Webanwendungen
Vorlesung – Hochschule Mannheim

JavaScript

Inhaltsverzeichnis

Einführung

Historie von JavaScript

- LiveScript (1995) im Netscape Navigator 2.0
  - Entwickelt von Brendan Eich
  - umbenannt in JavaScript (Name wurde von Sun lizensiert)
  - hat aber nichts mit Java zu tun
- JScript (1996) von Microsoft eingeführt
  - lizenzunabhängige Sprachvariante
  - mit Erweiterungen durch Microsoft

Brendan Eich, Quelle: Wikipedia

Anwendung von JavaScript

- Benutzerinteraktion auf Clientseite
- Verbesserte Navigation
- Überprüfung von Benutzereingaben vor dem Absenden des Formulars
- Fehlermeldungen

- Dynamische Modifikation von HTML-Seiten zur Laufzeit
- Strukturanpassungen durch Einfügen / Entfernen von HTML-Elementen
- Formatanpassungen durch Ändern von CSS-Einstellungen

Anwendung von JavaScript

- Nachladen von Daten, ohne die HTML-Seite neu zu laden
- AJAX (Asynchronous JavaScript and XML)

- Nachbildung von GUI-Elementen, die in HTML fehlen

Einbinden von JavaScript: Eigene Datei

- Ähnlich wie CSS:
  - Normalfall in vielen Web-Projekten
  - Unterstützt Wiederverwendung (kann von mehreren HTML-Dateien genutzt werden)
  - Einbinden über <script>-Element im Header der Datei

```
<script type="text/javascript"
src=“js/berechnung.js"> </script>
```

Einbinden von JavaScript: Eingebettet

- Eingebettet in HTML-Datei
  - JavaScript nur in dieser Datei gültig
  - Keine Wiederverwendung
  - Über <script>-Element im Header der Datei

```
<script type="text/javascript">
  <!--
    /* ... Some JavaScript ... */
   var SinnDesLebens = 42;
  -->
</script>
```

Beispiel: Hello World in JavaScript

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Hello World</title>

<script type="text/javascript">
<!--
  alert("Hello World!");
-->
</script>

</head>

<body>
</body>
</html>
```

Grundlagen der Sprache

Scriptsprachen

- Meist über einen Interpreter
- implizit deklarierte Variablen
- dynamische Typisierung,
- automatische Speicherverwaltung,

Eigenschaften von JavaScript

- C ähnlicher Syntax
- Frei formatierbar
- Interpretiert
- Schwaches Typsystem (loosely-typed)
- Garbage-Collection (ähnlich zu Java)
- Fließkommazahlen
- Keine Klassen
- Objekte haben Properties und Methoden

Eigenschaften von JavaScript

- objektbasiert aber nicht objektorientiert
- Variablen und Funktionen können jederzeit angelegt werden
- Variable Anzahl von Parametern für Funktionen
- Portabel und hardwareunabhängig
- Laufzeitumgebung: JavaScript-Engine (besitzen alle WebBrowser)

Vergleich mit Java

- Ähnlicher Syntax
- Blöcke mit { }
- Kontrollstrukturen: if, switch, for, while, do, try catch
- Kommentare: //, /*...*/
- Stringverkettung mit +
- Variablen sind dynamisch typisiert
- keine Typdeklaration
- Zahlen sind immer Gleitkommazahlen (kein int, kein short, …)
- Schlüsselworte var leitet Variablendeklaration ein
- Variable kann nacheinander unterschiedliche Typen halten
- function leitet Funktionsdeklaration ein

Vergleich mit Java

- Objekterzeugung erfolgt mit new
- Statements werden mit ; beendet
- Am Zeilenende darf ; fehlen, wird automatisch eingefügt

Datentypen in JavaScript

- Eingebaute Datentypen von JavaScript
  - Boolean – Wahrheitswert (true oder false)
  - Number – Ganz- oder Fließkommazahl
  - String – Zeichenkette
  - Date – Datum
  - Array – Indiziertes oder assoziatives Array
  - Objekt – Vor- oder selbstdefinierte Objekte

Vergleichsoperator

- Zwei Vergleichsoperatoren
  - == Vergleich zweier Werte auf Inhalt
  - === Vergleicht zusätzlich auf Variablentyp

- typeof(variable) liefert den Datentyp der Variablen

Beispiel: Vergleichsoperatoren

```
var a = "10";
var b = 10;

print(typeof(a)); // -> string
print(typeof(b)); // -> number

var c1 = (a == b);
var c2 = (a === b);

print(c1); // -> true
print(c2); // -> false
```

Arrays

- Arrays sind dynamisch und erweiterbar (Objekte)
  - Index beginnt bei 0
  - unterschiedliche Typen als Inhalt möglich
  - Länge über .length auslesbar
- Erzeugung mit new Array() oder einem Array-Literal
  - initiale Größe kann optional angegeben werden
  - Array kann bei Erzeugung direkt initialisiert werden
- Zugriff auf Elemente über [index]

Beispiel: Array

```
// Dynamisches Array
var a1 = new Array();
a1[0] = "Hallo";
a1[1] = 42;
a1[22] = 23;

print(a1); // -> Hallo,42,,,,,,,,,,,,,,,,,,,,,23
print(a1.length);
// 23

// Array mit vorallokierter Größe
// kann aber weiter wachsen
var a2 = new Array(3);
a2[0] = "Hallo";
a2[1] = 42;
a2[12] = 23;

print(a2);
// -> Hallo,42,,,,,,,,,,,23
print(a2.length);
// 13
```

Beispiel: Array

```
// Array direkt initialisieren
var a3 = new Array("Hallo", 24, 23);


// Array Literale
var leer = [];
var zahlen = [ "eins", "zwei", "drei" ];

print(leer[0]);   // -> undefined
print(zahlen[1]); // -> zwei


// Löschen von Elementen (Lücke bleibt)
var zahlen = [ "eins", "zwei", "drei" ];
delete zahlen[1];

print(zahlen);        // -> eins,,drei
print(zahlen.length); // -> 3
```

Beispiel: Array

