Diplomarbeit

Sprachen zur Unterstützung von Multimedia-Anwendungen

Thomas In der Rieden

Universität des Saarlandes
Fachrichtung 6.2 – Informatik
Lehrstuhl für Rechnerarchitektur und Parallelrechner
Prof. Dr. W. J. Paul

Juni 2002

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, im Juni 2002

Thomas In der Rieden

"Fortschritt ist der Weg vom Primitiven über das Komplizierte zum Einfachen." (Laotse)

Danksagung

Ich möchte den folgenden Personen danken, die mir das Erstellen dieser Arbeit ermöglicht haben:

- Professor Paul für die Vergabe dieses interessanten Themas, die ausgezeichnete Betreuung und das Vertrauen, das er in mich gesetzt hat,
- meiner ganzen Familie, insbesondere meinen Eltern, ohne deren bedingungslose Unterstützung ich nicht dort wäre, wo ich heute bin,
- Christoph Berg, Sven Beyer, Christian Jacobi, Michael Klein und Jochen Preiß für die vielen Korrekturen, fruchtbaren Gespräche und spannenden Tischfussballspiele, sowie
- dem gesamten Lehrstuhl Paul für das angenehme Arbeitsklima.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Projektbeschreibung 1

1.2 Gliederung 2

1.2.1 HTML 2

1.2.2 JavaScript 2

1.2.3 Java 3

2 JavaScript **5**

2.1 Einführung 5

2.2 Der Aufbau von JavaScript 5

2.2.1 Die Kernsprache 6

2.2.2 Clientseitiges JavaScript 6

2.3 Lexikalische Struktur 7

2.3.1 Groß- und Kleinschreibung 7

2.3.2 Leerzeichen und Zeilenenden 7

2.3.3 Semikolons 7

2.3.4 Kommentare 7

2.4 Primitive Datentypen 7

2.5 Variablen 8

2.6 Konstanten 9

2.7 Ausdrücke und Operatoren 9

2.7.1 Ausdrücke 9

2.7.2 Zuweisungsoperatoren 10

2.7.3 Vergleichsoperatoren 10

2.7.4 Arithmetische Operatoren 10

2.7.5 Bitweise Operatoren 10
2.7.6 Logische Operatoren 11
2.7.7 String-Operatoren 11
2.7.8 Spezielle Operatoren 12
2.8 Anweisungen 12
2.8.1 Zusammengesetzte Anweisungen 12
2.8.2 Bedingte Anweisungen 12
2.8.3 Schleifen-Anweisungen 13
2.8.4 Die `return`-Anweisung 14
2.8.5 Anweisungen zum Exception Handling 15
2.9 Funktionen 16
2.9.1 Definition von Funktionen 16
2.9.2 Aufruf von Funktionen 17
2.9.3 Vordefinierte Funktionen 17
2.10 Objekte 17
2.10.1 Das Prototypenprinzip 17
2.10.2 Zugriffe auf Objekteigenschaften 18
2.10.3 Das Erzeugen neuer Objekte 19
2.10.4 Der Prototypenmechanismus 19
2.11 Programme 21

3 HTML **23**
3.1 Einführung 23
3.2 Die Konstrukte von HTML 24
3.2.1 Elemente 24
3.2.2 Attribute 24
3.2.3 Sonderzeichen 25
3.2.4 Kommentare 25
3.3 Die Struktur eines HTML-Dokumentes 26
3.3.1 Der Prolog 26
3.3.2 Der Kopfteil 27
3.3.3 Der Rumpf 28
3.4 Text 29

3.4.1 Leerzeichen . . . 29
3.4.2 Strukturierter Text . . . 29
3.4.3 Zeilen und Absätze . . . 31
3.5 Listen und Aufzählungen . . . 33
3.5.1 Ungeordnete Listen, geordnete Listen und Listeneinträge . . . 33
3.5.2 Definitionslisten . . . 34
3.6 Tabellen . . . 34
3.6.1 Die Elemente einer Tabelle . . . 35
3.6.2 Tabellenformatierung . . . 38
3.6.3 Beispieltabelle . . . 40
3.7 Links . . . 41
3.7.1 Uniform Resource Identifiers - URI . . . 42
3.7.2 Das Element `A` . . . 42
3.7.3 Das Element `BASE` . . . 43
3.8 Objekte, Bilder und Applets . . . 43
3.8.1 Das Element `IMG` . . . 44
3.8.2 Die Elemente `OBJECT` und `PARAM` . . . 44
3.8.3 Image Maps . . . 46
3.8.4 Die Visualisierung von Objekten . . . 47
3.9 Stylesheets . . . 48
3.9.1 Das Hinzufügen von styles . . . 49
3.9.2 Externe Style Sheets . . . 50
3.10 Ausrichtung und Schriftarten . . . 51
3.10.1 Formatierung . . . 52
3.10.2 Schriftarten . . . 53
3.11 Frames . . . 55
3.11.1 Das Element `FRAMESET` . . . 55
3.11.2 Das Element `FRAME` . . . 56
3.11.3 Spezifikation von Zielframes . . . 57
3.11.4 Das Element `NOFRAME` . . . 57
3.11.5 Das Element `IFRAME` . . . 58
3.12 Formulare . . . 58

3.12.1 Kontrollelemente . . . 58
3.12.2 Das Element `FORM` . . . 59
3.12.3 Das Element `INPUT` . . . 60
3.12.4 Das Element `BUTTON` . . . 61
3.12.5 Menüs . . . 62
3.12.6 Das Element `TEXTAREA` . . . 65
3.12.7 Beschriftungen . . . 66
3.12.8 Strukturierung von Formularen . . . 66
3.12.9 Übertragung von Formularen . . . 67
3.13 Skripte . . . 68
3.13.1 Das Element `SCRIPT` . . . 68
3.13.2 Skriptsprache . . . 69
3.13.3 Das Ausführen von Skripten . . . 69

4 Java **71**
4.1 Einführung . . . 71
4.2 Lexikalische Struktur . . . 72
4.2.1 Trennzeichen . . . 72
4.2.2 Leerzeichen . . . 72
4.2.3 Kommentare . . . 72
4.2.4 Bezeichner . . . 72
4.2.5 Schlüsselwörter . . . 73
4.2.6 Literale . . . 73
4.2.7 Trennzeichen . . . 74
4.2.8 Operatoren . . . 74
4.3 Typen und Werte . . . 75
4.3.1 Primitive Typen und deren Werte . . . 75
4.3.2 Referenztypen und deren Werte . . . 75
4.4 Namen . . . 76
4.4.1 Einfache und qualifizierte Namen . . . 76
4.4.2 Gültigkeitsbereich . . . 76
4.5 Packages . . . 76
4.5.1 Compiliereinheiten . . . 77

4.5.2 Sub- und Superpackages 78
4.5.3 Elemente eines Packages 79
4.5.4 Unbenannte Packages 79
4.6 Klassen und Interfaces 79
4.6.1 Klassendeklaration 79
4.6.2 Interfacedeklaration 81
4.6.3 Variablendeklaration in Klassen 82
4.6.4 Konstantendeklaration in einem Interface 82
4.6.5 Methodendeklaration in Klassen 83
4.6.6 Methodendeklaration in Interfaces 84
4.6.7 Konstruktordeklarationen 84
4.6.8 Klassenarten und Namen 85
4.6.9 Kapselung 86
4.6.10 Vererbung und Polymorphie 87
4.6.11 Klassenelemente 89
4.6.12 Interfaceelemente 89
4.7 Arrays 89
4.7.1 Arraytypen 89
4.7.2 Deklaration eines Arrays 89
4.8 Ausdrücke 90
4.9 Anweisungen und Blöcke 90
4.9.1 Blöcke 90
4.9.2 Leere Anweisung 90
4.9.3 Ausdrucksanweisungen 90
4.9.4 Deklarationsanweisungen 92
4.9.5 Kontrollflussanweisungen 92

A JavaScript **95**

A.1 JavaScript-Operatoren 95
A.2 Vordefinierte Funktionen im JavaScript-Kern 96
A.3 Vordefinierte Objekte im JavaScript-Kern 97
A.3.1 Object 97
A.3.2 Array 97

A.3.3 Date 98
A.3.4 Math 98

B HTML **99**

B.1 Gemeinsame Attribute 99
B.2 Vordefinierte Farben in HTML 101

C Java **103**

Abbildungsverzeichnis

2.1 Die Struktur von Javascript 6

3.1 Cellspacing und -padding 40
3.2 Beispieltabelle im Internet Explorer 41
3.3 Geschachteltes dreiteiliges Framelayout 55
3.4 Formular im Internet Explorer 62
3.5 Drop-down Menü im Internet Explorer 65

4.1 Java Virtual Machine 71
4.2 Beispiel für die Struktur eines Javaprogramms 77

Tabellenverzeichnis

4.1 Java Schlüsselwörter 73
4.2 Erreichbarkeit in Klassen 87

A.1 JavaScript-Operatoren 95
A.2 Vordefinierte Funktionen im JavaScript-Kern 96
A.3 Methoden in `Math` 98

B.1 HTML-Kernattribute 99
B.2 HTML-Attribute zur Internationalisierung 99
B.3 Scripting Events in HTML 100
B.4 Vordefinierte Farben in HTML und Ihre RGB-Werte 101

C.1 Java-Operatoren 104

Kapitel 1

Einführung

1.1 Projektbeschreibung

Die vorliegende Arbeit ist aus einer Kooperation zwischen dem Ministerium für Bildung, Kultur und Wissenschaft des Saarlandes und den Fachrichtungen Informatik/Lehrstuhl Prof. Wolfgang Paul und Bildende Kunst & Kunsterziehung an der Universität des Saarlandes hervorgegangen.

Im Laufe dieses Projekts ist eine CD-ROM "Komposition als Mittel der Bildsprache" [HOIdR02] entwickelt worden. Sie ist für den Unterricht im Fach Bildende Kunst ab Klassenstufe 10 gedacht und soll dazu beitragen, dass sich die Schülerinnen und Schüler selbstständig und sachgerecht mit ästhetischen Objekten auseinander setzen können. In erster Linie sollen die dabei erworbenen Kenntnisse und Fähigkeiten bei der Analyse und Interpretation von ästhetischen Objekten angewendet werden, sie können aber auch bei der praktischen Arbeit hilfreich sein.

Im Mittelpunkt steht die Fragestellung, welche Rolle die Komposition für die Wirkung und Aussage eines ästhetischen Objektes spielen kann. Die ausgewählten Beispiele beziehen sich schwerpunktmäßig auf den Bereich Malerei; darüber hinaus werden Beispiele aus den Bereichen Grafik, Plastik/Objektkunst, Architektur und Werbung angesprochen.

Als interaktives Lernprogramm bietet die CD-ROM den Schülerinnen und Schülern nicht nur anschauliches Bildmaterial und die Möglichkeit zum eigenständigen Erarbeiten der Lerninhalte, sondern auch Aufgaben, durch die das Gelernte angewendet und der Lernerfolg überprüft werden kann.

Für den Unterricht gibt es vielfältige Einsatzmöglichkeiten. So kann die CD-ROM als Ganzes eine Unterrichtsreihe zum Thema "Komposition" begleiten oder - z.B. bei einem Unterricht mit kunstgeschichtlichem Schwerpunkt - abschnittsweise eingesetzt werden. Außerdem sind unterschiedliche Unterrichtsmethoden möglich. Die Lehrerin kann von einzelnen kompositorischen Mitteln ausgehen, deren Anwendung dann am Beispiel eines ästhetischen Objekts geklärt wird, sie kann aber auch den komplexen Einstieg wählen und vom einzelnen Objekt ausgehen und dann die relevanten kompositorischen Mittel untersuchen lassen.

Aufbau der CD-ROM

Die CD-ROM ist in zwei Inhaltsebenen gegliedert:

- Die **Erarbeitungsebene** gliedert sich in acht Lerneinheiten, in denen jeweils ein kompositorisches Mittel behandelt wird. An einigen Stellen hat die Schülerin bereits hier die Möglichkeit, gerade Erlerntes in einer Aufgabe zur Anwendung zu bringen.
- Die **Anwendungsebene** besteht aus Anwendungsaufgaben, mittels derer die Schülerin die Möglichkeit hat, das erlernte Wissen selbstständig zu überprüfen.

1.2 Gliederung

Die vorliegende Arbeit gliedert sich in drei Teile. In jedem Teil wird eine der im Multimedia-Projeklt verwendeten Programmiersprachen beschrieben.

1.2.1 HTML

Die grundlegende Struktur der CD-ROM basiert auf HTML-Seiten. Dazu wurde ein Frameset (siehe Kapitel 3.11) erzeugt, dass aus drei (in den Lerneinheiten) bzw. vier Teilfenstern (in den Anwendungsaufgaben) besteht. Die Texte in den Lerneinheiten bzw. in den Aufgaben sind einfacher HTML-Text (siehe Kapitel 3.4). Die Navigation auf der CD-ROM wird durch Links (siehe Kapitel 3.7) realisiert. Wir verwenden HTML weiterhin, um

- Bilder und Grafiken einzubetten (siehe Kapitel 3.8.1),
- JavaScripts einzubetten (siehe Kapitel 3.13),
- Java Applets einzubinden (siehe Kapitel 3.8) und
- Macromedia Flashanimationen [Mac] (siehe Kapitel 3.8) einzubinden.

Die Positionierung der einzelnen Elemente innerhalb des Inhaltsframes wird durch Tabellen realisiert (siehe Kapitel 3.6).

1.2.2 JavaScript

Wir haben JavaScript im Projekt dort zum Einsatz gebracht, wo eine einfache Interaktion des Benutzers mit der Software erforderlich war. Beispiele dafür sind der Austausch eines Bildes durch ein anderes oder die Steuerung einer Macromedia Flashanimation.

Um das Ausführen der Skripte zu steuern, haben wir uns der vordefinierten *scripting events* aus HTML bedient (siehe Anhang B.3). Der in den Internet Explorer eingebaute JavaScript Interpreter ist um client-seitiges JavaScript (siehe Kapitel 2.2) erweitert, was uns das Referenzieren einzelner Elemente des HTML-Dokumentes ermöglicht.

Da sich der Internet Explorer von Microsoft und der Netscape Navigator leider hinsichtlich des *document object models* (DOM) mehr und mehr unterscheiden und eine beidseitige Kompatibilität zu aufwändig gewesen wäre, mussten wir uns für einen Browser

entscheiden. Unsere Wahl viel dabei auf den Microsoft Internet Explorer, da er mit den Betriebssytemen von Microsoft vertrieben wird und man davon ausgehen kann, dass er auf den meisten Schulcomputern bereits installiert ist.

1.2.3 Java

Anwendungsaufgaben auf der CD-ROM sind prinzipiell in Java geschrieben. Wir haben dabei Java *Applets* erzeugt, die als eigene Dateien auf der CD-ROM vorliegen. Die Applets werden durch entsprechende HTML-Elemente (siehe Kapitel 3.8) innerhalb des Dokumentes platziert und ausgerichtet. Beispiele für Anwendungsaufgaben sind:

- das Auffinden einer bestimmten Stelle innerhalb eines Bildes,
- multiple-choice Aufgaben und
- Zuordnungsaufgaben.

Die große Flexibilität von Java hat es uns ermöglicht, einfach zu erlernende Techniken wie das Ankreuzen von Antworten, das Platzieren von vorgegebenen Antworten (*drag-and-drop*) oder das Bewegen von Elementen innerhalb eines Bildes zu realisieren. Alle Aufgaben lassen sich ohne die Tastatur lösen, es ist lediglich eine Maus notwendig. Die Schülerin kann ohne großen Einarbeitungsaufwand mit der Software auf der CD-ROM arbeiten.

Hinsichtlich der Programmierung haben wir uns das *abstract windowing toolkit* (AWT) von Java zunutze gemacht. Das AWT ist ein Package (siehe Kapitel 4.5), welches Klassen, Interfaces und Methoden (siehe Kapitel 4.6 und 4.6.5) zur Programmierung von grafischen Benutzerschnittstellen zur Verfügung stellt.

Die von uns programmierten Java Applets erfordern eine Java Virtual Machine der Version 1.2 oder höher.

Kapitel 2

JavaScript

2.1 Einführung

JavaScript ist eine objekt-orientierte, plattformunabhängige Skriptsprache, die von Netscape Communications 1996 als *„ein offener, freizügig lizenzierter Vorschlag für einen Standard, der der ganzen Netzgemeinschaft zur Verfügung steht."* veröffentlicht wurde [Cea95]. Netscape beabsichtigte, dem bis dahin statischen World Wide Web Leben einzuhauchen, indem man Webbrowser- und server befähigte, dynamische Webinhalte zu generieren, auf Benutzereingaben zu reagieren oder Datenbanken und Dateien zu lesen bzw. zu schreiben. Es gibt zwei grundlegende Einsatzarten von JavaScript-Programmen (Skripten): werden sie in den HTML-Quelltext einer Webseite eingebettet mit dieser heruntergeladen und dann von dem in den Browser integrierten JavaScript-Interpreter ausgeführt, so spricht man von *clientseitigem* JavaScript. Werden JavaScript- Programme auf dem Server ausgeführt, auf dem die Webseite bzw. das Skript liegt, dann spricht man von *serverseitigem* JavaScript.

2.2 Der Aufbau von JavaScript

JavaScript ermöglicht das Erstellen von Anwendungen, die über das Internet laufen - clientseitige in Webbbrowsern, serverseitige Anwendungen auf Webservern. Für die verschiedenen Einsatzbereiche kann die Kernsprache von JavaScript daher durch zusätzliche Objekte und Methoden erweitert werden:

- *clientseitiges JavaScript* erweitert die Kernsprache um Objekte, die das Steuern des Browsers und dessen Document Object Model (DOM) ermöglichen. So kann eine Anwendung Eintragungen in ein HTML-Formular vornehmen oder auf Benutzereingaben wie Formulareingaben oder Mausklicks reagieren.

- *serverseitiges JavaScript* erweitert die Kernsprache um Objekte, die für den Einsatz auf einem Server relevant sind. Eine Anwendung kann mit Hilfe dieser Objekte beispielsweise mit einer Datenbank kommunizieren oder Dateien auf dem Server manipulieren

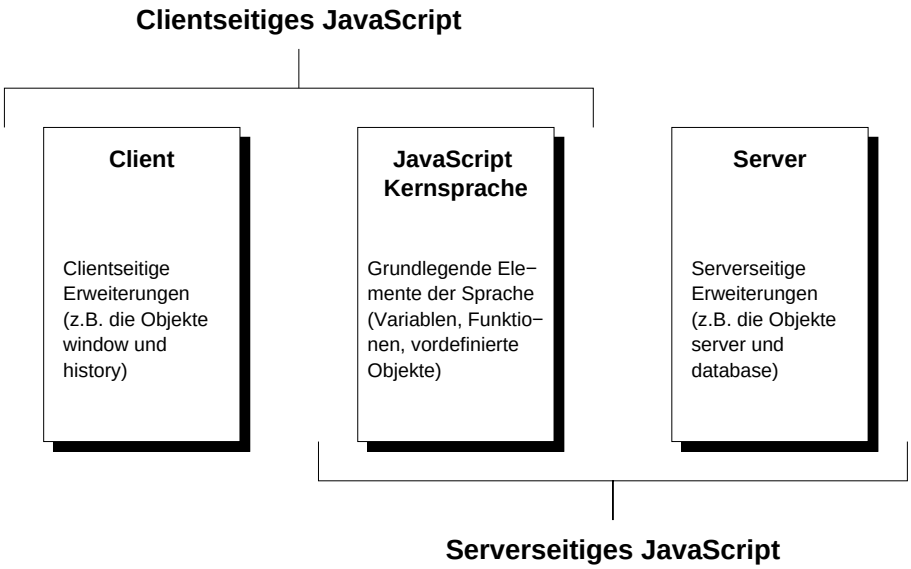

Abbildung 2.1: Die Struktur von Javascript

Zwar besteht auch die Möglichkeit, JavaScript in andere Kontexte als die Client-Server-Struktur des Webs einzubinden, jedoch sind solche Anwendungen Ausnahmeerscheinungen und werden daher an dieser Stelle nicht weiter behandelt. Programmierer, die beabsichtigen, JavaScript-Interpreter in Ihre Software zu integrieren, finden einen Referenz-Interpreter der Kernsprache mit dem dazugehörigen C-Quelltext bei Netscape in der JSRef-Distribution [Net].

2.2.1 Die Kernsprache

Die Kernsprache, auch *core* genannt, bildet das Fundament der JavaScript-Struktur. Sie ist in allen Interpretern vorhanden und definiert die grundlegenden Eigenschaften der Sprache:

- Schlüsselworte
- Grammatik und Syntax von Anweisungen
- Regeln für Ausdrücke und Variablen
- Vordefinierte Objekte und Methoden wie `Array`, `Date` und `Math`

2.2.2 Clientseitiges JavaScript

Clientseitiges JavaScript ist üblicherweise in eine HTML-Datei eingebettet. Fordert ein Browser diese Datei über das Netz an, so wird diese vollständig, das heißt mit HTML- und JavaScript- Anweisungen, vom Server übertragen. Anschließend werden die HTML-Inhalte dargestellt und die JavaScript-Anweisungen ausgeführt. Die Steuerung dieses Interpretationsprozesses wird dabei von sogenannten *events* bestimmt. Events sind Ereignisse, die

während des Ladens und Betrachtens der Webseite durch den Benutzer ausgelöst werden wie Formulareinträge oder das Bewegen der Maus (vgl. Kapitel 3.13 und Anhang B.3).

Wir werden uns im folgenden lediglich mit dem JavaScript-Kern beschäftigen, da er die grundlegenden Elemente der Sprache JavaScript definiert.

2.3 Lexikalische Struktur

2.3.1 Groß- und Kleinschreibung

JavaScript unterscheidet hinsichtlich der Groß- und Kleinschreibung. Das gilt für die Schlüsselwörter der Sprache sowie für Variablen, Funktionsnamen und alle anderen Bezeichner. Dementsprechend sind `online`, `Online`, `OnLine` und `ONLINE` die Namen von vier verschiedenen Variablen und lediglich "while" ist die richtige Schreibweise des Schlüsselwortes `while`.

2.3.2 Leerzeichen und Zeilenenden

JavaScript ignoriert Tabulatoren, Leerzeichen und Zeilenumbrüche zwischen Tokens in Programmen, wenn sie nicht in Stringkonstanten vorkommen. Das erlaubt dem Programmierer, seine Quelltexte nahezu beliebig zu formatieren und trägt somit zur guten Lesbarkeit und Verständlichkeit bei.

2.3.3 Semikolons

In JavaScript werden einzelne Anweisungen mit einem Semikolon getrennt. Wenn jede Anweisung in einer eigenen Zeile steht, ist es auch erlaubt, das Semikolon wegzulassen. Für eine bessere Strukturierung ist es jedoch empfehlenswert, die Semikolons zu benutzen.

2.3.4 Kommentare

JavaScript erlaubt Kommentare nach C und C++ Art. Zeilen, die mit `//` beginnen, werden ignoriert, ebenso Text (der auch mehrzeilig sein darf), der zwischen den Zeichen `/*` und `*/` steht. Eine Besonderheit stellen HTML-Kommentare dar: JavaScript erkennt den Anfang eines HTML-Kommentars `<!--` und behandelt ihn als einen einzeiligen Kommentar. Hingegen wird die Endsequenz `-->` nicht erkannt. Ein JavaScript-Programm, dessen erste Zeile mit `<!--` beginnt und dessen letzte Zeile mit `//-->` endet, ist daher komplett in einem HTML-Kommentar enthalten. So kann man Programme vor älteren WWW-Browsern ohne JavaScript-Unterstützung verbergen, ohne sie den Browsern vorzuenthalten, die sie verarbeiten können.

2.4 Primitive Datentypen

JavaScript kennt folgende Arten von primitiven Datentypen:

- **Number**: der Typ Number ist eine Menge von Werten, die Zahlen repräsentieren. Intern werden alle diese Werte (auch die ganzzahligen) in 64-bit Fließkommazahlen umgewandelt, die dem Format des IEEE 754 Standard entsprechen. Zusätzlich gibt es noch die speziellen Werte "not a number" (`NaN`), positiv unendlich (`POSITIVE_INFINITY`) und negativ unendlich (`NEGATIVE_INFINITY`). Beispiele für Werte des Typs Number sind `3.14159`, `42`, `5.12e-17` und `0xdeadbeef`
- **Boolean**: der Typ Boolean besteht aus genau zwei Werten: `true` und `false`.
- **Strings**: die Menge aller endlichen Folgen von 0 oder mehr ASCII-Zeichen in Anführungszeichen oder Hochkommata; Beispiele für Strings sind "Hallo Welt" bzw. 'Hallo Welt'.
- **Null**: der Datentyp Null hat genau einen Wert, `null`, der die 0, den leeren String oder das Fehlen einer Referenz darstellt.
- **Undefined**: der Datentyp Undefined hat ebenfalls nur einen Wert, `undefined`, der verwendet wird, wenn einer Variable noch kein Wert zugewiesen worden ist.

Intern wird keine Unterscheidung zwischen Fließkommazahlen und Integern gemacht; beide werden als 64-bit doppeltgenaue IEEE 754-Zahlen (vergleiche dazu [IEE85]) im Speicher abgelegt.

JavaScript ist eine *dynamisch getypte* Sprache. Das heißt, der Datentyp einer Variable muss bei ihrer Deklaration nicht festgelegt werden und während der Skriptausführung werden Datentypen bei Bedarf automatisch konvertiert. Das erlaubt es, eine Variable als

```
var Antwort = 42;
```

zu definieren und ihr zu einem späteren Zeitpunkt einen Stringwert zuzuweisen, beispielsweise

```
Antwort = 'Einen großen Milchkaffee, bitte!'
```

In Anweisungen mit dem Operator +, in denen Zahlen und Strings vorkommen, konvertiert JavaScript die Zahlenwerte in Strings:

```
x = 'Die Antwort ist ' + 42 // gibt 'Die Antwort ist 42' zurück
```

Dies gilt jedoch nicht für Anweisungen mit anderen Operatoren:

```
'37'-7 // gibt 30 zurück
```

2.5 Variablen

Der Name einer Variablen, der *Variablenbezeichner*, muss mit einem Buchstaben, einem Unterstrich (_) oder einem Dollarzeichen ($) beginnen. Die darauffolgenden Zeichen dürfen

auch Zahlen sein. Man beachte, dass JavScript zwischen Groß- und Kleinschreibung unterscheidet. Ab JavaScript 1.5 dürfen auch ISO8859-1 oder Unicode-Zeichen [Con00] wie å und ö in Bezeichnern vorkommen.

Definiert man eine Variable außerhalb einer Funktion, so handelt es sich um eine *globale Variable*. Dabei kann, wie bereits oben erwähnt, das Schlüsselwort `var` weggelassen werden. Wird die Variable innerhalb einer Funktion definiert, so ist es eine *lokale Variable*, die nur innerhalb der Funktion Gültigkeit hat. Lokale Variablen erfordern die Verwendung des Schlüsselwortes `var`.

Variablen können auf zwei Arten deklariert werden:

- Durch einfaches Zuweisen eines Wertes, beispielsweise `x = 42`
- Durch das Schlüsselwort `var`, beispielsweise `var x`

Eine Variable oder ein Arrayelement, denen noch kein Wert zugewiesen worden ist, haben den Wert `undefined`. Wenn man eine solche Variable auswertet, so hängt der Rückgabewert von der Deklaration ab:

- Wurde die Variable ohne `var` deklariert[1] , dann endet die Auswertung in einem Laufzeitfehler.
- Wurde die Variable mit `var` deklariert, so liefert die Auswertung `undefined` oder in numerischen Kontexten `NaN` (not a number).

Der Wert `undefined` verhält sich wie `false`, wenn er als boolescher Wert verwendet wird. Gleiches gilt für den Wert `null`, der sich zusätzlich noch wie 0 in numerischen Kontexten verhält.

