Proprietà o metodo	Descrizione
protocol	Protocollo in uso (HTTP o HTTPS).
host	Nome che identifica lo *host*.
pathname	Percorso della pagina attuale.
search	Contiene la *query-string*, ovvero l'eventuale parte di URL che segue il carattere «?» e specifica i valori di alcune proprietà.
reload( )	Ricarica la pagina.
replace(url)	Sostituisce la pagina attuale con quella riferita dall'URL specificato.

Assegnare un nuovo valore alla proprietà *href* comporta il caricamento della nuova pagina al posto di quella attuale. Se da tale pagina l'utente clicca sul pulsante *back* del browser viene mostrata la pagina originale; se invece si usa il metodo *replace* l'URL viene usato per caricare un nuovo documento e il nuovo indirizzo sovrascrive il precedente. In questo modo il pulsante *back* del browser non recupererà la pagina originale, ma, se esiste, visualizzerà quella ancora precedente.

ESEMPIO

Supponendo esista una pagina chiamata *test2.htm*, il codice seguente fa sì che, se si preme sul primo pulsante, la pagina *test2.htm* venga visualizzata ed è possibile ritornare alla pagina di partenza con il tasto *back* del browser. Se invece si preme sul secondo pulsante, *test2.htm* prende il posto della pagina corrente ma non vi è traccia della sua visualizzazione, tanto che il pulsante *back* risulta disabilitato.

```
<!doctype html>
<html>
  <body>
    <input type="button" value="Nuova"
      onclick="location.href='test2.htm'">
    <input type="button" value="Sovrascrivi"
      onclick="location.replace('test2.htm')">
  </body>
</html>
```

1.4 L'oggetto *history*

Per un'applicazione JavaScript è possibile navigare la storia delle pagine visitate precedentemente rispetto alla pagina dove viene eseguito lo *script*. Questo avviene grazie all'oggetto ***history*** e ai metodi che espone.

I metodi principali dell'oggetto *history* sono mostrati nella tabella:

Metodo	Descrizione
back( )	Simula il comportamento del tasto *back* del browser: visualizza la pagina precedente della cronologia.
forward( )	Simula il comportamento del tasto *forward* del browser: visualizza la pagina successiva della cronologia.
go( )	Visualizza, relativamente alla posizione 0 della pagina corrente, la pagina specificata: se il valore fornito come argomento è negativo visualizza la pagina che precede nella cronologia di un numero di posizioni pari al valore specificato; se il valore fornito come argomento è positivo visualizza la pagina che segue nella cronologia di un numero di posizioni pari al valore specificato.

Usando l'oggetto *history* è possibile creare applicazioni per navigare in modo relativo le pagine visitate.

Proprietà o metodo	Descrizione
protocol	Protocollo in uso (HTTP o HTTPS).
host	Nome che identifica lo *host*.
pathname	Percorso della pagina attuale.
search	Contiene la *query-string*, ovvero l'eventuale parte di URL che segue il carattere «?» e specifica i valori di alcune proprietà.
reload( )	Ricarica la pagina.
replace(url)	Sostituisce la pagina attuale con quella riferita dall'URL specificato.

Assegnare un nuovo valore alla proprietà *href* comporta il caricamento della nuova pagina al posto di quella attuale. Se da tale pagina l'utente clicca sul pulsante *back* del browser viene mostrata la pagina originale; se invece si usa il metodo *replace* l'URL viene usato per caricare un nuovo documento e il nuovo indirizzo sovrascrive il precedente. In questo modo il pulsante *back* del browser non recupererà la pagina originale, ma, se esiste, visualizzerà quella ancora precedente.

ESEMPIO

Supponendo esista una pagina chiamata *test2.htm*, il codice seguente fa sì che, se si preme sul primo pulsante, la pagina *test2.htm* venga visualizzata ed è possibile ritornare alla pagina di partenza con il tasto *back* del browser. Se invece si preme sul secondo pulsante, *test2.htm* prende il posto della pagina corrente ma non vi è traccia della sua visualizzazione, tanto che il pulsante *back* risulta disabilitato.

```
<!doctype html>
<html>
  <body>
    <input type="button" value="Nuova"
      onclick="location.href='test2.htm'">
    <input type="button" value="Sovrascrivi"
      onclick="location.replace('test2.htm')">
  </body>
</html>
```

1.4 L'oggetto *history*

Per un'applicazione JavaScript è possibile navigare la storia delle pagine visitate precedentemente rispetto alla pagina dove viene eseguito lo *script*. Questo avviene grazie all'oggetto ***history*** e ai metodi che espone.