```
// Entfernen von Elementen (ohne Lücke)
var zahlen = [ "eins", "zwei", "drei" ];
zahlen.splice(1, 1);
print(zahlen);        // -> eins,drei
print(zahlen.length); // -> 2


// Anhängen von Elementen
var a5 = [ "a", "b", "c" ];

a5[a5.length] = "d";
print(a5); // -> a,b,c,d

a5.push("e");
print(a5); // -> a,b,c,d,e
```

Assoziatives Array

- Besonderes Array mit String als Index
  - unterschiedliche Typen als Inhalt möglich
  - ähnlich einer Map in Java
- Erzeugung mit new Array()
- Verarbeitung
  - Auslesen eines Elements über ["Index"]
  - Iterieren über alle Elemente mit foreach-Schleife
    for (var e in array) { ... }

Beispiel: Assoziatives Array

```
// Assoziatives Array
var a4 = new Array();
a4["go"] = "gehen";
a4["walk"] = "gehen";
a4["sleep"] = "schlafen";

print(a4["go"]);
// -> gehen
for (var e in a4 ) { print(e); }


// Literal von assoziativem Array
var a5 = { "go": "gehen", "walk" : "gehen", "sleep" : "schlafen" };

for (var e in a5 ) { print(e); }
```

Sichtbarkeit von Variablen in JavaScript

- Variablen sind in der ganzen Funktion sichtbar
- Blöcke haben (anders als in Java) keine Auswirkungen auf die Sichtbarkeit

```
var a = 5;  // global sichtbar

function f() {
    var b = 7; // innerhalb der Funktion sichtbar

    if (b >= 7) {
        var c = 9; // innerhalb der Funktion sichtbar
    }

    print(a); // -> 5
    print(b); // -> 7
    print(c); // -> 9
}

f();
```

Das letzte mal…

- JavaScript Einführung
- Grundlagen der Sprache
- Datentypen, Vergleichsoperatoren === und ==
- Indizierte und assoziative Arrays
- Sichtbarkeit der Variablen

Funktionen

Funktionen

- Funktionen sind Objekte
- sie können über Variablen referenziert werden
- sie können an andere Funktionen übergeben werden
  - Deklaration einer Funktion
- mit dem Schlüsselwort function
- mit dem speziellen Konstruktor Function
  - Aufruf einer Funktion
- über ihren Namen
- über eine Variable, die auf die Funkion zeigt

Funktionen

- Parameterübergabe wie in Java
- pass-by-value: Kopie der Parameter wird angelegt
- formale Parameter sind lokale Variable
- bei Objekten wird die Referenz kopiert, nicht das Objekt
- Rückgabe von Werten
- Rückgabeparameter wird nicht deklariert
- Schlüsselwort return für Rückgabe
- Funktionen können Funktionen enthalten

Beispiel: Deklaration von Funktionen

```
// Deklaration einer Funktion
function add(a, b) {
    var result = a + b;
    return result;
}

// Deklaration einer Funktion
var sub = function(a, b) {
    return a - b;
};

// Funktionskonstruktor
var f = new Function("a", "b", "return a + b;");
var h = f(4, 8);
print(h);
// -> 12
```

Beispiel: Argumente einer Funktion

- Fehlende Argumente gelten als undefined und führen nicht zu einem Fehler.

- Ein Funktionsaufruf kann mehr Argumente übergeben, als die Funktion erwartet:

```
function summe( x, y )
{
   var add = x + y;
   return add;
 }

summe( 4, 5, 17 );
// gibt 9 zurück
```

- ignoriert den überflüssigen dritten Parameter.

Beispiel: Argumente einer Funktion

- JavaScript bietet Zugriff auf überzählige Parameter:

```
// Funktion mit beliebig vielen Parametern
function varag() {
    var summe = 0;
    for ( var i = 0; i < arguments.length; i++) {
        summe += arguments[i];
    }
    return summe;
}

var n = varag(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
print(n);
// -> 55
```

Beispiel: Übergabe einer Funktion

```
// Übergabe einer Funktion an eine Funktion
var add = function(a, b) { return a + b; };
var sub = function(a, b) { return a - b; };

function calc(operation, a, b) {
    return operation(a, b);
}

var o = calc(add, 4, 2); print(o);
// -> 6
var p = calc(sub, 4, 2); print(p);
// -> 2
```

Eingebaute Funktionen

- Beispiele für eingebaute Funktionen
  - eval(string) – interpretiert übergebenen String als Code
  - isFinite(wert) – prüft auf gültigen Wertebereich
  - isNaN(wert) – prüft auf ungültigen Wertebereich
  - parseFloat(string) – in Fließkommazahl umwandeln
  - parseInt(string) – String in Ganzzahl umwandeln
  - Number(wert) – In Zahl umwandeln
  - String(wert) – in Zeichenkette umwandeln
  - encodeURI(string) – in URI-Format umwandeln
  - decodeURI(string) – aus URI-Format umwandeln
  - Zahl.toFixed(n) – n Nachkommastellen erzwingen

JSON

Was ist JSON?

- *JSON*: *JavaScript Object Notation*
  - JavaScript kennt Objekt-Literale
  - Idee: Objekt-Literale als Textdateien austauschen = JSON
- Vorteile
  - deutlich kompakter als XML
  - Menschen lesbarer
  - auch für andere Programmiersprachen verfügbar, z. B. Java, C, C#, PHP ...

Arrays mit JSON

- Deklaration von Arrays mit JSON
  - Liste von Elementen, durch Komma getrennte
  - umrahmt von eckigen Klammern []
  - Syntax: [element1, element2, element3 ...]

