Università di Pisa
Dipartimento di Informatica

Programmazione in JavaScript

Vincenzo Ambriola

Versione 5.1 del 19 maggio 2015
Anno Accademico 2014/15

Prefazione

Per chi si avvicina alla programmazione gli ostacoli da superare sono tanti: un nuovo linguaggio (artificiale) da imparare, strumenti di sviluppo da provare per capirne la logica di funzionamento, esercizi da risolvere per apprendere i concetti di base e, successivamente, quelli più avanzati. Ci sono molti modi per insegnare a programmare e ogni docente, nel tempo, ha trovato il suo.

Questo libro è rivolto agli studenti del primo anno del Corso di laurea in *Informatica umanistica* che seguono *Elementi di programmazione* (uno dei due moduli dell'insegnamento di *Fondamenti teorici e programmazione*) e *Programmazione* (uno dei due moduli di *Progettazione e programmazione web*). L'approccio adottato si basa sull'introduzione graduale dei concetti di *JavaScript*, un linguaggio di programmazione ampiamente usato per la programmazione web.

Questo libro non è un manuale di JavaScript. Il linguaggio è troppo complesso per insegnarlo a chi non ha mai programmato e, soprattutto, di manuali di questo tipo ce ne sono tanti in libreria. Il libro presenta gli aspetti più importanti di JavaScript, necessari ma, soprattutto, sufficienti per la soluzione dei problemi proposti. Il lettore interessato agli aspetti più avanzati di JavaScript e al suo uso professionale è invitato a continuare lo studio e la pratica di questo linguaggio. I risultati supereranno di gran lunga le aspettative.

1.1 Struttura del libro

Il libro è strutturato in due parti: la prima presenta gli elementi di programmazione necessari per risolvere semplici problemi su numeri e testi; la seconda affronta il tema della programmazione web. Ogni parte è strutturata in capitoli, dedicati a un aspetto della programmazione. Alcuni capitoli si concludono con esercizi che il lettore è invitato a risolvere non solo con carta e matita ma mediante un calcolatore. Gli strumenti necessari sono alla portata di tutti: un browser di nuova generazione è più che sufficiente. La soluzione degli esercizi proposti è riportata al termine delle due parti.

Nel libro sono state usate le seguenti *convenzioni tipografiche*, per facilitare la lettura dei programmi presentati:

- il *corsivo* è usato per indicare la prima volta che un termine rilevante compare nel libro; l'indice analitico contiene l'elenco di questi termini, con l'indicazione della pagina in cui sono introdotti;
- la sintassi di JavaScript e gli esempi sono riportati all'interno di un riquadro colorato.

1.2 Ringraziamenti

La prima parte di questo libro nasce da una lunga collaborazione con *Giuseppe Costa*, coautore di *4 passi in JavaScript*. Senza le sue preziose indicazioni sarebbe stato praticamente impossibile capire le insidie e la bellezza di un linguaggio di programmazione complesso come JavaScript.

La seconda parte è stata scritta seguendo i consigli e i suggerimenti di *Maria Simi*, profonda conoscitrice del web, della sua storia e delle tante tecnologie ad esso collegate.

Indice

Parte prima

Elementi di programmazione

2 Linguaggi e grammatiche

Quando si parla di *linguaggio* viene subito in mente il linguaggio parlato che usiamo tutti i giorni per comunicare con chi ci circonda. In realtà, il concetto di linguaggio è molto più generale.

Possiamo individuare due categorie di linguaggi: *naturali* e *artificiali*. I primi sono linguaggi ambigui, perché il significato delle *parole* dipende dal contesto in cui sono inserite. I linguaggi naturali sono inoltre caratterizzati dal fatto di mutare con l'uso, per l'introduzione di neologismi e di parole provenienti da altri linguaggi (per l'italiano è il caso dei termini stranieri o delle espressioni dialettali).

Un esempio di frase ambigua è: *Ho analizzato la partita di calcio*. La parola "calcio" può significare "lo sport del calcio" nella frase *Ho analizzato la partita di calcio dell'Italia* o "il minerale calcio" nella frase *Ho analizzato la partita di calcio proveniente dall'Argentina*.

A differenza dei linguaggi naturali, i linguaggi artificiali hanno regole e parole che non cambiano con l'uso e il cui significato non dipende dal contesto in cui sono inserite. A questa famiglia appartengono i linguaggi utilizzati per descrivere le operazioni da far compiere a macchine o apparecchiature. Tali descrizioni non possono essere ambigue perché una macchina non può decidere autonomamente tra più possibilità di interpretazione.

Un esempio di linguaggio artificiale è quello usato per comandare un videoregistratore: una frase del linguaggio è una qualunque sequenza di *registra*, *riproduci*, *avanti*, *indietro*, *stop*. Le parole hanno un significato preciso e indipendente dal contesto in cui si trovano.

L'informatica ha contribuito notevolmente alla nascita di numerosi linguaggi artificiali, i cosiddetti *linguaggi di programmazione*, usati per la scrittura di *programmi* eseguibili da *calcolatori elettronici*. Questo libro è dedicato allo studio di uno di essi, il linguaggio JavaScript.

2.1 Alfabeto, linguaggio

Un *alfabeto* è formato da un insieme finito di *simboli*. Ad esempio, l'alfabeto $A = \{a, b, c\}$ è costituito da tre simboli.

Una *frase* su un alfabeto è una sequenza di lunghezza finita formata dai simboli dell'alfabeto. Ad esempio, con l'alfabeto A definito in precedenza si possono formare le frasi *aa*, *abba*, *caab*.

Dato un alfabeto A, l'insieme di tutte le frasi che si possono formare usando i suoi simboli è infinito, anche se ogni frase è di lunghezza finita. Per semplicità questo insieme è chiamato *insieme delle frasi di* A. Sempre usando l'esempio precedente, è possibile formare l'insieme delle frasi di A ma non è possibile riportarlo in questo libro perché, come già detto, la sua lunghezza è infinita. Ciononostante, è possibile mostrarne una parte finita, usando come artificio i puntini di sospensione per indicare la parte infinita: {*a*, *b*, *c*, *aa*, *ab*, *ac*, *ba*, *bb*, *bc*, *ca*, *cb*, *cc*, ...}.

Dato un alfabeto A, un *linguaggio L su* A è un sottoinsieme delle frasi di A. Questa definizione è diversa da quella data in precedenza: non tutte le frasi di A sono, infatti, anche frasi di L.

Abbiamo fatto riferimento a "un" linguaggio e non "del" linguaggio su un alfabeto perché può esistere più di un linguaggio su un dato alfabeto. Prendiamo come esempio il linguaggio italiano e quello inglese: entrambi si basano sull'alfabeto latino (detto anche alfabeto romano), ma sono costituiti da insiemi diversi di frasi.

I like walking on the grass è una frase del linguaggio inglese, ma non lo è di quello italiano. *Mi piace camminare sull'erba* è una frase del linguaggio italiano, ma non lo è di quello inglese, anche se il significato è lo stesso.

Sdksfdk skjfsfkj sdkfsakjfd, invece, non è una frase né del linguaggio italiano né di quello inglese, nonostante appartenga all'insieme delle frasi sull'alfabeto latino.

2.2 Grammatiche

Un linguaggio ha due aspetti importanti:

- *sintassi*: le regole per la formazione delle frasi del linguaggio (ad esempio, la grammatica italiana);
- *semantica*: il significato da attribuire ad ogni frase del linguaggio (ad esempio, tramite l'analisi logica delle frasi, che permette di individuare il soggetto di una data azione, l'oggetto e così via).

Esistono *formalismi* che definiscono i due aspetti di un linguaggio. In questo libro presentiamo un formalismo per la definizione della sintassi. Non trattiamo, invece, gli aspetti relativi alla definizione formale della semantica.

Una *grammatica* è un formalismo che permette di definire le regole per la formazione delle frasi di un linguaggio. Una grammatica è quindi un *metalinguaggio*, ovvero un linguaggio che definisce un altro linguaggio.

Una grammatica è formata da:

- *simboli terminali*: rappresentano gli *elementi sintattici* del linguaggio che, di norma, coincidono con l'alfabeto del linguaggio;
- *simboli non-terminali* o *metasimboli*: rappresentano le *categorie sintattiche* del linguaggio;
- *regole*: definiscono le relazioni tra i simboli terminali e i non-terminali; mediante la loro applicazione si ottengono le frasi del linguaggio.

Un linguaggio L le cui frasi rispettano le regole della grammatica G si dice *generato da* G.

2.3 Backus-Naur Form

La *Backus-Naur Form* (*BNF*) è uno dei formalismi più usati per definire le grammatiche. In accordo con la definizione di grammatica, la BNF prevede la definizione di un insieme di simboli non-terminali (di solito racchiusi tra parentesi angolari) e di un insieme di simboli terminali (di solito indicati in corsivo o in grassetto per distinguerli dai simboli non-terminali). Inoltre, deve essere indicato un *simbolo iniziale* che appartiene all'insieme dei simboli non-terminali. La funzione di questo simbolo sarà chiarita nel seguito.

Le regole per la formazione delle frasi sono di due tipi:

- $X ::= S$

  si legge "*X* può essere sostituito con *S*" o "*X* produce *S*" (da cui il nome di *produzioni* dato alle regole). *X* è un simbolo non-terminale, *S* è una sequenza di simboli terminali e non-terminali.

- $X ::= S_1 \mid S_2 \mid \ldots \mid S_n$

  si legge "*X* può essere sostituito con S_1, S_2, ..., o S_n". Anche in questo caso *X* è un simbolo non-terminale, mentre S_1, S_2, ..., S_n sono sequenze di simboli terminali e non-terminali.

Il linguaggio L generato da una grammatica G definita in BNF è l'insieme di tutte le frasi formate da soli simboli terminali, ottenibili partendo dal simbolo iniziale e applicando in successione le regole di G.

Applicare una regola *R* a una sequenza *s* di simboli significa sostituire in *s* un'occorrenza del simbolo non-terminale definito da *R* con una sequenza della parte sinistra di *R* .

Per rendere concreti i concetti presentati finora, mostriamo la grammatica G_1, definita in BNF:

- A = {*a*, *b*} alfabeto
- N = {<*S*>, <*A*>, <*B*>} simboli non-terminali
- I = <*S*> simbolo iniziale
- <*S*> ::= <*A*><*B*> | <*B*><*A*> regola di <*S*>
- <*A*> ::= *a* | *aa* regola di <*A*>
- <*B*> ::= *b* | *bb* regola di <*B*>

Il linguaggio generato da G_1 è {*ab*, *aab*, *abb*, *aabb*, *ba*, *bba*, *baa*, *bbaa*}. Si noti che, in questo esempio, il linguaggio è finito perché la grammatica genera esattamente otto frasi.

Una grammatica leggermente più complicata è G_2:

- A = {*a*, *b*}, alfabeto
- N = {<*S*>, <*A*>, <*B*>} simboli non-terminali
- I = <*S*> simbolo iniziale
- <*S*> ::= <*A*><*B*> regola di <*S*>
- <*A*> ::= *a* | *a* <*A*> regola di <*A*>
- <*B*> ::= *b* | *b* <*B*> regola di <*B*>

Il linguaggio infinito generato da G_2 è {*ab*, *aab*, *abb*, *aabb*, *aaab*, *aaabb*, *aaabbb*, *abbb*, *aabbb*, *aaabbb*, ...}.

2.4 Sequenze di derivazione

Il *processo di derivazione* di una frase si può rappresentare tramite una *sequenza di derivazione,* formata da una successione di frasi intermedie, costruite a partire dal simbolo iniziale, con l'indicazione della regola usata per passare da una frase alla successiva. Ogni frase, tranne l'ultima, è formata da simboli terminali e non-terminali. L'ultima frase è formata esclusivamente da simboli terminali e, pertanto, appartiene al linguaggio generato dalla grammatica a cui appartengono le regole di derivazione.

A titolo di esempio, consideriamo la sequenza di derivazione della frase *aab* appartenente al linguaggio generato da G_2:

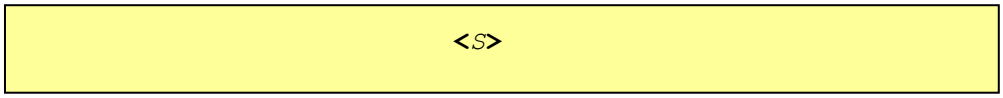
```
<S>
=> Regola <S> ::= <A> <B>
<A> <B>
=> Regola <A> ::= a | a <A> seconda opzione
a <A> <B>
=> Regola <A> ::= a | a <A> prima opzione
a a <B>
=> Regola <B> ::= b | b <B> prima opzione
a a b
```

2.5 Alberi di derivazione

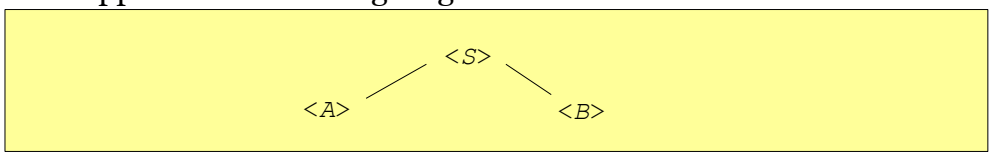
Un metodo alternativo per rappresentare il processo di derivazione si basa sugli *alberi di derivazione*. Per costruire l'*albero sintattico* di una frase si parte dal simbolo iniziale della grammatica, che ne costituisce la *radice*. Successivamente, per ogni *foglia* che è un simbolo non-terminale, si sceglie una regola da applicare sostituendo la foglia con un *nodo* e generando tanti figli quanti sono i simboli terminali e non-terminali dell'opzione scelta. Le foglie possono essere simboli non-terminali solo nelle fasi intermedie di costruzione. Il procedimento termina quando tutte le foglie sono simboli terminali.

A differenza del precedente, questo metodo fornisce una visione d'insieme del processo di derivazione ed evidenzia il simbolo non-terminale sostituito ad ogni passo. Anche l'albero sintattico per una data frase viene costruito applicando in sequenza le regole di derivazione a partire dal simbolo iniziale, ma la sua struttura finale non dipende dall'ordine di applicazione delle regole. Infine, un albero sintattico è caratterizzato dal fatto che la radice e tutti i nodi sono simboli non-terminali e le foglie sono simboli terminali.

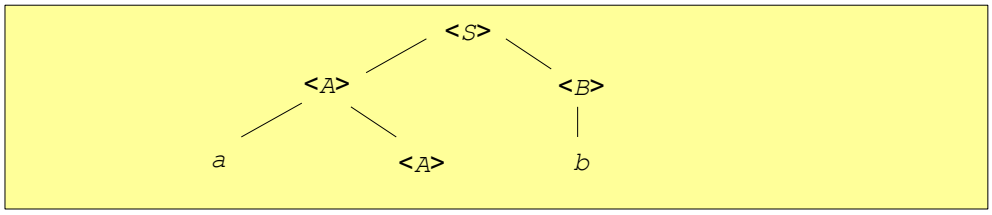
Ecco un esempio di costruzione dell'albero sintattico per la frase *aab* appartenente al linguaggio generato da G_2:



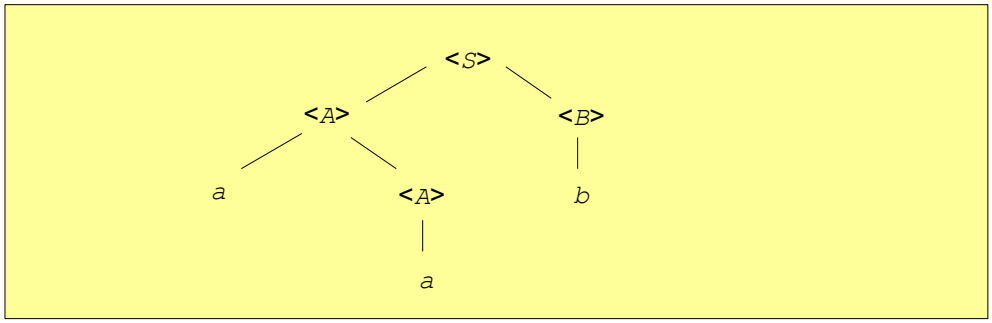
Applichiamo la regola <*S*> ::= <*A*> <*B*> al nodo radice dell'albero di derivazione. L'applicazione della regola genera due nuovi nodi.



Applichiamo la regola <*A*> ::= *a* | *a* <*A*> (seconda opzione) al nodo <*A*> e la regola <*B*> ::= *b* | *b* <*B*> (prima opzione) al nodo <*B*>.



Infine applichiamo la regola <*A*> ::= *a* | *a* <*A*> (prima opzione) al nodo <*A*>. La frase *aab* è formata dai simboli terminali presenti sulle foglie dell'albero di derivazione, letti da sinistra a destra.



3 Programmi, comandi e costanti

JavaScript è un linguaggio di programmazione, in particolare è un linguaggio di *script*, cioè un linguaggio per definire programmi eseguibili all'interno di altri programmi (chiamati *applicazioni*).

JavaScript è usato principalmente per la definizione di programmi eseguibili nei *browser*, applicazioni per la visualizzazione di *pagine web*. In questo ambito, JavaScript permette di rendere dinamici e interattivi questi documenti affinché mostrino contenuti diversi in base alle azioni dell'utente o a situazioni contingenti, come l'ora corrente o il tipo di browser utilizzato. Mediante JavaScript è anche possibile aggiungere effetti grafici o accedere alle funzionalità del browser in cui i documenti sono visualizzati (lanciare una stampa, aprire una nuova finestra, ridimensionare o spostare sullo schermo una finestra di visualizzazione).

Con l'evoluzione dei browser, JavaScript ha subito numerose modifiche, acquisendo progressivamente nuove funzionalità. Per evitare possibili confusioni, il libro fa riferimento a JavaScript 1.8.2, la versione del linguaggio accettata dal browser *Firefox* (versione 26.0 e successive).

Nella prima parte del libro JavaScript è presentato esclusivamente come linguaggio di programmazione, evitando di descriverne il suo uso come linguaggio di script. Questa scelta è motivata dalla necessità di introdurre gli aspetti di base del linguaggio prima di quelli più complessi, necessari per affrontare gli argomenti trattati nella seconda parte. Per rendere concreta la presentazione del linguaggio, il libro fa riferimento a un *ambiente di programmazione* chiamato *EasyJS*[1]. Questo ambiente permette di definire un programma JavaScript, eseguirlo e visualizzare il risultato dell'esecuzione.

3.1 Programma

In JavaScript un programma è una *sequenza* di *comandi*. Un comando può essere una *dichiarazione*, un *comando semplice* o un *comando composto*.

```
<Programma> ::= <Comandi>

<Comandi>   ::= <Comando>
              |  <Comando> <Comandi>
```

1 http://www.di.unipi.it/~ambriola/pw/radice.htm

```
<Comando>   ::= <Dichiarazione>
             |  <ComandoSemplice>
             |  <ComandoComposto>
```

L'esecuzione di un programma consiste nell'esecuzione della sua sequenza di comandi. I comandi sono eseguiti uno dopo l'altro, nell'ordine con cui compaiono nella sequenza. Questo comportamento è tipico dei *linguaggi di programmazione imperativi*, classe alla quale appartiene JavaScript.

Il *punto e virgola* indica la fine di una dichiarazione o di un comando semplice. Anche se è buona norma terminare, quando previsto, un comando con un punto e virgola, in JavaScript è possibile ometterlo se il comando è interamente scritto su una riga e sulla stessa riga non ci sono altri comandi. Il *ritorno a capo*, normalmente ignorato, in questo caso funge da *terminatore di comando*.

JavaScript è un linguaggio *case sensitive* perché fa distinzione tra lettere maiuscole e minuscole. Gli *spazi bianchi* e le *interruzioni di riga* servono per migliorare la leggibilità dei programmi. Si possono utilizzare per separare gli elementi del linguaggio, ma non possono essere inseriti al loro interno.

Un *commento* è formato da una o più righe di testo. In JavaScript ci sono due tipi di commenti:

- su una riga sola, introdotti da "//"
- su più righe, introdotti da "/*" e chiusi da "*/".

```
// questo è un commento su una riga
/* questo è un commento
   su due righe */
```

3.2 Costanti numeriche e logiche

In JavaScript una costante può essere un *valore numerico* o un *valore logico* (o *booleano*). I valori numerici appartengono al *tipo primitivo* dei numeri, quelli logici al tipo primitivo dei booleani.

```
<Costante>  ::= <Numero>
             |  <Booleano>

<Numero>    ::= <Intero>
             |  <Intero>.<Cifre>
             |  <Intero>E<Esponente>
             |  <Intero>.<Cifre>E<Esponente>

<Intero>    ::= <Cifra>
             |  <CifraNZ> <Cifre>
```

```
<Cifra>     ::= 0 | <CifraNZ>

<CifraNZ>   ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Cifre>     ::= <Cifra>
              |  <Cifra> <Cifre>

<Esponente> ::= <Intero>
              |  + <Intero>
              |  - <Intero>

<Booleano>  ::= true | false
```

Alcuni esempi di *costanti numeriche* sono i seguenti:

- *12*
- *159.757*
- *23E5*
- *51E+2*
- *630E–6*
- *2.321E4*
- *3.897878E+7*
- *9.7777E–1*

La costante *2* denota il valore *due*, la costante *159.757* denota il valore *centocinquantanove virgola settecentocinquantasette*. Le altre costanti, in *rappresentazione esponenziale* possono essere trasformate in *rappresentazione decimale*, tenendo presente che l'*esponente* positivo sposta la virgola verso destra e quello negativo verso sinistra:

- *23E5* equivale a *2 300 000*
- *51E+2* equivale a *5 100*
- *630E–6* equivale a *0.000630*
- *2.321E4* equivale a *23 210*
- *3.897878E+7* equivale a *38 978 780*
- *9.7777E–1* equivale a *0.97777*

La costante *23E5* denota il valore *duemilionitrecentomila*, la costante *51E+2* denota il valore *cinquemilacento*, mentre la costante *630E–6* denota il valore *zero virgola zerozerozeroseicentotrenta*. Il valore denotato dalle altre costanti è facilmente desumibile.

Le *costanti booleane*, dette anche *costanti logiche*, sono due. La costante *true* denota il valore di verità *vero*, la costante *false* denota il valore di verità *falso*.

3.3 Costanti stringa

Una *stringa* è una sequenza di *caratteri* ed è il tipo di dato usato in JavaScript per rappresentare testi. Una stringa che appare esplicitamente in un programma è chiamata *costante* stringa ed è una sequenza (anche vuota) di caratteri racchiusi tra *apici singoli* o *apici doppi*. I caratteri sono tutti i *caratteri stampabili*: le lettere alfabetiche minuscole e maiuscole, le *cifre numeriche* (da non confondere con i numeri), i *segni di interpunzione* e gli altri simboli che si trovano sulla *tastiera* di un calcolatore (e qualcuno di più).

```
<Costante>       ::= <Stringa>

<Stringa>        ::= ""
                   |  "<CaratteriStr>"
                   |  ''
                   |  '<CaratteriStr>'

<CaratteriStr>   ::= <CarattereStr>
                   |  <CarattereStr><CaratteriStr>

<CarattereStr>   ::= <Lettera>
                   |  <Cifra>
                   |  <Speciale>

<Lettera>        ::= a | b | c | d | e | f | g | h | i
                   |  j | k | l | m | n | o | p | q | r
                   |  s | t | u | v | w | x | y | z
                   |  A | B | C | D | E | F | G | H | I
                   |  J | K | L | M | N | O | P | Q | R
                   |  S | T | U | V | W | X | Y | Z

<Speciale>       ::= Space[2]
                   |  ! | " | # | $ | % | & | ' | ( | )
                   |  * | + | , | - | . | / | : | ; | <
                   |  = | > | ? | @ | [ | \ | ] | ^ | _
                   |  ` | { | | | } | ~
```

In un programma, le costanti stringa devono essere scritte su una sola riga. Per inserire ritorni a capo, *tabulazioni*, particolari caratteri o informazioni di *formattazione* si utilizza la *barra diagonale decrescente*: '\"[3] chiamata anche *ca-*

2 Carattere di spaziatura.
3 In inglese questo carattere si chiama *backslash*.

rattere di quotatura. La coppia formata da *backslash* e da un altro carattere è chiamata *sequenza di escape*. Le principali sequenze di *escape* sono:

- \n: *nuova riga*;
- \r: ritorno a capo;
- \t: *tabulazione orizzontale*;
- \': apice singolo;
- \": apice doppio;
- \\: *backslash*.

3.4 Comando di stampa

Il primo comando semplice che prendiamo in considerazione è il *comando di stampa*[4]:

```
<ComandoSemplice> ::= print(<Costante>);
```

L'esecuzione di un comando di stampa ha l'effetto di scrivere sulla finestra inferiore di *EasyJS* la *costante* racchiusa tra *parentesi tonde*.

Ad esempio, il seguente programma stampa tre costanti.

```
print(12);
print(true);
print("alfa");
```

4 Come vedremo nel seguito, il comando di stampa è un'invocazione di funzione. Per semplicità, in questo capitolo è trattato come un comando semplice.

4 Espressioni

Un'*espressione* rappresenta un calcolo che restituisce un valore. In JavaScript, come in tutti i linguaggi di programmazione, le espressioni possono essere *semplici* o *composte*. Una costante è un'espressione semplice, il valore della costante coincide con il valore dell'espressione, il tipo dell'espressione coincide con il tipo del suo valore. Un'espressione composta si ottiene combinando una o più espressioni mediante un *operatore*, che rappresenta l'*operazione* da effettuare sulle espressioni, dette *operandi*.

```
<Espressione> ::= <Costante>
                | (<Espressione>)
                | <UnOp> <Espressione>
                | <Espressione> <BinOp> <Espressione>
<UnOp>        ::= -  | +  | !
<BinOP>       ::= -  | +  | *  | /  | %
                | && | || |
                | <  | <= | >  | >= | == | !=
```

L'introduzione delle espressioni richiede la modifica della sintassi del comando di stampa.

```
<ComandoSemplice> ::= print(<Espressione>);
```

4.1 Operatori

Gli operatori che hanno un unico operando sono detti *unari*. Gli operatori unari sono:

- – : *segno negativo*
- + : *segno positivo*
- ! : *negazione*

L'operatore di segno negativo ha un operando il cui valore è un numero. L'operatore cambia il segno del valore. Anche l'operatore di segno positivo ha un operando il cui valore è un numero. In questo caso, però, il segno del valore non è modificato. Questi due operatori sono detti *numerici*, perché il loro risultato è un numero.

L'operatore di negazione ha un operando il cui valore è un booleano. Se il valore dell'operando è *true*, il risultato è *false*, se il valore è *false* il risultato è *true*. Questo operatore è detto *booleano*, perché il suo risultato è un booleano.

Gli operatori che hanno due operandi sono detti *binari*. Anche gli operatori binari possono essere numerici o booleani. Quelli numerici sono:

- – : *sottrazione*
- + : *addizione*
- * : *moltiplicazione*
- / : *divisione*
- % : *modulo*

I primi quattro operatori (sottrazione, addizione, moltiplicazione, divisione) sono quelli usualmente conosciuti e utilizzati per effettuare i calcoli aritmetici. Il modulo è un'operazione su numeri interi, che restituisce il resto della divisione intera del primo operando per il secondo. Il valore dell'espressione *5%2* è *1*, cioè il resto della divisione intera di *5* per *2*.

Gli operatori binari booleani si dividono in due gruppi: quelli che hanno due operandi booleani, quelli che hanno due operandi dello stesso tipo. Gli operatori che appartengono al secondo gruppo sono anche detti *operatori di confronto*.

L'elenco dei primi operatori è il seguente:

- && : *congiunzione*
- || : *disgiunzione*

L'operatore di congiunzione restituisce *true* solo se i suoi operandi valgono *true*, *false* altrimenti. Se il primo operando vale *false* il secondo non è valutato.

L'operatore di disgiunzione restituisce *true* se almeno uno dei suoi operandi vale *true*, *false* altrimenti. Se il primo operando vale *true* il secondo non è valutato.

Gli operatori di confronto sono:

- == : *uguaglianz*a
- != : *disuguaglianza*
- > : *maggiore*
- >= : *maggiore o uguale*
- < : *minore*
- <= : *minore o uguale*

Come già detto, gli operandi degli operatori di confronto devono essere dello stesso tipo. Nel caso dell'uguaglianza, l'operatore restituisce *true* se i valori dei due operandi sono uguali, *false* altrimenti. L'operatore di disuguaglianza si

comporta in maniera simmetricamente opposta. L'operatore maggiore restituisce *true* se il valore del primo operando è maggiore di quello del secondo, *false* altrimenti. L'operatore di maggiore o uguale è leggermente diverso, in quanto restituisce *true* se il valore del primo operando è maggiore o uguale a quello del secondo, *false* altrimenti. Gli altri due operatori si comportano in maniera simmetricamente diversa.

La *relazione di ordinamento* per i numeri è quella usuale, mentre per i booleani si assume che *true* sia maggiore di *false*. Per le stringhe vale la *relazione di ordinamento lessicografico*, basata sulla *relazione di ordinamento alfanumerico*, in cui le lettere minuscole e quelle maiuscole sono ordinate secondo l'alfabeto inglese, tutte le maiuscole precedono le minuscole, le cifre numeriche sono ordinate secondo il loro valore, tutte le cifre numeriche precedono tutte le lettere. L'ordinamento lessicografico prevede che una stringa *a* è minore di una stringa *b* se il primo carattere di *a* è minore del primo carattere di *b*, secondo la relazione di ordinamento alfanumerico. Se i due caratteri sono uguali si passa al carattere successivo e così via. Ad esempio, la stringa *"alfa"* è minore sia di *"beta"* che di *"alfio"*.

L'unico operatore sulle stringhe è l'*operatore di concatenazione*, rappresentato dal simbolo +. È un operatore binario che restituisce la *giustapposizione* di due stringhe. Ad esempio, il valore di *"Java" + "Script"* è *"JavaScript"*.

4.2 Valutazione delle espressioni

La *valutazione di un'espressione* avviene secondo regole ben precise. Si valutano prima le espressioni più interne e, utilizzando i valori ottenuti, si valutano le altre. Al termine di questo processo si ottiene il valore dell'espressione.

L'*ordine di valutazione* delle espressioni può essere alterato, tenendo in considerazione la *precedenza* degli operatori. Gli operatori con una precedenza più alta sono valutati prima degli altri. A parità di precedenza si valuta l'operatore più a sinistra. Queste due regole rendono univoca la valutazione di un'espressione, evitando possibili ambiguità. Ad esempio, il valore dell'espressione *2 + 3 * 4* può essere calcolato in due modi diversi. Un modo prevede il calcolo della somma tra *2* e *3* (risultato *5*) e poi il prodotto tra *5* e *4* (risultato *20*). Un altro, invece, prevede prima il prodotto tra *3* e *4* (risultato *12*) e poi la somma tra *2* e *12* (risultato *14*). Dando precedenza all'operatore di moltiplicazione rispetto a quello di addizione, si elimina questa ambiguità: il valore dell'espressione *2 + 3 * 4* è univocamente determinato ed è *14*.

La precedenza degli operatori è la seguente, in ordine di precedenza maggiore:

- segno negativo, segno positivo, negazione

- moltiplicazione, divisione, modulo, congiunzione, disgiunzione
- addizione, sottrazione
- operatori di confronto.

Le parentesi tonde permettono di alterare arbitrariamente l'ordine di valutazione determinato dalla priorità degli operatori. Ad esempio, il valore dell'espressione *(2 + 3) * 4* è *20* anziché *14*.

4.3 Casi particolari

L'addizione e la moltiplicazione possono dare come risultato il valore *Infinity* che rappresenta un numero troppo grande per essere rappresentato. Nel caso della moltiplicazione questo valore si può ottenere valutando l'espressione *1E308 * 2* oppure dividendo per zero un numero intero. Il valore *–Infinity*, che rappresenta un numero troppo piccolo per essere rappresentato, si ottiene valutando l'espressione *–1E308 * 2* oppure dividendo per zero un numero negativo.

Il modulo si basa su una versione leggermente modificata della definizione euclidea della divisione, in cui il resto è sempre un numero positivo. Se il primo operando è negativo il risultato è negativo. Ad esempio, valutando l'espressione *–10%3* si ottiene il valore *–1*.

4.4 Conversione implicita

Gli operatori numerici assumono che i due operandi siano numerici. Cosa succede se uno di questi operandi appartiene a un altro tipo? La risposta è articolata e si basa sul concetto di *conversione implicita di tipo*. Affrontiamo questo argomento per tutti gli operatori.

Un valore booleano che compare come operando di un operatore numerico è convertito in un numero. In particolare, il valore *true* è convertito nel valore 1, il valore *false* è convertito nel valore zero. La valutazione dell'espressione *1 + true* ha come risultato *2*, la valutazione dell'espressione *1 + false* ha come risultato *1*.

Se una stringa compare come operando di un operatore numerico, di norma il risultato è la costante *NaN* (*Not a Number*). Tuttavia, se la stringa rappresenta un numero, l'operatore converte la stringa nel numero corrispondente ed effettua correttamente l'operazione, restituendo un numero. Ad esempio, la valutazione dell'espressione *2 * '2'* restituisce il valore *4*, anziché *NaN*. L'operatore di addizione introduce un'ulteriore regola: se uno dei due operandi è una stringa che non rappresenta un numero, il risultato è una stringa ottenuta giustapponendo il valore del primo operando con quello del secondo. Ad esempio, la valutazione dell'espressione *200E3 + 'a'* restituisce il valore *200000a*.