2.6 Konstanten

Durch Verwendung des Schlüsselwortes `const` werden benannte Konstanten erzeugt. Für Konstantenbezeichner gelten dieselben Regeln wie für Variablenbezeichner. Während der Ausführung eines Skriptes können die Werte von Konstanten nicht verändert werden und sie können nicht erneut deklariert werden. Für die Gültigkeitsregeln von Konstanten gelten dieselben Bedingungen wie für Variablen; allerdings muss das `const` Schlüsselwort auch für globale Konstanten verwendet werden.

2.7 Ausdrücke und Operatoren

2.7.1 Ausdrücke

Ausdrücke bestehen aus Variablen, Literalen und Operatoren; sie können rekursiv aufgebaut sein. Literale sind entweder Stringkonstanten, numerische Konstanten, oder einer

[1] Anmerkung: das ist natürlich gleichbedeutend einer Nicht-Deklaration der Variablen, da sie nur durch `var` oder Zuweisung deklariert werden kann.

der primitiven Werte `true`, `false` oder `undefined`. Ausdrücke werden generell zu einem festen Wert ausgewertet.

2.7.2 Zuweisungsoperatoren

Zuweisungsoperatoren weisen dem linken Operanden den Wert des rechten Operanden zu. Der Zuweisungsoperator ist das Gleichheitszeichen (=). Wie in Java/C/C++ dürfen für Standardoperationen auch Abkürzungen verwendet werden, beispielsweise:

```
x += y // entspricht x = x+y
```

```
x *= y // entspricht x = x*y
```

Jeder Zuweisungsausdruck besitzt einen Wert, der gleich dem Wert des linken Operanden nach der Zuweisung ist.

2.7.3 Vergleichsoperatoren

Vergleichsoperatoren vergleichen ihre Operanden und liefern einen booleschen Wert, der vom Vergleichsergebnis abhängt. Strings werden dabei anhand der lexikographischen Ordnung miteinander verglichen. Die Vergleichsoperatoren entsprechen den Vergleichsoperatoren von Java, C und C++. Zusätzlich hat JavaScript die Operatoren `===`, der strikte Gleichheit prüft (Typ und Wert gleich), und `!==`, der strikte Ungleichheit (Wert ungleich und/oder Typ ungleich) prüft.

2.7.4 Arithmetische Operatoren

Arithmetische Operatoren haben numerische Werte (Literale oder Variablen) als Operanden und geben einen numerischen Wert zurück. Die arithmetischen Standardoperatoren `+`, `-`, `*`, `/` in JavaScript funktionieren so, wie man es von den meisten anderen Programmiersprachen gewohnt ist. Allerdings liefert der `/`-Operator auf Integern eine Fließkommazahl zurück und nicht - wie man es aus C und Java gewohnt ist - eine ganzzahlige Division.

```
1/2 // liefert 0.5 in JavaScript
```

```
1/2 // liefert 0 in Java
```

2.7.5 Bitweise Operatoren

Bitweise Operatoren sind auf numerischen Werten definiert. Sie betrachten ihre Operanden als 32-stellige Folgen von Nullen und Einsen. Sie geben jedoch einen ganzzahligen Wert zurück. Die bitweisen logischen Operationen laufen dabei wie folgt ab:

- Die Operanden werden in 32-bit Integer umgewandelt; ist dies nicht möglich, so wird `NaN` zurückgegeben.

- Das i-te Bit des ersten Operanden wird mit dem i-ten Bit des zweiten Operanden verknüpft.
- Das Ergebnis der Operation wird in eine ganze Zahl umgewandelt und zurückgegeben.

Die Shift-Operatoren behandeln - wie gewohnt - den ersten Operanden als den zu shiftenden Wert und nehmen den zweiten Operanden als die Shift-Distanz. Die Syntax der bitweisen Operatoren ist analog zu C/C++ und Java.

2.7.6 Logische Operatoren

Logische Operatoren werden normalerweise auf booleschen Werten angewendet und liefern in diesen Fällen auch solche zurück. Eine Ausnahme stellen die Operatoren `&&` und `||` dar, die auch auf anderen Datentypen arbeiten können:

- `expr1 && expr2` (das logische UND) gibt `expr1` zurück, wenn dieser zu `false` ausgewertet wird, andernfalls `expr2`.
- `expr1 || expr2` (das logische ODER) gibt `expr1` zurück, wenn dieser zu `true` ausgewertet wird, andernfalls `expr2`.

Dabei gelten die üblichen Auswertungsbedingungen:

- Alle Zahlen außer der 0 werden zu true ausgewertet.
- Alle Strings außer dem leeren werden zu true ausgewertet.
- `null` und `undefined` werden zu false ausgewertet.

2.7.7 String-Operatoren

Zusätzlich zu den auf Strings anwendbaren Vergleichsoperatoren gibt es den Konkatenationsoperator (+), der zwei Strings zu einem neuen verbindet. Auch die abkürzende Schreibweise `+=` darf angewendet werden. Dabei ist zu beachten, dass der Konkatenationsoperator mitunter nicht assoziativ ist, wenn er gemischt mit Strings und Zahlen gebraucht wird. Wenn einer der Operanden ein String ist, so versucht er auch den anderen Operanden in einen String umzuwandeln. Andererseits führt ein Vergleichsoperator nur dann einen Stringvergleich durch, wenn beide Operanden Strings sind.

```
s = 1+2+" Chinesen mit dem Kontrabass";

// ...ergibt "3 Chinesen mit dem Kontrabass"

t = Anzahl Chinesen mit Kontrabass: "+1+2;

// ...ergibt Anzahl Chinesen mit Kontrabass: 12"
```

2.7.8 Spezielle Operatoren

- Der **Konditionaloperator** ist der einzige JavaScript Operator mit drei Operanden. Je nach dem Ergebnis einer Bedingung kann der Operator zwei Werte liefern:

  `Bedingung ? Wert1 : Wert2`

  Fall `Bedingung` wahr ist, so gibt der Operator `Wert1` zurück, ansonsten `Wert2`.

- Der **`new`-Operator** erzeugt eine Instanz eines benutzerdefinierten Objekttyps oder der vordefinierten Objekttypen `Array`, `Boolean`, `Date`, `Function`, `Image`, `Number`, `Object`, `Option`, `RegExp` oder `String`. Die Syntax dieses Operators ist dieselbe wie in C++/Java:

  $$\textit{Objektname}\ \texttt{=}\ \texttt{new}\ \textit{Objekttyp}\,(\textit{Parameter}_1[,\textit{Parameter}_2,\ldots,\textit{Parameter}_n])\texttt{;}$$

- Der **`this`-Operator** gibt eine Referenz auf das Objekt zurück, in dessen Gültigkeitsbereich es aufgerufen wurde. Mehr zu diesem Operator und Objekten findet sich im Kapitel 2.10.

- Der Operator `typeof` liefert einen String zurück, der den Typ des darauf folgenden, nicht ausgewerteten Ausdrucks angibt. Die Syntax für diesen Operator ist

  `typeof` (*Ausdruck*)

2.8 Anweisungen

JavaScript-Programme sind Folgen von Anweisungen. Jeder Ausdruck (siehe Kapitel 2.7) ist gleichzeitig eine Anweisung. Zusätzlich gibt es noch die in den folgenden Abschnitten definierten Anweisungstypen. Mehrere Anweisungen in einer Zeile getrennt durch Semikolons sind zulässig, üblicherweise beginnt man aber für jede Anweisung eine neue Zeile.
Um optionale Code-Sequenzen in Anweisungen hervorzuheben habe ich diese im folgenden in eckige Klammern gesetzt; Wörter in `Schreibmaschinen-Schrift` sind Java-Script Schlüsselwörter, Wörter in *kursiver Schrift* sind Platzhalter für Ausdrücke oder Anweisungsfolgen.

2.8.1 Zusammengesetzte Anweisungen

Eine Folge von Anweisungen, die in geschweiften Klammern steht, nennt man **Anweisungsblock**. Blöcke definieren im Gegensatz zu C/C++/Java keine Gültigkeitsbereiche für Variablen. Der Anweisungsblock endet hinter der geschweiften Klammer ohne Semikolon.

2.8.2 Bedingte Anweisungen

Die `if`-Anweisung erlaubt es, das Ausführen einer oder mehrerer Anweisungen von der Auswertung eines Ausdrucks abhängig zu machen:

`if` (*Ausdruck*) {*Anweisungsfolge*$_1$} [`else` {*Anweisungsfolge*$_2$}]

Dazu wird zuerst der Ausdruck nach dem Schlüsselwort `if` ausgewertet. Ist er `true`, so wird die *Anweisungsfolge*$_1$ ausgeführt, andernfalls nicht. Optional kann man noch die `else`-Klausel mit angeben, deren *Anweisungsfolge*$_2$ im `false`-Fall ausgeführt wird. Für die booleschen Ergebnisse bei der Ausdrucksauswertung vergleiche den vorhergehenden Abschnitt über Ausdrücke.

Die `switch`-Anweisung macht die Ausführung eines Anweisungsblocks ebenfalls vom Wert eines Ausdrucks abhängig:

`switch` (*Ausdruck*) {
  `case` *label*:
    Anweisungsfolge
  `case` *label*:
    Anweisungsfolge
    ⋮
  [`case default`:
    Anweisungsfolge]
}

Bei der Ausführung einer `switch`-Anweisung wird zuerst der Ausdruck hinter dem Schlüsselwort `switch` ausgewertet. Anschließend wird dieser Wert mit den Labeln in top-down Reihenfolge auf strikte Gleichheit (Typ und Wert, `===`) überprüft. Ist ein passendes Label gefunden, so führt der Interpreter die dazugehörige Anweisungsfolge hinter dem Doppelpunkt aus, bis er auf die optionale Anweisung `break` stößt oder bis die `switch`-Anweisung mit der geschweiften Klammer endet. Wird kein passendes Label gefunden, so sucht der Interpreter nach dem optionalen `default` Label und führt die dazugehörige Anweisungsfolge aus; existiert auch dieses Label nicht, so wird keine Anweisung ausgeführt und die `switch`-Anweisung ist beendet.

2.8.3 Schleifen-Anweisungen

Eine Schleife ist eine Anweisungsfolge, die solange wiederholt wird, bis eine bestimmte Bedingung erfüllt ist. JavaScript unterstützt die `for`, `while` und `do...while` Schleifen-Anweisungen. Ferner ist die Verwendung der Anweisungen `break`, `continue` und `label` erlaubt, um erweiterte Funktionalität innerhalb von Schleifen zu ermöglichen.

Die `for`-Schleife führt eine Folge von Anweisungen solange aus, bis die Schleifenbedingung zu `false` ausgewertet wird. Die `for`-Schleife in JavaScript gleicht stark der `for`-Schleife in Java oder C:

`for` (*Initialisierung; Bedingung; Aktualisierung*)
{
  Anweisungen
}

Bei der Ausführung einer `for`-Schleife passiert folgendes:

1. Der Initialisierungsausdruck wird ausgewertet; üblicherweise werden dabei ein oder

mehrere Schleifenzähler initialisiert, die Syntax erlaubt aber auch Ausdrücke beliebiger Komplexität. In diesem Ausdruck dürfen auch Variablen deklariert werden.

2. Die Schleifenbedingung wird ausgewertet; ist der Wert der Bedingung `true`, so wird die Schleife ausgeführt, ansonsten terminiert die `for`-Schleife.

3. Die Anweisungsfolge im Innern der Schleife wird ausgeführt.

4. Der Aktualisierungsausdruck wird ausgewertet, anschließend wird mit Schritt 2 fortgefahren.

Die `while`-Schleife führt seine Anweisungen aus, solange die Schleifenbedingung `true` ergibt. Die Syntax sieht wie folgt aus:

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

Die Auswertung der Schleifenbedingung erfolgt vor der Ausführung des Anweisungsblocks. Ist die Bedingung `false`, so wird die Schleife nicht ausgeführt. Es kann daher sein, dass der Anweisungsteil einer `while`-Schleife gar nicht ausgeführt wird. Eine Alternative stellt die `do...while`-Schleife dar, deren Anweisungsblock mindestens einmal durchlaufen wird:

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

Schleifen können vorzeitig durch die Verwendung der `break`-Anweisung verlassen werden; ebenso kann die `continue`-Anweisung dazu eingesetzt werden, die momentane Schleifeniteration abzubrechen und mit einer neuen zu beginnen. Bei einer `while`-Schleife wird hierbei als nächstes die Schleifenbedingung überprüft, bei einer `for`-Schleife wird zuvor noch der Aktualisierungsausdruck ausgewertet.

Die Anweisung `break` wird verwendet, um eine `switch`-Anweisung oder eine Schleifenanweisung zu terminieren. Bei geschachtelten Schleifenkonstruktionen wird dabei nur die innerste Schleife beendet.

Die Anweisung `continue` beendet nicht die ganze Schleife, sondern lediglich den aktuellen Durchlauf derselbigen. In einer `while`-Schleife wird dann mit der Überprüfung der Schleifenbedingung fortgefahren, in `for`-Schleifen mit der Auswertung des Aktualisierungsausdrucks.

2.8.4 Die `return`-Anweisung

Jeder Funktionsaufruf ist gleichzeitig ein Ausdruck und hat somit einen Wert. Die `return`-Anweisung wird dazu benutzt, den Wert festzulegen, den eine Funktion zurückgibt; sie hat folgende Syntax:

`return` [*Ausdruck*]

Bei der Ausführung der `return`-Anweisung wird zunächst der - optionale - Ausdruck berechnet und als Funktionswert zurückgegeben. Gibt es keinen Ausdruck hinter dem Schlüsselwort, so wird der Wert `undefined` zurückgegeben. `return`-Anweisungen dürfen nur in Funktionen vorkommen. Sie beenden die Ausführung einer Funktion, auch wenn noch weitere Anweisungen im Funktionsrumpf auf sie folgen.

2.8.5 Anweisungen zum Exception Handling

JavaScript stellt zwei Anweisungen zur Verfügung, um Exceptions auszulösen bzw. zu behandeln. Die `throw`-Anweisung bietet dem Programmierer die Möglichkeit, eine benutzerdefinierte Exception auszulösen:

`throw` *Ausdruck*

Der Ausdruck nach dem Schlüsselwort definiert den Wert der Exception; dieser kann eine Zahl, ein String, ein boolescher Wert oder sogar ein Objekt sein, auf dessen Eigenschaften bei der anschließenden Behandlung der Exception zugegriffen wird.

Das eigentlich Exception Handling erfolgt mittels der `try...catch`-Anweisung. Diese Anweisung besteht aus einem `try`-Teil, der eine oder mehrere Anweisungen beinhaltet, die ausgeführt werden sollen, und einem oder mehreren `catch`-Blöcken, die den weiteren Programmverlauf spezifizieren, falls eine Exception im `try`-Teil ausgelöst werden sollte:

```
try
{
   Anweisungen
}
catch (Bezeichner)
{
   Anweisungen
}
[
catch (Bezeichner)
{
   Anweisungen
}
⋮
]
```

Der *Bezeichner* repräsentiert hierbei den Wert der Exception, die ausgelöst worden ist (siehe `throw`-Anweisung und die zur Unterscheidung der einzelnen Exceptions verwendet werden kann.

Ferner bietet JavaScript die Möglichkeit, die `try...catch`-Anweisung um einen `finally`-Block zu erweitern, der - gleich, ob eine Exception ausgelöst worden ist oder nicht - am

Ende der Anweisung ausgeführt wird. Dies bietet sich beispielsweise im Umgang mit Dateien auf Festplatte an, die am Ende immer geschlossen werden sollten:

```
openMyFile();
try {
   writeMyFile(theData);
}
finally{
   closeMyFile(); // immer schön schließen
}
```

2.9 Funktionen

Funktionen sind ein elementarer und bedeutender Bestandteil von JavaScript. Eine Funktion ist eine JavaScript- Prozedur - eine Folge von Anweisungen, die eine spezifische Aufgabe erfüllen. Eine Funktion muss zuerst definiert werden, bevor sie verwendet werden kann.

2.9.1 Definition von Funktionen

Die Definition einer Funktion besteht aus dem Schlüsselwort `function` gefolgt von

- dem *Namen* der Funktion.
- einer *Liste von Argumenten* der Funktion, umschlossen von runden Klammern, untereinander getrennt durch Kommata.
- den *Anweisungen*, die den Rumpf der Funktion bilden, umschlossen von geschweiften Klammern. Die Anweisungen in einer Funktion können auch Aufrufe von anderen Funktionen, die im selben Skript deklariert sind, beinhalten.

Die Syntax für Funktionsdefinitionen sieht folgendermaßen aus:

`function` *Funktionsname* ([$Parameter_1$] [,$Parameter_2, \ldots ,Parameter_n$])
{
   Anweisungen
}

Parameter von Funktionen werden als Wert übergeben. Änderungen am Wert eines Parameters innerhalb einer Funktion haben keinen globalen Effekt. Ist der übergebene Parameter ein Objekt und ändert die Funktion Eigenschaften dieses Objektes, so sind diese Änderungen auch außerhalb der Funktion sichtbar. Funktionen können optional mit einer `return`-Anweisung beendet werden (vgl. Kapitel 2.8.4). Funktionen dürfen rekursiv sein, das heißt in einer Funktionsdeklaration darf die Funktion sich selbst aufrufen.

Die Anweisung `function` stellt eine Besonderheit dar, da sie ein Objekt des vordefinierten Typs `Function` erzeugt und dabei nicht die übliche Syntax bei der Objekterzeugung

verwendet. Näheres zum Thema Objekterzeugung findet sich im Kapitel 2.10.3.

2.9.2 Aufruf von Funktionen

Der Aufruf einer Funktion geschieht durch den Funktionsnamen gefolgt von den Argumenten (auch Parametern genannt) in runden Klammern ( ). Zwischen den Klammern kann eine durch Kommata getrennte Liste von Parameterwerten oder -ausdrücken stehen. Beim Aufruf der Funktion wird jeder der zwischen den Klammern stehenden Ausdrücke ausgewertet und jedes dieser Ergebnisse bildet ein Argument der Funktion. Die Zuweisung der Argumente an die in der Funktionsdefinition deklarierten Variablen erfolgt in der Reihenfolge ihrer Deklaration. Da JavaScript eine untypisierte Sprache ist wird nicht überprüft, ob der Datentyp übergeben wird, den die Funktion erwartet; hier muss gegebenenfalls der Operator `typeof` verwendet werden. Überzählige Parameter werden ignoriert, fehlende Parameter durch den Wert `undefined` belegt.

2.9.3 Vordefinierte Funktionen

JavaScript bietet einige nützliche, vordefinierte Funktionen an, die das Programmieren wesentlich erleichtern:

- `String`(*Objekt*) und `Number`(*Objekt*) konvertieren ein Objekt in einen String oder eine Zahl.
- `parseInt`(*String*) und `parseFloat`(*String*) wandeln Strings in Integerzahlen oder Fließkommazahlen um.

Eine vollständige Wiedergabe aller vordefinierten Funktionen findet sich im Anhang A.2.

2.10 Objekte

Ein *Objekt* ist ein Datentyp, der aus einer Menge von Eigenschaften oder einem einzelnen primitiven Wert besteht. Jede *Eigenschaft* ist ein Paar (*Name, Wert*). Dabei ist *Name* der Bezeichner für die Eigenschaft, über den sie referenziert wird. Der *Wert* einer Eigenschaft kann jeden beliebigen Datentyp annehmen, also auch ein Objekt selbst sein. Insbesondere kann der Wert auch eine Funktion sein. Eine solche Funktion heißt dann *Methode*.

2.10.1 Das Prototypenprinzip

Klassenbasierte objektorientierte Programmiersprachen wie Java und C++ unterscheiden zwischen Klassen und Instanzen. Klassen definieren alle Eigenschaften und Methoden, die eine gewisse Anzahl von Objekten charakterisieren, sind selbst jedoch nicht Mitglied dieser Menge von Objekten. Instanzen sind Objekte, die anhand des "Bauplans", den die Klasse vorgibt, konstruiert worden sind. Sie haben genau die Eigenschaften und Methoden ihrer Klasse.

JavaScript hingegen ist eine *prototypbasierte* Programmiersprache und macht diese Unterscheidung nicht: es gibt lediglich Objekte. Jedes Objekt hat ein *Prototypobjekt*, das als Vorlage dient und seine Eigenschaften und ihre Werte bei der Instantiierung des Objektes demselbigen zur Verfügung stellt. Jedes Objekt kann zusätzlich eigene Eigenschaften spezifizieren, entweder bei der Erzeugung oder zur Laufzeit. Außerdem kann jedes Objekt als Prototyp für ein anderes Objekt dienen, was zur sogenannten *Prototypkette* führt (analog zur *Klassenkette* in den klassenbasierten Sprachen). An der Spitze dieser Kette steht das Objekt `Object` (siehe Anhang A.3), von dem alle anderen Objekte abgeleitet sind. Somit verfügen alle Objekte zumindest über die Eigenschaften und Methoden, die in `Object` definiert sind.

In klassenbasierten Sprachen definiert man Klassen in eigenen Klassendefinitionen; in diesen Definitionen kann man dann spezielle Methoden, Konstruktoren genannt, definieren. *Konstruktoren* spezifizieren die anfänglichen Werte der Eigenschaften einer Objektinstanz bei der Erzeugung mit `new`. Die dem Objekt zugrundeliegende Klasse gibt dabei seinen *Typ* an, der gleich dem Klassennamen ist.

JavaScript geht ähnlich vor, trennt aber die Klassendefinition nicht von der Konstruktordefinition. Jede JavaScript-Funktion kann als Konstruktor verwendet werden. Der Aufruf einer *Konstruktorfunktion* mit dem Operator `new` erzeugt ein neues Objekt und definiert seine initialen Eigenschaften und ihre Werte. Hierbei spezifiziert die verwendete Konstruktorfunktion den *Typ* des Objekts. Wir werden auf die Objekterzeugung im Kapitel 2.10.3 noch näher eingehen.

2.10.2 Zugriffe auf Objekteigenschaften

Der Punktoperator (.) wird sowohl zum Lesen als auch zum Schreiben von Objekteigenschaften verwendet:

$$objektName.eigenschaftName$$

Dabei gibt der Wert links vom Punkt eine Referenz auf ein Objekt an, der Wert rechts vom Punkt gibt den Bezeichner der Eigenschaft an.

Neue Objekteigenschaften werden erzeugt, indem man einfach ihren Wert setzt:

```
document.laenge = 512;
```

Existiert die Eigenschaft `laenge` im Objekt `document` noch nicht, so wird sie angelegt, damit ihr ein Wert zugewiesen werden kann. Versucht man, nicht existierende Eigenschaften auszulesen, so erhält man den Wert `undefined` zurück.

Beinhaltet ein Objekt lediglich einen einzigen primitiven Wert, so verwendet man die Methode `valueOf` um diesen Wert auszulesen[2].

[2] Über die Methode `valueOf` verfügen alle Objekte, da sie in `Object` definiert ist. Mehr zum vordefinierten Objekt `Object` und seinen Methoden findet sich im Anhang A.3.

2.10.3 Das Erzeugen neuer Objekte

Wie wir bereits in Kapitel 2.7 gesehen haben, wird zum Erzeugen neuer Objekte der Operator `new` verwendet. Die Syntax dafür lautet folgendermaßen:

`V = new F(`[*Argumente*]`);`

Dabei ist `V` der Name einer Variablen, `F` der Name einer Funktion und [*Argumente*] steht für eine mögliche optionale Argumentliste.

Bei der Erzeugung eines Objektes passiert dabei folgendes:

1. Der Operator `new` erzeugt ein neues, generisches Objekt `X` vom Typ `Object`.
2. Der Funktion `F` wird zusätzlich zu ihren expliziten Argumenten implizit ein Verweis auf dieses Objekt `X` übergeben, auf das im Funktionsrumpf mit dem Schlüsselwort `this` zugegriffen werden kann.
3. Die Anweisungsfolge im Rumpf von `F` wird abgearbeitet. Dabei werden gegebenenfalls Eigenschaften im Objekt `X` erzeugt bzw. geändert.
4. Der Variablen `V` wird das so erzeugte Objekt `X` zugewiesen.

Dabei erzeugen bzw. verändern genau die Anweisungen eine Eigenschaft mit Namen `e` in `X`, die eine Zuweisung der Form `this.e =` *Ausdruck* sind. Dabei gilt:

- Existiert bereits eine Eigenschaft mit Namen `e` in `X`, so wird der Wert von `e` auf den Wert des Ausdrucks *Ausdruck* gesetzt.
- Existiert diese Eigenschaft noch nicht, so wird eine neue Eigenschaft mit Namen `e` in `X` erzeugt und auf den Wert des Ausdrucks *Ausdruck* gesetzt.

2.10.4 Der Prototypenmechanismus

Wie bereits in Kapitel 2.10.1 erwähnt, basiert Vererbung in JavaScript auf dem *Prototypenprinzip*. Dieser Mechanismus beruht auf mehreren Elementen, die wir im folgenden näher betrachten werden.

Jedes beliebige Objekt `X` hat zwei Eigenschaften:

1. `X.constructor` heißt *Konstruktorfunktion* von `X` oder auch *Typ* von `X`.
2. `X.prototype` heißt *Prototyp* von `X`.

Für das generische Objekt `Object` gilt dabei `Object.constructor = Object` und `Object.prototype = null`.

Wie wir in den Kapiteln 2.9 und 2.10.3 gesehen haben, gibt es zwei Möglichkeiten, Objekte zu erzeugen:

1. Durch die Deklaration einer Funktion `F` wird ein Objekt `F` vom Typ `Function` erzeugt.
2. Mit Hilfe von `new` und einer Funktion `F` wird ein Objekt `X` vom Typ `F` erzeugt.

Im ersten Fall hat die Eigenschaft `F.constructor` den Wert `Function`, die Eigenschaft `F.prototype` den Wert `Object` (da ja alle Objekte standardmäßig vom generischen Objekt `Object` abgeleitet werden.).

Im zweiten Fall wird die Eigenschaft `X.constructor` auf den Wert `F` gesetzt, da `X` durch `F` erzeugt wurde. Die Eigenschaft `X.prototype` enthält einen Verweis auf `F.prototype`. Das bedeutet, dass Änderungen an `F.prototype` zur Laufzeit auf alle Objekte vom Typ `F` Auswirkungen haben.

Um den Prototyp aller Objekte `X` des Typs `F` zu ändern, weist man der Eigenschaft `F.prototype` eine Instanz des gewünschten neuen Prototypobjektes zu:

`F.prototype = new F'(`[*Argumente*]`);`

Das Objekt vom Typ `F'` hat seinerseits wiederum ein Prototypobjekt. So entsteht eine Vererbungsfolge, die man *Prototypenkette* nennt und an deren Anfang stets das generische Objekt `Object` steht.

Zugriff auf Eigenschaften in der Prototypenkette

Ein Objekt `X` vom Typ `F` hat Zugriff auf die Eigenschaften der Objekte, die in der Prototypenkette vor ihm stehen. Man spricht dabei von einer *lokalen* Eigenschaft, wenn diese im Objekt `X` selbst existiert, das heißt in `X` erzeugt worden ist. Bei *lesenden* Zugriff auf eine Eigenschaft `e` mit `X.e` geht der JavaScript-Interpreter dabei wie folgt vor:

1. Ist die Eigenschaft `e` *lokal*, dann gib deren Wert zurück.
2. Ist `e` in `X` nicht definiert und `X.prototype != null`, dann betrachte das Prototypobjekt und fahre mit Schritt 1 fort.
3. Ist `e` in `X` nicht definiert und `X.prototype == null`, dann gib `undefined` zurück[3].

Beim *schreibenden* Zugriff auf eine Eigenschaft `e` mit `X.e = y`, wobei `y` ein beliebiger Ausdruck ist, sieht der Ablauf anders aus:

1. Ist die Eigenschaft `e` *lokal*, dann ändere den Wert von `e` auf `y`.
2. Ist die Eigenschaft `e` *nicht lokal*, dann erzeuge eine lokale Eigenschaft `X.e` und setze deren Wert auf `y`.

