Benvenuto nel wikibook:

JavaScript

Autore: Ramac

```
12 function impostaCookie (nome, valore, percorso, scadenza) {
13    valore=escape(valore);
14
15    if (scadenza == "") {
16       var oggi = new Date();
17       oggi.setMonth(oggi.getMonth()+6);
18       scadenza=oggi.toGMTString();
19    }
20    if (percorso!="")
21       percorso= ";Path=" + percorso;
22
23    document.cookie = nome + "=" + valore + ";expires=" + scadenza + percorso;
24 }
25
```

Versione attuale del libro >>

© Copyright 2007, utenti di Wikibooks. Questo libro è stato pubblicato dagli utenti di Wikibooks.

GFDL 2007

E' permesso copiare, distribuire e/o modificare questo documento secondo i termini della GNU Free Documentation License, Versione 1.2 o qualsiasi versione successiva pubblicata dalla Free Software Foundation;, senza le Sezioni Invarianti costituite dalla prima di copertina e dal paragrafo "Licenza". Una copia della licenza è contenuta nella sezione intitolata "Licenza".

Autori: Ramac, Diablo, Bimbot, Wim_b.

Copertina: Un listato Javascript di esempio, ed è rilasciata nel pubblico dominio.

Indice generale

Premessa....4
Finalità....4
Libri correlati....4
Altri progetti....5
Bibliografia....5
Collegamenti esterni....5
Introduzione....5
Breve storia di JavaScript....5
Strumenti necessari....5
Limitazione nell'uso di JavaScript....6
Compatibilità tra browser....6
l nostro primo programma....6

Inserire un JavaScript in una pagina HTML....6
Hello, world!....7
Inserire dei commenti....8
Tipi di dati e variabili....8
Tipi di dati....8
Numeri....8
Not a Number....9
Stringhe....9
Dati booleani....9
Variabili....9
Operatori e casting....10
Operatori matematici....10
Operatori stringa....12
Operatori booleani....12
Comporre le espressioni....13
Le strutture di controllo....13
Selezione....13
Selezione semplice....13
Blocchi if annidati e condizioni multiple....14
else if....14
switch....15
Operatore ternario....17
Iterazione....17
Cicli con condizione....17
Cicli con contatore....18
Iterare sugli elementi di un array....19
Funzioni definite dall'utente....19
Creare le proprie funzioni: una panoramica....19
Usare i parametri....20
Impostare un valore di ritorno....20
Oggetti....20
Cosa sono gli oggetti....21
Gli oggetti in JavaScript....21
Proprietà....21
Metodi....22
La struttura with....22
Oggetto String....22
Costruttore....22
Proprietà....23
Metodi....23
toLowerCase() e toUpperCase()....23
charAt() e charCodeAt()....24
fromCharCode()....24
indexOf() e lastIndexOf()....24
substr()....25
replace()....25
split()....25
Oggetto Math....26
Proprietà....26
Metodi....26
Arrotondare i numeri....26
Generare numeri casuali....27

Un esempio pratico....27
Oggetto Date....28
Costruttore....28
Metodi....29
Recuperare i valori della data....29
Impostare i valori della data....31
Lavorare con l'ora....31
Gestire i fusi orari....31
Oggetto Array....32
Cos'è un array....32
Gli array in JavaScript....32
Proprietà....33
Metodi....33
concat()....33
sort()....33
reverse()....33
slice()....34
Iterare sugli elementi di un array....34
Cookie....34
Impostare i cookie....35
Ottenere i cookie....35
Verificare se i cookie sono attivi....36
Timer....37
One-shot timer....37
Impostare intervalli regolari....37
BOM....38
L'oggetto window....38
Proprietà....38
defaultStatus e status....38
frames, length e parent....38
opener....39
Oggetti come proprietà....39
Metodi....39
alert(), confirm() e prompt()....39
blur() e focus()....39
open()....39
close()....40
Gli eventi nel BOM....40
Gli eventi in JavaScript....41
Eventi del mouse....41
Eventi della tastiera....41
Eventi degli elementi HTML....41
Eventi della finestra....41
Eventi come attributi HTML....41
Eventi come proprietà....42
Restituire dei valori....42
Oggetto document....43
Metodi....43
Proprietà....44
Accedere ai link nella pagina....44
L'array images....44
Accedere ai moduli....45

I campi dei moduli....45
Caselle di testo....46
Aree di testo....46
Pulsanti....46
Pulsanti di opzione....47
Liste....47
Inserire e aggiungere elementi....48
Altri oggetti del BOM....48
location....48
history....48
navigator....49
screen....49
DOM....50
BOM vs DOM....50
La struttura gerarchica di un documento....50
Vari tipi di nodo....51
Proprietà e metodi del DOM....51
Ottenere un elemento....51
Lavorare su un elemento....52
Spostarsi nella struttura ad albero....52
Creare nuovi elementi....53
Gli eventi nel DOM....53
Analogie con quanto già visto....54
L'oggetto event....54
Proprietà....54
Appendice....55
Parole riservate....55
Debugging....55
Come evitare alcuni errori comuni....55
Semplice debugging....55
Licenza....56

Premessa

Il JavaScript è un linguaggio di scripting orientato agli oggetti dalla sintassi molto simile ai linguaggi C e Java usato prevalentemente per il Web.

Finalità

Il libro si propone di portare il lettore da un livello zero ad una buona conoscenza del linguaggio JavaScript, insegnando a creare applicazioni web e pagine dinamiche.
La lettura del libro non richiede alcun prerequisito in campo di programmazione. È tuttavia necessario avere una conoscenza almeno di base dei linguaggi per il web HTML e CSS, in quanto il libro affronterà prevalentemente l'uso di JavaScript nel campo della programmazione web.

Libri correlati

- HTML
- CSS

Altri progetti

- **Wikipedia** contiene una voce riguardante **JavaScript**

Bibliografia

- Paul Wilton. *JavaScript. Guida per lo sviluppatore*. Milano, Hoepli Editore, 2001. ISBN 8820329204

Collegamenti esterni

- http://javascript.html.it (tutorial, guide, script già pronti e molto altro)
- (**EN**) ECMA 262 ECMAScript Language Specification (specifiche per ECMAScript)
- (**EN**) Reference for JavaScript 1.5 (specifiche per JavaScript 1.5)
- (**EN**) Innovators of the Net: Brendan Eich and JavaScript (Marc Andreesen, Netscape TechVision, 24 Jun 1998) (storia del JavaScript)

Introduzione

JavaScript è un linguaggio di scripting: nel suo utilizzo più frequente, quello della programmazione per il web, consiste in un linguaggio formale che fornisce al browser determinate istruzioni da compiere. Una pagina creata in HTML è infatti *statica*, in quanto una volta che la pagina è stata interpretata dal browser la disposizione degli elementi rimane immutata, così come il loro contenuto.
Tramite il JavaScript, invece, è possibile conferire dinamicità alle pagine web permettendo, ad esempio, di creare rollover sulle immagini, modificare i contenuti in base a input dell'utente o creare semplici applicazioni per il Web.

Breve storia di JavaScript

Il linguaggio JavaScript fu sviluppato inizialmente nel 1995 dalla Netscape Communications con il nome di **LiveScript** e incluso nel browser Netscape Navigator; il nome fu poi cambiato in JavaScript anche per l'assonanza con il linguaggio Java, che rappresentava una delle tecnologie più avanzate per l'epoca, con cui JavaScript non ha niente in comune, se non la sintassi simile.
Dopo il suo decollo e dato il successo di JavaScript, Microsoft decise di aggiungere al proprio browser Internet Explorer un proprio linguaggio di scriptng, **JScript**, che aveva però notevoli differenze con la versione sviluppata dalla Netscape. Nacque così la necessità di standardizzare il JavaScript e venne sviluppato lo standard **ECMAScript**.

Strumenti necessari

Gli unici strumenti necessari per la programmazione JavaScript per il Web sono un semplice editor di testi e un browser per vedere il proprio lavoro in azione.
Esistono comunque programmi che aiutano lo sviluppatore JavaScript fornendo un'evidenziazione della sintassi JavaScript o finestre di dialogo per velocizzare il lavoro.

Limitazione nell'uso di JavaScript

Una delle principali limitazioni di JavaScript è la possibilità che essi vengano facilmente disabilitati dall'utente tramite le impostazioni del browser. Questo è possibile poiché il JavaScript è un linguaggio lato client, lavora cioè sul computer dell'utente, che ha quindi tutto il diritto di disabilitare alcune funzionalità.
Per questo è meglio non demandare funzioni importanti come la gestione di dati sensibili a JavaScript bensì a linguaggi lato server come PHP o Perl.

Compatibilità tra browser

Un'altra grande limitazione all'uso dei JavaScript è la **compatibilità**: più che per la programmazione HTML o CSS, un programmatore JavaScript deve essere molto attento che il suo **lavoro** sia **compatibile con differenti browser** e **versioni più o meno recenti**.
Ad esempio, le due versioni parallele di JavaScript sviluppate dalla Microsoft per Internet Explorer e dalla Netscape (ora ereditata dalla Mozilla) hanno ancora oggi molte differenze: nonostante la sintassi fondamentale non cambi, molte funzionalità non sono disponibili o sono differenti a seconda del browser in uso. In questo wikibook si cercherà il più possibile di implementare soluzioni compatibili con Mozilla Firefox e Windows Internet Explorer; nei casi in cui ciò non sarà possibile, verranno presi eventuali accorgimenti, segnalando comunque le differenze.

I nostro primo programma

Inserire un JavaScript in una pagina HTML

Per inserire un codice JavaScript in una pagina HTML, è necessario semplicemente usare l'etichetta **`<script>`**.

L'attributo più importante di questo tag è *type*, che specifica il tipo di script usato: nel nostro caso il valore da inserire è `application/x-javascript` ma è anche usato, nonostante non sia standard, `text/javascript`; per la maggior parte del browser è tuttavia possibile omettere l'attributo `type` in quanto JavaScript è il linguaggio di scripting web predefinito.
Esiste anche, ma è sconsigliato rispetto all'uso di `type`, un attributo `language` per specificare sempre il linguaggio usato (es `language="JavaScript"`).

All'interno del tag `script` una volta, per problemi di compatibilità con i browser che non supportavano i JavaScript, il testo veniva delimitato da i delimitatori di commento. Ad esempio:

```
<script>
<!--
 ...codice...
-->
</script>
```

Questo accorgimento è tuttavia oggi sconsigliato, oltre ad essere praticamente superfluo in quanto la maggior parte dell'utenza utilizza browser JavaScript-compatibili.

È possibile inoltre usare l'attributo `src` per inserire un codice JavaScript esterno (solitamente un file con estensione .js). In questo caso si lascia vuoto lo spazio all'interno del tag. Ad esempio:

```
<script src="script.js"></script>
```

Il codice JavaScript può essere inserito sia nella sezione `head` che nella sezione `body` della pagina HTML, ma le istruzioni vengono comunque eseguite in ordine (a parte nei casi di funzioni che però vedremo più avanti nel corso del libro).

Hello, world!

Eccoci finalmente al nostro primo programma JavaScript: queste poche righe di codice identificano una pagina HTML su cui viene stampato *Hello, world!*. Create il seguente file e apritelo con il vostro browser:

```
<html>
      <head>
          <title>La mia prima applicazione JavaScript</title>
      </head>
      <body>
          <script>
              document.write("Hello, world!");
          </script>
      </body>
 </html>
```

Analizziamo l'unica riga di codice JavaScript presente nella pagina:

```
document.write("Hello, world!");
```

Quando il browser analizza la pagina, nel momento in cui incontra le istruzioni, se non indicato diversamente (come vedremo più avanti), le esegue secondo l'ordine nel quale sono indicate, ovvero in **sequenza** (questa operazione è chiamata *parsing*).
Le istruzioni sono sempre separate da un punto e virgola (;), e a JavaScript non interessano quindi i ritorni a capo.

In particolare, la riga di codice analizzata stampa sul documento il testo *Hello, world* tramite l'istruzione il metodo `write` dell'oggetto `document`. Per ora ci basti sapere che un *metodo* è un sottoprogramma associato esclusivamente ad un oggetto, che può richiedere uno o più parametri per funzionare al meglio. Il metodo `write` dell'oggetto `document` stampa nella pagina il testo indicato tra virgolette all'interno delle parentesi.

Cerchiamo di capire meglio come avviene il *parsing* di uno script JavaScript. Essendo infatti il browser piuttosto veloce a interpretare il codice, non è possibile ai nostri occhi notare l'esecuzione dello script, e sulla pagina compare subito il testo *Hello, world!*. Modifichiamo quindi il nostro codice come segue:

```
...
          <script>
              alert("Questo è un messaggio");
              document.write("Hello, world!");
              alert("Questo è un altro messaggio");
          </script>
          ...
```



Un alertbox in tedesco visualizzata dal browser Konqueror

Il metodo `alert` mostra una finestrella contenente il testo indicato tra parentesi e un pulsante "OK". Quando viene eseguito, il *parsing* della pagina si ferma fino a quando l'utente non clicca sul bottone OK; possiamo quindi vedere capire meglio come lavora il browser: l'esecuzione dello script si fermerà infatti due volte in corrispondenza delle due istruzioni `alert`. Quando viene mostrato il primo messaggio la pagina in secondo piano apparirà vuota, perché non vi è ancora stato impostato il contenuto; quando viene mostrato il secondo messaggio comparirà anche il testo *Hello, world!* in quanto sarà stata già eseguita il metodo `write`.

Inserire dei commenti

In JavaScript è possibile inserire dei commenti, ovvero porzioni di testo che verranno ignorate dal parser, di una o più righe delimitandoli da `/*` e `*/`. È possibile inoltre prevedere commenti di una sola riga utilizzando `//`. Ad esempio:

```
questo codice verrà interpretato
 /* questo verrà ignorato*/
 questo codice verrà interpretato
 /*
 questo verrà ignorato
 anche questo
 */
 questo verrà interpretato //e invece questo no
 //e questo neppure!
```

Tipi di dati e variabili

Un elemento necessario per la programmazione è la possibilità di salvare delle informazioni introdotte dall'utente o elaborate a partire da altre dal programma. In informatica, queste vengono chiamate **dati** i quali possono essere salvati in celle di memoria identificate da una **variabile**.

Tipi di dati

I dati in JavaScript possono essere di varie tipologie e a seconda della tipologia si potrà lavorare in modo diverso con essi. Il JavaScript è un linguaggio chiamato a **tipizzazione debole**, in quanto ogni volta che si fa riferimento ad un dato non è necessario specifiare il tipo, che viene attribuito automaticamente dal parser in base al contesto.