```
var monate = ["Januar", "Februar", "März", "April", "Juni"];
var prim = [1, 2, 3, 5, 7, 11, 13];
var bools = [true, false, false, true];
```

Objekte in JSON

- Liste von Schlüssel-Wert-Paaren, umrahmt von Klammern { }
- Syntax: { schlüssel1: wert1, schlüssel2: wert2 … }

```
var student = {
    name: “Urlaub, Farin",
    alter: 25,
    note: 1.0,
    adresse: {
        strasse: “Ärztestr. 76",
        plz: 42231,
        ort: “Berlin"
    }
};

print(student.name);        // -> Urlaub, Farin
print(student.adresse.plz); // -> 42231
print(student.adresse.ort); // -> Berlin
```

Alternativer Syntax für JSON

- JavaScript-Objekte können auch als assoziatives Array aufgefasst werden
- Syntax: { "schlüssel1": "wert1", "schlüssel2": "wert2" … }

```
var student = {
    "name": “Felsenheimer, Bela B.",
    "alter": 51,
    "note": 1.0,
    "adresse": {
        "strasse": “Code-BStr 42",
        "plz": 42231,
        "ort": “Hamburg"
    }
};

print(student["name"]);          // -> Felsenheimer, Bela B.
print(student["adresse"]["plz"]); // -> 42231
```

Funktionen mit JSON

- Im Objekt-Literal können auch Funktionen zugewiesen werden
- Syntax: { name: function() { } ... }

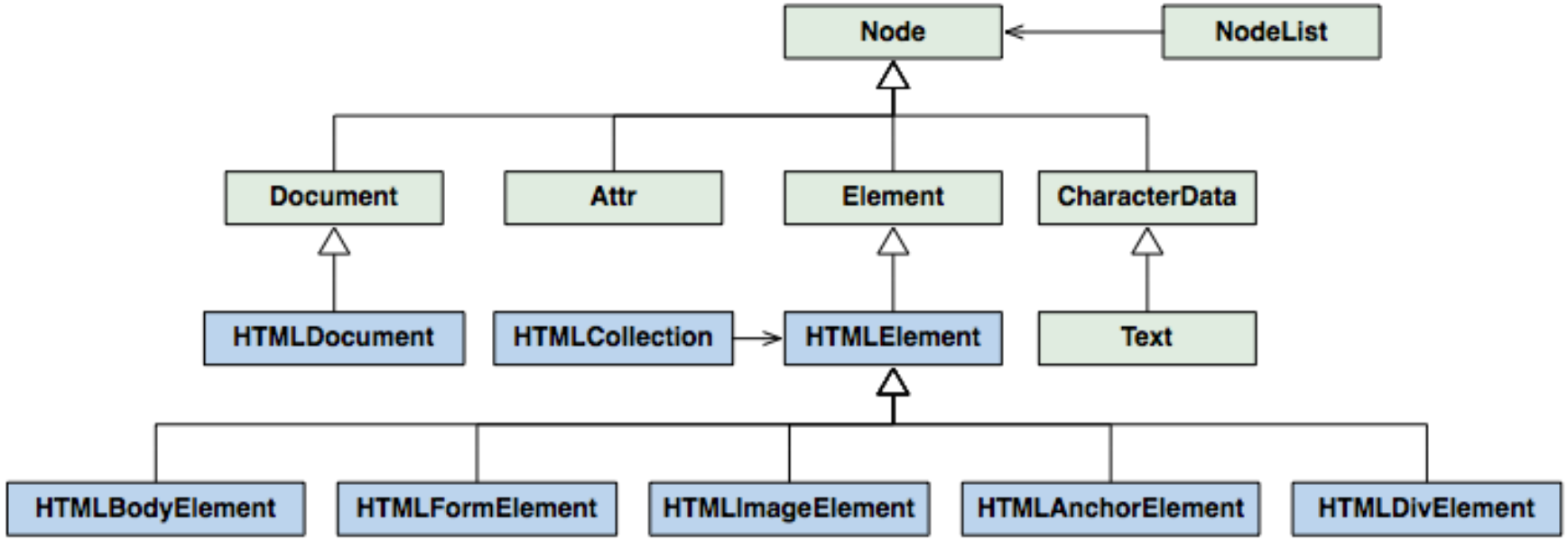
```
var student = {
    name: “Gonzales, Rodrigo",
    alter: 48,
    altern: function() { this.alter++; }
};