Möchte man den Wert einer Eigenschaft `e` für alle Objekte `X` vom Typ `F` ändern, so muss man diese Eigenschaft im Prototypobjekt ändern:

```
X.prototype.e = y;
```

Hierbei ist `y` ein beliebiger Ausdruck.

[3]Man ist dann am Anfang der Prototypkette, bei `Object` angelangt

2.11 Programme

JavaScript-Programme - auch Scripts oder Skripte genannt - sind Folgen von Anweisungen und Funktionsdeklarationen in beliebiger Reihenfolge. Wird das JavaScript-Programm zeilenweise interpretiert (beispielsweise in einer Shell), so ist ferner zu beachten, dass Funktionsaufrufe erst nach Deklaration der betreffenden Funktion erfolgen dürfen.

Kapitel 3

HTML

3.1 Einführung

HTML wurde ursprünglich am CERN *(Organisation Européenne pour la Recherche Nucléaire)* [Org] in Genf entwickelt. Die Physiker am CERN suchten Ende der achtziger Jahre nach einer Möglichkeit, ihre gewonnenen Erkenntnisse so zu sammeln und zu organisieren, so dass alle forschenden Teams darauf zugreifen konnten. Tim Berners-Lee, ein Informatiker im Rechenzentrum von CERN, hatte die Idee, diesen Informationspool nicht lediglich als eine reine Ansammlung von Dateien anzulegen, sondern die Texte selbst in diesen Dateien miteinander zu verknüpfen. Auf diese Art und Weise wäre es möglich, relevante Informationen aus anderen Dateien beim Lesen auf Knopfdruck aufzurufen, ohne eine erneute Suche starten zu müssen.

Berners-Lee entschloss sich, das von ihm entwickelte HTML stark an SGML (*Standard Generalized Markup Language*) [SGM01] anzulehnen, da SGML sich bereits als internationaler Standard durchgesetzt hatte und sich außerdem auf jedem Computer implementieren ließ. Er übernahm die grundlegenden Eigenschaften von SGML, wie beispielsweise die Unabhängigkeit des betrachteten Dokuments von der zum Betrachten verwendeten Software, und fügte die Möglichkeit hinzu, verschiedene Dokumente mit einander zu verknüpfen, zu "verlinken".

HTML weckte schnell weltweites Interesse, naturgemäß vor allem an den Universitäten; bereits 1992 begann man am NCSA (*National Center for Supercomputer Applications*) [Nat] an der Universität von Illinois mit der Entwicklung von *Mosaic*, dem ersten grafischen Browser für HTML-Dokumente. *Mosaic* implementierte zusätzlich zu Tim Berners-Lee's vorgeschlagenem Standard noch einige andere Funktionalitäten wie Formulare und Bilder, und wurde 1993 erstmals der Öffentlichkeit in einer Version für SUN Workstations zur Verfügung gestellt.

Seit Dezember 1994 beschäftigt sich das W3C (*World Wide Web Consortium*) [Wor] mit der Weiterentwicklung von HTML und mittlerweile auch anderer Standards im World Wide Web. Wir werden uns im weiteren Verlauf mit der Version 4.01 von HTML beschäftigen, die vom W3C im Dezember 1999 veröffentlicht worden ist [RLHJ99]. Diese HTML-Version ist momentan am weitesten verbreitet, was die Implementierung in Browsern und Anzahl an Webdokumenten betrifft, und wurde auch in unserem Projekt als Standard zugrunde gelegt.

3.2 Die Konstrukte von HTML

Wie der Name *Mark-up Language* schon sagt, handelt es sich bei HTML nicht um eine Programmiersprache im engeren Sinn. Man spricht daher auch nicht von einem HTML-*Programm*, sondern von einem HTML-*Dokument*. Ein HTML-Dokument besteht üblicherweise aus dem *Dokumentinhalt* und *Anweisungen*, wie dieser Inhalt dem Betrachter darzustellen ist. Der Dokumentinhalt besteht dabei aus Textbausteinen und/oder Verweisen auf andere Inhaltsquellen, die Anweisungen zur Darstellung bzw. Wiedergabe des Inhaltes bestehen aus sogenannten *Elementen*. Ein Programm, dass HTML-Dokumente darstellen will, generiert die Ausgabe indem es die Bedeutung der Elemente interpretiert und auf den Dokumentinhalt anwendet. Ein solches Programm wird *user agent* oder *Benutzeragent* genannt.

Die Art, wie die einzelnen Elemente interpretiert werden, ist vom Agenten abhängig. Das Element <B> wird beispielsweise von visuellen HTML-Browsern so interpretiert, dass der eingeschlossene Text fett dargestellt wird. Ein Benutzeragent, der HTML-Dokumente in Sprache übersetzt, könnte es hingegen durch eine Erhöhung der Lautstärke interpretieren.

Da HTML-Dokumente jedoch in den häufigsten Fällen mit Hilfe von grafischen Browsern wie dem Microsoft Internet Explorer oder dem Netscape Communicator dargestellt werden, werden wir uns im weiteren Verlauf lediglich mit der visuellen Art der Interpretation von Elementen beschäftigen.

3.2.1 Elemente

Elemente definieren die Struktur und das Verhalten von Dokumentteilen. So gibt es beispielsweise Elemente für Hyperlinks, Absätze, Tabellen oder Bilder. Jedes Element hat einen Namen, den man auch *Elementtyp* nennt. Das Vorkommen eines Elementes in einem HTML-Dokument nennt man *Deklaration*. Die Deklaration eines Elements umfasst normalerweise drei Teile: einen *Start-Tag*, den *Inhalt* und einen *End-Tag*. Ein Tag besteht aus dem Zeichen < gefolgt vom Elementnamen und dem Zeichen >. Der End-Tag unterscheidet sich vom Start-Tag durch das Voranstellen des Zeichens / vor den Elementnamen:

<Elementname> Inhalt </Elementname>

Es wird bei Tag-Namen nicht hinsichtlich der Groß- und Kleinschreibung unterschieden. Der Übersicht halber werden wir Element im weiteren Verlauf dieser Arbeit immer in Großbuchstaben schreiben. Elemente können geschachtelt werden, wenn es nicht besonders vermerkt ist, sie dürfen sich jedoch nicht überlappen.

Es gibt einige Elementtypen, die es dem Benutzer gestatten, den End-Tag wegzulassen, wie beispielsweise der Elementtyp für Absätze <P>. Andere Elementtypen haben keinen Inhalt, so der Typ <BR>, der einen Zeilenumbruch repräsentiert. Auf diese Besonderheiten wird an geeigneter Stelle näher eingegangen.

3.2.2 Attribute

Elemente können Eigenschaften haben, die *Attribute* genannt werden. Man unterscheidet prinzipiell zwischen *Attributen mit Wert* und *booleschen Attributen*. Attribute mit Wert

bestehen aus einem Attributnamen, gefolgt von einem Gleichheitszeichen und einem Wert in einfachen oder doppelten Anführungszeichen:

Attributname = '*Wert*'

Boolesche Attribute bestehen lediglich aus einem Attributnamen:

Attributname

Attribute dürfen nur im Start-Tag eines Elementes deklariert werden. Es dürfen beliebig viele, für das betreffende Element zulässige, Attribute gesetzt werden. Die Reihenfolge spielt dabei keine Rolle. Bei Attributnamen wird grundsätzlich nicht hinsichtlich Groß- und Kleinschreibung unterschieden, bei Attributwerten gilt dies nicht uneingeschränkt.

Es gibt Attribute, die für die meisten Elemente zulässig sind. Man unterteilt sie in die Kategorien *Kernattribute* (siehe Anhang B.1, Tabelle B.1), *Internationalisierungsattribute* Anhang B.1, Tabelle B.2) und *Scripting Events* (Anhang B.1, Tabelle B.3). Der Übersicht halber werden wir Attribute im weiteren Verlauf dieser Arbeit immer in Kleinbuchstaben schreiben.

3.2.3 Sonderzeichen

Sonderzeichen werden in HTML mittels *Referenzen* dargestellt; eine Referenz kann dabei entweder ein numerischer oder ein symbolischer Wert sein. Im letzteren Fall spricht man dann auch von einer *Entität*. Referenzen beginnen immer mit einem & und enden mit einem Semikolon (;). Einige Beispiele für Sonderzeichen:

- `<` und `>` repräsentieren das <- bzw. >-Zeichen.
- `å` repräsentiert das Zeichen å(die Referenz ist dabei dezimal).
- `水` repräsentiert das chinesische Zeichen für Wasser (die Referenz ist dabei hexadezimal).

Eine vollständige Liste aller Entitäten findet sich unter

http://www.htmlhelp.com/reference/html40/entities

die numerischen Referenzen entsprechen dem Unicode-Standard [Con00].

3.2.4 Kommentare

Kommentare in HTML beginnen mit der Sequenz `<!--` und enden mit der Sequenz `-->`. Der ganze Text, der sich zwischen diesen beiden Sequenzen befindet, wird als Kommentar behandelt. Es spielt dabei keine Rolle, ob sich der Kommentar in einer Zeile befindet oder sich über mehrere Zeilen erstreckt.

```
<!-- Das ist ein Kommentar -->
<!-- Das ist auch ein Kommentar,
     der sich allerdings über mehrere Zeilen erstreckt -->
```

3.3 Die Struktur eines HTML-Dokumentes

Ein HTML-Dokument besteht aus drei Teilen:

1. Einer Zeile, die die HTML Versionsinformation enthält, *Prolog* genannt.
2. Einem deklarativen Kopfteil, eingeschlossen vom Element `HEAD`.
3. Einem Rumpfteil, das den eigentlichen Dokumentinhalt enthält, entweder eingefasst vom Element `BODY` oder `FRAMESET`.

Dabei werden der Kopfteil und der Rumpfteil noch einmal zusätzlich vom Element `HTML` begrenzt. Hier ein Beispiel für ein sehr einfaches HTML-Dokument:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<HTML>
  <HEAD>
    <TITLE>Mein erstes HTML-Dokument</TITLE>
  </HEAD>
  <BODY>
    <P>Hallo Welt!
  </BODY>
</HTML>
```

3.3.1 Der Prolog

Der Prolog gibt an, welche Version von HTML im vorliegenden Dokument verwendet worden ist. Da HTML wie bereits erwähnt auf SGML basiert, wird diese Angabe mittels der *document type declaration* (DTD) gemacht. HTML kennt drei verschiedene DTDs:

1. Die *strikte* Deklaration verwendet alle Elemente und Attribute in HTML, die nicht veraltet sind oder für Dokumente mit Frames benötigt werden. Die entsprechende HTML-Schreibweise für Dokumente mit dieser DTD lautet:

   ```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
   ```

2. Die *transitionale* Deklaration beinhaltet alle Elemente aus der strikten Deklaration plus die bereits veralteten Elemente und Attribute. Die entsprechende Versionsinformation sieht folgendermaßen aus:

   ```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
   ```

3. Die *Frameset*-DTD beinhaltet zusätzlich zur transitionalen auch noch alle Elemente, die man für die Verwendung von Frames braucht. Die Versionsinformation sieht dann folgendermaßen aus:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN"
"http://www.w3.org/TR/html4/frameset.dtd">
```

Die URL, die der eigentlichen Deklaration hinzugefügt ist, gibt dabei die Adresse an, unter der der Browser die benötigten DTDs und Elemente herunterladen kann.

Nach der Document Type Declaration folgt der eigentliche HTML-Teil des Dokumentes, der von den beiden Tags `<HTML>` und `</HTML>` begrenzt wird, und sich in den Kopfteil und den Rumpf untergliedert.

3.3.2 Der Kopfteil

Der Kopfteil - auch *HEAD-Element* genannt - enthält Informationen über das aktuelle Dokument, wie beispielsweise den Titel und Schlüsselwörter für Suchmaschinen. Der Inhalt des Kopfteils wird normalerweise von Browsern nicht sichtbar dargestellt.

Das Element `TITLE`

Jedes HTML-Dokument muss genau ein Titelelement enthalten. Es wird von den Tags `<TITLE>` und `</TITLE>` eingefasst und enhält dazwischen den Titel des Dokuments. Dokumenttitel werden üblicherweise in der Kopfzeile des Browserfensters angezeigt. Außer den bereits erwähnten Sonderzeichen-Referenzen sind keine anderen Elemente im Element `TITLE` erlaubt.

Attribute:

- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2

Meta-Daten

Meta-Daten nennt man die optinionalen Informationen über die Struktur und den Inhalt eines Dokumentes. Normalerweise werden diese nicht von Browsern dargestellt. Sie dienen vor allem der maschinellen Verarbeitung, wie beispielsweise der Indizierung durch Suchmaschinenrobots. Zur Spezifikation von Meta-Daten wird das Element `META` verwendet, welches nur aus einem Tag `<META>` besteht. Innerhalb dieses Tags wird eine Eigenschaft spezifiziert und dieser dann ein Wert zugewiesen. In diesem Beispiel wird der Autor eines Dokuments hinterlegt:

```
<META name = "Author" content = "Thomas In der Rieden">
```

Das Attribut `name` legt dabei den Namen der Eigenschaft fest, das Attribut `content` den ihr zugewiesenen Wert. Sinnvollerweise sollte das Attribut `lang` (siehe auch Anhang B.2) verwendet werden, um die Sprache des Werts von `content` zu spezifieren, beispielsweise:

```
<-- Schlüsselwörter in Deutsch -->
<META name = "keywords"
      content = "Urlaub, Griechenland, Sonne" lang = "de">

<-- Keywords in US-English -->
<META name = "keywords"
      content = "vacation, Greece, sunshine" lang = "en-us">
```

Für eine vollständige Liste aller Länderkürzel siehe [Alv95]. Hinsichtlich der Benennung der Eigenschaften hat sich - was das WWW angeht - der *Dublin Core* durchgesetzt [Dub].

Attribute:

- `name`: spezifiziert den Namen einer Eigenschaft.
- `value`: spezifiziert den Wert einer Eigenschaft.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2

3.3.3 Der Rumpf

Der Rumpf eines HTML-Dokumentes wird von den beiden Tags `<BODY>` und `</BODY>` begrenzt und enthält den eigentlichen Inhalt des Dokuments. Alternativ kann der Rumpf auch durch ein `FRAMESET` definiert werden. Die Verwendung von Frames wird in Kapitel 3.11 genauer erläutert.

Bei der Strukturierung des Dokumentinhaltes im Rumpf unterscheidet man zwischen zwei Arten von Elementen:

- *inline* oder *text-level* Elemente können lediglich Daten oder andere *text-level* Elemente beinhalten. Sie erzeugen im Gegensatz zu *block-level* Elementen keinen neuen Absatz, wenn sie gerendert werden.
- *block-level* Elemente können *text-level* oder andere *block-level* Elemente enthalten. Sie generieren einen neuen Absatz, wenn sie visualisiert werden.

Gruppierungselemente

Die Elemente `DIV` und `SPAN` werden verwendet, um mit Hilfe der Attribute `id` und `class` (siehe dazu auch Anhang B.1) strukturelle Einheiten innerhalb eines Dokumentes zu erzeugen. Die beiden Elemente definieren ihren Inhalt als *text-level* (`SPAN`) oder *block-level* (`DIV`), haben aber keinerlei Auswirkung auf dessen Darstellung. Autoren von HTML-Dokumenten können mit Hilfe von *Stylesheets*, auf die wir in Kapitel 3.9 noch näher eingehen werden, die Elemente `SPAN` und `DIV` dazu verwenden, das Erscheinungsbild von einzelnen Gruppierungen ihren Wünschen anzupassen.

Attribute:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.

- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `align`, siehe Kapitel 3.10.
- `style`, siehe Kapitel 3.9.

Überschriften

In HTML werden Überschriften durch die Elemente `H1` bis `H6` realisiert, wobei `H1` die höchste Hierarchie darstellt und `H6` die niedrigste. Text, der von einem `H` Element umschlossen ist, wird entsprechend seiner Bedeutung gerendert: wichtigere Überschriften haben größere Schriftarten und sind fetter dargestellt als weniger wichtige. Benutzeragenten können beispielsweise die `H` Elemente eines Dokumentes dazu verwenden, um automatisch ein Inhaltsverzeichnis zu erstellen. In der hier betrachteten HTML-Version 4.01 wird leider noch keine automatische Nummerierung von Kapiteln und Unterkapiteln unterstützt.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `align`, siehe Kapitel 3.10.
- `style`, siehe Kapitel 3.9.

3.4 Text

3.4.1 Leerzeichen

HTML interpretiert Leerzeichen und Tabulatorzeichen im Quelltext als Trennzeichen zwischen zwei Wörtern; dabei spielt es keine Rolle, ob es sich um ein Leerzeichen oder mehrere handelt. Autoren, die ein explizites Leerzeichen im gerenderten Text erzeugen wollen, müssen dies mit Hilfe der Entität (siehe Kapitel 3.2.3) ` ` ('non breakable space') realisieren. Eine Ausnahme dazu bildet das `PRE` Element, welches im Abschnitt 3.4.3 näher erläutert wird.

3.4.2 Strukturierter Text

Phrasenelemente

Phrasenelemente dienen dazu, Textfragmente zu strukturieren. Dabei geht es erst in zweiter Linie um das grafische Erscheinungsbild, dass bei vielen Phrasenelemente gleich ist.

Vielmehr wird durch Verwendung von Phrasenelementen dem Inhalt eine Bedeutung verliehen, beispielsweise "das hier ist ein Zitat". Diese strukturelle Information kann dann beispielsweise von Suchmaschinen dazu verwendet werden, die Bibliographie eines Dokumentes zusammenzustellen. Folgend die wichtigsten Phrasenelemente in HTML 4.01:

- `EM` (*emphasis*): Hervorhebung von Text; wird üblicherweise als *kursiv* interpretiert.
- `STRONG`: starke Hervorhebung von Text; wird üblicherwesie als **fett** interpretiert.
- `CITE`: markiert Zitate; wird üblicherweise als *kursiv* dargestellt.
- `CODE`: markiert Computercode, beispielsweise Programmquelltext. Browser stellen `CODE` üblicherweise in einer Serifenschriftart mit fester Zeichenbreite dar.

Alle hier aufgezählten Phrasenelemente sind text-level Elemente.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.

Zitate

In HTML gibt es zwei Möglichkeiten, Zitate hervorzuheben. Möchte man einen ganzen Textabschnitt als Zitat kennzeichnen, so verwendet man das block-level Element `BLOCKQUOTE`. `BLOCKQUOTE` wird üblicherweise als eingezogener Textblock mit breiterem linken und rechten Rand dargestellt; zusätzlich wird noch etwas Leerraum ober- und unterhalb des Zitatblockes erzeugt.

Ein Autor, der lediglich ein Satzfragment als Zitat kennzeichnen will, sollte das text-level Element `Q` verwenden. Browser sollten den Text, der sich innerhalb der Tags von `Q` befindet, in Anführungszeichen wiedergeben [1].

Zur Sprachspezifikation kann das gemeinsame Attribut `lang` (siehe Anhang B.1) verwendet werden. Außerdem kann dem Attribute `cite` eine URI (siehe Kapitel 3.7.1) zugewiesen werden, die auf eine Online-Quelle verweist, aus der das Zitat stammt. Es ist geplant, dass zukünftige Benutzeragenten daraus einen Link auf die Quelle generieren werden und sollte deshalb wenn möglich bereits heute verwendet werden. Folgend ein Beispiel, dass einen Auszug aus J.R.R. Tolkiens *"Die zwei Türme"* im englischsprachigen Original als Blockzitat formatiert:

```
<BLOCKQUOTE cite="http://www.mycom.com/tolkien/twotowers.html">
<P>They went in single file, running like hounds on a strong scent,
```

[1] Leider ist das nicht immer der Fall. Beispielsweise lässt der Microsoft Internet Explorer die Anführungszeichen einfach weg.

```
and an eager light was in their eyes. Nearly due west the broad
swath of the marching Orcs tramped its ugly slot; the sweet grass
of Rohan had been bruised and blackened as they passed.</P>
</BLOCKQUOTE>
```

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `cite`: enthält eine URI, die auf eine Online-Quelle des Zitats verweist.

Hoch- und Tiefstellung

Zur Hochstellung von Text verwendet man das Element `SUP`, zur Tiefstellung das Element `SUB`. Beides sind text-level Elemente.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.

3.4.3 Zeilen und Absätze

Absätze

Das Element `P` repräsentiert einen Textabsatz in HTML. Die Verwendung eines End-Tags ist bei `P` optional. In `P` dürfen keine anderen block-level Elemente vorkommen, insbesondere auch kein weiteres `P` Element. Üblicherweise generiert der Browser eine Leerzeile vor dem von `P` umschlossenen Textblock. Eine Besonderheit des `P`-Elements ist der optionale End-Tag. Da ja keine anderen block-level Elemente in `P` vorkommen dürfen, interpretieren Benutzeragenten den Start-Tag eines block-level Elements gleichzeitig als Ende eines vorherigen `P`-Elements. Absätze können mittels des Attributs `align` ausgerichtet werden. Näheres dazu findet sich im Abschnitt 3.10.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.

- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `align`, siehe Kapitel 3.10.
- `style`, siehe Kapitel 3.9.

Zeilenumbrüche

Zeilenumbrüche im Quelltext eines HTML-Dokumentes werden ebenso wie Leerzeichen (siehe Kapitel 3.4.1) als Trennzeichen für zwei Wörter im Dokumenttext verstanden. Manuelle Zeilenumbrüche können mit dem Element `BR` generiert werden. Bei diesem Element wird lediglich der Start-Tag verwendet (<BR>), die Verwendung des End-Tags ist verboten.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `style`, siehe Kapitel 3.9.
- `clear`: gibt an, ob der Text ein *floating object* umgibt oder ob er unterhalb dieses Objekts fortgesetzt wird. Siehe Kapitel 3.10.1

Silbentrennung

HTML lässt prinzipiell die automatische Silbentrennung zu. Es gibt derzeit jedoch noch keinen Browser, der das implementiert. Im Gegensatz zu Zeilenumbrüchen und Leerzeichen gibt es hier auch keine Möglichkeit, manuell einzugreifen. Zeilenumbrüche erfolgen daher generell zwischen zwei Wörtern, das Zeichen - wird nicht als besonderes Trennzeichen behandelt.

Vorformatierter Text

Die Verwendung des Elements `PRE` ('preformatted') weist den Browser darauf hin, dass der von den beiden Tags eingeschlossene Text vorformatiert ist. Üblicherweise rendern Browser den Text daraufhin folgendermaßen:

- Der Text wird in einer Schriftart dargestellt, in der alle Buchstaben dieselbe Breite haben.
- Leerzeichen im Quelltext werden als echte Leerzeichen interpretiert und dargestellt.
- Tabulatorzeichen werden wie die entsprechende Anzahl an Leerzeichen interpretiert und dargestellt.
- Zeilenumbrüche im Quelltext werden als echte Zeilenumbrüche interpretiert und dargestellt.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `align`, siehe Kapitel 3.10.
- `style`, siehe Kapitel 3.9.

3.5 Listen und Aufzählungen

HTML bietet mehrere Mechanismen, um Text in Form von Listen und Aufzählungen zu generieren. Jede Liste/Aufzählung muss mindestens einen Eintrag enthalten. Listen/Aufzählungen können dabei folgende Arten von Einträgen enthalten:

- Ungeordnete Informationen (Listen)
- Geordnete Informationen (Aufzählungen)
- Definitionen

3.5.1 Ungeordnete Listen, geordnete Listen und Listeneinträge

Die beiden Listenelemente `UL` (*unordered list*) und `OL` (*ordered list*) bestehen beide aus Folgen von Listeneinträgen, die durch das Element `LI` definiert sind. Bei dem Element `UL` wird den einzelnen Listeneinträgen ein Symbol, bei dem Element `OL` Zahlen vorangestellt. Sowohl das Aufzählungszeichen als auch das Zahlenformat kann durch Verwendung von *style sheets* (siehe Kapitel 3.9) definiert werden.

Ein Listeneintrag wird durch das Element `LI` spezifiziert. Die Verwendung des End-Tags ist optional. Wird das Element `LI` im Element `OL` verwendet, so kann das Attribut `value` dazu verwendet werden, den Beginn der Aufzählung festzulegen. Ein Beispiel für eine ungeordnete Liste:

```
<UL>
  <LI> Erster Listeneintrag
  <LI> Zweiter Listeneintrag
  ...
</UL>
```

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.

- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `start`: nur bei `OL` gültig. Bestimmt den Startwert, bei dem die Nummerierung beginnt.
- `value`: nur bei `LI` gültig. Gibt die Nummerierung des betreffenden Listeneintrags an.

3.5.2 Definitionslisten

Definitionslisten werden durch das Element `DL` ("definition list") realisiert und können zwei Arten von Listeneinträgen enthalten:

- *Begriffe*, definiert durch das Element `DT`, und ihre
- *Beschreibung*, definiert durch das Element `DD`

Dabei spielt es keine Rolle, in welcher Reihenfolge oder Anzahl diese Listeneinträge folgen. Es ist durchaus möglich eine Beschreibung für mehrere Begriffe oder mehrere Beschreibungen für einen Begriff anzugeben. Auch bei diesen Listeneinträgen ist der End-Tag optional. Browser rendern Definitionslisten üblicherweise so, dass die die `DD`-Elemente etwas nach rechts einrücken.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.

3.6 Tabellen

Tabellen erlauben es dem Autor eines HTML-Dokumentes, Inhalte wie Bilder, Text oder andere Tabellen, anzuordnen. Obwohl es in den neueren Versionen von HTML andere Möglichkeiten gibt, Inhalte innerhalb des Dokumentes zu positionieren, so werden dennoch häufig aus Kompatibilitätsgründen Tabellen dafür verwendet.

Eine Tabelle besteht dazu aus *Zellen*, die in *Zeilen* und *Spalten* angeordnet sind. Jede Tabelle kann genau eine *Bezeichnung* haben. Zeilen bzw. Spalten können in *Gruppierungen* zusammengefasst werden. Eine Zelle enthält entweder *Kopfzeileninformation* oder *Daten*. Zellen dürfen sich über mehrere Zeilen und/oder Spalten erstrecken.

3.6.1 Die Elemente einer Tabelle

Das Element TABLE

Das Element TABLE stellt den Behälter für alle anderen Elemente dar, die die Formatierung der Tabelle bestimmen. Das Attribut width bestimmt dabei die Breite der Tabelle, entweder relativ zur Fenstergröße des Browsers (z.B. width = '70%') oder absolut in Pixeln (z.B. width = '700').

Attribute:

- id,class: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- lang: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (siehe Anhang B.1, Tabelle B.3).
- align, siehe Kapitel 3.10.
- style, siehe Kapitel 3.9.
- width: die Breite der Tabelle in Pixeln oder in Prozent.
- cellpadding und cellspacing: siehe Kapitel 3.6.2.
- frames, rules und borders: siehe Kapitel 3.6.2.
- bgcolor: siehe Kapitel 3.10.

Das Element CAPTION

Das Element CAPTION definiert den Titel der Tabelle. CAPTION ist optional; wenn es verwendet wird, muss es direkt auf den Start-Tag von TABLE folgen.

Attribute:

- id,class: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- lang: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup (siehe Anhang B.1, Tabelle B.3).
- style, siehe Kapitel 3.9.
- align: Position des Tabellentitels. Erlaubte Werte: *top, bottom, left, right.*

Zeilengruppierungen

Die Zeilen einer Tabelle können in den *Tabellenkopf*, den *Tabellenfuß* und einen oder mehrere *Tabellenrümpfe* gruppiert werden. Die entsprechenden Elemente dafür sind `THEAD`, `TFOOT` und `TBODY`. Jede Gruppierung muss mindestens eine Zeile enthalten. Die Verwendung von `TBODY` ist obligatorisch, es sei denn, die Tabelle enthält keinen Tabellenkopf und -fuß und lediglich einen Rumpf. Die End-Tags können bei allen drei Elementen weggelassen werden. Wird das Element `TFOOT` verwendet, so muss es vor dem ersten `TBODY` innerhalb des Elements `TABLE` stehen.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `align`, `valign`: siehe Kapitel 3.6.2.