Numeri

I dati **numerici** possono essere positivi e negativi e si distinguono in **integer** (numeri interi) e **float** (numeri a virgola mobile).
Per convertire un valore qualsiasi in un valore numerico, JavaScript mette a disposizione due funzioni: `parseInt` e `parseFloat`. Per ora ci basti sapere che una *funzione* è un sottoprogramma (come un metodo) che non è associato ad alcun oggetto ma restituisce comunque un valore in base ad eventuali parametri accettati in ingresso. Per convertire i dati, ad esempio una stringa, queste funzioni analizzano carattere per carattere la stringa fornita come input prendendo in considerazione solo i numeri e, nel caso di parseFloat, anche il separatore decimale .. Ad esempio:

```
parseFloat("34acb") //restituisce 34.
```

```
parseInt("3eac34") //restituisce 3
```

Not a Number

Può tuttavia succedere che il valore passato alle funzioni di parsing dei numeri non possa essere elaborato come numero. In questo caso la funzione restituisce un valore particolare, **NaN**, che è l'acronimo per **N**ot **a** **N**umber, non un numero. È possibile testare se un valore è NaN usando la funzione `isNaN`

Stringhe

In informatica una **stringa** è una sequenza di uno o più caratteri alfanumerici. In JavaScript le stringhe si indicano inserendole all'interno di virgolette doppie (") o apici (') Bisogna però fare attenzione a chiudere una stringa con lo stesso apice con la quale è stata aperta; sono ad esempio stringhe valide:

```
"Hello, world ! 1234"
"Peter O'Toole"
```

ma non lo è ad esempio

```
'Peter O'Toole'
```

in quanto il parser analizzerebbe la stringa e, arrivato a `O'` penserebbe che la stringa si chiuda, senza sapere cosa sia `Toole'`.
È possibile anche indicare che caratteri come " e ' non indicano la fine del testo ma un carattere qualsiasi facendole precedere dal **carattere di commutazione** `\`. Ad esempio sono stringhe valide:

```
'Peter O\'Toole'
"Questo libro è depositato su \"it.wikibooks\""
```

In realtà ogni carattere è commutabile in una sequenza esadecimale usando la notazione `\xNN` dove NN è un numero esadecimale che identifica un carattere nel set Latin-1.

Dati booleani

Il tipo di dato **booleano** può assumere i soli valori `true` (vero) e `false` (falso). Il tipo di dato booleano è fondamentale per la selezione binaria e per il *decision-making*.
Quando è atteso un tipo di dato booleano il parser si comporta in maniere differenti:

- se viene fornito un numero, questo viene converitito in `false` se è 0, in `true` se è 1
- se viene fornito una stringa, questa viene converitito in `false` se è vuota (""), in `true` negli altri casi

Variabili

Una **variabile** in JavaScript identifica una cella di memoria nella quale vengono salvati durante l'esecuzione dello script i dati.

Quando si lavora con le variabili, è necessario per prima cosa indicare al *parser* il suo nome

utilizzando l'istruzione `var`:

```
var nome_variabile;
```

dichiara una variabile `nome_variabile`. Essendo JavaScript un linguaggio a tipizzazione debole, non è necessario indicare il tipo di dato della variabile, a meno che non si stia lavorando con oggetti (si vedrà più avanti). In questo caso la variabile appena dichiarata non avrà valore, non è stata cioè ancora **inizializzata**; è possibile però inizializzare la variabile direttamente insieme alla dichiarazione:

```
var nome_variabile = espressione;
```

dichiara una variabile con il valore di `espressione`. Per **espressione** si intende una sequenza di di operatori, variabili e/o dati che restituisca a sua volta una valore.

Quando si scelgono dei nomi per le variabili, si faccia attenzione ad alcune regole:

- JavaScript è case-sensitive (sensibile alle maiuscole): `var1` e `Var1` sono due variabili differenti
- in JavaScript esistono, come in ogni linguaggio formale, delle parole riservate, che identificano cioè particolari comandi per il parser. Non bisogna quindi usare parole riservate come nomi di variabili. Per l'elenco delle parole riservate in JavaScript, si veda l'appendice.
- una variabile non può contenere caratteri particolari (? : ; , . ecc...), tranne l'*underscore*, e non può iniziare con un numero

L'operazione fondamentale da usare con le variabili è l'**assegnazione**, che consiste nell'assegnare, nell'attribuire ad una variabile un valore. La sintassi è intuitiva e semplice:

```
nome_variabile = valore;
```

dove `valore` è un'espressione. Ad esempio:

```
var1 = "ciao!";
var2 = 3;
var3 = false;
var4 = var3; //attribuisce a var4 una copia del valore di var3
```

Operatori e casting

In questo modulo affronteremo l'uso degli **operatori** e della **conversione** (***casting***) per lavorare con i dati in JavaScript. Di ogni operatore verrà indicato i tipi di dati con i quali lavora e il tipo di dato restituito.

Operatori matematici

Questi operatori lavorano con valori interi o float e restituiscono sempre un valore numerico; sono piuttosto intuitivi in quanto corrispondono ai normali operatori algebrici. Sono:

- + e - (addizione e sottrazione algebrica)
- * e / (moltiplicazione e divisione algebrica)

Quando si lavora con gli operatori numerici è bene fare attenzione alla precedenza algebrica delle

operazioni. Vengono infatti valutate prima moltiplicazioni e divisioni e poi addizioni e sottrazioni. È tuttavia possibile "sfuggire" a questo vincolo tramite l'uso delle parentesi. In JavaScript non esistono distinzioni tra parentesi tonde, quadre o graffe, in quanto si usano solo quelle tonde.

Vediamo, ad esempio, come convertire un valore di temperatura espresso in gradi Fahrenheit al suo corrispondente in gradi Celsius. Inseriamo nella pagina HTML questo codice:

```
<script>
 var gradiF = prompt("Introdurre la temperatura in gradi Fahrenheit",100);
 var gradiC = 5/9 * (gradiF - 32);
 alert(gradiC);
</script>
```

Analizziamo riga per riga il listato JavaScript:

- la prima riga dichiara la variabile `gradiF` e le assegna il valore restituito dalla funzione `prompt`.



Un esempio di finestra prompt

Il metodo `prompt` visualizza una finestra di input che mostra il testo indicato come primo argomento e un valore di default indicato come secondo argomento (separato tramite la virgola). Nel nostro caso, la funzione mostrerà una finestra chiedendo di introdurre il valore della temperatura in gradi Fahrenheit proponendo come valore 100. L'utente è libero di modificare o meno questo valore e di premere uno dei due pulsanti OK o Annulla. Premendo OK, la funzione restituisce il valore immesso dall'utente, altrimenti restituisce 0.

- la seconda riga calcola il valore della temperatura in gradi Celsius usando l'equazione apposita
- la terza riga mostra una finestra alert (già vista nei precedenti moduli) contenente il valore della temperatura in gradi Celsius.

Si noti che il metodo `prompt` restituisce sempre un valore stringa ma, nel momento in cui la variabile stringa `gradiF` viene usata insieme ad operatori numerici, il suo valore viene automaticamente convertito in un valore numerico.
Possiamo vedere inoltre che, poiché il metodo `alert` richiede un valore stringa mentre noi abbiamo passato un valore numerico, JavaScript converte automaticamente il numero in una stringa contente esattamente il valore richiesto.

Nel comporre le espressioni è necessario tenere conto della precedenza delle operazioni (prima vengono valutate moltiplicazioni e divisioni, poi addizioni e sottrazioni). Nel comporre le espressioni possono risultare utili le parentesi; in questo caso è possibile usare solo parentesi tonde (JavaScript non supporta l'uso di parentesi quadre e graffe nelle espressioni). Ad esempio:

```
3 + 4 / 2 //restituisce 5
(3 + 4) / 2 //restituisce 3.5
(3 + (2 + 6) * 2) /2 //restituisce 9.5
```

Esistono poi due operatori chiamati **unari** in quanto lavorano su un solo dato numerico. Sono:

- ++ incrementa di uno il valore della variabile
- -- decrementa di uno il valore della variabile

Questo operatore può essere usato da solo oppure all'interno di un'espressione:

```
var1++ //corrisponde a var1 = var1 + 1
var1-- //corrisponde a var1 = var1 - 1
var2 = var1++ - 15 // corrisponde a var2 = var1 -15; var1 = var1 +1
```

È possibile inoltre nel caso di istruzioni come

```
var1 = var1 + n;
var2 = var2 / n
```

Usare la notazione:

```
var1 += n;
var2 /= n;
```

Operatori stringa

L'operazione più utilizzata quando si lavora con le stringhe è la **concatenazione**, che consiste nell'unire due stringhe accostandole una di seguito all'altra. L'operatore di concatenazioni in JavaScript è +. Vediamo alcuni esempi:

```
alert("Io sono " + "un utente di wikibooks!");
var nome = "Luigi";
alert("Io mi chiamo " + nome);
```

L'output di questo breve listato sono due finestre alert contenti il testo *Io sono un utente di wikibooks!* e *Io mi chiamo Luigi*. Si noti lo spazio prima delle virgolette. Possiamo quindi ampliare l'esempio precedente modificando l'ultima riga con:

```
alert(gradiF + " gradi Farhenheit corrispondono a " + gradiC + " gradi
Celsisus");
```

In questo caso la variabile `gradiC` viene convertita automaticamente in una stringa.

Operatori booleani

Sono chiamati operatori **booleani** o **di confronto** quelli che restituiscono un valore booleano e confrontano due valori dello stesso tipo di dato. Sono:

- < (minore)
- <= (minore o uguale)
- == (uguale)
- >= (maggiore o uguale)
- > (maggiore)

Il loro uso è strettamente legato ai dati numerici ed è intuitivo. Ad esempio:

```
3 > 4 //restituisce true
4 == 5 //restituisce false
```

È possibile inoltre comparare le stringhe. Ad esempio:

```
"wikibooks" ==  "wikibooks" //true
"wikibooks" ==  "it.wikibooks" //false
"wikibooks" ==  "Wikibooks" //false (distingue maiuscole e minuscole)
```

È possibile inoltre comporre le espressioni booleane usando gli operatori della logica, che lavorano con valori boolean e restituiscono a loro volta un valore boolean. Sono:

- && (corrisponde alla congiunzione logica, *et*)
- || (corrisponde alla disgiunzione logica, *vel*)
- ! (corrisponde alla negazione logica, *non*)

Anche in questo caso è possibile comporre le espressioni con le parentesi

```
(a > 0) && (a < 100)
```

valuta se a è compreso tra 0 e 100 esclusi.

Se JavaScript si aspetta un dato booleano e riceve un altro tipo di dato, converte in `false` le stringhe vuote e il numero 0, in `true` il resto.

Comporre le espressioni

Dopo aver analizzato gli operatori, è possibile comporre le **espressioni**. Un'istruzione è un insieme di uno o più valori legati da operatori che restituisce un valore; per comporre un'espressione possiamo usare quindi qualsiasi istruzioni restituisca un valore, ad esempio:

```
alert("Ciao, " + prompt("Come ti chiami?"));
```

è una espressione valida, in quanto la funzione `prompt` restituisce un valore; il risultato è una casella di prompt seguita da un alertbox che dà il benvenuto all'utente.

Le strutture di controllo

Selezione

La **selezione** è, insieme alla sequenza (già vista nei moduli precedenti) e al ciclo (che vedremo nel prossimo), una delle tre strutture fondamentali della programmazione e al suo livello più semplice consiste in una scelta tra due blocchi di istruzioni da seguire in base ad una condizione valutata inizialmente, che può essere vera o falsa.

Selezione semplice

È la forma più semplice di selezione. Vediamo la sua sintassi:

```
if (condizione) {
  istruzioni1
 } else {
  istruzioni2
 }
```

Quando il parser raggiunge questo listato, valuta il valore dell'espressione booleana `condizione`. Nel caso `condizione` restituisca `true` verranno eseguite le istruzione comprese tra la prima coppia di parentesi graffe (nell'esempio `istruzioni1`) altrimenti vengono eseguite le istruzioni comprese tra la seconda coppia di parentesi (`istruzioni2`). Vediamo un esempio:

```
var a = prompt("In che anno Cristoforo Colombo scoprì le Americhe?",2000);
 if (a == 1492) {
 alert("Risposta esatta!");
 } else {
 alert("Hai sbagliato clamorosamente...!");
 }
```

Il metodo già visto `prompt` chiede all'utente di inserire la data della scoperta delle Americhe. Nella riga successiva compare la nostra selezione: se `a` (ossia il valore inserito dall'utente) è uguale a 1492 allora viene mostrato un messaggio di complimenti; in caso contrario, viene mostrato un altro messaggio che informa l'utente dell'errore commesso. Si noti l'uso dell'operatore booleano == (uguale) che può restituire `true` o `false`.

È possibile, nel caso non sia necessario, omettere il blocco `else`; se inoltre il primo blocco di istruzioni è costituito da una sola riga e non c'è il blocco `else`, è possibile tralasciare le parentesi graffe.