print(student.alter); // -> 25
student.altern();
print(student.alter); // -> 26
```

DOM Level 2

DOM Level 2

- *DOM Level 2*
  - Schnittstelle zum Zugriff auf HTML-Dokumente
- formal spezifiziert und standardisiert durch W3C
- von allen gängigen Browsern unterstützt
- auch für andere Programmiersprachen
  - Erweiterungen gegenüber DOM Level 0
- alle Elemente und alle Attribute einer HTML-Seite ansprechbar
- Zugriff auf XHTML-Dokumente
- Erzeugung von Elementen und Einhängen in den Dokumenten-Baum

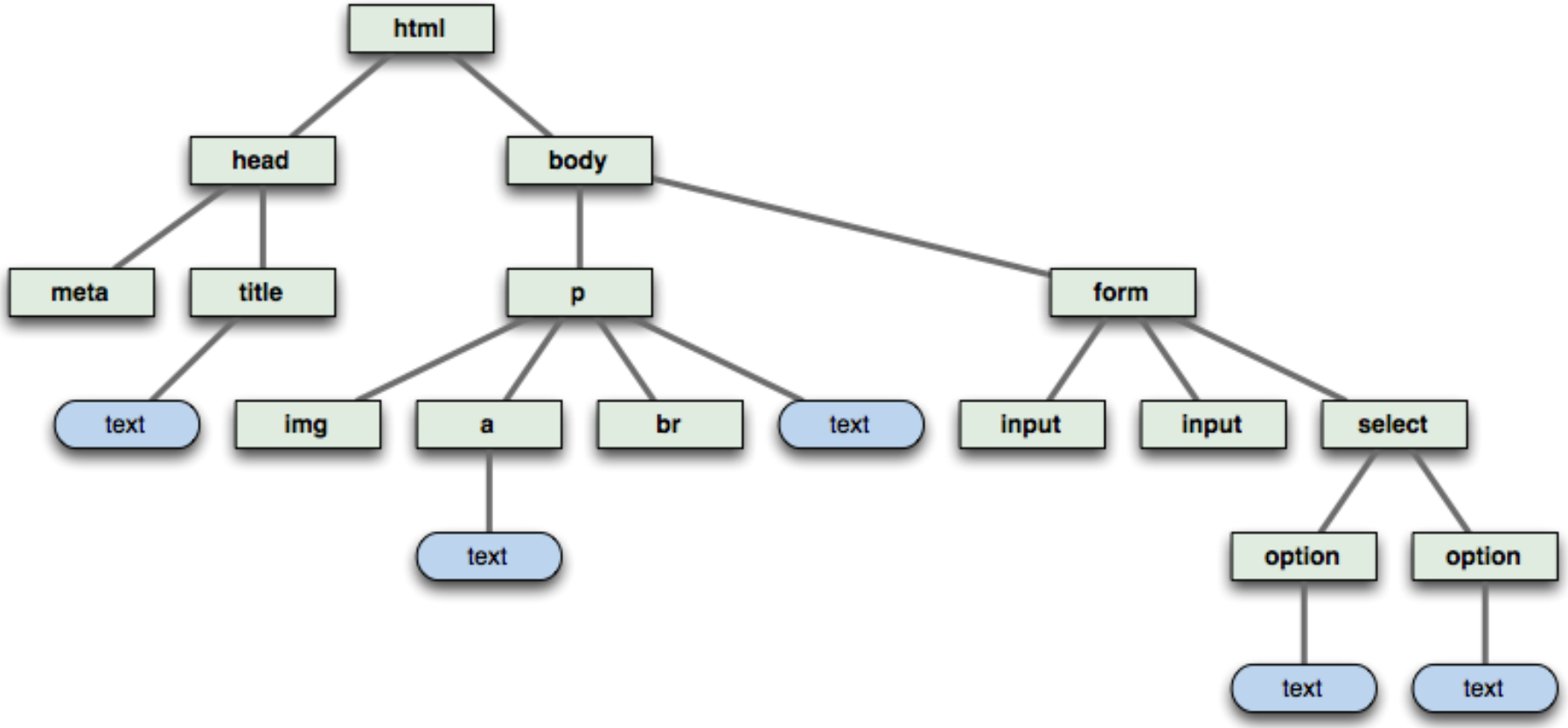
DOM Interfaces



Beispiel: HTML-Dokument

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>DOM-Tree</title>
</head>
<body>
  <p>
    Ein erster Absatz <img src="logo.gif"/> <br/>
    <a href="http://www.hs-mannheim.de">Home</a>
  </p>
  <form action="#" method="GET">
    <input type="text" name="nachname"/>
    <input type="text" name="vorname"/>
    <select name="geschlecht">
      <option value="m">männlich</option>
      <option value="f">weiblich</option>
    </select>
  </form>
</body>
</html>
```

Beispiel: DOM-Tree zum Dokument



Erweiterungen durch DOM Level 2

- DOM Level 2 erweitert das document-Objekt um
  - Methoden zum Erstellen von Knoten
  - Methoden zum Auffinden von Knoten über ID, Name-Attribut und Elementname

Methode	Bedeutung
createElement(tagName)	Erstellt Element vom Typ tagName
createAttribute(attrName)	Erstellt Attribut mit dem Namen attrName
createTextNode(initText)	Erstellt und initialisiert Textknoten
getElementById(id)	Liefert Element mit id-Attributwert id
getElementsByName(name)	Liefert Elemente mit dem Namen name
getElementsByTagName(tag)	Liefert Elemente des Tag-Typs tag

Baum-Operationen

- node, document und element haben Methoden zum Durchwandern und Manipulieren des Baums
  - Erzeugung neuer Knoten mit Hilfe des document-Objektes
  - Durchwandern des Baums mit Hilfe der node-Objekte
  - Methoden zur Strukturänderung der node-Objekte

Zugriff auf DOM-Knoten

- Direkter Zugriff auf Element per eindeutiger id
- jedes benötigte HTML-Element muss id-Attribut haben
- knoten = document.getElementById("ID")
  - Zugriff über Element-Typ
- knoten = document.getElementsByTagName("h3")[2]
  - Zugriff auf Elemente über deren name-Attribut
- nicht jeder Tag darf ein name-Attribut haben, evtl. mehrere Elemente mit demselben Namen (z. B. Radiobuttons)
- knoten = document.getElementsByName("abc")[0]
  - Zugriff über Position im Dokumentenbaum
  - Verwendung von DOM Level 0-Pfaden

Weitere Möglichkeiten zum Zugriff

- speziell beim Aufruf von Handlerfunktionen
  - this verweist auf das Element, in dem der Code steht
     ...
  - nur gültig innerhalb des HTML-Tags
- veraltete Methode für manche Objekte
  - Zugriff ohne Nennung der Collection über das name-Attribut
    document.MeinFormular.Eingabe.value = "alt";
    document.MeinBild.src = "bild.gif";
  - Sollte vermieden werden. Stattdessen besser
    document.getElementsByName("...") verwenden

name oder id für Adressierung?

- Für Adressierung ist id zu bevorzugen
  - name ist nur bei manchen Tags zulässig
  - name ist nicht eindeutig
  - name hat unterschiedliche Bedeutungen
  - <a> Sprungmarke
  - <input> Parametername
  - name ist nur aus Kompatibilitätsgründen noch zulässig in DOM2 Core nicht enthalten
  - id ist bei allen Tags zulässig

Das node-Objekt

- Knoten des DOM-Baumes werden durch node-Objekte repräsentiert
- Mögliche Arten von Knoten
  - *Elementknoten* – entspricht exakt einem Element
  - *Textknoten* – stellt textuellen Inhalt eines Elements dar
  - *Attributknoten* – entspricht exakt einem Attribut

node-Objekt: Attribute

Attribut	Bedeutung
nodeName	Elementname
nodeType	Knotentyp (1=Element, 2=Attribut, 3=Text)
nodeValue	Wert des Knotens
data	Rumpftext (nur bei Textknoten)
attributes	Array aller Attribute
parentNode	Vaterknoten
childNodes	Array aller Kinderknoten
firstChild	erster Kindknoten
lastChild	letzter Kindknoten knoten
previousSibling	linker Nachbarknoten
nextSibling	rechter Nachbarknoten

