Wir verschlüsseln Texte

Schon im Altertum hat man versucht, Botschaften zu verschlüsseln, um sie so vor Feinden geheim zu halten. Schließlich musste man damit rechnen, dass der Überbringer der Nachricht abgefangen und die Botschaft in falsche Hände geraten konnte.

So schrieb man den zu übermittelnden Text nach einem geheimen Verfahren, dem **Code**, zunächst um. Das bezeichnet man als **Kodieren**. Häufig geschieht dies durch systematisches Austauschen der einzelnen Buchstaben des Textes. Cäsar z. B. soll dabei folgendes Verfahren benutzt haben:

A	B	C	...	W	X	Y	Z
⬇	⬇	⬇		⬇	⬇	⬇	⬇
D	E	F	...	Z	A	B	C

Der Empfänger der so kodierten Botschaft muss diese nun wieder entschlüsseln, indem er das beim Kodieren benutzte Verfahren umkehrt. Das nennt man **Dekodieren**. Versuche einmal die in Abb. 1 dargestellte Botschaft zu dekodieren.

Natürlich gibt es inzwischen wesentlich komplexere Verschlüsselungen. Teilweise sind die Verfahren derart zeitaufwendig, dass man sie nur noch mithilfe eines Computers bewältigen kann. Umgekehrt ist es aber auch so, dass man einen Computer ohne Kodierung gar nicht sinnvoll benutzen kann. Die grundlegenden Informationseinheiten, mit denen der Computer arbeitet, sind nämlich die beiden Zustände STROM EIN (kurz: 1) und STROM AUS (kurz: 0). Will der Mensch nun mit dem Computer Informationen austauschen, muss er sie mithilfe dieser beiden Zeichen verschlüsseln, z. B. durch die Kodierung

A	B	C	D	...
⬇	⬇	⬇	⬇	
00001	00010	00011	00100	...

Zu Beginn des Computerzeitalters wurde eine solche Kodierung tatsächlich noch manuell von Menschen durchgeführt. Heute wird das Kodieren üblicherweise von der Tastatur und einem zusätzlichen Hilfsprogramm, dem Tastaturtreiber, übernommen; zur Ausgabe des Textes auf dem Monitor muss wieder dekodiert werden, dabei helfen die Grafikkarte und die Grafikkartentreiber.

Wir sehen: Der Computer ist nicht nur ein wichtiges Hilfsmittel bei der Verschlüsselung, vielmehr ist eine sinnvolle Arbeit am Computer ohne Kodierung und Dekodierung gar nicht denkbar.

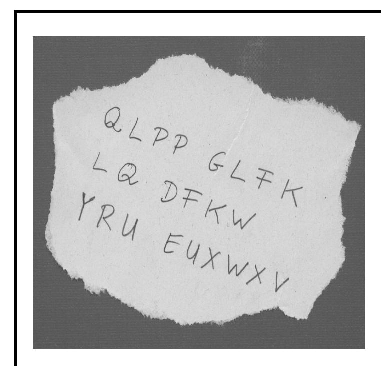

Abb. 1: Geheimbotschaft an Cäsar

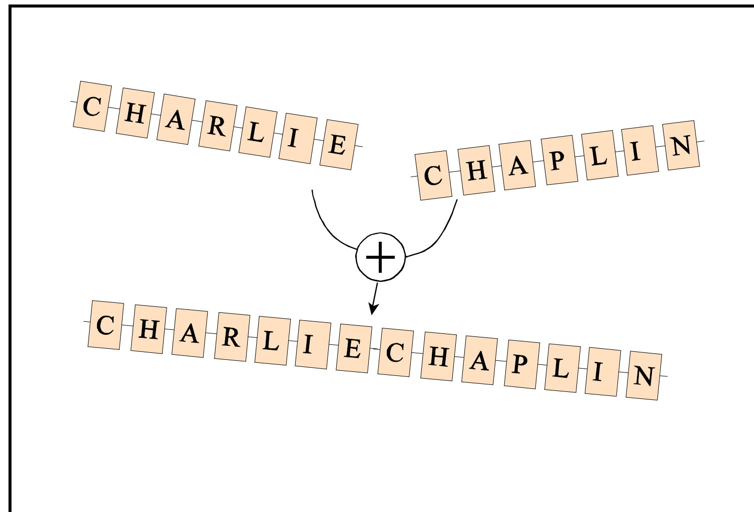

Abb. 2: Verketten von Zeichenketten

Zeichenketten und Zeichenkettenfunktionen

Um Texte zu verarbeiten, benutzt man sogenannte **Zeichenketten**. Diese stellt man sich am besten – der Name deutet es schon an – als Kette von Steinen vor, die jeweils mit einem Zeichen beschrieben sind. Dabei können die Zeichen nicht nur große und kleine Buchstaben, sondern auch Sonderzeichen wie Punkt, Komma, Leerzeichen oder auch Rechenzeichen sein. Lediglich der fallende Schrägstrich (\) bildet hier eine Ausnahme, weil er zu Formatierungszwecken eingesetzt wird. Wir werden in einem späteren Abschnitt darauf zurückkommen. Für Zeichenketten benutzt man auch oft die englische Bezeichnung „Strings“.