Blocchi if annidati e condizioni multiple

All'interno di un blocco `if` è ovviamente possibile inserire a loro volta altri blocchi di selezione, che sono chiamati **annidati** (uno dentro l'altro). Si faccia però attenzione al seguente esempio:

```
if (cond1) {
  if (cond2) {
   istruzioni
  }
 }
```

Questo codice è inutilmente lungo, in quanto è possibile riassumerlo con un unico blocco usando gli operatori logici già visti in precedenza:

```
if (cond1 && cond2) {
  istruzioni
 }
```

else if

Esiste una forma particolare di `else` che permette di eseguire all'interno del suo blocco un'altra selezione. Vediamo questo codice:

```
if (cond1) {
  istruzioni;
 } else {
  if (cond2) {
   istruzioni
  } else {
   istruzioni;
  }
 }
```

Se all'interno del blocco `else` c'è una sola selezione è possibile usare questa sintassi più abbreviata

```
if (cond1) {
  istruzioni;
 } else if (cond2) {
  istruzioni
 } else {
  istruzioni;
 }
```

switch

L'istruzione `switch` è un tipo particolare di selezione che permette di verificare i possibili valori dell'espressione presa in considerazione. La sintassi è:

```
switch (espressione) {
  case val1:
   istruzioni1;
  case val2:
   istruzioni2;
 ...
 case val_n:
  istruzioni_n;
 default:
  istruzioni
 }
```

Quando il browser incontra questa istruzione, scorre tutti i valori `val1`, `val2`, ...`val_n` fino a che non incontra un valore uguale all'espressione indicata tra parentesi oppure un blocco `default`. In questo caso inizia ad eseguire il codice che segue i due punti. Se non c'è alcun blocco `default` e non ci sono valori che soddisfino l'uguaglianza, il parser prosegue eseguendo le istruzioni dopo la chiusura delle parentesi graffe.
Si faccia però attenzione a questo spezzone:

```
var a = 1;
 switch (a + 1) {
  case 1:
   alert("a = zero");
  case 2:
   alert("a = uno");
  case 3:
   alert("a = due");
  case 4:
   alert("a = tre");
  default:
   alert("a maggiore di 3");
 }
```

Si potrebbe supporre che l'output di questo codice sia solo una alertbox contenente il messaggio `a = 1` in quanto il blocco 2 soddisfa l'uguaglianza (`a + 1 = 2`). Non è tuttavia così: il parser dopo essersi fermato al blocco 2 proseguirà leggendo anche i blocchi successivi e il blocco default. Per evitare ciò dobbiamo inserire un'istruzione `break`. L'istruzione `break` indica al parser di interrompere la lettura della struttura `switch` e di proseguire oltre le parentesi graffe. L'esempio corretto è:

```
switch (a + 1) {
  case 1:
```

```
   alert("a = zero");
   break;
  case 2:
   alert("a = uno");
   break;
  case 3:
   alert("a = due");
   break;
  case 4:
   alert("a = tre");
   break;
  default:
   alert("a maggiore di 3");
 }
```

Ovviamente il `break` non è necessario per l'ultimo blocco.
Vediamo poi un altro listato:

```
switch (a + 1) {
  default:
   alert("a maggiore di 3");
   break;
  case 1:
   alert("a = zero");
   break;
  case 2:
   alert("a = uno");
   break;
  case 3:
   alert("a = due");
   break;
  case 4:
   alert("a = tre");
 }
```

Bisogna fare attenzione a non porre il blocco `default` in cima, in quanto il parser si fermerebbe subito lì e, incontrata l'istruzione `break`, salterebbe alla fine del blocco `switch` senza valutare le altre espressioni.

Vediamo un altro esempio di un semplice programma che converta un voto numerico in un giudizio:

```
var voto = prompt("Introduci il voto dell'alunno",6);
 var msg;
 switch (voto) {
 case "0":
  msg="Il compito non è stato consegnato (non classificabile)";
  break;
 case "1":
 case "2":
  msg="Insufficienza grave";
  break;
 case "3":
 case "4":
  msg="Insufficienza";
  break;
 case "5":
  msg="Sufficienza scarsa";
  break;
 case "6":
  msg="Sufficienza";
```

```
 break;
case "7":
 msg="Sufficienza piena";
 break;
case "8":
 msg="Buono";
 break;
case "9":
 msg="Discreto";
 break;
case "10":
 msg="Ottimo con lode";
 break;
default:
 msg="Dati non validi";
}
alert(msg);
```

Analizzando il codice, possiamo vedere che nei casi `voto` sia uguale ad esempio a 1 e 2 o a 3 e 4 viene eseguita la stessa operazione in quanto non c'è un `break` per ogni blocco `case`. Se nessuno dei valori soddisfa l'uguaglianza con `a` viene restituito un messaggio di errore

Operatore ternario

Esiste oltre agli operatori già menzionati nel capitolo precedente anche un cosiddetto **operatore ternario**, che lavora con tre valori. La sua sintassi è:

```
condizione ? esp1 : esp2
```

Quando il parser incontra questa notazione, valuta il valore booleano di `condizione`. Se è vero, restituisce il valore di `esp1` altrimenti quello di `esp2`. Questo permette di creare semplici selezioni; ad esempio:

```
var anni = prompt('Quanti anni hai?', 20);
var msg = "Ciao, vedo che sei " + (anni >= 18 ? "maggiorenne" : "minorenne") +
"!";
alert(msg);
```

In questo caso l'operatore ternario restituisce `"maggiorenne"` se `anni` è maggiore o uguale a 18, altrimenti restituisce `"minorenne"`.

Iterazione

L'**iterazione** è una struttura di controllo (come la selezione) che permette l'esecuzione di una sequenza di una o più istruzioni fino al verificarsi di una data condizione.

Cicli con condizione

La forma più semplice di ciclo è composta da una condizione valutata inizialmente e ad un blocco di istruzioni eseguito sino a quando la condizione iniziale non risulti false. In JavaScript usiamo in questo caso un ciclo `while` la cui sintassi è:

```
while (condizione)
{
```

```
 istruzioni;
}
```

Quando JavaScript incontra questa struttura:

1. valuta il valore booleano di `condizione`. Possiamo avere quindi due casi:
    1. se è vero, esegue il blocco di istruzioni delimitato dalle parentesi graffe e quindi ricomincia dal punto 1
    2. se è falso, salta al punto 2
2. prosegue la lettura delle istruzioni successive

Si noti che:

- è possibile che il blocco di istruzioni non venga mai eseguito. Questo è possibile se il valore della condizione iniziale è falso già dalla sua prima valutazione
- se il blocco di istruzioni non modifica in qualche modo le variabili che intervengono nella condizione iniziale, può accadere che questo venga ripetuto all'infinito

Esiste poi la possibilità di porre la condizione alla fine del blocco di istruzioni; in questo caso si è sicuri che questo verrà eseguito almeno una volta. La sintassi è:

```
do
{
 istruzioni;
} while (condizione)
```

Vediamo alcuni esempi; miglioramo ad esempio il piccolo programma che valutava se l'utente era maggiorenne o minorenne:

```
var anni;
do
{
 anni = prompt("Quanti anni hai?",20);
} while (isNaN(anni))
var msg = "Ciao, vedo che sei " + (anni >= 18 ? "maggiorenne" : "minorenne") +
"!";
alert(msg);
```

In questo modo abbiamo inserito un controllo per verificare che il valore immesso dall'utente sia un numero: la richiesta dell'età viene infatti ripetuta se il valore introdotto **non è** un numero.

Cicli con contatore

JavaScript, come molti altri linguaggi programmazione, offre la funzionalità di cicli con un contatore che viene incrementato o decrementato ad ogni ripetizione. La sintassi è:

```
for (condizione_iniziale; condizione_finale; incremento_contatore) {
 istruzioni;
}
```

Ad esempio:

```
for (var i = 1; i <= 10; i++) {
 alert("i = " + i);
}
```

Eseguendo il codice appena visto otterremo come output dieci messaggi che indicano il valore di `i` ad ogni iterazione.

È possibile inoltre fare in modo che il contatore venga incrementato di più di un'unità al secondo:

```
for (var i = 1; i <= 20; i+=2) {
 alert("i = " + i);
}
```

Questo codice mostra sempre i primi 10 numeri pari.

Se l'operazione da eseguire è molto semplice, come nel primo esempio che abbiamo visto, è possibile anche usare una sintassi abbreviata:

```
for (var i = 1; i<= 10; alert("i = " + i++));
```

L'istruzione di alert contiene infatti un operatore unario "++" applicato alla variabile contatore `i`; ad ogni ripetizione del ciclo, JavaScript mostra prima il messaggio di *alert* e poi incrementa la variabile.

Iterare sugli elementi di un array

Per iterare in modo automatico gli elementi di un array (funzionalità trattata più avanti nel corso di questo libro), esiste un ciclo particolare, la cui sintassi è:

```
for (indice in nomeArray) {
 //...
}
```

L'array assegna man mano ad `indice` i valori dell'array indicato.

Funzioni definite dall'utente

Come molti linguaggio di programmazione, anche JavaScript da la possibilità di creare le proprie funzioni personalizzate.

Come è stato già accenato in precedenza, una funzione è un sottoprogramma identificato da una sequenza di caratteri che può accettare o meno nessuno o più parametri e può restituire un valore.

Creare le proprie funzioni: una panoramica

La sintassi per la creazione di una nuova funzione è la seguente:

```
function nome_funzione (arg1, arg2, argN...) {
    ''codice''
}
```

Ad esempio:

```
function benvenuto (nome) {
    alert("Benvenuto, " + nome);
}
```

Se vogliamo eseguire il codice contenuto nella funzione dobbiamo **richiamarla** (o **invocarla**). Ad esempio:

```
var n = prompt("Come ti chiami?");
if (n != "")
 benvenuto (n);
```

Con questo breve spezzone chiediamo all'utente di inserire il nome e, se non risponde con una stringa vuota e non ha premuto Annulla, gli porge il benvenuto (si noti che il metodo `prompt` restituisce una stringa vuota se l'utente fa clic su "Annulla").

Usare i parametri

Come abbiamo visto, è possibile prevedere che l'utente possa passare alcuni parametri alla funzione. Dando una sguardo alla funzione di benvenuto creata precedentemente, vediamo che il parametro `nome` della funzione diventa poi automaticamente una variabile; se quando viene chiamata la funzione viene omesso, il parametro assumerà un valore nullo (che diventa 0 per i numeri, una stringa vuota per i valori alfanumerici, falso per i valori booleani).

Impostare un valore di ritorno

Impostare un valore di ritorno di una funzione è molto semplice, basta seguire la sintassi:

```
return valore;
```

Quando incontra l'istruzione `return`, JavaScript interrompe l'esecuzione della funzione e restituisce il valore indicato. Ad esempio:

```
function somma (a, b)
{ //una semplice funzione
    return a+b;
}

var c = somma(3,5); //c assumerà il valore 8
somma(4,12); //in questo caso non succede nulla
```

Si noti che una funzione può essere richiamata solamente dopo che è già stata inserita nel programma. Un codice come questo genererebbe errore:

```
var c = somma(3,5); //la funzione somma non esiste ancora!

function somma (a, b)
{ //una semplice funzione
    return a+b;
}
```

Oggetti

Il concetto di **oggetto** è molto importante nella programmazione in JavaScript. In questo modulo verranno spiegate le caratteristiche comuni degli oggetti; nei moduli seguenti verranno invece trattati nel dettaglio gli oggetti intrinsechi di JavaScript.

L'uso degli oggetti e delle loro funzionalità entra a far parte del paradigma della **programmazione**

orientata agli oggetti (abbreviata **00P**, **O**bject **O**riented **P**rogramming)

Cosa sono gli oggetti

Per avvicinarci al concetto di oggetto in informatica, possiamo pensare al concetto di oggetto nel mondo reale.
Per creare un nuovo oggetto è necessario partire da un modello (in informatica una **classe**) che indichi come creare ogni oggetto di quella tipologia (ogni oggetto è un'**istanza** della suddetta classe).
Per fare un esempio concreto, ciascun oggetto macchina viene costruita in base a dei progetti che ne definiscono la struttura.

Gli oggetti possono inoltre possedere delle caratteristiche (**proprietà**): nel caso della nostra macchina, saranno la cilindrata, le dimensioni, il costo, ecc...).

Ciascuna istanza espone inoltre la possibilità di effettuare delle operazione su di essi (**metodi**): per la nostra macchina, metterla in moto o guidare. Queste operazioni modificheranno delle caratteristiche come il livello del suo carburante o la velocità.

Una volta introdotto il concetto di oggetto, dobbiamo avere però la capacità di astrarre: gli oggetti dell'informatica non corrispondono a quelli della realtà; hanno però numerosi vantaggi, tra i quali la possibilità di trattare dati più complessi di numeri e stringhe.

Gli oggetti in JavaScript

JavaScript permette di creare le proprie classi personalizzate; tuttavia noi lavoreremo solo su quelle predefinite.
Per creare un nuovo oggetto è necessario associarlo ad una variabile usando la sintassi:

```
var nome_oggetto = new nome_classe ();
```

In questo modo la variabile `nome_oggetto` sarà l'unico modo per fare riferimento all'istanza di `nome_classe` appena creata.

Le classi possono per la creazione dell'oggetto prevedere anche l'uso facoltativo di un **costruttore**, cioè un metodo che imposta automaticamente alcune proprietà dell'oggetto. La sintassi in questo caso è:

```
var nome_oggetto = new nome_classe (parametri_del_costrutture);
```

Per fare un esempio con la nostra macchina:

```
var la_mia_macchina = new macchina ("Utilitaria", "Rosso");
```

Ovviamente la classe macchina conterrà un costruttore che prevede come parametri il tipo e il colore della macchina.

Proprietà

Possiamo pensare ad una proprietà come ad una variabile associata al singolo oggetto; il suo valore

viene attribuito inizialmente dal costruttore (se viene usato) e successivamente viene modificato agendo sull'oggetto (operando sui metodi, ecc...). Per fare riferimento alla proprietà (per il recupero o per l'assegnazione) si usa la sintassi:

```
nome_oggetto.nome_proprietà
```

Alcune proprietà possono essere di sola lettura, cioè il loro valore può essere letto ma non modificato.

Con la nostra macchina:

```
alert('La mia macchina è lunga ' + la_mia_macchina.lunghezza + ' m!');
```

Come si potrà pensare, la proprietà `lunghezza` è di sola lettura.

Metodi

Un metodo è una funzione associata al singolo oggetto e definita nella sua classe; se nella classe di un oggetto è prevista una funzione `metodo_esempio` sarà possibile eseguire la funzione tramite la sintassi:

```
nome_oggetto.metodo_esempio () //ricordarsi le parentesi anche se non passiamo
parametri!
```

Dal momento che le funzioni possono prevedere un valore di ritorno, sarà possibile inserire la notazione vista precedentemente all'intero di un'espressione.

Ad esempio, con la nostra macchina:

```
la_mia_macchina.rifornisci (20) //20 euro di benzina
```

La struttura with

Quando si lavora con gli oggetti, può risultare comodo il costrutto `with`, che permette di accedere più volte ad un oggetto senza dover ogni volta specificare il suo nome. Ad esempio:

```
with (la_mia_macchina) {
 .rifornisci (20) //20 euro di benzina
 alert(.lunghezza); //ricordarsi l'uso del punto!
 altra_macchina.rifornisci(50); //per riferirmi ad altri oggetti devo indicare
il loro nome
} //qui si conclude il blocco with
```

Oggetto String

L'oggetto **String** permette di effettuare numerose operazioni sulle stringhe quali ricerca, isolamento di un carattere e altro ancora.

Costruttore

Per creare un nuovo oggetto della classe String usiamo la sintassi:

```
var nome_variabile = new String(stringa);
```

Il costruttore prende come parametro la stringa che sarà manipolata nel corso dello script.

L'oggetto String è un po' particolare, in quanto, come vi sarete già accorti, corrisponde ad un "doppione" del tipo di dato primitivo stringa, analizzato precedentemente nel corso del libro.
Dal momento che JavaScript effettua automaticamente la conversioni dei dati quando necessario, la differenza è dal punto di vista pratico, in quanto:

- se creiamo una stringa con la normale sintassi

```
var variabile = "testo";
```

  e successivamente volessimo trattarla come un oggetto String usando metodi o proprietà della classe String, JavaScript converte automaticamente la stringa in un oggetto String

- se abbiamo un oggetto String e volessimo recuperare la sua stringa, JavaScript converte automaticamente l'oggetto String in una stringa contente la stringa indicata nel costruttore.