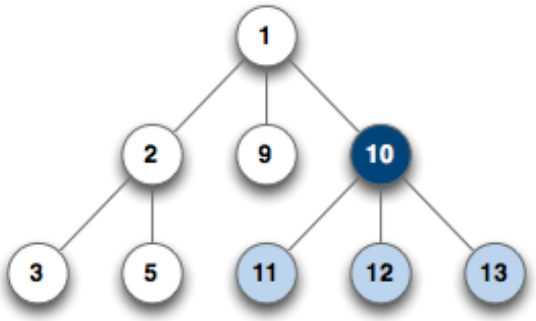
node-Objekt: Methoden

Methode	Bedeutung
cloneNode(recursiveFlag)	Liefert Knotenkopie (rekursiv falls *recursiveFlag=true*)
hasChildNodes()	Liefert *true*, wenn Kindknoten vorhanden sind, sonst *false*
appendChild(new)	Knoten *new* als letztes Kind anfügen
insertBefore(new, node)	Knoten *new* links von bestehendem Knoten *node* einfügen
removeChild(node)	Kind *node* entfernen
replaceChild(new, old)	Ersetzt Kind *old* durch *new*
getAttribute(name)	Liefert Wert des Attributes *name*
setAttribute(name, value)	Setzt Wert des Attributes *name* auf *value*
removeAttribute(name)	Attribut mit Namen *name* entfernen
getAttributeNode(name)	Liefert Knoten des Attributs *name*
setAttributeNode(node)	Attributknoten *node* hinzufügen
removeAttributeNode(node)	Attributknoten *node* entfernen

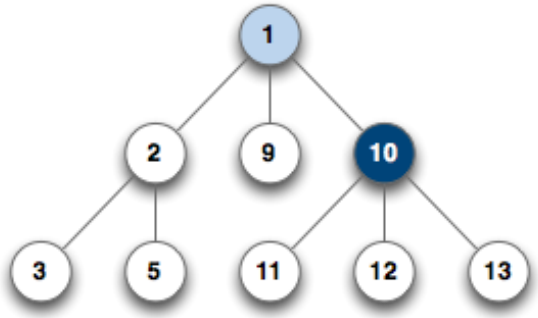
node-Objekt: Methoden

Methode	Bedeutung
appendData(text)	Hängt *text* an
insertData(start, text)	Fügt *text* ab der Position *start* ein
replaceData(start, length, text)	Ersetzt *length* Zeichen des Textes ab *start* mit *text*
deleteData(start, length)	Entfernt *length* Zeichen des Rumpftextes ab *start*

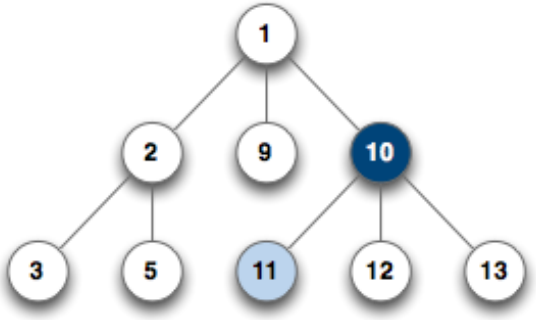
Knoten-Beziehungen



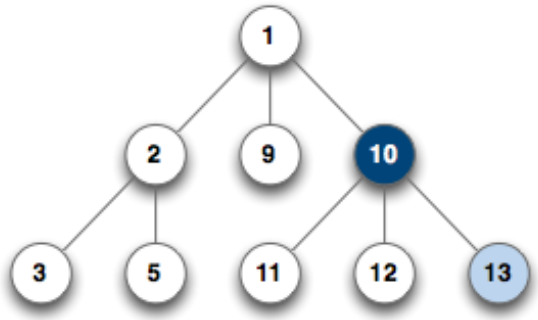
childNodes



parent

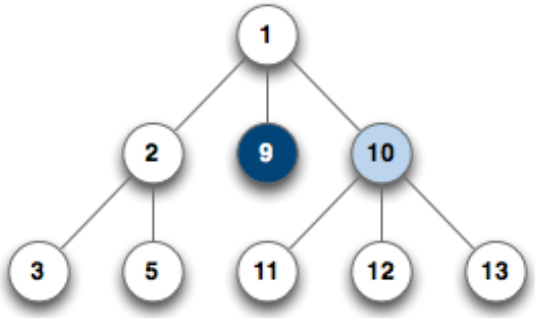


firstChild

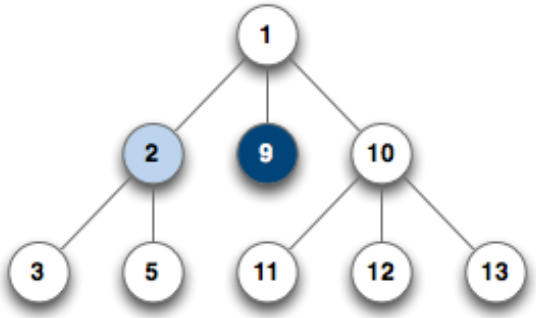


lastChild

Knoten-Beziehungen



followingSibling



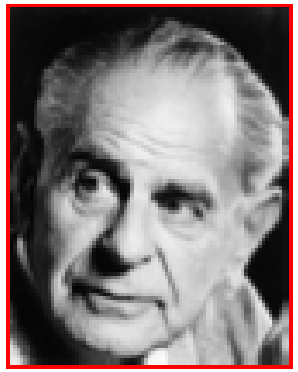
previousSibling

Zugriff auf Attribute

- Zwei Zugriffsmöglichkeiten auf Attribute
  - über die Kernobjekte von DOM 2
  - Attribute sind in Unterknoten gespeichert
  - Zugriff über element.getAttribute() bzw. element.setAttribute()
  - z. B. img.setAttribute("src", "bilder/portrait.jpg");
  - über DOM 2 HTML
  - Objekte haben Attribute, die direkt angesprochen werden können
  - Namen entsprechen HTML-Namen (Ausnahme class wird zu className)
  - z. B. img.src = "bilder/portrait.jpg";