I metodi principali dell'oggetto *history* sono mostrati nella tabella:

Metodo	Descrizione
back( )	Simula il comportamento del tasto *back* del browser: visualizza la pagina precedente della cronologia.
forward( )	Simula il comportamento del tasto *forward* del browser: visualizza la pagina successiva della cronologia.
go( )	Visualizza, relativamente alla posizione 0 della pagina corrente, la pagina specificata: se il valore fornito come argomento è negativo visualizza la pagina che precede nella cronologia di un numero di posizioni pari al valore specificato; se il valore fornito come argomento è positivo visualizza la pagina che segue nella cronologia di un numero di posizioni pari al valore specificato.

Usando l'oggetto *history* è possibile creare applicazioni per navigare in modo relativo le pagine visitate.

ESEMPIO Il codice illustra come scorrere le pagine visitate avanti o indietro premendo su uno dei primi due pulsanti; per «saltare» alla pagina visitata in una posizione specificata da un campo di testo è possibile premere sul terzo pulsante.

```
<!doctype html>
<html>
  <body>
    <input type="button" value="vai indietro"
      onclick="history.back()">
    <input type="button" value="vai avanti"
      onclick="history.forward()">
    <br>Pagina
    <input type="text" id="num" name="num">
    <input type="button" value="vai"
      onclick="history.go(document.getElementById('num').value)">
  </body>
</html>
```

OSSERVAZIONE I metodi descritti permettono la navigazione basandosi sulla storia delle pagine visitate, ma nessuno permette di recuperarne il contenuto o l'indirizzo: questa è una scelta progettuale precisa per evitare che un'applicazione web possa conoscere gli URL dei siti web visitati prima di accedere a essa.
Si può infatti facilmente immaginare cosa accadrebbe se, visitando per esempio un sito commerciale, esso potesse aver accesso alle pagine già visitate, che potrebbero essere quelle della concorrenza, ma anche quelle della propria banca o di un servizio riservato. L'esposizione di questi dettagli comporterebbe un notevole rischio per la privacy che invece, almeno da questo punto di vista, è garantita.

1.5 L'oggetto *screen*

L'oggetto ***screen*** fornisce informazioni sullo schermo su cui viene visualizzato il browser e sulle sue caratteristiche in termini di risoluzione video e di colori supportati.

In particolare, le proprietà *width* ed *height* si riferiscono, rispettivamente, alla risoluzione orizzontale e verticale dello schermo. In FIGURA 1 viene illustrato il risultato dell'esecuzione della funzione *mostra*, introdotta in un esempio precedente, applicata all'oggetto *screen*.

FIGURA 1

1.6 Collezioni di oggetti dell'oggetto *document*

Il browser espone, come proprietà dell'oggetto ***document***, collezioni di oggetti grazie alle quali è possibile accedere a determinati tipi di elementi di una pagina HTML; in particolare espone le collezioni *anchors* e *link* che contengono, rispettivamente, tutte le ancore e i *link* della pagina; *forms* che contiene tutti i *form* e *images* per le immagini.

Ciascuna collezione è accessibile sia in base al nome dell'elemento, se presente, sia per posizione: gli indici partono da 0 per il primo elemento di quel tipo nel documento, 1 per il secondo e così via.