Proprietà

L'oggetto String dispone di una proprietà rilevante, la proprietà `length` che restituisce il numero di caratteri contenuti in una stringa:

```
var esempio = "Sono una stringa primitiva"
alert (esempio.length); //restituisce 26
```

Nell'esempio appena visto, nella seconda riga la variabile `esempio` viene convertita in un oggetto String per accedere alla proprietà `length`.
Sempre per quanto appena detto, potremmo semplicemente scrivere:

```
alert ("Sono una stringa primitiva".length); //restituisce 26
```

Metodi

I metodi della classe String permettono di eseguire molteplici operazioni sulle stringhe; si noti che questi metodi lavorano sulla stringa contenuta nell'oggetto ma restituiscono il valore desiderato senza modificare il valore dell'oggetto stringa.

toLowerCase() e toUpperCase()

Questi due metodi restituiscono la stringa forzando la capitalizzazione o tutta minuscola o tutta minuscola. Attenzione:

```
var t1 = new String ("TeStO");
var t2 = t1.toLowerCase() //restituisce "testo"
var t3 = t1.toUpperCase() //Restituisce "TESTO"
// attenzione: alla fine di questo codice la variabile t1 contiene ancora
"TeStO"!
```

charAt() e charCodeAt()

Il metodo `charAt()` restituisce il carattere della stringa che si trova alla posizione specificata; il primo carattere è identificato dalla posizione 0. Ad esempio:

```
var testo = prompt("Inserisci un testo", "Hello, world!");
var ultimoCarattere = testo.charAt(testo.length - 1);
alert (ultimoCarattere);
```

Questo semplice codice estrapola dalla stringa fornita in input dall'utente l'ultimo carattere. Per fare ciò recupera il carattere alla posizione `testo.length - 1`: infatti, dal momento che il conteggio dei caratteri parte da 0, nel caso di "Hello, world!" avremo queste posizioni:

0	1	2	3	4	5	6	7	8	9	10	11
H	e	l	l	o	,	w	o	r	l	d	!

Quindi, nonostante la stringa sia composta da 12 caratteri, l'ultimo si trova alla posizione 11, ovvero 12 - 1.

Il metodo `charCodeAt()` funziona come `charAt()` ma invece del carattere restituisce il suo codice Ascii.

fromCharCode()

Il metodo `fromCharCode()` è l'opposto di `charCodeAt()`: infatti prende come argomento un numero qualsiasi di numeri che vengono interpretati come codici Ascii e trasformati in una stringa. Questo metodo è però un po' particolare, in quanto non necessita di lavorare su un oggetto String; per questo viene detto **statico**. Per utilizzare il metodo sarà sufficiente usare la sintassi:

```
String.fromCharCode(parametri);
```

facendo cioè riferimento alla classe String.
Vediamo un esempio:

```
var stringa;
for (codice = "0".charCodeAt(0); codice <= "9".charCodeAt(0); codice ++ ) {
 stringa = stringa + String.fromCharCode(codice);
}
```

Questo semplice *snippet* scorre dal codice del carattere "0" a quello del carattere "9" creando così la stringa:

```
0123456789
```

indexOf() e lastIndexOf()

Uno dei metodi dell'oggetto String che useremo più di frequente sarà `indexOf()` la cui sintassi è:

```
oggetto_string.indexOf(search, start);
```

In pratica, il metodo cerca nella stringa la prima occorrenza della sottostringa `search` e ne restituisce la posizione (a partire da 0); se la stringa non viene trovata, restituisce -1. Il parametro opzionale `start` specifica la posizione dalla quale iniziare la ricerca (di default è 0). Ad esempio;

```
var stringa = "Ma la volpe col suo balzo ha raggiunto il quieto fido"
//ricordiamoci che JS converte automaticamente le stringhe
var posiz = stringa.indexOf("v"); //contiene il valore 6)
var posiz2 = stringa.indexOf("k"); // restituisce -1
```

Il metodo `lastIndexOf()` funziona analogamente ad `indexOf()` ma inizia la ricerca dalla fine della stringa e non dall'inizio

substr()

Il metodo `substr()` presenta la sintassi:

```
nome_oggetto_string.substr(start, length)
```

Il metodo estrae dalla stringa indicata una sottostringa a partire dalla posizione indicata dal parametro `start` un numero `length` di caratteri. Se quest'ultimo parametro non è specificato, il metodo include anche tutti i caratteri a partire da quello iniziale fino alla fine della stringa. Ad esempio:

```
var stringa = "Questo è un esempio";
var str2 = stringa.substr(0, 6); //restituisce "Questo"
var str3 = stringa.substr(7); //restituisce "è un esempio"
```

replace()

Il metodo `replace()` restituisce la stringa iniziale sostituendo a tutte le occorrenze della stringa indicata come primo parametro quella fornita come secondo. Ad esempio:

```
var s = "L'utente Pinco ha modificato questa pagina"
var s2 = s.replace("Pinco", "Pallino");
//s2 ora contiene "L'utente Pallino ha modificato questa pagina"
//s1 contiene ancora "L'utente Pinco ha modificato questa pagina"
```

Con questo metodo è possibile contare il numero di occorrenze di una sottostringa in una stringa. Ecco una funzione:

```
function substrCount (string, subString) {
 return (string.length - (string.replace(subString, "")).length) /
subString.length
}
```

La funzione elimina tutte le occorrenze della sottostringa nella stringa tramite il metodo `replace()`; poi sottrae la lunghezza della stringa principale a quella della stringa appena ottenuta; in questo modo si ottiene il numero di caratteri occupati dalla sottostringa che, divisi per la sua lunghezza, ne danno il numero di occorrenze.

split()

Il metodo `split()` viene presentato ora in quanto fa parte della classe String, ma il suo uso richiede la conoscenza degli array, che verranno esaminati più avanti nel corso del libro.
La sintassi è:

```
stringa.split(separatore);
```

Il metodo restituisce un array contenente le diverse sottostringhe in cui la stringa è divisa dal separatore. Ad esempio:

```
"a,e,i,o,u".split(","); //restituisce "a", "e", "i", "o", "u"
"a,e,i,o,u,".split(","); //attenzione: restituisce "a", "e", "i", "o", "u", ""
```

Oggetto Math

L'oggetto **Math** mette a disposizione dello sviluppatore JavaScript numerose funzioni e costanti matematiche; è un oggetto particolare, in quanto non richiede di lavorare su istanze di oggetti ma permette di accedere ai suoi metodi usando la notazione:

```
Math.nome_metodo();
Math.nome_proprietà
```

Proprietà

Le proprietà dell'oggetto Math consistono in diverse costanti matematiche:

- `PI`: restituisce il valore approssimato di Pi greco
- `E`: restituisce il valore della costante matematica E
- `LN2` e `LN10`: il valore del logaritmo naturale di 2 e di 10

Ovviamente sono tutte proprietà di sola lettura

Metodi

Ecco una lista dei metodi corrispondenti alle principali funzioni matematiche e trigonometriche:

- `abs()`: restituisce il valore assoluto (o modulo) del numero fornito come argomento
- `min()` e `max()`: restituiscono il minimo e il massimo tra i due numeri passati come parametri
- `sqrt()` e `pow(b, n)`: restituiscono la radice quadrata del numero passato o l'ennesima potenza del numero `b`
- `sin()`, `cos()` e `tan()`: restituiscono il seno, il coseno e la tangente trigonometrica dell'angolo passato in radianti
- `log()`: restituisce il logaritmo naturale in base  del numero passato come parametro

Arrotondare i numeri

Per l'arrotondamento l'oggetto Math mette a disposizione tre metodi:

- `ceil()`: arrotonda il numero al numero intero maggiore più vicino (per eccesso). Ad esempio:

```
Math.ceil(10.01) //restituisce 11
Math.ceil(-6.3) //restituisce -6
```

- `floor()`: arrotonda l'oggetto al numero intero minore più vicino (per difetto). Ad esempio:

```
Math.floor(10.01) //restituisce 10
Math.floor(-6.3) //restituisce -7
```

- `round()`: arrotonda l'oggetto per eccesso se la parte decimale è maggiore o uguale a 5, altrimenti arrotonda per difetto:

```
Math.round(10.01) //restituisce 10
Math.round(-6.3) //restituisce -6
Math.round(2.5) //restituisce 3
```

Tutte queste tre funzioni differiscono da `parseInt()` che non arrotonda i numeri, bensì li tronca. Ad esempio:

```
Math.parseInt(10.01) //restituisce 10
Math.parseInt(-6.3) //restituisce -6
Math.parseInt(2.5) //restituisce 2
```

Generare numeri casuali

L'uso di Math permette anche di generare numeri (pseudo) casuali tramite il metodo:

```
Math.random()
```

che restituisce un numero casuale decimale compreso tra 0 (incluso) e 1 (escluso). Nella pratica non è molto utile se non viene trasformato in un numero intero. Ad esempio questo codice simula il lancio di un dado:

```
var numLanci = prompt ("Quanti lanci vuoi effettuare?", 10);
var i; var lancio;
for (i = 0; i < numLanci; i++) {
 lancio = Math.floor(Math.random() * 6 + 1);
 alert("Lancio " + (i + 1) + ": " + lancio);
}
```

La riga che ci interessa è quella dell'assegnazione della variabile `lancio`. Per capire l'algoritmo usato, ragioniamo sul risultato che vogliamo ottenere.
Il numero che vogliamo ottenere (che chiameremo *d*) deve essere un numero intero compreso tra 1 e 6 inclusi. Dal momento che il generatore restituisce numeri tra 0 incluso e 1 escluso, dovremo:

- moltiplicare per sei il numero casuale ottenuto (che chiameremo *n*): in questo modo possiamo ottenere un numero che va da 0 (se $n = 0$, allora $d = 0$) fino a sei escluso. Infatti *d* potrebbe essere uguale a 6 solo se *n* potesse essere uguale a 1, cosa che non è possibile
- sommare uno al valore ottenuto: in questo modo *d* sarà compreso tra 1 (incluso) e 7 (escluso)
- arrotondare il valore per difetto: in questo modo *d* sarà un numero intero compreso tra 1 e 6; infatti, dato che il 7 è escluso, arrotondando per difetto potremo ottenere al massimo 6.

Un esempio pratico

Di seguito viene mostrata un'applicazione pratica dei metodi dell'oggetto Math per la creazione di una funzione che arrotondi le cifre alla *n*esima cifra decimale indicata.

```
function arrotonda (num, dec, mod) {
 var div = Math.pow(10, dec);
 switch (mod) {
  case 'e':
  case 'E': //eccesso
   return Math.ceil(num * div) / div;
  break;
  case 'd':
  case 'D': //difetto
   return Math.floor(num * div) / div;
  break;
  case 't':
  case 'T': //troncamento
   return parseInt(num * div) / div;
  break;
  case 'a':
  case 'A':
  default: //arrotonda
   return Math.round(num * div) / div;
 }
}
//per testarla usiamo queste righe di codice...
alert(arrotonda(3.43, 1, 't'));
alert(arrotonda(3.42, 1, 'e'));
alert(arrotonda(3.469, 2, 'd'));
alert(arrotonda(3.427, 2, 'e'));
alert(arrotonda(3.55, 1, 'a'));
alert(arrotonda(3.46, 1, 't'));
```

La funzione appena mostrata arrotonda il numero fornito come primo parametro alla cifra decimale fornita come secondo parametro usando diversi metodi di arrotondamento. Il tutto sta nel fatto che le funzioni di Math agiscono solo sui numeri interi e non su quelli decimali.
Per effettuare gli arrotondamenti, quindi, spostiamo la virgola di *n* posizioni decimali fornite come parametro ed eseguiamo l'arrotondamento desiderato; successivamente, spostiamo di nuovo la virgola all'indietro. Per spostare la virgola è sufficiente moltiplicare e dividere per una potenza di dieci: per fare ciò ci serviamo del metodo `Math.pow()`

Oggetto Date

L'oggetto **Date** permette di gestire in modo semplice e veloce le date; nella pratica, un istanza della classe Date identifica una data e le funzionalità della classe permettono di aggiungere o sottrarre anni, mesi o giorni o recuperarne il valore.

Costruttore

Il metodo più semplice per creare una data è:

```
var data = new Date()
```

Nel caso non si passino valori al costruttore, verrà creata un oggetto Date contente la data e l'ora corrente.
Altrimenti possiamo creare date precise con altri metodi:

- fornendo il numero di millisecondi passati dalla mezzanotte del primo gennaio 1970 (questo metodo è macchinoso ma è quello che più si avvicina al metodo con cui il computer ragiona con le date)

- fornendo una stringa che indichi la data, ad esempio:

```
var data = new Date("10 July 2007");
var data2 = new Date("10-7-2007");
```

Usando il metodo descritto nel secondo esempio, bisogna fare attenzione in quando, essendo JavaScript eseguito con le impostazioni del client, per un utente USA invece che il 10 luglio 2007 la data sarà 7 ottobre 2007

- l'ultimo metodo è il migliore in quanto affidabilità e praticità messe assieme. La sintassi è:

```
var data = new Date(anno, mese, giorno, ora, minuti, secondi, millisecondi)
```

Crea un oggetto data in base ai parametri passati; normalmente è possibile omettere l'ultimo parametro. Attenzione, gennaio è il mese 0 e non 1 e di conseguenza anche gli altri cambiano.

Metodi

Recuperare i valori della data

- `getDay()`: restituisce un numero da 0 a 6 corrispondente al giorno della settimana della data (Domenica è 0, Lunedì 1, ... Venerdì 6)
- `getDate()`: restituisce il giorno del mese
- `getMont()`: restituisce il numero del mese (Gennaio è 0, Dicembre è 11)
- `getFullYear()`: restituisce il numero dell'anno con quattro cifre.