Spaltengruppierungen

Spaltengruppierungen dienen der strukturellen Unterteilung innerhalb einer Tabelle. Eine Tabelle hat entweder eine implizite Spaltengruppierung (es wird kein `COLGROUP`-Element spezifiziert) oder eine beliebige Anzahl von Spaltengruppierungen, von denen jede durch ein Element des Typs `COLGROUP` definiert wird.

HTML stellt zwei Elemente zur Realisierung von Spaltengruppierungen zur Verfügung, `COLGROUP` und `COL`. `COLGROUP` erzeugt eine explizite Spaltengruppierung innerhalb einer Tabelle. Um die Anzahl der Spalten in dieser Gruppierung festzulegen, existieren zwei einander ausschließende Möglichkeiten:

1. Das Attribut `span` hat einen ganzzahligen Wert größer 0. Dann ist dieser Wert die Anzahl der Spalten in der Gruppierung.
2. Jedes Element vom Typ `COL` innerhalb des Elements `COLGROUP` spezifiziert eine oder mehr Spalten in der Gruppierung.

Verwendet man erstere Möglichkeit zur Festlegung der Spaltenanzahl in der Gruppierung, so kann man mittels des Attributs `width` die Breite der einzelnen Spalten in Pixeln oder als Prozentzahl definieren. Die Verwendung des End-Tags bei `COLGROUP` ist optional.

Das Element `COL` erlaubt es dem Autor, Spalten in einer Gruppierung noch einmal in Untergruppierungen zu zerlegen. Mit Hilfe des Attributs `span` kann auch hier die Anzahl der Spalten in der Untergruppierung festgelegt werden. Das Attribut `width` spezifiziert wiederum die Breite der einzelnen Spalten. Im Gegensatz zu `COLGROUP` gruppiert `COL` Spalten nicht strukturell. `COL`-Elemente sind daher inhaltslos, die Verwendung des End-Tags ist verboten. `COL` dient vielmehr dazu, Attribute auf mehrere Spalten anzuwenden, beispielsweise Formatierungsanweisungen (siehe dazu Abschnitt 3.10).

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `align`, `valign`: siehe Kapitel 3.6.2.
- `span`: ganze Zahl > 0. Gibt die Anzahl von Spalten an, die in der Gruppierung sind.
- `width`: die Breite der einzelnen Spalten in Pixeln oder Prozentzahlen.

Zeilen

Zeilen in einer HTML-Tabelle werden durch das Elemente `TR` (*table row*) repräsentiert. Der End-Tag dieses Elements ist optional.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `align`, `valign`: siehe Kapitel 3.6.2.
- `bgcolor`: siehe Kapitel 3.10.

Zellen

Die Zellen einer Tabelle können zwei Arten von Information enthalten: *Daten* oder *Spaltenüberschriften*. Das Element `TD` definiert eine Zelle, die Daten enhält, das Element `TH` eine Zelle, die eine Spaltenüberschrift enthält. Bei beiden Elementen ist die Verwendung des End-Tags optional. Zellen lassen sich mittels der folgenden beiden Attribute in ihrer Größe festlegen:

- `colspan` gibt an, über wieviele Spalten sich die Zelle erstreckt. Der Standardwert von `colspan` ist 1. Der Wert 0 bedeutet, dass sich die Zelle über alle Spalten der Spaltengruppierung erstreckt, in der sie definiert ist.
- `rowspan` gibt an, über wieviele Zeilen sich die Zelle erstreckt. Der Standardwert von `rowspan` ist 1. Der Wert 0 bedeutet, dass sich die Zelle über alle Zeilen der Zeilengruppierung erstreckt, in der sie definiert ist.

Attribute:

- `id,class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`, siehe Kapitel 3.9.
- `align`, `valign`: siehe Kapitel 3.6.2.
- `bgcolor`: siehe Kapitel 3.10.
- `width`: die Breite der Zelle in Pixeln oder als Prozentzahl.
- `height`: die Höhe der Zelle in Pixeln oder als Prozentzahl.
- `rowspan`: Anzahl von Zeilen, über die sich die Zelle erstreckt.
- `colspan`: Anzahl von Spalten, über die sich die Zelle erstreckt.

3.6.2 Tabellenformatierung

Grafische Benutzeragenten formatieren Tabellen anhand verschiedener Attribute. HTML bietet - abgesehen von den Möglichkeiten, die Stylesheets bieten - Mechanismen, um die Rahmenart, die horizontale und vertikale Ausrichtung und die Zelleinbettung zu beeinflussen.

Rahmen und Trennstriche

Zur Steuerung der Rahmenart gibt es drei Attribute.

1. `frame`: spezifiziert, welche Seitenränder der Tabelle sichtbar sind. Mögliche Werte sind:
   - *void*: Standardwert, keine Seitenränder.
   - *above*: nur oberer Rand.
   - *below*: nur unterer Rand.
   - *hsides*: nur oberer und unterer Rand.
   - *lhs*: nur linke Seite.
   - *rhs*: nur rechte Seite.
   - *vsides*: nur linke und rechte Seite.
   - *box*: alle vier Seiten.
   - *border*: alle vier Seiten.
2. `rules`: spezifiziert, welche Trennstriche zwischen den einzelnen Zellen einer Tabelle sichtbar sind. Mögliche Werte sind:

- *none*: Standardwert, keine Trennstriche.
- *groups*: Trennstriche zwischen Spalten- und Zeilengruppierungen.
- *rows*: Trennstriche zwischen einzelnen Zeilen.
- *cols*: Trennstriche zwischen einzelenen Spalten.
- *all*: Trennstriche zwischen allen Zellen.

3. `border`: spezifiziert die Stärke der Seitenränder der Tabelle in Pixel.

Ausrichtung von Tabellenelementen

Um den Inhalt einer Tabellenzelle ausrichten zu können, stellt HTML die beiden Attribute `align` (horizontal) und `valign` (vertikal) zur Verfügung.

`align` kann dabei die folgenden Werte annehmen:

- *left*: Standardwert, linksbündig.
- *right*: rechtsbündig.
- *center*: zentriert.
- *justify*: Blocksatz (wirkt nur bei Textinhalt, ansonsten wie `left`).

`valign` kann die folgenden Werte annehmen:

- *top*: am oberen Rand ausgerichtet.
- *bottom*: am unteren Rand ausgerichtet.
- *middle*: mittig ausgerichtet.
- *baseline*: wirkt nur bei Textinhalt; die ersten Textzeilen aller Zellen einer Zeile werden auf einer Grundlinie ausgerichtet.

Bei der Visualisierung werden Ausrichtungsattribute von außen nach innen ausgewertet, das heißt innere Ausrichtungsattribute (beispielsweise im Element `TD`) haben eine höhere Priorität als äußere (beispielsweise in `TABLE`).

Die Einbettung von Zellen und Zelleninhalten

Es gibt zwei Attribute, die die Einbettung von Zellen und Zelleninhalten steuern:

1. `cellspacing` legt fest, wieviele Pixel Platz der Browser zwischen den einzelnen Zellen und zwischen den Zellen und dem Tabellenrand lassen soll.
2. `cellpadding` legt fest, wieviele Pixel Platz der Browser zwischen dem Zellenrand und dem Inhalt einer Zelle lassen soll.

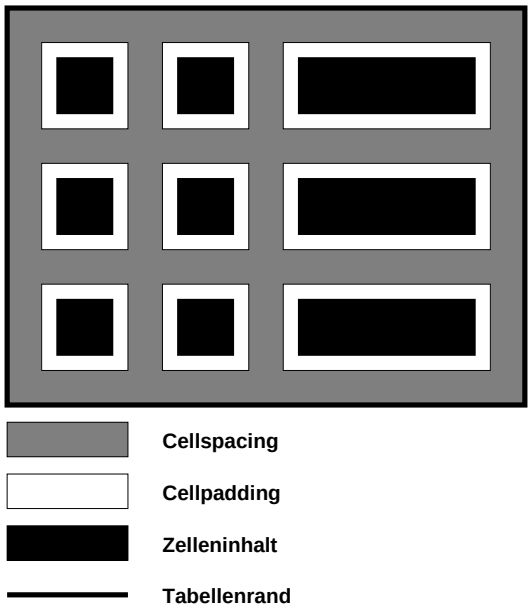

Abbildung 3.1: Cellspacing und -padding

3.6.3 Beispieltabelle

Das folgende Beispiel zeigt den HTML-Code einer einfachen Tabelle und ihre Darstellung im Internet Explorer. Das Beispiel stammt aus Nadine Kano's *Developing International Software*.

```
<TABLE border="2" frame="hsides" rules="groups"
summary="Code page support in different versions
of MS Windows.">
<CAPTION>CODE-PAGE SUPPORT IN MICROSOFT WINDOWS</CAPTION>
<COLGROUP align="center">
<COLGROUP align="left">
<COLGROUP align="center" span="2">
<COLGROUP align="center" span="3">
<THEAD valign="top">
<TR>
<TH>Code-Page<BR>ID
<TH>Name
<TH>ACP
<TH>OEMCP
<TH>Windows<BR>NT 3.1
<TH>Windows<BR>NT 3.51
<TH>Windows<BR>95
<TBODY>
<TR><TD>1200<TD>Unicode (BMP of ISO/IEC-10646)<TD><TD><TD>X<TD>X<TD>*
<TR><TD>1250<TD>Windows 3.1 Eastern European<TD>X<TD><TD>X<TD>X<TD>X
<TR><TD>1251<TD>Windows 3.1 Cyrillic<TD>X<TD><TD>X<TD>X<TD>X
```

```
<TR><TD>1252<TD>Windows 3.1 US (ANSI)<TD>X<TD><TD>X<TD>X<TD>X
<TR><TD>1253<TD>Windows 3.1 Greek<TD>X<TD><TD>X<TD>X<TD>X
<TR><TD>1254<TD>Windows 3.1 Turkish<TD>X<TD><TD>X<TD>X<TD>X
<TR><TD>1255<TD>Hebrew<TD>X<TD><TD><TD><TD>X
<TR><TD>1256<TD>Arabic<TD>X<TD><TD><TD><TD>X
<TR><TD>1257<TD>Baltic<TD>X<TD><TD><TD><TD>X
<TR><TD>1361<TD>Korean (Johab)<TD>X<TD><TD><TD>**<TD>X
<TBODY>
<TR><TD>437<TD>MS-DOS United States<TD><TD>X<TD>X<TD>X<TD>X
<TR><TD>708<TD>Arabic (ASMO 708)<TD><TD>X<TD><TD><TD>X
<TR><TD>709<TD>Arabic (ASMO 449+, BCON V4)<TD><TD>X<TD><TD><TD>X
<TR><TD>710<TD>Arabic (Transparent Arabic)<TD><TD>X<TD><TD><TD>X
<TR><TD>720<TD>Arabic (Transparent ASMO)<TD><TD>X<TD><TD><TD>X
</TABLE>
```

CODE-PAGE SUPPORT IN MICROSOFT WINDOWS

Code-Page ID	Name	ACP	OEMCP	Windows NT 3.1	Windows NT 3.51	Windows 95
1200	Unicode (BMP of ISO/IEC-10646)			X	X	*
1250	Windows 3.1 Eastern European	X		X	X	X
1251	Windows 3.1 Cyrillic	X		X	X	X
1252	Windows 3.1 US (ANSI)	X		X	X	X
1253	Windows 3.1 Greek	X		X	X	X
1254	Windows 3.1 Turkish	X		X	X	X
1255	Hebrew	X				X
1256	Arabic	X				X
1257	Baltic	X				X
1361	Korean (Johab)	X			**	X
437	MS-DOS United States		X	X	X	X
708	Arabic (ASMO 708)		X			X
709	Arabic (ASMO 449+, BCON V4)		X			X
710	Arabic (Transparent Arabic)		X			X
720	Arabic (Transparent ASMO)		X			X

Abbildung 3.2: Beispieltabelle im Internet Explorer

3.7 Links

Links, auf deutsch *Verknüpfungen*, machen das Besondere an HTML gegenüber den meisten anderen Markup Languages aus und sind wohl einer der Hauptgründe für den Erfolg des World Wide Webs. Ein *Link* ist die Verknüpfung einer Resource mit einer anderen. Ein Link hat einen Anfang, *source anchor* oder *Quellanker* genannt und ein Ende, *destination anchor* oder *Zielanker* genannt.

Zur Realisierung von Links in HTML-Dokumenten werden die beiden Elemente `A` und `BASE` verwendet, die wir im folgenden näher betrachten werden. Zuerst müssen wir jedoch die Art und Weise verstehen, wie Resourcen im *World Wide Web* adressiert werden.

3.7.1 Uniform Resource Identifiers - URI

Uniform Resource Identifiers wurden erstmals in [BLFM98] eingeführt und lösen die veralteten *Uniform Resource Locators* aus [BLMM94] ab. Jede Resource im Web hat eine Adresse, die in Form eines URI dargestellt werden kann. Üblicherweise bestehen URI aus drei Teilen:

1. Der Name des Protokolls oder Mechanismus, mit dem auf die Resource zugegriffen werden soll.
2. Der Name des Rechners, auf dem die Resource zu finden ist.
3. Der Name der Resource als Pfad im Verzeichnisbaum.

Ein Beispiel für eine URI ist `http://www.uni-saarland.de/Forschung.html`. Normalerweise ist der verwendete Mechanismus das *HTTP-Protokoll*, wie es in seiner neuesten Version in [FGM+99] spezifiziert ist, andere Möglichkeiten sind beispielsweise *ftp* oder *mailto*.

Fragment Identifiers

Einige URI verweisen auf Stellen innerhalb einer Resource. Solche URI enden mit einem "#" gefolgt von einem sogenannten *fragment identifier*. Das folgende Beispiel verweist auf einen Zielanker namens `kapitel2` innerhalb der Resource `index.html`:

```
http://www.irgendwo.de/index.html#kapitel2
```

Relative URI

Eine relative URI enthält lediglich den Namen der Resource im Verzeichnisbaum relativ zum aktuellen Dokument. In relativen URI dürfen die relativen Pfadkomponenten `..` (übergeordnetes Verzeichnis) und `.` (aktuelles Verzeichnis) verwendet werden.

3.7.2 Das Element `A`

Das Element `A` dient zur Erzeugung eines Ankers. Der Inhalt des Elementes repräsentiert dabei die Position des Ankers (beispielsweise ein Bild oder ein Stück Text). Zur Definition der Verknüpfung verfügt `A` über zwei Attribute:

1. `name` weist dem Element einen Namen zu, so dass es der Zielanker eines oder mehrerer Links sein kann. Der zugewiesene Name muss eindeutig sein (Groß- und Kleinschreibung spielen keine Rolle), wobei der Gültigkeitsbereich das aktuelle HTML-Dokument ist. Ein solcher Anker heißt *fragment identifier* (siehe auch Abschnitt 3.7.1).
2. `href` spezifiziert die URI einer Resource und macht den Anker zu einem Quellanker für genau einen Link.

Elemente vom Typ `A` dürfen nicht geschachtelt werden.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onfocus`, `onblur`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `name`: der eindeutige Name des Ankers bzgl. des Dokuments.
- `href`: enthält die URI einer Webresource.
- `target`: das Zielframe. Siehe Kapitel 3.11.
- `shape` und `coords`: siehe Kapitel 3.8.3.

3.7.3 Das Element `BASE`

Relative URI werden normalerweise relativ zum aktuellen Dokument aufgelöst. Das Element `BASE` erlaubt es dem Autor eines HTML-Dokumentes, eine absolute URI zu spezifizieren, zu der die relativen Links des betreffenden Dokumentes aufgelöst werden. Dazu verfügt das Element `BASE` über das Attribut `href`, dem die Bezugs-URI als Wert zugewiesen wird. Das `BASE`-Element hat lediglich einen Start-Tag und muss im Kopfteil eines HTML-Dokumentes deklariert werden.

Attribute:

- `href`: enthält eine absolute URI, die als Bezugs-URI für relative Links dient.
- `target`: spezifiziert das Zielframe. Siehe Kapitel 3.11.

3.8 Objekte, Bilder und Applets

Während ältere Versionen von HTML lediglich die Einbettung von Bildern und Java-Applets in HTML-Dokumente erlaubten, wurde mit HTML 4.01 die Möglichkeit geschaffen, auch andere Medientypen darzustellen. Dazu wurde das neue Element `OBJECT` geschaffen, das durch seine Flexibilität dem Autor gestattet, nahezu alle Medientypen in Dokumente einzubetten.

Nichtsdestotrotz verwenden immer noch die meisten HTML-Autoren das Element `IMG` zur Einbettung ihrer Bilder. Wir werden daher auch noch auf dieses Element eingehen. Ferner werden wir im Kapitel 3.11 das Element `IFRAME` betrachten, mit dessen Hilfe man andere HTML-Dokumente in bestehende Dokumente einbetten kann.

Es empfiehlt sich jedoch zukünftig, von der Verwendung anderer Elemente als `OBJECT` bei der Einbettung von Medien abzusehen, da es nicht sicher ist, ob diese Elemente in zukünftigen HTML-Versionen noch implementiert sein werden. Im Falle von `APPLET` ist das Element sogar schon als *deprecated - veraltet* eingestuft worden.

3.8.1 Das Element IMG

Bilder werden mit Hilfe des Elements `IMG` in HTML-Dokumente eingebunden. Das Element verfügt über die folgenden wichtigen Attribute:

- `src` (source) spezifiziert eine URI, unter der man das einzubettende Bild findet. Derzeit unterstützen die meisten Browser Bilder der Typen `.jpg`, `.gif` und `.png`.
- `alt` (alternate) enthält einen Text, der alternativ von Browsern dargestellt werden kann, beispielsweise von textbasierten Benutzeragenten.

Im Abschnitt 3.8.4 werden wir noch auf weitere Attribute zur Positionierung von Objekten eingehen. Das Attribut `usemap` zur Verwendung von *image maps* wird im Abschnitt 3.8.3 näher erläutert. Das Element `IMG` hat lediglich einen Start-Tag, die Verwendung des End-Tags ist verboten.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `src`: enthält eine URI, die auf die Bilddatei verweist.
- `width`, `height`: Höhe und Breite des Bildes. Siehe Kapitel 3.8.4.
- `hspace`, `vspace`: siehe Kapitel 3.8.4.
- `border`, `align`: siehe Kapitel 3.8.4.
- `usemap`: siehe Kapitel 3.8.3.
- `alt`: alternativ darzustellender Text.

3.8.2 Die Elemente OBJECT und PARAM

Heutige Browser verfügen bereits über eine große Anzahl von Mechanismen, um Standard Datenformate wie Text, `.gif` und `.jpg` Grafiken sowie verschiedene Schriftarten darstellen zu können. Die fortschreitende Entwicklung neuer Medienformate macht es notwendig, Methoden zu implementieren, um diese Formate darstellen zu können obwohl sie ursprünglich nicht im Browser vorgesehen waren. Zu diesem Zweck wurde die Elemente `OBJECT` und `PARAM` mit HTML 4 eingeführt. Diese Elemente erlauben es dem Dokumentautor festzulegen, wie und wo die Daten dargestellt werden sollen. Dafür müssen im wesentlichen drei Informationen übermittelt werden:

1. Wo befindet sich das einzubindende Objekt?
2. Wo befinden sich die Daten, die dargestellt werden sollen?

3. Gibt es gegebenenfalls noch Parameter, die dem Objekt zur Laufzeit übergeben werden müssen?

Diese drei Informationen sind nicht immer alle notwendig, da nicht alle Objekte Eingabedaten oder Parameter benötigen. Zur Spezifikation dieser Information stellt HTML eine Reihe Attribute zur Verfügung:

- Der Wert von `classid` ist eine URI und verweist auf das einzubettende Objekt.
- Auch `data` enthält gegegebenenfalls eine URI, die auf die Daten des Objekts verweist.
- `codetype` und `type` spezifizieren den Datentyp des Objektes bzw. seiner Daten. Der Wert dieser Attribute ist ein String, der einen *MIME-Typ* repräsentiert, wie sie in [FB96a] und [FB96b] definiert sind.
- Die Basis-URI für relative URI kann mit Hilfe des Attributs `codebase` festgelegt werden.

Parameter, die ein Objekt gegebenenfalls zur Laufzeit benötigt, werden durch das eigene Element `PARAM` übermittelt. Jedes `PARAM`-Element besteht lediglich aus einem Start-Tag und spezifiziert genau einen Parameter. Der Name des Parameters wird durch das Attribut `name`, der Wert des Parameters durch das Attribut `value` spezifiziert. Zusätzlich kann man mittels des Attributs `valuetype` den Wertetyp angeben. Mögliche Werte dieses Attributs sind:

- Der Standardwert `data` bedeutet, dass der Wert von `value` vom Browser ausgewertet wird und dem Objekt als String übergeben wird.
- `ref` bedeutet, dass der Wert von `value` eine URI ist, die auf die Laufzeitparameter verweist. In diesem Fall verwendet man zusätzlich das Attribut `type`, das den Typ der referenzierten Daten als MIME-Typ spezifiziert.
- `object` weist darauf hin, dass der Wert von `value` ein *fragment identifier* (siehe dazu auch 3.7.1) auf ein anderes `OBJECT`-Element im selben Dokument ist.

`PARAM`-Elemente dürfen nur innerhalb eines `OBJECT`-Elementes vorkommen. Dort müssen sie direkt auf den Start-Tag des `OBJECT`-Elementes folgen. Ein Element vom Typ `OBJECT` darf beliebig viele `PARAM`-Elemente beinhalten.

Text, der innerhalb der beiden Tags des `OBJECT`-Elementes steht, wird nur dann dargestellt, wenn der Benutzeragent aus irgendeinem Grund das eingebettete Objekt nicht darstellen kann. `OBJECT`-Elemente sind schachtelbar. Benutzeragenten versuchen, zuerst das am äußersten definierte Objekt darzustellen; falls dies nicht möglich ist, versuchen sie das Objekt der nächsten Schachtelungsebene. Erst wenn keines der geschachtelten Objekte darstellbar ist, wird der möglicherweise beinhaltete Text der innersten Schachtelungsebene ausgegeben.

Im Abschnitt 3.8.4 werden wir noch auf weitere Attribute zur Positionierung von Objekten eingehen.

Attribute von `OBJECT`:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `src`: enthält eine URI, die auf die Bilddatei verweist.
- `width`, `height`: Höhe und Breite des Bildes. Siehe Kapitel 3.8.4.
- `hspace`, `vspace`: siehe Kapitel 3.8.4.
- `border`, `align`: siehe Kapitel 3.8.4.
- `usemap`: siehe Kapitel 3.8.3.
- `classid`: enthält eine URI, die auf die Implementierung des Objekts verweist.
- `data`: enthält eine URI, die auf eventuell benötigte Daten verweist.
- `codetype`: spezifiziert gemäß [FB96a], [FB96b] den MIME-Typ des Objekts.
- `type`: spezifiziert gemäß [FB96a], [FB96b] den MIME-Typ der benötigten Daten.
- `codebase`: absolute Bezugs-URI für relative URIs in `classid`, `data`

Attribute von `PARAM`:

- `id`: Kernattribut, siehe Anhang B.1, Tabelle B.1.
- `name`: Name des Parameters.
- `value`: Wert des Parameters.
- `valuetype`: Art des Parameterwertes, siehe oben.
- `type`: nur gülitg, wenn `valuetype = ref`. Spezifiziert dann gemäß [FB96a], [FB96b] den MIME-Typ der Laufzeitparameter.

3.8.3 Image Maps

Image maps ermöglichen es dem Autor, bestimmte Teile seiner Bilder zu markieren und ihnen eine Aktion zuzuweisen. Diese Aktion wird ausgeführt, wenn der Benutzer die entsprechende Region aktiviert, indem er beispielsweise bei einem grafischen Browser darauf klickt. Es gibt zwei Arten von image maps, *server-seitige* und *client-seitige*. Bei der server-seitigen Lösung werden die Koordinaten relativ zur linken oberen Ecke des Bildes an den Server übermittelt, der dann die eigentliche Aktion ausführt. Bei der client-seitigen Variante wertet der Benutzeragent selbst die Koordinaten aus und bestimmt die betreffende Region. Anschließend folgt er dem Link, der für die entsprechende Region hinterlegt ist. Da die Verwendung von server-seitigen image maps hier zu weit führen würde, betrachten wir lediglich die client-seitigen, die auch am häufigsten zum Einsatz kommen.

Um eine image map zu definieren verwendet man die Elemente `MAP` und `AREA`. Das `MAP`-Element definiert dabei eine client-seitige image map, deren im Dokument eindeutiger Name durch den Wert des Attributs `name` spezifiziert wird. Die Verbindung eines Bildes mit der map erfolgt über diesen Namen, den man dem Attribut `usemap` des betreffenden Bildes zuweist. Ein `MAP`-Element kann beliebig viele `AREA`-Elemente beinhalten. Diese spezifizieren die einzelnen Regionen und den Link, dem gefolgt wird, wenn eine diese Regionen aktiviert wird. Die dazu benötigten Attribute des Elements `AREA` sind:

- `shape` legt die Umrissform der Region fest. Mögliche Werte sind `default` (die ganze Region), `rect` (rechteckig), `circle` (kreisförmig) und `poly` (polygonale Region).
- `coords` definiert die Koordinaten der Region abhängig von der Umrissform:
  - `rect`: x_{links}, y_{oben}, x_{rechts}, y_{unten}.
  - `circle`: x_{Mitte}, y_{Mitte}, $Radius$.
  - `poly`: x_1, y_1,...,x_n,y_n. Der erste Punkt sollte dieselben Koordinaten haben wie der letzte Punkt, damit die Form geschlossen ist. Ist dies nicht der Fall, so berechnen die meisten Benutzeragenten automatisch einen weiteren Zwischenpunkt.
- Das Attribut `href`, das wir bereits aus dem Kapitel 3.7 kennen, verweist auf eine URI, zu der verzweigt wird, wenn die Region aktiviert ist. Möchte man keinen Link spezifizieren, so verwendet man das boolesche Attribut `nohref`.

3.8.4 Die Visualisierung von Objekten

Wie alle Attribute, die das Erscheinungsbild von Objekten und Bildern beeinflussen, sind auch die folgenden seit HTML 4 als *deprecated - veraltet* eingestuft worden. Zukünftig sollen diese Aufgaben von *style sheets*, wie wir sie in Kapitel 3.9 kennenlernen werden, übernommen werden. Da jedoch immer noch die überwältigende Mehrheit der HTML-Autoren die veralteten Attribute aufgrund ihrer einfachen Anwendbarkeit benutzt, werden wird diese hier vorstellen.

Breite und Höhe

Zur Spezifikation von Breite und Höhe von Objekten benutzt man die Attribute `width` und `height`. Beide Attribute können sowohl absolute Werte in Pixeln als auch relative Werte in Prozenten annehmen. Verwendet der Autor relative Größenangaben, so wird als Grundlage der tatsächlich zur Verfügung stehende Platz vom Browser berechnet (beispielsweise die Fensterbreite). Dabei stellt auch der Browser den Vergrößerungs- bzw. Verkleinerungsalgorithmus zur Vefügung.

Freiräume um Objekte

Mit Hilfe der Attribute `hspace` (oben und unten) und `vspace` (rechts und links) kann man etwas Platz um Objekte schaffen. Beide Attribute haben Pixelwerte.

Rahmen

Das Attribut `border` erzeugt einen Rahmen um Objekte und Bilder. Der Wert des Attributs ist in Pixeln angegeben.

Ausrichtung

Das Attribut `align` bestimmt die Ausrichtung eines Objekts hinsichtlich des es umgebenden Textes. Mögliche Werte sind `top` (an der oberen Kante ausgerichtet), `bottom` (Standardwert - an der unteren Kante ausgerichtet) und `middle` (mittig). Ferner gibt es die beiden Möglichkeiten `right` und `left`, die wir allerdings erst im Kapitel 3.10 vorstellen werden.