ESEMPIO A partire da una pagina web, è possibile utilizzare il seguente *script* per generare, in fondo alla pagina stessa, una sequenza di tutte le immagini superiori ai 50×50 pixel (questo è un limite arbitrario: è stato scelto allo scopo di evitare di includere icone o altri elementi grafici d'impaginazione) presenti nel documento. Per comodità le immagini sono visualizzate con una grandezza massima (in orizzontale e in verticale) che è indicata nel parametro della funzione.

```
<!doctype html>
<html>
  <head>
    <script>
      function aggiungiImmagini(dimensione) {
        var len = document.images.length;
        for (i=0; i<len; i++) {
          var attuale = document.images[i];
          var max = attuale.width; // massimo valore delle dimensioni
          if (max<attuale.height) {
            max = attuale.height;
          }
          if (max>50) { // non è un'icona
            var img = document.createElement("img");
            img.setAttribute("src", attuale.src);
            img.setAttribute("width", attuale.width*dimensione/max);
            img.setAttribute("height", attuale.height*dimensione/max);
            // aggiunge l'immagine in fondo al documento
            document.body.appendChild(img);
          }
        }
      }
    </script>
  </head>
  <body onload="aggiungiImmagini(100)">
    <!-- contenuto -->
  </body>
</html>
```

Invece che per inserire tutte le immagini del documento, lo *script* può essere, per esempio, modificato per creare un sommario inserendo altri elementi della pagina (titoli dei vari livelli, *link*, ...).
La FIGURA 2 illustra lo *script* applicato alla voce «Italia» di Wikipedia (per generarlo è stato copiato il contenuto della pagina nel *body* dell'esempio precedente e sono stati modificati i riferimenti alle immagini affinché fossero assoluti e non relativi, riferiti al dominio **https://it.wikipedia.org**).

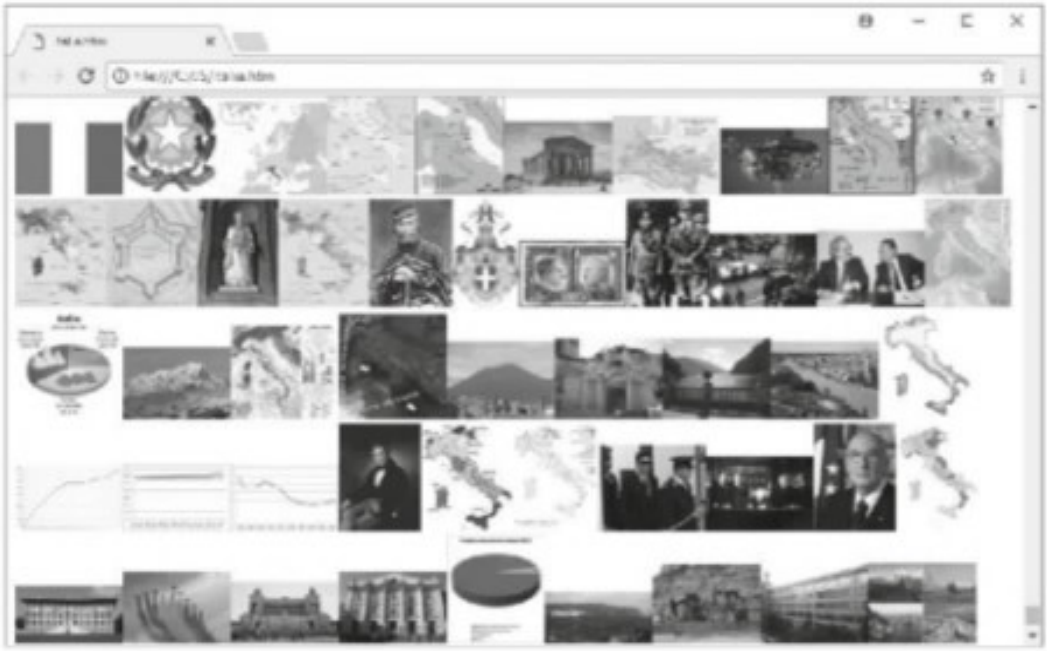


FIGURA 2

1.7 Accedere agli elementi di un *form*

Un ***form*** può contenere vari elementi come aree di testo, caselle di testo, liste di scelta (singola o multipla), caselle di selezione (a selezione singola, *radio button*, o a selezione multipla, *check-box*): ciascuno di essi espone proprietà per leggere e impostare i valori inseriti e/o selezionati.

Per leggere o scrivere il valore di un campo di tipo *text* o *hidden* si può far riferimento alla sua proprietà *value*.