Zeichenketten können, wie wir es auch schon von den Zahlen her kennen, in Variablen abgespeichert werden. Durch

```
var meinText = "Das ist eine Zeichenkette";
```

wird der Text, der zwischen den Anführungszeichen steht, in der Variablen `meinText` abgespeichert. Die Anführungszeichen sind dabei sehr wichtig; ohne sie würde JavaScript den rechts vom Gleichheitszeichen stehenden Text als Namen einer Variablen deuten und versuchen, dessen Inhalt unter `meinText` abzuspeichern.

Zeichenketten können auch aus einem Eingabefeld gelesen werden. Dies geschieht genauso wie bei Zahlen. Bei der Eingabe des Textes brauchen dabei keine Anführungszeichen gesetzt zu werden. JavaScript erkennt in der Regel automatisch, ob es sich um eine Zahl oder um einen Text handelt.

Zeichenketten lassen sich nun auf vielfältige Art und Weise bearbeiten. Z. B. kann man zwei Zeichenketten zu einer einzigen zusammenfügen; man bezeichnet dies als **Verkettung**. Anschaulich betrachtet wird dabei der Anfang der zweiten Zeichenkette an das Ende der ersten angesetzt (s. Abb. 2). In JavaScript geschieht das durch den Plus-Operator:

```
var s1 = "CHARLIE";
```

```
var s2 = "CHAPLIN";
var s = s1 + s2;
```

In der Variablen `s` liegt dann die Zeichenkette „CHARLIECHAPLIN“. Beachte, dass jetzt zwischen CHARLIE und CHAPLIN kein Leerzeichen ist. Wenn man dies haben möchte, muss die letzte Programmzeile durch

```
var s = s1 + " " + s2;
```

ersetzt werden.

In der Abb. 3 wurden die Inhalte der oberen beiden Felder durch ein + verknüpft, das Ergebnis im unteren Textfeld ausgegeben. 1+1 = 11? Das kann doch nicht stimmen! Wenn du das denkst, hast du vollkommen recht. Aber auf der anderen Seite hat hier JavaScript auch keinen Fehler gemacht. Die beiden Eingaben wurden hier lediglich als Zeichenketten gedeutet und durch den Plus-Operator verkettet; das kann man leicht mit weiteren Beispielen nachprüfen (`source\texte\plus.htm`). Tatsächlich deutet JavaScript alle Eingaben zunächst als Zeichenketten. Erst wenn es unbedingt nötig ist, werden sie als Zahlen gedeutet.

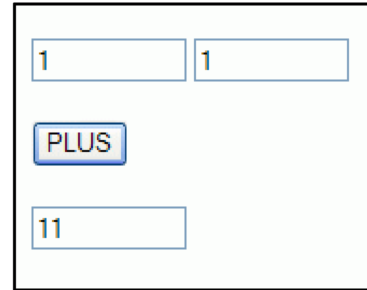

Abb. 3: Ist 1+1=11?

Wann liegt nun ein solcher Fall vor? Das ist immer dann der Fall, wenn mit den Inhalten etwas gemacht wird, was man nur mit Zahlen machen kann, z.B. Multiplizieren oder Subtrahieren. Wenn in dem Programm von Abb. 3 also das Pluszeichen durch ein Multiplikationszeichen ersetzt wird, werden die eingegebenen Zahlen wirklich multipliziert. Es gibt nämlich für die Zeichenketten weder eine Multiplikation noch eine Subtraktion; die Deutung als Zeichenkette muss an dieser Stelle also scheitern, es bleibt die Deutung als Zahlen. Letztendlich liegt die Ursache für das Problem mit unserer 1+1-Rechenaufgabe natürlich darin, dass hier die Addition von Zahlen und die Verkettung von Texten mit ein und demselben Zeichen gekennzeichnet werden.

Bedeutet dies nun, dass wir zwei eingegebene Zahlen gar nicht addieren können? Nein, wir können nämlich die Deutung als Zahlen zu erzwingen. Am einfachsten gelingt dies, indem man die Eingaben jeweils mit 1 multipliziert:

```
pform.s.value = pform.s1.value * 1 + pform.s2.value * 1;
```

Wegen der Multiplikation müssen jetzt die Inhalte der Eingabefelder als Zahlen gedeutet werden. Jetzt werden die Inhalte tatsächlich korrekt addiert.

Den Plus-Operator können wir zum Verschlüsseln gut einsetzen. Mit ihm können wir die einzeln kodierten Buchstaben wieder zu einer einzigen Zeichenkette zusammensetzen. Vorher müssen wir allerdings die ursprüngliche Zeichenkette erst in einzelne Buchstaben zerlegen und diese dann kodieren. JavaScript stellt hierzu eine Reihe von Funktionen bereit. In der folgenden Tabelle ist eine Auswahl von solchen Funktionen zusammengefasst. Dabei steht <Zk> für den Variablennamen einer Zeichenkette; die Variable `n` im Beispiel soll den Inhalt “Maier” haben.