Se l'operatore booleano di negazione ha come operando un numero diverso da zero o un carattere il risultato è *false*. Se il valore dell'operando è zero il risultato è *true*.

Diverso è il comportamento degli operatori di congiunzione e di disgiunzione. Se il primo operando dell'operatore di congiunzione (disgiunzione) vale *false* (*true*) il risultato è sempre *false* (*true*). Se il primo operando dell'operatore di congiunzione (disgiunzione) vale *true* (*false*) il risultato è sempre il valore del secondo operando.

4.5 Esercizi

I seguenti problemi devono essere risolti usando costanti e operatori e visualizzando il risultato con il comando di stampa.

1. Calcolare la somma dei primi quattro multipli di 13.
2. Verificare se la somma dei primi sette numeri primi è maggiore della somma delle prime tre potenze di due.
3. Verificare se 135 è dispari, 147 è pari, 12 è dispari, 200 è pari.
4. Calcolare l'area di un triangolo rettangolo i cui cateti sono 23 e 17.
5. Calcolare la circonferenza di un cerchio il cui raggio è 14.
6. Calcolare l'area di un cerchio il cui diametro è 47.
7. Calcolare l'area di un trapezio la cui base maggiore è 48, quella minore è 25 e l'altezza è 13.
8. Verificare se l'area di un quadrato di lato quattro è minore dell'area di un cerchio di raggio tre.
9. Calcolare il numero dei minuti di una giornata, di una settimana, di un mese di 30 giorni, di un anno non bisestile.
10. Verificare se conviene acquistare una camicia che costa 63 € in un negozio che applica uno sconto fisso di 10 € o in un altro che applica uno sconto del 17%.

5 Variabili e assegnamento

Per descrivere un calcolo è necessario tener traccia dei valori intermedi, memorizzandoli per usarli in seguito. Nei linguaggi di programmazione questo ruolo è svolto dalle *variabili*. In JavaScript, come negli altri linguaggi di programmazione imperativi, una *variabile* è un *identificatore* a cui è associato un valore.

```
<Identificatore>  ::= <CarIniziale>
                    |  <CarIniziale> <Caratteri>

<CarIniziale>     ::= <Lettera>
                    |
                    |  _
                    |  $

<Caratteri>       ::= <CarNonIniziale>
                    |  <CarNonIniziale> <Caratteri>

<CarNonIniziale>  ::= <Lettera>
                    |  <Cifra>
                    |
                    |  _
                    |  $
```

Un identificatore è formato da una sequenza di lettere e cifre e dai caratteri _ e $. Questa sequenza deve iniziare con una lettera o con uno dei caratteri _ e $ ma non può iniziare con una cifra.

Non tutti gli identificatori sono utilizzabili come variabili. In JavaScript, infatti, le *parole riservate* non possono essere usate come variabili. Le parole riservate usate nel libro sono le seguenti.

```
break, case, default, else, false, for, function, if, in, new, null,
return, switch, this, true, var, while
```

La *dichiarazione di variabile* è un comando che definisce una variabile associandole un identificatore. Quando la dichiarazione comprende l'assegnamento di un valore si parla più propriamente di *inizializzazione*. Se la variabile non è stata inizializzata il suo valore è il valore speciale *undefined*.

```
<Dichiarazione>   ::= <DicVariabile>

<DicVariabile>    ::= var <Identificatore>;
                    |  var <Identificatore> = <Espressione>;
```

5.1 Dichiarazione di costante

Una convenzione molto diffusa prevede che le costanti definite in un programma abbiano un identificatore formato solo da lettere maiuscole, da numeri e dal carattere _. Ciò permette di distinguerle immediatamente dalle variabili. La costante π (*pi greco*), il cui valore approssimato alle prime dieci cifre decimali è 3,1415926535, può essere dichiarata come una variabile con un valore iniziale.

```
 var PI_GRECO =  3,1415926535;
```

5.2 Variabili ed espressioni

Le variabili possono essere usate nelle espressioni. In particolare, una variabile è un'espressione semplice il cui valore è proprio quello della variabile.

```
<Espressione> ::= <Identificatore>
```

La possibilità di dichiarare variabili non inizializzate, il cui valore è *undefined*, richiede di definire il comportamento degli operatori per trattare questo caso particolare. Se almeno uno degli operandi di un operatore numerico è il valore *undefined*, il risultato sarà *NaN*.

Molto più complessa è la casistica relativa agli operatori booleani. Se l'operatore di negazione ha un operando che vale *undefined* il risultato è *true*. Se gli operandi dell'operatore di congiunzione valgono *undefined* il risultato è *undefined*, tranne quando il primo operando vale *false*, nel qual caso il risultato è *false*. Se il primo operando dell'operatore di disgiunzione vale *undefined* il risultato è *true* se il secondo operando vale *true*, *false* se il secondo operando vale *false*. Se, invece, il primo operando vale *false* il risultato è *undefined*. Infine, se entrambi gli operandi dell'operatore di congiunzione o di quello di disgiunzione valgono *undefined*, il risultato è *undefined*.

Gli operatori di confronto seguono una logica più semplice. Se solo un operando vale *undefined*, il risultato è sempre *false*, tranne nel caso dell'operatore di disuguaglianza, per il quale il risultato è *true*. Se i due operandi valgono *undefined*, il risultato è sempre *false*.

5.3 Comando di assegnamento

Il *comando di assegnamento* modifica il valore associato a una variabile, assegnandole quello di un'espressione.

```
<ComandoSemplice> ::= <Assegnamento>

<Assegnamento>    ::= <Identificatore> = <Espressione>;
```

Vediamo alcuni esempi di dichiarazione di variabile, di assegnamento e di uso delle variabili nelle espressioni.

```
var x = 11;
print(x);
var y;
print(y);
y = 7;
print(x + y);
x = y + 10;
print(x);
x = x + 10;
print(x);
```

La prima dichiarazione definisce la variabile *x* e le assegna il valore *11*. Il comando di stampa successivo stampa il valore di *x*. La seconda dichiarazione definisce la variabile *y*, senza assegnarle un valore iniziale. Pertanto, il comando di stampa successivo stampa il valore *undefined*. Il primo comando di assegnamento assegna il valore *7* a *y*. Il successivo comando stampa il valore *18*, ottenuto sommando *11* a *7*. Il secondo comando di assegnamento assegna a *x* la somma del valore di *y* e di *10*, in questo caso *17*, come si può verificare osservando l'effetto del successivo comando di stampa. L'ultimo comando di assegnamento assegna a *x* il suo valore sommato a *10*, in questo caso *27*, risultato verificabile osservando l'effetto dell'ultimo comando di stampa.

5.4 Abbreviazioni del comando di assegnamento

In JavaScript è possibile usare alcune *abbreviazioni* per incrementare o decrementare il valore di una variabile. Il comando $i = i + 1$ è equivalente a $i{+}{+}$ e il comando $i = i - 1$ è equivalente a $i{-}{-}$.

Un'altra caratteristica di JavaScript è la possibilità di abbreviare la forma del comando di assegnamento, quando è riferito a una variabile. Per incrementare il valore della variabile *x* del valore della variabile *i*, anziché scrivere $x = x + i$ è possibile usare la forma più compatta $x \mathrel{+}= i$, il cui significato è: applica l'operatore + alla variabile *x* e all'espressione alla destra del segno = e assegna il risultato alla variabile *x*. Lo stesso ragionamento è applicabile agli altri operatori numerici.

```
<Assegnamento> ::= <Identificatore>++;
                 |  <Identificatore>--;
                 |  <Identificatore> += <Espressione>;
                 |  <Identificatore> -= <Espressione>;
                 |  <Identificatore> *= <Espressione>;
                 |  <Identificatore> /= <Espressione>;
                 |  <Identificatore> %= <Espressione>;
```

5.5 Esercizi

Risolvere i seguenti problemi utilizzando variabili, costanti e operatori. Il risultato deve essere visualizzato mediante il comando di stampa *print*, disponibile nell'ambiente *EasyJS*.

1. Calcolare il costo di un viaggio in automobile, sapendo che la lunghezza è 750 Km, che il consumo di gasolio è 3,2 litri ogni 100 Km, che un litro di gasolio costa 1,432 €, che due terzi del percorso prevedono un pedaggio pari a 1,2 € ogni 10 Km.
2. Calcolare il costo di una telefonata, sapendo che la durata è pari a 4 minuti e 23 secondi, che il costo alla chiamata è pari a 0,15 €, che i primi 30 secondi sono gratis, che il resto della telefonata costa 0,24 € al minuto.
3. Calcolare il costo di un biglietto aereo acquistato una settimana prima della partenza, sapendo che il costo di base è pari a 200 € (se acquistato il giorno della partenza) e che questo costo diminuisce del 2,3% al giorno (se acquistato prima del giorno della partenza).
4. Calcolare il costo di un prodotto usando la seguente formula

   $$costo = (prezzo + prezzo \cdot 0{,}20) - sconto$$

   e sapendo che il prezzo è 100 € e lo sconto è 30 €.
5. Calcolare la rata mensile di un mutuo annuale usando la seguente formula

   $$rata = \frac{importo}{12} \cdot (1 + tasso)$$

   e sapendo che l'importo annuale è 240 € e il tasso è il 5%.

6 Funzioni

Una *dichiarazione di funzione* è un comando che definisce un identificatore a cui è associata una *funzione*. La definizione della funzione comprende un'*intestazione* e un blocco di comandi.

```
<Dichiarazione> ::= <DicFunzione>

<DicFunzione>   ::= function <Identificatore>()
                      <Blocco>
                  |  function <Identificatore>(<Parametri>)
                     <Blocco>

<Parametri>     ::= <Identificatore>
                  |  <Identificatore>, <Parametri>
```

L'intestazione della funzione definisce la lista dei suoi *parametri* (chiamati anche *parametri formali*) racchiusi tra due parentesi tonde. La lista dei parametri formali può essere vuota.

Come esempio, definiamo la funzione *stampaSomma* che stampa la somma dei suoi parametri.

```
function stampaSomma(n, m) {
   print(n + m);
}
```

Dopo aver dichiarato una funzione, è possibile usarla in un punto qualunque del programma. Il punto in cui si usa una funzione è detto punto di *chiamata* o di *invocazione*. Quando si invoca la funzione, si assegna a ciascun parametro formale il valore dell'espressione utilizzata nel punto di chiamata e poi si esegue il blocco di comandi.

Le espressioni utilizzate nel punto di invocazione (chiamate anche *parametri attuali*) sono valutate prima dell'esecuzione del blocco di comandi associato alla funzione e il loro valore è associato ai parametri formali. Questo procedimento è detto *passaggio dei parametri*. L'associazione tra i parametri formali e i valori dei parametri attuali avviene con il *sistema posizionale*, ovvero ad ogni parametro formale è associato il valore del parametro attuale che occupa la medesima posizione nella rispettiva lista. Il primo parametro attuale è dunque legato al

primo parametro formale, il secondo parametro attuale è legato al secondo parametro formale e così via.

```
<ComandoSemplice> ::= <Invocazione>

<Invocazione>     ::= <Identificatore> ();
                    |  <Identificatore> (<Espressioni>);

<Espressioni>     ::= <Espressione>
                    |  <Espressione>, <Espressioni>
```

Gli esempi che seguono mostrano due invocazioni di funzione con un numero di parametri attuali pari a quello dei parametri formali. L'effetto delle invocazioni è, rispettivamente, stampare prima il valore *5* e poi il valore *15*.

```
stampaSomma(10, -5);
stampaSomma(10, 5);
```

Normalmente il numero di parametri attuali è uguale a quello dei parametri formali definiti nella dichiarazione. Se il numero dei parametri attuali è diverso da quello dei parametri formali, non si ottiene un errore, ma:

- se il numero dei parametri attuali è minore di quello dei parametri formali, il valore *undefined* è assegnato ai parametri formali che non hanno un corrispondente parametro attuale, ovvero agli ultimi della lista;
- se il numero di parametri attuali è maggiore di quello dei parametri formali, i parametri attuali in eccesso sono ignorati.

Una funzione può *restituire* un valore al termine della propria esecuzione, eseguendo il comando *return* seguito da un'espressione.

```
<ComandoSemplice> ::= return <Espressione>;
```

Ad esempio, definiamo una funzione che calcola e restituisce la somma dei suoi parametri.

```
function calcolaSomma(x, y) {
   return x + y;
}
```

Una funzione che restituisce un valore può essere invocata in un qualunque punto di un programma in cui è lecito usare un'espressione, ad esempio in un comando di assegnamento o in un comando di stampa.

```
var x = 1;
var y = 2;
x = calcolaSomma(x, y);
print(calcolaSomma(x, y));
```

La sintassi delle espressioni è estesa con l'invocazione di funzione.

```
<Espressione>     ::= <Identificatore>()
                    |  <Identificatore>(<espressioni>)
```

Le funzioni che restituiscono un valore booleano sono chiamate *predicati*. Come esempio, definiamo il predicato *compreso* che verifica se *x* appartiene all'*intervallo* [*a*, *b*)[5].

```
function compreso(x, a, b) {
   return (x >= a) && (x < b);
}
```

6.1 Visibilità

Quando si dichiara una variabile o una funzione è necessario tenere in debita considerazione la loro *visibilità*. In pratica è necessario sapere quali variabili sono definite nel programma e quali sono, invece, definite nel corpo della funzione. Il modello adottato da JavaScript è complesso, essendo basato sull'esecuzione del programma ma anche sulla sua struttura sintattica. Ai fini di questo libro, tuttavia, si ritiene sufficiente presentare una versione semplificata del modello, versione che permette di spiegare il comportamento degli esempi riportati.

Un *ambiente* è formato da un insieme di variabili e di funzioni. In un ambiente non possono esserci due variabili che hanno lo stesso identificatore, altrimenti non sarebbe possibile sapere quale dei due dovrà essere utilizzato al momento della valutazione della variabile. Allo stesso modo non ci possono essere due funzioni con lo stesso identificatore.

Un ambiente è modificato dalle dichiarazioni, che introducono nuove variabili e nuove funzioni, e dai comandi, che modificano il valore delle variabili presenti nell'ambiente. Ogni punto di un programma ha un ambiente, formato dalle variabili definite in quel punto. Al contrario delle variabili, tutte le funzioni dichiarate in un programma fanno parte del suo ambiente.

Ogni variabile è visibile in una porzione ben definita del programma in cui essa è dichiarata. In particolare, la variabile è visibile dal punto in cui è dichiarata la prima volta (o assegnata la prima volta) fino al termine del programma. Una variabile visibile solo in alcune porzioni del programma è detta *variabile locale*. Una variabile la cui visibilità è l'intero programma è detta *variabile globale*. Una variabile globale può essere nascosta da una variabile locale che ha lo stesso identificatore. Un variabile nascosta è definita, ma non è accessibile.

5 L'intervallo [*a, b*) è formato dai numeri maggiori o uguali di *a* e minori di *b*. Per convenzione si assume che *a* sia minore o uguale di *b*. Quando *a* è uguale a *b* l'intervallo è vuoto.

In JavaScript le uniche variabili locali sono quelle dichiarate nel corpo delle funzioni. I parametri formali di una funzione sono trattati come variabili dichiarate al suo interno e quindi locali. Tutte le altre variabili sono globali.

```
var risultato = 0;
function calcolaSomma(x, y) {
   var somma = 0;
   somma = x + y;
   return somma;
}
risultato = calcolaSomma(2, 4);
```

In questo esempio compaiono le variabili *x*, y, *somma* e *risultato*. Le variabili *x*, *y* e *somma* sono locali al corpo della funzione *calcolaSomma*. La variabile *risultato* è globale. In base alla definizione di visibilità, le variabili locali sono utilizzabili solo nel corpo di *calcolaSomma*, mentre la variabile globale *risultato* è utilizzabile sia nel corpo di *calcolaSomma*, sia nel resto del programma.

Il seguente programma sfrutta la visibilità delle variabili per ottenere lo stesso risultato.

```
var risultato = 0;
function calcolaSomma(x, y) {
   var somma = 0;
   somma = x + y;
   risultato = somma;
}
calcolaSomma(2, 4);
```

Questo esempio mostra come sia possibile modificare il valore di una variabile globale all'interno del corpo di una funzione. L'utilità di questa tecnica sarà apprezzata nella seconda parte del libro, quando presenteremo problemi che richiedono soluzioni complesse.

6.2 Funzioni predefinite

JavaScript mette a disposizione numerose funzioni matematiche di uso corrente, chiamate *funzioni predefinite*. Un elenco non esaustivo di queste funzioni è il seguente:

- *Math.abs* (*x*): restituisce il valore assoluto di *x*,
- *Math.ceil* (*x*): restituisce il primo intero più grande di *x*,
- *Math.floor* (*x*): restituisce il primo intero più piccolo di *x*,
- *Math.log* (*x*): restituisce il *logaritmo* naturale in base *e* di *x*,
- *Math.pow* (*x*, *y*): restituisce *x* elevato alla potenza di *y*,

- *Math.random* (): restituisce un numero casuale compreso tra zero e uno,
- *Math.round* (x): restituisce l'intero più vicino a *x*,
- *Math.sqrt* (*x*): restituisce la radice quadrata di *x*.

6.3 Esercizi

1. Definire in JavaScript un predicato che verifica se l'intersezione dell'intervallo [*a*, *b*) con l'intervallo [*c*, *d*) è vuota.

   Il predicato ha quattro parametri: *a*, *b*, *c*, *d*.

   Invocare il predicato con i seguenti valori: *2*, *4*, *5*, *7*; *2*, *4*, *4*, *7*; *2*, *4*, *3*, *7*; *5*, *7*, *2*, *4*; *4*, *7*, *2*, *4*; *3*, *7*, *2*, *4*.
2. L'*equazione di secondo grado* $ax^2 + bx + c = 0$ ha due radici [Wikipedia, alla voce *Equazione di secondo grado*]:

   $$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   Definire in JavaScript una funzione che stampa le radici di un'equazione i cui coefficienti sono *a*, *b* e *c* (con *a* diverso da zero).

   La funzione ha tre parametri:*a*, *b*, *c*.

   Invocare la funzione con i seguenti valori: *1*, *-5*, *6*; *1*, *8*, *16*; *1*, *2*, *3*.
3. Definire in JavaScript una funzione che calcola e restituisce la somma delle cifre di un intero che appartiene all'intervallo [*0, 100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *23*, *99*.

7 Comandi condizionali

La sintassi di JavaScript che abbiamo definito finora permette di scrivere programmi e funzioni molto semplici, formati da una sequenza di dichiarazioni di variabili, di assegnamenti e di comandi di stampa. Con questi comandi è possibile affrontare solo problemi la cui soluzione richiede una sequenza di calcoli. Problemi più complessi richiedono l'uso di comandi composti. In questo capitolo presentiamo i *comandi condizionali*.

7.1 Comando condizionale

Il *comando if* è un comando composto che consente di scegliere i comandi da eseguire in base al valore di un'espressione booleana.

```
<ComandoComposto> ::= <If>
                    |  <Blocco>

<If>              ::= if (<Espressione>)
                         <Blocco>
                    |  if (<Espressione>)
                         <Blocco>
                       else <Blocco>

<Blocco>          ::= {<Comandi>}
```

Il comando inizia con la parola riservata *if*, seguita da un'espressione tra parentesi tonde, detta *condizione*. Il valore della condizione deve essere un booleano. Se la condizione vale *true* (la condizione è vera), si esegue il primo *blocco di comandi*, detto *blocco*. Se la condizione vale *false* (la condizione è falsa) ci possono essere due casi: se è presente ed è preceduto dalla parola riservata *else*, viene eseguito il secondo blocco, altrimenti non si esegue alcun comando.

Un blocco è costituito da una *sequenza di comandi* racchiusa tra *parentesi graffe*. Un blocco è considerato un comando unico e permette di utilizzare una sequenza di comandi dove normalmente sarebbe possibile usarne solo uno.

Nel seguito mostriamo un programma che usa due comandi condizionali. Nel primo comando la condizione è vera se il valore della variabile *x* è maggiore di zero: in tal caso il suo valore è decrementato di uno. Non è prevista alcuna azione nel caso in cui il valore di *x* sia minore o uguale a zero. Il secondo comando è

un esempio che mostra come sia possibile assegnare a x un valore maggiore di zero, indipendentemente dal valore di y. Se il valore di y è minore di zero e quindi la condizione è vera, a x è assegnato tale valore cambiato di segno (e dunque positivo). In caso contrario a x è assegnato il valore di y.

```
if (x > 0){
   x--;
}
if (y < 0){
   x = -y;
} else {
   x = y;
}
```

I comandi condizionali possono essere *annidati* per effettuare una scelta tra più di due possibilità. Il seguente esempio mostra due comandi condizionali annidati e mostra anche come sia possibile evidenziare la struttura di un programma mediante l'*indentazione*.

```
if (a < 6){
   b = 1;
} else {
   if (a < 8){
      b = 2;
   } else {
      b = 3;
   }
}
```

Un'altra indentazione è la seguente, preferibile alla prima quando sono presenti più di due alternative.

```
if (a < 6){
   b = 1;
} else if (a < 8){
   b = 2;
} else {
   b = 3;
}
```

7.2 Comando di scelta multipla

Il *comando di scelta multipla* è un comando composto che rappresenta un'alternativa a una sequenza di comandi condizionali annidati. Si usa quando si devono eseguire comandi diversi, associati a determinati valori di un'espressione.

```
<ComandoComposto> ::=<Switch>

<Switch>          ::= switch (<Espressione>)
                         {<Alternative>}
                    | switch (<Espressione>)
                         {<Alternative> default: <Comandi>}

<Alternativa>     ::= case <Costante>: <Comandi>

<Alternative>     ::= <Alternativa>
                    | <Alternativa> <Alternative>

<ComandoSemplice> ::= break;
```

L'espressione (chiamata anche *selettore*) è valutata e il suo valore è confrontato con quello delle costanti di ciascuna alternativa, partendo dalla prima. Se il confronto ha esito positivo per un'alternativa, si eseguono i comandi ad essa associati e quelli di tutte le alternative successive. Se il confronto ha esito negativo per tutte le alternative, non si esegue alcun comando.

```
switch (x) {
   case 5 : y += 5;
   case 6 : z += 6;
}
```

In questo esempio, se il valore dell'espressione *x* è uguale a *6*, la variabile *z* è incrementata di *6*. Se il valore è *5*, la variabile *y* è incrementata di *5* e la variabile *z* è incrementata di *6*. Se il valore è diverso da *5* o da *6*, non si esegue alcun comando.

In molti casi è preferibile che dopo l'esecuzione del codice associato a un'alternativa l'esecuzione passi al comando successivo al comando di selezione multipla. Per ottenere questo comportamento l'ultimo comando di quelli associati a un'alternativa deve essere il comando *break*.

```
switch (x) {
   case 5 : y += 5; break;
   case 6 : z += 6; break;
}
```

Con questa modifica, anche quando il valore del selettore è uguale a *5*, la variabile *z* non è incrementata.

È possibile inserire, rigorosamente per ultima, un'alternativa speciale i cui comandi sono eseguiti quando il confronto ha avuto esito negativo per tutte le alternative.

```
switch (x) {
   case  5 : y += 5; break;
   case  6 : z += 6; break;
   default : z++;
}
```

Con questa modifica, in tutti i casi in cui il valore del selettore è diverso da *5* o da 6, la variabile *z* è incrementata di uno.

7.3 Anno bisestile

L'*anno bisestile* è un anno solare in cui il mese di febbraio ha 29 giorni anziché 28. Questa variazione evita lo slittamento delle stagioni che ogni quattro anni accumulerebbero un giorno in più di ritardo. Nel *calendario giuliano* è bisestile un anno ogni quattro (quelli la cui numerazione è divisibile per quattro). Nel *calendario gregoriano* si mantiene questa variazione ma si eliminano tre anni bisestili ogni 400 anni [Wikipedia, alla voce *anno bisestile*].

Il predicato *bisestile* verifica se un anno è bisestile.

```
function bisestile(anno) {
   if (anno % 400 == 0) {
      return true;
   } else if (anno % 100 == 0) {
      return false;
   } else if (anno % 4 == 0) {
      return true;
   } else {
      return false;
   }
}
```

Una versione più semplice del predicato *bisestile* non fa uso del comando condizionale.

```
function bisestile(anno) {
   return (anno % 400 == 0) ||
          ((anno % 4 == 0) && (anno % 100 != 0));
}
```

7.4 Esercizi

1. I giorni di un anno possono essere numerati consecutivamente, assumendo che al primo gennaio sia assegnato il valore *1* e al trentuno dicembre il valore *365* negli anni non bisestili o *366* negli anni bisestili.

   Definire in JavaScript una funzione che restituisce il numero assegnato a un giorno dell'anno, assumendo che i mesi siano numerati da *1* a *12*.

La funzione ha tre parametri: *anno*, *mese*, *giorno*.

Invocare la funzione con i seguenti valori: *1957*, *4*, *25*; *2004*, *11*, *7*; *2000*, *12*, *31*; *2012*, *2*, *29*.

2. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta il valore in lettere di un intero che appartiene all'intervallo [*0*, *100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *12*, *21*, *32*, *43*, *70*, *88*, *90*.

3. Definire in JavaScript una funzione che calcola il tipo di un triangolo (*equilatero*, *isoscele*, *rettangolo*, *scaleno*) in base alla lunghezza dei suoi lati.

   La funzione restituisce una stringa che rappresenta il tipo del triangolo e ha tre parametri : *a*, *b*, *c*.

   Invocare la funzione con i seguenti valori: *3*, *3*, *3*; *3*, *4*, *4*; *3*, *4*, *5*; *3*, *4*, *6*.

8 Comandi iterativi

Molti problemi richiedono un calcolo che deve essere ripetuto più volte per ottenere il risultato finale. In JavaScript la ripetizione di un calcolo si ottiene utilizzando un *comando iterativo*.

8.1 Comando iterativo determinato

Il *comando iterativo determinato* esegue un blocco di comandi un numero determinato di volte. Il comando è formato da un'intestazione che contiene un comando di inizializzazione di una variabile (chiamata anche *indice di iterazione*) che conta il numero delle iterazioni, un'espressione (chiamata *guardia*) che controlla quante volte il blocco di comandi è eseguito, un comando che aggiorna il valore dell'indice di iterazione.

La forma sintattica del comando iterativo determinato è molto generale ma in questo libro ne adotteremo una che ci permette di risolvere i problemi che richiedono l'uso di un'iterazione determinata.

```
<ComandoComposto> ::= <For>
<For> ::= for (<Comando>; <Espressione>; <Comando>)
             <Blocco>
```

L'esecuzione di un comando iterativo determinato segue il seguente schema:

- l'indice di iterazione è inizializzato;
- la guardia è valutata prima di eseguire il blocco di comandi;
- se il valore della guardia è *true* (cioè se la guardia è verificata), il blocco è eseguito, l'indice di iterazione è aggiornato e il ciclo è ripetuto;
- se il valore della guardia è *false* (cioè se la guardia non è verificata), il blocco non è eseguito e l'esecuzione dell'intero comando termina.

La seguente funzione calcola la somma dei numeri da uno a *n*.

```
function somma(n) {
   var s = 0;
   for (var i = 1; i <= n; i++) {
      s += i;
   }
```

```
   return s;
}
```

La funzione può essere definita usando un comando iterativo determinato che somma i numeri da *n* a uno.

```
function somma(n) {
   var s = 0;
   for (var i = n; i > 0; i--) {
      s += i;
   }
   return s;
}
```

8.2 Comando iterativo indeterminato

Il *comando iterativo indeterminato* è utilizzato quando un blocco di comandi deve essere eseguito più volte, ma non è possibile sapere a priori quante. Il comando è formato da un'espressione, chiamata guardia, e un blocco di comandi.

```
<ComandoComposto> ::= <While>
<While>           ::= while (<Espressione>)
                          <Blocco>
```

L'esecuzione di un comando iterativo indeterminato segue il seguente schema:

- la guardia è valutata ad ogni iterazione, prima di eseguire il blocco di comandi;
- se il valore della guardia è *true* (cioè se la guardia è verificata), il blocco è eseguito e poi si ripete il ciclo;
- se il valore della guardia è *false* (cioè se la guardia non è verificata), il blocco non è eseguito e l'esecuzione dell'intero comando termina.

In JavaScript un comando iterativo determinato può sempre essere espresso mediante un comando iterativo indeterminato (ma non viceversa).

```
function sommaA(n) {
   var s = 0;
   var i = 1;
   while (i <= n) {
      s += i;
      i++;
   }
   return s;
}
```

Da un punto di vista pratico è bene utilizzare il comando iterativo determinato quando il numero di volte che sarà eseguito il blocco di comandi è noto a priori, usando il comando iterativo indeterminato in tutti gli altri casi.

8.3 Primalità

Un *numero primo* è un numero naturale maggiore di uno, divisibile solamente per uno e per sé stesso [Wikipedia, alla voce *Numero primo*]. I primi dieci numeri primi sono *2*, *3*, *5*, *7*, *11*, *13*, *17*, *19*, *23*, *29*.

Dato un numero naturale maggiore di uno è possibile verificare se è primo, cioè se soddisfa la condizione di primalità, definendo in JavaScript il predicato *ePrimo* basato su un comando di iterazione determinata.

```
function ePrimo (n) {
   var c = 0;
   for (var i = 2; i < n; i++) {
      if (n % i == 0) {
         c++;
      }
   }
   return (c == 0);
}
```

Una soluzione alternativa si basa su una variabile booleana.

```
function ePrimo (n) {
   var b = true;
   for (var i = 2; i < n; i++) {
      b = b && (n % i != 0);
   }
   return b;
}
```

L'espressione usata per aggiornare il valore della variabile booleana può essere utilizzata come guardia di un comando iterativo indeterminato.

```
function ePrimo (n) {
   var i = 2;
   while ((i < n) && (n % i != 0)) {
      i++;
   }
   return (i == n);
}
```

La soluzione basata sul comando iterativo indeterminato può essere migliorata tenendo conto del fatto che se n è primo allora non esiste un divisore d di n minore della sua radice quadrata:

$$d < \sqrt{n}$$
$$d^2 < n$$

```
function ePrimo (n) {
   var i = 2;
   while ((i * i < n) && (n % i != 0)) {
      i++;
   }
   return (i * i > n);
}
```

8.4 Radice quadrata

La *radice quadrata* di un numero razionale non negativo *z* è un numero *x*, anch'esso non negativo, che soddisfa l'equazione

$$x^2 = z$$

[Wikipedia, alla voce *Radice quadrata*]. Ad esempio, la radice quadrata di *2* è *1,4142135623*.

La *radice quadrata intera* di un numero razionale non negativo *z* è un intero positivo *x* che soddisfa l'equazione

$$x^2 \leq z < (x+1)^2$$

Ad esempio, la radice quadrata intera di *5* è *2*. In JavaScript si può definire una funzione che calcola la radice quadrata intera di un numero razionale non negativo.

```
function radiceQuadrata(z) {
   var x = 0;
   while (x * x <= z) {
      x++;
   }
   return x - 1;
}
```

8.5 Esercizi

1. Un anno è perfetto per una persona se è un multiplo della sua età in quell'anno, assumendo come età massima cento anni. Ad esempio gli anni perfetti per chi è nato nel *1984* sono nove: *1985*, *1986*, *1988*, *1992*, *2000*, *2015*, *2016*, *2046*, *2048*.

   Definire in JavaScript una funzione che stampa gli anni perfetti relativi a un anno di nascita.

   La funzione ha un parametro: *annoDiNascita*.

Invocare la funzione con i seguenti valori: *1984*, *1990*, *1992*, *2000*, *2009*.

2. In matematica, con reciproco di un numero x si indica il numero che moltiplicato per x dà come risultato 1.

   Definire in JavaScript una funzione che calcola e restituisce la somma dei reciproci dei primi n numeri interi maggiori di zero.

   La funzione ha un parametro: n.

   Invocare la funzione con i seguenti valori: *1*, *2*, *4*, *7*.

3. Definire in JavaScript una funzione che stampa, se esistono, le radici intere di un'equazione di secondo grado di coefficienti a, b, c comprese nell'intervallo $[l, u)$.

   La funzione ha i seguenti parametri: a, b, c, l, u.

   Invocare la funzione con i seguenti valori:

   1, *−2*, *−8*, *1*, *5*

   1, *−2*, *−8*, *−5*, *5*

   1, *−2*, *−8*, *5*, *10*.

9 Array

In JavaScript, oltre ai numeri, i booleani e i caratteri, esistono altri tipi di dato che, per come sono strutturati, sono chiamati *tipi composti*, per differenziarli da quelli primitivi. In questo capitolo presentiamo gli array, un tipo di dato indispensabile per la soluzione di numerosi problemi.

9.1 Elementi e indici di un array

Un *array* è un tipo composto, formato da una sequenza numerata di valori[6]. Ogni valore è detto *elemento* dell'array e il numero a esso associato è detto *indice*. Il primo elemento di un array ha sempre indice zero.