ESEMPIO

Nel codice seguente, quando l'utente fa click sul pulsante, il contenuto del campo nascosto viene copiato in quello visibile; a titolo di esempio il campo nascosto viene acceduto usando il suo nome (*nascosto*) come proprietà del *form* recuperato specificando il suo nome come indice per l'insieme *forms*:

```
document.forms['nomeForm'].nascosto
```

L'altro elemento viene recuperato in maniera simile, ma accedendo alla posizione del *form* (che è la prima, cioè quella con indice 0):

```
document.forms[0].visibile.value
```

```
<!doctype html>
<html>
  <head>
    <script>
      function porta() {
        var val = document.forms['nomeForm'].nascosto.value;
        document.forms[0].visibile.value = val;
        return true;
      }
    </script>
  </head>
  <body>
    <form name="nomeForm" action="post">
```

```
        <input type="text" name="visibile" value="">
        <input type="hidden" name="nascosto" value="Una prova">
        <input type="button" value="Porta" onclick="porta()">
      </form>
    </body>
  </html>
```

Nel caso di liste, realizzate mediante il *tag* **<select>**, le opzioni possono essere un numero qualsiasi; se è ammessa la selezione multipla, anche gli elementi selezionati possono essere più di uno. A livello di DOM, una volta acceduto l'elemento **<select>** (mediante il suo *id* o il suo nome), ci si può riferire alle sue opzioni utilizzando l'*array options*. Ciascun elemento dell'*array* ha una proprietà, *selected*, che permette di sapere se l'elemento è stato selezionato o meno.

ESEMPIO

L'esempio seguente mostra un *form* con un campo di inserimento testo e due pulsanti. Facendo click sul primo pulsante il contenuto del campo viene aggiunto a una lista; premendo sul secondo pulsante vengono mostrati i valori selezionati:

```
<!doctype html>
<html>
  <head>
    <script>
      function aggiungi() {
        var val = document.forms['nomeForm'].visibile.value;
        var opt = document.forms['nomeForm'].lista.options;
        opt.length = opt.length + 1;
        opt[opt.length - 1] = new Option(val, val);
      }
      function mostra() {
        var val = "";
        var opt = document.forms['nomeForm'].lista.options;
        for (var i=0; i<opt.length; i++)
          if (opt[i].selected)
            val = val + opt[i].value + "; ";
          if (val == "")
            alert("Nessun elemento selezionato");
          else
            alert("Valori selezionati: " + val);
      }
    </script>
  </head>
  <body>
    <form name="nomeForm" action="post">
      <input type="text" name="visibile" value="">
      <input type="button" value="Aggiungi" onclick="aggiungi()">
      <input type="button" value="Mostra" onclick="mostra()">
        <select name="lista" multiple="multiple">
         <option value="">Selezionare voci...</option>
      </select>
    </form>
  </body>
</html>
```

Option è un costruttore per l'oggetto che rappresenta un'opzione e riceve due parametri: il valore e la descrizione dell'opzione da creare.

Anche le *checklist* sono gestite mediante un vettore di elementi: la proprietà che indica se gli elementi sono selezionati o meno è *checked*; tale proprietà è accessibile anche in scrittura: se le si assegna `true`, l'elemento viene selezionato, se `false` viene deselezionato.

ESEMPIO Nello *script* quando l'utente fa click sul primo pulsante vengono letti e mostrati i valori selezionati della *checklist*; altri due pulsanti permettono, rispettivamente, di selezionare e deselezionare tutti gli elementi:

```
<!doctype html>
<html>
  <head>
    <script>
      function assegna(cosa) {
        var ck = document.forms['nomeForm'].test;
        for (var i=0; i<ck.length; i++)
          ck[i].checked = cosa;
      }

      function selezionati() {
        var val = "";
        var opt = document.forms['nomeForm'].test;
        for (var i=0; i<opt.length; i++)
          if (opt[i].checked)
            val = val + opt[i].value + "; ";
        if (val == "")
          alert("Nessun elemento selezionato");
        else
          alert("Valori selezionati: " + val);
      }
    </script>
  </head>
  <body>
    <form name="nomeForm" action="post">
      <input type="checkbox" name="test" value="1">Prima
      <input type="checkbox" name="test" value="2">Seconda
      <input type="button" value="Mostra" onclick="selezionati()">
      <input type="button" value="Seleziona" onclick="assegna(true)">
      <input type="button" value="Deseleziona"
          onclick="assegna(false)">
    </form>
  </body>
</html>
```