Anweisung	Bedeutung	Beispiel	Ergebnis
<Zk>.length	Länge der Zeichenkette	n.length	5
<Zk>.charAt(i)	i-tes Zeichen	n.charAt(3)	“e”
<Zk>.indexOf(Zeichen)	Index des Zeichens	n.indexOf(“a”)	1
<Zk>.substring(Anfang, Ende)	Teilstring	n.substring(1,3)	“ai”
<Zk>.substr(Anfang, Anzahl)	Teilstring	n.substr(3,2)	“er”
<Zk>.toLowerCase()	Großbuchstaben	n.toLowerCase()	“maier”
<Zk>.toUpperCase()	Kleinbuchstaben	n.toUpperCase()	“MAIER”

Besonders zu beachten ist, dass die zu bearbeitende Zeichenkette mit ihrem Variablennamen nicht als Parameter der Funktion auftaucht; vielmehr wird sie – durch einen Punkt abgesetzt – dem Funktionsnamen vorangestellt. Diese Schreibweise erinnert an unsere Igel-Anweisungen und die Eigenschaften von Objekten. Tatsächlich sind sowohl der Igel als auch die Zeichenketten ebenfalls Objekte. Eine Eigenschaft des Zeichenkettenobjekts ist die Länge (length). Bei den restlichen Anweisungen der Tabelle handelt es sich um sogenannte **Methoden**. Das sind Funktionen, welche sozusagen Privateigentum jedes einzelnen Zeichenkettenobjekts sind. Darauf werden wir in einem späteren Kapitel noch genauer eingehen.

Als Beispiel für die Anwendung dieser Methoden wollen wir nun das Programm aus Abb. 3 folgendermaßen abändern: Wenn in den oberen beiden Eingabefeldern der Vor- und Nachname einer Person eingegeben wird, dann soll in dem unteren Feld als Ergebnis die Abkürzung des Vornamens, gefolgt von dem Nachnamen, angezeigt werden. Die Felder mögen wieder in dem Formular `pform` liegen und die Namen `s1`, `s2` und `s` haben.

Zunächst müssen wir aus dem Vornamen den ersten Buchstaben herausholen. Dazu benutzen wir die Methode `charAt(i)`. Der Parameter i gibt dabei die Position des Buchstabens in der Kette an; er wird auch **Index** genannt.

```
var vorname = pform.s1.value;
var ersterBuchstabe = vorname.charAt(0);
```

Die Methode `vorname.charAt(0)` liefert den ersten Buchstaben der Zeichenkette `vorname`. Der Index ist hier 0 und nicht etwa 1, weil die Zählung von Indizes immer bei 0 beginnt. Entsprechendes gilt übrigens auch für die `indexOf`-Methode.

Anschließend erfolgt die Verkettung und die Ausgabe:

```
var ergebnis = ersterBuchstabe + ". " + pform.s2.value;
pform.s.value = ergebnis;
```

Aufgaben

1. Probiere die verschiedenen Zeichenketten-Methoden an Beispielen aus. U. A. soll der eingegebene Text in Großbuchstaben [Kleinbuchstaben] ausgegeben werden. Benutze dazu die Datei `source\texte\test.htm`.

2. Was geschieht durch diese Funktion? Erkläre auch die Bedeutung der Variablen `zk`.

```
function wasMacheIch()
{
     var s = textform.text1.value;
     var zk = "";
     var buchstabe;
     for (var k = 0;  k < s.length;  k++)
     {
          buchstabe = s.charAt(k);
          zk = zk + buchstabe + buchstabe;
     }
     textform.text2.value = zk;
}
```

Für die folgenden Aufgaben kannst du den HTML-Quelltext von Aufgabe 1 benutzen und die JavaScript-Funktion entsprechend ändern.

3. Der eingegebene Text soll gesperrt ausgegeben werden, das heißt zwischen je zwei Buchstaben ist ein Leerzeichen einzufügen.

4. Aus einem eingegebenen Satz sollen Kommata und Leerzeichen entfernt werden. (Tipp: Gehe den String durch wie in Aufgabe 2; in den Ausgabestring werden nur diejenigen Zeichen übernommen, die nicht gleich dem Leerzeichen (" ") und nicht gleich dem Komma sind.)

5. Ein eingegebener Text soll verschlüsselt werden, indem nach jedem Zeichen ein „a" gesetzt wird. Schreibe auch ein Entschlüsselungsprogramm und teste es aus.

6. Lasse alle Vokale eines Satzes durch Sternchen ("*") ersetzen.

7. Wie Aufgabe 6; jetzt soll statt des Sternchens ein einzugebender Buchstabe für die Vokale eingesetzt werden.

8. Es soll bestimmt werden, wie häufig der Buchstabe „e" im Eingabestring auftaucht. Der Benutzer soll zwischen den Optionen „absolute Häufigkeit" und „relative Häufigkeit" entscheiden können.