Per dichiarare un array è necessario indicarne il nome, un identificatore, e un valore determinato dall'espressione a destra del segno di assegnamento. La categoria sintattica delle espressioni è così estesa.

```
<Espressione>  ::= []
                |  [<Espressioni>]
```

Vediamo alcuni esempi di dichiarazione di array.

```
var a = [];
var b = [12];
var c = ['f', 'u', 'n', 'e']
```

Nel primo caso si ottiene un array vuoto. Nel secondo caso un array con un solo elemento, *12*. Nel terzo caso un array con quattro elementi, i caratteri *f*, *u*, *n*, ed *e*.

Una volta creato un array, si può accedere ai suoi elementi per utilizzarne i valori. L'espressione tra parentesi quadre deve avere un valore maggiore o uguale a zero.

```
<Espressione>  ::= <Identificatore>[<Espressione>]
```

Vediamo un esempio in cui si dichiara un array e si utilizzano i suoi elementi.

6 In JavaScript gli array possono essere *omogenei* (tutti i valori sono dello stesso tipo) o *disomogenei* (i valori sono di più tipi). Per semplicità di presentazione assumiamo che gli array siano omogenei.

```
var a = [2, 3, 5, 7, 11];
print(a[0] + a[1] + a[2] + a[3] + a[4]);
```

Il valore degli elementi di un array può essere modificato mediante il comando di assegnamento. Anche in questo caso l'espressione tra parentesi quadre deve avere un valore maggiore o uguale a zero.

```
<Assegnamento> ::= <Identificatore>[<Espressione>] =
                                    <Espressione>;
```

Vediamo un esempio in cui si dichiara l'array *a* e si modifica il secondo elemento, il cui indice è uno.

```
var a = [2, 3, 5, 7, 11];
a[1] = 0;
print(a[0] + a[1] + a[2] + a[3] + a[4]);
```

Per scandire gli elementi di un array in JavaScript si usa il comando di iterazione determinata. La funzione *stampaElementi* stampa tutti gli elementi di un array. La funzione è invocata prima con un array di tre elementi e poi con uno di cinque elementi.

```
function stampaElementi(a) {
   for (i in a) {
      print(a[i]);
   }
}
var a1 = [2, 3, 5];
stampaElementi(a1);
var a2 = [0, -9, 17, 4, 100];
stampaElementi(a2);
```

9.2 Lunghezza di un array

Come vedremo nella seconda parte, in JavaScript è possibile definire *oggetti*, tipi di dato composti caratterizzati da un insieme di *proprietà* e di *metodi*.

Una proprietà è un valore associato a un oggetto. Per accedere al valore di una proprietà si utilizza la cosiddetta *notazione a punti* (*dot notation*): l'oggetto è seguito da un punto e dal nome della proprietà.

Un metodo è una funzione associata a un oggetto. Anche per invocare il metodo si utilizza la notazione a punti. Maggiori dettagli su proprietà e metodi saranno forniti nella seconda parte.

Gli array sono *oggetti predefiniti* che hanno la proprietà *length*, il cui valore è pari al numero degli elementi dell'array. Questa proprietà può essere utilizzata per scandire un array, come mostrato nella funzione *stampaElementi*.

```
function stampaElementi(a) {
   for (var i = 0; i < a.length; i++) {
      print(a[i]);
   }
}
```

Anche quando non si intende effettuare una scansione completa di un array è spesso utile usare esplicitamente la proprietà *length*. Ad esempio, la funzione *stampaElementiIndicePari*, è così definita.

```
function stampaElementiIndicePari(a) {
   for (var i = 0; i < a.length; i += 2) {
      print(a[i]);
   }
}
```

9.3 Array dinamici

In JavaScript, a differenza di altri linguaggi di programmazione, è possibile aggiungere dinamicamente nuovi elementi a un array. Per fare ciò, si esegue un assegnamento all'elemento che si vuole aggiungere, come se già esistesse. Nell'esempio che segue creiamo un array di quattro elementi a cui, dopo, ne aggiungiamo uno.

```
var a = [2, 3, 5, 7, 11];
a[5] = 13;
print(a[0] + a[1] + a[2] + a[3] + a[4] + a[5]);
```

Quando si aggiunge un elemento il cui indice non è immediatamente successivo a quello dell'ultimo elemento definito dell'array, automaticamente tutti gli elementi intermedi sono aggiunti e inizializzati con il valore *undefined*.

```
var a = [2, 3, 5, 7, 11];
a[10] = 13;
```

Nell'esempio precedente si aggiunge l'elemento di indice *10* a un array il cui ultimo elemento definito ha indice *4*. L'aggiunta di questo elemento provoca la creazione degli elementi di indice *5*, *6*, *7*, *8* e *9*, tutti inizializzati con il valore *undefined*.

Per aggiungere un elemento a un array si usa il metodo *push*.

```
var a = [2, 3, 5, 7, 11];
a.push(13);
```

9.4 Array associativi

In JavaScript è possibile utilizzare come indici di un array anche valori di tipo stringa. In questo caso si parla di *array associativi*.

Un array associativo può essere creato in due modi: esplicitamente, mediante l'indicazione dei suoi indici e dei valori associati; implicitamente, mediante assegnamenti che creano dinamicamente l'array. La creazione esplicita di un array associativo formato da tre elementi è indicata nel seguito.

```
var a = {alfa : 10, beta : 20, gamma : 30};
print(a["alfa"]);
print(a["beta"]);
print(a["gamma"]);
```

La creazione implicita dello stesso array è la seguente.

```
var a = {};
a["alfa"] = 10;
a["beta"] = 20;
a["gamma"] = 30;
```

Per gli array associativi la proprietà *length* non è definita perché gli indici di questi array non hanno un valore numerico. Per scandire tutti gli elementi di un array associativo si usa una forma particolare di iterazione determinata in cui l'indice di iterazione assume tutti i valori utilizzati per definire l'array[7].

```
<For> ::= for (var <Identificatore> in <Espressione>)
          <Blocco>
```

Nell'esempio che segue l'indice *i* assume, in sequenza, i valori *alfa*, *beta* e *gamma*.

```
var a = {alfa : 10, beta : 20, gamma : 30};
for (var i in a) {
   print (i);
}
```

9.5 Stringhe di caratteri

Un caso particolare di array è costituito dalle stringhe di caratteri che in JavaScript sono oggetti predefiniti con alcune proprietà e metodi. Tutto quanto detto sugli array si applica alle stringhe. In particolare, anche per le stringhe è definita la proprietà *length*, il cui valore è pari al numero dei caratteri di una stringa.

7 Il valore dell'indice di iterazione non è di tipo numerico ma di tipo stringa.

```
var alfa = "ciao";
print(alfa.length);
```

Per selezionare un carattere di una stringa si utilizza la stessa notazione prevista per gli array: la posizione del carattere è indicata tra parentesi quadre.

```
var alfa = "ciao";
print(alfa[0]);
print(alfa[1]);
print(alfa[2]);
print(alfa[3]);
```

La variabile *alfa* è inizializzata con la stringa *ciao*. I quattro caratteri di *alfa* sono stampati in sequenza. A differenza degli array, tuttavia, non è possibile modificare un carattere di una stringa che, pertanto, è trattata in JavaScript come una costante.

Per scandire i caratteri di una stringa si usa il comando di iterazione determinata.

```
function stampaCaratteri(a) {
   for (var i = 0; i < a.length; i++) {
      print(a[i]);
   }
}
function stampaElementiIndicePari(a) {
   for (var i = 0; i < a.length; i += 2) {
      print(a[i]);
   }
}
```

Le stringhe hanno molti metodi, usati per svolgere operazioni di manipolazione su testi. Quelli più comunemente usati sono i seguenti:

- *substr*: ha due parametri *p* e *n*. Restituisce, a partire dalla posizione indicata dal valore di *p*, *n* caratteri della stringa su cui è invocato.

```
var a = 'alfabeto'
var b = a.substr(2, 4);      // b vale 'fabe'
```

- *indexOf*: ha un parametro *s*. Restituisce l'indice della prima occorrenza della stringa *s*, incontrata a partire da sinistra, nella stringa su cui è invocato. Il valore restituito è *-1* se non vi sono occorrenze di *s*.

```
var a = 'alfabeto'
var b = a.indexOf('beto');    // b vale  4
var c = a.indexOf('Alfa');    // c vale -1
```

- *toLowerCase:* non ha parametri. Restituisce una stringa uguale a quella su cui è invocato, ma con tutti i caratteri minuscoli.

```
var a = 'ALFAbeto'
var b = a.toLowerCase();      // b vale 'alfabeto'
```

Questi metodi non modificano la stringa su cui sono invocati.

9.6 Ricerca lineare

Molti problemi di programmazione che prevedono l'uso di array possono essere risolti utilizzando uno schema chiamato *ricerca lineare*, che può essere *certa* o *incerta*. Lo schema di *ricerca lineare certa* si utilizza quando si ha la certezza a priori che il valore cercato sia presente nell'array. Negli altri casi si utilizza lo schema di *ricerca lineare incerta*.

Lo schema di ricerca lineare certa si basa su un'iterazione indeterminata in cui la guardia è formata da un'unica espressione logica che è vera se la condizione di ricerca non è soddisfatta. Un esempio, molto semplice, di problema che può essere risolto con questo schema è il seguente: dato un array *a*, definire una funzione che calcola e restituisce il valore dell'indice di un elemento di *a* che vale *k*, sapendo a priori che un tale elemento appartiene all'array.

```
function indice(a, k) {
   var i = 0;
   while ((i < a.length) && (a[i] != k)) {
      i++;
   }
   return i;
}
```

Anche lo schema di ricerca lineare incerta si basa su un'iterazione indeterminata e da una guardia formata da due espressioni logiche in congiunzione tra loro: la prima è vera se il valore dell'indice è valido per l'array (cioè se appartiene all'intervallo compreso tra zero e la lunghezza dell'array meno uno), la seconda è vera se la condizione di ricerca non è soddisfatta. Un problema che può essere risolto usando questo schema è il seguente: definire una funzione che calcola e restituisce il valore dell'indice di un elemento di *a* che vale *k*, senza sapere a priori che un tale elemento appartiene all'array.

```
function appartiene(a, k) {
   var i = 0;
   while ((i < a.length) && (a[i] != k)) {
      i++;
   }
   return (i < a.length);
}
```

Alcuni problemi si incontrano spesso in varie formulazioni. La loro soluzione è considerata un classico della programmazione. Di seguito presentiamo alcuni dei più conosciuti.

9.7 Minimo e massimo di un array

Dato un array *a* (non vuoto) di numeri, il *minimo* di *a* è quell'elemento di *a* minore o uguale a tutti gli altri elementi di *a*. Analogamente, il *massimo* di *a* è quell'elemento di *a* maggiore o uguale a tutti gli altri elementi di *a*. La funzione *minimo* calcola e restituisce il minimo di *a*.

```
function minimo(a) {
   var min = a[0];
   for (var i = 1; i < a.length; i++) {
      if (a[i] < min) {
         min = a[i];
      }
   }
   return min;
}
```

La funzione *massimo* calcola e restituisce il massimo di *a*.

```
function massimo(a) {
   var max = a[0];
   for (var i = 1; i < a.length; i++) {
      if (a[i] > max) {
         max = a[i];
      }
   }
   return max;
}
```

9.8 Array ordinato

Un array (non vuoto) è ordinato se ogni coppia adiacente di elementi soddisfa una relazione di ordinamento. In un array ordinato in senso crescente (decrescente) l'ultimo elemento è il massimo (minimo) dell'array e il primo elemento è il minimo (massimo) dell'array. Per ogni coppia di elementi adiacenti, il primo

elemento è minore (maggiore) del secondo. Gli array possono essere anche ordinati in senso non decrescente (non crescente). In questo caso la relazione di ordinamento tra gli elementi adiacenti è quella di minore o uguale (maggiore o uguale).

Il predicato *ordinatoCrescente* verifica se l'array *a* è ordinato in senso crescente.

```
function ordinatoCrescente(a) {
   var c = 0;
   for (var i = 1; i < a.length; i++) {
      if (a[i - 1] < a[i]) {
         c++;
      }
   }
   return (c == (a.length - 1));
}
```

La funzione scandisce tutte le coppie adiacenti, incrementando di uno la variabile *c* per ogni coppia che rispetta l'ordinamento. Se tutte le coppie rispettano l'ordinamento, la funzione restituisce il valore *true*. Una versione basata sullo schema di ricerca lineare incerta è mostrata nel seguito.

```
function ordinatoCrescente(a) {
   var i = 1;
   while ((i < a.length) && (a[i - 1] < a[i])) {
      i++;
   }
   return (i == a.length);
}
```

Il predicato *ordinatoDecrescente* si ottiene a partire da *ordinatoCrescente* invertendo la relazione di ordinamento.

9.9 Filtro

Un *filtro* è uno strumento per la selezione (filtraggio) delle parti di un insieme [Wikipedia, alla voce Filtro]. Un esempio di filtro, chiamato *filtro passa banda*, seleziona tutti i valori che appartengono all'intervallo [*lo*, *hi*). I valori *lo* e *hi* sono chiamati *livello basso* (*low level*) e *livello alto* (*high level*).

La funzione *filtro* ha tre parametri (*a*, *lo*, hi) e restituisce un nuovo array formato da tutti gli elementi di *a* i cui valori sono compresi tra *lo* e *hi*.

```
function filtro(a, lo, hi) {
   var b = [];
   for (var i = 0; i < a.length; i++) {
      if ((a[i] >= lo) && (a[i] < hi)) {
```

```
            b.push(a[i]);
         }
      }
      return b;
}
```

9.10 Inversione di una stringa

Data una stringa, la sua stringa inversa si ottiene leggendola al contrario. Ad esempio, la stringa inversa di *alfa* è *afla*.

La funzione *inverti* ha un parametro (*s*) e restituisce la stringa inversa di *s*.

```
function inverti(s) {
   var t = "";
   for (var i = 0; i < s.length; i++) {
      t = s[i] + t;
   }
   return t;
}
```

Un'altra soluzione utilizza un'iterazione determinata che inizia dall'indice dell'ultimo carattere di *s* e termina con zero.

```
function inverti(s) {
   var t = "";
   for (var i = s.length - 1; i >= 0; i--) {
      t += s[i];
   }
   return t;
}
```

9.11 Palindromo

Un *palindromo* è una sequenza di caratteri che, letta al contrario, rimane identica [Wikipedia, alla voce *Palindromo*]. Alcuni esempi sono il nome *Ada*, la voce verbale *aveva* e la frase *I topi non avevano nipoti*.

Il predicato *ePalindromo* ha un parametro (*s*) e verifica se la stringa *s* è un palindromo.

```
function ePalindromo(s) {
   return s == inverti(s);
}
```

Un'altra soluzione che non fa uso della funzione *inverti* è la seguente.

```
function ePalindromo(s) {
   var c = 0;
```

```
   for (var i = 0; i < s.length; i++) {
      if(s[i] == s[s.length - 1 - i]) {
         c++;
      }
   }
   return (c == s.length);
}
```

Per evitare di scandire tutta la stringa quando non è un palindromo è possibile utilizzare un'iterazione indeterminata.

```
function ePalindromo(s) {
   var i = 0;
   while ((i < s.length) &&
          (s[i] == s[s.length - 1 - i])) {
      i++;
   }
   return (i == s.length);
}
```

9.12 Ordinamento di array

Un array può essere ordinato in senso crescente o decrescente usando un *algoritmo di ordinamento*. Il più semplice di questi algoritmi si basa sulla ricerca successiva del minimo (ordinamento crescente) o del massimo (ordinamento decrescente).

```
function ordina(a) {
   for (var i = 0; i < a.length - 1; i++) {
      for (var j = i + 1; j < a.length; j++) {
         if (a[j] < a[i]) {
            var tmp = a[j];
            a[j] = a[i];
            a[i] = tmp;
         }
      }
   }
}
```

Un altro algoritmo si basa su una scansione che individua e scambia le coppie di elementi che non rispettano l'ordinamento. La scansione è ripetuta fino a quando tutti gli elementi dell'array sono ordinati.

```
function ordina(a) {
   var ordinato = false;
   while (!ordinato) {
      ordinato = true;
      for (var i = 1; i < a.length; i++) {
         if (a[i - 1] > a[i]) {
```

```
            var tmp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = tmp;
            ordinato = false;
         }
      }
   }
}
```

La stessa funzione può essere riscritta usando la forma alternativa del comando iterativo indeterminato.

```
function ordina(a) {
   do {
      var scambi = 0;
      for (var i = 1; i < a.length; i++) {
         if (a[i - 1] > a[i]) {
            var tmp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = tmp;
            scambi++;
         }
      }
   } while (scambi > 0);
}
```

9.13 Esercizi

1. La *media aritmetica semplice* di *n* numeri è così definita [Wikipedia, alla voce *Media (statistica)*]:

   $$m = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

   Ad esempio, la media aritmetica semplice di *3*, *12*, *24* è *13*.

   Definire in JavaScript una funzione che calcola e restituisce la media aritmetica semplice degli elementi di un array *a* formato da *n* numeri.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [3, 12, 24]

   [5, 7, 9, -12, 0].

2. Dato un array *a* e un valore *k*, il numero di occorrenze di *k* in *a* è definito come il numero degli elementi di *a* il cui valore è uguale a *k*.

   Definire in JavaScript una funzione che calcola e restituisce il numero di occorrenze del valore *k* nell'array *a*.

La funzione ha due parametri: *a*, *k*.

Invocare la funzione con i seguenti valori:

[10, −5, 34, 0], 1

[10, −5, 34, 0], −5.

3. Definire in JavaScript un predicato che verifica se ogni elemento di un array di numeri (tranne il primo) è pari alla somma degli elementi che lo precedono.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, 6, 10, 32]

   [1, 2, 6, 8, 31].

4. Definire in JavaScript una funzione che ha come parametro un array *a* di numeri e che restituisce un nuovo array che contiene le differenze tra gli elementi adiacenti di *a*.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, -6, 0, 3]

   [2, 2, 3, 3, 4, 4].

10 Soluzione degli esercizi della prima parte

10.1 Esercizi del capitolo 4

1. Calcolare la somma dei primi quattro multipli di 13.

```
print((13 * 1) + (13 * 2) + (13 * 3) + (13 * 4));
```

2. Verificare se la somma dei primi sette numeri primi è maggiore della somma delle prime tre potenze di due.

```
Print((2+3+5+7+11+13+17) > (2*1+2*2+2*2*2));
```

3. Verificare se 135 è dispari, 147 è pari, 12 è dispari, 200 è pari.

```
print(135 % 2 == 1);
print(147 % 2 == 0);
print(12  % 2 == 1);
print(200 % 2 == 0);
```

4. Calcolare l'area di un triangolo rettangolo i cui cateti sono 23 e 17.

```
print((23 * 17) / 2);
```

5. Calcolare la circonferenza di un cerchio il cui raggio è 14.

```
print(3.141592 * 2 * 14);
```

6. Calcolare l'area di un cerchio il cui diametro è 47.

```
print(3.141592 * (47 / 2) * (47 / 2));
```

7. Calcolare l'area di un trapezio la cui base maggiore è 48, quella minore è 25 e l'altezza è 13.

```
print(((48 + 25) / 2) * 13);
```

8. Verificare se l'area di un quadrato di lato quattro è minore dell'area di un cerchio di raggio tre.

```
print((4 * 4) < (3.141592 * 3 * 3));
```

9. Calcolare il numero dei minuti di una giornata, di una settimana, di un mese di 30 giorni, di un anno non bisestile.

```
print(60 * 24);
print(60 * 24 * 7);
print(60 * 24 * 30);
print(60 * 24 * 365);
```

10. Verificare se conviene acquistare una camicia che costa 63 € in un negozio che applica uno sconto fisso di 10 € o in un altro che applica uno sconto del 17%.

```
print((63 - 10) < (63 - 63 * (17 / 100)));
```

10.2 Esercizi del capitolo 5

1. Calcolare il costo di un viaggio in automobile, sapendo che la lunghezza è 750 Km, che il consumo di gasolio è 3,2 litri ogni 100 Km, che un litro di gasolio costa 1,432 €, che due terzi del percorso prevedono un pedaggio pari a 1,2 € ogni 10 Km.

```
var lunghezza = 750;
var kmGasolio = 3.2 / 100;
var ltGasolio = 1.432;
var carburante = lunghezza * kmGasolio * ltGasolio;
var lunghezzaAutostrada = lunghezza * (2 / 3);
var kmPedaggio = 1.2 / 10;
var pedaggio = lunghezzaAutostrada * kmPedaggio;
var costoTotale = carburante + pedaggio;
print(costoTotale);
```

2. Calcolare il costo di una telefonata, sapendo che la durata è pari a 4 minuti e 23 secondi, che il costo alla chiamata è pari a 0,15 €, che i primi 30 secondi sono gratis, che il resto della telefonata costa 0,24 € al minuto.

```
var minuti = 4;
var secondi = 23;
var costoFisso = 0.15;
var tariffaMinuto = 0.24;
var durata = minuti * 60 + secondi;
var tariffaSecondo = tariffaMinuto / 60;
var costoVariabile = (durata - 30) * tariffaSecondo;
var costo = costoFisso + costoVariabile;
print(costo);
```

3. Calcolare il costo di un biglietto aereo acquistato una settimana prima della partenza, sapendo che il costo di base è pari a 200 € (se acquistato il giorno della partenza) e che questo costo diminuisce del 2,3% al giorno (se acquistato prima del giorno della partenza).

```
var costoBase = 200;
var scontoGiornaliero = 2.3 / 100;
var giorni = 7;
var sconto = costoBase * scontoGiornaliero * giorni;
var costo = costoBase - sconto;
print(costo);
```

4. Calcolare il costo di un prodotto usando la seguente formula

   $costo = (prezzo + prezzo \cdot 0{,}20) - sconto$

   e sapendo che il prezzo è 100 € e lo sconto è 30 €.

```
var prezzo = 100;
var sconto = 30;
var costo = (prezzo + prezzo * 0.2) - sconto;
print(costo);
```

5. Calcolare la rata mensile di un mutuo annuale usando la seguente formula

   $$rata = \frac{importo}{12} \cdot (1 + tasso)$$

   e sapendo che l'importo annuale è 240 € e il tasso è il 5%.

```
var importo = 240;
var tasso = 5 / 100;
var rata = (importo / 12) * (1 + tasso);
print(rata);
```

10.3 Esercizi del capitolo 6

1. Definire in JavaScript un predicato che verifica se l'intersezione dell'intervallo [*a*, *b*) con l'intervallo [*c*, *d*) è vuota.

   Il predicato ha quattro parametri: *a*, *b*, *c*, *d*.

   Invocare il predicato con i seguenti valori: *2*, *4*, *5*, *7*; *2*, *4*, *4*, *7*; *2*, *4*, *3*, *7*; *5*, *7*, *2*, *4*; *4*, *7*, *2*, *4*; *3*, *7*, *2*, *4*.

```
function intersezione(a, b, c, d) {
   return ((b <= c) || (d <= a));
}
print(intersezione(2, 4, 5, 7));
print(intersezione(2, 4, 4, 7));
print(intersezione(2, 4, 3, 7));
print(intersezione(5, 7, 2, 4));
```

```
print(intersezione(4, 7, 2, 4));
print(intersezione(3, 7, 2, 4));
```

2. L'equazione di secondo grado $ax^2 + bx + c = 0$ ha due radici [Wikipedia, alla voce *Equazione di secondo grado*]:

$$x_{1/2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Definire in JavaScript una funzione che stampa le radici di un'equazione i cui coefficienti sono *a*, *b* e *c* (con *a* diverso da zero).

La funzione ha tre parametri:*a*, *b*, *c*.

Invocare la funzione con i seguenti valori: *1*, *-5*, *6*; *1*, *8*, *16*; *1*, *2*, *3*.

```
function radici(a, b, c) {
   var delta = b*b - 4*a*c;
   if (delta > 0) {
      print ((-b + Math.sqrt(delta)) / (2*a));
      print ((-b - Math.sqrt(delta)) / (2*a));
   } else if (delta == 0) {
      print (-b / (2*a));
   } else { // (delta < 0)
      print ("radici immaginarie");
   }
}
radici(1, -5, 6);
radici(1, 8, 16);
radici(1, 2, 3)
```

La funzione predefinita *Math.sqrt* calcola la radice quadrata del suo parametro.

3. Definire in JavaScript una funzione che calcola e restituisce la somma delle cifre di un intero che appartiene all'intervallo [*0, 100*).

La funzione ha un parametro: *n*.

Invocare la funzione con i seguenti valori: *0*, *1*, *23*, *99*.

```
function sommaCifre (n) {
   var u = n % 10;
   var d = (n - u) / 10;
   return u + d;
}
print(sommaCifre(0));
print(sommaCifre(1));
print(sommaCifre(23));
print(sommaCifre(99));
```

10.4 Esercizi del capitolo 7

1. I giorni di un anno possono essere numerati consecutivamente, assumendo che al primo gennaio sia assegnato il valore *1* e al trentuno dicembre il valore *365* negli anni non bisestili o *366* negli anni bisestili.

   Definire in JavaScript una funzione che restituisce il numero assegnato a un giorno dell'anno, assumendo che i mesi siano numerati da *1* a *12*.

   La funzione ha tre parametri: *anno*, *mese*, *giorno*.

   Invocare la funzione con i seguenti valori: *1957*, *4*, *25*; *2004*, *11*, *7*; *2000*, *12*, *31*; *2012*, *2*, *29*.

```
function numeroGiorno(anno, mese, giorno) {
   var n = giorno;
   switch (mese) {
      case 12: n += 30;
      case 11: n += 31;
      case 10: n += 30;
      case  9: n += 31;
      case  8: n += 31;
      case  7: n += 30;
      case  6: n += 31;
      case  5: n += 30;
      case  4: n += 31;
      case  3: if((anno % 400 == 0) ||
                  ((anno % 4 == 0) &&
                   (anno % 100 != 0))) {
                  n += 29;
               } else {
                  n += 28;
               }
      case  2: n += 31;
   }
   return n;
}
print(numeroGiorno(1957, 4, 25));
print(numeroGiorno(2004, 11, 7));
print(numeroGiorno(2000, 12, 31));
print(numeroGiorno(2012, 2, 29));
```

2. Definire in JavaScript una funzione che calcola e restituisce una stringa che rappresenta il valore in lettere di un intero che appartiene all'intervallo [*0*, *100*).

   La funzione ha un parametro: *n*.

   Invocare la funzione con i seguenti valori: *0*, *1*, *12*, *21*, *32*, *43*, *70*, *88*, *90*.

```
function lettere(n) {
 // da definire
}
print(lettere(0));
print(lettere(1));
print(lettere(12));
print(lettere(21));
print(lettere(32));
print(lettere(43));
print(lettere(70));
print(lettere(88));
print(lettere(90));
```

3. Definire in JavaScript una funzione che calcola il tipo di un triangolo (*equilatero*, *isoscele*, *rettangolo*, *scaleno*) in base alla lunghezza dei suoi lati.

   La funzione restituisce una stringa che rappresenta il tipo del triangolo e ha tre parametri : *a*, *b*, *c*.

   Invocare la funzione con i seguenti valori: *3, 3, 3*; *3, 4, 4*; *3, 4, 5*; *3, 4, 6*.

```
function triangolo (a, b, c) {
   if ((a == b) &&
       (b == c)) {
      return "equilatero";
   } else if (((a == b) && (a != c)) ||
              ((a == c) && (a != b)) ||
              ((b == c) && (a != b))) {
      return "isoscele";
   } else if ((Math.sqrt(a*a + b*b) == c) ||
              (Math.sqrt(a*a + c*c) == b) ||
              (Math.sqrt(b*b + c*c) == a)) {
     return "rettangolo";
  } else {
     return "scaleno";
  }
}
print(triangolo(3, 3, 3));
print(triangolo(3, 4, 4));
print(triangolo(3, 4, 5));
print(triangolo(3, 4, 6));
```

10.5 Esercizi del capitolo 8

1. Un anno è perfetto per una persona se è un multiplo della sua età in quel-l'anno, assumendo come età massima cento anni. Ad esempio gli anni

perfetti per chi è nato nel *1984* sono nove: *1985*, *1986*, *1988*, *1992*, *2000*, *2015*, *2016*, *2046*, *2048*.

Definire in JavaScript una funzione che stampa gli anni perfetti relativi a un anno di nascita.

La funzione ha un parametro: *annoDiNascita*.

Invocare la funzione con i seguenti valori: *1984*, *1990*, *1992*, *2000*, *2009*.

```
function anniPerfetti(annoDiNascita) {
   var ETA_MASSIMA = 100;
   for (var i = 1; i <= ETA_MASSIMA; i++) {
      if (((annoDiNascita + i) % i) == 0) {
         print(annoDiNascita + i);
      }
   }
}
anniPerfetti(1984);
anniPerfetti(1990);
anniPerfetti(1992);
anniPerfetti(2000);
anniPerfetti(2009);
```

2. In matematica, con reciproco di un numero x si indica il numero che moltiplicato per x dà come risultato 1.

   Definire in JavaScript una funzione che calcola e restituisce la somma dei reciproci dei primi n numeri interi maggiori di zero.

   La funzione ha un parametro: n.

   Invocare la funzione con i seguenti valori: *1*, *2*, *4*, *7*.

```
function armonica(n) {
   var s = 0;
   for (var i = 1; i <= n; i++) {
      s += 1 / i;
   }
   return s;
}
print(armonica(1));
print(armonica(2));
print(armonica(4));
print(armonica(7));
```

3. Definire in JavaScript una funzione che stampa, se esistono, le radici intere di un'equazione di secondo grado di coefficienti a, b, c comprese nell'intervallo $[l, u)$.

   La funzione ha i seguenti parametri: a, b, c, l, u.

Invocare la funzione con i seguenti valori:

1, –2, –8, 1, 5

1, –2, –8, –5, 5

1, –2, –8, 5, 10.

```
function radici(a, b, c, l, u) {
   for (var i = l; i < u; i++) {
      if (a * i * i + b * i + c == 0) {
         print(i);
      }
   }
}
radici(1, -2, -8, 1, 5);
radici(1, -2, -8, -5, 5);
radici(1, -2, -8, 5, 10);
```

10.6 Esercizi del capitolo 9

1. La *media aritmetica semplice* di *n* numeri è così definita [Wikipedia, alla voce *Media (statistica)*]:

$$m = \frac{x_1 + x_2 + \ldots + x_n}{n}$$

Ad esempio, la media aritmetica semplice di *3*, *12*, *24* è *13*.

Definire in JavaScript una funzione che calcola e restituisce la media aritmetica semplice degli elementi di un array *a* formato da *n* numeri.

La funzione ha un parametro: *a*.

Invocare la funzione con i seguenti valori:

[3, 12, 24]

[5, 7, 9, -12, 0].

```
function media(a) {
   var s = 0;
   for (var i = 0; i < a.length; i++) {
      s += a[i];
   }
   return s / a.length;
}
print(media([3, 12, 24]));
print(media([5, 7, 9, -12, 0]));
```

2. Dato un array *a* e un valore *k*, il numero di occorrenze di *k* in *a* è definito come il numero degli elementi di *a* il cui valore è uguale a *k*.

Definire in JavaScript una funzione che calcola e restituisce il numero di occorrenze del valore *k* nell'array *a*.

La funzione ha due parametri: *a*, *k*.

Invocare la funzione con i seguenti valori:

[10, −5, 34, 0], 1

[10, −5, 34, 0], −5.

```
function occorrenze(a, k) {
   var c = 0;
   for (var i = 0; i < a.length; i++) {
   if (a[i] == k) {
      c++;
   }
}
return c;
}
print(occorrenze([10, -5, 34, 0], 1));
print(occorrenze([10, -5, 34, 0], -5));
```

3. Definire in JavaScript un predicato che verifica se ogni elemento di un array di numeri (tranne il primo) è pari alla somma degli elementi che lo precedono.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

   [1, 2, 6, 10, 32]

   [1, 2, 6, 8, 31].

```
function verificaSomma (a) {
   var c = a[0];
   var i = 1;
   while ((i < a.length) && (a[i] == c)) {
      c += a[i];
      i++;
   }
   return (i == a.length);
}
print(verificaSomma([1, 2, 6, 10, 32]));
print(verificaSomma([1, 1, 2, 4, 8]));
```

4. Definire in JavaScript una funzione che ha come parametro un array *a* di numeri e che restituisce un nuovo array che contiene le differenze tra gli elementi adiacenti di *a*.

   La funzione ha un parametro: *a*.

   Invocare la funzione con i seguenti valori:

[1, 2, -6, 0, 3]
[2, 2, 3, 3, 4, 4].

```
function differenze (a) {
   var b = [];
   for (var i = 0; i < a.length - 1; i++) {
      b.push(a[i] - a[i + 1]);
   }
   return b;
}
print(differenze([1, 2, -6, 0, 3]));
print(differenze([2, 2, 3, 3, 4, 4]));
```

Parte seconda

Programmazione web

11 Ricorsione

In JavaScript è possibile definire *funzioni ricorsive*. Una funzione è ricorsiva quando nel suo corpo compare un'invocazione alla funzione stessa. Il concetto di ricorsione è stato già introdotto nella definizione delle grammatiche: una regola è ricorsiva quando la definizione di un simbolo non-terminale contiene il simbolo non-terminale stesso.

Per definire una funzione ricorsiva non è necessario modificare le regole sintattiche viste finora. Il capitolo presenta alcuni problemi la cui soluzione può essere definita naturalmente mediante una funzione ricorsiva.