ESEMPIO I *radio button* permettono la scelta di un'unica opzione; nello *script* se l'utente preme sul pulsante, viene mostrato il valore dell'elemento eventualmente selezionato (FIGURA 3):

```
<!doctype html>
<html>
  <head>
    <script>
      function selezionati() {
        var val = "";
        var opt = document.forms['nomeForm'].test;
        for (i=0; i<opt.length; i++)
          if (opt[i].checked)
```

```
            val = val + opt[i].value + "; ";
        if (val==="")
          alert("Nessun elemento selezionato");
        else
          alert("Valori selezionati: " + val);
      }
    </script>
  </head>
  <body>
    <form name="nomeForm" action="post">
      <input type="radio" name="test" value="1">Prima
      <input type="radio" name="test" value="2">Seconda
      <input type="button" value="Mostra" onclick="selezionati()">
    </form>
  </body>
</html>
```

FIGURA 3

1.8 Accedere e modificare i CSS

Attraverso l'utilizzo del DOM è possibile specificare anche le proprietà normalmente gestite mediante CSS. È infatti possibile accedere alla proprietà *style* come a qualsiasi altra proprietà del DOM e, di conseguenza, si può accedere alle specifiche proprietà CSS come proprietà di *style*.

ESEMPIO Nel codice che segue si ha una pagina HTML con numerosi elementi `<div>`; da una lista di selezione è possibile impostare quale valore assegnare alla proprietà *display* di tutti gli elementi `<div>` che sono recuperati invocando il metodo *getElementsByTagName*:

```
<!doctype html>
<html>
  <head>
    <script>
      function visualizza(come) {
        if (come) {
          var divs = document.getElementsByTagName("div");
          for (var i=0; i<divs.length; i++)
            divs[i].style.display = come;
        }
      }
```

JavaScript e i CSS

Impostare la proprietà *display* al valore *none* è un modo comunemente utilizzato per nascondere parti di una pagina HTML utilizzando JavaScript. Infatti, per ragioni di efficienza, è preferibile nascondere parti del DOM piuttosto che modificarlo.

```
      </script>
    </head>
    <body>
      <select onchange="visualizza(this.value)">
        <option selected="selected">seleziona...</option>
        <option value="none">none</option>
        <option value="inline">inline</option>
        <option value="block">block</option>
        <option value="inline-block">inline-block</option>
      </select>
      <div style="width:400px">Primo</div>
      <div>Secondo</div>
      <div>Terzo</div>
      <div>Quarto</div>
      <div>Quinto</div>
    </body>
  </html>
```

La proprietà CSS *display* ha un corrispondente con lo stesso nome come proprietà JavaScript della proprietà *style*, ed è per questo motivo che nel codice dell'esempio precedente è stata acceduta come *style.display*. Questo è vero per tutte le proprietà che hanno un nome costituito da un solo termine, mentre per le proprietà che hanno nomi composti da più termini si ha una convenzione diversa: eventuali nomi di proprietà CSS contenenti uno o più simboli «-» hanno il corrispondente JavaScript senza simboli «-», ma con l'iniziale maiuscola di ciascun termine escluso il primo: questa notazione è nota come *camel notation*.

ESEMPIO

La proprietà CSS *font-weight* diventa la proprietà JavaScript *style.fontWeight*:

```
<!doctype html>
<html>
  <head>
    <script>
      function bold(cosa) {
        cosa.style.fontWeight = 'bold';
      }
    </script>
  </head>
  <body onload="bold(document.getElementsByTagName('p')[0])">
    <p>Esempio di testo</p>
    <p>Esempio di testo</p>
  </body>
</html>
```

Anche se è possibile impostare utilizzando JavaScript tutti gli attributi di stile desiderati, è buona regola non farne un uso estensivo o, meglio ancora, evitarlo del tutto. Come per la realizzazione di pagine HTML è preferibile non inserire direttamente gli stili come attributi degli elementi, ma assegnare a questi classi logiche utilizzate per assegnare lo stile mediante codice CSS, così con JavaScript è preferibile limitarsi alla manipolazione del DOM e assegnare ai vari elementi la classe di stile di loro competenza, demandando a un foglio di stile CSS la definizione dello stile.