Visti questi metodi, possiamo già creare una funzione che restituisca una stringa contente la data formattata. La funzione prende come parametro un oggetto Date e formatta la data in esso contenuta:

```
function formattaData(data) {
 var giornoS = data.getDay();
 var giornoM = data.getDate();
 var mese = data.getMonth();
 var anno = data.getFullYear();

 switch (giornoS) { //converte il numero in nome del giorno
  case 0: //domenica
   giornoS = "domenica";
   break;
  case 1:
   giornoS = "lunedì";
   break;
  case 2:
   giornoS = "martedì";
   break;
  case 3:
   giornoS = "mercoledì";
   break;
  case 4:
   giornoS = "giovedì";
   break;
  case 5:
   giornoS = "venerdì";
   break;
  case 6: //sabato
```

```
   giornoS = "sabato";
   break;
 }

 switch (mese) { //converte in numero in nome del mese
  case 0:
   mese = "gennaio";
   break;
  case 1:
   mese = "febbraio";
   break;
  case 2:
   mese = "marzo";
   break;
  case 3:
   mese = "aprile";
   break;
  case 4:
   mese = "maggio";
   break;
  case 5:
   mese = "giugno";
   break;
  case 6:
   mese = "luglio";
   break;
  case 7:
   mese = "agosto";
   break;
  case 8:
   mese = "settembre";
   break;
  case 9:
   mese = "ottobre";
   break;
  case 10:
   mese = "gennaio";
   break;
  case 11:
   mese = "novembre";
   break;
 }
 //es. martedì 9 luglio 2007
 return giornoS + " " + giornoM + " " + mese + " " + anno;
}
```

Possiamo usare la funzione in questi modi:

```
var d1 = new Date () //oggi
var d2 = new Date (2007, 6, 10);
document.write(formattaData(d1));
document.write(formattaData(d2));
```

La funzione potrebbe sembrare inutile data l'esistenza del metodo dell'oggetto Date `toLocaleString()` che restituisce una stringa contenente la data formattata secondo le impostazioni locali dell'utente; tuttavia, per questo motivo, ad esempio, un utente inglese invece che `martedì 9 luglio 2007` otterrebbe `Tuesday, Jule 9, 2007`.

Impostare i valori della data

L'oggetto Date permette inoltre di modificare la data memorizzata nell'oggetto Date tramite i metodi `setDate()`, `setMonth` e `setFullYear`. Questi metodi sono molto utili, perché hanno la caratteristica di ragionare come è giusto fare per le date. Ad esempio:

```
var d = new Date(2007, 3, 4); //4 aprile 2007
d.setDate(31);
```

In questo esempio impostiamo come giorno del mese di aprile il valore di 31, nonostante aprile conti solo 30 giorni. JavaScript lo sa e per questo porta la data al 1 di maggio. Questa funzionalità è spesso usata insieme ai metodi per ottenere le date:

```
d.setMonth(d.getMonth() - 6); //sei mesi prima
```

Lavorare con l'ora

L'oggetto Date mette a disposizione funzioni analoghe anche per gestire le ore:

- `getHours()`, `getMinutes()`, `getSeconds()` e `getMilliseconds` restituiscono rispettivamente l'ora, i minuti, i secondi e i millisecondi dell'oggetto Date. Ad esempio:

```
var d = new Date(2007, 3, 4, 23, 45, 6, 12); //4 aprile 2007, ore 23:45:6 (11
pm)
d.getSeconds(); //restituisce 6
```

Il metodo `getMilliseconds` permette anche di cronometrare l'esecuzione di una porzione di codice o il caricamento della pagina HTML. Inseriamo in testa alle istruzioni da cronometrare il codice:

```
var dataPrima = new Date(); //adesso
//memorizza secondi e millisecondi
var sPrima = dataPrima.getSeconds();
var msPrima = dataPrima.getMilliseconds();
```

In questo modo salviamo i secondi e i millisecondi del momento in cui inizia il caricamento; poi inseriamo in fondo alla pagina:

```
var dataDopo = new Date(); //adesso
//memorizza secondi e millisecondi
var sDopo = dataDopo.getSeconds();
var msDopo = dataDopo.getMilliseconds();

document.write('Pagina caricata in ' + (sDopo - sPrima) + ' secondi e ' +
Math.abs(msDopo - msPrima) + ' millisecondi');
```

Gestire i fusi orari

Tutte le funzioni che abbiamo visto lavorano sull'ora dell'utente o su quella impostata dalla pagina web. Non abbiamo però considerato che Internet è una rete che coinvolge centinaia di paesi e quasi tutti i fusi orari del nostro pianeta; per questo motivo JavaScript mette a disposizione delle funzioni basate sull'**UTC** (**Universal Time Coordination**), una convenzione precentemente nota come **GMT** (**Greenwich Mean Time**) che rappresenta la data basandosi su quella del meridiano 0, passante per la famosa località di Greenwich. Ovviamente anche queste funzioni sono basate su una corretta configurazione del computer del client.

I metodi UTC ricalcano i metodi per il locale; per impostare l'ora UTC è necessario ad esempio usare il metodo `set`**`UTC`**`Hour`; esiste inoltre un metodo `toUTCString` che restituisce l'ora formattata secondo le convenzioni UTC e un metodo `getTimeZoneOffset` che restituisce in minuti la differenza di ora tra il fuso di Greenwich e quello dell'utente (può essere anche negativo).

Oggetto Array

Last but not the least esaminiamo gli array, una funzionalità presente nella maggior parte dei linguaggi di alto livello, che in JavaScript è disponibile tramite l'uso della classe Array.

Cos'è un array

In informatica, un array (o vettore) è un tipo di dato strutturato (non semplice, che dispone quindi di altri parametri) che permette di organizzare i dati secondo un indice; in pratica, lo stesso oggetto contiene a sua volta numerosi valori, ciascuno dei quali contrassegnato da una chiave numerica. Questo concetto può essere rappresentato da un tabella a singola entrata:

0	**1**	**2**	**3**	**..**
valore_0	valore_1	valore_2	valore_3	...

Gli array in JavaScript

Come è stato già detto, per creare un vettore in JavaScript facciamo riferimento alla classe Array:

```
var vettore = new Array (); //crea un array vuoto
```

Il costruttore della classe può essere tuttavia usato in maniere differenti:

```
var vettore = new Array (5); //crea un array contenente 5 elementi
var vocali = new Array ("A", "E", "I", "O", "U"); //crea un array contente le
vocali
var lettere_straniere = ["J", "K", "W", "X", "Y"]; //metodo breve
```

Per accedere ad un elemento dell'array, per leggerlo o modificarlo, usiamo la notazione:

```
vettore[indice]
```

Ad esempio, facendo riferimento alla variabile `vocali`, avremo questa tabella:

Indici	**0**	**1**	**2**	**3**	**4**
Valori	A	E	I	O	U

e potremo lavorare in questo modo:

```
alert(vocali[0]); //mostra "A"
vocali[1] = "ciao"
alert(vocali[1]); //mostra "ciao"
```

Da notare che la chiave dell'array è numerica e parte da 0 e che il valore di ciascun elemento può essere di qualsiasi tipo di dato, ciascuno diverso dall'altro.
Ad esempio, potremmo stabilire un metodo per memorizzare le informazioni su dei prodotti

decidendo che all'elemento 0 corrisponde il nome, all'elemento 1 il modello, ecc... creando diversi array:

```
var descCampi = new Array ("ID", "Modello", "Prezzo"
var prod1 = new Array (1, "3000 plus", 35);
var prod2 = new Array (5, "4000 minus, 12);
```

Proprietà

L'oggetto Array dispone di una sola proprietà interessante, `length`, che restituisce il numero di elementi in esso contenuti (contando anche quelli vuoti); da notare che, poiché gli indici partono da 0, un array di 5 elementi avrà gli elementi con le chiavi 0, 1, 2, 3 e 4.

Metodi

L'uso dell'oggetto Array diventa utile in relazione all'uso dei metodi.

concat()

Il metodo `concat()` restituisce un altro array contente tutti gli elementi dell'array a cui è applicato seguiti da tutti gli elementi dell'array passato come parametro. Ad esempio:

```
var a = new Array (1, 2, 3);
var b = new Array (4, 5, 6);
var c = a.concat(b); //1, 2, 3, 4, 5, 6
```

Da notare che `a` contiene ancora solo gli elementi 1, 2 e 3.

sort()

Il metodo `sort()` ordina l'array secondo l'ordine alfabetico:

```
var a = new Array ("e", "a", "u", "i", "o");
a.sort() /ora le vocali sono ordinate nell'array a
//attenzione: lavora direttamente sull'array!
```

Attenzione: in JavaScript le minuscole seguono alle maiuscole, quindi ad esempio

```
var a = new Array ("E", "a", "U", "i", "o");
a.sort() /ora le vocali sono ordinate in questo modo:
//E U a i o!
```

Per ovviare a questo problema possiamo lavorare su stringhe solo minuscole, usando il metodo `toLowerCase` in questo modo:

```
var a = new Array ("E", "a", "U", "i", "o"); //il nostro array
for (var i = 0; i < a.length, i++) { //itera sui singoli elementi
 a[i] = a[i].toLowerCase(); //si rende minuscoli
}
```

reverse()

Questo metodo agisce sull'array invertendo l'ordine degli elementi in esso contenuti. Ad esempio:

```
var vettore = new Array (5); //crea un array contenente 5 elementi
var vocali = new Array ("A", "E", "I", "O", "U");
vocali.reverse(); //ora contiene U O I E A
```

slice()

Il metodo `slice()`, infine, serve per copiare porzioni di array. La sua sintassi è:

```
arr.slice(a, b)
```

Il metodo restituisce un array contente gli elementi di `arr` compresi tra `a` (incluso) e `b` (escluso). Se `b` non è indicato, vengono copiati tutti gli elementi a partire da `a` fino alla fine. Ad esempio:

```
var vettore = new Array (5); //crea un array contenente 5 elementi
var a = new Array ("A", "E", "I", "O", "U");
var b = a.slice(1,4); //b contiene gli elementi E I O
```

Iterare sugli elementi di un array

Come abbiamo visto precedentemente, è possibile iterare su tutti gli elementi di un array con un ciclo particolare; ad esempio, volendo azzerare il valore di tutti gli elementi di un array, possiamo usare il codice:

```
var a = new Array ("A", "E", "I", "O", "U");
for (indice in a) {
 a[indice] = ""; //annulla il valore
}
//ora l'array contiene 5 elementi tutti vuoti
```

Cookie

Una funzionalità molto interessante messa a disposizione da JavaScript è la possibilità di **gestire i cookie**.
In informatica, i cookie sono piccoli file di testo memorizzati sul computer dell'utente che contengono informazioni salvate dai siti web. La comodità dei cookie sta quindi nella possibilità di memorizzare in modo **permanente** delle informazioni univoche rispetto ad un utente; hanno tuttavia lo svantaggio di poter essere eliminati e disabilitati dall'utente.
Ciascun cookie è composto da alcuni parametri, tra i quali:

- **nome**: un nome identificativo per il cookie
- **valore**: il valore da memorizzare
- **scadenza** (*expiration date*): è opzionale, stabilisce la data di scadenza del cookie, cioè la data dopo la quale questi vengono eliminati dal disco rigido dell'utente.

Impostare i cookie



la funzione `ImpostaCookie` visualizzata dall'editor Bluefish

In JavaScript per impostare i cookie usiamo la proprietà `cookie` dell'oggetto `document` passandole come valore una stringa contenente i vari parametri separati da punto e virgola. Ecco ad esempio una semplice stringa di cookie:

```
NomeUtente=Ramac;expires=Tue, 28 August 2007 00:00:00
```

Questa stringa imposta un cookie NomeUtente al valore Ramac con scandenza 28 Agosto 2007. Potremo salvare questo cookie con l'istruzione

```
document.cookie="NomeUtente=Ramac;expires=Tue, 28 August 2007 00:00:00"
```

La sintassi è quindi:

```
NomeCookie=Valore;parametri
```

Nel caso si volesse inserire nella stringa del valore un punto e virgola, è necessario usare la funzione `escape()` che converte tutti i caratteri particolari nel corrispondente esadecimale nel set di caratteri Latin-1. Questo vale anche per gli spazi, gli apostrofi, le virgole, ecc....

Per comodità, è possibile creare una funzione che crei i cookie partendo da tre parametri:

```
function impostaCookie (nome, valore, scadenza) {

    if (scadenza == "") {
        var oggi = new Date();
        oggi.setMonth(oggi.getMonth() + 3);
        //restituisce la data nel formato necessario
        scadenza = oggi.toGMTString();
    }
    valore = escape(valore);
    document.cookie=nome + "=" + valore + ";expires=" + scadenza;
}
```

Ottenere i cookie

Per ottenere i valori dei cookie relativi al proprio documento è necessario accedere alla proprietà `document.cookie` che restituisce una stringa contenente coppie di nome-valore di ciascun dato separate da un punto e virgola e uno spazio. Ad esempio:

```
NomeUtente=Ramac; UltimaVisita=Thu, 30 August 2007 13:34:12
```

Possiamo notare due punti importanti:

- mentre è possibile impostare valori come la data di scadenza del cookie, non è possibile ottenerli una volta modificati
- i dati per una comoda lettura necessitano di una manipolazione stringa

Per la lettura dei cookie, può risultare utile questa funzione che resituisce il valore di un cookie a partire dal suo nome:

```
function valoreCookie (nome) {
   var valore=document.cookie; //ottiene la stringa di cookie
   var inizioCookie=valore.indexOf(" " + nome + "="); //trova il cookie
desiderato

   if (inizioCookie == -1) { //se non esiste, magari è all'inizio della stringa
      inizioCookie = valore.indexOf(nome + "=");
   }

   if (inizioCookie == -1) { //il cookie non esiste proprio
      valore = null;
   }

   if (inizioCookie >= 0) //il cookie esiste
      inizioCookie = valore.indexOf("=", inizioCookie) + 1; //qui inizia la
stringa del valore
      var fineCookie = valore.indexOf(";", inizioCookie); //qui finisce
      if (fineCookie == -1)  //se non viene trovato, allora è l'ultimo cookie
         fineCookie = valore.length;
      valore = unescape(valore.substring(inizioCookie, fineCookie)); //elimina i
caratteri commutati
   }

   return valore;
}
```

Verificare se i cookie sono attivi

Come è già stato accennato in precedenza, uno degli svantaggi dei cookie risiede nel fatto che **possono essere facilmente disabilitati** dall'utente; è pertanto necessario sviluppare soluzioni alternative ai cookie, soprattutto se sono parte essenziale della propria applicazione web.

Il modo più semplice per verificare se i cookie sono attivati è quello di crearne uno fittizio e verificare se è possibile ottenerne il valore. La seguente è una funzione che restituisce `true` se i cookie sono abilitati:

```
function cookieAttivi () {
    ris = false; //imposta il risultato a falso
    impostaCookie("testCookie", "test"); //crea il cookie fittizio
    if (valoreCookie("testCookie") == "test") { //se esiste
        ris = true; //allora i cookie sono abilitati
    }
    return ris;
}
```

Timer

Un'interessante funzionalità offerta da JavaScript nonché indispensabile per rendere una pagina dinamica in molte occasioni, è quella dei cosidetti ***timer***.
Grazie ad essi possiamo far effettuare un'istruzione ad intervalli regolari o dopo un numero prestabilito di secondi.