11.1 Fattoriale

Il *fattoriale* di *n* (indicato con *n*!) è il prodotto dei primi *n* numeri interi positivi minori o uguali di quel numero [Wikipedia, alla voce *Fattoriale*]. 0! vale 1, 1! vale 1, 2! vale 2, 3! vale 6, 4! vale 24 e così via.

La funzione *fattorialeR* ha un parametro (*n*) e restituisce il fattoriale di *n*. La funzione è definita in maniera ricorsiva, sfruttando la definizione matematica del fattoriale.

```
function fattorialeR (n) {
   if ((n == 0) || (n == 1)) {
      return 1;
   } else {
      return n * fattorialeR(n - 1);
   }
}
```

Il calcolo del fattoriale può essere effettuato anche con una funzione iterativa, che non fa uso della ricorsione. Anche in questo caso la funzione ha un parametro (*n*) e restituisce il fattoriale di *n*.

```
function fattorialeI (n) {
   var r = 1;
   for (var i = 1; i <= n; i++) {
      r *= i;
   }
   return r;
}
```

11.2 Successione di Fibonacci

La *successione di Fibonacci* [Wikipedia, alla voce *Successione di Fibonacci*] è una sequenza di numeri interi naturali così definita: i primi due termini valgono zero e uno, rispettivamente; i termini successivi sono definiti come la somma dei due precedenti. Storicamente il primo termine della successione non era zero ma uno.

La sequenza prende il nome dal matematico pisano del XIII secolo *Leonardo Fibonacci* e i termini di questa successione sono chiamati *numeri di Fibonacci*. Definendo questa successione, Fibonacci voleva stabilire una legge che descrivesse la crescita di una popolazione di conigli. Fibonacci fece alcune assunzioni: la prima coppia diventa fertile al compimento del primo mese e dà alla luce una nuova coppia al compimento del secondo mese; le coppie nate nei mesi successivi si comportano in modo analogo; le coppie fertili, dal secondo mese di vita, danno alla luce una coppia di conigli al mese.

Per verificare la validità della successione si parte con una singola coppia di conigli, che dopo un mese sarà fertile. Dopo due mesi ci saranno due coppie di conigli, di cui una sola fertile. Nel mese seguente le coppie saranno tre (*2 + 1 = 3*), perché solo la coppia fertile avrà partorito. Di queste tre, ora saranno due le coppie fertili e, quindi, nel mese seguente ci saranno (*3 + 2 = 5*) cinque coppie. Come si può vedere, il numero di coppie di conigli di ogni mese coincide con i valori della successione di Fibonacci.

In base a queste considerazioni si può definire una funzione ricorsiva che ha come parametro un intero *n*, maggiore o uguale a zero, e che restituisce il corrispondente numero della successione di Fibonacci (assumendo che il primo numero della successione abbia indice zero).

```
function fibonacciR (n) {
   if ((n == 0) || (n == 1)) {
      return n;
   } else {
      return fibonacciR(n - 2) +
             fibonacciR(n - 1);
   }
}
```

Il calcolo di un elemento della successione di Fibonacci può essere effettuato definendo un'altra funzione, questa volta iterativa, più complessa e meno intuitiva ma molto più efficiente in termini di numero di operazioni elementari (ad esempio le somme) necessarie per effettuare il calcolo.

```
function fibonacciI (n) {
   if ((n == 0) || (n == 1)) {
      return n;
```

```
    }
    var f1 = 0;
    var f2 = 1;
    var f;
    for (var i = 2; i <= n; i++) {
        f = f1 + f2;
        f1 = f2;
        f2 = f;
    }
    return f;
}
```

11.3 Aritmetica di Peano

Gli *assiomi di Peano* sono stati ideati dal matematico *Giuseppe Peano* per definire l'insieme dei numeri naturali. Informalmente tali assiomi possono essere così enunciati [Wikipedia, alla voce *Assiomi di Peano*]:

- esiste un numero naturale, lo *zero*
- ogni numero naturale ha un numero naturale *successore*
- numeri diversi hanno successori diversi
- *zero* non è il successore di alcun numero naturale
- ogni insieme di numeri naturali che contiene lo *zero* e il successore di ogni suo elemento coincide con tutto l'insieme dei numeri naturali (induzione).

L'*aritmetica di Peano* è una *teoria del prim'ordine* basata su una versione degli a*ssiomi di Peano* espressi nel linguaggio del prim'ordine [Wikipedia, alla voce *Aritmetica di Peano*]. In questa aritmetica le operazioni sono definite per *induzione* usando la costante *zero*, l'operatore unario *successore*[8], le relazioni di *uguaglianza* e *disuguaglianza* con *zero*.

Le operazioni di *addizione* (indicata con $\pm$) e di *moltiplicazione* (indicata con $\times$) sono così definite per induzione:

$x \pm 0 = x$

$x \pm y = x \pm y - 1 + 1 = (x \pm (y - 1)) + 1 \; se \; y \neq 0$

$x \times 0 = 0$

$x \times y = x \times (y - 1 + 1) = (x \times (y - 1)) \pm x \; se \; y \neq 0$

In JavaScript è possibile definire ricorsivamente le funzioni che realizzano queste operazioni usando esclusivamente la costante *0*, l'operatore binario di ugua-

8 Per semplicità di presentazione, l'operatore unario successore è stato sostituito dagli operatori unari di incremento (*+1*) e decremento (*–1*) unitario.

glianza (==), l'operatore binario di disuguaglianza (!=), l'operatore di incremento unitario (*+1*) e l'operatore di decremento unitario (*–1*).

```
function addizione(x, y) {
   if (y == 0) {
      return x;
   } else {
      return addizione(x, y -1) +1;
   }
}
function moltiplicazione(x, y) {
   if (y == 0) {
      return 0;
   } else {
      return addizione(moltiplicazione(x, y -1), x);
   }
}
```

L'operazione di *sottrazione*[9] (indicata con *S*) è così definita per induzione:

$x\ S\ 0 = x$

$x\ S\ y = x\ S\ y\ +1\ -1 = (x\ S\ (y\ -1))\ -1\ se\ y \neq 0$

L'operazione di *potenza* è così definita per induzione:

$x^0 = 1\ se\ x \neq 0$

$x^y = x \times x^{y\ -1}\ se\ x \neq 0\ e\ y \neq 0$

$0^y = 0\ se\ y \neq 0$

Da notare che 0^0 non è definito.

Per le operazioni *divisione* e *resto* è necessaria la definizione induttiva dell'operatore di relazione <:

$x < 0 = false$

$0 < x = true\ se\ y \neq 0$

$x < y = x\ -1 < y\ -1\ se\ x \neq 0\ e\ y \neq 0$

Le operazioni *divisione* (indicata con *D*) e *resto* (indicata con *R*) sono così definite per induzione:

$x\ D\ y = 0\ se\ x < y$

$x\ D\ y = 1\ se\ x = y$

$x\ D\ y = (x\ S\ y \pm y)\ D\ y = (x\ S\ y)\ D\ y \pm y\ D\ y = (x\ S\ y)\ D\ y\ +1\ se\ y < x$

9 L'operazione è definita se e solo se $x \geq y$.

Da notare che $x\ D\ 0$ non è definito.

$x\ R\ y = x$ *se* $x < y$

$x\ R\ y = 0$ *se* $x = y$

$x\ R\ y = (x\ S\ y)\ R\ y$ *se* $y < x$

Anche in questo caso $x\ R\ 0$ non è definito.

11.4 Esercizi

1. Definire in Javascript una funzione ricorsiva che calcola e restituisce la differenza tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   5, 0

   3, 3

   9, 2.
2. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la relazione di minore tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 0

   0, 4

   3, 0

   2, 6

   9, 2.
3. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

   3, 3

   9, 2.
4. Definire in JavaScript una funzione ricorsiva che calcola e restituisce il resto della divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

3, 3
9, 2.

5. Definire in JavaScript una funzione ricorsiva che calcola e restituisce l'operatore potenza definito tra due interi maggiori o uguali a zero.
   La funzione ha due parametri: x, y.
   Invocare la funzione con le seguenti coppie di valori:
   6, 0
   3, 3
   0, 2.

12 Oggetti

In JavaScript, oltre agli oggetti predefiniti presentati nella prima parte (stringhe e array) è possibile usare *oggetti personalizzati*, con proprietà e metodi specifici.

Per creare un oggetto personalizzato è necessario dichiarare una *funzione costruttrice* (o *costruttore*) che ne definisce le *proprietà* e i *metodi*.

```
<fun_let>       ::= function ()
                    <blocco>
                  | function (<parametri>)
                    <blocco>

<prop_met>      ::= this.<identificatore> = <espressione>;
                  | this.<identificatore> = <fun_let>

<blocco_costr>  ::= <prop_met>
                  | <prop_met> <blocco_costr>

<dic_costr>     ::= function <identificatore>()
                    {<blocco_costr>}
                  | function <identificatore>(<parametri>)
                    {<blocco_costr>}
```

La dichiarazione di costruttore è una dichiarazione di funzione. Anche il costruttore ha un'intestazione con una lista di parametri formali, eventualmente vuota, e un blocco che costituisce la definizione vera e propria del costruttore. Nel corpo della dichiarazione di costruttore sono presenti dichiarazioni di proprietà e di metodi. La parola riservata *this* fa riferimento all'oggetto in fase di costruzione.

La dichiarazione di una proprietà è simile alla dichiarazione di una variabile, con l'unica differenza che il nome della proprietà deve essere preceduto dalla parola riservata *this* e da un punto. La dichiarazione di un metodo è simile alla dichiarazione di una funzione. Anche in questo caso il nome della funzione deve essere preceduto dalla parola riservata *this* e da un punto. Ciò che cambia è l'intestazione in cui manca il nome della funzione, precedentemente indicato nella dichiarazione.

Per convenzione, i nomi dei costruttori iniziano con una lettera maiuscola, al contrario delle variabili e delle funzioni il cui nome inizia sempre con una lettera minuscola.

```
function Automobile (mar, mod, col, cil){
   this.marca = mar;
   this.modello = mod;
   this.colore = col;
   this.cilindrata = cil;
}
```

Il costruttore *Automobile* definisce un oggetto che ha quattro proprietà: *marca*, *modello*, *colore*, *cilindrata*. Il costruttore non ha metodi associati.

Il valore iniziale delle proprietà di un oggetto è stabilito al momento della sua creazione, effettuata mediante il comando *new* in cui si indicano il nome del costruttore e una lista di parametri.

```
<espressione> ::= new <identificatore>()
                | new <identificatore>(<espressioni>)
```

Dopo aver definito il costruttore *Automobile* è possibile utilizzarlo per creare alcuni oggetti.

```
var a1 = new Automobile('Ford',   'Focus', 'Grigio', 1.8);
var a2 = new Automobile('Fiat',   'Panda', 'Rosso',  1.1);
var a3 = new Automobile('Volvo',  'V70',   'Nero',   2.4);
var a4 = new Automobile('Lancia', 'Libra', 'Blu',    1.9);
```

Dopo aver creato un oggetto, è possibile accedere alle sue proprietà per utilizzarne o modificarne il valore. Anche per le proprietà degli oggetti si utilizza la notazione a punti vista nella prima parte.

```
var descrizione = a1.marca + ' ' + a1.modello;
a2.modello = 'Punto';
a2.colore = 'Nero';
a2.cilindrata = 1.3;
```

Il primo comando dichiara la variabile *descrizione* a cui è assegnato come valore iniziale una stringa ottenuta concatenando i valori delle proprietà *marca* e *modello* dell'oggetto *a1*. I tre comandi successivi trasformano l'oggetto *a2*, facendolo diventare una *Punto nera* di *1.3* di cilindrata. Il valore della proprietà *marca* resta invariato.

12.1 Metodi

Nella dichiarazione di un costruttore è possibile dichiarare, oltre alle proprietà, anche i metodi, funzioni che possono utilizzare e modificare il valore delle proprietà dell'oggetto stesso.

```
function Automobile (mar, mod, col, cil){
   this.marca = mar;
   this.modello = mod;
   this.colore = col;
   this.cilindrata = cil;
   this.serbatoio = 0;
   this.rifornisci =
      function (c){
         this.serbatoio += c;
      }
   this.descrivi =
      function () {
         return this.marca + ' ' +
                this.modello + ' ' +
                this.colore + ' ' +
                this.cilindrata;
      }
}
```

Al costruttore *Automobile* è stata aggiunta una nuova proprietà, *serbatoio*, inizializzata a zero al momento della creazione dell'oggetto.

Oltre alla nuova proprietà, il costruttore definisce due metodi: *rifornisci* e *descrivi*. Il primo incrementa il valore della proprietà *serbatoio*, simulando così il rifornimento effettuato presso una stazione di servizio. Il secondo restituisce una stringa che descrive l'oggetto.

```
a1.rifornisci(50);
var descrizione = a1.descrivi();
```

L'invocazione del metodo *rifornisci* provoca la modifica della proprietà *serbatoio* che passa dal valore zero, assegnato al momento della creazione dell'oggetto, al valore *50*. L'invocazione del metodo *descrivi* restituisce la stringa *Ford Focus Grigio 1.8*.

12.2 Il convertitore di valuta

Un *convertitore di valuta* permette di calcolare il valore di un importo di denaro espresso in una valuta (ad esempio *Euro*) nel corrispondente importo espresso in un'altra valuta (ad esempio *Lire*). Nel seguito mostriamo il codice Java-

Script di un costruttore personalizzato per oggetti che rappresentano *convertitori di valuta*.

L'oggetto ha i seguenti metodi:

- *imposta (valuta, fattore)*: imposta la sigla della *valuta* e il *fattore* di conversione da *Euro*;
- *converti (importo)*: converte l'*importo*, utilizzando il fattore di conversione, e restituisce il valore calcolato preceduto dalla sigla della valuta.

```
function Convertitore () {
   this.valutaCorrente;
   this.fattoreCorrente;
   this.imposta =
      function (valuta, fattore) {
         this.valutaCorrente = valuta;
         this.fattoreCorrente = fattore;
      }
   this.converti =
      function (importo) {
         return this.valutaCorrente + " " +
                importo * this.fattoreCorrente;
      }
}
```

Per verificare il corretto funzionamento dell'oggetto, definiamo una *funzione di prova*[10] che:

- crea un convertitore di valuta,
- imposta la valuta in lire con un fattore di conversione pari a 1936.27,
- converte 100.00 €.

```
function foo () {
   var c = new Convertitore();
   c.imposta("Lira", 1936.27);
   print(c.converti(100.00));
}
```

Dopo aver definito la funzione di prova è necessario invocarla.

```
foo();
```

10 Per convenzione, il nome di una funzione di prova è *foo*.

12.3 Insiemi

Un *insieme* [Wikipedia, alla voce *Insieme*] è una collezione di oggetti chiamati *elementi dell'insieme*. Il concetto di insieme è intuitivo, caratterizzato dalle seguenti proprietà:

- un elemento può appartenere o non appartenere a un insieme;
- un elemento può comparire solo una volta in un insieme;
- due insiemi coincidono se e solo se hanno gli stessi elementi.

La creazione di un oggetto che rappresenta un *insieme vuoto* avviene mediante l'invocazione del costruttore *Insieme*. Gli elementi di un insieme sono memorizzati in una proprietà dell'oggetto, un array associativo chiamato *elementi*, inizializzata con un array vuoto al momento della sua creazione.

```
function Insieme (){
   this.elementi = {};
}
```

L'oggetto *Insieme* ha i seguenti metodi: *cardinalità*, *appartiene*, *aggiungi*, *intersezione*, *unione*, *visualizza*.

Il metodo *cardinalità* restituisce il numero di elementi dell'insieme. Il ciclo che conta gli elementi è necessario perché la proprietà *length* non è definita per gli array associativi.

```
this.cardinalità =
   function () {
      var c = 0;
      for (var i in this.elementi) {
         c++;
      }
      return c;
   }
```

Il metodo *appartiene* è un predicato che verifica se un elemento appartiene a un insieme.

```
this.appartiene =
   function (x) {
      return (x in this.elementi);
   }
```

Il metodo *aggiungi* estende l'insieme con un elemento. Il metodo sfrutta le caratteristiche degli array associativi.

```
this.aggiungi =
   function (k) {
      this.elementi[k] = k;
   }
```

Il metodo *intersezione* ha un parametro (un altro insieme) e restituisce un insieme formato dagli elementi in comune ai due insiemi.

```
this.intersezione =
   function (x) {
      var n = new Insieme();
      for (var i in this.elementi) {
         if (x.appartiene(i)) {
             n.aggiungi(i);
         }
      }
      return n;
   }
```

Il metodo *unione* ha un parametro (un altro insieme) e restituisce un insieme formato dagli elementi dei due insiemi. Un elemento comune ai due insiemi compare solo una volta nel nuovo insieme.

```
this.unione =
   function (x) {
      var n = new Insieme();
      for (var i in this.elementi){
         n.aggiungi(i);
      }
      for (var j in x.elementi){
         n.aggiungi(j);
      }
      return n;
   }
```

Il metodo *visualizza* restituisce una stringa formata dagli elementi dell'insieme separati da virgole. Conformemente alla notazione standard degli insiemi, la stringa è delimitata da parentesi graffe.

```
this.visualizza =
   function () {
      var s = "{";
      var b = true;
      for (var i in this.elementi) {
         if (b) {
            s += i;
            b = false;
         } else {
```

```
            s += ", " + i;
         }
      }
      s += "}";
      return s;
   }
```

La seguente funzione di prova

- crea l'insieme *alfa*
- aggiunge due elementi ad *alfa*
- calcola la cardinalità di *alfa*
- visualizza *alfa*
- verifica l'appartenenza di un numero ad *alfa*
- crea l'insieme *beta*
- aggiunge due elementi a *beta*
- calcola l'intersezione tra *alfa* e *beta*
- calcola l'unione tra *alfa* e *beta*.

```
function foo () {
   var alfa = new Insieme();
   alfa.aggiungi(10);
   alfa.aggiungi(20);
   print(alfa.cardinalità());
   print(alfa.visualizza());
   var x = 10;
   if (alfa.appartiene(x)) {
      print (x + " appartiene ad alfa");
   } else {
      print (x + " non appartiene ad alfa");
   }
   var beta = new Insieme();
   beta.aggiungi(10);
   beta.aggiungi(30);
   print(alfa.intersezione(beta).visualizza());
   print(alfa.unione(beta).visualizza());
}
```

12.4 Esercizi

1. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *sveglie*.

   L'oggetto ha i seguenti metodi:

- *oraCorrente* (*h*, *m*): imposta l'ora corrente. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
- *allarme* (*h*, *m*): imposta l'allarme. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
- *tic* (): fa avanzare di uno i minuti dell'ora corrente. Se i minuti arrivano a 60, azzera i minuti e fa avanzare di uno le ore. Se le ore arrivano a 24, azzera le ore. Se l'ora corrente è uguale all'allarme impostato, restituisce il valore *true*, *false* altrimenti.

Definire una funzione di prova che

- crea una sveglia,
- imposta l'ora corrente alle 13:00,
- imposta l'allarme alle 13:02,
- fa avanzare l'ora corrente di due minuti.

2. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *distributori automatici di caffè*.

   L'oggetto ha i seguenti metodi:

   - *carica* (*n*): aggiunge *n* capsule per erogare *n* caffè;
   - *eroga* (*n*, *c*): verifica se è possibile erogare *n* caffè e, in caso positivo, li eroga addebitandoli al codice *c*;
   - *rapporto* (*c*): restituisce il numero di caffè addebitati al codice *c* e il numero di capsule disponibili.

   Definire una funzione di prova che:

   - crea un distributore automatico di caffè,
   - carica 20 capsule,
   - eroga 12 caffè per il codice *Carlo*,
   - genera il rapporto per il codice *Carlo*.

13 Alberi

Nella teoria dei grafi un *albero* è un *grafo non orientato* nel quale due *vertici* qualsiasi sono connessi da uno e un solo *cammino* (grafo non orientato connesso e privo di cicli) [Wikipedia, alla voce *Albero (grafo)*]. In questo capitolo presentiamo alcuni oggetti per la rappresentazione e la manipolazione di alberi.

13.1 Alberi binari

Un *albero binario* è un albero i cui nodi possono avere al massimo due *sottoalberi* o *figli*. Un nodo senza figli è anche chiamato *foglia*. Un nodo che non ha un *nodo padre* è chiamato *radice*.

In JavaScript è possibile realizzare gli alberi binari mediante l'oggetto *AlberoBinario*. Il suo costruttore ha tre parametri: un valore *v*, un albero sinistro *sx*, un albero destro *dx*. Per memorizzare questi valori si utilizzano le proprietà *valore*, *sinistro* e *destro*.

```
this.valore   = v;
this.sinistro = sx;
this.destro   = dx;
```

La seguente funzione crea un albero binario formato da una radice e da due foglie. La creazione dell'albero inizia dalle foglie e procede verso la radice. Il valore *null* è utilizzato per indicare l'assenza di un figlio in un albero.

```
function foo () {
   var foglia1 = new AlberoBinario(2, null, null);
   var foglia2 = new AlberoBinario(3, null, null);
   var radice = new AlberoBinario(1, foglia1, foglia2);
}
```

Un albero può essere visualizzato mostrando i suoi valori in sequenza. La costruzione della sequenza dei valori si effettua mediante una *visita* dei nodi che, nel caso degli alberi binari, può essere organizzata in tre modi diversi: *visita anticipata*, *visita differita*, *visita simmetrica*.

La visita anticipata costruisce una sequenza in cui prima compare il valore della radice seguito dalla visita anticipata del figlio sinistro e dalla visita anticipata del figlio destro. La visita anticipata dell'albero binario creato nell'esempio precedente è *1 2 3*.

Il metodo *visitaAnticipata* costruisce una stringa che rappresenta la visita anticipata di un albero binario.

```
this.visitaAnticipata =
   function () {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visitaAnticipata();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaAnticipata();
      } else {
         vd = "";
      }
      return this.valore + " " + vs + vd;
}
```

La visita differita costruisce una sequenza in cui prima compare la visita anticipata del figlio sinistro seguita dalla visita anticipata del figlio destro e, alla fine, dal valore della radice. La visita differita dell'albero binario creato nell'esempio precedente è *2 3 1*.

Il metodo *visitaDifferita* costruisce una stringa che rappresenta la visita differita di un albero binario.

```
this.visitaDifferita =
   function () {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visitaDifferita();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaDifferita();
      } else {
         vd = "";
      }
      return vs + vd + this.valore + " ";
   }
```

La visita simmetrica costruisce una sequenza in cui prima compare la visita anticipata del figlio sinistro, poi il valore della radice e, alla fine, la visita anticipata

del figlio destro. La visita simmetrica dell'albero binario creato nell'esempio precedente è *2 1 3*.

Il metodo *visitaSimmetrica* costruisce una stringa che rappresenta la visita simmetrica di un albero binario.

```
this.visitaSimmetrica =
   function () {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visitaSimmetrica();
      } else {
         vs = "";
      }
      var vd;
      if (this.destro != null) {
         vd = this.destro.visitaSimmetrica();
      } else {
         vd = "";
      }
      return vs + this.valore + " " + vd;
   }
```

L'*altezza* di un albero è la lunghezza del massimo cammino dalla radice a una foglia. In particolare, l'altezza di un albero formato da un solo nodo radice è uguale a zero. L'altezza dell'albero binario dell'esempio precedente è *1*.

Il metodo *altezza* restituisce l'altezza di un albero binario.

```
this.altezza =
   function () {
      if (this.sinistro == null &&
          this.destro == null) {
         return 0;
      }
      var as;
      if (this.sinistro != null) {
         as = this.sinistro.altezza();
      } else {
         as = 0;
      }
      var ad;
      if (this.destro != null) {
         ad = this.destro.altezza();
      } else {
         ad = 0;
      }
      if (as > ad) {
         return as + 1;
      } else {
         return ad + 1;
      }
   }
```

La *frontiera* di un albero è la sequenza dei valori delle foglie visitate da sinistra verso destra. La frontiera dell'albero binario dell'esempio precedente è *2 3*.

Il metodo *frontiera* restituisce la frontiera di un albero binario.

```
this.frontiera =
   function () {
      if (this.sinistro == null &&
          this.destro == null) {
         return this.valore + " ";
      }
      var fs;
      if (this.sinistro != null) {
         fs = this.sinistro.frontiera();
      } else {
         fs = "";
      }
      var fd;
      if (this.destro != null) {
         fd = this.destro.frontiera();
      } else {
         fd = "";
      }
      return fs + fd;
   }
```

La seguente funzione di prova

- crea un albero binario *alfa*
- effettua la visita anticipata di *alfa*
- effettua la visita differita di *alfa*
- effettua la visita simmetrica di *alfa*
- calcola l'altezza di *alfa*
- costruisce la frontiera di *alfa*.

```
function foo () {
   var alfa = new AlberoBinario(1,
                  new AlberoBinario(2,
                     new AlberoBinario(4, null, null),
                     new AlberoBinario(5, null, null)),
                  new AlberoBinario(3, null, null));
   print(alfa.visitaAnticipata());
   print(alfa.visitaDifferita());
   print(alfa.visitaSimmetrica());
   print(alfa.altezza());
   print(alfa.frontiera());
}
```

13.2 Alberi di ricerca

Un *albero di ricerca* è un albero binario la cui visita simmetrica genera una sequenza ordinata di valori. Ci possono essere due tipi di alberi di ricerca: con ripetizioni o senza ripetizioni. Nel primo caso la relazione di ordinamento tra i valori prevede anche l'uguaglianza (ad esempio non decrescente), nel secondo caso l'uguaglianza è esclusa (ad esempio crescente).

La ricerca di un valore in un albero di ricerca è molto efficiente: partendo dalla radice è possibile escludere dalla ricerca metà dell'albero, perché il valore cercato non può appartenere a quella metà. Anche l'inserzione di un valore in un albero di ricerca è un'operazione efficiente, perché si basa su una ricerca e sull'eventuale aggiunta di una nuova foglia.

Il costruttore dell'oggetto *AlberoDiRicerca* (ordinamento crescente senza ripetizioni) è simile a quello dell'oggetto *AlberoBinario* in quanto ha un parametro e tre proprietà: valore, figlio sinistro, figlio destro.

Il metodo *aggiungi* ha un parametro *k* e aggiunge un nodo all'albero di ricerca, mantenendo l'ordinamento crescente. Il metodo è definito ricorsivamente.

```
this.aggiungi =
   function (k) {
      if (k < this.valore) {
         if (this.sinistro == null) {
            this.sinistro = new AlberoDiRicerca(k);
         } else {
            this.sinistro.aggiungi(k);
         }
      }
      if (k > this.valore) {
         if (this.destro == null) {
            this.destro = new AlberoDiRicerca (k);
         } else {
            this.destro.aggiungi(k);
         }
      }
   }
```

Il metodo *cerca* è un predicato con un parametro *k* che verifica se esiste un nodo il cui valore è uguale a *k*. Anche questo metodo è definito ricorsivamente.

```
this.cerca =
   function (k) {
      if (k == this.valore) {
         return true;
      }
      if (k < this.valore) {
```

```
         if (this.sinistro == null) {
            return false;
         } else {
            return this.sinistro.cerca(k);
         }
      }
      if (k > this.valore) {
         if (this.destro == null) {
            return false;
         } else {
            return this.destro.cerca(k);
         }
      }
   }
```

Infine, il metodo *visita* restituisce una stringa ottenuta visitando l'albero con una visita simmetrica. Questo metodo è uguale al metodo *visitaSimmetrica* definito per gli alberi binari.

La seguente funzione di prova

- crea l'albero di ricerca *alfa* con il valore *1*
- aggiunge *5*, *3* e *8* ad *alfa*
- effettua una visita simmetrica di *alfa*
- cerca *1* e *4* in *alfa*.

```
function foo () {
   var a = new AlberoDiRicerca(1);
   a.aggiungi(5);
   a.aggiungi(3);
   a.aggiungi(8);
   print(a.visita());
   print(a.cerca(1));
   print(a.cerca(4));
}
```

13.3 Alberi n-ari

Un *albero n-ario* è un albero i cui nodi, a differenza di quelli degli alberi binari, possono avere un numero qualsiasi di figli. Un'altra differenza con gli alberi binari consiste nel fatto che sono definite solo due visite: anticipata e differita. La visita simmetrica, infatti, non avrebbe senso.

Come per gli alberi binari, i nodi di un albero n-ario hanno un valore associato. Il costruttore *AlberoNArio* ha un solo parametro *v,* memorizzato nella proprietà *valore*. I figli di un albero n-ario sono memorizzati nella proprietà *figli*, inizializzata con un array vuoto.

```
this.valore = v;
this.figli = [];
```

Il metodo *aggiungiFiglio* ha un parametro *n*, un nodo da aggiungere in coda ai figli di un albero n-ario.

```
this.aggiungiFiglio =
   function (n) {
      this.figli.push(n);
   }
```

La visita anticipata di un albero n-ario è calcolata dal metodo *visitaAnticipata* che si comporta analogamente allo stesso metodo definito per gli alberi binari. La scansione dei figli è effettuata mediante un'iterazione determinata.

```
this.visitaAnticipata =
   function () {
      var visita = "";
      for (var i = 0; i < this.figli.length; i++) {
         visita += " " + this.figli[i].visitaAnticipata();
      }
      return this.valore + visita;
   }
```

La visita differita, calcolata dal metodo *visitaDifferita*, è simile all'analoga visita definita per gli alberi binari.

```
this.visitaDifferita =
   function () {
      var visita = "";
      for (var i = 0; i < this.figli.length; i++) {
         visita += this.figli[i].visitaDifferita() + " ";
      }
      return visita + this.valore;
   }
```

L'altezza di un albero n-ario è calcolata dal metodo *altezza*, che usa l'algoritmo per il calcolo del massimo, già definito per gli array.

```
this.altezza =
   function () {
      if (this.figli.length == 0) {
         return 0;
      }
      var maxAltezza = this.figli[0].altezza();
      for (var i = 1; i < this.figli.length; i++) {
         var altezzaFiglio = this.figli[i].altezza();
         if (altezzaFiglio > maxAltezza) {
```

```
            maxAltezza = altezzaFiglio;
         }
      }
      return maxAltezza + 1;
   }
```

L'ultimo metodo, *frontiera*, calcola la frontiera di un albero n-ario. Anche questo metodo è simile a quello definito per gli alberi binari.

```
this.frontiera =
   function () {
      if (this.figli.length == 0) {
         return this.valore + " ";
      }
      var f = "";
      for (var i = 0; i < this.figli.length; i++) {
         f += this.figli[i].frontiera();
      }
      return f;
   }
```

La seguente funzione di prova

- crea l'albero n-ario *a1* composto da sette nodi
- effettua la visita anticipata e differita di *a1*
- calcola l'altezza e la frontiera di *a1*.

```
function foo () {
   var a1 = new AlberoNArio(1);
   var a2 = new AlberoNArio(2);
   var a3 = new AlberoNArio(3);
   var a4 = new AlberoNArio(4);
   var a5 = new AlberoNArio(5);
   var a6 = new AlberoNArio(6);
   var a7 = new AlberoNArio(7);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiFiglio(a4);
   a2.aggiungiFiglio(a5);
   a2.aggiungiFiglio(a6);
   a3.aggiungiFiglio(a7);
   print(a1.visitaAnticipata());
   print(a1.visitaDifferita());
   print(a1.frontiera());
   print(a1.altezza());
}
```

13.4 Alberi n-ari con attributi

Un *albero n-ario con attributi* è un albero n-ario esteso associando uno o più attributi ad ogni nodo. Il costruttore *AlberoNArioAttr* contiene una proprietà *attributi* il cui valore è un array associativo, inizialmente vuoto. Le proprietà del costruttore sono tre: *valore*, *figli*, *attributi*.

```
this.valore = v;
this.figli = [];
this.attributi = {};
```

Il primo metodo nuovo da definire è *aggiungiAttributo*, con due parametri: il nome dell'attributo *n*, il valore dell'attributo *v*.

```
this.aggiungiAttributo =
   function (n, v) {
      this.attributi[n] = v;
   }
```

Il secondo metodo nuovo da definire è *leggiAttributo*, con un parametro, il nome dell'attributo *n*. Il metodo restituisce il valore dell'attributo *n*, *undefined* se l'attributo non è definito per l'albero.

```
this.leggiAttributo =
   function (n) {
      if (n in this.attributi) {
         return this.attributi[n];
      } else {
         return undefined;
      }
   }
```

La seguente funzione di prova

- crea l'albero n-ario con attributi *a1* composto da due nodi
- aggiunge l'attributo *x* associandogli il valore *100*
- stampa il valore dell'attributo *x* di *a1*.

```
function foo () {
   var a1 = new AlberoNArioAttr(1);
   var a2 = new AlberoNArioAttr(2);
   var a3 = new AlberoNArioAttr(3);
   a1.aggiungiFiglio(a2);
   a1.aggiungiFiglio(a3);
   a1.aggiungiAttributo("x", 100);
   print(a1.leggiAttributo("x"));
}
```

13.5 Esercizi

1. Un'*espressione aritmetica* può essere rappresentata mediante un albero binario i cui nodi sono *operatori* (addizione, sottrazione, moltiplicazione, divisione) e le foglie sono valori numerici.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano espressioni aritmetiche.

   L'oggetto ha i seguenti metodi:

   - *calcola* (): restituisce il valore dell'espressione,
   - *visualizza* (): restituisce una stringa che rappresenta l'espressione.