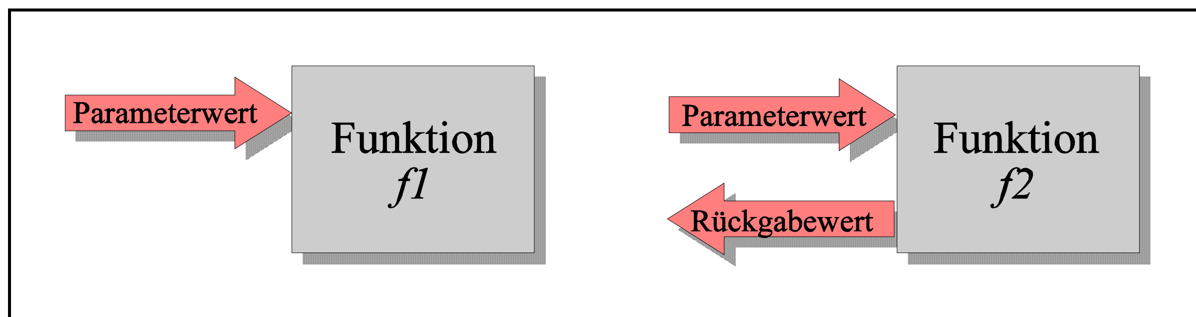

Abb. 4: Funktionen ohne (links) und mit (rechts) Rückgabewert

Funktionen mit Rückgabewert

Die Zeichenkettenbearbeitungsfunktionen aus dem letzten Abschnitt unterscheiden sich in einem wichtigen Gesichtspunkt von den Funktionen, die wir bisher mit JavaScript selbst programmiert haben: Sie besitzen einen Rückgabewert. Abb. 4 verdeutlicht dies: Die Funktion *f1* erhält einen Parameterwert und bearbeitet ihn; gegebenenfalls werden auch Ergebnisse über Textfelder (oder wie beim Igel auch in einem Grafikbereich) dargestellt. Auch die Funktion *f2* erhält einen Wert und bearbeitet ihn, aber sie liefert auch ein Ergebnis zurück. Das kann sie z.B. an eine Variable abgeben; dazu muss sie auf der rechten Seite des Gleichheitszeichens stehen:

```
wert = f2(x)
```

JavaScript-Funktionen mit Rückgabewert ähneln stark den aus der Mathematik bekannten Funktionen. Tatsächlich hatten wir mit der Wurzel-Funktion `Math.sqrt(x)` in einem früheren Kapitel schon eine solche Funktion kennen gelernt.

Wie können wir nun aber selbst solche Funktionen mit Rückgabewert programmieren? Dazu betrachten wir ein einfaches Beispiel. Die Funktion `quadrat(x)` soll das Quadrat des Parameters `x` zurückgeben.

Beispielprogramm	Erläuterung
`function quadrat(x)`	Funktionskopf (Funktionsname = "`quadrat`"; Parameter = `x`)
`{`	
`var y;`	(lokale) Variablendeklaration für `y`
`y = x*x;`	`x` quadrieren und unter `y` speichern
`return y;`	den Wert von `y` an das aufrufende Programm zurückgeben
`}`	

Neu an diesem Programm ist lediglich die vorletzte Zeile: Das Schlüsselwort `return` sorgt hier dafür, dass der Inhalt von `y` an die Programmzeile zurückgegeben wird, welche die Funktion aufgerufen hat. Was das im Einzelnen bedeutet, wollen wir noch einmal klar machen, indem wir die Wertübergaben bildlich darstellen:

Abb. 5 zeigt im Detail, welche Übergaben beim Aufruf der `quadrat`-Funktion stattfinden. Zunächst wird der Parameterwert 3 an die Funktion `quadrat` übergeben. In deren Funktionsrumpf wird nun an allen Stellen `x` mit dem Wert 3 belegt. Die Variable `y` erhält somit den Wert 3 * 3, also 9. Dieser Wert wird nun durch die `return`-Anweisung an die Zeile zurückgegeben, welche die Funktion `quadrat` aufgerufen hat: Die rechte Seite wird durch den Rückgabewert 9 ersetzt. Dieser Wert wird schließlich in der Variablen `ergebnis` abgespeichert. Vereinfachend kann man sich also vorstellen:

Beim Aufruf einer Funktion wird diese durch ihren Rückgabewert ersetzt.

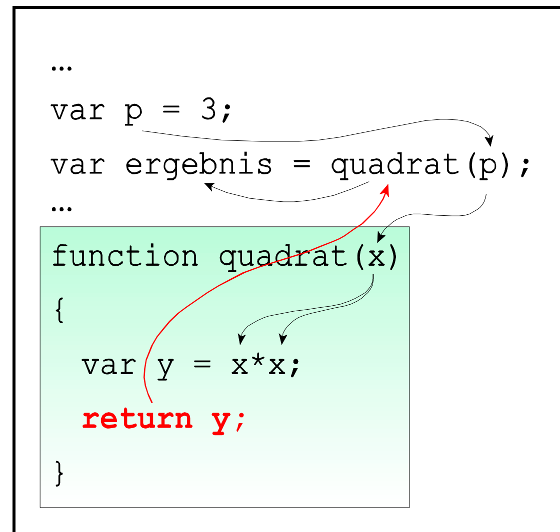

Abb. 5: Wertübergaben bei der Quardat-Funktion

Aufgaben

1. Die Potenz einer Zahl soll berechnet werden. Benutze dazu nicht(!) die `Math.pow`-Funktion. Schreibe dazu vielmehr eine eigene Funktion `potenz(x,n)`.