Liste der Visualisierungsattribute

- `width`, `height`: Breite bzw. Höhe eines Objekts in Pixeln oder als Prozentzahl.
- `vspace`, `hspace`: Freiraum in Pixeln um ein Objekt.
- `border`: Stärke des Rahmens um Objekte in Pixeln.
- `align`: Horizontale Ausrichtung des Objekts: *top, bottom, middle, right, left.*

3.9 Stylesheets

Das Web ist ursprünglich von Wissenschaftlern entwickelt worden, denen es in erster Linie um den Inhalt ihrer Dokumente ging als um deren grafisches Layout. Das heutige Web wird jedoch von quasi jedermann benutzt und die Benutzer stellen immer höhere Ansprüche an das Erscheinungsbild von Internetpräsenzen.

Die anfänglichen Möglichkeiten, die HTML hinsichtlich der Formatierung zur Verfügung gestellt hatte, waren schnell an ihre Grenzen gelangt und auch das Hinzufügen von Elementen und Attributen hat diesen Zustand nur wenig verändert. Das Erstellen grafisch gelungener Webpräsentationen entwickelte sich mehr und mehr zu einer eigenen Wissenschaft, in der der Autor sich oft ausgefallener Hilfskonstruktionen bedienen muss, um sein Ziel zu erreichen.

Um dieses Problem zu beheben und gleichzeitig einen für die Zukunft tauglichen Standard zu entwickeln wurde in HTML 4 das Prinzip der *style sheets* eingeführt. Style sheets lösen eine Reihe von Problemen im Zusammenhang mit der Gestaltung von Webseiten:

- **Flexibilität**. Ein Autor kann style sheets auf mehrere Dateien verteilen. Er kann entscheiden, ob einzelne Formatierungsanweisungen ganze Websites oder nur einzelne Seiten betreffen.
- **Unabhängigkeit**. Es gibt viele verschiedene Sprachen für style sheets, einfache und komplexe. Jeder Autor kann gemäß seinen Wünschen und Bedürfnissen eine passende wählen.

- **Kaskadierung**. Ein Autor kann style sheets auf mehreren hierarchischen Ebenen aufbauen. Dabei überschreiben Formatierungsanweisungen aus später eingebundenen sheets die Anweisungen aus vorhergegangenen.
- **Alternative Stile**. Autoren können verschiedene style sheets für Dokumente zu schreiben, beispielsweise mit verschiedenen Schriftgrößen für verschiedene Auflösungen.

Die am weitesten vebreitete und genutzte Sprache für style sheets ist *Cascading Style Sheets (CSS)*, die in [BL99] und [BLLJ98] definiert wird. Alle Codebeispiele im weiteren Verlauf dieses Kapitels beziehen sich auf style sheets in dieser Sprache.

3.9.1 Das Hinzufügen von styles

Autoren müssen die Sprache ihrer style sheets, die sie in einem Dokument verwenden, dem Benutzeragenten mitteilen. Dies geschieht durch ein `META`-Element im Kopf des HTML-Dokumentes. Beispielsweise definiert

```
<META http-equiv='Content-Style-Type' content='text/css'>
```

CSS als style sheet Sprache. Der zugewiesene Wert ist dabei ein MIME-Typ (siehe [FB96a] und [FB96b]), der die style sheet Sprache repräsentiert.

Zur Definition eines styles zu einem Element gibt es zwei Möglichkeiten. Entweder verwendet man das Attribut `style`, das nahezu alle Elemente zur Verfügung stellen, oder man definiert eine style Klasse im Kopf des Dokuments unter Verwendung des Elements `STYLE`.

Das Attribut `style`

Das Attribut `style` spezifiziert den style des dazugehörigen Elements. Die Syntax des Attributwertes richtet sich dabei nach der im Dokumentkopf festgelegten style sheet Sprache. In CSS sind style Deklarationen ein Paar der Form *Name:Wert*, wobei Mehrfachdeklarationen durch Semikolon voneinander getrennt sind. Ein Beispiel für einen Absatz mit einem style in CSS:

```
<P style='font-size: 12pt; color: fuchsia'>
  Am Anfang erschuf Gott Himmel und Erde.
```

Das Element `STYLE`

Das `STYLE`-Element darf nur im Kopf eines HTML-Dokumentes auftreten. Es erlaubt dem Autor, styles für ganze Gruppen von Elementen zu deklarieren. Ein HTML-Dokument kann eine beliebige Anzahl von `STYLE`-Elementen haben. Die Syntax des Inhalts eines `STYLE`-Elementes hängt dabei von der style sheet Sprache ab. Diese wird entweder durch die im Kopf definierte style sheet Sprache festgelegt oder durch das Attribut `type` des

STYLE-Elementes. Dabei hat der Wert dieses Attributs eine höhere Priorität als die im Kopf festgelegte Sprache.

CSS gestattet es, mit Hilfe des STYLE-Elementes styles für

- alle Instanzen eines Elementes (beispielsweise alle Vorkommen von H1)
- alle Instanzen eines Elementes, die zu einer bestimmten Klasse gehören (d.h., deren class-Attribut ein Wert zugewiesen worden ist (siehe Anhang B.1, Tabelle B.1)). Solche Elementklassen werden durch die Syntax *Elementname.Klassenname* referenziert, wobei *Klassenname* der Wert des Attributs class der betreffenden Elemente ist.
- einzelne Vorkommen eines Elements (d.h., deren id-Attribut ein Wert zugewiesen worden ist (siehe Anhang B.1, Tabelle B.1)). Solche Elemente werden durch die Syntax #*Name* referenziert, wobei *Name* der Wert des Attributs id des betreffenden Elements ist.

Beispiele dafür sind:

```
<!-- Definiert einen style für alle Elemente
     H1 dieses Dokumentes -->
<STYLE type="text/css">
  H1 {border.width: 1; border: solid; text-align: center}
</STYLE>
<!-- Definiert einen Style für alle Elemente P
     der Klasse "klein" in diesem Dokument -->
<STYLE type="text/css">
  P.klein {font-size: 8pt}
</STYLE>
<!-- Definiert einen Style für das Element
     mit der ID "meineklasse" in diesem Dokument -->
<STYLE type="text/css">
  #meineklasse {font-color: red}
</STYLE>
```

Attribute:

- lang: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- type: enthält MIME-Typ gemäß [FB96a], [FB96b], der die Stylesheetsprache spezifiziert.

3.9.2 Externe Style Sheets

Externe style sheets bieten die Möglichkeit, style Deklarationen auf mehrere Dokumente (beispielsweise alle Seiten eines Firmen Webauftritts) anzuwenden ohne sie in jedem Dokument einzeln definieren zu müssen. So wird auch das Verändern von style Deklarationen erleichtert, da nur eine Datei modifiziert werden muss.

Prinzipiell unterscheidet man drei Arten externer style sheets:

1. *Alternative* (alternate) style sheets lassen dem Benutzer die Möglichkeit, sich für einen style zu entscheiden. So können sehbehinderte beispielsweise große Schriftarten für die Darstellung wählen.

2. Der Autor kann eines der alternativen style sheets als *bevorzugt* (preferred) deklarieren, welches vom Benutzeragenten standardmäßig verwendet wird.

3. *Persistente* (persistent) style sheets müssen vom Browser zusätzlich zum gewählten alternativen style sheet bei der Visualisierung befolgt werden.

Zur Spezifizierung externer style sheets verwendet man das Element `LINK`, das lediglich im Kopf von HTML- Dokumenten auftreten darf. `LINK` stellt zu diesem Zwecke folgende Attribute zur Verfügung:

- Der Wert von `href` ist eine URI, die auf das style sheet verweist.
- In `type` ist die Sprache des verlinkten style sheets hinterlegt.
- `rel` und `title` spezifizieren, um welche Art style sheet es sich handelt:
  - bei persistenten style sheets wird `rel` auf den Wert *stylesheet* gesetzt, `title` wird kein Wert zugewiesen.
  - Um ein style sheet zum bevorzugten zu machen, wird `rel` ebenfalls auf *stylesheet* gesetzt. Außerdem muss das style sheet einen Namen bekommen, der `title` zugewiesen wird.
  - Bei alternativen style sheets wird `rel` auf *alternate stylesheet* gesetzt. Außerdem wird dem style sheet ein Name gegeben, der `title` zugewiesen wird.
- Weitere Attribute:
  - `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
  - `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
  - `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

3.10 Ausrichtung und Schriftarten

Im Laufe der Entwicklung von HTML wurden zahlreiche Elemente der Sprache hinzugefügt, die der Beeinflussung der Darstellungsweise von Dokumenten dienen sollen. Aufgrund der strategischen Überlegungen des W3C, zukünftig die Visualisierungsart vom eigentlichen HTML zu entkoppeln sind deshalb viele dieser Elemente als veraltet eingestuft worden. Nichtsdestotrotz verwenden die meisten Autoren von HTML-Dokumenten immer noch diese Elemente, da sie

1. einfach zu handhaben sind,

2. problemlos von den meisten Browsern verarbeitet werden und

3. die schnelle Dokumentformatierung erleichtern.

Aus diesem Grund haben wir uns entschlossen, diese Elemente auch in dieser Arbeit zu besprechen.

3.10.1 Formatierung

Hintergrundfarbe

Mit Hilfe des Attributes `bgcolor`, das im Kontext mit den Elementen `BODY` (siehe Abschnitt 3.3.3), `TABLE`, `TR`, `TH` und `TD` (siehe Abschnitt 3.6) verwendet werden darf, lässt sich die Farbe des Dokumenthintergrundes bzw. des Tabellenhintergrundes festlegen. Der Wert, der diesem Attribut zugewiesen wird, ist dabei der Name einer der 16 vordefinierten Farben in HTML (siehe Anhang B.2), oder ein hexadezimaler RGB-Wert. Dieser Wert setzt sich dabei aus dem Zeichen # gefolgt von einem Byte Rotanteil, einem Byte Grünanteil und einem Byte Blauanteil zusammen. Für nähere Informationen vergleiche [SACM96].

Ausrichtung

Das Attribut `align` ermöglicht es, block-level Elemente, wie Tabellen, Bilder oder Absätze, auszurichten. Die erlaubten Werte, die diesem Attribut zugewiesen werden können, hängen vom Elementkontext ab. Wir werden an dieser Stelle lediglich die Werte vorstellen, die im Zusammenhang mit Text eingesetzt werden können.

- *left*: linksbündig
- *right*: rechtsbündig
- *center*: zentriert
- *justify*: Blocksatz

Floating Objects

Objekte und Bilder werden normalerweise *inline*, also an der Stelle im gerenderten Text dargestellt, an der sie im Quelltext definiert worden sind. *Floating objects* hingegen erlauben es dem Benutzer, diese Objekte an die Ränder der Dokumentseite zu stellen und den Text links oder rechts von ihnen darzustellen.

Zur Realisierung dieser floating objects bietet HTML zwei Attribute, `align` und `clear`. `align` wird mit dem Objektelement verwendet und kann die Werte *left* oder *right* annehmen. `clear` dient dazu, dem Browser mitzuteilen, wo der Text nach einem Zeilenumbruch mit `BR` fortgesetzt werden soll. Dazu kann `clear` die folgenden Werte annehmen:

- *none* (Standardwert): der Text wird normal fortgesetzt.
- *all*: der Text wird in der nächsten Zeile unter einem floating object fortgesetzt.

3.10.2 Schriftarten

Schrifteffekte

Um Textsegmenten Effekte zuzuweisen, wie beispielsweise Schrägstellung oder Hervorhebung, bietet HTML eine Reihe von Elementen. Die letztendliche Darstellung der Effekte auf dem Bildschirm ist abhängig vom Browser und ggf. Browsereinstellungen, die der Benutzer vorgenommen hat. Alle hier vorgestellten Elemente brauchen einen Start- und End-Tag, sie dürfen geschachtelt werden.

- `TT` stellt Text in Schreibmaschinenschrift dar.
- `I` stellt Text in Kursivschreibweise dar.
- `B` lässt Text fett erscheinen.
- `BIG` stellt den Text mit einer großen Schriftart dar.
- `SMALL` stellt den Text mit einer kleinen Schriftart dar.
- `U` unterstreicht den eingeschlossenen Text.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`: siehe Kapitel 3.9.

Die Elemente `FONT` und `BASEFONT`

Das Element `FONT` erlaubt das einfache Formatieren des eingeschlossenen Textes hinsichtlich Schriftart, Größe und Farbe. Dazu hat es drei Attribute:

1. `size` bestimmt die Größe. Erlaubte Werte sind ganze Zahlen zwischen 1 und 7 (wobei 7 die größte Darstellung ist) oder relative Werte wie +3 oder -1. Bei der Verwendung von relativen Werten werden diese auf die momentan gültige Schriftgröße addiert. Ist das Resultat kleiner 1, so wird 1 als Größe genommen, ist es größer als 7 so wird 7 als Größe genommen.
2. `face` bestimmt die Schriftart. Der Wert von `face` ist üblicherweise eine kommagetrennte Liste von Schriftartnamen. Bei der Darstellung versucht der Browser, die erste Schriftart auf dem Benutzersystem zu finden. Ist diese unauffindbar, so versucht er das mit der zweiten usw. Ist keine der Schriftarten verfügbar, so nimmt er die Standardschriftart des Browsers.

3. `color` legt die Farbe des Textes fest. Genau wie bei der Definition der Hintergrundfarbe sind auch hier die vordefinierten HTML-Farben (siehe Anhang B.2) und hexadezimale RGB-Werte erlaubt ([SACM96]).

Das Element `BASEFONT` legt die Basis-Schriftgröße fest, auf die sich die relativen Schriftgrößen beziehen. Dazu wird das Attribut `size` auf die gewünschte absolute Basis-Schriftgröße gesetzt. Wird `BASEFONT` nicht gesetzt, so ist die Basis-Schriftgröße 3.

Attribute:

- `id`,`class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `size`: die Schriftgröße. Erlaubt sind absolute Werte wie 1 bis 7 oder relative Angaben wie -1 oder +3.
- `face`: die Schriftart.
- `color`: die Schriftfarbe. Enthält einen RGB-Farbcode wie in [SACM96] oder eine HTML-Farbe aus Anhang B.2.

Trennlinien

Trennlinien, wie man sie oft zum Abtrennen einzelner Dokumentteile verwendet, werden mit dem Element `HR` erzeugt. `HR` besteht lediglich aus einem Start-Tag, die Verwendung des End-Tags ist verboten. Zur Formatierung der Trennlinie können folgende Attribute verwendet werden:

- `width` legt die Länge der Linie fest. Erlaubte Werte sind absolute Pixelangaben oder aber relative Prozentangaben (beispielsweise 50%), die sich dann auf die Umgebungsbreite beziehen: im Rumpf auf die Dokumentbreite, in Tabellen auf die Zellenbreite.
- `noshade` ist ein boolesches Attribut, das dem Browser mitteilt, dass die Linie ohne den standardmäßigen 3D-Schatten gezeichnet werden soll.
- `size` gibt die Höhe der Linie in Pixeln an. Der Standardwert hängt vom Browser ab.
- `align` gibt die Ausrichtung der Linie an. Erlaubte Werte sind *left*, *right* und *center*.
- Weitere Attribute:
  - `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
  - `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
  - `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

3.11 Frames

Frames gestatten es Autoren von HTML-Dokumenten, mehrere Dokumente gleichzeitig in verschiedenen Teilfenstern darzustellen. Beispielsweise kann das Browserfenster so unterteilt werden, dass die Navigationsleiste ständig sichtbar bleibt, während man in einem anderen Teilfenster scrollbaren Textinhalt darstellt. Ein solches Teilfenster heißt *frame*.

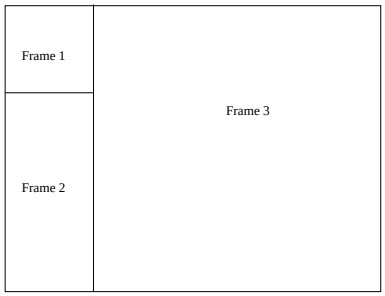

Abbildung 3.3: Geschachteltes dreiteiliges Framelayout

HTML-Dokumente, die ein Framelayout beschreiben, heißen *Frameset Dokumente*. Ihre Struktur unterscheidet sich von anderen HTML-Dokumenten dahingehend, dass sie statt des Dokumentrumpfes - also anstatt des Elements `BODY` (siehe Kapitel 3.3.3) - einen Frameset-Bereich haben, der vom Element `FRAMESET` gebildet wird. Dieses Element spezifiziert die geometrische Unterteilung des Browserfensters und dient als Behälter für den im Frame darzustellenden Inhalt. Optional kann ein sogenannter *noframe content* angegeben werden, der von Browsern angezeigt wird, die keine Frames darstellen können.

3.11.1 Das Element `FRAMESET`

Mit Hilfe des Elements `FRAMESET` wird das Browserfenster in rechteckige Teilfenster gegliedert. Die Abmessungen der einzelnen Teilfenster werden durch die Attribute `rows` und `cols` festgelegt, wobei `rows` für die horizontale und `cols` für die vertikale Untergliederung sorgt. Beide Attribute können zusammen verwendet werden, um ein Raster zu erzeugen. Die Werte der Attribute sind sogenannte *Multilängen-Listen*. Multilängen-Listen sind kommagetrennte Listen von Pixeln, Prozentzahlen oder relativen Längen. Eine relative Länge hat die Form i*, wobei i eine ganze Zahl ist. Bei der Aufteilung des verfügbaren Raums verarbeiten Browser zuerst die Pixel- und Prozentlängen zuerst, anschließend wird der verbleibende Raum auf die relativen Längen verteilt. Dabei erhält jede relative Länge den Bruchteil des Raums, der durch das i spezifiziert wird. Sollten beispielsweise 60 Pixel Raum auf die relativen Längen 1*, 2* und 3* aufgeteilt werden, so würde das einer Zuteilung von 10, 20 und 30 Pixeln entsprechen.

Der Standardwert der beiden Attribute ist 100%, was ein Frameset von einer Zeile und einer Spalte bedeutet. Die Erstellung von Frames erfolgt für Zeilen von oben nach unten und für Spalten von links nach rechts. Hier ein Stück HTML-Code, das ein Frameset für ein 2x3 Raster darstellt:

```
<FRAMESET rows='30%,70%' cols='33%,33%,34%'>
```

FRAMESET-Elemente dürfen bis zu einer beliebigen Tiefe ineinander geschachtelt werden, um einzelne Frames noch weiter zu untergliedern. Das Beispiel aus Abbildung 3.3 könnte beispielsweise durch folgende Schachtelung produziert werden:

```
<FRAMESET cols='200,600'>
    <FRAMESET rows='200,*'>
      <FRAME src='frame1.html'>
      <FRAME src='frame2.html'>
    </FRAMESET>
    <FRAME src='frame3.html'>
</FRAMESET>
```

Attribute:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- Die *scripting events* `onLoad` und `onUnLoad`, siehe Anhang B.1, Tabelle B.3.
- `style`: siehe Kapitel 3.9.
- `rows`: kommagetrennte Multilängenliste, die die horizontale Untergliederung spezifiziert.
- `cols`: kommagetrennte Multilängenliste, die die vertikale Untergliederung spezifiziert.

3.11.2 Das Element FRAME

Während das Element `FRAMESET` die Abmessungen des Frames spezifiziert, so definiert `FRAME` dessen Inhalt und Erscheinungsbild. Die dazu benötigten Attribute sind:

- `src` enthält eine URI, die auf den darzustellenden Inhalt des Frames verweist.
- `name` enthält den Namen des Frames, den wir bei Links auf dieses Frame benötigen, wie wir im nächsten Abschnitt noch genauer sehen werden.
- Das boolesche Attribut `noresize` verhindert, dass die Größe des Frames vom Benutzer geändert werden kann.
- `marginwidth` und `marginheight` legen fest, wieviel Pixel Raum der Browser zwischen dem Inhalt des Frames und seinen Rändern lassen soll.
- `scrolling` kann die Werte *no* (keine Scrollbalken anzeigen), *yes* (immer Scrollbalken anzeigen), und *auto* (Scrollbalken anzeigen wenn nötig) annehmen.
- `frameborder` veranlasst den Browser, den Framerand sichtbar darzustellen (Wert 1) oder ihn unsichtbar zu lassen (Wert 0).
- `noresize`: das Teilfenster kann durch den Benutzer nicht verkleinert oder vergrößert werden.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.

- `style`: siehe Kapitel 3.9.

Das Element `FRAME` besteht lediglich aus einem Start-Tag und darf nur innerhalb des Elements `FRAMESET` vorkommen.

3.11.3 Spezifikation von Zielframes

Um auf ein Frame Bezug nehmen zu können muss dieses durch Setzen des Attributs `name` benannt werden. Autoren, die auf ein bestimmtes Frame Bezug nehmen wollen, verwenden dazu das Attribut `target`, das für Links (Elemente `A`, `BASE`), image maps (`AREA`) und Formularen (`FORM`) definiert ist. Soll ein Link auf das Dokument *test.html* erstellt werden, dass in einem Frame mit Namen *MeinFrame* geöffnet werden soll, so führt das zu folgender Syntax:

```
<A HREF="test.html" target="MeinFrame">
```

Zusätzlich bietet HTML noch vier vordefinierte Zielframes, denen eine spezielle Bedeutung zukommt:

- *_blank*: das verlinkte Dokument wird in einen neuen, unbenannten Fenster geöffnet.
- *_self*: das verlinkte Dokument wird in demselben Fenster geöffnet, in dem sich das Element befindet, das auf dieses Dokument verweist (Eigenreferenz).
- *_parent*: das verlinkte Dokument soll in dem übergeordneten Frameset geöffnet werden. Wenn es kein übergeordnetes Frameset gibt, so ist dieser Wert gleichbedeutend mit *_self*.
- *_top*: das verlinkte Dokument soll im ganzen Fenster geöffnet werden, das Framelayout wird dadurch aufgelöst.

3.11.4 Das Element `NOFRAME`

`NOFRAME` wird dazu verwendet, einen alternativen Inhalt zu spezifizieren, der von Browsern dargestellt wird, die keine Frames darstellen können. `NOFRAME` hat Start- und End-Tag und darf nur innerhalb eines `FRAMESET`-Elements auftreten. Hier ein Codebeispiel:

```
<FRAMESET cols="50%, 50%">
  <FRAME src="main.html">
  <FRAME src="toc.html">
  <NOFRAMES>
    <P>Klicken Sie <A href="main_noframes.html">hier</A> für eine Version
       dieses Dokuments ohne Frames.
  </NOFRAMES>
</FRAMESET>
```

3.11.5 Das Element IFRAME

IFRAME dient zur Spezifikation von *inline frames*. Ein inline frame ist ein Frame, das nicht in einem eigenen Teilfenster dargestellt wird, sondern in einen Textabschnitt eingebunden wird. Analog zur Einbettung von Objekten in Dokumente (siehe Abschnitt 3.8) können diese inline frames ebenfalls an dem umgebenden Text ausgerichtet werden (siehe Abschnitt 3.10). IFRAME verfügt ebenfalls über das Attribut src, das eine URI enthält, die auf den entsprechenden Inhalt verweist. Der eigentliche Inhalt des IFRAME-Elements soll hingegen nur von Browsern dargestellt werden, die keine Frames darstellen können. IFRAME verfügt über dieselben Attribute wie FRAME bis auf noresize, da ihre Größe standardmäßig nicht verändert werden kann.

3.12 Formulare

Ein Formular ist ein Dokumentabschnitt, der gewöhnlichen Inhalt, spezielle Elemente - Kontrollelemente genannt - (bespielsweise Menüs, Radio Buttons oder Schalter) und dazugehörige Beschriftungen beinhaltet. Normalerweise vervollständigen Benutzer ein solches Formular durch Ausfüllen von Feldern und Auswählen von Optionen und leiten es dann an einen Agenten (Webserver, Mailserver etc.) weiter.

3.12.1 Kontrollelemente

Benutzer vervollständigen Formulare mittels der Kontrollelemente. Kontrollelemente haben einen Namen, der durch das Attribut name spezifiziert wird, einen *ursprünglichen Wert*, der durch das Attribut value spezifiziert wird, und einen *aktuellen Wert*. Der aktuelle Wert des Kontrollelements wird anfänglich auf den ursprünglichen Wert gesetzt. Wird ein Formular zurückgesetzt, so werden alle Kontrollelemente auf ihren ursprünglichen Wert zurückgesetzt.

Wie Kontrollelemente dargestellt werden, hängt zum größten Teil vom Browser ab. Dem Autor eines Formulars bieten sich nur wenige Möglichkeiten, auf die Darstellung Einfluss zu nehmen. Er kann lediglich die Reihenfolge der Kontrollelemente und ihre Ausrichtung innerhalb des Dokuments beeinflussen, grafische Details wie die Größe von Kontrollkästchen obliegen jedoch dem Browser.

HTML kennt die folgenden Typen von Kontrollelementen:

- **Buttons**.
  - *Submit Button*: diese Schaltfläche leitet nach Aktivierung das Formular weiter. Mehrere Submit Buttons pro Formular sind erlaubt.
  - *Reset Button*: setzt bei Aktivierung alle Kontrollelemente auf ihren ursprünglichen Wert zurück.
  - *Push Buttons*: Push Buttons führen nach Aktivierung ein client-seitiges Skript aus, das der Autor für diesen Fall vorgesehen hat.
- **Kontrollkästchen**. Kontrollkästchen haben zwei Zustände: ein und aus.

- **Optionsschalter**. Optionsschalter verhalten sich wie Schalter bzgl. ihrer Zustände. Allerdings kann immer nur ein Optionsschalter aus einer Menge von Optionsschaltern mit gleichem Namen aktiviert sein.
- **Menüs**. Menüs stellen dem Benutzer mehrere Möglichkeiten zur Verfügung, unter denen er auswählen kann.
- **Texteingabe**. Es gibt zwei Arten von Texteingabefeldern: einzeilige und mehrzeilige.
- **Dateifeld**. Dieses Kontrollelement erlaubt es dem Benutzer, eine Datei auszuwählen, die er mit dem Formular zusammen weiterleiten will.
- **Verborgene Felder**. Verborgene Felder werden vom Browser nicht dargestellt. Autoren können diese Kontrollelemente dazu verwenden, um Informationen zu speichern, die aufgrund der zustandslosen Natur von HTML ansonsten verloren gehen würden.

3.12.2 Das Element `FORM`

Das Element `FORM` dient als Behälter für Kontrollelemente. Es spezifiziert

- den Namen des Formulars durch das Attribut `name`, dessen Wert eindeutig innerhalb des aktuellen Dokumentes sein muss.
- das Programm, das die weitere Verarbeitung des weitergeleiteten Formulars übernimmt. Dazu enthält das Attribut `action` eine URI, die auf das entsprechende Programm verweist.
- die HTTP-Methode, die verwendet wird, um das Formular zu übertragen. Mögliche Werte sind dabei *get* und *post*. Wir werden auf diese Methoden in Abschnitt 3.12.9 noch näher eingehen.
- die Kodierung des Formularinhalts, wenn dieser an den Server weiterübermittelt wird. Dazu wird dem Attribut `enctype` ein MIME-Typ (siehe dazu [FB96a] und [FB96b]) als Wert zugewiesen.

`FORM` kann neben den Kontrollelementen auch einfachen Text oder andere HTML-Elemente beinhalten. Das Schachteln von Formularen ist nicht erlaubt.

Attribute:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `onsubmit`, `onreset`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `style`: siehe Kapitel 3.9.
- `action`: enthält eine URI, die auf den weiterverarbeitenden Agenten verweist.

- `method`: die Formularübertragungsmethode, erlaubte Werte sind *get* (Standard) und *post*.
- `enc-type`: enthält einen MIME-Typ wie in [FB96a], [FB96b], der die Kodierung der Daten spezifiziert, wenn diese zum Server übertragen werden.