One-shot timer

Per **one-shot timer**, o **timeout**, si intendono quei timer che permettono di eseguire un'istruzione dopo un determinato numero di millisecondi. Il metodo `setTimeout` dell'oggetto `window` prende come parametri una stringa contente l'istruzione da eseguire e il numero di secondo dopo i quali deve essere eseguita. Ad esempio:

```
var tim = setTimeout('alert("ciao!")', 3000); //l'oggetto window può essere
sottointeso
```

Inserendo questa istruzione al caricamento della pagina, dopo 3 secondi (3000 millisecondi) verrà visualizzato un messaggio di alert.

Attenzione: i timeout **non interrompono** il normale flusso degli script. Ad esempio:

```
var tim = setTimeout('alert("ciao!")', 3000);
alert("secondo alert");
```

In questo caso verrà mostrato prima il secondo alert in quanto l'esecuzione del primo è ritardata di tre secondi.
Nella stringa è possibile anche inserire una funzione, ovviamente.

Il metodo setTimeout restituisce un ID numerico che identifica univocamente ciascun timer impostato; questo è utile in relazione al metodo `clearTimeout()` che elimina il timeout impostato con il metodo visto precedentemente. Questo può servire per permettere all'utente di fermare un timer già iniziato. Ad esempio:

```
var tim = setTimeout('alert("la bomba è esplosa")', 3000); //la bomba esploderà
tra 3 secondi
var pwd = prompt("se inserisci la password corretta potrai disinnescare la bomba
prima che esploda....");
if (pwd == "disattiva") {
  clearTimeout(tim); //disinnesca il timer (la "bomba")
}
```

Impostare intervalli regolari

Diversi dai timeout sono quei timer che impostano azioni da eseguire ad intervalli regolari. Ad esempio:

```
var tim = setInterval ("aggiornaPagina()", 1000);
```

Con questa istruzione facciamo sì che la funzione `aggiornaPagina` venga eseguita ogni secondo. Come per i timeout, possiamo eliminare i timer ricorrendo ad un'apposita funzione:

```
clearInterval(tim);
```

che interrompe l'esecuzione del timer precedentemente avviato.

BOM

Arriviamo finalmente a scoprire le funzionalità di JavaScript che lo rendono il linguaggio di scripting per le pagine web tra i più diffusi: la possibilità di interagire con il browser e la pagina HTML.

Tutto questo è possibile grazie all'**oggetto `window`** e, più generalmente, a quello che è il ***BOM*** (***Browser Object Model***), il *modello oggetto del browser*: per esso si intendono l'oggetto `window` tutti gli oggetti che da essi dipendono.

L'oggetto window

L'oggetto `window` rappresenta la finestra del browser che contiene la pagina stessa; nella pratica, questo vuol dire avere accesso, per esempio, alle dimensioni della finestra del browser, al testo della sua barra di stato, e molto altro.

L'oggetto `window` è un **oggetto globale**, il che significa che non è necessario specificarlo per utilizzare i suoi metodi: ne è un esempio l'uso che abbiamo sempre fatto del metodo `window.alert()`, che abbiamo sempre richiamato senza specificare l'oggetto al quale apparteneva.

Proprietà

defaultStatus e status

Questa proprietà permette di ottenere o di impostare il testo predefinito della barra di stato del browser. Per esempio, sulla maggior parte dei browser, mostra la scritta "Completato" dopo l'avvenuto caricamento di una pagina.
Per usarla vi accediamo semplicemente:

```
window.defaultStatus = "Testo personalizzato";
//oppure anche
defaultStatus = "Testo personalizzato";
```

La seconda proprietà, `status`, indica invece il testo *corrente* nella barra di stato, ad esempio l'`href` di un link ipertestuale se vi sta passando sopra il mouse. Anche qui la sintassi è semplice:ù

```
window.status = "Testo personalizzato";
```

frames, length e parent

Queste tre proprietà sono legate all'uso di *frame* nelle pagine web.
Essendo l'uso dei frame una tecnica piuttosto obsoleta, sorpassata oramai da tecnologie lato server e deprecata dal W3C, il loro uso non sarà approfondito nel corso di questo libro.

opener

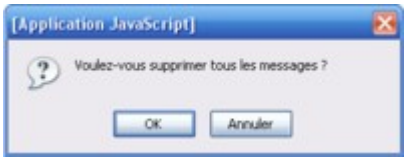
Questa proprietà restituisce un riferimento all'oggetto `window` che ha aperto la finestra corrente (si veda più avanti il metodo open()).

Oggetti come proprietà

La maggior parte delle proprietà dell'oggetto `window` sono a loro volta altri oggetti. I più importanti e utilizzati saranno trattati nel dettaglio più avanti in questo modulo e nei prossimi capitoli.

Metodi

alert(), confirm() e prompt()



Una finestra di confirm, questa volta in francese

Due di questi metodi, `alert()` e `prompt()`, sono già stati trattati in precedenza; è simile invece l'uso del metodo `confirm()`, in quanto anch'esso mostra una finestra di dialogo. La sua sintassi è:

```
window.confirm("testo");
```

Il metodo mostra una finestra di dialogo che mostra il testo indicato come parametro e due pulsanti, *OK* e *Annulla* (*Cancel* in inglese), e restituisce `true` se l'utente clicca su OK, `false` negli altri casi.
Questo metodo serve per chiedere conferme all'utente prima di effettuare operazioni lunghe o importanti.

blur() e focus()

Questi due metodi rispettivamente tolgono il focus dalla finestra del browser e spostano il focus sul browser. Questo serve quando ad esempio sono aperte più finestre contemporaneamente.

open()

Questo metodo apre una nuova istanza del browser, apre cioè una nuova finestra del browser in uso. La sua sintassi è:

```
window.open("url", "titolo", "parametri")
```

I primi due parametri sono, come si può capire, indicano rispettivamente l'indirizzo in cui si aprirà la nuova finestra e il titolo che questa avrà. È possibile inoltre aprire una finestra vuota passando come URL una stringa vuota ("").
Il terzo parametro è invece una stringa di parametri costruita con la sintassi:

```
nomeParametro=valore;nomeParametro2=valore2;
```

I principali parametri sono (non sono tutti compatibili con entrambi i browser):

Nome parametro	Possibili valori	Descrizione
`copyHistory`	`yes, no`	Copia o meno la cronologia della finestra padre nella finestra figlia
`width`, `height`, `top`, `left`	numero intero	Indicano rispettivamente le dimensioni e la posizione sullo schermo della finestra
`location`	`yes, no`	Indica se mostrare la barra dell'indirizzo
`menubar`	`yes, no`	Indica se mostrare la barra dei menu
`resizable`	`yes, no`	Abilita il ridimensionamento della finestra
`scrollbars`	`yes, no`	Mostra/nasconde le barre di scorrimento laterali
`status`	`yes, no`	Mostra la barra di stato
`toolbar`	`yes, no`	Mostra la barra degli strumenti

La funzione restituisce un riferimento all'oggetto window appena aperto. Ad esempio:

```
finestra = window.open("pagina.html", "Nuova finestra",
"toolbar=no;location=yes");
finestra.defaultStatus = "Benvenuti nella nuova finestra!";
```

Possiamo anche creare una generica funzione, per poi creare nuove finestre partendo dai link:

```
function apriFinestra(url, titolo, width, height) {
var params = "width:" + width + ";height=" + height;
window.open(url, titolo, params);
}
```

e poi per usarla la inseriamo per esempio in un link:

```
<a href="javascript:apriFinestra('pagina.html', 'Nuova finestra',
200,200)">Apri</a>
```

In questo modo, grazie alla parola "javascript:", cliccando sul link verrà eseguita la funzione appena creata.

Come è stato spiegato prima, dal codice della finestra aperta si potrà ottenere un riferimento alla finestra che la ha aperta tramite la proprietà `window.opener`.

close()

Chiude una finestra del browser.

Gli eventi nel BOM

Prima di completare la descrizione degli oggetti del BOM, ci occuperemo finalmente di un modo per rendere la nostra pagina HTML veramente dinamica: l'utilizzo degli **eventi**.

Un evento in JavaScript permette di eseguire determinate porzioni di codice a seconda delle **azioni**

dell'utente o a **stati della pagina**: è possibile ad esempio collegare del codice al *click* di un oggetto oppure al ridimensionamento della pagina.

Gli eventi in JavaScript

Di seguito è elencata una lista dei possibili eventi verificabili in JavaScript; da notare che gli eventi si riferiscono sempre ad un oggetto specifico (successivamente sarà spiegato come).

Eventi del mouse

- **onclick** e **ondblclick** si verificano quando l'utente fa click o doppio click sull'elemento in questione
- **onmousdown** e **onmouseup** si verificano quando l'utente schiaccia il pulsante del mouse e quando lo rilascia
- **onmouseover** e **onmouseout** si verificano quando l'utente sposta il mouse dentro l'oggetto in questione e quando lo sposta fuori
- **onmousemove** si verifica quando l'utente muove il cursore dentro l'oggetto

Eventi della tastiera

- **onkeypress** si verifica quando l'utente preme un tasto sulla tastiera quando il *focus* è sull'oggetto specificato
- **onkeydown** e **onkeyup** si verificano quando l'utente schiaccia un tasto e lo rilascia

Eventi degli elementi HTML

- **onfocus** e **onblur** si verificano quando l'utente sposta il focus sull'oggetto in questione e quando toglie il focus dall'oggetto
- **onchange** si verifica quando l'utente modifica il contenuto dell'oggetto (è applicabile solo ai campi di modulo)
- **onsubmit** e **onreset** si possono applicare solo a moduli HTML e si verificano quando l'utente invia il form (pulsante di *submit*) o quando lo resetta (pulsante *reset*)
- **onselect** si verifica quando l'utente selezione del testo

Eventi della finestra

- **onload** e **onunload** si verificano quando la pagina viene caricata o quando viene chiusa
- **onresize** si verifica quando l'utente ridimensiona la finestra del *browser*
- alcuni browser supportano anche l'evento **onscroll** che si verifica allo scrolling della pagina

Eventi come attributi HTML

Il modo più semplice di definire un evento è quello di inserirlo come **attributo agli elementi HTML** della pagina stessa. Ad esempio:

```
<img src="logo.png" width="100px" onclick="alert('hai fatto click
sull\'immagine!')" alt="Clicca per saluti" />
```

Nell'esempio associamo il codice JavaScript che mostra l'alert al click dell'utente sull'immagine HTML.

Ovviamente possiamo anche fare riferimento a funzioni o variabili dichiarate precedentemente nel corso della pagina HTML:

```
<img src="logo.png" width="100px" onclick="salutaUtente()" alt="Clicca per
saluti" />
```

Tramite questo sistema possiamo usufruire dell'oggetto `this` per riferirci all'elemento HTML a cui è stato applicato. Ad esempio:

```
<img src="logo.png" width="100px" onmouseover="this.src='logo2.png'"
onmouseout="this.scr='logo.png'" alt="rollover" />
```

Con questa riga di HTML e qualche istruzione JavaScript creiamo quello che si chiama un *rollover*, cioè l'effetto di cambiamento dell'immagine quando il mouse passa sopra di essa.

🔍 *Per approfondire, vedi la pagina **JavaScript/Oggetto document**.*

Eventi come proprietà

Gli eventi possono essere anche impostati **tramite il BOM**, cioè tramite JavaScript. Ad esempio:

```
window.onload = "alert('pagina in caricamento...');";
```

Questo può risultare comodo per aggiungere eventi quando non si possono modificare direttamente gli oggetto HTML, ad esempio nel caso di script esterni creati per essere utilizzati più volte.

Attenzione a non incappare in questo tipo di errori:

```
function saluta () {
 alert("pagina in caricamento");
};
window.onload = saluta(); //non funziona
```

In questo codice l'intenzione è quella di mostrare un avviso al caricamento della pagina; scrivendo `saluta()`, però, il browser lo valuta come funzione; nel nostro caso, poiché tale funzione non restituisce alcun valore, ad caricamento della pagina non succederà nulla. Per ottenere il risultato desiderato è necessario utilizzare la sintassi:

```
window.onload = saluta; //ore funziona
```

In questo modo facciamo riferimento non al risultato della funzione bensì al suo codice vero e proprio.

Un altro metodo consiste nell'assegnare alla funzione un **oggetto `function`** creato sul momento:

```
window.onload = function () {
 alert("pagina in caricamento");
};
```

Restituire dei valori

Quando si collega del codice ad un evento, è possibile impostare un **valore di ritorno**: se questo è `false`, verrà annullata l'**azione predefinita**:

```
<a href="pagina.htm" onclick="return confirm('vuoi veramente seguire il
link?');">cliccami!</a>
```

Cliccando sul link verrà mostrato un messaggio di conferma; se si clicca il pulsante *annulla* la funzione restituirà false e di conseguenza restituirà false il codice collegato all'evento; in questo modo si elimina l'azione predefinita per l'evento click su un link, cioè non verrà caricata la pagina.

Questa funzionalità serve soprattutto per la **convalida dei form**: si collega all'evento `onsubmit` del form una funzione che restiutisce `false` nel caso di errori nella compilazione dei campi. Ovviamente non si può fare completo affidamento su questa funzionalità, in quanto i JavaScript possono essere facilmente disabilitati dall'utente.

Oggetto document

Oltre alle proprietà elencate nel modulo precedente, l'oggetto globale `window` espone altre importanti proprietà, tra le quali **l'oggetto `window.document`**.
Questo oggetto permette di ottenere un riferimento al documento HTML e agli elementi in essa contenuti.

Metodi

L'oggetto espone solo pochi metodi:

- **open()** e **close()** rispettivamente aprono e chiudono un flusso di informazioni per l'uso dei metodi elencati sotto
- **write()** e **writeln()** scrivono nella pagina la stringa (formattata in HTML) passata come parametro:
    - se la pagina è già stata caricata:
        - se è stato aperto il flusso di informazioni, il testo verrà inserito alla fine della pagina
        - se il flusso non è stato aperto, il testo sostituirà il testo della pagina
    - se la pagina non è ancora stata caricata, se si usando blocchi <script> all'interno del <body>, le informazioni si inseriscono nel punto in cui è presente lo script

Ad esempio:

```
<body>
  <p>Testo<br />
  <script type="application/x-javascript">
    document.write('<a href="pagina.html">link</a>');
  </script>
  Testo</p>
</body>
```

genererà il codice:

```
<body>
  <p>Testo<br /><a href="pagina.html">link</a>
  Testo</p>
</body>
```

Proprietà

Le proprietà dell'oggetto document possono essere relative:

- agli attributi dell'elemento <body> della pagina
- al contenuto vero e proprio del corpo del documento
- altre proprietà del documento HTML

Nel primo gruppo, troviamo proprietà come:

- `fgColor` e `bgColor`: indicano rispettivamente il colore di primo piano e di sfondo della pagina. Corrispondono agli attributi `text` e `background`.
- `linkColor`, `alinkColor`, `valinkColor`: il colore rispettivamente dei link allo stato normale, dei collegamenti attivi e dei collegamenti visitati. Corrisponde agli attributi HTML `link`, `alink` e `vlink`.