   Definire una funzione di prova che

   - crea l'espressione *2 * 3 + 5 – 6 / 2 + 1*
   - stampa la rappresentazione dell'espressione
   - calcola e stampa il valore dell'espressione.

2. L'*albero genealogico* è una rappresentazione (parziale) che mostra i rapporti familiari tra gli antenati di un individuo. Abitualmente un albero genealogico è realizzato utilizzando delle caselle, quadrate per i maschi e circolari per le femmine, contenenti i nomi di ciascuna persona, spesso corredate di informazioni aggiuntive, quali luogo e data di nascita e morte, occupazione o professione. Questi simboli, disposti dall'alto verso il basso in ordine cronologico, sono connessi da vari tipi di linee che rappresentano matrimoni, unioni extra coniugali, discendenza [Wikipedia, alla voce *Albero genealogic*o].

   Un *albero genealogico patrilineare* è un particolare tipo di albero genealogico in cui sono rappresentati tutti e soli i discendenti per via paterna.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano alberi genealogici patrilineari.

   Il costruttore ha un parametro: *persona*.

   L'oggetto ha i seguenti metodi:

   - *aggiungiConiuge* (*persona*, *coniuge*): aggiunge, nell'albero genealogico, il *coniuge* a una *persona*;
   - *aggiungiDiscendente* (*persona*, *discendente*): aggiunge, nell'albero genealogico, un *discendente* diretto a una *persona*;
   - *visualizzaPersona* (*persona*): restituisce una stringa contenente le informazioni relative a una *persona*;
   - *grado* (*persona*): calcola il grado di parentela tra una *persona* e il suo antenato più anziano nell'albero genealogico;

- *linea* (*persona*): restituisce una stringa contenente la linea di discendenza tra una *persona* e il suo antenato più anziano nell'albero genealogico.

Definire una funzione di prova che

- crea un albero genealogico con queste caratteristiche: *Bruno*, *Carlo* e *Daniela* sono figli di *Aldo*, *Enzo* e *Fabio* sono figli di *Bruno*, *Giacomo* è figlio di *Carlo*, *Alessia* è la moglie di *Aldo*, *Beatrice* è la moglie di *Bruno*, *Cecilia* è la moglie di *Carlo*;
- visualizza le informazioni di *Aldo* ed *Enzo*;
- calcola il grado di parentela di *Bruno*, *Giacomo* e *Fabio*;
- visualizza la linea di discendenza di *Enzo* e *Giacomo*.

14 Document Object Model

Un sito web è formato da pagine definite in HTML e rappresentate come alberi DOM. Questo capitolo presenta il linguaggio HTML e l'interfaccia programmatica DOM.

14.1 HTML

HTML (*HyperText Mark-up Language*) è un linguaggio per la descrizione strutturale di *documenti*, usato prevalentemente per definire *siti web*. La sua presentazione è volutamente molto ridotta: il lettore interessato ha a disposizione un'ampia letteratura su HTML, utile per approfondirne la conoscenza.

HTML non è un linguaggio di programmazione, in quanto non descrive algoritmi, ma si basa sull'uso di *etichette* o *marche* (*tag*) per la *marcatura* di porzioni di testo di un *documento HTML*[11]. Una marca è un elemento inserito nel testo di un documento per indicarne la struttura. Una marca è formata da una stringa racchiusa tra *parentesi angolari* ("<" e ">"). Subito dopo la parentesi angolare aperta si trova il nome della marca, seguito da eventuali attributi, che ne specificano le caratteristiche.

Poiché ogni marca determina la formattazione di una parte di un documento è necessario che questa parte sia indicata da una *marca di apertura*, che segna l'inizio della porzione di testo da formattare, e da una *marca di chiusura*, che ne segna la fine. La marca di chiusura è formata dal nome della marca, preceduto dal simbolo "/". Nel testo racchiuso dalle due marche (ovvero compreso tra quella di apertura e quella di chiusura) possono comparire altre marche il cui effetto si aggiunge a quello delle marche più esterne.

Ogni marca ha un diverso insieme di possibili attributi, dipendente dalla funzione della marca stessa. Alcune marche hanno uno o più attributi obbligatori, senza i quali non possono svolgere la loro funzione. La sintassi della definizione di un attributo è: *NomeAttributo*="*valore*". Il valore dell'attributo è racchiuso tra doppi apici.

Un documento è formato da un testo racchiuso dalla marca *html* e si divide in due parti: *intestazione* e *corpo*. L'intestazione, racchiusa dalla marca *head*, riporta il *titolo* del documento, mostrato sulla *barra del browser* quando il docu-

11 Nel seguito, per semplicità, i documenti HTML saranno chiamati documenti quando il contesto non crea ambiguità.

mento è visualizzato. Il corpo rappresenta il documento vero e proprio e contiene sia il testo da visualizzare, sia le marche di formattazione. Il corpo del documento è racchiuso dalla marca *body*.

Lo standard HTML 5 prevede che prima della marca *html* ci sia una dichiarazione DOCTYPE e che nella parte *head* ci sia una marca *meta* che indica l'insieme dei caratteri da usare nella visualizzazione del documento.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Titolo del documento</title>
 </head>
 <body>
  Contenuto del documento
<!-- questo è un commento -->
 </body>
</html>
```

Un programma JavaScript può essere associato a un documento tramite la marca *script*, che può comparire sia nell'intestazione che nel corpo del documento. In questo libro la marca *script* è sempre usato solo nell'intestazione. Il programma JavaScript ha la funzione di rendere interattivo il comportamento del documento quando è visualizzato da un browser. Tra i vari attributi della marca *script* quelli utilizzati nel libro sono due: *type*, che definisce il tipo di script, e *src*, descritto nel seguito. Per usare JavaScript all'interno di un documento è necessario assegnare il valore *text/javascript* all'attributo *type*.

Un programma JavaScript può essere associato a un documento in due modi:

- inserendolo direttamente tra la marca *script* di apertura e di chiusura;
- scrivendolo in un file separato che ha, per convenzione, *estensione* ".js" e indicando il nome del file come valore dell'attributo *src*.

Gli esempi presentati in questo libro usano la seconda tecnica.

14.2 Struttura e proprietà del DOM

Il *Document Object Model* (spesso abbreviato come *DOM*), letteralmente modello a oggetti del documento, è un'*interfaccia programmatica* per la rappresentazione di documenti. DOM è uno standard ufficiale del *World Wide Web Consortium* (spesso abbreviato come *W3C*), comunemente utilizzato per la programmazione di *applicazioni web* basate su documenti in formato XML e HTML [Wikipedia, alla voce *Document Object Model*].

Secondo il DOM, un documento è rappresentato da un albero che può essere visitato, modificato e trasformato mediante strumenti specifici o programmi scrit-

ti a tal scopo. Un *albero DOM*[12] è un albero n-ario con attributi. I nodi dell'albero sono di varia natura:

- *nodi documento* (document nodes)
- *nodi elemento* (element nodes)
- *nodi testo* (text nodes)
- *nodi attributo* (attribute nodes).

La struttura principale di un albero è formata da nodi elemento che possono avere un numero variabile di figli. La radice è sempre un nodo documento e i nodi testo possono essere solo foglie di un albero. I nodi attributo definiscono gli attributi dei nodi elemento. La radice dell'albero associato a un documento corrisponde all'oggetto *document*.

Ogni nodo ha le seguenti proprietà, oltre ad altre descritte nel seguito:

- *nodeType*
- *nodeName*
- *nodeValue*
- *childNodes*
- *attributes*.

Il *tipo* di un nodo, definito dalla proprietà *nodeType*, può avere i seguenti valori:

- 1: nodo elemento
- 2: nodo attributo
- 3: nodo testo
- 9: nodo documento.

Il *nome* di un nodo, definito dalla proprietà *nodeName*, ha un significato che dipende dal tipo del nodo:

- nodo elemento: marca HTML dell'elemento;
- nodo attributo: nome dell'attributo;
- nodo testo: valore predefinito *#text*,
- nodo documento: valore predefinito *#document*.

Il *valore* di un nodo, definito dalla proprietà *nodeValue*, ha un significato che dipende dal tipo del nodo:

- nodo elemento: *null*;

12 Nel seguito, per semplicità, gli alberi DOM saranno chiamati alberi quando il contesto non crea ambiguità.

- nodo attributo: valore dell'attributo;
- nodo testo: stringa associata al nodo;
- nodo documento: *null*.

Il valore della proprietà *childNodes* è un array che contiene i figli di un nodo elemento o di un nodo documento. Se un nodo non ha figli, il valore della proprietà è un array vuoto.

Il valore della proprietà *attributes* è un array che contiene gli attributi di un nodo elemento o di un nodo documento. Se un nodo non ha attributi, il valore della proprietà *attributes* è *null*.

14.3 Navigazione

Il modello DOM prevede diverse tecniche per visitare gli alberi e per muoversi (navigare) tra i suoi nodi. Queste tecniche si basano sul valore della proprietà *childNodes* e di altre proprietà che, quando sono definite (cioè quando il loro valore non è *null*), collegano un nodo ai suoi nodi vicini:

- *firstChild*: primo figlio del nodo;
- *lastChild*: ultimo figlio del nodo;
- *firstSibling*: primo fratello del nodo;
- *lastSibling*: ultimo fratello del nodo;
- *previousSibling*: fratello precedente del nodo;
- *nextSibling*: fratello successivo del nodo;
- *parentNode*: padre del nodo.

La visita dei figli di un nodo si effettua con un'iterazione determinata che usa la proprietà *childNodes*.

```
for (var i = 0; i < n.childNodes.length; i++) {
   var nodoFiglio = n.childNodes[i];
}
```

In alternativa, la stessa visita può essere effettuata con un'iterazione indeterminata, utilizzando le proprietà *firstSibling* e *nextSibling*.

```
var nodoFiglio = n.firstChild;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.nextSibling;
}
```

La visita può essere effettuata anche in ordine inverso, partendo dall'ultimo figlio e arrivando al primo. In questo caso si usano le proprietà *lastSibling* e *previousSibling*.

```
var nodoFiglio = n.lastChild;
while (nodoFiglio != null) {
   nodoFiglio = nodoFiglio.previousSibling;
}
```

Per visitare un albero partendo da un nodo fino ad arrivare alla sua radice si usa la proprietà *parentNode*. Nell'esempio che segue la visita parte dalla radice del nodo, procede scegliendo il primo figlio e poi torna indietro fino a ritornare alla radice.

```
while (nodo.firstChild != null) {
   nodo = nodo.firstChild;
}
while (nodo.parentNode != null) {
   nodo = nodo.parentNode;
}
```

14.4 Ricerca

La ricerca di un nodo si effettua utilizzando i seguenti metodi:

- *getElementsByTagName*
- *getElementById*.

Il metodo *getElementsByTagName* ha un unico argomento, una stringa *t* e può essere invocato su un nodo elemento o su un nodo documento. Il metodo effettua una visita anticipata dell'albero radicato nel nodo su cui è stato invocato e raccoglie in una collezione tutti e soli gli elementi con marca *t*, ovvero tutti e soli i nodi la cui proprietà *nodeName* ha un valore uguale a *t*. Alla fine della visita il metodo restituisce la collezione (possibilmente vuota).

Nell'esempio che segue la ricerca parte dalla radice dell'albero. L'iterazione determinata[13] scandisce la collezione così ottenuta.

```
var collezione = document.getElementsByTagName("script");
var nomiFile = [];
for (var i = 0; i < collezione.length; i++) {
   nomiFile.push(collezione[i].getAttribute("src"));
}
```

Il metodo *getElementById* ha un unico argomento *v* e può essere invocato su un nodo elemento o su un nodo documento. Il metodo effettua una visita anticipata dell'albero radicato nel nodo su cui è stato invocato e restituisce il nodo il cui attributo *id* ha un valore uguale a *v*, *null* altrimenti.

13 La scansione deve essere effettuata indicando esplicitamente la lunghezza della collezione.

14.5 Attributi

Gli attributi di un nodo sono gestiti dai seguenti metodi:

- *getAttribute*
- *setAttribute*
- *removeAttribute*.

Il metodo *getAttribute* ha un unico argomento, una stringa *a*. Può essere invocato su un nodo elemento o un nodo documento e restituisce il valore dell'attributo il cui nome è *a*, *undefined* altrimenti.

```
var valore = nodo.getAttribute("a");
```

Il metodo *setAttribute* ha due argomenti: il nome *a* di un attributo e il valore *v*. Il metodo modifica il valore dell'attributo *a* assegnandogli il valore *v*. Se l'attributo *a* non esiste, il metodo lo crea e gli assegna il valore *v*.

```
nodo.setAttribute("a", 123);
nodo.setAttribute("b", true);
```

Il metodo *removeAttribute* ha un parametro, il nome *a* dell'attributo che si intende eliminare dagli attributi nel nodo su cui si invoca il metodo.

```
nodo.removeAttribute("a");
```

14.6 Creazione e modifica

Un albero può essere modificato in due modi diversi: creando un nuovo nodo e aggiungendolo all'albero, eliminando un nodo esistente. Un nodo può essere creato usando i seguenti metodi:

- *createElement*
- *createTextNode*.

Il metodo *createElement* ha un parametro, una stringa, che determina il nome del nodo elemento che si vuole creare. Il metodo *createTextNode* ha un parametro, una stringa, che determina il valore del nodo testo che si vuole creare. Questi due metodi devono essere invocati sull'oggetto *document*.

```
var nodo1 = document.createElement("br");
var nodo2 = document.createTextNode("alfa");
```

Dopo aver creato un nuovo nodo è possibile aggiungerlo all'albero, utilizzando uno dei seguenti metodi:

- *appendChild*
- *insertBefore*

- *insertAfter*.

Il metodo *appendChild* ha un parametro, un nodo elemento, che è aggiunto in coda ai figli del nodo elemento su cui si invoca il metodo.

```
var nodo1 = document.createElement("br");
nodo.appendChild(nodo1);
```

Il metodo *insertBefore* ha due parametri, il nodo che si intende inserire e il nodo prima del quale deve essere inserito. Il metodo è invocato sul padre del secondo parametro.

```
var nodo2 = document.createTextNode("alfa");
nodo.insertBefore(nodo2, nodo.firstChild);
```

Il metodo *insertAfter* ha due parametri, il nodo che si intende inserire e il nodo dopo il quale deve essere inserito. Il metodo è invocato sul padre del secondo parametro.

```
var nodo2 = document.createTextNode("alfa");
nodo.insertAfter(nodo2, nodo.firstChild);
```

Un nodo, e tutto l'albero di cui tale nodo è la radice, può essere rimosso usando i seguenti metodi:

- *removeChild*
- *replaceChild*.

Il metodo *removeChild* ha un parametro, il nodo che si intende rimuovere. Il metodo deve essere invocato sul padre del nodo che si intende eliminare.

```
nodo.removeChild(nodo.firstChild);
```

Il metodo *replaceChild* ha due parametri, il nodo che si intende inserire e il nodo che si intende sostituire. Il metodo deve essere invocato sul padre del nodo che si intende sostituire.

```
var nodo1 = document.createElement("br");
nodo.replaceChild(nodo1, nodo.firstChild);
```

In molte situazioni in cui è necessario generare dinamicamente una parte di un documento. Anziché creare un albero DOM mediante i metodi di creazione appena visti e collegarlo al nodo di un albero DOM già esistente, è possibile creare il nuovo albero a partire da una stringa che rappresenta un frammento di codice HTML e assegnare questa stringa alla proprietà *innerHTML* del nodo. La stringa è trasformata in un albero DOM e tale albero è inserito come figlio del nodo.

```
nodo.innerHTML = "<br />Una riga dinamica<br />";
```

Pur non essendo definita nello standard *W3C* questa proprietà è, di fatto, gestita dalla maggior parte dei browser.

14.7 Esercizi

1. Definire una funzione JavaScript che visita un albero DOM e che restituisce il numero dei suoi nodi di tipo testo. Invocare la funzione usando come argomento la radice dell'albero DOM corrispondente alla pagina web dell'ambiente *EasyJS*.

15 Interattività

Una delle caratteristiche più interessanti delle pagine web è la loro *interattività*, cioè la capacità di reagire ad azioni effettuate dall'utente. Questo capitolo presenta i concetti e le tecniche di base necessarie per rendere interattiva una pagina web.

15.1 Eventi

Un *evento* è qualcosa che accade in seguito a un'azione di un utente che sta visualizzando un documento HTML. Gli eventi sono raggruppati in base ai seguenti aspetti che ne causano l'attivazione:

- tasti del mouse
- movimenti del mouse
- trascinamento del mouse
- tastiera
- modifiche del documento
- cambiamento del *focus*
- caricamento degli oggetti
- movimenti delle finestre
- pulsanti speciali.

Nel seguito è presentata una parte degli eventi, quelli più comuni. Il lettore interessato può trovare l'elenco completo degli eventi consultando la relativa letteratura.

Gli eventi attivati dai tasti del mouse sono:

- *click*: attivato quando si clicca su un oggetto;
- *dblClick*: attivato con un doppio click;
- *mouseDown*: attivato quando si schiaccia il tasto sinistro del mouse;
- *mouseUp*: attivato quando si alza il tasto sinistro del mouse precedentemente schiacciato.

Gli eventi *mouseDown* e *mouseUp* sono attivati dai due movimenti del tasto sinistro del mouse, il primo quando si preme il tasto e il secondo quando lo si solleva dopo il click. Il doppio click è un evento che ingloba gli altri e, per la precisione, attiva in successione *mouseDown*, *mouseUp*, *click*.

Gli eventi attivati dai movimenti del mouse sono:

- *mouseOver*: attivato quando il mouse si muove su un elemento;
- *mouseOut*: attivato quando il mouse si sposta fuori da un elemento;

Gli eventi *mouseOver* e *mouseOut* sono complementari: il primo è attivato nel momento in cui il puntatore è posto nell'area dell'oggetto e il secondo quando ne esce.

Gli eventi attivati dalla tastiera sono:

- *keyPress*: attivato quando si preme e si rilascia un tasto o anche quando lo si tiene premuto;
- *keyDown*: attivato quando si preme un tasto;
- *keyUp*: attivato quando un tasto, che era stato premuto, viene rilasciato.

Gli eventi legati al caricamento degli oggetti sono:

- *load*: attivato quando si carica un oggetto, ad esempio un documento o un'immagine;
- *unload*: è l'opposto del precedente ed è attivato quando si lascia una finestra per caricarne un'altra o per ricaricare la stessa (col tasto *refresh*).

15.2 Gestione degli eventi

Quando si verifica un evento il browser determina l'azione da compiere in risposta a tale evento. Ad esempio, quando si verifica un evento *click* accade quanto segue:

- l'utente punta un elemento di un documento HTML visualizzato in una pagina e clicca il pulsante sinistro;
- il browser individua il nodo che corrisponde all'elemento puntato dal mouse e verifica se questo nodo può "sentire" l'evento *click*;
- in caso affermativo, il browser esegue il codice che il nodo ha associato all'evento;
- in caso negativo, il browser ignora l'evento.

La gestione di un evento è effettuata da un frammento di codice JavaScript chiamato *gestore di evento*, associato al nodo coinvolto nell'evento. Questo codice è il valore di un attributo il cui nome è determinato convenzionalmente facendo precedere da *on* il nome dell'evento. Ad esempio, l'attributo per l'evento *click* è *onclick*.

Il valore degli attributi che fanno riferimento agli eventi è stabilito usando due tecniche distinte: indicandolo nel codice HTML, assegnandolo nel codice JavaScript associato al documento HTML. La prima tecnica è sconsigliata, perché introduce nel documento HTML parti di codice JavaScript che controllano il comportamento dinamico della pagina. Una buona regola da seguire consiste nel separare nettamente i due linguaggi. La seconda tecnica è più complessa da attuare ma evita l'inconveniente appena citato.

Per definire i gestori degli eventi degli elementi interattivi si definisce una funzione, di norma chiamata *inizializza*, e si associa questa funzione all'evento *load* dell'oggetto *window*. Questo evento accade quando termina il caricamento del documento HTML.

```
window.onload = inizializza;
```

La funzione *inizializza* è, a tutti gli effetti, il gestore dell'evento *load*. Nella prima parte della funzione *inizializza* si identificano tutti i nodi che saranno interessati agli eventi, utilizzando gli identificatori che compaiono nel documento HTML come valore dell'attributo *id* degli elementi interattivi. Ad ogni nodo sarà associato, tramite l'opportuno attributo, il nome del gestore dell'evento che si prenderà cura dell'evento stesso, al momento della sua attivazione. Nella seconda parte della funzione si inizializzano tutte le variabili globali dichiarate ma non inizializzate nel codice JavaScript.

15.3 Il convertitore di valuta interattivo

Per mostrare in concreto questa tecnica si può riprendere l'esempio del convertitore di valuta presentato in precedenza. Il convertitore di valuta diventa una pagina interattiva definita da un documento HTML che contiene due pulsanti (*Imposta*, *Converti*), tre campi di inserimento dati (*Valuta*, *Fattore*, *Importo da convertire*) e un campo di sola lettura (*Importo convertito*). Per semplicità di presentazione non si utilizzano *fogli di stile* CSS.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Convertitore di valuta</title>
  <script type="text/javascript"
          src="ConvertitoreValutaInterattivo.js"></script>
 </head>
 <body>
  <form>
   <input type="text"
```

```
          id="valuta"/> Valuta
   <input type="text"
          id="fattore"/> Fattore di conversione
   <input type="button"
          id="imposta"
          value="Imposta"/>
   <br/>
   <input type="text"
          id="importo"/> Importo da convertire
   <input type="button"
          id="converti"
          value="Converti"/>
   <br/>
   <input type="text"
          id="risultato"
          readonly="readonly"/> Importo convertito
  </form>
 </body>
</html>
```

Il comportamento interattivo della pagina è molto semplice e intuitivo. Per definire la valuta e il fattore di conversione l'utente inserisce i relativi valori nei campi *Valuta* e *Fattore* e poi pigia il pulsante *Imposta*. Una volta definita la valuta e il fattore di conversione l'utente converte un importo inserendo il suo valore nel campo *Importo da convertire* e pigiando il pulsante *Converti*. Il valore dell'importo, convertito secondo il fattore di conversione, apparirà nel campo *Importo convertito*.

Il programma JavaScript che realizza questo comportamento interattivo è il seguente. Il costruttore personalizzato *Convertitore*, relativo al convertitore di valuta, è esattamente quello definito in precedenza.

```
function Convertitore () {
   this.valutaCorrente;
   this.fattoreCorrente;
   this.imposta =
      function (valuta, fattore) {
         this.valutaCorrente = valuta;
         this.fattoreCorrente = fattore;
      }
   this.converti =
      function (importo) {
         return this.valutaCorrente + " " +
                importo * this.fattoreCorrente;
      }
}
function gestoreImposta () {
```

```
   var nodoV = document.getElementById("valuta");
   var valuta = nodoV.value;
   var nodoF = document.getElementById("fattore");
   var fattore = nodoF.value;
   c.imposta(valuta, fattore);
}
function gestoreConverti () {
   var nodoI = document.getElementById("importo");
   var importo = nodoI.value;
   var nodoR = document.getElementById("risultato");
   nodoR.value = c.converti(importo);
}
var c;
function inizializza () {
   var nodoI = document.getElementById("imposta");
   nodoI.onclick = gestoreImposta;
   var nodoC = document.getElementById("converti");
   nodoC.onclick = gestoreConverti;
   c = new Convertitore();
}
window.onload = inizializza;
```

Quando il browser carica il file *ConvertitoreValutaInterattivo.html*, non appena incontra la marca *script* esegue il codice JavaScript associato. Poiché la marca fa riferimento al file *ConvertitoreValutaInterattivo.js* il browser esegue il programma JavaScript contenuto nel file. Il risultato dell'esecuzione del programma sarà la dichiarazione del costruttore *Convertitore*, la dichiarazione della variabile *c*, la dichiarazione della funzione *inizializza,* l'associazione all'evento *load* dell'oggetto *window*, che rappresenta la finestra in cui compare il documento appena caricato, alla funzione *inizializza*.

Al termine del caricamento del file *ConvertitoreValutaInterattivo.html* viene generato l'evento *load*, gestito dalla funzione *inizializza*, che cerca il nodo il cui attributo *id* vale *imposta* e assegna la funzione *gestioneImposta* alla sua proprietà *onclick*, cerca il nodo il cui attributo *id* vale *converti* e assegna la funzione *gestioneConverti* alla sua proprietà *onclick* e, infine, assegna alla variabile *c* un oggetto *Convertitore*. Da questo momento in poi il documento diventa interattivo e le due funzioni *gestioneImposta* e *gestioneConverti* sono pronte a gestire gli eventi *click* generati dai pulsanti *Imposta* e *Converti*.

Quando l'utente pigia il pulsante *Imposta*, l'evento *click* viene generato e gestito dalla funzione *gestioneImposta* associata al nodo relativo al pulsante. La funzione cerca il nodo il cui attributo *id* vale *valuta* e assegna il campo *value* del nodo alla variabile *valuta*, cerca il nodo il cui attributo *id* vale *fattore* e assegna

il campo *value* del nodo alla variabile *fattore*, invoca il metodo *imposta* sull'oggetto *c*, usando come parametri le variabili *valuta* e *fattore*.

Quando l'utente pigia il pulsante *Converti*, l'evento *click* viene generato e gestito dalla funzione *gestioneConverti* associata al nodo relativo al pulsante. La funzione cerca il nodo il cui attributo *id* vale *importo* e assegna il campo *value* del nodo alla variabile *importo*, invoca il metodo *converti* sull'oggetto *c*, usando come parametro la variabile *importo*, cerca il nodo il cui attributo *id* vale *risultato* e assegna il valore restituito dal metodo all'attributo *value* del nodo.

15.4 Validazione dei valori di ingresso

Uno dei problemi da affrontare nella realizzazione di una pagina interattiva è la *validazione* dei *valori di ingresso*, ovvero di quei valori che sono stati inseriti dall'utente negli appositi campi. Nel caso del convertitore di valuta interattivo, i campi di ingresso sono tre: *Valuta*, *Fattore* e *Importo*. L'utente stabilisce quali valori inserire in questi campi prima di pigiare i tasti *Imposta* e *Converti*.

La validazione è effettuata dai gestori degli eventi che accedono in lettura al valore dei campi di ingresso. La validazione consiste in una sequenza di controlli che individuano i valori che non soddisfano i *vincoli* relativi ai campi. Se un valore non soddisfa anche uno solo dei vincoli, il gestore interrompe il calcolo e comunica all'utente che il valore non può essere accettato per portare a termine l'operazione associata all'evento. La comunicazione deve essere chiara ed esplicita, affinché l'utente possa modificare il valore e ripetere la richiesta.

Per semplicità, per comunicare all'utente l'errore commesso si utilizza il comando *alert(x)* che apre una finestra in cui compare il valore *x*. Per continuare l'esecuzione del programma l'utente deve pigiare il pulsante *OK* visualizzato nella finestra. Il comando *alert* deve essere usato con parsimonia e sempre per segnalare all'utente situazioni eccezionali. Le tecniche per comunicare correttamente con l'utente saranno presentate nel seguito.

Nel caso del convertitore di valuta il primo campo di ingresso da validare è quello relativo alla valuta (*Valuta*). Il valore che l'utente deve inserire è una qualsiasi stringa di caratteri, purché la sua lunghezza sia maggiore di zero. Tuttavia, per rendere realistica l'interfaccia, si può trasformare questo vincolo richiedendo che la stringa sia lunga esattamente tre caratteri. Una rapida consultazione di un qualsiasi programma per la conversione di valuta disponibile su Internet rivela, infatti, che il codice internazionale per la codifica delle valute prevede una stringa di tre caratteri. Ad esempio, il codice per la lira italiana è *LIT*, per il dollaro statunitense è *USD*, per la valuta giapponese è *YEN*.

Il secondo campo da validare è quello relativo al fattore di conversione (*Fattore*). L'utente deve inserire un valore numerico maggiore di zero. La validazione avviene a fasi successive. Per prima cosa si verifica che il campo non sia vuo-

to, poi si verifica che il valore contenuto nel campo sia numerico e, infine, che il valore numerico sia maggiore di zero. Il primo controllo si basa sulla lunghezza della stringa, che deve essere maggiore di zero. Il secondo controllo prevede la conversione della stringa in un numero mediante la funzione predefinita *Number(x)*, che riceve una stringa *x* e restituisce un valore numerico, *NaN* altrimenti. La costante *NaN* è riconosciuta nel secondo controllo dal predicato predefinito *isNaN(x)*. Il terzo controllo usa un predicato sul valore numerico.

Il gestore dell'evento associato al pulsante *Imposta* è così modificato.

```
function gestoreImposta () {
   var nodoV = document.getElementById("valuta");
   var valuta = nodoV.value;
   if (valuta.length != 3) {
      alert("la valuta non \e0056 valida");
      return;
   }
   var nodoF = document.getElementById("fattore");
   if (nodoF.value == "") {
      alert("il fattore di conversione \u00E8 vuoto");
      return;
   }
   var fattore = Number(nodoF.value);
   if (isNaN(fattore)) {
      alert(nodoF.value + " non \u00E8 un numero");
      return;
   }
   if (fattore <= 0) {
      alert("il fattore di conversione non \u00E8 valido");
      return;
   }
   c.imposta(valuta, fattore);
}
```

Il terzo campo da validare è quello relativo all'importo da convertire (*Importo*). Come per il fattore di conversione, l'utente deve inserire un valore numerico maggiore di zero. I controlli da effettuare sono analoghi a quelli del campo *Fattore*. Il gestore dell'evento associato al pulsante *Converti* è così modificato.

```
function gestoreConverti () {
   var nodoI = document.getElementById("importo");
   if (nodoI.value == "") {
      alert("l'importo \u00E8 vuoto");
      return;
   }
   var importo = Number(nodoI.value);
   if (isNaN(importo)) {
```

```
      alert(nodoI.value + " non \u00E8 un numero");
      return;
   }
   if (importo <= 0) {
      alert("l'importo non \u00E8 valido");
      return;
   }
   var nodoR = document.getElementById("risultato");
   nodoR.value = c.converti(importo);
}
```

15.5 Inizializzazione dei campi

Il caricamento della pagina a partire dal documento HTML fa sì che i campi siano inizializzati dal browser. In pratica, i quattro campi del convertitore interattivo saranno vuoti. Se l'utente decide di ricaricare la pagina, utilizzando il comando *refresh* del browser, i quattro campi mantengono i valori che avevano al momento del ricaricamento della pagina. Questo comportamento può essere alterato, inizializzando esplicitamente i campi al caricamento della pagina. La funzione *inizializza* è così modificata.

```
function inizializza () {
   var nodoI = document.getElementById("imposta");
   nodoI.onclick = gestoreImposta;
   var nodoC = document.getElementById("converti");
   nodoC.onclick = gestoreConverti;
   var nodoV = document.getElementById("valuta");
   nodoV.value = "";
   var nodoF = document.getElementById("fattore");
   nodoF.value = "";
   var nodoIm = document.getElementById("importo");
   nodoIm.value = "";
   var nodoR = document.getElementById("risultato");
   nodoR.value = "";
   c = new Convertitore();
}
```

Anche *gestoreImposta* deve essere modificato per tener conto del fatto che quando si imposta una nuova valuta e un nuovo fattore l'importo precedentemente inserito e l'importo convertito devono essere cancellati.

```
function gestoreImposta () {
   var nodoV = document.getElementById("valuta");
   var valuta = nodoV.value;
   if (valuta.length != 3) {
      alert("la valuta non \e0056 valida");
      return;
```

```
    }
    var nodoF = document.getElementById("fattore");
    if (nodoF.value == "") {
       alert("il fattore di conversione \u00E8 vuoto");
       return;
    }
    var fattore = Number(nodoF.value);
    if (isNaN(fattore)) {
       alert(nodoF.value + " non \u00E8 un numero");
       return;
    }
    if (fattore <= 0) {
       alert("il fattore di conversione non \u00E8 valido");
       return;
    }
    var nodoIm = document.getElementById("importo");
    nodoIm.value = "";
    var nodoR = document.getElementById("risultato");
    nodoR.value = "";
    c.imposta(valuta, fattore);
}
```

15.6 Dialogo

L'interazione tra l'utente e la pagina segue un *dialogo*, le cui regole stabiliscono la sequenza corretta di azioni che l'utente deve effettuare per ottenere i risultati desiderati. Nel caso del convertitore interattivo, l'unica regola che l'utente deve seguire prevede che prima di convertire un importo è necessario impostare la valuta e il fattore di conversione. Da quel momento in avanti l'utente può convertire uno o più importi, cambiare la valuta e il fattore di conversione e continuare a convertire altri importi. In pratica, l'utente non può effettuare come prima azione dopo il caricamento della pagina la conversione dell'importo.

Per controllare questo semplice dialogo si può utilizzare un *automa a stati finiti* che ha due stati: *iniziale*, *conversione*. Al momento del caricamento della pagina l'automa è nello stato *iniziale*. Se in questo stato si verifica l'evento relativo al pulsante *Imposta*, l'automa transita nello stato *conversione*. Se invece si verifica l'evento relativo al pulsante *Converti*, l'automa resta nello stato *iniziale* e si comunica all'utente che l'azione appena effettuata non è valida. Una volta entrato nello stato *conversione* le due azioni *Imposta* e *Converti* fanno restare l'automa nello stesso stato.