2. Schreibe eine Funktion `spiegel(s)`, welche als Rückgabewert die gespiegelte Zeichenkette (Text von hinten gelesen) besitzt.

3. Unter einem Palindrom versteht man ein Wort, welches sich von vorne wie von hinten gleich liest. Otto oder Reliefpfeiler sind Beispiele für Palindrome. Schreibe eine Funktion `palindrom(s)`, welche den Wert `true` zurückgibt, wenn `s` ein Palindrom ist, und ansonsten den Wert `false`.

4. Schreibe eine Funktion `fix2(x)`, welche `x` auf zwei Stellen hinter dem Komma gerundet zurückgibt.
   Hinweis: Benutze die Funktion `Math.round(x)`.

5. Schreibe eine Funktion `suche(s1, s2)`, welche angibt, wie häufig eine Zeichenkette `s2` in der Zeichenkette `s1` auftaucht.

6. Schreibe eine Funktion `suche_und_ersetze(text, suche, ersetze)`, welche in der Zeichenkette `text` alle Teilstrings `suche` durch `ersetze` ersetzt und die so entstandene neue Zeichenkette zurückgibt.

Textbereiche und ihre Formatierung

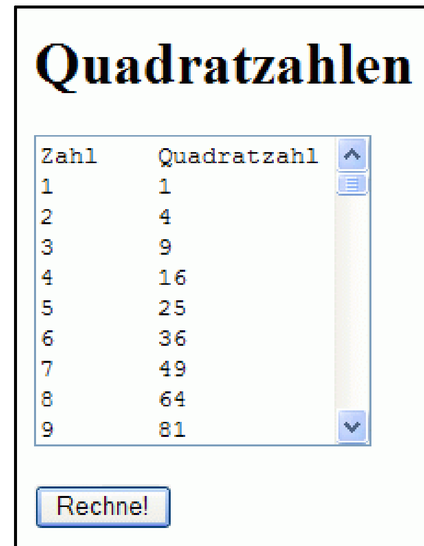

Abb. 6: Quadratzahlentabelle

Mit den bisher benutzten Textfeldern lassen sich weder längere Texte noch Tabellen übersichtlich darstellen. Denn einerseits sind diese Textfelder häufig einfach zu klein; andererseits lassen sie auch keine Formatierung zu. Aus diesem Grund gibt es die so genannten **Textbereiche**, welche diese Beschränkungen nicht haben: Die Abb. 6 macht deutlich, dass wir hier nicht nur mehr als eine Zeile zur Verfügung haben, sondern offensichtlich auch Tabulatoren, die es erlauben, Daten in Tabellenform darzustellen. Die zugehörigen HTML-Tags lauten <TextArea> bzw. </TextArea>; der zwischen den Tags stehende Text wird im Textbereich angezeigt.

Folgende Attribute stehen zur Verfügung:

Attribut	**Beschreibung**
name	Name des Objekts
cols	bestimmt die Breite des Textbereichs (Anzahl der Spalten)
rows	gibt die Anzahl der (sichtbaren) Zeilen an
value	Text des Textbereichs
wrap	legt fest, ob und wann ein Zeilenumbruch erfolgt: wrap="off": kein automatischer Zeilenumbruch; wenn Text zu lang wird, erscheint ein horizontaler Rollbalken wrap="physical": automatischer Zeilenumbruch; in die Zeichenkette werden Zeilenumbruchzeichen eingefügt wrap="virtual": automatischer Zeilenumbruch; in die Zeichenkette werden keine Zeilenumbruchzeichen eingefügt

Zu beachten ist dabei, dass der Text des Anzeigebereichs eine einzige Zeichenkette darstellt. Man kann also nicht auf die einzelnen Zeilen separat zugreifen. Umgekehrt erfolgt auch die Ausgabe eines Textes in einem solchen Textbereich zunächst völlig unformatiert: Der Text wird fortlaufend in die einzelnen Zeilen geschrieben (wenn die automatische Zeilenumbruchfunktion aktiviert ist). Will man die Zeilenumbrüche selbst steuern oder weitere Formatierungsmaßnahmen wie Tabulatoren vornehmen, muss man entsprechende Steuerzeichen in die Zeichenkette einfügen.