Beispiel: Erzeugung (DOM-Stil)

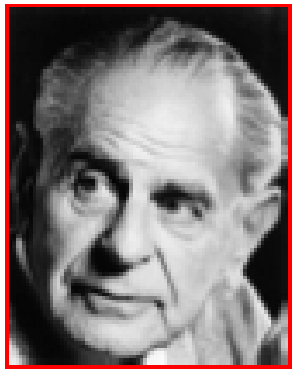
```
.....
<head><title>DOM-Erzeugung</title></head>
<body>
  <div id="leer"></div>

  <script type="text/javascript">
    var div = document.getElementById("leer");

    var img = document.createElement("img");
    img.setAttribute("src", "bilder/portrait.jpg");
    img.setAttribute("alt", "Sir Karl Popper");
    img.setAttribute("width", "100px");
    img.setAttribute("style", "border: 1px solid; border-color: red;");
    div.appendChild(img);

    var p = document.createElement("p");
    var txt = document.createTextNode("Dies ist ein Absatz");
    p.appendChild(txt);
    div.appendChild(p);
  </script>
</body>
.....
```



Dies ist ein Absatz

Beispiel: Erzeugung (HTML-Stil)

```
...
<head><title>DOM-Erzeugung</title></head>
<body>
  <div id="leer"></div>

  <script type="text/javascript">
    var div = document.getElementById("leer");

    var img = document.createElement("img");
    img.src = "bilder/portrait.jpg";
    img.alt = "Sir Karl Popper";
    img.width = 100;
    img.style.borderWidth = "1px";
    img.style.borderStyle = "solid";
    img.style.borderColor = "red";
    div.appendChild(img);

    var p = document.createElement("p");
    var txt = document.createTextNode("Dies ist ein Absatz");
    p.appendChild(txt);
    div.appendChild(p);
  </script></body>
....
```



Dies ist ein Absatz

Zugriff auf Inline-Styles

- Zugriff auf Style des HTML-Elements (nicht CSS-Datei)
  - haben Vorrang vor CSS-Datei, vgl. Kaskadierungsregeln
  - Werte sind Strings (enthalten px, %, pt, em)
  - Style ist als Unterobjekt realisiert:
    `document.getElementById("Hugo").style.fontSize = "12pt";`
- Bildung der CSS-Attributnamen
  - Bindestriche sind nicht zulässig in JavaScript-Bezeichnern
  - Bindestrich entfernen, nächsten Buchstaben groß schreiben:
    `fontSize`, `fontWeight`, `backgroundColor` …

CSS-Klassen und JavaScript

Entfernen einer Klasse

```
1  var id='myElementId';
2  var myClassName=" className";
3  var d;
4  d=document.getElementById(id);
5  d.className=d.className.replace(myClassName,"");
```

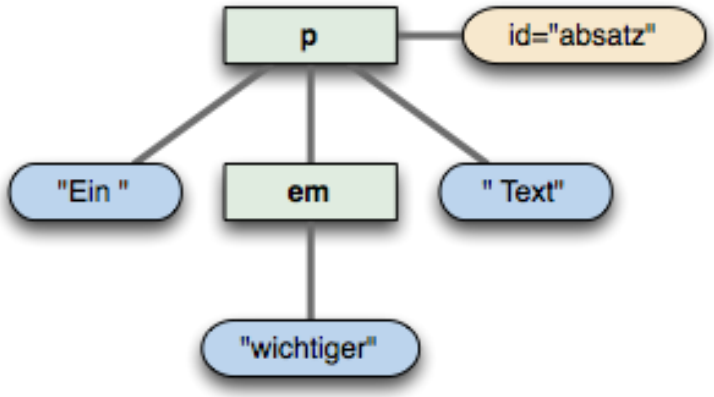
Hinzufügen einer Klasse

```
var id='myElementId';
var myClassName=" className";
var d;
d=document.getElementById(id);
d.className=d.className.replace(myClassName,"");
d.className = d.className + myClassName;
```

Zugriff auf Text

- Text in einem Tag wird in Unterknoten gespeichert
  - Text steht im Attribut *nodeValue*
  - kann durch Zuweisung geändert werden (ohne HTML-Tags)

```
<p id="absatz">Ein <em>wichtiger</em> Text</p>
```



OnLoad

- Der DOM – Baum muss zuerst vollständig gerendert werden bevor manipuliert werden kann
- Möglich sobald „onload“ Event gefeuert wird:

```
<body onload="onload()">
```

Aufgabe

Implentieren Sie einen Toogle-Button

- Default Zustand: OFF und rot
- Bei Klick auf die rechte Schaltfläche verschwindet das OFF und der Hintergrund wird weiß.
  Auf der rechten Schaltfläche wieder der Hintergrund grün und ein ON erscheint

Zurück zu reinem JavaScript

Callbacks

- Funktionen sind Objekte
  - Vorteil: Sie können als Parameter einer anderen Funktion übergeben werden
- Normalerweise:
  - Input (in Form von Parametern)
  - Funktionsvoodoo
  - Output (z.B. eine Zahl)
- Callbacks verhindern das Blockieren von Programmen

Traditionelle I/O ohne Callbacks

```
var db_result = db.query("select * from table");
doSomething(db_result); //wait!
doSomeOtherVoodoo();    //blocked
```

I/O mit Callbacks: Non-blocking I/O

```
db.query(“select * from table“,function(db_result){
   doSomething(db_result); //wait
});
doSomeOtherVoodoo();
```

objektbasiert statt objektorientiert

Erzeugen von Objekten

- Objekte können erzeugt werden über
  - ein Objekt-Literal (*JSON*)
  - eine Konstruktor-Funktion, die mit new aufgerufen wird
  - eine Factory-Funktion, die ohne new aufgerufen wird
- Konvention
  - Konstruktor-Funktionen fangen mit Großbuchstaben an
  - Factory-Funtionen fangen mit Kleinbuchstaben an