3.12.3 Das Element `INPUT`

`INPUT` dient zur Erzeugung gleich mehrerer Arten von Kontrollelementen. Der darzustellende Typ wird im einzelnen durch den Wert des Attributs `type` bestimmt. Erlaubte Werte sind dabei:

- *text*: erzeugt einzeiliges Eingabefeld für Text.
- *password*: wie *text*, allerdings wird die Eingabe (jedoch nicht der eigentliche Wert) durch Sternchen ersetzt.
- *checkbox*: erzeugt ein Kontrollkästchen.
- *radio*: erzeugt einen Optionsschalter.
- *submit*: erzeugt einen Submit Button.
- *image*: erzeugt einen grafischen Submit Button. In diesem Fall enthält das Attribut `src` eine URI, die auf das darzustellende Bild verweist. Wird auf diese Bild geklickt, so wird das Formular und zusätzlich noch die Koordinaten an den Server übermittelt. Die Koordinaten werden von der linken oberen Ecke des Bildes in Pixeln berechnet und in der Form $name.x = x.Koordinate$ und $name.y = y.Koordinate$ übermittelt. Dabei steht *name* für den Wert des Attributs `name` des Kontrollelements. Bei *image* dürfen außerdem noch die Attribute `width` und `height` angegeben werden, die die Größe des Bildes festlegen.
- *reset*: erzeugt einen Reset Button.
- *button*: erzeugt einen Push Button. In diesem Fall wird der Wert des Attributs `value` als Beschriftung der Schaltfläche verwendet.
- *hidden*: erzeugt ein verborgenes Feld.
- *file*: erzeugt ein Dateifeld.

Desweiteren verfügt `INPUT` über die folgenden Attribute:

- `name`: der Name des Kontrollelementes, der eindeutig im aktuellen Element `FORM` sein muss.
- `value`: der ursprüngliche Wert.
- `size`: die Breite des Kontrollelementes in Zeichen, wenn das Kontrollelement vom Typ *password*, *text* oder *file* ist.
- `maxlength`: die maximale Eingabelänge für Kontrollelemente des Typs *password* oder *text*.

- `checked`: boolesches Attribut, dass bei Elementen des Typs *radio* und *checkbox* angibt, ob das Kästchen aktiviert ist.
- `src`: spezifiziert eine URI, unter der das darzustellende Bild bei Elementen des Typs *image* zu finden ist.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `align`: siehe Kapitel 3.10.
- `alt`: Alternativ darzustellender Text.
- `readonly`
- `usemap`: siehe Kapitel 3.8.3.
- `onfocus, onblur, onselect, onchange, onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

Die Verwendung des End-Tags ist bei `INPUT` verboten, daher kann ein Element dieses Typs auch keinen Inhalt haben.

Hier ein Beispiel für ein Formular, das aus verschiedenen Elementen des Typs `INPUT` zusammen gebaut ist:

```
<FORM action="http://somesite.com/prog/adduser" method="post">
  <P>
  Vorname: <INPUT type="text" name="vorname"><BR>
  Nachname: <INPUT type="text" name="nachname"><BR>
  eMail: <INPUT type="text" name="email"><BR>
  <INPUT type="radio" name="geschlecht" value="maennlich"> m&auml;nnlich<BR>
  <INPUT type="radio" name="sex" value="weiblich"> weiblich <BR>
  <INPUT type="submit" value="Abschicken"> <INPUT type="reset">
  </P>
</FORM>
```

Der Internet Explorer von Microsoft stellt dieses Formular folgendermaßen dar:

3.12.4 Das Element `BUTTON`

Das Element `BUTTON` verfügt ebenfalls über das Attribut `type`, kann aber nur drei Arten von Kontrollelementen darstellen, nämlich Submit, Reset und Push Buttons. Die entsprechenden Werte dafür sind *submit*, *reset* und *button*. Im Gegensatz zu `INPUT` muss bei `BUTTON` der End-Tag gesetzt werden. Die Funktionalität der Buttons ist bei beiden Elementen dieselbe, jedoch darf das `BUTTON`-Element andere Elemente und Text beinhalten und ist somit flexibler als `INPUT`. `BUTTON` verfügt ebenso wie `INPUT` über das Attribut `name`,

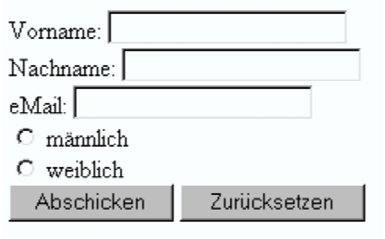

Abbildung 3.4: Formular im Internet Explorer

dem ein im Formular eindeutiger Name zugewiesen werden muss, und das Attribut `value`, dem der ursprüngliche Wert des Elementes zugewiesen wird.

Attribute:

- `name`: der Name des Kontrollelementes, der eindeutig im aktuellen Element `FORM` sein muss.
- `value`: der ursprüngliche Wert.
- `type`: die Art des Buttons, *submit*, *reset* oder *button*.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onfocus`, `onblur`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (siehe Anhang B.1, Tabelle B.3).

3.12.5 Menüs

Auswahlmenüs werden in HTML durch drei Elemente erzeugt. `SELECT` erzeugt das Menü und dient als Behälter für die beiden anderen Elemente. Jeder Eintrag im Menü wird durch ein Element `OPTION` repräsentiert. Mehrere `OPTION`-Elemente können mit Hilfe des Elements `OPTGROUP` zu logischen Blöcken zusammengefasst werden, die dem Benutzer die Navigation in großen Menüs erleichtern.

Das Element `SELECT`

Jedes `SELECT`-Element muss mindestens ein Element vom Typ `OPTION` enthalten. Sowohl Start-Tag als auch End-Tag müssen gesetzt werden. Außer den beiden Elementen `OPTION` und `OPTGROUP` sind keine anderen Elemente innerhalb von `SELECT` zulässig. `SELECT` verfügt über die folgenden Attribute:

- `name`: der Name des Menüs, der eindeutig im Formular sein muss.

- `multiple`: wenn dieses boolesche Attribut gesetzt ist, so kann der Benutzer mehrere Optionen aus dem Menü wählen. Ansonsten darf lediglich ein Eintrag ausgewählt werden.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

Das Element `OPTION`

`OPTION` erzeugt einen Eintrag im Menü. Jedes Menü muss mindestens einen Eintrag enthalten. Der Inhalt des Elementes wird dabei als Beschriftung des Menüpunktes verwendet. Mit Hilfe des Attributs `value` wird der Wert spezifiziert, der mit diesem Menüpunkt verbunden ist. Ferner kann man mit `label` eine alternative Kurzbeschreibung angeben, die von Browsern verwendet wird, wenn der Platz nicht ausreicht um die komplette Beschriftung darzustellen.

HTML verlangt, dass mindestens ein Menüpunkt vorgewählt ist. Um einen Menüpunkt vorzuwählen setzt der Autor das boolesche Attribut `selected` in dem betreffenden `OPTION`-Element. Es ist erlaubt, in Menüs mit mehrfacher Auswahl mehrere Optionen vorzuwählen. Wird kein Menüeintrag durch den Autor vorgewählt, so sollte der Browser gemäß [BLC95] das erste `OPTION`-Element in `SELECT` automatisch vorwählen.

Die Verwendung des End-Tags ist bei `OPTION` optional. Elemente dieses Typs dürfen nur innerhalb von Elementen des Typs `SELECT` oder `OPTGROUP` vorkommen.

Attribute:

- `selected`: wenn dieses boolesche Attribute gesetzt ist, dann ist der entsprechende Eintrag vorgewählt.
- `value`: der ursprüngliche Wert des Eintrags.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

Das Element `OPTGROUP`

Mehrere Menüeinträge können mit Hilfe des Elements `OPTGROUP` zu Gruppierungen zusammengefasst werden. Solche Optionsgruppen werden vom Browser dann beispielsweise

durch hierarchische Menüs oder hervorgehobenen Gruppierungsnamen dargestellt. Der Gruppierungsname ist dabei der Wert des Attributs `label`.

`OPTGROUP` darf nur innerhalb eines `SELECT`-Elements auftreten. Der End-Tag muss verwendet werden.

Attribute:

- `label`: Der Name der Gruppierung.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

Beispiel für ein Menü

Folgend ein Beispiel für ein Menu mit Optionsgruppierung und einer vorgewählten Option. Die Abbildung zeigt die Darstellung im Internet Explorer bei aufgeklapptem Menü.

```
<FORM action="http://somesite.com/prog/someprog" method="post">
  <SELECT name="ComOS">
    <OPTION selected value="none">None</OPTION>
    <OPTGROUP label="PortMaster 3">
      <OPTION value="pm3_3.7.1">
        PortMaster 3 with ComOS 3.7.1
      </OPTION>
      <OPTION value="pm3_3.7">
        PortMaster 3 with ComOS 3.7
      </OPTION>
      <OPTION value="pm3_3.5">
        PortMaster 3 with ComOS 3.5
      </OPTION>
    </OPTGROUP>
    <OPTGROUP label="PortMaster 2">
      <OPTION value="pm2_3.7">
        PortMaster 2 with ComOS 3.7
      </OPTION>
      <OPTION value="pm2_3.5">
        PortMaster 2 with ComOS 3.5
      </OPTION>
    </OPTGROUP>
    <OPTGROUP label="IRX">
      <OPTION value="IRX_3.7R">
        IRX with ComOS 3.7R
      </OPTION>
```

```
      <OPTION value="IRX_3.5R">
        IRX with ComOS 3.5R
      </OPTION>
    </OPTGROUP>
  </SELECT>
</FORM>
```

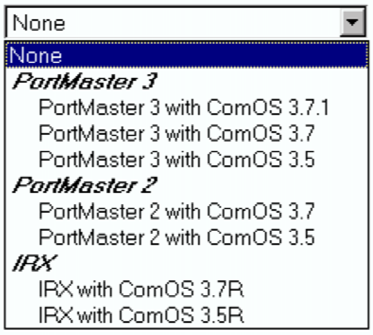

Abbildung 3.5: Drop-down Menü im Internet Explorer

3.12.6 Das Element `TEXTAREA`

Mehrzeilige Texteingabefelder werden mit Hilfe des Attributs `TEXTAREA` erzeugt. Dabei spezifizieren die Werte der beiden Attribute `cols` und `rows` die Anzahl der Spalten und Zeilen in Zeichen, die das Feld groß sein soll. Browser sollten es allerdings ermöglichen, dass Benutzer längere Texte eingeben können, indem sie die Zeilen automatisch umbrechen und gegebenenfalls einen Scrollbalken zur Verfügung stellen. Auch bei diesem Element ist der ursprüngliche Wert, der vom Browser als Text im Feld dargestellt wird, der Wert des Attributs `value`. Möchte der Formularautor den Inhalt dieses Feldes unveränderbar machen, so kann er das boolesche Attribut `readonly` setzen.

Attribute:

- `name`: der Name des Menüs, der eindeutig im Formular sein muss.
- `rows`: Höhe in Zeilen.
- `cols`: Breite in Spalten.
- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onfocus`, `onblur`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (siehe Anhang B.1, Tabelle B.3).

3.12.7 Beschriftungen

Einige Kontrollelemente in Formularen haben bereits implizit eine Beschriftung, beispielsweise Schaltflächen. Verwendet man das Element `INPUT` zur Erzeugung von Schaltflächen, so spezifiziert der Wert des Attributs `value` diese Beschriftung. Bei der Verwendung von `BUTTON` wird der Inhalt des Elements zur Beschriftung herangezogen.

Alle Elemente, die nicht über implizite Beschriftungen verfügen, können mit Hilfe des Elements `LABEL` beschriftet werden. `LABEL` dient dabei als Behälter für ein Kontrollelement und die dazugehörige Beschriftung. Ein Codebeispiel dafür:

```
<LABEL>
  Vorname:
  <INPUT type="text" name="vorname">
</LABEL>
<LABEL>
  Nachname:
  <INPUT type="text" name="nachname">
</LABEL>
```

Die Verwendung des End-Tags ist bei `LABEL` verpflichtend.

Attribute:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onfocus`, `onblur`, `onclick`, `ondblclick`, `onmousedown`, `onmouseup`, `onmouseover`, `onmousemove`, `onmouseout`, `onkeypress`, `onkeydown`, `onkeyup` (siehe Anhang B.1, Tabelle B.3).

3.12.8 Strukturierung von Formularen

Große Formulare sind aufgrund der Vielzahl von Kontrollelementen oft schlecht überschaubar. Es empfiehlt sich daher, solche Formulare zu strukturieren. HTML stellt dafür die beiden Elemente `FIELDSET` und `LEGEND` zur Verfügung.

Mit Hilfe von `FIELDSET` können mehrere Kontrollelemente thematisch gruppiert werden. Dazu dient `FIELDSET` als Behälter für diese Elemente. Eine solche Gruppierung kann dann noch durch `LEGEND` beschriftet werden, wobei der Inhalt des Elements als Beschriftung dient.

Elemente des Typs `FIELDSET` dürfen nicht geschachtelt werden. Bei beiden Elementen müssen die End-Tags verwendet werden.

Attribute von `FIELDSET`:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.

- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).

Attribute von `LEGEND`:

- `id`, `class`: Kernattribute, siehe Anhang B.1, Tabelle B.1.
- `lang`: Sprachinformation, siehe Anhang B.1, Tabelle B.2.
- `style`: siehe Kapitel 3.9.
- `onclick, ondblclick, onmousedown, onmouseup, onmouseover, onmousemove, onmouseout, onkeypress, onkeydown, onkeyup` (siehe Anhang B.1, Tabelle B.3).
- `align`: die Position der Beschriftung bezüglich der Gruppierung; erlaubte Werte sind *top, bottom, left, right.*

3.12.9 Übertragung von Formularen

Übertragungsmethoden

Das Attribut `method` des Elements `FORM` spezifiziert die Art und Weise, wie der Formularinhalt an den bezeichneten Agenten übertragen wird. Es gibt dafür zwei Möglichkeiten:

- Der Wert *get* bedeutet, dass der Formularinhalt an die durch `action` spezifizierten URI getrennt durch ein ? angehängt wird und diese neue URI an den weiterverarbeitenden Agenten gesendet wird.
- *post* bedeutet, dass der Formularinhalt im Formularrumpf beinhaltet ist und so an den weiterverarbeitenden Agenten gesendet wird.

Wird die Methode *get* verwendet, so sind lediglich ASCII-Werte erlaubt. Die Methode *post* ermöglicht hingegen die Verwendung des gesamten ISO10646 Zeichensatzes [ISO93].

Erfolgreiche Kontrollelemente

Ein erfolgreiches Kontrollelement ist per Definition ein Kontrollelement, dass bei der Weiterleitung des Formularinhaltes mit übertragen werden soll. Für jedes erfolgreiche Kontrollelement wird dafür ein Paar mit dem Namen und dem aktuellen Wert des Kontrollelements gebildet. Für erfolgreiche Kontrollelemente gelten folgende Regeln:

- Enthält ein Formular mehrere Submit Buttons, so ist lediglich der angeklickte ein erfolgreiches Kontrollelement.
- Alle aktivierten Kontrollkästchen sind erfolgreich.

- Bei mehreren Optionsschaltern, die denselben Wert in `name` haben, ist nur der aktivierte Schalter erfolgreich.
- Alle ausgewählten Einträge in einem Menü sind erfolgreich. Für jeden ausgewählten Eintrag wird ein Paar aus dem Wert von `name` des `SELECT`-Elements und dem Wert von `value` des betreffenden `OPTION`- Elements gebildet. Ist kein Eintrag ausgewählt worden, so ist das gesamt `SELECT`-Element nicht erfolgreich.
- Ist der aktuelle Wert eines Dateiauswahlfeldes nicht leer, so ist es erfolgreich.
- Verborgene Felder, die einen Wert haben, sind erfolgreich.
- Felder, die keinen aktuellen Wert haben, sind nicht erfolgreich.
- Reset Buttons sind nicht erfolgreich.

Formularverarbeitung

Bei der Verarbeitung von Formularen gehen Benutzeragenten folgendermaßen vor:

1. Bestimme alle erfolgreichen Kontrollelemente.
2. Bilde für alle erfolgreichen Kontrollelemente die entsprechenden Paare (*Name*, *aktueller Wert*). Die Menge aller dieser Paare heißt *Menge der Formulardaten* (*form data set*).
3. Kodiere die Menge der Formulardaten entsprechend dem Wert des Attributs `enctype` des Elements `FORM`.
4. Versende die kodierte Menge der Formulardaten. Ist die spezifizierte Methode *get*, dann werden Dateien als Pfadnamen versendet. Ist die Methode *post*, so werden die ausgewählten Dateien selbst gesendet.

3.13 Skripte

Ein client-seitiges Skript ist ein Programm, das mit einem HTML-Dokument assoziiert ist und auf dem Rechner des Benutzers ausgeführt wird. Der Quelltext des Skripts kann dabei in das HTML-Dokument selbst eingebettet sein oder in einer externen Datei vorliegen. HTML unterstützt Skripte unabhängig von der Programmiersprache, in der sie geschrieben wurden. Der Browser, mit dem der Benutzer das Dokument betrachtet, muss jedoch über eine *scripting engine* für die Sprache verfügen, in der das Programm geschrieben ist.

3.13.1 Das Element `SCRIPT`

Mit Hilfe des Elements `SCRIPT` werden Skripte mit einem HTML-Dokument assoziiert. `SCRIPT` darf beliebig oft im Kopf oder Rumpf eines HTML-Dokumentes vorkommen, das heißt es dürfen beliebig viele Skripte mit einem Dokument assoziiert werden.

Ein Autor, der ein externes Skript verwenden möchte, verwendet das Attribut `src` von `SCRIPT`. `src` hat eine URI als Wert, die auf die Skriptdatei verweist, welche vom Browser

dann geladen wird. Wird das Attribut `src` nicht verwendet, so wird der Inhalt des Elements `SCRIPT` als Skript interpretiert. Für externe Stylesheets muss zusätzlich noch der verwendete Zeichensatz gemäß [Cha02] als Wert des Attributs `charset` festgelegt werden.

Attribute:

- `charset`: der Zeichensatz gemäß [Cha02], in dem externe Skripts hinterlegt sind.
- `src`: die URI, die auf ein externes Skript verweist.
- `type`: der MIME-Typ gemäß [FB96a] und [FB96b], der die Skriptsprache spezifiziert.
- `defer`: boolesches Attribut; weist daraufhin, dass das Skript keinen Dokumentinhalt generiert, das heißt der Browser kann mit der Visualisierung fortfahren.

3.13.2 Skriptsprache

Es gibt zwei Möglichkeiten, die Skriptsprache festzulegen. Autoren können im Kopf mittels eines `META`-Elementes spezifizieren, was sie als Standardskriptsprache bevorzugen. Der Beispielcode für JavaScript sieht folgendermaßen aus:

```
<META http-equiv='Content-Script-Type' content='text/javascript'>
```

Der zugewiesene Wert ist dabei ein MIME-Typ, der eine Skriptsprache repräsentiert (siehe [FB96a] und [FB96b]).

Die zweite Möglichkeit besteht darin, dem Attribut `type` des Elements `SCRIPT` einen MIME-Typ zuzuweisen. Ist dieses Attribut gesetzt, so kommt ihm höhere Priorität zu, als der Deklaration der Standardskriptsprache.

3.13.3 Das Ausführen von Skripten

Skripte, die über einen Anweisungsteil verfügen, werden direkt beim Laden des Dokumentes ausgeführt. Es gibt jedoch die Möglichkeit, den Zeitpunkt der Ausführung an sogenannten Ereignissen oder *scripting events* festzumachen. Ein Ereignis ist dabei ein Paar (*Element*, *Aktion*). Ein Ereignis wird in HTML durch ein Attribut im betreffenden Element realisiert, wobei der Name des Attributs durch die auslösende Aktion bestimmt wird. Für eine vollständige Listung aller auslösenden Aktionen siehe Tabelle B.3. Ein Ergebnis tritt ein, wenn die spezifizierte Aktion am betreffenden Element ausgelöst wird. Das kann Beispielsweise das Klicken auf ein Bild oder das Ausfüllen eines Textfeldes sein.

Die Reaktion auf ein Ereignis wird durch den Wert des dazugehörigen Attributs festgelegt. Der Wert dieses Attributs ist ein Skriptquelltext selbst, wobei als Sprache die Standardskriptsprache aus dem Dokumentkopf angenommen wird, oder der Verweis auf eine Prozedur bzw. Funktion, die in einem mit dem Dokument assoziierten Skript deklariert ist. Tritt das Ereignis ein, so führt der Browser das so spezifizierte Skript aus, beispielsweise die Überprüfung, ob ein Formulareintrag gültig ist. Das folgende Codebeispiel stellt einen Text in der Statuszeile des Browsers da, sobald der Benutzer den Mauszeiger über das Bild bewegt, und löscht diesen wieder, sobald sich der Mauszeiger nicht mehr über dem Bild befindet:

```
<IMG src="pics/meinbild.gif" onMouseOver="window.status='Mein Text'"
     onMouseOut="window.status=''">
```

Das Referenzieren von HTML-Elementen ist abhängig von der Skriptsprache und wird nicht durch HTML spezifiziert. Die meisten Sprachen wie JavaScript oder VBScript definieren dazu ein *Document Object Model* (DOM), das eine hierarchische Struktur für alle Elemente eines Dokuments definiert. Einzelne Elemente können durch das Attribut `id` (siehe Anhang B.1, Tabelle B.1) benannt werden und dann anhand dieses Namens referenziert werden.

Kapitel 4

Java

4.1 Einführung

Java wurde von Sun Microsystems, Inc. [Sun] 1995 als eine *"einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, hochleistungsfähige, multithread-fähige und dynamische Sprache"* [Fla98] der Öffentlichkeit präsentiert. Die Idee von Java ist es, dieselbe Software auf vielen verschiedenen Arten von Computern und elektronischen Geräten zum Laufen zu bringen. Daher ist Java eine interpretierte Sprache. Der Java-Compiler erzeugt keine direkten Sequenzen von Maschinencode sondern sogenannte *Byte-Codes*. Diese Byte-Codes werden von der Java Virtual Machine (JVM) interpretiert. Da diese Java Byte-Codes maschinenunabhängig sind, können Java Programme auf jedem Sytem ausgeführt werden, für welches die JVM portiert worden ist.

Wir haben dieser Arbeit die zweite Ausgabe der *Java Language Specification* zugrunde gelegt [GJSB00].

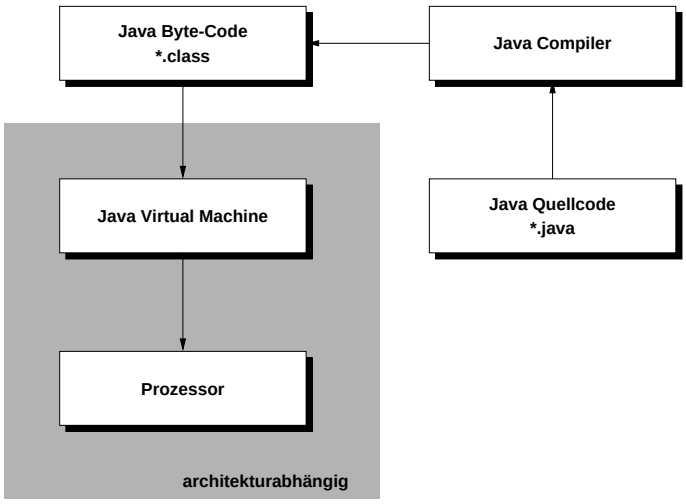

Abbildung 4.1: Java Virtual Machine

4.2 Lexikalische Struktur

Java Programme werden in Unicode [Con00] geschrieben. Es werden auch die Unicodezeichen unterstützt, die sich nicht auf der Tastatur befinden. Solche Zeichen werden durch sogenannte *Escape-Sequenzen* dargestellt. Unicode Escape-Sequenzen haben die Form $\backslash uxxxx$, wobei $xxxx$ der hexadezimale Code des entsprechenden Unicodezeichens ist. Java unterscheidet zwischen *Leerzeichen*, *Kommentaren* und *Tokens*. Tokens sind *Bezeichner*, *Schlüsselwörter*, *Literale*, *Trennzeichen* oder *Operatoren*.

4.2.1 Trennzeichen

Eine Folge von Unicodezeichen wird anhand von *Zeilenenden* in Zeilen zerlegt. Um die unterschiedlichen Betriebssystem unterstützen zu können, erkennt Java die folgenden Zeilenenden:

- das ASCII-Zeichen LF (line feed)
- das ASCII-Zeichen CR (carriage return)
- das ASCII-Zeichen CR gefolgt von LF.

4.2.2 Leerzeichen

Die folgenden Zeichen plus Zeilenenden werden in Java als Leerzeichen betrachtet:

- das ASCII-Zeichen SP (space)
- das ASCII-Zeichen HT (horizontal tab)
- das ASCII-Zeichen FF (form feed)

4.2.3 Kommentare

In Java existieren zwei Arten von Kommentaren:

1. *Traditionelle Kommentare* beginnen mit der Zeichenfolge /* und enden mit */. Alle Zeichen zwischen diesen beiden Zeichenfolgen und sie selbst werden ignoriert.
2. *Zeilenkommentare* beginnen mit der Zeichenfolge //. Alle Zeichen ab dieser Folge bis zum nächsten Zeilenende werden ignoriert.

4.2.4 Bezeichner

Ein Bezeichner ist eine in der Länge unbeschränkte Folge von Javazeichen und -ziffern. Javazeichen sind alle ASCII-Zeichen vom a-z, von A-Z, der Unterstrich (_) und das Dollarzeichen ($). Javaziffern sind alle ASCII-Zeichen von 0-9. Bezeichner müssen mit einem Javazeichen beginnen und müssen von Schlüsselwörtern, den booleschen Literalen und dem null Literal verschieden sein.

4.2.5 Schlüsselwörter

Folgende Wörter sind als Schlüsselwörter reserviert und dürfen nicht als Bezeichner verwendet werden:

`abstract`	`default`	`if`	`private`	`this`
`boolean`	`do`	`implements`	`protected`	`throw`
`break`	`double`	`import`	`public`	`throws`
`byte`	`else`	`instanceof`	`return`	`transient`
`case`	`extends`	`int`	`short`	`try`
`catch`	`final`	`interface`	`static`	`void`
`char`	`finally`	`long`	`strictfp`	`volatile`
`class`	`float`	`native`	`super`	`while`
`const`	`for`	`new`	`switch`	
`continue`	`goto`	`package`	`synchronized`	

Tabelle 4.1: Java Schlüsselwörter

4.2.6 Literale

Ein Literal ist die Quelltextrepräsentation eines Wertes

- eines primitiven Typs.
- vom Typ `String`.
- vom Typ `null`.

Integerliterale

Integerliterale können hexadezimal, dezimal oder oktal dargestellt werden. Wird dem Literal das ASCII-Zeichen `L` oder `l` hinten angestellt, so ist das Integerliteral vom Typ `long`, ansonsten vom Typ `int`. Integerliterale vom Typ `int` sind 32-bit groß, Integerliterale vom Typ `long` sind 64-bit groß.

Dezimale Integerliterale sind entweder 0 oder bestehen aus einer ASCII-Ziffer zwischen 1 und 9, optional gefolgt von einer Folge von ASCII-Ziffern zwischen 0 und 9. Negative dezimale Integerliterale werden mit Hilfe des unären Negationsoperators - erzeugt (siehe dazu Anhang C, Tabelle C.1).

Hexadezimale Integerliterale beginnen mit den ASCII-Zeichen `0x` oder `0X`, gefolgt von einer oder mehreren hexadezimalen Ziffern. Hexadezimale Integerliterale können negative und positive Integerwerte sowie die 0 repräsentieren.

Oktale Integerliterale beginnen mit der ASCII-Ziffer 0, gefolgt von einer oder mehreren Ziffern zwischen 0 und 7. Oktale Integerliterale können ebenfalls negative und positive Integerwerte sowie die 0 darstellen.

Fließkommaliterale

Ein Fließkommaliteral besteht aus einem ganzzahligen Teil, einem Dezimalpunkt (das ASCII-Zeichen für .), einem Nachkommateil, einem Exponent und einem Typsuffix. Gültige Typsuffixe sind F bzw. f für den Datentyp `float` und D bzw d für den Datentyp `double`. Exponenten beginnen mit dem Zeichen E bzw. e.