Tra le proprietà del documento HTML la più rilevante è `document.title`, che permette di accedere al titolo del documento (etichetta <title&t;.

Accedere ai link nella pagina

Il primo attributo che ci permette realmente di accedere agli elementi della pagina è l'array `links` che restituisce un riferimento agli oggetti <a> presenti nel documento:

```
var l = document.links[0]; //restituisce un riferimento al primo link nella
pagina
alert('Nella pagina ci sono ' + document.links.length + ' link'); //ricordiamoci
che è un array
```

Ciascun oggetto link presenta le seguenti proprietà:

- `href`: restituisce o imposta il valore dell'attributo `href` del link. A partire da esso derivano alcuni attributi relativi all'URL a cui punta il link:
  - `hostname`: indica il nome dell'host dell'URL
  - `pathname`: indica il percorso oggetto indicato dell'URL
  - `port`: indica la porta dell'URL
  - `protocol`: indica il protocollo della risorsa a cui punta l'URL (per esempio FTP o HTTP)
  - `search`: restituisce l'eventuale *querystring* (ad esempio `pagina.htm?` **`name=Mario&cognome=Rossi`**)
- `target`: il valore dell'attributo `target` del link

Ad esempio possiamo decidere di impostare il target di tutti i link nella pagina a `_blank`:

```
for (i in document.links) {
 document.links[i].target = "_blank"; //imposta l'attributo target
}
```

L'array images

Analogamente all'array `links`, esiste anche un array `images` che permette di accedere a tutte le

immagini presenti nella pagina.
Rispetto a `links`, questo array presenta una differenza: è possibile accedere agli elementi della pagina anche attraverso il loro attributo `name`. Ad esempio:

```
<img name="logo" src="logo.png" width="130px" height="130px" />
```

Possiamo accedere a quell'immagine tramite l'indice numerico oppure tramite il nome, in questo modo:

```
document.images["logo"]
```

L'oggetto `image` espone come proprietà gli attributi di qualsiasi immagine HTML, come `src`, `border`, ecc... Presenta inoltre un'interessate proprietà `complete` che restituisce true se l'immagine è già stata caricata dal browser.

Accedere ai moduli

 *Per approfondire, vedi la pagina **HTML/Moduli**.*
Per ultimo trattiamo, in quanto introduce un argomento piuttosto vasto, l'array `document.forms[]` che, come si potrà facilmente intuire, permette di accedere a tutti gli elementi <form> nella pagina (moduli HTML).

Per accedere a ciascun form nella pagina possiamo utilizzare tre metodi:

```
document.forms[0] //accede al primo form nella pagina
document.forms["dati"] //accede al form con name="dati"
document.dati //forma abbreviata
```

Il metodo presenta due interessanti metodi:

- **reset()** che resetta i valori dei campi nei moduli (come cliccare sul pulsante "reset" del form)
- **submit()** invece invia il form (come cliccare sul pulsante "submit" o "invia" del modulo)

Questo da la possibilità di inviare il modulo per esempio al semplice click di un collegamento oppure di aniticipare il normale invio dei dati:

```
<form name="opz">
    <select name="opzione" onchange="document.opz.submit()">
        <option value="modifica">modifica</option>
        <option value="elimina">elimina</option>
        <option value="...">...</option>
    </select>
</form>
```

In questo modo il form si invia automaticamente quando l'utente sceglie un'opzione dal menu a discesa, senza l'utilizza del pulsante di invio.

L'oggetto presenta anche alcune proprietà corrispondenti agli attributi dell'oggetto HTML <form>, quali **action**, **encoding**, **method**, **name** e **target**.

I campi dei moduli

Analizziamo infine, tra le proprietà dell'oggetto form, l'utile array `elements[]` che permette di

accedere a tutti i campi contenuti form:

```
document.dati.elements[0]  //accede al primo elemento del form
document.dati.elements["opzione"]  //accede al campo con name="opzione"
document.dati.opzione //forma abbreviata
```

L'oggetto restituito da questo array dipende dal tipo di oggetto a cui accede: potrà essere un campo di testo, un pulsante oppure una *textarea*. Tuttavia tutti i campi condividono:

- una proprietà **form** che restituisce un rimando al form genitore;
- una proprietà **value** che permette di ottenere o impostare al valore del campo, in modo diverso a secondo dei singoli tipi di campo;
- la proprietà **type** che ne specifica il tipo;
- la propreità **name** che ne restituisce il nome (attributo HTML `name`;
- due metodi **focus()** e **blur()** che rispettivamente attribuiscono e tolgono il *focus* dall'elemento

Caselle di testo

Questi elementi sono creati utilizzando il tag HTML `<input>` impostando l'attributo `type` ai valori *text*, *password* o *hidden*: la differenza è che il secondo mostra degli asterischi al posto dei caratteri reali e il terzo è invisibile, ma noi li tratteremo insieme in quanto condividono proprietà e metodi.

Questi tre tipi di moduli restituiscono come attributo **type** il valore rispettivamente *text*, *password* e *hidden*. Hanno poi due proprietà, **size** e **maxlength**, che corrispondono ai rispettivi attributi HTML.

Per quanto riguarda i metodi, invece, presentano **select()**, che seleziona l'intero testo contenuto nel campo (ovviamente non serve per i campi di testo nascosti).

L'evento associato normalmente a questi campi è **onchange**, che viene chiamato dopo la modifica del valore nel campo.

Aree di testo

Le aree di testo sono create tramite l'utilizzo del tag HTML `<textarea>` e sono rappresentante dall'omonimo oggetto *textarea*in JavaScript.

Questi oggetti presentano proprietà analoghe ai campi di testo: **value** imposta o restituisce il valore della casella di testo, **rows** e **cols** corrispondono ai rispettivi attributi HTML.
L'oggetto *textarea* permette di inserire anche degli accapo nel testo, ma in questo caso bisogna stare attenti, in quanto il carattere di accapo è espresso diversamente a seconda del sistema operativo in uso: su Windows è `\n\r`, su Linux è `\n` mentre su Mac è `\r` (`\` è il carattere di commutazione visto in precedenza per gli apici singoli e doppi).

Pulsanti

Per quanto riguarda i pulsanti dei moduli, questi possono essere di tre tipi: *submit*, *reset* e *button*; tuttavia espongono lo stesso modello oggetto, che in realtà è molto semplice, in quanto non presenta proprietà o metodi particolari oltre a quelli condivisi da tutti i campi di modulo.

L'utilizzo di JavaScript con questi pulsanti è più che altro legato agli eventi del mouse (analizzati

nel modulo precedente).

Pulsanti di opzione

Iniziamo ad analizzare ora alcuni elementi più complicati: i *checkbox* e i pulsanti *radio* (corrispondono rispettivamente ad `<input type="checkbox" />` e `<input type="radio" />`).

Nel primo la proprietà più utile non è *value* come negli altri casi, bensì **checked** che indica se la casella è stata marcata o meno.

Per quanto riguarda i *radio*, il discorso è simile, ma bisogna anche considerare che diversi elementi di opzione normalmente condividono lo stesso attributo `name`; per accedere quindi ai singoli elementi è necessario quindi appoggiarsi ad un array:

```
// supponiamo di avere 5 opzioni con name="età"
document.dati.età[1].checked=true; //seleziona la seconda opzione
alert(document.dati.età.value); //restituirà il valore del campo
                                // cioè il valore della casella selezionata
alert(document.dati.età[3].value); //restituisce il valore del quarto elemento
```

Liste

Analizziamo per ultimo gli elementi `<select>`, che servono per creare liste e menu a tendina. Prendiamo il seguente spezzone di codice HTML in un ipotetico form chiamato "dati":

```
<select name=città size=5>
 <option value=mi>Milano</option>
 <option value=to>Torino</option>
 <option value=pe>Pescara</option>
 <option value=pa>Palermo</option>
</select>
```

Questo codice crea un oggetto *select* accessibile come sempra con la sintassi `document.dati.città`. Questo espone alcune proprietà interessanti:

- **value** è il valore del campo del modulo (nel nostro caso potrà essere *mi*, *to*, ecc...)
- **option** è un'array contenente un riferimento a ciascun oggetto *option* appartenente all'elenco
- **selectedIndex** restituisce l'indice dell'elemento al momento selezionato dall'utente (Milano sarà 0, Torino 1, ecc...)

Ogni oggetto *option* espone le seguenti proprietà:

- **value** indica il suo valore
- **text** indica il testo che mostrerà all'utente
- **index** restituisce la sua posizione nell'array.

Ad esempio, associamo all'evento *onchange* del campo "città" alla seguente funzione:

```
function mostraCittà() {
 var testo = document.dati.città.option[document.dati.città.selectedIndex].text;
}
```

Cliccando su ciascuna opzione otterremo così un'alertbox contenente la città selezionata.

Inserire e aggiungere elementi

Per inserire o aggiungere nuovi elementi è sufficiente creare un nuovo oggetto *option* e assegnarlo ad una posizione dell'array:

```
var nuovaOpzione = new Option("Bologna", "bo");
document.dati.città.options[0] = nuovaOpzione;
```

L'esempio appena visto aggiunge la città "Bologna" come primo elemento della lista. Il browser si occuperà poi di far scalare le posizioni degli altri elementi dell'array. Per eliminare un'opzione è sufficiente assengare all'elemento dell'array il valore *null*:

```
document.dati.città.options[2] = null;
```

La seguente funzione aggiunge una città come ultimo elemento nella lista:

```
function aggiungiCittà () {
 var testo = prompt("Che città vuoi aggiungere?");
 var sigla = prompt("Inserisci la sigla della città");

 var nuovaCittà = new Option(testo, sigla);
 document.dati.città.options[document.dati.città.options.length] = nuovaCittà
}
```

Altri oggetti del BOM

Oltre all'oggetto document, l'oggetto window espone altre proprietà che restituiscono oggetti particolarmente utili.

location

Questo oggetto restituisce informazioni relative all'indirizzo della pagina caricata.

Le sue proprietà sono le stesse che per l'oggetto `link` trattato nel modulo precedente
Espone inoltre di due interessanti metodi:

- **reload()**: ricarica la pagina
- **replace()**: carica l'URL indicata come parametro sostituendola in cronologia alla voce corrente. Usando invece `location.href = url` vengono salvate entrambe le pagine in cronologia.

history

Questo oggetto permette di accedere alla cronologia delle pagine visitate dall'utente. Espone di una sola proprietà **length** che restituisce il numero di pagine nella cronologia. Espone poi i metodi:

- **go()**: carica la pagina nella posizione indicata rispetto alla pagina corrente; ad esempio:

```
history.go(-2); //carica la seconda pagina caricata prima di quella corrente
history.go(3); //come premere 3 volte il pulsante "avanti" del browser
```

- **back()** e **forward()**: caricano la pagina precedente e successiva (corrispondono a `history.go(-1)` e `history.go(1)`)

Spesso l'oggetto history è usato in associazione agli eventi della pagina per creare bottoni "indietro":

```
<a href="#" onclick="history.back(); return false;">&lt;&lt; Indietro</a>
```

navigator

Questo oggetto consente di ottenere informazioni riguardo al browser in uso dall'utente e al suo sistema operativo. Espone interessanti proprietà:

- **appName** è una stringa che restituisce il nome del browser (Opera, Netscape - valore restituito anche da Firefox -, Microsoft Internet Explorer...).
- **appVersion** contiene dati sul browser, quali la versione della *release* e il sistema operativo sul quale sta girando.
- **plugins** restituisce un array del *plugin* installati.

Tramite la prima proprietà, possiamo verificare il browser in uso per agire in modo diverso ed eliminare così alcuni problemi di compatibilità:

```
var ns = ( navigator.appName.indexOf("Netscape") > -1 ) //se l'utente naviga con
Firefox/Netscape, contiene true
var mie = ( navigator.appName.indexOf("Microsoft") > -1 ) //microsoft internet
explorer
var wie = ( navigator.appName.indexOf("Windows") > -1 ) //windows internet
explorer
var opera = ( navigator.appName.indexOf("Opera") > -1 )
//etc...

if (ns) {
 //codice per netscape/firefox
}
else if (mie || wie) {
 //codice per ie
} else if (opera) {
 //codice per opera
} else {
 //codice per altri browser
}
```

Normalmente è possibile omettere i controlli per browser quali Opera, che normalmente mantengono una grande compatibilità con Firefox, in quanto entrambi seguono le stesse convenzioni del W3C; spesso il problema è trovare una soluzione compatibile anche con IE.

screen

Questo oggetto permette di ottenere informazioni sullo schermo dell'utente e sulla sua configurazione. Tra le sue proprietà ricordiamo:

- **height** e **width** che indicano la risoluzione verticale e orizzontale dello schermo in pixel
- **colorDepth** (profondità del colore) indica il numero di bit utilizzati per i colori (ad esempio, 2 indica che l'utente sta navigando con uno schermo in bianco e nero - fatto altamente improbabile, perché ormai la maggior parte dei monitor supportano la configurazione 16 o 32 bit).

DOM

Il **DOM** è una tecnonologia standard del W3C per muoversi nella struttura di documenti XML.
In relazione a JavaScript questo è molto interessante, in quanto un documento HMTL ben formato è in realtà un documento XML, e tramite il DOM di JavaScript è possibile accedere a *qualsiasi* elemento nella pagina per modificarne le proprietà o per attivarne i metodi. Il DOM definisce anche un modello oggetto più evoluto che il BOM che tratteremo quindi in modo più approfondito.


*Per approfondire vai su Wikipedia, vedi la voce **Document Object Model**.*

BOM vs DOM

Ci si potrebbe a questo punto chiedere dove si sovrappongano il DOM che andremo ad analizzare e BOM che abbiamo appena visto e, soprattutto, perché abbiamo esaminato anche il BOM, nonostante sia meno evoluto.

Il problema sorge in realtà dall'evoluzione dei browser, che hanno creato dei propri standard prima della definizione del W3C. In realtà il DOM non è altro che un'evoluzione del BOM sviluppato inizialmente da browser come Netscape o Internet Explorer con l'avvento di JavaScript.

Solo successivamente il DOM è stato standardizzato, ma si continua ancora oggi ad usare il BOM, per problemi di compatibilità (anche se ormai questi sono praticamente minimi) o anche perché quest'ultimo è più "intuitivo" e non richiede di capire la gerarchia XML della pagina.