Interne Repräsentation von Objekten

- JavaScript Objekte sind assoziative Arrays
- Attribute und Methoden sind Array-Elemente
- Daraus ergeben sich folgende Erkenntnisse:

1. Der Punkt-Operator ist (fast) äquivalent zum []-Operator.
2. Man kann einem JavaScript-Objekt nach seiner Instanziierung neue Eigenschaften und Methoden zuweisen.
3. Man muss nicht den Punkt-Operator benutzen, um auf Eigenschaften oder Methoden zuzugreifen.
4. Man kann vorhandene Methoden (Attribute sowieso) überschreiben

Beispiel: Objekt-Literal (JSON)

```
// Objekt-Literal
var schulze = {
        name: "Schulze, Herbert",
        geboren: "4.5.1954",
        details: function() {
            return "Name: " + this.name
                + ", geboren: " + this.geboren;
        }
    };

print(schulze.details());
```

Beispiel: Konstruktor-Funktion

```
// Konstruktor-Funktion, muss mit new aufgerufen werden
function Mitarbeiter(name, geboren) {
    this.name = name;
    this.geboren = geboren;
    this.details = function() {
        return "Name: " + this.name
            + ", geboren: " + this.geboren;
    };
}

var meier = new Mitarbeiter("Meier, Franz", "1.5.1972");
var mueller = new Mitarbeiter("Müller, Peter", "8.11.1981");
print(meier.details());
print(mueller.details());
```

Beispiel: Factory-Funktion

```
// Objekt-Fabrik, muss ohne new aufgerufen werden
function makeMitarbeiter(name, geboren) {
    var that = new Object();
    that.name = name;
    that.geboren = geboren;
    that.details = function() {
        return "Name: " + this.name
            + ", geboren: " + this.geboren;
    };

    return that;
}

var meier = makeMitarbeiter("Meier, Franz", "1.5.1972");
var mueller = makeMitarbeiter("Müller, Peter", "8.11.1981");
print(meier.details());
print(mueller.details());
```

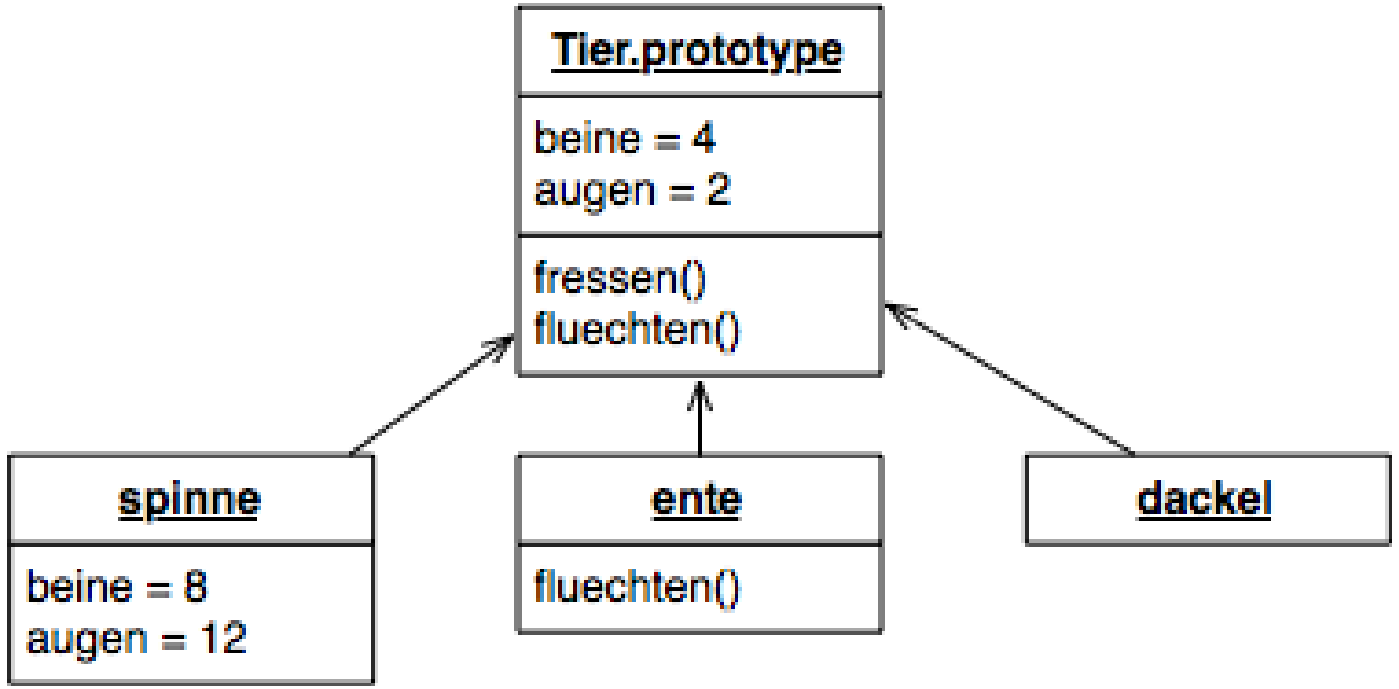
Objektbasierend statt objektorientiert

- JavaScript kennt keine Klassen, sondern nur Objekte
  - alles in JavaScript ist ein Objekt (auch Funktionen)
  - Objekte erben von Objekten (prototypische Vererbung)

Prototype

- Zu jeder Funktion gehört ein Prototyp (*prototype object*)
- Zugriff über Funktionsname.prototype
- Wird die Funktion als Konstruktor (mit new) verwendet, dann hat das erzeugte Objekt eine Referenz zum Prototypen
- Sind Attribute und Funktionen nicht im Objekt definiert, dann werden sie im Prototypen gesucht
- Mit delete Funktionsname.prototype.attribute/method kann ein dynamisch hinzugefügtes Attribut/ Methode wieder gelöscht werden