Boolesche Literale

Der Typ `boolean` hat zwei Werte, die durch die Zeichenfolgen `true` und `false` dargestellt werden.

Zeichenliterale

Ein Zeichenliteral ist entweder ein einzelnes Zeichen oder eine Escapesequenz für ein Unicode-Zeichen, jeweils in einfache Hochkommata gefasst. Im Gegensatz zu C und C++ repräsentiert ein Zeichenliteral in Java immer genau ein Zeichen. Zeichenliterale sind immer vom Typ `char`.

Stringliterale

Ein Stringliteral besteht aus 0 oder mehr Zeichen in doppelten Anführungszeichen. Jedes Zeichen kann alternativ als Escapesequenz dargestellt werden. Zeilenumbrüche müssen mit Hilfe der Sequenz `\n` realisiert werden. Stringliterale sind immer vom Typ `String`. Jedes Stringliteral ist eine Referenz auf eine einzige Instanz der Klasse `String`. Wir werden auf Referenzen in Kapitel 4.3.2, auf Klassen in Kapitel 4.6 noch näher eingehen.

Das Nullliteral

Der Typ `null` hat genau einen Wert, die Nullreferenz, die durch das Literal `null` repräsentiert wird.

4.2.7 Trennzeichen

Es gibt neun Trennzeichen in Java:

( ) { } [ ] ; , .

4.2.8 Operatoren

Java kennt insgesamt 37 verschiedene Operatoren. Eine vollständige Listung dieser findet sich in Anhang C.

4.3 Typen und Werte

Man unterscheidet in Java zwei grundlegende Typen, *primitive Typen* und *Referenztypen*. Ferner gibt es den speziellen Nulltyp, der keinen Namen hat.

4.3.1 Primitive Typen und deren Werte

Primitive Typen sind in Java bereits vordefiniert. Jeder primitive Typ ist durch ein reserviertes Schlüsselwort benannt. Man unterscheidet folgende primitiven Typen:

- `boolean`
- numerische Typen
  - integrale Typen
    - `byte` (8-bit)
    - `short` (16-bit)
    - `char` (16-bit)
    - `int` (32-bit)
    - `long` (64-bit)
  - Fließkommatypen
    - `float` (32-bit)
    - `double` (64-bit)

Die Werte der integralen Typen sind dabei vorzeichenbehaftete Integerzahlen im 2er-Komplement. Die Ausnahme bildet `char`, dessen Werte vorzeichenfrei sind und Unicode-Zeichen repräsentieren [Con00].

Die Werte der Fließkommatypen bilden dabei die Menge der IEEE 754 32-bit bzw. 64-bit Fließkommazahlen [IEE85]. Der spezielle Wert *Not A Number* wird durch die Konstante `NaN` repräsentiert. Ferner sind die Werte plus Unendlich und minus Unendlich durch die Konstanten `POSITIVE_INFINITY` und `NEGATIVE_INFINITY` definiert.

4.3.2 Referenztypen und deren Werte

Java unterscheidet drei grundlegende Referenztypen: *Klassen*, *Interfaces* und *Arrays*. Ein *Objekt* ist eine Instanz einer Klasse oder eines Arrays. Der Wert eines Referenztyps ist eine *Referenz* (d.h. die Speicheradresse) auf ein Objekt dieses Referenztyps oder die Nullreferenz `null`. Der Begriff Referenz wurde dabei absichtlich zur Unterscheidung von Pointern in C/C++ gewählt. Während C/C++ Pointerarithmetik zulässt, ist das Konzept *"Adresse eines Objektes"* aus Java vollständig entfernt worden, um die Sprache sicherer und architekturunabhängig zu machen. Die tiefer liegenden Prozeduren der Dereferenzierung und Speicherarithmetik werden komplett durch die Java Virtual Machine erledigt.

4.4 Namen

Namen werden zur Referenzierung von Entitäten verwendet, die in einem Programm deklariert sind. Deklarationen führen eine Entität ein und vergeben dabei einen Bezeichner (siehe Kapitel 4.2.4) an sie. Java unterstützt das dynamische Laden von Modulen über das Internet. Aus diesem Grund wurde besonderer Wert auf die Konfliktvermeidung im Namensraum gelegt. Java kennt daher keine globalen Variablen oder Funktionen bzw. Prozeduren. Die Entwickler der Programmiersprache haben eine Konvention zur Namensvergabe vorgeschlagen, die Konflikte vermeiden soll. Siehe dazu [GJSB00], Kapitel 6.8.

4.4.1 Einfache und qualifizierte Namen

Java kennt zwei Arten von Namen: *einfache* und *qualifizierte* (auch *vollständig qualifizierte* genannt). Ein einfacher Name besteht aus einem einzelnen Bezeichner. Ein vollständig qualifizierter Name besteht aus einer Folge von mindestens zwei Bezeichnern, getrennt durch einen Punkt (.):

Name:
EinfName
QualiName

EinfName:
Bezeichner

QualiName:
Bezeichner.Bezeichner
Bezeichner.QualiName

4.4.2 Gültigkeitsbereich

Der *Gültigkeitsbereich* einer Deklaration ist in Java als der Bereich definiert, in dem eine Entität durch Verwendung des einfachen Namens referenziert werden kann. Man sagt, eine Entität ist in diesem Bereich *gültig*.

4.5 Packages

Ein *Package* stellt die oberste Hierarchieebene in der Codestruktur von Java dar. Java-Programme setzen sich aus einem oder mehreren Packages zusammen. Ein Package setzt sich aus Klassen- und Interfacedeklarationen (auch *Typdeklarationen* genannt) zusammen, die sich in den *Compiliereinheiten* eines Packages befinden. Ferner kann ein Package ein oder mehrere *Subpackages* haben.

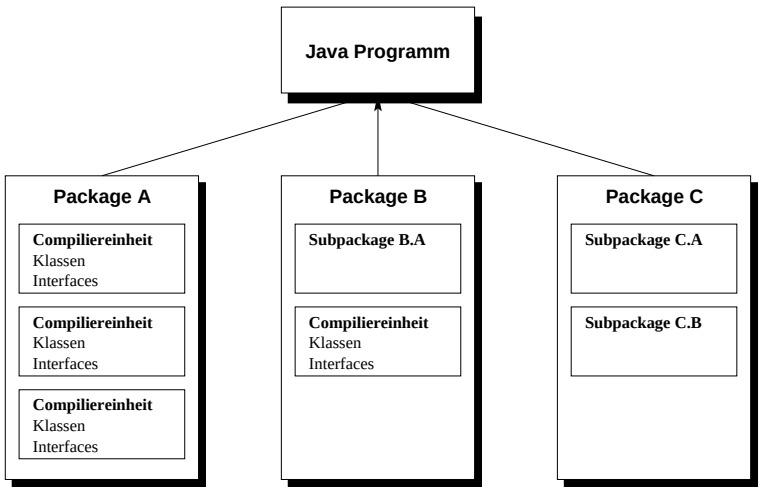

Abbildung 4.2: Beispiel für die Struktur eines Javaprogramms

4.5.1 Compiliereinheiten

Eine Compiliereinheit repräsentiert eine Quelltextdatei und wird durch folgende Produktion definiert:

Compiliereinheit:
$PackageDeklaration_{opt}$
$ImportDeklarationen_{opt}$
$TypDeklarationen_{opt}$

Alle drei Bestandteile einer Compiliereinheit sind optional.

Packagedeklaration

Eine Packagedeklaration hat folgende Syntax:

`package` *Packagename*;

Dabei ist *Packagename* der vollständig qualifizierte Name des Packages, zu dem die Compiliereinheit gehört. Ein Package mit vollständig qualifiziertem Namen A ist implizit definiert als Menge aller Subpackages von A und aller Quelltextdateien, die eine Packagedeklaration der Form `package` A`;` spezifizieren.

Importdeklaration

Eine Importdeklaration hat folgende Syntax:

`import` *Typname*;

oder

`import` *Packagename*`.*;`

Dabei ist *Typname* der vollständig qualifizierte Name einer Klasse oder eines Interfaces. *Packagename* ist der vollständig qualifizierte Name eines Packages. Die erste der beiden Möglichkeiten heißt *explizite Importdeklaration*, die zweite *Importdeklaration nach Bedarf.*

Importieren bedeutet in Java, dass anstelle des vollständig qualifizierten Namens der einfache Name des importierten Typs innerhalb der betreffenden Compiliereinheit verwendet werden darf. Bei einer expliziten Importdeklaration wird genau der Typ importiert, der durch *Typname* spezifiziert wird. Eine Importdeklaration nach Bedarf importiert alle Typen des Packages *Packagename*, die in der betreffenden Compiliereinheit verwendet werden.

Für explizite Importdeklarationen gilt:

- Importiert man einen Typ, der denselben einfachen Namen hat, wie eine Klasse oder ein Interface in der obersten Hierarchieebene der betreffenden Compiliereinheit, so führt das zu einem Fehler.
- Importiert man zwei Typen, die denselben einfachen Namen haben, so führt das zu einem Compilezeitfehler, es sei denn, die beiden Typen sind identisch.

Für Importdeklarationen nach Bedarf gilt:

- Importiert man zwei Packages mit Namen A und B und beide deklarieren einen Typ mit einfachem Namen x, so muss anstelle des einfachen Namens weiterhin der vollständig qualifizierte Name des Typs, nämlich $A.x$ bzw. $B.x$ verwendet werden.
- Importiert man ein Package mit Namen A, das einen Typ x deklariert und wird in der aktuellen Compiliereinheit ein Typ mit Namen x deklariert, so referenziert der einfache Name x den in der Compiliereinheit deklarierten Typ. Man kann mit $A.x$ weiterhin auf den importierten Typ zugreifen.

Das Package `java.lang.*` wird vom Compiler automatisch immer importiert.

Typdeklarationen

Der dritte Teil eines Packages enthält die *Typdeklarationen.* Eine Typdeklaration ist entweder eine Klassendeklaration oder eine Interfacedeklaration. Ein Typ, der direkt in einem Package deklariert wird, heißt *top-level Typ.*

4.5.2 Sub- und Superpackages

Sei A der vollständig qualifizierte Name eines Packages und x ein beliebiger gültiger Bezeichner. Dann heißt das Package mit dem Namen $A.x$ *Subpackage von* A. A heißt dann *Superpackage von* $A.x$. Ein Package A mit Subpackage $A.x$ darf keine Typdeklarationen mit einfachem Namen x enthalten. Ein Beispiel dafür: `java.awt` hat ein Subpackage `java.awt.image` und darf deshalb keinen Typ mit Namen `image` deklarieren.

4.5.3 Elemente eines Packages

Die top-level Typdeklarationen und die Subpackages eines Packages bilden die *Elemente eines Packages.*

4.5.4 Unbenannte Packages

Eine Compiliereinheit ohne Packagedeklaration gehört zu einem *unbenannten Package.* Üblicherweise gehören alle Compiliereinheiten ohne Packagedeklaration im selben Verzeichnis zum selben unbenannten Package. Diese Regel ist jedoch nicht durch [GJSB00] abgedeckt, sondern wird durch das Hostsystem bestimmt.

4.6 Klassen und Interfaces

Eine *Klasse* ist eine Menge von Variablen-, Methoden-, Konstruktor-, Klassen- und Interfacedeklarationen.

Ein *Interface* ist eine Menge von Konstanten-, Methoden-, Klassen- und Interfacedeklarationen.

4.6.1 Klassendeklaration

Eine *Klassendeklaration* hat folgende Syntax:

$$Modifikatoren_{opt}\ \texttt{class}\ Klassenname\ Super_{opt}\ Interfaces_{opt}\ Klassenrumpf$$

Modifikatoren

$Modifikatoren_{opt}$ ist eine Folge von einem oder mehreren Schlüsselwörtern aus der Menge

$$Sichtbarkeitsmodifikatoren \cup Klassenmodifikatoren$$
$$Sichtbarkeitsmodifikatoren := \{\texttt{public}, \texttt{private}, \texttt{protected}\}$$
$$Klassenmodifikatoren := \{\texttt{static}, \texttt{final}\}$$

Es darf dabei höchstens ein Element aus $Sichtbarkeitsmodifikatoren$ verwendet werden.

Klassenname

$Klassenname$ ist ein gültiger Bezeichner, der den einfachen Namen der deklarierten Klasse spezifiziert.

Sub- und Superklassen

$Super_{opt}$ spezifiziert die *direkte Superklasse* der Klasse $Klassenname$ und hat folgende Syntax:

`extends` *Superklassenname*;

Superklassenname ist dabei der Name der *direkten Superklasse* von *Klassenname*. Eine Klasse B heißt *direkte Subklasse von* A wenn A direkte Superklasse von B ist. *Subklasse* und *Superklasse* sind die irreflexiv-transitiven Hüllen der Relationen *direkte Subklasse* und *direkte Superklasse*.

Eine Ausnahme bildet die in Java vordefinierte Klasse `java.lang.Object`, die keine Superklasse hat. Fehlt der (optionale) Teil $Super_{opt}$, so ist `java.lang.Object` implizit die direkte Superklasse der betreffenden Klasse *Klassenname*.

Superinterfaces

$Interfaces_{opt}$ definiert die *direkten Superinterfaces* von *Klassenname* und hat folgende Syntax:

`implements` *Interfaceliste*;

Interfaceliste ist dabei ein einzelner Name eines Interfaces oder eine durch Kommata getrennte Liste von Interfacenamen. Alle Interfaces aus *Interfaceliste* heißen *direkte Superinterfaces* von *Klassenname*. *Superinterface* ist die irreflexiv-transitive Hülle der Relation *direktes Superinterface*. Eine Klasse B *implementiert* ein Interface A genau dann, wenn A Superinterface von B ist.

Klassenrumpf

Der *Klassenrumpf* wird definiert durch folgende Syntax:

$$\{Elementdeklarationen_{opt}\ Konstruktordeklarationen_{opt}\}$$

$Elementdeklarationen_{opt}$ ist dabei eine Folge von

- *Variablendeklarationen*
- *Methodendeklarationen*
- *Klassendeklarationen*
- *Interfacedeklarationen*

Finale Klassen

Eine *finale Klasse* wird durch das Schlüsselwort `final` deklariert. Finale Klassen können nicht als Superklassen für andere Klassen verwendet werden.

Statische Klassen

Eine *statische Klasse* wird durch das Schlüsselwort `static` deklariert. Statische Klassen können nicht instantiiert werden (vgl. dazu Kapitel 4.9.3).

4.6.2 Interfacedeklaration

Eine *Interfacedeklaration* hat folgende Syntax:

$$Modifikatoren_{opt}\ \texttt{interface}\ Interfacename\ Interfaces_{opt}\ Interfacerumpf$$

Modifikatoren

$Modifikatoren_{opt}$ ist ein Schlüsselwort aus der Menge der Sichtbarkeitsmodifikatoren {`public, private, protected`}.

Interfacename

$Interfacename$ ist ein gültiger Bezeichner, der den einfachen Namen des deklarierten Interfaces spezifiziert.

Sub- und Superinterfaces

$Interfaces_{opt}$ spezifiziert die *direkten Superinterfaces* des Interfaces $Interfacename$ und hat folgende Syntax:

$$\texttt{extends}\ Interfaceliste;$$

$Interfaceliste$ ist dabei ein einzelner Name eines Interfaces oder eine durch Kommata getrennte Liste von Interfacenamen. Alle Interfaces aus $Interfaceliste$ heißen *direkte Superinterfaces* von $Interfacename$. Ein Interface B heißt *direktes Subinterface von* A, wenn A direktes Superinterface von B ist. $Subinterface$ bzw. $Superinterface$ sind die irreflexiv-transitiven Hüllen der Relationn *direktes Subinterface* bzw. *direktes Superinterface*.

Interfacerumpf

In Interfaces dürfen keine Konstruktoren deklariert werden, sondern nur Elemente. Ferner dürfen keine Variablen deklariert werden, lediglich Compilezeitkonstanten sind erlaubt.

Der $Interfacerumpf$ hat folgende Syntax:

$$\{Elementdeklarationen_{opt}\}$$

$Elementdeklarationen_{opt}$ ist dabei eine Folge von

- $Konstantantendeklarationen$
- $Methodendeklarationen$
- $Klassendeklarationen$
- $Interfacedeklarationen$

4.6.3 Variablendeklaration in Klassen

Eine *Variablendeklaration* innerhalb einer Klasse hat folgende Syntax:

$$Modifikatoren_{opt}\ Typ\ Namen$$

$Modifikatoren_{opt}$ ist dabei ein Element aus der Menge

$$Sichtbarkeitsmodifikatoren \cup \{\texttt{static}, \texttt{final}\}$$

Dabei darf höchstens ein Element aus der Menge *Sichtbarkeitsmodifikatoren* verwendet werden. *Typ* spezifiziert den Typ der Variablen und ist entweder

- ein Schlüsselwort, dass einen primitiven Datentyp kennzeichnet,
- ein einfacher Name für einen in diesem Bereich gültigen Referenztyp,
- ein vollständig qualifizierter Name für einen in diesem Bereich nicht gültigen Referenztyp.

Namen ist entweder ein einzelner Bezeichner oder eine Folge von Bezeichnern, die durch Kommata getrennt sind. Eine Deklaration mit mehreren Bezeichner heißt *Mehrfachdeklaration.*

Klassen- und Instanzvariablen

Eine Variable, die mit dem Schlüsselwort `static` deklariert worden ist, heißt *Klassenvariable*, ansonsten spricht man von *Instanzvariablen.* Eine Klassenvariable existiert genau ein einziges Mal, egal wie viele Instanzen einer Klasse existieren. Eine Instanzvariable existiert für jede Instanz der betreffenden Klasse.

Finale Variablen

Eine Variable, die mit dem Schlüsselwort `final` deklariert wird, heißt *finale Variable.* Einer finalen Variablen darf lediglich einmal ein Wert zugewiesen werden. Jede weitere Zuweisung führt zu einem Compilezeitfehler. Der Compiler überprüft dabei nicht, ob die Anweisung möglicherweise in einer Schleife steht, die gar nicht durchlaufen wird.

4.6.4 Konstantendeklaration in einem Interface

Eine Konstantendeklaration in einem Interface hat folgende Syntax:

$$Typ\ Name = Wert;$$

Für *Typ* und *Name* gelten dieselben Regeln wie für Variablen in Klassen (Kapitel 4.6.3). *Wert* muss ein für den Typ der Konstante zulässiger Wert sein (vgl. Kapitel 4.3) und zur Compilezeit konstant sein

4.6.5 Methodendeklaration in Klassen

Eine *Methode* ist eine Funktion oder Prozedur, die innerhalb einer Klasse durch eine *Methodendeklaration* definiert wird. Eine Methodendeklaration besteht aus einem *Methodenkopf* und einem *Methodenrumpf*

Methodenkopf

Ein *Methodenkopf* hat folgende Syntax:

$$Modifikatoren_{opt}\ Rückgabetyp\ Name\ (Parameter_{opt})$$

$Modifikatoren_{opt}$ ist dabei ein Element aus der Menge

$$Sichtbarkeitsmodifikatoren \cup \{\texttt{static}, \texttt{final}\}$$

$Name$ ist ein Bezeichner, der den Namen der Methode spezifiziert. *Rückgabetyp* ist entweder ein primitiver Typ, ein Referenztyp oder das Schlüsselwort `void`, wenn die Methode keinen Wert zurückgibt.

Methodenparameter

$Parameter_{opt}$ spezifiziert entweder keine, einen oder mehrere Parameter, die durch Kommata getrennt sind. Ein Parameter hat die Form

$$\texttt{final}_{opt}\ Typ\ Name$$

Einem Parameter, der als `final` deklariert worden ist, darf im Methodenrumpf kein Wert mehr zugewiesen werden. Typ spezifiziert den Typ der Variablen durch ein Schlüsselwort (primitive Typen), den einfachen Namen (gültiger Referenztyp) oder den vollständig qualifizierten Namen (Referenztyp sonst). Für $Name$ gilt:

- wird ein Array deklariert, so ist $Name$ ein Bezeichner gefolgt von einer beliebigen Menge von Paaren eckiger Klammern []. Jedes Paar steht dabei für eine Arraydimension.
- ansonsten ist $Name$ ein Bezeichner.

Es dürfen keine zwei Parameter mit demselben Namen deklariert werden.

Signatur einer Methode

Zwei Methoden haben dieselbe Signatur S, wenn sie in Namen, Parameteranzahl und -typen übereinstimmen.

Methodenrumpf

Ein Methodenrumpf ist besteht aus einer Folge von lokalen Variablendeklarationen und Anweisungen oder einem Semikolon. Der Rumpf wird eingefasst von geschweiften Klammern.

Lokale Variablen

Lokale Variablen werden wie Variablen in Klassen deklariert, allerdings sind keine Sichtbarkeitsmodifikatoren zulässig. Es gibt zwei vordefinierte lokale Variablen, `this` und `super`. Sie enthalten Referenzen auf das aktuelle Objekt, in dem die Methode aufgerufen wird, bzw. auf die direkte Superklasse des aktuellen Objekts. Lokale Variablen sind innerhalb der Methode sichtbar und gültig, in der sie deklariert werden.

Klassen- und Instanzmethoden

Eine Methode, die mit dem Schlüsselwort `static` deklariert worden ist, heißt *Klassenmethode*, ansonsten spricht man von *Instanzmethoden*. Eine Klassenmethode kann lediglich auf Klassenvariablen zugreifen, während Instanzmethoden auch auf Instanzvariablen zugreifen können.

4.6.6 Methodendeklaration in Interfaces

Für Methodendeklarationen in Interfaces gelten im wesentlichen dieselben Regeln wie für Klassen. Im folgenden werden wir nur die Unterschiede diskutieren.

Methodenkopf

Im Gegensatz zu Methoden in Klassen haben Interfacemethoden keine Modifikatoren. Die Syntax sieht daher folgendermaßen aus:

$$Rückgabetyp\ Name\ (Parameter_{opt})$$

Methodenrumpf

Der Rumpf einer Methode in einem Interface besteht aus einem Semikolon:

;

4.6.7 Konstruktordeklarationen

Ein *Konstruktor* ist eine spezielle Methode, die zur Initialisierung von Instanzen einer Klasse verwendet wird. Die Deklaration eines Konstruktors in einem Interface ist verboten, da Interfaces nicht instantiiert werden. Wie eine Methode besteht ein Konstruktor aus einem *Konstruktorkopf* und einem *Konstruktorrumpf*.

Konstruktorkopf

Ein Konstruktorkopf hat folgende Syntax:

$$Sichtbarkeitsmodifikatoren_{opt}\ Name\ (Parameter_{opt})$$

Es darf höchstens ein Element aus der Menge $Sichtbarkeitsmodifikatoren$ verwendet werden. $Name$ ist der einfache Name der Klasse, in der der Konstruktor definiert wird. Für die Parameter eines Konstruktors gelten dieselben Regeln wie für Methodenparameter (siehe Kapitel 4.6.5).

Signatur eines Konstruktors

Zwei Konstruktoren in der gleichen Klassen haben dieselbe Signatur S, wenn sie in Parameteranzahl und -typen übereinstimmen.

Konstruktorrumpf

Ein Konstruktorrumpf unterscheidet sich lediglich darin von einem Methodenrumpf, dass er als erste Anweisung einen alternativen Konstruktoraufruf spezifizieren darf:

- entweder wird ein Konstruktor mit verschiedener Signatur in derselben Klasse aufgerufen:
  `this`$(Parameter_{opt})$;
- oder ein alternativer Konstruktor in der direkten Superklasse:
  `super`$(Parameter_{opt})$;

Da alle Konstruktoren denselben einfachen Namen haben, wie die Klasse, in der sie deklariert sind, dürfen `this` und `super` anstelle der expliziten Konstruktornamen verwendet werden.

4.6.8 Klassenarten und Namen

Top-level Klassen

Eine Klasse mit Namen C, die direkt in einem Package P deklariert wird, heißt *top-level Klasse*. Ist P der vollständig qualifizierte Name des Packages, zu dem die Klasse gehört, dann ist $P.C$ der vollständig qualifizierte Name der Klasse. Ist die Klasse in einem unbenannten Package deklariert, dann hat C keinen vollständig qualifizierten Namen.

Top-level Interfaces

Ein Interface mit Namen I, das direkt in einem Package P deklariert wird, heißt *top-level Interface*. Ist P der vollständig qualifizierte Name des Packages, zu dem das Interface gehört, dann ist $P.I$ der vollständig qualifizierte Name des Interfaces. Ist das Interface

in einem unbenannten Package deklariert, dann hat I keinen vollständig qualifizierten Namen.

Geschachtelte Klassen

Eine Klasse, die keine top-level Klasse ist, heißt *geschachtelte Klasse*. Eine geschachtelte Klasse mit Namen C in einer Klasse oder einem Interface mit vollständig qualifiziertem Namen A hat $A.C$ als vollständig qualifizierten Namen. Hat die umgebende Klasse bzw. das umgebende Interface keinen vollständig qualifizierten Namen, so hat auch C keinen vollständig qualifizierten Namen.

Geschachtelte Interfaces

Ein Interface, das kein top-level Interface ist, heißt *geschachteltes Interface*. Ein geschachteltes Inteface mit Namen I in einer Klasse oder eines Interfaces mit vollständig qualifiziertem Namen C hat $C.I$ als vollständig qualifizierten Namen. Hat die umgebende Klasse bzw. das umgebende Interface keinen vollständig qualifizierten Namen, so hat auch I keinen vollständig qualifizierten Namen.

Klassenmethoden und -variablen

Sei M der Name einer Klassenmethode/Klassenvariable in einer Klasse mit vollständig qualifiziertem Namen C. Dann ist $C.M$ der vollständig qualifizierte Name dieser Klassenmethode/Klassenvariable.

Konstruktoren

Konstruktoren haben dieselben vollständigen Namen wie die Klassen, in denen sie deklariert sind.

4.6.9 Kapselung

Kapselung heißt die Eigenschaft von Java, Zugriff auf Klassen/Interfaces und die darin deklarierten Entitäten zu steuern. Die unterschiedlichen Zugriffsberechtigungen werden durch die *Sichtbarkeitsmodifikatoren* `public`, `protected`, `private` geregelt. Ist einer Entität A der Zugriff auf eine Entität B erlaubt, so sagt man *B ist für A erreichbar.*

Packages

Packages sind immer für alle Entitäten erreichbar.

Top-level Typen

Top-level Typen dürfen nur mit dem Zugriffsmodifikator `public` oder ohne Zugriffsmodifikator deklariert werden. Ein top-level Typ, der `public` ist, kann von jeder anderen Entität erreicht werden. Ansonsten kann der top-level Typ nur innerhalb des Packages erreicht werden, in dem er deklariert ist.

Entitäten in Klassen

Für die Erreichbarkeit von in Klassen deklarierten Entitäten gilt:

Erreichbar für Entitäten aus	public	protected	kein Modifikator	private
gleicher Klasse	ja	ja	ja	ja
gleiches Package	ja	ja	ja	nein
Subklasse, anderes Package	ja	ja	nein	nein
keine Subklasse, anderes Package	ja	nein	nein	nein

Tabelle 4.2: Erreichbarkeit in Klassen

Entitäten in Interfaces

Alle in einem Interface deklarierten Entitäten haben implizit den Modifikator `public`. Es gelten dieselben Regeln für Erreichbarkeit wie in der Spalte `public` in oben stehender Tabelle.

4.6.10 Vererbung und Polymorphie

Variablen in Klassen

Eine Variable V heißt *verborgen* für eine direkte Subklasse C, wenn

1. V für C erreichbar ist und
2. C selbst eine Variable mit Namen V deklariert.

Der Typ von V spielt dabei keine Rolle. Eine Klasse erbt alle Variablen ihrer direkten Superklasse und ggf. alle Konstanten ihrer direkten Superinterfaces, die erreichbar und nicht verborgen sind.

Konstanten in Interfaces

Eine Konstante K heißt *verborgen* für ein direktes Subinterface I, wenn

1. K für I erreichbar ist und
2. I selbst eine Konstante mit Namen K deklariert.

Der Typ von K spielt dabei keine Rolle. Ein Interface erbt alle Konstanten seiner direkten Superinterfaces die nicht verborgen sind.