In questo caso è utile il ricorso alla proprietà *className* dell'elemento. In pratica assegnare un nuovo valore alla proprietà *className* equivale a sostituire l'eventuale classe esistente o a crearne una nuova; per eliminare una classe è possibile assegnare la stringa vuota, mentre per aggiungerne una nuova a quelle già esistenti è sufficiente concatenare il nome della nuova classe lasciando uno spazio tra i due nomi.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <style>
      .grassetto{ font-weight:bold }
      .normale{ font-weight:normal }
    </style>
  </head>
  <body onload="document.getElementsByTagName('p')[0].className='grassetto'">
    <p class="normale">Esempio di testo</p>
    <p class="normale">Esempio di testo</p>
  </body>
</html>
```

È possibile utilizzare più nomi di classe per ciascun elemento; in questo caso i nomi vanno separati con uno spazio.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <style>
      .grassetto { font-weight:bold }
      .normale { font-weight:normal }
      .sottolineato { text-decoration:underline }
      .italico { font-style:italic }
    </style>
    <script>
        function aggiungi(cosa) {
          cosa.className = "grassetto";
          cosa.className += " sottolineato";
          cosa.className += " italico";
        }
    </script>
  </head>
  <body onload="aggiungi(document.getElementsByTagName('p')[0])">
    <p class="normale">Esempio di testo</p>
    <p class="normale">Esempio di testo</p>
  </body>
</html>
```

Se due classi definiscono la stessa proprietà, è importante l'ordine con cui esse sono definite: ha infatti precedenza la classe che è stata definita prima. Nell'esempio precedente la classe *grassetto* non viene aggiunta, ma assegnata: questo perché se venisse aggiunta alla classe normale, entrambe definirebbero la proprietà *font-weight*, con il risultato che l'aggiunta di *grassetto* non avrebbe effetto.

1.9 Prestazioni e compatibilità

Le tecniche di codifica introdotte fin qui permettono la massima riutilizzabilità delle funzioni JavaScript definite. Tuttavia rimane un problema da affrontare: la scrittura di programmi che possono essere eseguiti su browser diversi, anche perché nuove versioni dei browser possono comprendere *feature* che migliorano le prestazioni, o esporre altre caratteristiche utili. Adattare il proprio codice a ogni novità nel campo dei browser rappresenta evidentemente una soluzione poco percorribile.

Un modo per affrontare questo problema consiste nell'impiego di *framework* o librerie che, anche per i metodi base come quelli per l'accesso al DOM e per la sua gestione, forniscono un'API di un livello di astrazione più elevato e si fanno carico delle eventuali incompatibilità del livello più basso: un'API focalizzata su «cosa fare» più che su «come fare»; è infatti più realistico che siano un *framework* o una libreria e non il codice dell'applicazione a gestire le *feature* più recenti esposte dai browser. Nel seguito sarà presa in considerazione una delle librerie più diffuse per l'uso del linguaggio JavaScript nel contesto di un browser: *jQuery*.

2 La libreria *jQuery*

Browser diversi possono presentare incompatibilità a vari livelli: alcune versioni forniscono solo un supporto parziale alle specifiche del DOM, altre offrono implementazioni errate, e altre ancora forniscono modalità alternative per eseguire determinate operazioni. Scrivere codice che gestisca queste differenze richiede una conoscenza specialistica dei diversi browser, necessaria per gestire correttamente le differenze. Esistono inoltre operazioni di per sé complesse da gestire: funzionalità come il *drag & drop* di oggetti grafici o altri effetti avanzati necessitano di molto codice. Tutti questi aspetti possono essere enormemente semplificati impiegando un *framework* o una libreria per il linguaggio JavaScript che se ne facciano carico, permettendo allo sviluppatore di concentrarsi solo sul codice che realizza il comportamento desiderato dell'applicazione web.