Steuerzeichen	**Bedeutung**
\n	Zeilenumbruch (new line)
\t	Tabulatorsprung

Die Liste der Quadratzahlen in Abb. 6 kann man damit z.B. folgendermaßen erstellen lassen:

```
function rechnen()
{
  var ausgabe = "Zahl"+"\t"+"Quadratzahl"+"\n";
  var q;
  for (var k=1; k<=100; k++)
  {
    q = k*k;
    ausgabe = ausgabe + k + "\t" + q + "\n";
  }
  listenform.liste.value = ausgabe;
}
```

Der Tabulator `\t` sorgt dafür, dass die Quadratzahlen alle untereinander stehen. Man könnte auch versuchen, entsprechend viele Leerzeichen zwischen der Zahl und der zugehörigen Quadratzahl einzufügen. Diese Anzahl von Leerzeichen müsste aber jeweils verringert werden, wenn die links stehende Zahl eine Stelle mehr bekommt. Jedes Zeilenende, insbesondere auch die Überschrift, wird mit einem `\n`-Zeichen markiert. Auf diese Art können auch Leerzeilen erzeugt werden.

Beachte, dass hier die Zahlenwerte von k und q automatisch als Zeichenketten verkettet werden.

Aufgaben

1. Erläutere die Bedeutung der Variable `ausgabe` in der Funktion `rechnen()`.

2. Erstelle ein Programm für eine Kapitaltabelle. Es soll Eingabefelder für das Startkapital, die Anzahl der Jahre und den Zins besitzen. In einem Textbereich sollen das Jahr und das zugehörige Kapital tabellarisch angegeben werden.

3. Die Ein- und Ausgabe der Suchen-und Ersetzen-Funktion von Aufgabe 5 des letzten Abschnitts soll in einem Textbereich erfolgen. Teste sie dann folgendermaßen aus: Lade den Text `grimm.txt` aus dem Verzeichnis `source\texte` über einen Editor und die Zwischenablage in den Textbereich; ersetze alle „Märchen" durch „Schweinchen".

4. JavaScript kann eine Zeichenkette als Term auswerten. Das geschieht mit der `eval`-Funktion. Wenn z.B. in der Variablen `x` die Zahl 5 und in der Variablen `term` die Zeichenkette `3*x+7` gespeichert ist, dann ist das Ergebnis von `eval(term)` die Zahl 22. Erstelle eine Webseite, welche eine Wertetabelle erstellt, wenn der Funktionsterm, die beiden Intervallgrenzen und die Schrittweite eingegeben werden.

5. Funktionsgraphen sollen in einem Textbereich wie in Abb. 7 durch Sternchen angezeigt werden.

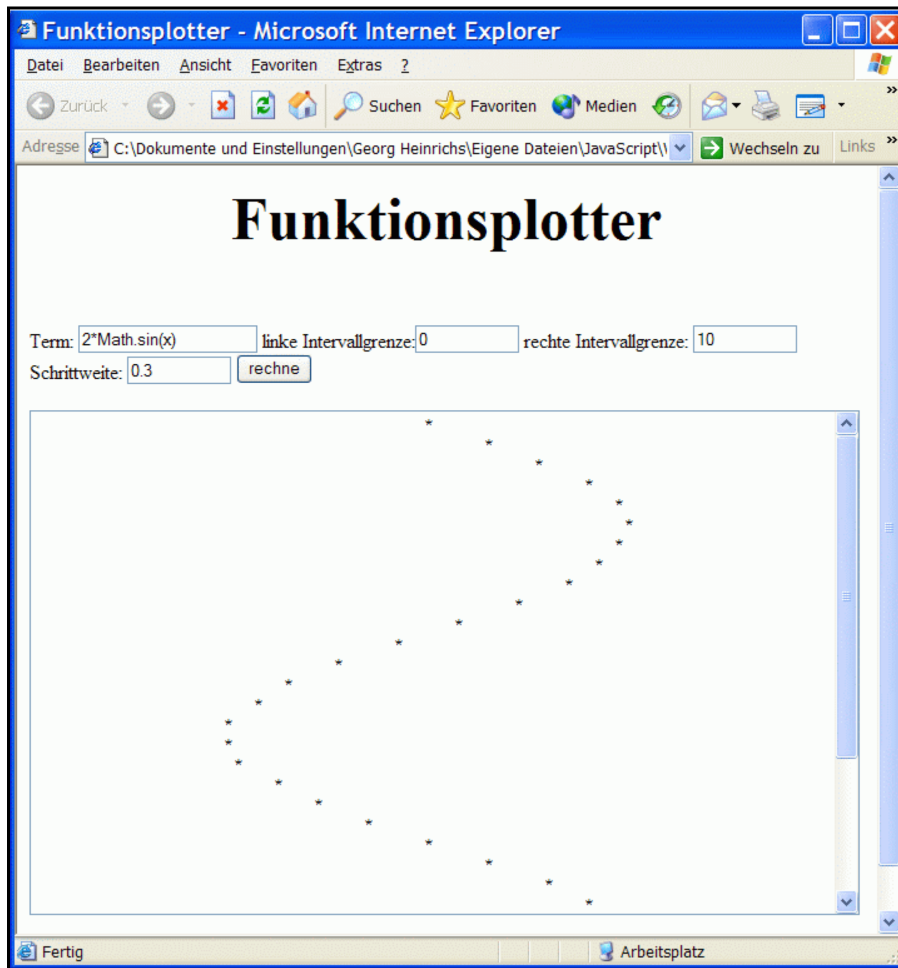

Abb. 7: So kann man mit Sternchen Graphen zeichnen.