Methoden in Klassen

Eine Instanzmethode M_1 der Klasse C *überschreibt* eine Methode M_2 einer Klasse A mit $Signatur(M_1) = Signatur(M_2)$ genau dann, wenn

1. C ist direkte Subklasse von A und
2. M_2 ist für C erreichbar.

Eine Klassenmethode M_1 der Klasse C *verbirgt* eine Methode M_2 einer Klasse A mit $Signatur(M_1) = Signatur(M_2)$ genau dann, wenn

1. C ist direkte Subklasse von A und
2. M_2 ist für C erreichbar.

Verbirgt eine Klassenmethode eine Instanzmethode bzw. überschreibt eine Instanzmethode eine Klassenmethode, so führt das zu einem Compilezeitfehler. Überschreibt oder überlädt M_1 die Methode M_2 und hat M_2 einen anderen Rückgabetyp als M_1 oder M_2 ist als `final` deklariert, dann ist das ebenfalls ein Fehler.

Eine Klasse erbt alle Methoden ihrer direkten Superklasse, die erreichbar und nicht *verborgen* oder *überschrieben sind.*

Eine Klasse C mit direktem Superinterface I muss für alle Methoden $M_1, \ldots, M_n$ aus I eine Deklaration mit gleicher Signatur bereitsstellen. Der Rumpf dieser Methode muss mindestens aus einem Semikolon bestehen.

Methoden in Interfaces

Eine Methode M_1 des Interfaces I *überschreibt* eine Methode M_2 eines Interfaces J mit $Signatur(M_1) = Signatur(M_2)$ genau dann, wenn

1. I ist direktes Subinterface von J und
2. M_2 ist für I erreichbar.

Ein Interface erbt alle Methoden seiner direkten Superinterfaces, die nicht *überschrieben sind.*

Überladen von Methoden und Konstruktoren

Haben zwei Methoden/Konstruktoren einer Klasse - wobei es egal ist, ob die Methoden geerbt oder in der Klasse deklariert sind - dieselben Namen, aber ansonsten verschiedene Signaturen, so sagt man, der Methodenname/Kosntruktorname ist *überladen.* Der

Rückgabetyp spielt dabei im Falle von Methoden keine Rolle. Beim Aufruf eines überladenen Methodennamens/Konstruktornamens wird zur Laufzeit anhand der Parameter überprüft, welche Implementierung die passende ist. Anschließend wird der betreffende Methodenrumpf/Konstruktorrumpf ausgeführt.

Oben genannte Regeln gelten ebenfalls für Methoden in Interfaces.

4.6.11 Klassenelemente

Die *Elemente einer Klasse* sind

- alle geerbten und nicht verborgenen Variablen,
- alle geerbten und nicht verborgenen oder überschriebenen Methoden sowie
- die in ihrem Rumpf deklarierten Variablen, Methoden, Klassen und Interfaces.

4.6.12 Interfaceelemente

Die *Elemente eines Interfaces* sind

- alle geerbten und nicht verborgenen Konstanten,
- alle geerbten und nicht überschriebenen Methoden sowie
- die in ihrem Rumpf deklarierten Konstanten, Methoden, Klassen und Interfaces.

4.7 Arrays

Ein Array ist eine Datenstruktur, die eine gewisse Anzahl von Variablen desselben Typs enthält. Die Variablen eines Arrayas heißen *Komponenten.* Komponenten haben keinen Namen, sie werden über einen *Index* referenziert. Ein *Index* ist eine nicht-negative Integerzahl. Ein einfaches Array mit n Komponenten hat die Indizes 0 bis $n - 1$.

4.7.1 Arraytypen

Der *Typ* eines Arrays ist der Typ seiner Komponenten gefolgt von beliebig vielen Paaren eckiger Klammern []. Ein Paar dieser Klammern steht dabei für eine *Dimension.* Arrays mit mehr als einer Dimension heißen *multidimensionale Arrays.* Jede Dimension hat dabei einen eigenen Index.

4.7.2 Deklaration eines Arrays

Ein Array wird dadurch deklariert, dass man eine Variable mit einem Arraytyp deklariert. Es gibt zwei äquivalente syntaktische Möglichkeiten zur Arraydeklaration:

1. *Typ Dimension Name*; bzw.

2. *Typ Name Dimension*;

Typ ist dabei der Typ der Arraykomponenten, *Dimension* eine beliebige Anzahl von Paaren eckiger Klammern und *Name* ein gültiger Javabezeichner.

4.8 Ausdrücke

Ein *Ausdruck* besteht aus Variablen, Literalen, Operatoren und Methodenaufrufen, die zu einem einzelnen Wert ausgewertet werden können. Ausdrücke dürfen rekursiv aufgebaut sein.

Variablen können dabei Variablen in Klassen und Interfaces oder lokale Variablen sein. Literale wurden ausführlich in Kapitel 4.2.6 besprochen. Da Methodenaufrufe auch Anweisungen sind, werden wir diese im nächsten Kapitel behandeln. Eine vollständige Listung aller Operatoren in Java findet sich im Anhang C.

4.9 Anweisungen und Blöcke

Java kennt drei Arten von Anweisungen: *Deklarationsanweisungen*, *Kontrollflussanweisungen* und *Ausdrucksanweisungen*. Deklarationsanweisungen deklarieren Variablen. Eine Kontrollflussanweisung bestimmt, wann und welche Anweisungen ausgeführt werden. Im Gegensatz zu C und JavaScript sind in Java nicht alle Ausdrücke automatisch auch Anweisungen. Die Ausdrücke, die als Anweisungen verwendet werden können, bilden die Menge der *Ausdrucksanweisungen*.

4.9.1 Blöcke

Ein *Block* ist eine Folge von null oder mehr Anweisungen in geschweiften Klammern. Handelt es sich um eine einzelne Anweisung, so dürfen die geschweiften Klammern auch weggelassen werden.

Ein Block definiert ebenfalls den Gültigkeitsbereich einer lokalen Variablen (vgl. Kapitel 4.4 und 4.9.4).

4.9.2 Leere Anweisung

Die *leere Anweisung* hat keinerlei Auswirkungen und besteht lediglich aus einem Semikolon (;).

4.9.3 Ausdrucksanweisungen

Methodenaufruf

Ein *Methodenaufruf* kann sich auf eine Instanzmethode oder eine Klassenmethode beziehen. Handelt es sich um eine Klassenmethode, so hat der Aufruf folgende Syntax:

$$Name(Argumentliste);$$

Dabei ist $Name$ entweder der vollständig qualifizierte Name oder der einfache Name der Klassenmethode, wenn diese an der betreffenden Programmstelle gültig ist.

Handelt es sich um eine Instanzmethode, so hat der Aufruf folgende Syntax:

$$Objektreferenz.Name(Argumentliste);$$

Dabei ist $Objektreferenz$ eine Referenz auf ein Objekt im Speicher, in dem die Methode mit dem Namen $Name$ aufgerufen werden soll.

$Argumentliste$ ist in beiden Fällen eine Folge von Argumenten, die durch Kommata getrennt sind und mit den Parametern der Methode konsistent sind.

Ein Methodenaufruf ist sowohl Anweisung als auch Ausdruck.

Objekterzeugung

Ein Objekt wird durch den Operator `new` erzeugt. Bei der Syntax wird unterschieden, ob es sich um eine Klasse handelt oder um ein Array, das instantiiert werden soll. Handelt es sich um eine Klasse, so sieht die Syntax folgendermaßen aus:

`new` $Konstruktorname$ `(`$Argumentliste$`);`

$Konstruktor$ ist dabei entweder der vollständig qualifizierte Namen des Konstruktors oder der einfache Name, wenn der Konstruktor an der betreffenden Programmstelle gültig ist. Handelt es sich um ein Array, so sieht die Syntax so aus:

`new` $Komponententyp$ $Dimensionen$;

$Dimensionen$ ist dabei eine Folge von Paaren eckiger Klammern [$size$], wobei $size$ ein Ausdruck ist, der die Größe der betreffenden Dimension definiert.

Zuweisungen

Zuweisungen erfolgen mittels des Operators `=`:

$Variable$ `=` $Ausdruck$`;`

$Variable$ ist dabei entweder der vollständig qualifizierte Name der Variable (wenn es sich um eine Klassenvariable handelt) eine Objektreferenz gefolgt von einem Punkt und dem Bezeichner der Variablen: $Objektreferenz.Variablenname$ oder eine Arrayvariable gefolgt von einem $Indizierungsausdruck$. Ein $Indizierungsausdruck$ besteht aus Anzahl der Dimensionen vielen Paaren eckiger Klammern, die jeweils einen Index für die zu referierende Komponente spezifizieren. Ein Beispiel dafür:

```
myArray[13][j+i] = 4711;
```

Post- und Präfixanweisungen

Postfix- und Präfixausdrücke sind gleichzeitig auch Anweisungen. Ein Präfixausdruck ist ein Ausdruck der Form

$$PraefixOp\ Ausdruck$$

Erlaubte Präfixoperatoren sind `++`, `--`, `!`, `~`, `+`, `-`.

Ein Präfixausdruck ist ein Ausdruck der Form

$$Ausdruck\ PostfixOp$$

Erlaubte Postfixoperatoren sind `++` und `--`.

Für eine Listung aller Operatoren und ihrer Bedeutungen siehe auch Anhang C.1.

4.9.4 Deklarationsanweisungen

Eine Deklarationsanweisung deklariert eine lokale Variable. Vergleiche dazu Kapitel 4.6.5.

4.9.5 Kontrollflussanweisungen

Schleifenanweisungen

Java kennt folgende Schleifenanweisungen:

- `while`,
- `do...while`,
- `for`.

Da für sie dieselben Regeln gelten wie für die entsprechenden Anweisungen in JavaScript (vgl. Kapitel 2), werden wir hier nicht weiter auf sie eingehen.

Bedingte Anweisungen

Java kennt folgende bedingte Anweisungen:

- `if...else`,
- `switch...case`.

Da für sie dieselben Regeln gelten wie für die entsprechenden Anweisungen in JavaScript (vgl. Kapitel 2), werden wir hier nicht weiter auf sie eingehen.

Benannte Anweisungen

Anweisungen können in Java benannt werden. Dies geschieht durch das Voranstellen eines Labels:

Label: *Amnweisung*

Label ist dabei ein Java-Bezeichner.

`break` und `continue`

`break` kann in zwei Formen verwendet werden:

`break;`
`break` *Label*`;`

Im ersteren Fall wird die Programmausführung mit der ersten Anweisung hinter dem derzeitigen Block fortgesetzt, im zweiten Fall wir die Programmausführung mit der benannten Anweisung fortgesetzt.

`continue` kann ebenfalls in zwei Formen vorkommen:

`continue;`
`continue` *Label*`;`

`continue` leitet die nächste Iteration einer Schleifenanweisung ein. Wird `continue` mit einem Label verwendet, so wird der neue Schleifendurchlauf mit der bezeichneten Anweisung begonnen.

`return`

`return` wird dazu verwendet, eine Methode zu verlassen. Hat die Methode einen Rückgabetyp verschieden von `void`, so hat `return` die Syntax

`return` *Ausdruck*`;`

, wobei der Wert von *Ausdruck* den in der Methodendeklaration spezifizierten Typ haben muss. Ist die Methode mit Rückgabetyp `void` deklariert, dann lautet die Syntax

`return;`

Anhang A

JavaScript

A.1 JavaScript-Operatoren

P	A	Operator	Operandentyp(en)	Durchgeführte Operation
0	L	.	Objekt, Eigenschaft	Zugriff auf Eigenschaft
	L	[]	Array, Ganzzahl	Arrayindizierung
	L	()	Funktion, Parameter	Funktionsaufruf
1	R	++	Zahl	Prä-/Postinkrement (unär)
	R	−−	Zahl	Prä-/Postdekrement (unär)
	R	−	Zahl	unäres Minus (Negation)
	R	~	Ganzzahl	bitweises Komplement (unär)
	R	!	boolescher Wert	Logisches Komplement (unär)
	R	typeof	beliebig	Rückgabe des Datentyps als String (unär)
	R	new	Konstruktoraufruf	Erzeugen eines neuen Objektes (unär)
	R	void	beliebig	Rückgabe des undefinierten Werts (unär)
2	L	*, /, %	Zahlen	Multiplikation, Division, Rest
3	L	+, −	Zahlen	Addition, Subtraktion
	L	+	Strings	Stringverkettung
4	L	<<	Ganzzahlen	Shift nach links
	L	>>	Ganzzahlen	Shift nach rechts mit Vorzeichenerweiterung
	L	>>>	Ganzzahlen	Shift nach rechts mit 0-Auffüllung
5	L	<, <=	Zahlen oder Strings	kleiner als, kleiner oder gleich
	L	>, >=	Zahlen oder Strings	größer als, größer oder gleich
6	L	==	elementare Typen	gleich (haben identische Werte)
	L	!=	elementare Typen	ungleich (haben verschiedene Werte)
	L	==	Referenztypen	gleich (beziehen sich auf dasselbe Objekt)
	L	!=	Referenztypen	ungleich (beziehen sich auf unterschiedliche Objekte)
7	L	&	Ganzzahlen	bitweises UND
8	L	^	Ganzzahlen	bitweises XOR
9	L	|	Ganzzahlen	bitweises ODER
10	L	&&	boolesche Werte	logisches UND
11	L	||	boolesche Werte	logisches ODER
12	R	? :	boolescher Wert, beliebig, beliebig	Konditionaloperator (tenär)
13	R	=	Variable, beliebig	Zuweisung
	R	*=, /=, %=, +=, −=, <<=, >>=, >>>=, &=, ^=, |=	Variable, beliebig	Zuweisung mit Operatoren
14	L	,	beliebig	mehrfache Berechnung

Tabelle A.1: JavaScript-Operatoren

A.2 Vordefinierte Funktionen im JavaScript-Kern

Funktionsname	Syntax	Beschreibung
`eval`	`eval` (*Argument*)	Wenn das Argument ein Ausdruck ist, so wertet ihn `eval` aus, wenn es eine Anweisung oder eine Folge von Anweisungen ist, so führt `eval` sie aus.
`isFinite`	`isFinite` (*Number*)	Gibt `false` zurück, wenn das Argument `NaN`, positiv unendlich oder negativ unendlich ist, ansonsten `true`.
`isNaN`	`isNaN` (*Number*)	`parseInt` und `parseFloat` liefern `NaN`, wenn der zu parsende String sich nicht in eine Zahl umwandeln ließ. Die Funktion `isNaN` kann dann zum Test dafür verwendet werden.
`parseInt`	`parseInt` (*String [, Basis]*)	Wandelt einen String in eine ganze Zahl um und gibt diese zurück. Der optionale Parameter *Basis* gibt die Basis des Zahlensystems an (z.B. 10 für dezimal, F für hexadezimal, 8 für oktal).
`parseFloat`	`parseFloat` (*String*)	Wandelt einen String in eine Fließkommazahl um und gibt diese zurück.
`Number`	`Number` (*Objektreferenz*)	Wandelt ein Objekt in eine Zahl um.
`String`	`String` (*Objektreferenz*)	Wandelt ein Objekt in einen String um.

Tabelle A.2: Vordefinierte Funktionen im JavaScript-Kern

A.3 Vordefinierte Objekte im JavaScript-Kern

A.3.1 Object

`Object` ist der grundlegende Objekttyp in JavaScript, von dem alle anderen Objekte abgeleitet sind. Das bedeutet, dass alle Objekte in JavaScript (auch die anderen vordefinierten Objekte) zumindest die Methoden und Eigenschaften besitzen, die auch `Object` hat. `Object` wird durch den Konstruktoraufruf `new Object()` explizit erzeugt und ist außerdem die Grundlage aller benutzerdefinierter Objekte (siehe dazu auch Kapitel 2.10.3). Eigenschaften in `Object`:

Eigenschaftsname	Beschreibung
`constructor`	Enthält eine Referenz auf die Funktion, mittels derer das Objekt erzeugt worden ist. Die Konstruktorfunktion setzt diesen Wert bei der Instantiierung. Standardwert ist eine Referenz auf `Object`.
`prototype`	Enthält eine Referenz auf das Prototypenobjekt.

Interessante Methoden in `Object`:

Methodenname	Beschreibung
`toString`	Gibt einen `String` zurück, der das entsprechende Objekt repräsentiert. Diese Funktion sollte in benutzerdefinierten Objekten nach Bedarf überladen werden, standardmäßig gibt sie [`object` *Objekttyp*] zurück, wobei *Objekttyp* der Name der Konstruktorfunktion ist.
`valueOf`	Gibt einen primitiven Wert zurück, der das Objekt repräsentiert. Im Normalfall wird diese Methode bei Bedarf automatisch vom JavaScript-Interpreter aufgerufen. Sie ist für den Programmierer nur dann interessant, wenn er sie in benutzerdefinierten Objekten mit einer eigenen Methode überladen will.

A.3.2 Array

JavaScript hat keinen primitiven Array-Datentyp, stellt jedoch das vordefinierte Objekt `Array` für diesen Zweck zur Verfügung. Die Syntax dafür sieht folgendermaßen aus:

Arrayname = `new Array(`*Element 0,...,Element n*`)` oder alternativ:
Arrayname = `new Array(`*Arraylänge*`)`

Der Zugriff auf das i-te Element erfolgt durch den Ausdruck *Arrayname*`[i]`. Jedes Objekt vom Typ `Array` hat die Eigenschaft `length`, die die Anzahl der Elemente im Array angibt.

Einige wichtige Methoden des Objekts `Array` sind:

- `concat`: Verbindet zwei Arrays und gibt ein neues Array zurück.

- `joins`: Verbindet alle Elemente eines Arrays zu einem String und gibt diesen zurück.
- `reverse`: Sortiert das Array in umgekehrter Reihenfolge.
- `sort`: Sortiert das Array in alphabetischer Reihenfolge

A.3.3 Date

Das Objekt `Date` stellt dem JavaScript-Programmierer ein komfortables Werkzeug für kalendarische Daten zur Verfügung. JavaScript speichert Daten intern als Anzahl von Millisekunden seit dem 01.01.1970, 00:00:00 Uhr bis zu dem gegebenen Datum. Die Syntax für das Erzeugen eines neuen `Date`-Objekts sieht folgendermaßen aus:

Datumsname `= new Date(`*[Parameter]*`)`
Der Parameter kann dabei entweder ein String in der Form 'Monat Tag, Jahr Stunden:Minuten:Sekunde' sein oder eine Folge von ganzzahligen Werten in der Form 'Jahr, Monat, Tag' bzw. 'Jahr, Monat, Tag, Stunde, Minute, Sekunde'. Wird kein Parameter übergegeben, so wird das aktuelle Datum und Uhrzeit herangezogen.

A.3.4 Math

Das Objekt `Math` stellt Methoden und Eigenschaften für mathematische Funktionen und Konstanten zur Verfügung. So hat beispielsweise die Eigenschaft `Math.PI` den Wert von π. Die folgende Tabelle listet die Methoden auf, die im Objekt `Math` definiert sind:

Methode	**Beschreibung**
`abs`	Absolutbetrag
`sin, cos, tan`	Die üblichen trigonometrischen Funktionen; das Argument muss im Bogenmaß angegeben werden.
`acos, asin, atan`	Die Umkehrfunktionen zu den trigonometrischen Funktionen; das Argument muss im Bogenmaß angegeben werden.
`exp, log`	Exponentialfunktion und natürlicher Logarithmus, Basis e.
`ceil, floor`	Geben den kleinsten (größten) ganzzahligen Wert zurück, der größer (kleiner) oder gleich dem Argument ist.
`round`	Rundet das Argument auf die nächste ganze Zahl (kaufmännisches Runden).
`min, max`	Geben den kleineren (größeren) Wert von zwei Argumenten zurück.
`pow`	Potenzrechnung; das erste Argument gibt die Basis an, das zweite den Exponenten.
`sqrt`	Quadratwurzel

Tabelle A.3: Methoden in `Math`

Anhang B

HTML

B.1 Gemeinsame Attribute

Attributname	Beschreibung
`ID`	Bezeichnet ein Element eindeutig innerhalb eines Dokuments. Der Attributwert muss mit einem Buchstaben (A-Z, a-z) beginnen und kann dann mit anderen Buchstaben, Ziffern oder einem der Zeichen :,.,_,- fortgesetzt werden.
`CLASS`	Kennzeichnet ein Element als zu einer Klasse zugehörig. Klassen können dazu benutzt werden, um verschiedene Untergruppen zu einem Element zu erzeugen.

Tabelle B.1: HTML-Kernattribute

Attributname	Beschreibung
`LANG`	Spezifiziert die Sprache der Attributswerte eines Elements und aller eingeschlossenen Elemente, sofern sie keine eigenes `LANG` Attribut haben. Der Wert dieses Attributs ist ein Länderkürzel gemäß [Alv95].
`DIR`	Gibt die Leserichtung eines Textes an. Standard ist links-nach-rechts (Wert `ltr`), rechts-nach-links wird durch `rtl` abgekürzt.

Tabelle B.2: HTML-Attribute zur Internationalisierung

Attributname	Beschreibung
onload	Wird ausgelöst, wenn der Browser ein Dokument geladen hat bzw. alle Frames eines Framesets geladen hat.
onunload	Wird ausgelöst, wenn der Browser ein Dokument aus dem Fenster entfernt bzw. ein Frame aus einem Teilfenster entfernt.
onclick	Wird ausgelöst, wenn man mit der Maus auf das Element klickt.
ondblclick	Wird ausgelöst, wenn man mit der Maus auf das Element doppelt klickt.
onmousedown	Wird ausgelöst, wenn der Button der Maus über dem Element gedrückt wird.
onmouseup	Wird ausgelöst, wenn der Button der Maus über dem Element losgelassen wird.
onmouseover	Wird ausgelöst, wenn die Maus über das Element bewegt wird.
onmousemove	Wird ausgelöst, wenn man die Maus bewegt während man über dem Element ist.
onmouseout	Wird ausgelöst, wenn man die Maus über dem Element wegbewegt.
onkeypress	Wird ausgelöst, wenn eine Taste über dem Element gedrückt und losgelassen wird.
onkeydown	Wird ausgelöst, wenn eine Taste über dem Element gedrückt wird.
onkeyup	Wird ausgelöst, wenn eine Taste über dem Element losgelassen wird.
onfocus	Tritt ein, wenn ein Element durch die Tab-Taste oder die Maus aktiviert wird (beispielsweise ein Textfeld in einem Formular)
onblur	Tritt ein, wenn die Aktivierung eines Elements aufgehoben wird.
onsubmit	Tritt ein, wenn ein Formular abgeschickt wird.
onreset	Tritt ein, wenn ein Formular zurückgesetzt wird
onselect	Tritt ein, wenn ein Benutzer in einem Formulartextfeld Text markiert.
onchange	Wie onBlur, allerdings muss sich der Wert des betroffenen Elements seit Aktivierung zusätzlich geändert haben.

Tabelle B.3: Scripting Events in HTML

B.2 Vordefinierte Farben in HTML

RGB-Wert	Farbbezeichnung	RGB-Wert	Farbbezeichnung
#000000	Black	#008000	Green
#C0C0C0	Silver	#00FF00	Lime
#808080	Gray	#808000	Olive
#FFFFFF	White	#FFFF00	Yellow
#800000	Maroon	#000080	Navy
#FF0000	Red	#0000FF	Blue
#800080	Purple	#008080	Teal
#FF00FF	Fuchsia	#00FFFF	Aqua

Tabelle B.4: Vordefinierte Farben in HTML und Ihre RGB-Werte

Anhang C

Java

Art	Name	Beschreibung
Arithmetisch	+	Addition
	-	Subtraktion
	*	Multiplikation
	/	Division
	%	Modulus
Relational	¿	größer als
	¡	kleiner als
	¿=	größer gleich als
	¡=	kleiner gleich als
	!=	ungleich
	==	gleich
Logisch	!	NOT
	&&	AND
	||	OR
Bitweise	-	Komplement
	&	AND
	|	OR
	^	XOR
	<<	Linksshift
	>>	Rechtsshift
	>>>	Rechtsshift mit Nullauffüllung
Zuweisung	=	Zuweisung
	++	Inkrementiere und weise zu
	–	Dekrementiere und weise zu
	+=	Addiere und weise zu
	-=	Subtrahiere und weise zu
	*=	Multipliziere und weise zu
	/=	Dividiere und weise zu
	%=	Dividiere mit Rest und weise diesen zu
	&=	AND und weise zu
	|=	OR und weise zu
	^=	XOR und weise zu
	<<=	Linksshift und weise zu
	>>=	Rechtsshift und weise zu
	>>>=	Rechtsshift mit Nullauffüllung und weise zu
Besondere	(*Typ*) *a*	Konvertiere *a* nach *Typ*
	a `instanceof` *b*	Ist *a* Instanz von *b*?
	`new`	Erzeuge neues Objekt oder Array
	a ? *b* : *c*	if *a* then *b* else *c*

Tabelle C.1: Java-Operatoren

Literaturverzeichnis

[Alv95] Harald Alvestrand. *Tags for the Identification of Languages.* `http://www.ietf.org/rfc/rfc1766.txt`, März 1995.

[BL99] B. Bos and H.W. Lie. Cascading Style Sheets, Level 1. `http://www.w3.org/TR/1999/REC-CSS1-19990111`, November 1999.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext Markup Language 2.0. `http://www.ietf.org/rfc/rfc1866.txt`, November 1995.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. `http://www.ietf.org/rfc/rfc2396.txt`, August 1998.

[BLLJ98] B. Bos, H.W. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, Level 2. `http://www.w3.org/TR/1998/REC-CSS2-19980512`, Mai 1998.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators. `http://www.ietf.org/rfc/rfc1738.txt`, Dezember 1994.

[Cea95] Netscape Communications Corporation and et al. *Announcement of JavaScript, the Open, Cross-platform Object Scripting Language.* Dezember 1995.

[Cha02] Registered Character Sets. Technical report, IANA, April 2002.

[Con00] The Unicode Consortium. *The Unicode Standard, Version 3.0.* Addison-Wesley, 2000.

[Dub] Dublin Core Metadata Initiative. Dublin Core Homepage. `http://www.dublincore.org`.

[FB96a] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. `http://www.ietf.org/rfc/rfc2045.txt`, November 1996.

[FB96b] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. `http://www.ietf.org/rfc/rfc2046.txt`, November 1996.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, Juni 1999.

[Fla98] David Flanagan. *Java in a Nutshell.* O'Reilly, 2nd edition, 1998.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, 2nd edition, Juni 2000.

[HOIdR02] Barbara Heil, Elke Ott, and Thomas In der Rieden. *Komposition als Mittel der Bildsprache.* Universität des Saarlandes, 2002.

[IEE85] IEEE standard 754-1985 for binary floating-point arithmetic. Technical report, IEEE, 1985.

[ISO93] Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane. Technical Report 10646-1:1993, ISO/IEC, 1993.

[Jaw97] Jamie Jaworski. *Java 1.1 Developer's Guide.* Sams.net Publishing, 2nd edition, 1997.

[Mac] Macromedia Inc. Macromedia Homepage. `http://www.macromedia.com`.

[Nat] National Center for Supercomputer Applications. NCSA Homepage. `http://www.ncsa.uiuc.edu/`.

[Net] Netscape Communications Corporation. Netscape DevEdge. `http://developer.netscape.com`.

[Org] Organisation Européenne pour la Recherche Nucléaire. CERN Homepage. `http://www.cern.ch`.

[RLHJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. `http://www.w3.org/TR/1999/REC-html401-19991224/`, Dezember 1999.

[SACM96] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta. A Standard Default Color Space for the Internet. `http://www.w3.org/Graphics/Color/sRGB`, November 1996.

[SGM01] Standard General Markup Language ISO8879:1986. Technical report, International Organization for Standards, August 2001.

[Sun] Sun Microsystems Inc. Sun Homepage. `http://www.sun.com`.

[Wor] World Wide Web Consortium. W3C Homepage. `http://www.w3c.org`.