Per realizzare l'automa si utilizza una variabile *stato*, inizializzata alla costante *INIZIALE* (il cui valore può essere, ad esempio, zero). Il controllo sullo stato è effettuato dai due gestori degli eventi. Il primo, *gestoreImposta*, legge i campi *Valuta* e *Fattore*, invoca il metodo *imposta* e poi assegna alla variabile *stato* la

costante *CONVERSIONE* (il cui valore può essere, ad esempio, uno). Il secondo, *gestoreConverti*, controlla se il valore della variabile *stato* è uguale alla costante *INIZIALE* e, in questo caso, comunica all'utente l'errore nel dialogo, altrimenti legge il campo *Importo* e invoca il metodo *converti*. Le modifiche appena descritte sono riportate nel seguito.

```
function gestoreImposta () {
   var nodoV = document.getElementById("valuta");
   var valuta = nodoV.value;
   if (valuta.length != 3) {
      alert("la valuta non \e0056 valida");
      return;
   }
   var nodoF = document.getElementById("fattore");
   if (nodoF.value == "") {
      alert("il fattore di conversione \u00E8 vuoto");
      return;
   }
   var fattore = Number(nodoF.value);
   if (isNaN(fattore)) {
      alert(nodoF.value + " non \u00E8 un numero");
      return;
   }
   if (fattore <= 0) {
      alert("il fattore di conversione non \u00E8 valido");
      return;
   }
   var nodoIm = document.getElementById("importo");
   nodoIm.value = "";
   var nodoR = document.getElementById("risultato");
   nodoR.value = "";
   c.imposta(valuta, fattore);
   stato = CONVERSIONE;
}
function gestoreConverti () {
   if (stato == INIZIALE) {
      alert("valuta e fattore indefiniti");
      return;
   }
   var nodoI = document.getElementById("importo");
   if (nodoI.value == "") {
      alert("l'importo \u00E8 vuoto");
      return;
   }
   var importo = Number(nodoI.value);
   if (isNaN(importo)) {
      alert(nodoI.value + " non \u00E8 un numero");
```

```
      return;
   }
   if (importo <= 0) {
      alert("l'importo non \u00E8 valido");
      return;
   }
   var nodoR = document.getElementById("risultato");
   nodoR.value = c.converti(importo);
}
var c;
var stato;
var INIZIALE = 0;
var CONVERSIONE = 1;
function inizializza () {
   var nodoI = document.getElementById("imposta");
   nodoI.onclick = gestoreImposta;
   var nodoC = document.getElementById("converti");
   nodoC.onclick = gestoreConverti;
   var nodoV = document.getElementById("valuta");
   nodoV.value = "";
   var nodoF = document.getElementById("fattore");
   nodoF.value = "";
   var nodoIm = document.getElementById("importo");
   nodoIm.value = "";
   var nodoR = document.getElementById("risultato");
   nodoR.value = "";
   c = new Convertitore();
   stato = INIZIALE;
}
```

15.7 Gestione delle eccezioni

Durate l'esecuzione di un programma JavaScript possono verificarsi degli *errori dinamici*, detti anche errori a *run time* o *eccezioni*. Ad esempio, il programma può accedere agli elementi di un array senza averlo inizializzato correttamente. Se si verifica un errore dinamico l'esecuzione del programma è interrotta e l'errore è segnalato nell'apposita *console* del browser[14].

Per controllare queste situazioni, che in un programma corretto non dovrebbero verificarsi mai, si usa il comando *try* la cui sintassi è riportata nel seguito.

14 Ogni browser ha la sua console in cui sono visualizzati gli errori dinamici. In *Firefox 37.0.2* questa console (chiamata *Console web*) può essere aperta pigiando *CTRL+Maiusc+K* oppure mediante la sequenza di menu *Strumenti*, *Sviluppo web*, *Console web*.

```
<ComandoComposto> ::= <Try>
<Try> ::= try {<Blocco>}
          catch (<Identificatore>) {<Blocco>}
        | try {<Blocco>}
          catch (<Identificatore>) {<Blocco>}
          finally {<Blocco>}
```

Il comando *try* esegue il primo blocco di comandi. Se durante l'esecuzione si verifica un'eccezione viene eseguito il secondo blocco di comandi nel quale il valore della variabile indicata dopo la parola chiave *catch* è un oggetto che contiene alcune informazioni sull'eccezione. La versione estesa del comando *try* prevede un terzo blocco di comandi, che segue la parola chiave *finally*. Questo blocco sarà sempre e comuunque eseguito al termine dell'esecuzione del primo blocco. Di norma si utilizza la prima versione del comando *try*.

Il comando try può essere usato per controllare l'esecuzione dei gestori degli eventi, in modo da intercettare gli eventuali e molto probabili errori di programmazione che si commettono durante la messa a punto del codice JavaScript associato a una pagina interattiva. La tecnica consigliata prevede l'uso di un comando *alert* che riporta il nome del gestore durante la cui esecuzione si è verificato l'eccezione e il messaggio che la descrive. Ad esempio, *gestoreImposta* sarà così modificato.

```
function gestoreImposta () {
   try {
      var nodoV = document.getElementById("valuta");
      var valuta = nodoV.value;
      if (valuta.length != 3) {
         alert("la valuta non \e0056 valida");
         return;
      }
      var nodoF = document.getElementById("fattore");
      if (nodoF.value == "") {
         alert("il fattore di conversione \u00E8 vuoto");
         return;
      }
      var fattore = Number(nodoF.value);
      if (isNaN(fattore)) {
         alert(nodoF.value + " non \u00E8 un numero");
         return;
      }
      if (fattore <= 0) {
         alert("il fattore di conversione non \u00E8 valido");
         return;
      }
      var nodoIm = document.getElementById("importo");
```

```
      nodoIm.value = "";
      var nodoR = document.getElementById("risultato");
      nodoR.value = "";
      c.imposta(valuta, fattore);
      stato = CONVERSIONE;
   } catch ( e ) {
      alert("gestoreImposta " + e);
   }
}
```

La stessa modifica deve essere effettuata per *gestoreConverti* e per *inizializza*.

Da notare l'uso della rappresentazione esadecimale *\u00E8* per il carattere *è*. Se si fosse utilizzato direttamente tale carattere nella stringa il messaggio sarebbe stato visualizzato erroneamente perché la codifica delle costanti stringa utilizzate da JavaScript non prevede i caratteri speciali com segni diacritici.

In alcuni casi è necessario generare un'eccezione, per segnalare che durante l'esecuzione si è verificata una situazione inattesa. Per fare ciò si usa il comando *throw*, la cui sintassi è riportata nel seguito.

```
<ComandoSemplice> ::= <Throw>
<Throw> ::= throw <Espressione>;
```

Il comando *throw* deve essere usato solo ed esclusivamente per segnalare casi eccezionali e non per gestire situazioni normali che devono essere affrontate con altre tecniche.

15.8 Visualizzazione

L'uso del comando *alert* per comunicare con l'utente deve essere limitato a situazioni eccezionali. In tutti gli altri casi è bene utilizzare altre forme di comunicazione che si limitano a informare l'utente senza richiedere alcuna reazione da parte sua. Il comando *alert*, infatti, richiede che l'utente pigi il pulsante *OK* per chiudere la finestra che contiene la comunicazione.

Tra le tante forme possibili di comunicazione si deve prendere in considerazione la scrittura del messaggio che si vuole presentare all'utente in un'area della finestra. Nel caso del convertitore di valuta i messaggi sono relativi all'impostazione della valuta e alla conversione dell'importo. I primi possono essere visualizzati in un'area subito a destra del pulsante *Imposta*, i secondi subito a destra del pulsante *Converti*. Naturalmente altre soluzioni solo possibili: subito sotto (o sopra) il campo interessato al messaggio, in un'area unica destinata a visualizzare tutti i messaggi e così via.

Per visualizzare i messaggi in due aree della pagina è necessario modificare il codice HTML, introducendo due marche *span* a cui sono associati gli identificatori *messaggioImposta* e *messaggioConverti*.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Convertitore di valuta</title>
  <script type="text/javascript"
          src="ConvertitoreValutaInterattivo.js"></script>
 </head>
 <body>
  <form>
   <input type="text"
          id="valuta"/> Valuta
   <input type="text"
          id="fattore"/> Fattore di conversione
   <input type="button"
          id="imposta"
          value="Imposta"/>
   <span id="messaggioImposta"></span>
   <br/>
   <input type="text"
          id="importo"/> Importo da convertire
   <input type="button"
          id="converti"
          value="Converti"/>
   <span id="messaggioConverti"></span>
   <br/>
   <input type="text"
          id="risultato"
          readonly="readonly"/> Importo convertito
  </form>
 </body>
</html>
```

Il gestore di eventi *gestoreImposta* deve essere modificato, sostituendo opportunamente le invocazioni del comando *alert* che comunicano all'utente gli errori relativi ai campi *Valuta* e *Fattore*. Anziché essere usata come argomento del comando *alert*, la stringa che contiene la comunicazione dell'errore deve essere assegnata alla proprietà *innerHTML* del nodo associato alla marca *span* identificato come *messaggioImposta*. L'uso della proprietà *innerHTML* è giustificato dal fatto che la comunicazione modifica la pagina, visualizzando il messaggio esattamente nel punto della pagina relativo alla marca *span*.

```
function gestoreImposta () {
   try {
      var nodoMI = document.getElementById("messaggioImposta");
      var nodoV = document.getElementById("valuta");
```

```
      var valuta = nodoV.value;
      if (valuta.length != 3) {
         alert("la valuta non \e0056 valida");
         return;
      }
      var nodoF = document.getElementById("fattore");
      if (nodoF.value == "") {
         nodoMI.innerHTML = "il fattore di conversione \u00E8 vuoto";
         return;
      }
      var fattore = Number(nodoF.value);
      if (isNaN(fattore)) {
         nodoMI.innerHTML = nodoF.value + " non \u00E8 un numero";
         return;
      }
      if (fattore <= 0) {
         nodoMI.innerHTML = "il fattore di conversione " +
                            "non \u00E8 valido");
         return;
      }
      var nodoIm = document.getElementById("importo");
      nodoIm.value = "";
      var nodoR = document.getElementById("risultato");
      nodoR.value = "";
      c.imposta(valuta, fattore);
      stato = CONVERSIONE;
   } catch ( e ) {
      alert("gestoreImposta " + e);
   }
}
```

Analogamente si dovrà operare per *gestoreConverti*.

Questa soluzione, però, presenta un inconveniente: i messaggi non sono mai cancellati dall'area in cui compaiono, confondendo l'utente che non sa più se ha correttamente inserito i dati nei campi. Per eliminare l'inconveniente è sufficiente cancellare l'eventuale messaggio da ogni area come prima azione effettuata dai gestori degli eventi. Non è infatti sufficiente che ogni gestore cancelli i messaggi scritti sulla sua area perché l'utente può pigiare i pulsanti in qualsiasi ordine. Il codice modificato, relativo alla funzione *gestoreImposta*, è il seguente. Per semplicità è mostrata solo la parte iniziale della funzione.

```
function gestoreImposta () {
   try {
      var nodoMI = document.getElementById("messaggioImposta");
      nodoMI.innerHTML = "";
      var nodoMC = document.getElementById("messaggioConverti");
```

```
      nodoMI.innerHTML = "";
      ...
}
```

15.9 Selezione statica

La possibilità di impostare la valuta e il fattore di conversione è un aspetto dell'interfaccia utente che può essere migliorato, semplificando il dialogo con l'utente. Nella versione corrente l'utente può scegliere liberamente il codice della valuta e il fattore di conversione. Nella pratica, però, i codici delle valute sono definiti da uno standard e i fattori di conversione sono stabiliti in tempo reale dalle banche centrali. Sarebbe opportuno, quindi, presentare all'utente i codici possibili e recuperare i fattori di conversione mediante un collegamento con un opportuno *servizio web*. Il primo aspetto può essere agevolmente affrontato modificando il documento HTML e il programma JavaScript, il secondo richiede un impegno e competenze tecniche che esulano dagli obiettivi di questo libro.

Per evitare che l'utente debba inserire il nome della valuta in un campo di testo si può utilizzare un *menu di selezione* in cui compaiono tutti i nomi delle valute conosciute dal convertitore. Ad esempio, il menu potrebbe contenere le seguenti valute: *LIT* (Lira italiana), *USD* (Dollaro statunitense), *YEN* (Yen giapponese). La modifica riguarda solo il campo *Valuta*.

```
  <select
   <option>LIT</option>
   <option>USD</option>
   <option>YEN</option>
  </select> Valuta
```

L'uso di un menu di selezione modifica sostanzialmente il dialogo con l'utente. Al momento dell'inizializzazione, infatti, il menu presenta la prima opzione. L'utente può iniziare a convertire un importo senza dover impostare la valuta. Questo cambiamento elimina la necessità di avere due stati (*INIZIALE*, *CONVERSIONE*) e di controllare lo stato nei gestori degli eventi. Sarà necessario eliminare la variabile *stato* e le due costanti *INIZIALE* e *CONVERSIONE* e tutti i comandi in cui questa variabile e queste costanti sono utilizzate. Ovviamente, non essendo più necessario, si dovrà eliminare il pulsante *Imposta*.

L'altra modifica sostanziale riguarda l'impossibilità dell'utente di commettere errori nell'impostazione della valuta. Il nome della valuta sarà scelto tra quelli mostrati nel menu e il fattore di conversione sarà visualizzato dall'interfaccia, senza che l'utente debba inserirlo. Pertanto, non sarà più necessario effettuare i controlli sul campo *Fattore* e visualizzare gli eventuali messaggi in *messaggioImposta*.

Per visualizzare il fattore di conversione della valuta selezionata è necessario aggiungere una variabile globale *valuteFattori*, il cui valore è un array associativo che contiene i nomi delle valute e i corrispondenti fattori di conversione.

```
var valuteFattori = {
                     LIT : 1937.26,
                     USD : 0.95,
                     YEN : 0.13
                    }
```

La funzione gestoreImposta è così modificata, per tener conto delle osservazioni appena riportate.

```
function gestoreImposta () {
   try {
      var nodoMC = document.getElementById("messaggioConverti");
      nodoMC.innerHTML = "";
      var nodoV = document.getElementById("valuta");
      var valuta = nodoV.value;
      var fattore = valuteFattori[valuta];
      var nodoF = document.getElementById("fattore");
      nodoF.value = fattore;
      var nodoI = document.getElementById("importo");
      nodoI.value = "";
      var nodoR = document.getElementById("risultato");
      nodoR.value = "";
      c.imposta(valuta, fattore);
   } catch ( e ) {
      alert("gestoreImposta: " + e);
   }
}
```

La funzione *gestoreConverti* è quasi invariata. Deve essere solo eliminato il riferimento al campo *messaggioImposta*.

La funzione *inizializza* richiede qualche modifica sostanziale. Per prima cosa è necessario eliminare l'inizializzazione della variabile *stato*. Poi si deve eliminare il collegamento tra il nodo del pulsante *Imposta* (eliminato perché inutile) e il gestore degli eventi *gestoreImposta*. Al suo posto dovrà essere creato un collegamento tra il nodo del menu di selezione *Valuta* e *gestoreImposta*. Questa volta l'evento da gestire sarà *change* e l'attributo da usare sarà *onchange*. Infine, per impostare correttamente il convertitore al momento della sua attivazione sarà necessario invocare la funzione *gestoreImposta*, simulando ciò che avrebbe fatto l'utente selezionando la valuta mostrata nel menu.

```
function inizializza () {
   try {
```

```
      c = new Convertitore();
      var nodoC = document.getElementById("converti");
      nodoC.onclick = gestoreConverti;
      var nodoV = document.getElementById("valuta");
      nodoV.onchange = gestoreImposta;
      var nodoIm = document.getElementById("importo");
      nodoIm.value = "";
      var nodoR = document.getElementById("risultato");
      nodoR.value = "";
      gestoreImposta();
   } catch ( e ) {
      alert("gestoreLoad: " + e);
   }
}
```

15.10 Selezione dinamica

La soluzione appena presentata ha un difetto strutturale: le valute sono elencate sia in HTML che in JavaScript. Questa duplicazione può essere fonte di errori e imprecisioni in caso di modifica della lista. Per eliminare questo difetto è necessario eliminare la duplicazione, ad esempio mantenendo la lista nel codice JavScript e togliendola dal codice HTML.

Togliere la lista delle valute dal codice HTML significa avere un campo *select* in cui non compaiono *staticamente* le relative opzioni. Le opzioni dovranno essere create *dinamicamente* al momento del caricamento della pagina da parte del programma JavaScript. Per fare ciò è necessatio definire una funzione chiamata *opzioniValuta*, da invocare nel corpo della funzione *inizializza*. La funzione *opzioniValuta* restituisce una stringa esattamente uguale al codice HTML che definiva staticamente le opzioni del campo *select*. Tale stringa dovrà essere assegnata all'attributo *innerHTML* del nodo corrispondente al campo *Valuta*.

La funzione *opzioniValuta* è così definita.

```
function opzioniValuta () {
   var s = "";
   for (var valuta in valuteFattori) {
      s += "<option>" + valuta + "</option>"
   }
   return s;
}
```

La parte relativa al campo *Valuta* nella funzione *inizializza* è così modificata.

```
   var nodoV = document.getElementById("valuta");
   nodoV.innerHTML = opzioniValuta();
```

15.11 Cookie

Un *cookie* è un'informazione trasmessa da un sito web e memorizzata dal browser mentre l'utente sta visitando una pagina. Ogni volta che l'utente carica la pagina relativa al sito, il browser invia il cookie al server del sito per informarlo dell'attività svolta in precedenza dall'utente. In questo modo, il server tiene traccia del comportamento e delle abitudini dell'utente, raccogliendo informazioni utili che hanno un rilevante valore commerciale [Wikipedia, alla voce HTTP cookie].

Ci sono numerose categorie di cookie, ognuna delle quali svolge un ruolo nella navigazione su *Internet*. Per semplicità, il libro presenta solo i cosiddetti *cookie di sessione*, ovvero di quei cookie temporanei che vivono solo durante una *sessione di connessione* a una pagina web e che sono cancellati quando l'utente chiude il browser.

Un cookie di sessione ha la forma *nome = valore*, dove *nome* è un identificatore che indica il nome del cookie e *valore* è una stringa che ne rappresenta il valore. I cookie sono memorizzati in una variabile chiamata *document.cookie*, gestita dal browser. In scrittura è possibile assegnare a questa variabile una stringa che contiene il nome e il valore di un cookie, in lettura si ottiene una stringa che contiene tutti i cookie memorizzati fino a quel momento. La parte più complessa per la gestione dei cookie è l'estrazione del valore di uno specifico cookie dalla stringa memorizzata nella variabile *document.cookie*.

I cookie di sessione scadono (e sono eliminati dalla stringa memorizzata in *document.cookie*) quando l'utente chiude il browser. Per far sì che un cookie di sessione sopravviva alla fine della sessione è necessario indicare una *data di scadenza*. In questo caso, il cookie sarà cancellato solo allo scadere di tale data. La data di scadenza deve essere indicata quando si crea un cookie o quando se ne cambia il valore usando il formato *nome = valore; expires = data*, dove *data* è una stringa che codifica la data nel formato *UTC*[15]. Gli altri aspetti relativi ai cookie (cammino, dominio, sicurezza) non sono affrontati in questo libro.

Il convertitore di valuta può utilizzare un cookie di sessione per memorizzare l'ultima impostazione di valuta effettuata dall'utente. In questo semplice caso, il cookie deve soltanto memorizzare il nome della valuta. Al momento del caricamento della pagina, la funzione inizializza leggerà il cookie e imposterà la valuta utilizzando il valore memorizzato nel cookie.

15 UTC (un acronimo che sta per *Coordinated Universal Time*) è il *fuso orario* di riferimento da cui sono calcolati tutti gli altri fusi orari del mondo. Deriva dal *tempo medio di Greenwich* (in inglese *Greenwich Mean Time, GMT*), con il quale coincide a meno di approssimazioni infinitesimali [Wikipedia, alla voce UTC].

Prima di procedere alla modifica del convertitore di valuta è bene definire due funzioni che permettono di creare un cookie (*creaCookie*) e una che estrae un cookie (*estraiCookie*).

La funzione *creaCookie* restituisce una stringa nel formato previsto per i cookie. La parte più complessa consiste nella costruzione della stringa relativa alla data di scadenza, se il parametro *scadenza* è definito ed è un intero. Per prima cosa la funzione crea un oggetto *data* del tipo *Date*, inizializzato con la data di creazione. Poi calcola il numero di millisecondi corrispondenti ai giorni di scadenza (*scadenzaMilliSecondi*), estrae da data il numero di millisecondi corrispondenti alla data di creazione[16] (*dataOdierna*), somma a questo valore la *scadenzaMilliSecondi* e assegna il valore ottenuto a *dataFutura*, aggiorna l'oggetto *data* utilizzando il valore di *dataFutura*. Dopo questo laborioso procedimento la funzione crea la stringa nel formato previsto e la restituisce.

```
function creaCookie (nome, valore, scadenza) {
   if (scadenza == undefined || isNaN(scadenza)) {
      return nome + "=" + valore;
   }
   var data = new Date();
   var scadenzaMilliSecondi = scadenza * 24 * 60 * 60 * 1000;
   var dataOdierna = data.getTime();
   var dataFutura = dataOdierna + scadenzaMilliSecondi;
   data.setTime(dataFutura);
   return nome + "=" + valore + "; expires=" + data.toGMTString();
}
```

La funzione *estraiCookie* estrae il valore del cookie *nome* da *stringa*, in cui è memorizzato il valore di *document.cookie*. Per prima cosa la funzione crea un array (*coppie*) che contiene tutte le sottostringhe contenute in *stringa* e separate dal punto e virgola (per mezzo del metodo *split* del tipo *String*). Poi la funzione scandisce l'array *coppie* cercando la stringa che inizia con la sottostringa *nome*. Se non la trova restituisce *undefined*, altrimenti restituisce la seconda parte della coppia, ovvero il valore del cookie. La funzione usa il metodo *trim*, il metodo *substr* e il metodo *indexOf* del tipo *String*.

```
function estraiCookie (nome, stringa) {
   var coppie = stringa.split(";");
   var i = 0;
   while (i < coppie.length &&
          (coppie[i].trim().indexOf(nome) != 0)) {
      i++;
   }
```

16 L'oggetto *data* memorizza il numero di millisecondi trascorsi dalle ore 00:00:00 del 1 gennaio 1970. Per convenzione, questa è la data di inizio del tempo.

```
   if (i < coppie.length) {
      var l = nome.length;
      var s = coppie[i].trim();
      return s.substr(l + 1, s.length - l);
   } else {
      return undefined;
   }
}
```

La prima modifica da apportare è la scrittura del cookie. Poiché la valuta è impostata da *gestoreImposta* sarà proprio questa funzione a creare e scrivere il cookie il cui nome sarà proprio *valuta*. Per fare questo si utilizza la funzione *creaCookie*, a cui sono passati due argomenti: il nome del cookie e il suo valore.

```
   c.imposta(valuta, fattore);
   document.cookie = creaCookie("valuta", valuta);
```

La seconda modifica è relativa alla lettura del cookie. Nella versione corrente la valuta è impostata nella funzione inizializza usando il primo valore contenuto in *valuteFattori*. Il campo *valuta*, infatti, è un menu di selezione per il quale la creazione della lista delle opzioni è effettuata dalla funzione *opzioniValuta*. Per cambiare il valore di questo campo è sufficiente estrarre il valore del cookie e assegnarlo al campo *value* del suo nodo. Poiché il cookie potrebbe non essere stato creato in precedenza, è necessario verificare che il valore restituito dalla funzione *estraiCookie* non sia *undefined*. La modifica apportata alla funzione inizializza è la seguente.

```
   var valuta = estraiCookie("valuta", document.cookie);
   if (valuta != undefined) {
     nodoV.value = valuta;
   }
   gestoreImposta();
```

Il cookie utilizzato per il convertitore è di sessione. Ciò significa che chiudendo il browser il cookie è cancellato e, quindi, la valuta mostrata all'utente sarà quella che compare per prima nell'array *valuteFattore* e non quella impostata l'ultima volta dall'utente.

15.12 Esercizi

1. Definire un documento HTML relativo a una sveglia. Il documento contiene: un pulsante per impostare l'ora corrente e due campi di testo per l'inserimento dell'ora e dei minuti, un pulsante per impostare l'allarme e due campi di testo per l'inserimento dell'ora e dei minuti, un pulsante per fare avanzare di uno i minuti dell'ora corrente e un campo di sola lettura

che segnala l'allarme. Il valore dell'ora corrente e quello dell'allarme sono visualizzati in due campi di sola lettura.

2. Definire un documento HTML relativo a un distributore automatico di caffè. Il documento contiene: un pulsante che aggiunge un numero di capsule inserito in un campo di testo, un pulsante che eroga un numero di caffè inserito in un campo di testo addebitando il numero di caffè erogati a un codice inserito in un campo di testo, un pulsante che visualizza in un campo di testo di sola lettura il numero di caffè addebitati a un codice inserito in un campo di testo.

16 Gestione dei contenuti

Una pagina web visualizza informazioni di varia natura: testuale, grafica, audiovisiva. La gestione di queste informazioni, chiamate anche *contenuti*, diventa sempre più complessa all'aumentare della loro dimensione e varietà. Per affrontare questo problema è necessario ricorrere a tecniche di rappresentazione dei contenuti, di rappresentazione e di trattamento dinamico.

16.1 XML

XML è un linguaggio per la rappresentazione strutturata di dati. L'acronimo XML significa e**X**tensible **M**arkup **L**anguage. Come HTML, anche XML è un linguaggio di marcatura ma, a differenza di HTML, le marche non sono predefinite. La loro definizione è a carico di chi è interessato a rappresentare i propri dati. Un'altra differenza rispetto ad HTML consiste nel fatto che XML è stato progettato per rappresentare documenti ai fini della loro memorizzazione e trasmissione.

Questo semplice esempio di documento XML permette di codificare le valute e il relativo fattore di conversione.

```
<?xml version="1.0"?>
<valute>
 <valuta>
  <codice>LIT</codice>
  <fattore>1937.26</fattore>
 </valuta>
 <valuta>
  <codice>USD</codice>
  <fattore>0.95</fattore>
 </valuta>
 <valuta>
  <codice>YEN</codice>
  <fattore>0.13</fattore>
 </valuta>
</valute>
```

Il documento contiene quattro marche che descrivono le valute: *valute*, *valuta*, *codice*, *fattore*. La scelta di usare queste marche è dettata dal desiderio di codificare le informazioni contenute nel documento in maniera chiara e non ambi-

gua. Ovviamente, per comprendere il contenuto del documento è necessario conoscere il significato delle marche utilizzate al suo interno.

In alternativa è possibile descrivere le stesse informazioni trasformando le marche *codice* e *fattore* in due attributi con lo stesso nome.

```
<?xml version="1.0"?>
<valute>
 <valuta codice="LIT" fattore="1937.26"</valuta>
 <valuta codice="USD" fattore="0.95"   </valuta>
 <valuta codice="YEN" fattore="0.13"   </valuta>
</valute>
```

La scelta tra l'uso di una marca o di un attributo è spesso arbitraria. Di norma si usa una marca quando le informazioni associate sono strutturate. Ad esempio, nella prima versione del documento XML la marca *valute* contiene tante marche *valuta*; a sua volta la marca *valuta* contiene le marche *codice*, *fattore*. Nella seconda versione le marche codice e fattore sono state sostituite da due attributi. La scelta di usare queste due marche anziché usare due attributi è, pertanto, non giustificata. A giustificazione di tali scelte va detto che la struttura del documento XML è da considerare solo da un punto di vista esemplificativo. Infine, il documento è memorizzato nel file *Valute.xml*.

16.2 Il parser XML

Il contenuto di un documento XML può essere letto in JavaScript mediante un *parser*, un programma che riconosce la struttura del documento espressa mediante le marche e gli attributi associati e costruisce l'albero DOM corrispondente al documento. L'invocazione del parser può essere effettuata in modi diversi, che dipendono dal particolare browser utilizzato. La funzione *caricaXML* invoca il parser e funziona correttamente con la maggior parte dei browser.

```
function caricaXML(nomeFile) {
   var xmlhttp;
   if (window.XMLHttpRequest) {
      // IE7+, Firefox, Chrome, Opera, Safari
      xmlhttp = new XMLHttpRequest();
   } else {
      // IE6, IE5
      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
   }
   xmlhttp.open("GET", nomeFile, false);
   xmlhttp.send();
   return xmlhttp.responseXML;
}
```

La funzione restituisce l'albero che rappresenta il documento XML memorizzato nel file indicato nel parametro *nomeFile*.

Per ragioni di sicurezza, il parser può accedere solo a file memorizzati nella stessa cartella che contiene il documento HTML a cui è associato il programma JavaScript che invoca il parser. Per leggere il contenuto del file *Valute.xml* si usa il seguente comando nella funzione *inizializza* del convertitore e si invoca la funziona *creaDaDOM* che riceve in ingresso la radice dell'albero DOM appena costruito dal parser e che restituisce un array di valute e fattori, assegnato alla variabile *valuteFattori*.

```
   var radice = caricaXML("Valute.xml");
   valuteFattori = creaDaDOM(radice);
```

La funzione *creaDaDOM* ha il compito di trovare tutti i nodi la cui marca è *valuta* e, per ogni nodo, trovare il nodo con la marca *codice* e quello con la marca *fattore*. Trovati questi due nodi è immediato estrarre i valori corrispondenti e utilizzarli per costruire l'array associativo che sarà assegnato a *valuteFattori*.

```
function creaDaDOM (nodo) {
   var nodiValuta = nodo.getElementsByTagName("valuta");
   var a = {};
   for (var i = 0; i < nodiValuta.length; i++) {
      var nodoValuta = nodiValuta[i];
      var nodoCodice = nodoValuta.getElementsByTagName("codice")[0];
      var nodoFattore = nodoValuta.getElementsByTagName("fattore")[0];
      var codice = nodoCodice.firstChild.nodeValue;
      var fattore = Number(nodoFattore.firstChild.nodeValue);
      a[codice] = fattore;
   }
   return a;
}
```

Naturalmente, poiché l'array *valuteFattori* è creato a partire dal contenuto del file *Valute.xml* non sarà più necessario inizializzarlo esplicitamente in JavaScript. Così facendo, il convertitore di valuta può utilizzare un numero variabile di valute senza che sia necessario modificare il codice HTML e quello JavaScript.

La possibilità di scrivere pagine interattive il cui contenuto è completamente indipendentemente dal codice è una caratteristica fondamentale nella programmazione web. Come già visto per il convertitore di valuta, ciò permette di trasformare una *pagina web statica*, cioè una pagina che visualizza esclusivamente informazioni definite in un documento HTML o nel programma JavaScript associato, in una *pagina web dinamica* in cui le informazioni visualizzate sono contenute in un file XML.

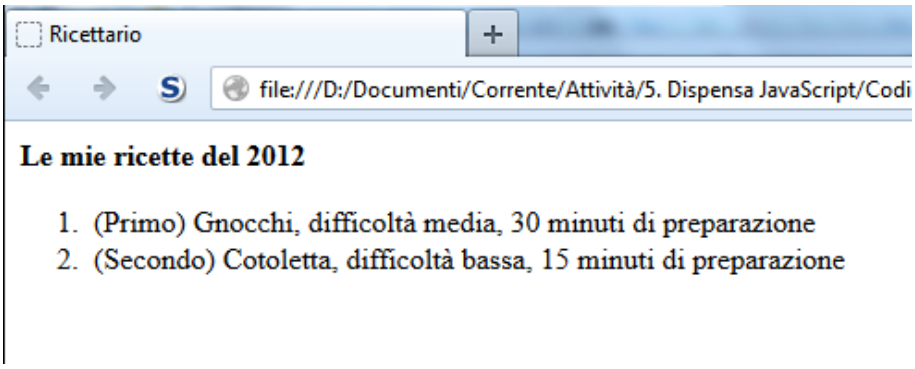
16.3 Creazione di oggetti definiti mediante XML

Nei casi più semplici i dati contenuti in un documento XML sono immediatamente utilizzabili in una pagina web dinamica per inizializzare uno o più array. In casi più complessi è necessario utilizzare una tecnica che crea un numero opportuno di oggetti a partire dai dati contenuti in un documento XML.

Si consideri, ad esempio, il seguente documento HTML che visualizza un *ricettario* formato da due semplici ricette.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Ricettario</title>
 </head>
 <body>
  <b>Le mie ricette del 2012</b>
  <br/>
  <ol>
   <li>(Primo) Gnocchi, difficolt&agrave; media,
       30 minuti di preparazione</li>
   <li>(Secondo) Cotoletta, difficolt&agrave; bassa,
       15 minuti di preparazione</li>
  </ol>
 </body>
</html>
```

Il ricettario sarà così visualizzato:



Volendo rendere dinamica questa pagina è necessario codificare i dati relativi alle ricette in un documento XML.

```
<?xml version="1.0"?>
<ricettario anno="2012">
 <ricetta categoria="Primo">
  <nome>Gnocchi</nome>
  <difficolta>media</difficolta>
  <preparazione>30</preparazione>
 </ricetta>
 <ricetta categoria="Secondo">
  <nome>Cotoletta</nome>
  <difficolta>bassa</difficolta>
  <preparazione>15</preparazione>
 </ricetta>
</ricettario>
```

Il documento, memorizzato nel file *Ricettario.xml*, contiene le seguenti marche che descrivono i dati del ricettario: *ricettario*, *ricetta*, *nome*, *difficolta*[17], *preparazione*. Oltre alle marche il documento contiene due attributi: *anno*, *categoria*.