Der ASCII-Code

Alle Dinge, die im Rechner gespeichert oder von ihm bearbeitet werden, müssen durch Bits dargestellt sein. Ein Bit ist die elementare Informationseinheit, mit der der Rechner arbeitet: STROM EIN oder STROM AUS bzw. Null oder Eins. Insbesondere müssen auch die einzelnen Zeichen einer Zeichenkette im Rechner als Bits vorliegen. Die Kodierung, welche wohl am häufigsten dabei benutzt wird, ist der ASCII-Code. ASCII steht für *American Standard Code for Information Interchange*. Beim ASCII-Code wird jedes Zeichen durch eine Folge von 8 Bits dargestellt. Wenn man diese Folge als Zahl im Zweiersystem auffasst, dann wird jedem Zeichen also eindeutig eine Zahl zugeordnet. Da nur die ersten 7 Bits für die Kodierung benutzt werden (das letzte Bit ist ein so genanntes Prüfbit), kann man mit diesem Code $2^7 = 128$ Zeichen darstellen:

Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen
32	<leer>	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	’	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	‘	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	M	102	f	126	~
55	7	79	O	103	g	127	DEL

Die Zeichen zu den Codes 0 bis 31 sind hier nicht angegeben. Es handelt sich hier nämlich um so genannte Steuercodes, die keinen ausdruckbaren Zeichen entsprechen. Sie werden vielmehr benutzt, um Fernschreiber oder Drucker zu steuern. Einige dieser Steuerzeichen erzeugen z.B. einen Zeilenvorschub oder einen Seitenwechsel. Das Steuerzeichen BEL soll eine kleine Glocke am Fernschreiber zum Läuten bringen; dadurch kann dem Empfänger das Eingehen einer neuen Meldung angekündigt werden. Auch wenn es praktisch keine Fernschreiber mehr gibt, das Glocken-Steuerzeichen gibt es immer noch...

Die ASCII-Codes können nun den Umgang mit Zeichenketten wesentlich erleichtern: Denken wir z.B. an die Kodierung mit der Cäsarmethode: Jedem Buchstaben wird hier ein Buchstabe zugeordnet, welcher im Alphabet 3 Schritte weiter rechts liegt. Eine Kodierfunktion, bei der jeder einzelne Buchstabe durch eine eigene Anweisung bearbeitet wird, etwa in der Art

```
if (zeichen == "A") { zeichen = "D" };
if (zeichen == "B") { zeichen = "E" };
...
```

wäre einfach zu langwierig. Die Benutzung des ASCII-Codes führt zu einem wesentlich kürzeren Programm:

```
function kodieren()
{
      var q = codeform.quelle.value;
      var code;
      var z = "";
      for (var k=0; k<=q.length-1; k++)
      {
            code = q.charCodeAt(k);
            if (code < 88)
            {
                  code = code + 3;
            }
            else
            {
                  code = code - 23;
            }
            z = z + String.fromCharCode(code);
      }
      codeform.ziel.value=z;
}
```

Zwei Funktionen sind hier neu: Die Methode `q.charCodeAt(k)` liefert den ASCII-Code des k-ten Zeichens der Zeichenkette q; auch hier beginnt die Zählung wieder bei 0. Die Methode `String.fromCharCode(x)` liefert umgekehrt das Zeichen, welches den Code `x` hat. Beachte, dass diese Funktion nicht die Methode eines speziellen Zeichenkettenobjekts ist: Vor dem Methodennamen `fromCharCode` steht deswegen immer derselbe Bezeichner „`String`“.

Wie funktioniert nun unser Kodier-Programm? Zunächst wird der Text aus dem Eingabebereich „quelle“ in der Variablen `q` gespeichert. In einer Schleife wird jetzt der Reihe nach der Codewert jedes einzelnen Buchstabens ermittelt und um 3 erhöht; von A gelangt man so zu D, von B zu E u.s.w. (Abb. 8). Wenn der zu kodierende Buchstabe X, Y oder Z ist, müssen wir wieder an den Anfang des Alphabets springen; in diesem Fall ist der Codewert 88 oder größer und dann subtrahieren wir 23, um zum entsprechenden Buchstaben am Anfang des Alphabets zu gelangen. Aus den so berechneten Codes wird anschließend wieder das zugehörige Zeichen bestimmt; alle so bestimmten Zeichen werden in der Zeichenkette `z` aufgereiht. Am Ende steht in dieser Zeichenkette der gesamte verschlüsselte Text.

