegt der Unterschied zwischen logischen und syntaktischen Fehlern?

Schleifen mit Zwischenspeicher

Carl Friederich Gauss (Abb. 8) war einer der genialsten Mathematiker, die je gelebt haben. Seine Begabung zeigte sich schon früh: Eines Morgens – so die Legende – bekam der kleine Gauss zusammen mit seinen Mitschülern eine längere Aufgabe gestellt. Böse Zungen behaupten, der Lehrer habe nach einer durchzechten Nacht die Schüler beschäftigen und sich selbst etwas Ruhe gönnen wollen! Die Aufgabe lautete: Addiere alle Zahlen von 1 bis 100. Du kennst sicherlich den Trick, mit dessen Hilfe Gauss die Aufgabe in kürzester Zeit erledigte. Hier nun soll JavaScript diese Aufgabe ganz stur rechnend erledigen.

Abb. 8: C. F. Gauss

Natürlich könnte man die gesamte Aufgabe in der Form 1 + 2 + 3 + 4 + ... eingeben. Das wäre aber sehr mühselig, erst recht wenn man diese Vorgehensweise auf 1000 oder mehr Summanden anwenden wollte. Deswegen zerlegen wir zunächst die Summationsaufgabe in viele kleine Teilaufgaben, so wie wir sie auch im Kopf lösen würden:



Die zu merkenden Zahlen speichern wir jeweils in einer Variablen. Weil wir nur immer das letzte Ergebnis brauchen, können wir dazu immer dieselbe Variable nehmen; die nicht mehr gebrauchten Werte werden dann überschrieben. Wenn wir vor Beginn der Rechnung die Zahl 1 in dieser Variablen speichern, laufen alle Rechnungen nach demselben Schema ab:

> *Addiere zu der gespeicherten Zahl jeweils eine weitere Zahl i und speichere das Ergebnis wieder in derselben Variablen.*

Diese weitere Zahl i ändert sich von Rechnung zu Rechnung: Sie beginnt bei 2 und wird jeweils um 1 erhöht. Zur Programmierung bietet sich deswegen eine Schleife an. Diese Programmiertechnik haben wir schon im letzten Kapitel kennen gelernt. Die Funktion, welche diese Summe berechnet, lautet:

```
function addieren()
{
  var bis = 100;
  var von = 2;
  var zwischensumme=1;
  for (i=von; i <= bis; i=i+1)
  {
     zwischensumme = zwischensumme + i;
  }
  gaussform.summe.value = zwischensumme;
}
```

In der Variablen `zwischensumme` werden die Zwischenergebnisse gespeichert. Beim ersten Schleifendurchlauf ist ihr Wert zunächst 1. Die Zahl i = 2 wird addiert, das Ergebnis wieder in der Variable `zwischensumme` gespeichert. Beim nächsten Durchlauf wird zu diesem Wert i = 3 addiert und das Ergebnis, also die Zahl 6, wieder in der Variablen `zwischensumme` gespeichert. So geht das weiter. Beim 14. Schleifendurchlauf ist dann zu Beginn der Schleife in `zwischensumme` die Zahl

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14

gespeichert. Nun wird die Zahl 15 addiert. Am Ende dieses Schleifendurchlaufs ist in der Variablen `zwischensumme` also die Summe der ersten 15 natürlichen Zahlen gespeichert. Beim Durchlauf der letzten Schleife ist der Index 100. Am Ende sind also sämtliche ganzen Zahlen von 1 bis 100 addiert worden.

Aufgaben

1. Die Summe aller natürlichen Zahlen von 1 bis zu einer einzugebenden Zahl *e* soll ermittelt werden.

2. Die natürlichen Zahlen zwischen *a* und *b* sollen multipliziert werden.

3. Schreibe ein Programm, welches zwei ganze Zahlen miteinander multipliziert, ohne das Multiplikationszeichen * zu benutzen.

4. Die Funktion in der Abb. 9 wird mit den Parametern x = 4 und y = 3 aufgerufen. Welchen Wert besitzt die Variable z am Ende?

```
function f(x, y)
{
     var z = x;
     for (var i=y; i≥1; i++)
     {
          z = z + 1;
     }
}
```

Abb. 9: Zu Aufgabe 4