In questo caso è immediato riconoscere gli oggetti che rappresentano le informazioni contenute nel documento XML: *Ricettario* e *Ricetta*. Questi oggetti sono, infatti, implicitamente definiti dalle marche che hanno lo stesso nome. L'oggetto *Ricettario* ha una proprietà *anno*, mentre l'oggetto *Ricetta* ha le proprietà *categoria*, *nome*, *difficolta* e *preparazione*. I due costruttori sono così (parzialmente) definiti.

```
function Ricettario () {
   this.anno;
   this.lista = [];
}
function Ricetta () {
   this.categoria;
   this.nome;
   this.difficolta;
   this.preparazione;
}
```

Dopo aver definito il documento XML che contiene le informazioni che saranno caricate dinamicamente dalla pagina, è possibile rivedere il documento HTML, eliminando tali informazioni e aggiungendo due identificatori, *titolo* e *lista*, usati dal programma JavaScript contenuto nel file *Ricettario.js*.

In particolare, la prima modifica è l'eliminazione della stringa *Le mie ricette del 2012* e l'aggiunta dell'identificatore *titolo* associato alla marca *b*. La seconda modifica consiste nell'eliminazione delle ricette e nell'aggiunta dell'identificatore *titolo* associato alla marca *ol*.

17 Nelle marche XML non si possono usare segni diacritici.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script type="text/javascript"
          src="Ricettario.js">
  </script>
  <title>Ricettario</title>
 </head>
 <body>
  <b id="titolo"></b>
  <br/>
  <ol id="lista"></ol>
 </body>
</html>
```

Il file *Ricettario.js* contiene un programma JavaScript che carica il file *Ricettario.xml*, crea un oggetto Ricettario e tanti oggetti Ricetta quante sono le ricette contenute nel documento XML, modifica dinamicamente il documento HTML.

La funzione *inizializza*, associata all'evento *onload*, carica il contenuto del file *Ricettario.xml* invocando la funzione *caricaXML*, crea il ricettario e invoca il metodo *inizializza*. La variabile globale *ilRicettario* è usata per memorizzare l'oggetto creato.

```
var ilRicettario;
function inizializza () {
   try {
      var radice = caricaXML("Ricettario.xml");
      ilRicettario = new Ricettario();
      ilRicettario.inizializza(radice);
   } catch ( e ) {
      alert("inizializza " + e);
   }
}
window.onload = inizializza;
```

Il metodo *inizializza* dell'oggetto *Ricettario* utilizza l'informazione relativa all'anno, recuperata dall'albero DOM costruito dal parser XML. Successivamente recupera tutti i nodi etichettati dalla marca *ricetta*, per ognuno di essi crea un oggetto *Ricetta*, lo inizializza e lo aggiunge alla lista delle ricette.

```
this.inizializza =
   function (radice) {
      var nodo = radice.getElementsByTagName("ricettario")[0];
      this.anno = nodo.getAttribute("anno");
      var nodi = nodo.getElementsByTagName("ricetta");
```

```
      for (var i = 0; i < nodi.length; i++) {
         var nodo = nodi[i];
         var ricetta = new Ricetta();
         ricetta.inizializza(nodo);
         this.lista.push(ricetta);
      }
   }
```

Il metodo *inizializza* dell'oggetto *Ricetta* utilizza le informazioni relative alla categoria, al nome, alla difficoltà e alla preparazione, recuperate dall'albero DOM costruito dal parser XML.

```
this.inizializza =
   function (nodo) {
      var c = nodo.getAttribute("categoria");
      this.categoria = c;
      var nodoN = nodo.getElementsByTagName("nome")[0];
      var n = nodoN.firstChild.nodeValue;
      this.nome = n;
      var nodoD = nodo.getElementsByTagName("difficolta")[0];
      var d = nodoD.firstChild.nodeValue;
      this.difficolta = d;
      var nodoP = nodo.getElementsByTagName("preparazione")[0];
      var p = nodoP.firstChild.nodeValue;
      this.preparazione = p;
   }
```

Dopo aver caricato il file XML e creato gli oggetti *Ricettario* e *Ricetta*, si procede a modificare dinamicamente il documento HTML in due punti: nell'elemento il cui *id* è *titolo* e in quello il cui *id* è *lista*. Per fare ciò si invocano i metodi *creaTitoloHTML* e *creaListaHTML*. La funzione *inizializza* è così modificata.

```
function inizializza () {
   try {
      var radice = caricaXML("Ricettario.xml");
      ilRicettario = new Ricettario();
      ilRicettario.inizializza(radice);
      var nodoTitolo = document.getElementById("titolo");
      nodoTitolo.innerHTML = ilRicettario.creaTitoloHTML();
      var nodoLista = document.getElementById("lista");
      nodoLista.innerHTML = ilRicettario.creaListaHTML();
      } catch ( e ) {
         alert("inizializza " + e);
      }
}
```

Il metodo *creaTitoloHTML* genera una stringa in formato HTML utilizzando la proprietà *anno* dell'oggetto *Ricettario*.

```
this.creaTitoloHTML =
   function () {
      var s = "Le mie ricette del " + this.anno;
      return s;
   }
```

Il metodo *creaListaHTML* genera una stringa in formato HTML invocando, per ogni oggetto *Ricetta*, il metodo *creaHTML*.

```
this.creaListaHTML =
   function () {
      var s = "";
      for (i = 0; i < this.lista.length; i++) {
         s += this.lista[i].creaHTML();
      }
      return s;
   }
```

Il metodo creaHTML genera una stringa in formato HTML utilizzando le proprietà dell'oggetto *Ricetta*.

```
this.creaHTML =
   function () {
      var s = "<li>" +
              "(" + this.categoria + ") " +
              this.nome + ", difficolt&agrave; " +
              this.difficolta + ", " +
              this.preparazione + " minuti di preparazione" +
              "</li>";
      return s;
   }
```

16.4 Ricerca esatta

Molto spesso in una pagina web è necessario effettuare delle *ricerche* sui dati gestiti dalla pagina stessa. La ricerca consiste nell'impostazione, da parte dell'utente, della *chiave* (o delle chiavi) di ricerca seguita dalla ricerca vera e propria delle informazioni che corrispondono alla chiave (o alle chiavi) e alla visualizzazione del risultato della ricerca. I *motori di ricerca* attualmente disponibili sul web utilizzano questo schema generale.

La prima tecnica di ricerca, chiamata *ricerca esatta*, consiste nell'indicare una chiave, una stringa, e cercare esattamente questa stringa nei dati gestiti dalla pagina. Ad esempio, la pagina che gestisce il ricettario potrebbe avere una

funzione di ricerca che permette all'utente di cercare una ricetta che ha un nome ben preciso. Se la ricetta è presente, il risultato della ricerca sarà la visualizzazione delle informazioni relative alla ricetta trovata, altrimenti sarà visualizzato un messaggio che indica che non esistono ricette con il nome indicato.

Il documento HTML relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script type="text/javascript"
          src="Ricettario.js">
  </script>
  <title>Ricettario</title>
 </head>
 <body>
  <form>
   <b id="titolo"></b>
   <br/>
   <input type="text" id="chiave">
   <input type="button" value="Cerca" id="cerca"/>
   <br/>
   <div id="lista"></div>
  </form>
 </body>
</html>
```

Il documento definisce un campo testuale (*chiave*) e un pulsante (*cerca*). L'utente inserisce la chiave nel campo testuale e pigia il pulsante. Il risultato della ricerca è mostrato nell'elemento *lista*.

Il codice JavaScript che realizza la ricerca esatta è così strutturato. La parte di inizializzazione è simile a quella già vista in precedenza. Per gestire il pulsante *cerca* si definisce la funzione *gestoreCerca*, che gestisce l'evento *onclick*.

```
function inizializza() {
   try {
      var radice = caricaXML("Ricettario.xml");
      ilRicettario = new Ricettario();
      ilRicettario.inizializza(radice);
      var nodoTitolo = document.getElementById("titolo");
      nodoTitolo.innerHTML = ilRicettario.creaTitoloHTML();
      var nodoCerca = document.getElementById("cerca");
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("inizializza " + e);
```

```
    }
}
```

La funzione *gestoreCerca* recupera il nodo associato all'elemento *chiave*, ne estrae il valore (la stringa di ricerca) e invoca il metodo *ricercaEsatta* dell'oggetto *Ricettario*. A seconda del risultato ottenuto, visualizza la ricetta trovata o una stringa che indica il fallimento della ricerca.

```
function gestoreCerca () {
   try {
      var nodoChiave = document.getElementById("chiave");
      var chiave = nodoChiave.value;
      var ricetta = ilRicettario.ricercaEsatta(chiave);
      var nodoLista = document.getElementById("lista");
      if (ricetta != null) {
         nodoLista.innerHTML = ricetta.creaHTML();
      } else {
         nodoLista.innerHTML = "Nessuna ricetta trovata";
      }
   } catch ( e ) {
      alert("gestoreCerca " + e);
   }
}
```

L'oggetto *Ricettario* è sostanzialmente uguale a quello definito in precedenza, ad eccezione del metodo *ricercaEsatta* che scandisce le ricette cercando quella la cui proprietà *nome* è esattamente uguale alla stringa *chiave*.

```
this.ricercaEsatta =
   function (chiave) {
      var i = 0;
      while (i < this.lista.length && this.lista[i].nome != chiave) {
         i++;
      }
      if (i < this.lista.length) {
         return this.lista[i];
      } else {
         return null;
      }
   }
```

L'oggetto *Ricetta* è identico a quello già visto in precedenza.

16.5 Ricerca con selezione statica

Lo svantaggio della ricerca esatta consiste nel fatto che l'utente deve conoscere esattamente la chiave di ricerca per ottenere il risultato voluto. Molto spesso, però, l'utente non conosce il valore della chiave di ricerca ma vuole scegliere tra

un insieme di valori possibili. In questo caso la ricerca avviene mediante un menu che presenta le opzioni possibili. Nel caso del ricettario si potrebbe presentare un *menu statico* che indica tutte le categorie con cui sono catalogate le ricette (ad esempio, le categorie potrebbero essere: antipasto, primo, secondo, contorno, dolce) e visualizzare tutte le ricette che appartengono alla categoria selezionata dall'utente. Se alla categoria selezionata non appartiene alcuna ricetta, la visualizzazione del risultato indica il fallimento della ricerca.

Il documento relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script type="text/javascript"
          src="Ricettario.js">
  </script>
  <title>Ricettario</title>
 </head>
 <body>
  <form>
   <b id="titolo"></b>
   <br/>
   <select id="selectCategoria">
    <option value="Antipasto">Antipasto</option>
    <option value="Primo">Primo</option>
    <option value="Secondo">Secondo</option>
    <option value="Contorno">Contorno</option>
    <option value="Dolce">Dolce</option>
  </select>
  <input type="button" value="Cerca" id="cerca"/>
  </br>
  <div id="lista"></div>
 </form>
</body>
</html>
```

Il documento definisce un menu statico (*selectCategoria*) con cinque opzioni, corrispondenti ai cinque possibili valori della proprietà *categoria* dell'oggetto *Ricetta*. Il pulsante per la ricerca (*cerca*) resta invariato, così come l'elemento (*lista*) utilizzato per visualizzare l'esito della ricerca.

Il codice JavaScript che realizza la ricerca mediante un menu statico è così strutturato. La parte di inizializzazione è identica a quella già vista in precedenza. La funzione *gestoreCerca* recupera la categoria selezionata dall'utente e invoca il metodo *ricercaCategoria* dell'oggetto *Ricettario* ottenendo, come risul-

tato, tutte le ricette che appartengono a quella categoria. Le ricette, se presenti, sono visualizzate dinamicamente.

```
function gestoreCerca () {
   try {
      var nodoSelect = document.getElementById("selectCategoria");
      var categoria = nodoSelect.value;
      var ricette = ilRicettario.ricercaCategoria(categoria);
      var nodoLista = document.getElementById("lista");
      if (ricette.length > 0) {
         var s = "";
         for (var i = 0; i < ricette.length; i++) {
            s += ricette[i].creaHTML();
         }
         nodoLista.innerHTML = s;
      } else {
         nodoLista.innerHTML = "Nessuna ricetta trovata";
      }
   } catch ( e ) {
      alert("gestoreCerca " + e);
   }
}
```

La ricerca vera e propria è effettuata dal metodo *ricercaCategoria* dell'oggetto *Ricettario*. Il metodo scandisce tutte le ricette del ricettario e costruisce un array con quelle la cui proprietà *categoria* è uguale al parametro *categoria*.

```
this.ricercaCategoria =
   function (categoria) {
      var ricette = [];
      for (var i = 0; i < this.lista.length; i++) {
         if (this.lista[i].categoria == categoria) {
            ricette.push(this.lista[i]);
         }
      }
      return ricette;
   }
```

L'oggetto *Ricetta* è sempre identico a quello già visto in precedenza.

16.6 Ricerca con selezione dinamica

Lo svantaggio dell'uso di un menu statico consiste nel fatto che l'utente può selezionare una categoria per la quale non sono presenti ricette. Per evitare questo inconveniente è necessario creare un *menu dinamico* in cui compaiono solo le categorie alle quali sono associate delle ricette. In questo modo la ricerca non avrà mai esito negativo.

Il documento relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script type="text/javascript"
          src="Ricettario.js">
  </script>
  <title>Ricettario</title>
 </head>
 <body>
  <form>
   <b id="titolo"></b>
   <br/>
   <select id="selectCategoria">
   </select>
   <input type="button" value="Cerca" id="cerca"/>
   </br>
   <div id="lista"></div>
  </form>
 </body>
</html>
```

Il documento definisce un menu (*selectCategoria*) che non ha opzioni. Le opzioni sono calcolate dinamicamente dopo aver caricato i dati dal file XML e determinato quali sono le categorie effettivamente presenti nelle ricette.

Il codice JavaScript che realizza la ricerca mediante un menu dinamico è così strutturato. La parte di inizializzazione è leggermente modificata rispetto a quella già vista in precedenza. La modifica riguarda l'aggiunta della creazione del menu dinamico, effettuata dal metodo *creaSelect* dell'oggetto *Ricettario*.

```
function inizializza () {
   try {
      var radice = caricaXML("Ricettario.xml");
      ilRicettario = new Ricettario();
      ilRicettario.inizializza(radice);
      var nodoTitolo = document.getElementById("titolo");
      nodoTitolo.innerHTML = ilRicettario.creaTitoloHTML();
      var nodoSelect = document.getElementById("selectCategoria");
      nodoSelect.innerHTML = ilRicettario.creaSelect();
      var nodoCerca = document.getElementById("cerca");
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("inizializza " + e);
   }
}
```

Il metodo *creaSelect* scandisce tutte le ricette e costruisce un array associativo (*categorie*) che contiene tutte le categorie presenti nelle ricette. Utilizzando questo array il metodo crea il menu dinamico.

```
this.creaSelect =
   function () {
      var categorie = {};
      for (var i = 0; i < this.lista.length; i++) {
         categorie[this.lista[i].categoria] = true;
      }
      var s = "";
      for (var i in categorie) {
         s += '<option value="' + i + '">' + i + '</option>';
      }
      return s;
   }
```

Il resto del codice resta invariato.

16.7 Ricerca con chiavi multiple

Le ricerche definite finora utilizzano solo la proprietà *categoria* delle ricette. Sarebbe utile poter cercare le ricette in base a più di una proprietà, in modo da rendere più precisa la richiesta dell'utente. Nel caso delle ricette, ad esempio, sarebbe utile poter indicare anche il tempo di preparazione, indicato per ogni ricetta dalla proprietà *preparazione*. In questo caso si tratta di una *ricerca a chiave multipla*, in cui gli elementi cercati devono soddisfare non un criterio ma una molteplicità di criteri.

Le ricerche a chiave multipla si suddividono, a loro volta, in ricerche in cui gli elementi cercati devono avere tutte le chiavi indicate (*ricerca a chiave multipla congiuntiva*) o in cui gli elementi devono avere almeno una delle chiavi indicate (*ricerca a chiave multipla disgiuntiva*). Tra questi due estremi si trovano le ricerche per le quali alcune chiavi devono essere necessariamente presenti e altre possono anche mancare. Nel seguito sarà presentata una ricerca a chiave multipla congiuntiva che utilizza due chiavi: *categoria* e *preparazione*.

Il documento relativo a questa tecnica di ricerca è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <script type="text/javascript"
          src="Ricettario.js">
  </script>
  <title>Ricettario</title>
```

```
 </head>
 <body>
  <form>
   <b id="titolo"></b>
   <br/>
   <select id="selectCategoria">
   </select>
   <select id="selectPreparazione">
   </select>
   <input type="button" value="Cerca" id="cerca"/>
   </br>
   <div id="lista"></div>
  </form>
</body>
</html>
```

Il documento definisce un nuovo menu (*selectPreparazione*) che non ha opzioni. Le opzioni sono calcolate dinamicamente dopo aver caricato i dati dal file XML e determinato quali sono i tempi di preparazione effettivamente presenti nelle ricette.

Il codice JavaScript che realizza la ricerca mediante due menu dinamici è così strutturato. La parte di inizializzazione è leggermente modificata rispetto a quella già vista in precedenza. La modifica riguarda l'aggiunta della creazione del menu dinamico relativo alla preparazione, effettuata dal metodo *creaSelectPrparazione* dell'oggetto *Ricettario*. Un'altra piccola modifica riguarda il metodo *creaSelect* che diventa *creaSelectCategoria*.

```
function inizializza () {
   try {
      var radice = caricaXML("Ricettario.xml");
      ilRicettario = new Ricettario();
      ilRicettario.inizializza(radice);
      var nodoTitolo = document.getElementById("titolo");
      nodoTitolo.innerHTML = ilRicettario.creaTitoloHTML();
      var nodoSelectC = document.getElementById("selectCategoria");
      nodoSelectC.innerHTML = ilRicettario.creaSelectCategoria();
      var nodoSelectP = document.getElementById("selectPreparazione");
      nodoSelectP.innerHTML = ilRicettario.creaSelectPreparazione();
      var nodoCerca = document.getElementById("cerca");
      nodoCerca.onclick = gestoreCerca;
   } catch ( e ) {
      alert("inizializza " + e);
   }
}
```

Il metodo *creaSelectPreparazione* scandisce tutte le ricette e costruisce un array associativo (*preparazione*) che contiene tutte le categorie presenti nelle ricette. Utilizzando questo array il metodo crea il menu dinamico. Il metodo è praticamente identico a *creaSelectCategoria*, già visto in precedenza.

La ricerca deve prendere in considerazione il valore delle due chiavi, selezionate dai due menu dinamici creati in fase di inizializzazione. Dopo aver estratto questi due valori il gestore del pulsante *cerca* (*gestoreCerca*) invoca il metodo *ricercaMultipla* ottenendo un array, possibilmente vuoto, contenente tutte le ricette che soddisfano la ricerca a chiave multipla. Il risultato è visualizzato come visto in precedenza.

```
function gestoreCerca () {
   try {
      var nodoSelectC = document.getElementById("selectCategoria");
      var categoria = nodoSelectC.value;
      var nodoSelectP = document.getElementById("selectPreparazione");
      var preparazione = nodoSelectP.value;
      var ricette = ilRicettario.ricercaMultipla(categoria,preparazione);
      var nodoLista = document.getElementById("lista");
      if (ricette.length > 0) {
         var s = "";
         for (var i = 0; i < ricette.length; i++) {
         s += ricette[i].creaHTML();
      }
         nodoLista.innerHTML = s;
      } else {
         nodoLista.innerHTML = "Nessuna ricetta trovata";
      }
   } catch ( e ) {
      alert("gestoreCerca " + e);
   }
}
```

Il metodo *ricercaMultipla* dell'oggetto *Ricettario* scandisce tutte le ricette selezionando quelle i cui attributi *categoria* e *preparazione* sono uguali ai valori dei parametri.

```
this.ricercaMultipla =
   function (categoria, preparazione) {
      var ricette = [];
      for (var i = 0; i < this.lista.length; i++) {
         if (this.lista[i].categoria == categoria &&
             this.lista[i].preparazione == preparazione) {
            ricette.push(this.lista[i]);
         }
      }
```

```
    return ricette;
}
```

16.8 Esercizi

1. Definire un documento HTML che visualizza dinamicamente una programmazione cinematografica definita in un documento XML. La programmazione contiene una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione.
2. Definire un documento HTML relativo alla programmazione cinematografica definita in un documento XML. La programmazione contiene una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione. Il documento contiene un menu a tendina, generato dinamicamente, contenente tutti e soli i cinema presenti nella programmazione, un pulsante che visualizza dinamicamente la lista dei film in programmazione presso il cinema selezionato.
3. Definire un documento HTML che visualizza dinamicamente un testo, permettendo di evidenziare selettivamente tutte le occorrenze di alcune parole. Il testo è memorizzato in un file XML in cui le parole evidenziabili sono opportunamente marcate.

17 Un esempio completo

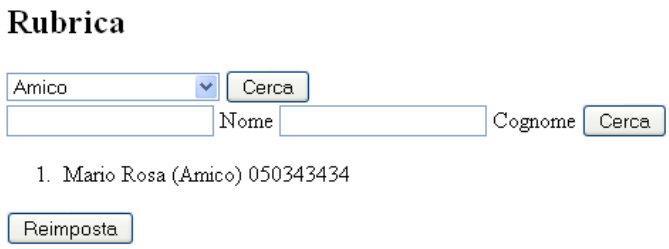
In questo capitolo presentiamo un esempio che ci serve per mettere in pratica i concetti presentati finora. Abbiamo scelto di mostrare la realizzazione di una rubrica sia per ragioni storiche sia perché con questo esempio possiamo affrontare una buona parte dei problemi che si presentano quando si vuole rendere dinamico un documento HTML mediante un programma JavaScript.

17.1 Il problema

Una *rubrica* è un elenco di *voci*, ognuna delle quali contiene informazioni su una persona: nome e cognome, numeri di telefono, indirizzi, data di nascita, tanto per citarne alcune. Tradizionalmente realizzate su carta, le rubriche si sono trasformate in oggetti digitali che troviamo su cellulari, calcolatori, telefoni. Questo esempio affronta il problema di realizzare una rubrica che permette di cercare le voci in base a due criteri di ricerca. Per semplicità, per ogni voce la rubrica memorizza il nome, il cognome, il numero di telefono e il gruppo di appartenenza di una persona.

17.2 Il codice HTML

La rubrica è realizzata mediante un documento HTML. La ricerca per gruppo utilizza un menu a tendina mediante il quale si seleziona uno dei gruppi di appartenenza delle persone. Il pulsante alla destra del menu effettua la ricerca delle voci che saranno poi elencate nella parte bassa del documento. La ricerca per nome o per cognome utilizza due campi di testo nei quali si inseriscono i valori da usare per la ricerca. Anche in questo caso, il pulsante alla destra dei due campi effettua la ricerca. L'esito negativo della ricerca è segnalato da una finestra di *alert*. Il pulsante *Reimposta* cancella il contenuto delle caselle e aggiorna la scelta corrente del menu.



La figura mostra il risultato della ricerca delle voci che appartengono al gruppo *Amico*.

Il codice completo del documento *Rubrica.html* è il seguente.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <title>Rubrica</title>
  <script type="text/javascript"
          src="Rubrica.js"></script>
 </head>
 <body>
  <h2>Rubrica</h2>
  <form>
   <select id="selectGruppo" size="1"></select>
   <input type="button"
          value="Cerca"
          id="cercaPerGruppo"/>
   <br/>
   <input type="text"
          id="nome"/> Nome
   <input type="text"
          id="cognome"/> Cognome
   <input type="button"
          value="Cerca"
          id="cercaPerNomeCognome"/>
   <br/>
   <ol id="elenco"></ol>
   <input type="reset"
          id="reimposta"/>
  </form>
 </body>
</html>
```

Il documento contiene un modulo al cui interno sono dichiarati un menu, tre pulsanti, due campi di testo, una lista di elementi puntati. Il menu e la lista sono generati dinamicamente, come vedremo nel seguito. Il codice JavaScript è contenuto nel file *Rubrica.js*, come indicato dal tag *script*.

17.3 Gli oggetti Rubrica e Voce

Le informazioni relative alla rubrica sono mantenute in due oggetti: *Voce* e *Rubrica*. L'oggetto *Voce* memorizza le informazioni relative a una voce. L'oggetto *Rubrica* gestisce tutti gli aspetti relativi alla rubrica vera e propria e contiene un array di oggetti *Voce*.

Il costruttore dell'oggetto *Voce* ha quattro proprietà: *nome*, *cognome*, *telefono*, *gruppo*.

```
function Voce () {
   this.nome;
   this.cognome;
   this.telefono;
   this.gruppo;
}
```

Il costruttore dell'oggetto *Rubrica* ha due proprietà *voci* e *gruppi*, inizializzate entrambi con un array vuoto. La proprietà *voci* contiene le voci della rubrica, la proprietà *gruppi* contiene i gruppi di appartenenza delle voci.

```
function Rubrica () {
   this.voci = [];
   this.gruppi = {};
}
```

17.4 Caricamento e inizializzazione

La creazione della rubrica, inizialmente vuota, viene effettuata dal gestore dell'evento *load* (evento generato al termine del caricamento del documento HTML) che invoca il metodo *inizializza* del costruttore *Rubrica*, il cui unico parametro è il nome del file (*Rubrica.xml*) che contiene la descrizione del contenuto della rubrica.

```
<?xml version="1.0"?>
<rubrica>
 <voce gruppo="Amico">
  <nome>Mario</nome>
  <cognome>Rosa</cognome>
  <telefono>050343434</telefono>
 </voce>
 <voce gruppo="Parente">
  <nome>Carlo</nome>
  <cognome>Neri</cognome>
  <telefono>050434343</telefono>
 </voce>
</rubrica>
```

Il gestore dell'evento *load* è il seguente.

```
var r;

function inizializza () {
   try {
```

```
      r = new Rubrica();
      r.inizializza("Rubrica.xml");
   } catch ( e ) {
      alert("inizializza " + e);
   }
}

window.onload = inizializza;
```

Il metodo *inizializza* dell'oggetto *Rubrica* è il seguente.

```
this.inizializza =
   function (nomeFile) {
      var doc = caricaXML(nomeFile);
      var l = doc.getElementsByTagName("voce");
      for (var i = 0; i < l.length; i++) {
         var v = new Voce();
         v.inizializza(l[i]);
         this.voci.push(v);
         this.gruppi[v.gruppo] = v.gruppo;
      }
   }
```

Questo metodo utilizza il metodo *inizializza* dell'oggetto *Voce*.

```
this.inizializza =
   function (nodo) {
      var nodoN = nodo.getElementsByTagName("nome")[0];
      this.nome = nodoN.firstChild.nodeValue;
      var nodoC = nodo.getElementsByTagName("cognome")[0];
      this.cognome = nodoC.firstChild.nodeValue;
      var nodoT = nodo.getElementsByTagName("telefono")[0];
      this.telefono = nodoT.firstChild.nodeValue;
      this.gruppo = nodo.getAttribute("gruppo");
   }
```

17.5 Gestione degli eventi

La creazione dinamica del menu di selezione per la ricerca basata sul nome del gruppo è effettuata dal metodo *creaSelect* dell'oggetto *Rubrica*.

```
this.creaSelect =
   function () {
      var s = "";
      s += '<option value="" selected="selected">' +
           'Seleziona un gruppo</option>'
      var l = this.gruppi;
      for (i in l) {
```

```
         s += '<option value="' + l[i] + '">' +
              l[i] + '</option>';
      }
      return s;
   }
```

La gestione degli eventi *click* (eventi generati premendo uno dei tre pulsanti presenti nel documento) viene effettuata estendendo la definizione della funzione *inizializza*.

```
function inizializza () {
   try {
      var p1 = document.getElementById("cercaPerGruppo");
      p1.onclick = cercaPerGruppo;
      var p2 = document.getElementById("cercaPerNomeCognome");
      p2.onclick = cercaPerNomeCognome;
      var p3 = document.getElementById("reimposta");
      p3.onclick = reimposta;
      r = new Rubrica();
      r.inizializza("Rubrica.xml");
      var nodo = document.getElementById("selectGruppo");
      nodo.innerHTML = r.creaSelect();
   } catch ( e ) {
      alert("inizializza " + e);
   }
}
```

17.6 Ricerca di voci

Le funzioni di ricerca previste per la rubrica sono attivate dai due pulsanti che affiancano il menu e le caselle di testo. Per ogni pulsante è stata definita una funzione, associata al relativo evento *click*. Per la ricerca basata sul nome del gruppo la funzione si chiama *cercaPerGruppo*, per la ricerca basata sul nome o sul cognome la funzione si chiama *cercaPerNomeCognome*.

```
function cercaPerGruppo () {
   try {
      var s = document.getElementById("selectGruppo");
      var i = 0;
      while ((i < s.options.length) &&
             !s.options[i].selected) {
         i++;
      }
      var l = r.cercaGruppo(s.options[i].value);
      var nodo = document.getElementById("elenco");
      if (l == "") {
         nodo.innerHTML = null;
```

```
         alert("Nessuna voce trovata");
      } else {
         nodo.innerHTML = r.visualizza(l);
      }
   } catch ( e ) {
      alert("cercaPerGruppo " + e);
   }
}

function cercaPerNomeCognome () {
   try {
      var n = document.getElementById("nome").value;
      var c = document.getElementById("cognome").value;
      var l = r.cercaNomeCognome(n, c);
      var nodo = document.getElementById("elenco");
      if (l == "") {
         nodo.innerHTML = null;
         alert("Nessuna voce trovata");
      } else {
         nodo.innerHTML = r.visualizza(l);
      }
   } catch ( e ) {
      alert("cercaPerNomeCognome " + e);
   }
}
```

Per effettuare la ricerca le due funzioni invocano, rispettivamente, il metodo *cercaGruppo* e *cercaNomeCognome* dell'oggetto *Rubrica*.

```
this.cercaGruppo =
   function (g) {
      var l = [];
      for (i in this.voci) {
         if (this.voci[i].gruppo == g) {
            l.push(this.voci[i]);
         }
      }
      return l;
   }
this.cercaNomeCognome =
   function (n, c) {
      var l = [];
      for (i in this.voci) {
         if ((this.voci[i].nome == n) ||
             (this.voci[i].cognome == c)) {
            l.push(this.voci[i]);
         }
```

```
      }
      return l;
   }
```

17.7 Visualizzazione

La visualizzazione dei risultati utilizza una finestra di *alert*, in caso di esito negativo, il metodo *visualizza* dell'oggetto *Rubrica* altrimenti.

```
this.visualizza =
   function (l) {
      var s = "";
      for (i in l) {
         s += l[i].visualizza();
      }
      return s;
   }
```

La visualizzazione delle informazioni di una voce è a carico del metodo *visualizza* dell'oggetto *Voce*.

```
this.visualizza =
   function () {
      var n = this.nome;
      var c = this.cognome;
      var g = this.gruppo;
      var t = this.telefono;
      return '<li>' +
             n + ' ' + c + " (" + g + ") " + t +
             '</li>';
   }
```

Infine, il pulsante *Reimposta* è associato alla funzione *reimposta*.

```
function reimposta () {
   try {
      var nodo = document.getElementById("elenco");
      nodo.innerHTML = null;
   } catch ( e ) {
      alert("reimposta " + e);
   }
}
```

18 Soluzione degli esercizi della seconda parte

18.1 Esercizi del capitolo 11

1. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la differenza tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   5, 0

   3, 3

   9, 2.

```
function sottrazione (x, y) {
   if (y == 0) {
      return x;
   } else {
      return sottrazione(x, y -1) -1;
   }
}
print(sottrazione(5, 0));
print(sottrazione(3, 3));
print(sottrazione(9, 2));
```

2. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la relazione di minore tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 0

   0, 4

   3, 0

   2, 6

   9, 2.

```
function minoreDi (x, y) {
   if (y == 0) {
      return false;
   } else if (y != 0 && x == 0) {
      return true;
   } else {
      return minoreDi(x -1, y -1);
   }
}
print(minoreDi(0, 0));
print(minoreDi(0, 4));
print(minoreDi(3, 0));
print(minoreDi(2, 6));
print(minoreDi(9, 2));
```

3. Definire in JavaScript una funzione ricorsiva che calcola e restituisce la divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

   3, 3

   9, 2.

```
function divisione (x, y) {
   if (minoreDi(x, y)) {
      return 0;
   } else {
      return divisione(sottrazione(x, y), y) +1;
   }
}
print(divisione(0, 4));
print(divisione(3, 3));
print(divisione(9, 2));
```

4. Definire in JavaScript una funzione ricorsiva che calcola e restituisce il resto della divisione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   0, 4

   3, 3

   9, 2.

```
function resto (x, y) {
   if (minoreDi(x, y)) {
```

```
        return x;
    } else {
        return resto(sottrazione(x, y), y);
    }
}
print(resto(0, 4));
print(resto(3, 3));
print(resto(9, 2));
```

5. Definire in JavaScript una funzione ricorsiva che calcola e restituisce l'esponenziazione tra due interi maggiori o uguali a zero.