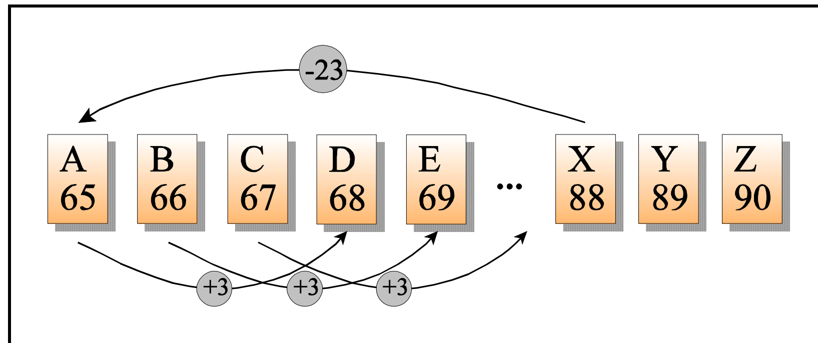

Abb. 8: Cäsars Kodierung im ASCII-Code

Die Modulo-Operation

Bei der Kodierung von Hand hat man lange Zeit Kodierscheiben wie in Abb. 9 benutzt. Dabei ist die innere Scheibe gegenüber der äußeren drehbar, so dass auch andere Verschiebungen im Alphabet realisiert werden können.

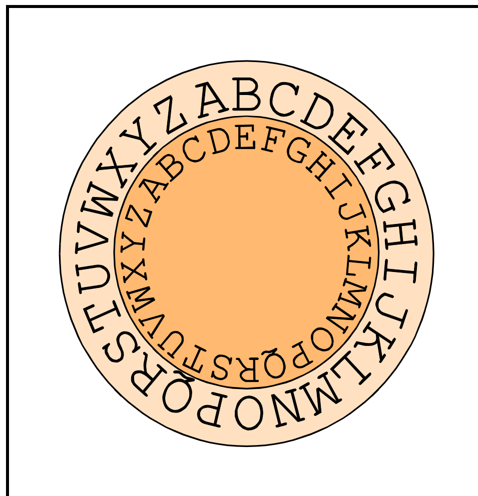

Abb. 9: Kodierscheibe

Die ringförmige Anordnung bringt es mit sich, dass auch am Ende des Alphabets eine Verschiebung um +3 zur richtigen Kodierung führt. Mathematisch kann man diese Ringstruktur von 26 Buchstaben mithilfe der **Modulo**-Operation nachbilden:

Neuer Ascii-Code = (Alter Ascii-Code - 65 + 3) mod 26 + 65

Dabei bezeichnet **a mod b** den Rest der Division von a durch b; z. B. ist 27 mod 4 = 3, weil 27 : 4 gleich 6 Rest 3 ist. In JavaScript wird diese Modulo-Operation durch das Prozentzeichen dargestellt. Unsere Cäsar-Kodierung kann also auch kürzer durch die Zeile

```
code = (code - 65 + 3) % 26 + 65
```

realisiert werden.

Du hast sicherlich schon gemerkt, dass unser Kodierprogramm nur mit Großbuchstaben korrekt arbeitet. Außerdem wird aus dem Leerzeichen ein #. Durch einige wenige Ergänzungen können diese Probleme allerdings gelöst werden (vgl. Aufgabe 2).

Aufgaben

1. Schreibe eine Funktion, welche alle Buchstaben in Großbuchstaben umwandelt. Benutze dabei nicht die `toUpperCase`-Methode.

2. Ergänze die Funktion `kodieren()` aus `source\texte\kodieren.htm` folgendermaßen: Wandle die Buchstaben der Zeichenkette vor dem eigentlichen Kodiervorgang zunächst in Großbuchstaben um. Ändere dann das Programm so ab, dass das Leerzeichen nicht kodiert wird.

3. Schreibe eine Funktion `dekodieren()`, welche die Cäsar-Kodierung aus Aufgabe 2 rückgängig macht. Füge beide Funktionen in ein HTML-Dokument mit entsprechenden Textbereichen und Schaltflächen ein. Teste sie aus.

4. Was ändert sich an der Kodieren-Funktion, wenn die Buchstaben nicht um 3, sondern um 7 nach rechts verrückt werden? Warum ist die Benutzung der Modulo-Operation so vorteilhaft?

5. Facharbeit: Kodierungen, welche mithilfe einer einfachen Verschiebung realisiert werden, können rasch mithilfe einer Häufigkeitenanalyse geknackt werden. Schreibe ein Programm, welches eine Häufigkeitenanalyse für einen Text durchführen kann. Ermittle die relativen Häufigkeiten für sämtliche Buchstaben des Alphabetes für einen deutschen und für einen englischen Text. Vergleiche sie! Warum muss man für diese Untersuchungen längere Texte wählen?

6. Facharbeit: Schreibe ein Programm, welches einen Text mithilfe eines Schlüsselwortes kodiert. Dabei sollen die einzelnen Buchstaben des Schlüsselwortes angeben, um wie viele Stellen im Alphabet die einzelnen Buchstaben des zu kodierenden Textes verschoben werden sollen. Wenn das Schlüsselwort z. B. „ABER“ lautet, wird folgendermaßen kodiert:

Original	J	A	V	A	S	C	R	I	P	T		
Schlüsselwort	A	B	E	R	A	B	E	R	A	B	E	R
Verschiebung	1	2	5	18	1	2	5	18	1	2	5	18
kodierter Text	K	C	A	S	T	E	W	A	Q	V		

   Erläutere dabei auch den Vorteil gegenüber der Cäsar-Methode. Warum sollte das Schlüsselwort nicht zu kurz sein?