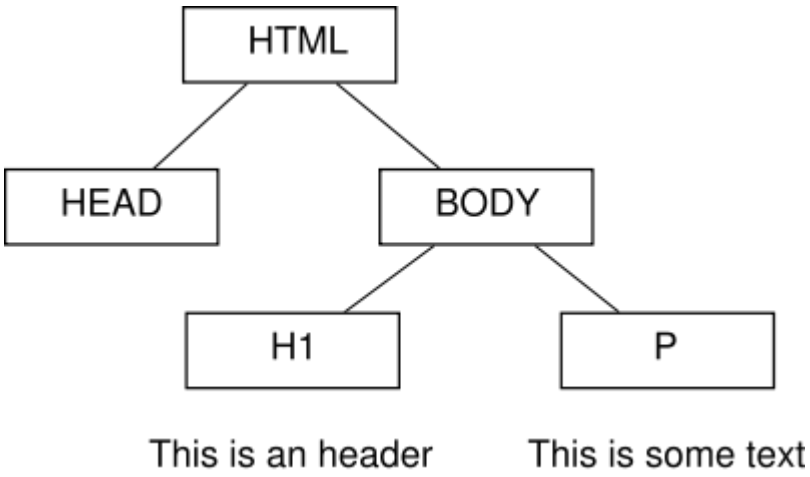
La struttura gerarchica di un documento

Per poter lavorare con il DOM è però necessario capire la struttura ad albero di un documento, perché è su questa che si muove. Questa struttura è astratta e parte dalla gerarchia degli elementi in una pagina.

Prendiamo ad esempio questa pagina HTML:

```
<html>
<head></head>
<body>
<h1>This is an header</h1>
<p>This is some text</p>
</body>>
</html>
```

È un semplice documento HTML contenente un'intestazione e un paragrafo. La sua struttura ad albero sarà la seguente:



Ogni elemento, di qualsiasi tipo esso sia, occupa una posizione ben definita. Ogni elemento nella struttura ad albero viene chiamato "*nodo*", e rappresenta nel modo più astratto ciascun elemento della pagina.
Ad ogni nodo si può associare un nodo *genitore*, tranne che per il nodo radice del documento, nel nostro caso `<html>`. Ogni nodo può avere dei nodi *figli* (ad esempio l'intestazione e il paragrafo sono figli dell'elemento `<body>` e dei nodi fratelli (ad esempio `<p>` è fratello di `<h1>`).

Questa è un'altra differenza dal BOM: non esiste più un oggetto per ogni elemento nella pagina, ma esiste un oggetto astratto che li rappresenta tutti. Non sarà più quindi necessario studiare *singolarmente* ogni singolo elemento della pagina, ma solo i tipi di nodi più importanti.

Vari tipi di nodo

L'oggetto *Node* è quindi, come abbiamo detto, il modo più astratto per indicare un elemento della pagina; per i diversi tipi di elemento esistono poi oggetti più specifici, che sono delle *estensioni* dell'oggetto *Node*; tra i più importanti troviamo:

- *Document* è un oggetto radice del documento. L'oggetto *window.document* è un'istanza di questo oggetto.
- *Element* è un nodo che rappresenta un elemento HTML (come `<p>`)
- *Text* è un nodo di testo, che costituisce l'ultimo nodo figlio di un elemento (ad esempio *This is some text* è un nodo di testo)

Proprietà e metodi del DOM

Dopo l'introduzione generale al DOM e alla sua struttura, passiamo ora ad analizzare i suoi oggetti per applicare quanto visto.

Ottenere un elemento

Vediamo ora il primo passo da compiere quando si lavora con il DOM: ottenere il riferimento ad un elemento HTML e memorizzarlo in una variabile.

Per fare questo, nella maniera più semplice, il DOM utilizza l'attributo `id` che è possibile assegnare ad ogni elemento:

```
var par = document.getElementById("p1"); //ottiene il paragrafo con id="p1"
```

Per ottenere invece l'elemento root del file HTML, si usa la proprietà `document.documentElement`.

Lavorare su un elemento

Una volta ottenuto l'elemento (questo è il modo più semplice, ma ne esistono anche altri), è possibile modificarne gli attributi o modificare gli stili:

- la proprietà **tagName** restituisce il nome del tag (per esempio *a* o *img*)
- la proprietà **style** permette di impostarne lo stile in modo veloce (questa caratteristica non è in realtà uno standard DOM, ma funziona bene su tutti i browser) tramite il nome della proprietà CSS, ad esempio `p1.style.color = "#f00";` imposta a rosso il colore del paragrafo ottenuto prima
- esistono poi tre metodi per impostare o eliminare gli attributi:
    - **setAttribute** imposta un attributo tramite il nome e il valore (`p1.setAttribute("align", "center");`);
    - **getAttribute** restituisce il valore un attributo tramite il suo nome (`p1.getAttribute("align");`);
    - **removeAttribute** rimuove un attributo (tornando così al valore di default) tramite il suo nome (`p1.removeAttribute("align");`).

Spostarsi nella struttura ad albero

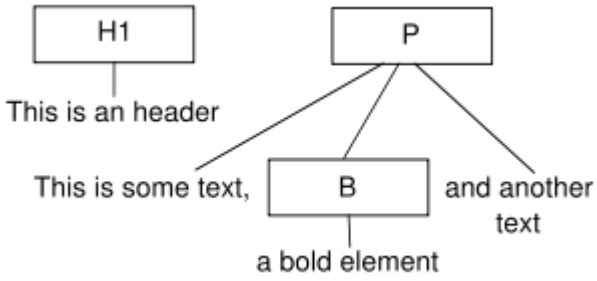
Arriva ora il momento di sfruttare la struttura ad albero che abbiamo visto ora. L'*node* espone infatti delle proprietà e dei metodi che permettono di spostarsi nella struttura ad albero, per ottenere nodi fratelli, genitori o figli:

- **firstChild** e **lastChild** restituiscono il primo e l'ultimo nodo figlio di un elemento (se è uno solo, restituiscono lo stesso elemento). Se per esempio abbiamo un paragrafo che contiene
- **previousSibling** e **nextSibling** restituiscono il precedente e il successivo nodo "fratello" del nodo
- **parentNode** restituisce il nodo genitore
- **ownerDocument** restituisce l'elemento del documento che contiene il nodo (ad esempio `<body>`
- **getElementsByName()** è un metodo che restituisce un elenco di tutti gli elementi HTML con uno stesso nome (per esempio "*img*"); per accedere ad un elemento si usa il metodo `item()` oppure tramite la notazione degli array.

Una volta ottenuto un nodo, è possibile usare la proprietà **nodeValue** che ne restituisce il valore e la proprietà **nodeType** che ne restituisce il tipo come numero.
Analizziamo ad esempio questo spezzone di HTML:

```
<h1>This is an header</h1>
<p id="p1">This is some text, <b>a bold element</b> and another text</p>
```

Questa è la sua rappresentazione del DOM:



Vediamo ora come agire. La cosa più semplice è partire dall'elemento con un ID univoco:

```
var p1 = document.getElementById("p");
var h1 = document.previousSibling; //ottiene l'elemento H1

alert(p1.nodeValue); //restituisce "null" (un elemento HTML non ha un valore)

var testo1 = p1.firstChild;
alert(testo1.nodeValue); //restituisce il nome dell'oggetto, ovvero "This is
some text"
alert(testo1.nextSibling); //restituisce "Object BElement"
// in quanto il fratello del primo nodo di testo è l'elemento <b>
alert(testo1.nextSibling.firstChild); //restituisce il testo
```

Creare nuovi elementi

La possibilità di gestire qualsiasi elemento della pagina come oggetto da anche la possibilità di creare nuovi elementi, semplicemente creando nuovi oggetti di nodo e poi inserendoli nella struttura ad albero. Per fare ciò esistono alcuni particolari metodi, prima di tutto dell'oggetto `document`:

- **createElement**: crea un nodo di elemento con il nome specificato come parametro
- **createTextNode**: crea un nodo di testo con il testo specificato

Una volta creato un nodo, sia esso di elemento o di testo, va aggiunto ai nodi già esistenti:

- **appendChild** aggiunge al nodo il nodo figlio passato come parametro, in ultima posizione
- **insertNode** inserire il nodo specificato prima del nodo specificato come secondo argomento
- **removeChild** elimina il nodo figlio indicato come parametro
- **replaceChild** sostituisce il nodo figli indicato nel primo parametro con il secondo
- **hasChildNodes** restituisce *true* se il nodo ha nodi figli.

Vediamo un esempio:

```
var body = document;
var link = document.createElement("a");
var testo = document.createTextNode("Segui questo link");

link.setAttribute("href", "pagina.html");

link.appendChild(testo);
body.appendChild(link);
```

Gli eventi nel DOM

Il **modello evento del DOM** in realtà non offre grandi funzionalità in più rispetto a quello del BOM

di cui abbiamo visto una parte in precedenza; tuttavia il modello evento del DOM ha un migliori supporto e compatibilità tra i browser e, essendo il DOM l'ultima tecnologia sviluppata in questo campo, l'evoluzione del JavaScript non si baserà sul BOM bensì sul DOM.

Analogie con quanto già visto


*Per approfondire, vedi la pagina **JavaScript/Gli eventi nel BOM**.*

In realtà molte di ciò che è stato visto per il BOM è riutilizzabile anche con il DOM: sono uguali i nomi degli eventi, il funzionamento del collegamento tra oggetti ed eventi, l'uso dell'oggetto *this*

L'oggetto event

La funzionalità interessante che tratteremo ora è l'**oggetto *event***, che permette di ottenere informazioni sull'evento appena scaturito, come l'elemento che lo ha generato, o la posizione del mouse.

Possiamo utilizzarlo dentro alla dichiarazione dell'evento, ad esempio:

```
<a
href="pagina.html"
onmouseover="alert('Il mouse è alla posizione ' + event.screenX + ', ' +
event.screenY +  'dello schermo');"
>
link
</a>
```

In questo esempio utilizziamo le due proprietà *screenX* e *screenY* dell'oggetto event, che restituiscono la posizione del cursore del mouse rispetto allo schermo.

Rispetto all'utilizzo di questo oggetto, bisogna fare attenzione quando si richiama da delle funzioni *handler*: l'oggetto event infatti ha una visibilità privata, non può quindi essere richiamato esternamente alla dichiarazione dell'evento (nell'attributo "onclick" per esempio). Per ovviare a questo problema è sufficiente passare l'oggetto event come argomento. Ad esempio, si crea la funzione:

```
function posizione (e) {
alert('Il mouse è alla posizione ' + e.screenX + ', ' + e.screenY +  'dello
schermo');
}
```

Nell'HTML si inserirà: link </source>

Proprietà

- **timestamp** (funziona solo su FF) restituisce la data e l'ora in cui si è verificato l'evento;
- **target** e **relatedTarget** (**srcElement** e **toElement** su Internet Explorer) restituiscono rispettivamente il nodo che ha generato l'evento e il nodo su cui probabilmente sarà il mouse dopo che l'evento si sarà concluso (ad esempio è utile per gli eventi *mouseOut*). Quando si usano queste proprietà bisogna prima controllare il browser in uso dall'utente (vedi questa pagina);
- **altKey**, **ctrlKey** e **shiftKey** indicano rispettivamente se è premuto il tasto *alt*, *ctrl* o *shift* mentre avviene evento;

- **button** indica quale pulsante del mouse è stato premuto (0 il tasto sinistro, 1 quello destro);
- **clientX** e **clientY** indicano le posizioni del cursore rispetto alla finestra del browser (a partire dall'angolo in altro a destra);
- **screenX** e **screenY** indicano le posizioni del cursore rispetto allo schermo dell'utente (a partire dall'angolo in altro a destra).

Appendice

Parole riservate

```
break case catch continue default delete
else false finally for function if in
instanceof new null return switch this
throw true try typeof var void while with
```

Debugging

Il termine inglese *debugging* significa "togliere gli scarafaggi", dove gli scarafaggi sono i *bug*, gli errori nel codice.
I tipi di errore possono essere diversi: dai semplici **errori di sintassi** (possono essere paragonati ad una frase grammaticalmente scorretta in italiano) come la mancata chiusura di una parentesi, fino agli **errori logici** (come una frase che non ha alcun senso, nonostante sia grammaticalmente corretta); ad esempio, può essere un errore logico dimenticarsi di incrementare una variabile quando serve, oppure utilizzare un algoritmo sbagliato.

Come evitare alcuni errori comuni

- Leggi attentamente il tuo codice per eventuali errori di battitura
- Assicurati che tutte le parentesi siano chiuse (può aiutare in questo un buon editor)
- Ricorda che JavaScript è *case-sensitive*: `Name` e `name` sono due variabili differenti
- Non usare parole riservate come nomi di variabili o di funzioni
- Ricordati di effettuare l'*escape* degli apici singoli/doppi:
- Quando converti i numeri usando la funzione `parseInt` ricorda che "08" o "09" sono valutati come numeri in base otto, a causa dello zero.
- Ricorda che JavaScript è indipendente dalla piattaforma, ma lo è dal browser!

Semplice debugging

Oltre a verificare la lista degli errori comuni, uno dei modi più semplici di effettuare il debugging del proprio codice consiste nell'utilizzare le istruzioni `alert` mostrando il contenuto di una variabile, oppure semplicemente come "segnaposto" per sapere a che punto si è del codice.

Supponiamo ad esempio che lo script non faccia quanto desiderato, magari fermandosi a metà: come si può individuare il punto esatto in cui lo script si ferma? Il modo più semplice è inserire prima e/o dopo blocchi di istruzioni che consideriamo "sensibili" o di cui non siamo certi della correttezza con istruzioni del tipo:

```
//...
alert ("inizio blocco");
//...
```

```
alert ("fine blocco");
//...
```

Per esempio, se verrà mostrato il primo messaggio, significa che fino a quel punto è tutto corretto ma se, ad esempio, nono viene mostrato il secondo, significa che in mezzo ai due *alert* c'è l'istruzione "buggata".

Questo sistema è utile ad esempio nei cicli: per controllare ad ogni passaggio l'andamento del programma può essere comodo inserire nel ciclo un *alert* che mostri il valore del contatore e il valore delle altre variabili che intervengono nel programma, così da capire passo per passo cosa avviene effettivamente.

Licenza

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of

mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the

back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- **H.** Include an unaltered copy of this License.

- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document

under the terms of the GNU Free Documentation License, Version 1.2

or any later version published by the Free Software Foundation;

with no Invariant Sections, no Front-Cover Texts, and no Back-Cover

Texts. A copy of the license is included in the section entitled "GNU

Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the

Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.