   La funzione ha due parametri: *x*, *y*.

   Invocare la funzione con le seguenti coppie di valori:

   6, 0

   3, 3

   0, 2.

```
function esponente (x, y) {
    if (x != 0 && y == 0) {
        return 1;
    } else if (x != 0 && y != 0) {
        return moltiplicazione(x, esponente(x, y -1));
    } else if (x == 0 && y != 0) {
        return 0;
    }
}
print(esponente(6, 0));
print(esponente(3, 3));
print(esponente(0, 2));
```

18.2 Esercizi del capitolo 12

1. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *sveglie*.

   L'oggetto ha i seguenti metodi:

   - *oraCorrente* (*h*, *m*): imposta l'ora corrente. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
   - *allarme* (*h*, *m*): imposta l'allarme. Se *h* è maggiore di 23 o *m* è maggiore di 59, i rispettivi valori sono impostati a zero;
   - *tic* (): fa avanzare di uno i minuti dell'ora corrente. Se i minuti arrivano a 60, azzera i minuti e fa avanzare di uno le ore. Se le ore arrivano a 24, azzera le ore. Se l'ora corrente è uguale all'allarme impostato, restituisce il valore *true*, *false* altrimenti.

Definire una funzione di prova che

- crea una sveglia,
- imposta l'ora corrente alle 13:00,
- imposta l'allarme alle 13:02,
- fa avanzare l'ora corrente di due minuti.

```
function Sveglia () {
   this.ore;
   this.minuti;
   this.oreAllarme;
   this.minutiAllarme;
   this.oraCorrente =
      function (h, m) {
         if (h > 23) {
            this.ore = 0;
         } else {
            this.ore = h;
         }
         if (m > 59) {
            this.minuti = 0;
         } else {
            this.minuti = m;
         }
      }
   this.allarme =
      function (h, m) {
         if (h > 23) {
            this.oreAllarme = 0;
         } else {
            this.oreAllarme = h;
         }
         if (m > 59) {
            this.minutiAllarme = 0;
         } else {
            this.minutiAllarme = m;
         }
      }
   this.tic =
      function () {
         this.minuti++;
         if(this.minuti == 60) {
            this.minuti = 0;
            this.ore++;
         }
         if (this.ore == 24) {
```

```
            this.ore = 0;
         }
         return ((this.ore == this.oreAllarme) &&
                 (this.minuti == this.minutiAllarme));
      }
}
function foo () {
   var s = new Sveglia();
   s.oraCorrente(13, 0);
   s.allarme(13, 2);
   print(s.tic());
   print(s.tic());
}
```

2. Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano *distributori automatici di caffè*.

   L'oggetto ha i seguenti metodi:

   - *carica* (*n*): aggiunge *n* capsule per erogare *n* caffè;
   - *eroga* (*n*, *c*): verifica se è possibile erogare *n* caffè e, in caso positivo, li eroga addebitandoli al codice *c*;
   - *rapporto* (*c*): restituisce il numero di caffè addebitati al codice *c* e il numero di capsule disponibili.

   Definire una funzione di prova che:

   - crea un distributore automatico di caffè,
   - carica 20 capsule,
   - eroga 12 caffè per il codice *Carlo*,
   - genera il rapporto per il codice *Carlo*.

```
function Distributore () {
   this.capsule = 0;
   this.crediti = {};
   this.carica =
      function (n) {
         this.capsule += n;
      }
   this.eroga =
      function (n, c) {
         if (n <= this.capsule){
            this.capsule -= n;
            if (c in this.crediti) {
               this.crediti[c] += n;
            } else {
               this.crediti[c] = n;
```

```
            }
         }
      }
   this.rapporto =
      function (c) {
         return c + " " + this.crediti[c] +
                " Capsule " + this.capsule;
      }
}
function foo () {
   var d = new Distributore();
   d.carica(20);
   d.eroga(12, "Carlo");
   print(d.rapporto("Carlo"));
}
```

18.3 Esercizi del capitolo 13

1. Un'*espressione aritmetica* può essere rappresentata mediante un albero binario i cui nodi sono *operatori* (addizione, sottrazione, moltiplicazione, divisione) e le foglie sono valori numerici.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano espressioni aritmetiche.

   L'oggetto ha i seguenti metodi:

   - *calcola* (): restituisce il valore dell'espressione,
   - *visualizza* (): restituisce una stringa che rappresenta l'espressione.

   Definire una funzione di prova che

   - crea l'espressione *2 * 3 + 5 – 6 / 2 + 1*
   - stampa la rappresentazione dell'espressione
   - calcola e stampa il valore dell'espressione.

```
function Espressione (v, sx, dx) {
   this.valore   = v;
   this.sinistro = sx;
   this.destro   = dx;
   this.visualizza =
      function () {
      var vs;
      if (this.sinistro != null) {
         vs = this.sinistro.visualizza ();
      } else {
         vs = "";
      }
```

```
      var vd;
      if (this.destro != null) {
         vd = this.destro.visualizza ();
      } else {
         vd = "";
      }
      return vs + " " + this.valore + vd;
   }
      this.calcola =
         function () {
            if (this.sinistro == null &&
                this.destro == null) {
               return this.valore;
            }
            switch (this.valore) {
               case "+": var vs = this.sinistro.calcola();
                         var vd = this.destro.calcola();
                         return vs + vd;
               case "-": var vs = this.sinistro.calcola();
                         var vd = this.destro.calcola();
                         return vs - vd;
               case "*": var vs = this.sinistro.calcola();
                         var vd = this.destro.calcola();
                         return vs * vd;
               case "/": var vs = this.sinistro.calcola();
                         var vd = this.destro.calcola();
                         return vs / vd;
             }
         }
   }
   function foo () {
      var e = new Espressione("+",
                 new Espressione("-",
                  new Espressione("+",
                   new Espressione("*",
                    new Espressione(2, null, null),
                    new Espressione(3, null, null)),
                   new Espressione(5, null, null)),
                  new Espressione("/",
                   new Espressione(6, null, null),
                   new Espressione(2, null, null))),
                 new Espressione(1, null, null));
      print(e.visualizza());
      print(e.calcola());
   }
```

2. L'*albero genealogico* è una rappresentazione (parziale) che mostra i rapporti familiari tra gli antenati di un individuo. Abitualmente un albero genealogico è realizzato utilizzando delle caselle, quadrate per i maschi e circolari per le femmine, contenenti i nomi di ciascuna persona, spesso corredate di informazioni aggiuntive, quali luogo e data di nascita e morte, occupazione o professione. Questi simboli, disposti dall'alto verso il basso in ordine cronologico, sono connessi da vari tipi di linee che rappresentano matrimoni, unioni extra coniugali, discendenza [Wikipedia, alla voce *Albero genealogic*o].

   Un *albero genealogico patrilineare* è un particolare tipo di albero genealogico in cui sono rappresentati tutti e soli i discendenti per via paterna.

   Definire in JavaScript un costruttore personalizzato per oggetti che rappresentano alberi genealogici patrilineari.

   Il costruttore ha un parametro: *persona*.

   L'oggetto ha i seguenti metodi:

   - *aggiungiConiuge* (*persona*, *coniuge*): aggiunge, nell'albero genealogico, il *coniuge* a una *persona*;
   - *aggiungiDiscendente* (*persona*, *discendente*): aggiunge, nell'albero genealogico, un *discendente* diretto a una *persona*;
   - *visualizzaPersona* (*persona*): restituisce una stringa contenente le informazioni relative a una *persona*;
   - *grado*(*persona*): calcola il grado di parentela tra una *persona* e il suo antenato più anziano nell'albero genealogico;
   - *linea* (*persona*): restituisce una stringa contenente la linea di discendenza tra una *persona* e il suo antenato più anziano nell'albero genealogico.

   Definire una funzione di prova che

   - crea un albero genealogico con queste caratteristiche: *Bruno*, *Carlo* e *Daniela* sono figli di *Aldo*, *Enzo* e *Fabio* sono figli di *Bruno*, *Giacomo* è figlio di *Carlo*, *Alessia* è la moglie di *Aldo*, *Beatrice* è la moglie di *Bruno*, *Cecilia* è la moglie di *Carlo*;
   - visualizza le informazioni di *Aldo* ed *Enzo*;
   - calcola il grado di parentela di *Bruno*, *Giacomo* e *Fabio*;
   - visualizza la linea di discendenza di *Enzo* e *Giacomo*.

```
function AlberoGenealogico (persona) {
   this.nome = persona;
   this.coniuge = "";
```

```
      this.figli = [];
      this.aggiungiConiuge =
         function (p, c) {
            if (p == this.nome) {
               this.coniuge = c;
            } else {
               for (var i in this.figli) {
                  this.figli[i].aggiungiConiuge(p, c);
               }
            }
         }
      this.aggiungiDiscendente =
         function (p, d) {
            if (p == this.nome) {
               var n = new AlberoGenealogico(d);
               this.figli.push(n);
            } else {
               for (var i in this.figli) {
                 this.figli[i].aggiungiDiscendente(p, d);
               }
            }
         }
      this.visualizzaPersona =
         function (p) {
            if (p == this.nome) {
               return(this.nome +
                      " (" + this.coniuge + ")");
            } else {
               var s = ""
               for (var i in this.figli) {
                 s += this.figli[i].visualizzaPersona(p);
               }
            return s;
            }
         }
      this.grado =
         function (p) {
            if (p == this.nome) {
               return 1;
            } else {
               var d = 0;
               for (var i in this.figli) {
                  var df = this.figli[i].grado(p);
                  if (df > d) {
                     d = df;
                  }
               }
```

```
            if (d > 0) {
               return d + 1;
            } else {
               return 0;
            }
          }
        }
    this.linea =
       function (p) {
          if (p == this.nome) {
             return this.nome;
          }
          var s = "";
          for (var i in this.figli) {
             s += this.figli[i].linea(p)
          }
          if (s != "") {
             return this.nome + " - " + s;
          } else {
             return s;
          }
       }
}

function foo () {
   var ag = new AlberoGenealogico("Aldo");
   ag.aggiungiConiuge("Aldo", "Alessia");
   ag.aggiungiDiscendente("Aldo", "Bruno");
   ag.aggiungiConiuge("Bruno", "Beatrice");
   ag.aggiungiDiscendente("Aldo", "Carlo");
   ag.aggiungiConiuge("Carlo", "Cecilia");
   ag.aggiungiDiscendente("Aldo", "Daniela");
   ag.aggiungiDiscendente("Bruno", "Enzo");
   ag.aggiungiDiscendente("Bruno", "Fabio");
   ag.aggiungiDiscendente("Carlo", "Giacomo");
   print(ag.visualizzaPersona("Aldo"));
   print(ag.visualizzaPersona("Enzo"));
   print("Bruno : grado " + ag.grado("Bruno"));
   print("Giacomo : grado " + ag.grado("Giacomo"));
   print("Fabio : grado " + ag.grado("Fabio"));
   print(ag.linea("Enzo"));
   print(ag.linea("Giacomo"));
}
```

18.4 Esercizi del capitolo 14

1. Definire una funzione JavaScript che visita un albero DOM e che restituisce il numero dei suoi nodi di tipo testo. Invocare la funzione usando come argomento la radice dell'albero DOM corrispondente alla pagina web dell'ambiente *EasyJS*.

```
function contaNodiTesto (nodo) {
   switch (nodo.nodeType) {
      case 3 :
         return 1;
      case 1 :
      case 9 :
         var c = 0;
         for (var i = 0; i < nodo.childNodes.length; i++) {
            c += contaNodiTesto(nodo.childNodes[i]);
         }
         return c;
      default :
         return 0;
   }
}
print(contaNodiTesto(document));
```

18.5 Esercizi del capitolo 15

1. Definire un documento HTML relativo a una sveglia. Il documento contiene: un pulsante per impostare l'ora corrente e due campi di testo per l'inserimento dell'ora e dei minuti, un pulsante per impostare l'allarme e due campi di testo per l'inserimento dell'ora e dei minuti, un pulsante per fare avanzare di uno i minuti dell'ora corrente e un campo di sola lettura che segnala l'allarme. Il valore dell'ora corrente e quello dell'allarme sono visualizzati in due campi di sola lettura.

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8">
 <script type="text/javascript"
         src="Sveglia.js">
 </script>
 <title>Sveglia</title>
</head>
<body>
 <form>
  <input type="text"
         id="oreCorrente"/> Ore
```

```
    <input type="text"
           id="minutiCorrente"/> Minuti
    <br/>
    <input type="button"
           id="impostaOraCorrente"
           value="Imposta ora corrente"/>
    <br/>
    <input type="text"
           id="orarioCorrente"
           readonly="readonly"/>
    <br/>
    <input type="text"
           id="oreAllarme"/> Ore
    <input type="text"
           id="minutiAllarme"/> Minuti
    <br/>
    <input type="button"
           id="impostaAllarme"
           value="Imposta allarme"/>
    <br/>
    <input type="text"
           id="orarioAllarme"
           readonly="readonly"/>
    <br/>
    <input type="button"
           id="avanzaUnMinuto"
           value="Avanza un minuto"/>
    <br/>
    <input type="text"
           id="messaggioAllarme"
           readonly="readonly"/>
  </form>
 </body>
 </html>
```

```
 function Sveglia {…}

 function gestoreImpostaOraCorrente () {
    try {
       var nodo1 = document.getElementById("oreCorrente");
       var ore = eval(nodo1.value);
       var nodo2 = document.getElementById("minutiCorrente");
       var minuti = eval(nodo2.value);
       laSveglia.oraCorrente(ore, minuti);
       var nodo3 = document.getElementById("orarioCorrente");
```

```
        nodo3.value = ore + ":" + minuti;
     } catch ( e ) {
        alert("gestoreImpostaOraCorrente " + e);
     }
  }
  function gestoreImpostaAllarme () {
     try {
        var nodo1 = document.getElementById("oreAllarme");
        var ore = eval(nodo1.value);
        var nodo2 = document.getElementById("minutiAllarme");
        var minuti = eval(nodo2.value);
        laSveglia.allarme(ore, minuti);
        var nodo3 = document.getElementById("orarioAllarme");
        nodo3.value = ore + ":" + minuti;
     } catch ( e ) {
        alert("gestoreImpostaAllarme " + e);
     }
  }
  function gestoreAvanzaUnMinuto () {
     try {
        var b = laSveglia.tic();
        var nodo1 = document.getElementById("orarioCorrente");
        nodo1.value = laSveglia.ore + ":" + laSveglia.minuti;
        var nodo2 = document.getElementById("messaggioAllarme");
        if (b) {
           nodo2.value = "sveglia!";
        } else {
           nodo2.value = "";
        }
     } catch ( e ) {
        alert("gestoreAvanzaUnMinuto " + e);
     }
  }
  var laSveglia;
  function inizializza () {
     try {
        var nodo1 = document.getElementById("impostaOraCorrente");
        nodo1.onclick = gestoreImpostaOraCorrente;
        var nodo2 = document.getElementById("impostaAllarme");
        nodo2.onclick = gestoreImpostaAllarme;
        var nodo3 = document.getElementById("avanzaUnMinuto");
        nodo3.onclick = gestoreAvanzaUnMinuto;
        laSveglia = new Sveglia();
     } catch ( e ) {
        alert("inizializza " + e);
     }
```

```
}
window.onload = inizializza;
```

2. Definire un documento HTML relativo a un distributore automatico di caffè. Il documento contiene: un pulsante che aggiunge un numero di capsule inserito in un campo di testo, un pulsante che eroga un numero di caffè inserito in un campo di testo addebitando il numero di caffè erogati a un codice inserito in un campo di testo, un pulsante che visualizza in un campo di testo di sola lettura il numero di caffè addebitati a un codice inserito in un campo di testo.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<script type="text/javascript"
        src="Distributore.js">
</script>
<title>Distributore automatico</title>
</head>
<body>
  <form>
    <input type="text"
           id="numeroCapsule"/> Numero capsule
    <br/>
    <input type="button"
           id="caricaCapsule"
           value="Carica capsule"/>
    <br/>
    <input type="text"
           id="numeroCaffe"/> Numero caff&egrave;
    <input type="text"
           id="codiceAddebito"/> Codice
    <br/>
    <input type="button"
           id="eroga"
           value="Eroga" />
    </br>
    <input type="text"
           id="codiceRapporto"/> Codice
    <br/>
    <input type="button"
           id="rapporto"
           value="Rapporto"/>
    <br/>
    <input type="text"
```

```
            id="messaggioRapporto"
            readonly="readonly"/>
    </form>
  </body>
  </html>
```

```
  function Distributore () { … }

  function gestoreCaricaCapsule () {
     try {
        var nodo1 = document.getElementById("numeroCapsule");
        var numeroCapsule = eval(nodo1.value);
        ilDistributore.carica(numeroCapsule);
      } catch ( e ) {
        alert("gestoreCaricaCapsule " + e);
      }
  }
  function gestoreEroga () {
     try {
        var nodo1 = document.getElementById("numeroCaffe");
        var numeroCaffe = eval(nodo1.value);
        var nodo2 = document.getElementById("codiceAddebito");
        var codiceAddebito = nodo2.value;
        ilDistributore.eroga(numeroCaffe, codiceAddebito);
     } catch ( e ) {
        alert("gestoreEroga " + e);
     }
  }
  function gestoreRapporto () {
     try {
        var nodo1 = document.getElementById("codiceRapporto");
        var codiceRapporto = nodo1.value;
        var nodo2 = document.getElementById("messaggioRapporto");
        nodo2.value = ilDistributore.rapporto(codiceRapporto);
     } catch ( e ) {
        alert("gestoreRapporto " + e);
     }
  }
  var ilDistributore;
  function inizializza () {
     try {
        var nodo1 = document.getElementById("caricaCapsule");
        nodo1.onclick = gestoreCaricaCapsule;
        var nodo2 = document.getElementById("eroga");
        nodo2.onclick = gestoreEroga;
```

```
        var nodo3 = document.getElementById("rapporto");
        nodo3.onclick = gestoreRapporto;
        ilDistributore = new Distributore();
     } catch ( e ) {
        alert("inizializza " + e);
     }
  }
  window.onload = inizializza;
```

18.6 Esercizi del capitolo 16

1. Definire un documento HTML che visualizza dinamicamente una programmazione cinematografica definita in un documento XML. La programmazione contiene una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione.

```
<?xml version="1.0"?>
<programmazione>
 <film titolo="Django Unchained">
  <cinema>Apollo</cinema>
  <cinema>Nuovo</cinema>
  <cinema>Centrale</cinema>
 </film>
 <film titolo="Flight">
  <cinema>Diana</cinema>
  <cinema>Nuovo</cinema>
  <cinema>Centrale</cinema>
 </film>
 <film titolo="Argo">
  <cinema>Majestic</cinema>
  <cinema>Lanteri</cinema>
 </film>
</programmazione>
```

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="UTF-8">
 <script type="text/javascript"
         src="programmazioneFilm.js">
 </script>
 <title>Programmazione film</title>
 </head>
 <body>
 <ol id="lista"></ol>
```

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

```
function caricaXML (nomeFile) { … }

function Programmazione () {
   this.lista = [];
   this.inizializza =
      function (radice) {
         var nodi = radice.getElementsByTagName("film");
         for (var i = 0; i < nodi.length; i++) {
            var film = new Film();
            film.inizializza(nodi[i]);
            this.lista.push(film);
         }
      }
   this.creaListaHTML =
      function () {
         var s = "";
         for (i = 0; i < this.lista.length; i++) {
            s += this.lista[i].creaHTML();
         }
         return s;
      }
}
function Film () {
   this.titolo;
   this.lista = [];
   this.inizializza =
      function (nodo) {
         this.titolo = nodo.getAttribute("titolo");
         var nodi = nodo.getElementsByTagName("cinema");
         for (var i = 0; i < nodi.length; i++) {
            var cinema = nodi[i].firstChild.nodeValue;
            this.lista.push(cinema);
         }
     }
   this.creaHTML =
      function () {
         var s = "<li><b>" + this.titolo + "</b></li>";
         s += "<ul>"
         for (var i = 0; i < this.lista.length; i++) {
            s += "<li>" + this.lista[i] + "</li>";
         }
         s += "</ul>";
```

```
            return s;
        }
    }
    function inizializza () {
       try {
          var radice = caricaXML("programmazioneFilm.xml");
          laProgrammazione = new Programmazione();
          laProgrammazione.inizializza(radice);
          var nodoLista = document.getElementById("lista");
          nodoLista.innerHTML = laProgrammazione.creaListaHTML();
       } catch ( e ) {
          alert("inizializza " + e);
       }
    }
    window.onload = inizializza;
```

2. Definire un documento HTML relativo alla programmazione cinematografica definita in un documento XML. La programmazione contiene una lista di film, a ognuno dei quali è associata una lista di cinema presso i quali il film è in programmazione. Il documento contiene un menu a tendina, generato dinamicamente, contenente tutti e soli i cinema presenti nella programmazione, un pulsante che visualizza dinamicamente la lista dei film in programmazione presso il cinema selezionato.

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="UTF-8">
 <script type="text/javascript"
         src="programmazioneFilm.js">
 </script>
 <title>Programmazione film</title>
 </head>
 <body>
  <form>
   <select id="selectCinema">
   <input type="button" value="Mostra film" id="mostraFilm"/>
   <br/>
   </select>
   <ol id="lista"></ol>
  </form>
 </body>
</html>
```

```
function caricaXML (nomeFile) { … }

function Programmazione () {
   this.lista = [];
   this.inizializza =
      function (radice) { … }
   this.creaSelect =
      function () {
         var listaCinema = {};
         for (var i = 0; i < this.lista.length; i++) {
            var film = this.lista[i];
            film.aggiungiCinema(listaCinema);
         }
         var s = "";
         for (var i in listaCinema) {
            s += '<option value ="' + i + '">' +
                 i +
                 '</option>';
         }
         return s;
      }
   this.creaListaHTML =
      function (cinema) {
         var s = "";
         for (i = 0; i < this.lista.length; i++) {
            var film = this.lista[i];
            if (film.inProgrammazione(cinema)) {
               s += "<li>" + film.titolo + "</li>";
            }
         }
         return s;
      }
}
function Film () {
   this.titolo;
   this.lista = [];
   this.inizializza =
      function (nodo) { … }
   this.aggiungiCinema =
      function (listaCinema) {
         for (var i = 0; i < this.lista.length; i++) {
            listaCinema[this.lista[i]] = true;
         }
      }
   this.inProgrammazione =
      function (cinema) {
         var i = 0;
```

```
            while (i < this.lista.length &&
                   this.lista[i] != cinema) {
               i++;
            }
            return (i < this.lista.length);
         }
   }
   function gestoreMostraFilm () {
      try {
         var nodoSelect = document.getElementById("selectCinema");
         var cinema = nodoSelect.value;
         var nodoL = document.getElementById("lista");
         nodoL.innerHTML = laProgrammazione.creaListaHTML(cinema);
      } catch ( e ) {
        alert("gestoreMostraFilm " + e);
      }
   }
   function inizializza () {
      try {
         var radice = caricaXML("programmazioneFilm.xml");
         laProgrammazione = new Programmazione();
         laProgrammazione.inizializza(radice);
         var nodoSelect = document.getElementById("selectCinema");
         nodoSelect.innerHTML = laProgrammazione.creaSelect();
         var nodoMF = document.getElementById("mostraFilm");
         nodoMF.onclick = gestoreMostraFilm;
      } catch ( e ) {
         alert("inizializza " + e);
      }
   }
   window.onload = inizializza;
```

3. Definire un documento HTML che visualizza dinamicamente un testo, permettendo di evidenziare selettivamente tutte le occorrenze di alcune parole. Il testo è memorizzato in un documento XML in cui le parole evidenziabili sono opportunamente marcate.

```
<?xml version="1.0"?>
<testo>
Definire in JavaScript una <m>funzione</m> che calcola e
restituisce una <m>stringa</m> che rappresenta un
<m>documento</m> XML.
La <m>funzione</m> ha come <m>argomento</m> l'<m>albero</m> DOM
creato dal parser XML a partire dal <m>documento</m> XML stesso.
La <m>funzione</m> assume che nel <m>documento</m> XML le parti
testuali siano solo all'interno di coppie di marche terminali
(che non hanno figli, cioé).
```

```
Infine, la <m>funzione</m> visualizza anche il valore degli
attributi XML, rispettandone la sintassi.
</testo>
```

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="UTF-8">
 <script type="text/javascript"
         src="ElaborazioneTesto.js">
 </script>
 <title>Elaborazione testo</title>
 </head>
 <body>
  <div id="testo"></div>
  <hr />
  <form>
   <select id="parole"></select>
  </form>
 </body>
</html>
```

```
function caricaXML (nomeFile) { … }
function cercaParole () {
   parole = {};
   var lista = radice.getElementsByTagName("m");
   for (var i = 0;  i < lista.length; i++) {
      var parola = lista[i].firstChild.nodeValue;
      parole[parola] = parola;
   }
}
function frontiera (nodo, parolaScelta) {
   if (nodo.nodeType == 1) {
      if (nodo.nodeName == "m") {
         var parola = nodo.firstChild.nodeValue;
         if (parola == parolaScelta) {
            return "<b>" + parola + "</b>";
         } else {
            return parola;
         }
      }
   var s = "";
   var nodi = nodo.childNodes;
   for (var i = 0; i < nodi.length; i++) {
```

```
            s += frontiera(nodi[i], parolaScelta);
         }
         return s;
      } else if (nodo.nodeType == 3) {
         return nodo.nodeValue;
      } else if (nodo.nodeType == 9) {
         return frontiera(nodo.childNodes[0], parolaScelta);
      } else {
         return "";
      }
   }
   function opzioniParola () {
      var s = "";
      for (var parola in parole) {
         s += "<option>" + parola + "</option>"
      }
      return s;
   }
   function gestoreParola () {
      try {
         var nodoP = document.getElementById("parole");
         var parola = nodoP.value;
         var nodoT = document.getElementById("testo");
         nodoT.innerHTML = frontiera(radice, parola);
      } catch ( e ) {
         alert("gestoreParola " + e);
      }
   }
   var parole;
   var radice;
   function inizializza () {
      try {
         radice = caricaXML("Testo.xml");
         cercaParole();
         var nodoP = document.getElementById("parole");
         nodoP.onchange = gestoreParola;
         nodoP.innerHTML = opzioniParola();
         gestoreParola();
      } catch ( e ) {
         alert("inizializza " + e);
      }
   }
   window.onload = inizializza;
```

19 Grammatica di JavaScript

La seguente grammatica non copre in modo esaustivo la sintassi di JavaScript, ma riporta le regole sintattiche introdotte nel libro. Il lettore interessato può trovare la sintassi completa in un qualsiasi manuale di JavaScript.

19.1 Parole riservate

```
false, true, null

this, new

var, function

return, break

if, else

switch, case, default

while, for, in

try, catch, finally, throw
```

19.2 Caratteri

```
<Lettera>  ::= a | b | c | d | e | f | g | h | i
             | j | k | l | m | n | o | p | q | r
             | s | t | u | v | w | x | y | z
             | A | B | C | D | E | F | G | H | I
             | J | K | L | M | N | O | P | Q | R
             | S | T | U | V | W | X | Y | Z

<Cifra>    ::= 0
             | <CifraNz>

<CifraNz>  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Speciale> ::= Space[18]
             | ! | " | # | $ | % | & | ' | ( | )
             | * | + | , | - | . | / | : | ; | <
             | = | > | ? | @ | [ | \ | ] | ^ | _
             | ` | { | | | } | ~
```

18 Carattere di spaziatura.

19.3 Identificatore

```
<Identificatore> ::= <CarIniziale>
                   |  <CarIniziale> <Caratteri>

<CarIniziale>    ::= <Lettera>
                   |  _
                   |  $

<Caratteri>      ::= <CarNonIniziale>
                   |  <CarNonIniziale> <Caratteri>

<CarNonIniziale> ::= <Lettera>
                   |  <Cifra>
                   |  _
                   |  $
```

19.4 Costante

```
<Costante>       ::= <Numero>
                   |  <Booleano>
                   |  <Stringa>
                   |  null

<Numero>         ::= <Intero>
                   |  <Intero>.<Cifre>
                   |  <Intero>E<Esponente>
                   |  <Intero>.<Cifre>E<Esponente>

<Intero>         ::= <Cifra>
                   |  <CifraNZ> <Cifre>

<Cifre>          ::= <Cifra>
                   |  <Cifra> <Cifre>

<Esponente>      ::= <Intero>
                   |  + <Intero>
                   |  - <Intero>

<Booleano>       ::= true
                   |  false

<Stringa>        ::= ""
                   |  "<CaratteriStr>"
                   |  ''
                   |  '<CaratteriStr>'

<CaratteriStr>   ::= <CarattereStr>
                   |  <CarattereStr><CaratteriStr>

<CarattereStr>   ::= <Lettera>
                   |  <Cifra>
                   |  <Speciale>
```

19.5 Espressione

```
<Espressione> ::= <Costante>
                |  <Identificatore>
                |  (<Espressione>)
                |  <UnOp> <Espressione>
                |  <Espressione> <BinOp> <Espressione>
                |  this
                |  <Espressione>.<Identificatore>
                |  <Espressione>.<Chiamata>
                |  <Espressione>[<Espressione>]
                |  <Chiamata>
                |  new <Identificatore>()
                |  new <Identificatore>(<Espressioni>)
                |  []
                |  [<Espressioni>]
                |  {}
                |  {<Coppie>}

<UnOp>         ::= -  | +  | !
<BinOP>        ::= -  | +  | *  | /  | %
                |  && | || |
                |  <  | <= | >  | >= | == | !=

<Espressioni> ::= <Espressione>
                |  <Espressione>, <Espressioni>

<Chiamata>     ::= <Identificatore>()
                |  <Identificatore>(<Espressioni>)

<Coppie>       ::= <Coppia>
                |  <Coppia>, <Coppie>
<Coppia>       ::= <Identificatore> : <Espressione>
                |  <Costante> : <Espressione>
```

19.6 Programma, dichiarazione, comando, blocco

```
<Programma>        ::= <Comandi>

<Comandi>          ::= <Comando>
                     |  <Comando> <Comandi>

<Comando>          ::= <Dichiarazione>
                     |  <ComandoSemplice>
                     |  <ComandoComposto>

<Dichiarazione>    ::= <DicVar>
                     |  <DicFun>
                     |  <DicCostr>

<ComandoSemplice>  ::= <Assegnamento>
                     |  <Invocazione>
                     |  <Return>
                     |  <Break>
                     |  <Throw>

<ComandoComposto>  ::= <Blocco>
                     |  <If>
                     |  <Switch>
                     |  <For>
                     |  <While>
                     |  <Try>

<Blocco>           ::= {<Comandi>}
```

19.7 Dichiarazione

```
<DicVar>       ::= var <Identificatore>;
                |  var <Identificatore> = <Espressione>;

<DicFun>       ::= function <Identificatore>()
                      <Blocco>
                |  function <Identificatore>(<Parametri>)
                      <Blocco>

<Parametri>    ::= <Identificatore>
                |  <Identificatore>, <Parametri>

<DicCostr>     ::= function <Identificatore>()
                      {<BloccoCostr>}
                |  function <Identificatore>(<Parametri>)
                      {<BloccoCostr>}

<BloccoCostr>  ::= <PropMet>
                |  <PropMet> <BloccoCostr>

<PropMet>      ::= this.<Identificatore> = <Espressione>;
                |  this.<Identificatore> = <FunLet>

<FunLet>       ::= function ()
                      <Blocco>
                |  function (<Parametri>)
                      <Blocco>
```

19.8 Comando semplice

```
<Assegnabile>  ::= <Identificatore>
                 |  <Assegnabile>[<Espressione>]
                 |  this.<Assegnabile>
                 |  <Identificatore>.<Assegnabile>

<Assegnamento> ::= <Assegnabile> = <Aspressione>;
                 |  <Assegnabile> += <Aspressione>;
                 |  <Assegnabile> -= <Espressione>;
                 |  <Assegnabile> *= <Espressione>;
                 |  <Assegnabile> /= <Espressione>;
                 |  <Assegnabile> %= <Espressione>;
                 |  <Assegnabile>++;
                 |  <Assegnabile>--;

<Invocazione>  ::= <Assegnabile> ();
                 |  <Assegnabile> (<Espressioni>);

<Return>       ::= return <Espressione>;

<Break>        ::= break;

<Throw>        ::= throw <Espressione>;
```

19.9 Comando composto

```
<If>          ::= if (<Espressione>)
                  <Blocco>
                | if (<Espressione>)
                  <Blocco>
                  else <Blocco>

<Alternativa> ::= case <Costante>: <Comandi>

<Alternative> ::= <Alternativa>
                | <Alternativa> <Alternative>

<Switch>      ::= switch (<Espressione>)
                  {<Alternative>}
                | switch (<Espressione>)
                  {<Alternative> default: <Comandi>}

<For>         ::= for (<Comando>; <Espressione>; <Comando>)
                  <Blocco>
                | for (var <Identificatore> in <Espressione>)
                  <Blocco>

<While>       ::= while (<Espressione>)
                  <Blocco>
<Try>         ::= try {<Blocco>}
                  catch (<Identificatore>) {<Blocco>}
                | try {<Blocco>}
                  catch (<Identificatore>) {<Blocco>}
                  finally {<Blocco>}
```

Indice analitico