Beispiel: Prototype



Beispiel: Prototype

```
function Tier(name) {
    this.name = name;
}

Tier.prototype.fressen = function() { print("Mampf"); };
Tier.prototype.fluechten = function() { print("Lauf weg"); };
Tier.prototype.beine = 4;

var spinne = new Tier("Spinne");

spinne.beine = 8;
spinne.augen = 12;

spinne.fressen();     // -> Mampf
spinne.fluechten();   // -> Lauf weg
print(spinne.beine);  // -> 8
print(spinne.augen);  // -> 12
```

Beispiel: Prototype

```
var dackel = new Tier("Dackel");

Tier.prototype.augen = 2;

dackel.fressen();     // -> Mampf
dackel.fluechten();   // -> Lauf weg
print(dackel.beine);  // -> 4
print(dackel.augen);  // -> 2


var ente = new Tier("Ente");
ente.fluechten = function() { print("Flieg weg"); };

Tier.prototype.beine = 2;

ente.fressen();
// -> Mampf
ente.fluechten();
// -> Flieg weg
print(ente.beine);
// -> 4
```

Prototypische Vererbung

- Objekte können als Vorlage für neue Objekte dienen
  - Es gibt verschiedene Möglichkeiten ein Objekt zu erzeugen, dass das gegebene Objekt als Vorlage benutzt
  - Das erzeugte Objekt kann man anpassen (erweitern, Methoden verändern etc.)
  - Von dem neuen Objekt kann man wieder weitere Objekte erzeugen

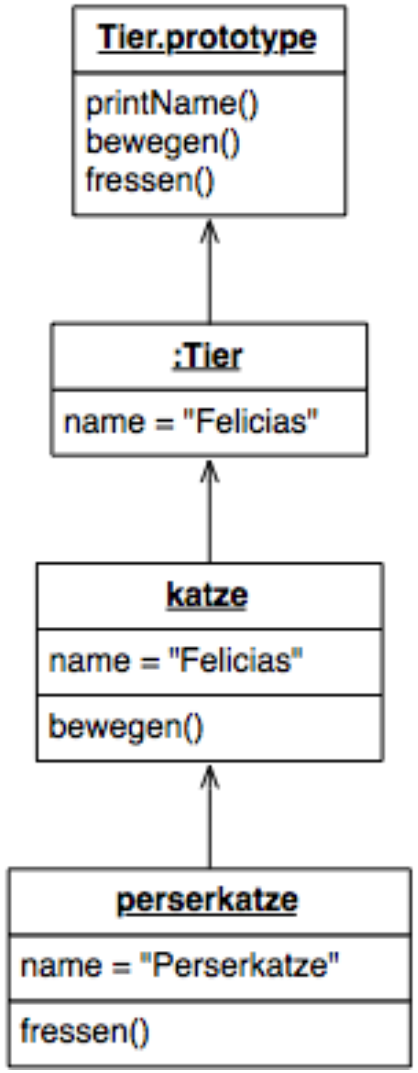
Beispiel: Prototypische Vererbung (create)

```
// Basis-Objekt für Tiere
function Tier(name) {
    this.name = name;
}

Tier.prototype.printName = function() { print(this.name); };
Tier.prototype.bewegen = function() { print("bewege mich"); };
Tier.prototype.fressen = function() { print("fresse"); };

// Katze anlegen, mit Tier als Vorlage
var katze = Object.create(new Tier("Felizitas"));
katze.bewegen = function() { print("leise schleichen"); };

katze.bewegen();
// -> leise schleichen
katze.fressen();
// -> fresse
katze.printName();
// -> Felizitas
```

Beispiel: Prototypische Vererbung (create)

```
// Perserkatze anlegen, mit Katze als Vorlage
var perserkatze = Object.create(katze);
perserkatze.name = "Perserkatze";
perserkatze.fressen = function() { print("Sheba geniessen"); };

perserkatze.bewegen();
// -> leise schleichen
perserkatze.fressen();
// -> Sheba geniessen
perserkatze.printName();
// -> Perserkatze
```

Beispiel: Prototypische Vererbung



Beispiel: Prototypische Vererbung

```
var User = function () {};
 User.prototype.setIp = function (ip) {
    this.ip = ip;
};

User.prototype.setLastVisit = function (date) {
    this.lastVisit = date;
};

var Customer = function (name, email) {
    this.name = name;
    this.email = email;
};

Customer.prototype = new User();
Customer.prototype.constructor = Customer;
```

Kapselung und Information-Hiding

- JavaScript kennt keine Zugriffsbeschränkungen
- Alle Attribute eines Objektes sind immer erreichbar
- Information-Hiding kann trotzdem erreicht werden

- Man verwendet eine Factory-Methode
- Methode erzeugt ein Objekt für die privaten Variablen
- Methode erzeugt ein Objekt für die öffentlichen Funktionen
- Dem privaten Objekt werden alle privaten Variablen und privaten Funktionen hinzugefügt
- Das öffentliche Objekt greift auf die privaten Daten zu (was es wegen der Sichtbarkeit der Daten innerhalb der Funktion darf)
- Das öffentliche Objekt wird zurückgegeben

Beispiel: Kapselung

```
function erzeugeTier(name, my) {
    var that = {}; // leeres Objekt anlegen
    var my = {};   // leeres Objekt anlegen

    my.name = name;

    that.getName = function() { return my.name; };
    that.setName = function(name) { my.name = name; };

    return that;
}

var hund = erzeugeTier("Hasso");
print(hund.getName());    // -> Hasso
hund.setName("Archibald");
print(hund.getName());    // -> Archibald
// my.name ist nicht sichtbar
```

Beispiel: Kapselung

```
function erzeugeKatze(name) {
    var that = erzeugeTier(name);

    that.schnurre = function() { print("Brrrrrrr"); };

    return that;
}

var katze = erzeugeKatze("Felizitas");
print(katze.getName());  // -> Felizitas
katze.schnurre();        // -> Brrrrrrr
```