Esistono numerosi *framework* e librerie per il linguaggio JavaScript, tra i più noti: *AngularJS*, *Bootstrap*, *jQuery*, *React*, ... Sono tutti strumenti diffusamente utilizzati, ben documentati e supportati da un'ampia comunità di sviluppatori.

jQuery è una libreria per il linguaggio JavaScript, liberamente[1] scaricabile dal sito https://jquery.com, che ha come finalità principale quella di semplificare la manipolazione e la gestione di documenti HTML, in maniera *cross* browser, con codice fortemente ottimizzato, sia in termini di prestazioni sia di dimensioni della libreria stessa.
Il nome stesso della libreria *jQuery* ricorda che una delle sue caratteristiche fondamentali è l'interrogazione condotta sugli elementi del DOM di una pagina web con una gestione il più possibile omogenea e semplificata degli stessi.

1. *jQuery* è distribuita con licenza *open-source* MIT.

Dopo aver scaricato l'ultima versione della libreria *jQuery* e dopo averla salvata nel sistema che esegue il browser, o sul server dove si trovano le pagine HTML che ne fanno uso, per utilizzarla è necessario includerla nello *header* della pagina web (spesso il nome del file della libreria comprende la versione della stessa, per esempio *jquery-3.2.1.js*):

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

Disporre della libreria in locale offre la certezza che sia disponibile la versione desiderata, ma l'utilizzo di URL assoluti, come quello che segue, è una pratica diffusa:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.js">
</script>
```

La prima parte dell'URL è standard e specifica di caricare la libreria *jQuery* da un *repository*; di seguito viene specificata la versione, che è in questo caso la 3.2.1.

Usare un URL assoluto presenta il vantaggio che, se l'utente ha già visitato una pagina web che faceva uso della stessa versione di *jQuery*, il browser l'ha probabilmente mantenuta nella propria *cache* ed evita di eseguire un nuovo download. Anche nel caso non dovesse essere presente nella *cache* del browser, è probabile che il download sia più veloce da un server specifico, piuttosto che dal server dove si trovano le pagine HTML che fanno uso della libreria.

OSSERVAZIONE Quando si include la libreria *jQuery* da un *repository* conviene selezionarne la versione minimizzata che ha nel nome la sottostringa *min*: in questo tipo di libreria sono eliminati gli spazi inutili e sono usati nomi di variabili e funzioni il più corti possibile, sono stati rimossi i commenti e così via. Tutto a discapito della leggibilità del codice, ma a vantaggio del numero di byte da scaricare.

Dentro la libreria

L'analisi del codice sorgente della libreria *jQuery* non è semplice; la soluzione all'incompatibilità tra browser diversi aiuta però a capire come l'uso di una libreria astragga dai dettagli e dalle incompatibilità fornendo primitive di alto livello. Analizzare una libreria come *jQuery* può aiutare a esplorare alcune tecniche di programmazione avanzate con un approccio *learn by example*.

2.1 Selettori

Una delle operazioni più comuni nella programmazione web lato client è quella di selezionare un elemento della pagina HTML; *jQuery* mette a disposizione una funzione, denominata ***selettore***, invocabile utilizzando il solo carattere «$».

Un selettore si applica in questo modo

```
$(nome)
```

dove se il *nome* dell'elemento HTML è preceduto dal simbolo «#», allora ne specifica l'identificatore. Con il ricorso alla libreria *jQuery*, l'esecuzione del codice JavaScript avviene correttamente anche in versioni di browser in cui la funzione *getElementById* può fallire ricercando in realtà gli elementi per nome.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      function mostra(id) {
        var v = $("#" + id);
        alert(v.html());
      }
    </script>
  </head>
  <body>
    <form name="ID2" id="ID1" action="">
      <div>
        <h2>FORM con name=ID2, id=ID1</h2>
      </div>
    </form>
    <div id="ID2">
      <h2>DIV con ID2</h2>
    </div>
    <input type="button" onclick="mostra('ID1')" value="id='ID1'">
    <input type="button" onclick="mostra('ID2')" value="id='ID2'">
  </body>
</html>
```

Il modo di riferirsi a un elemento con uno specifico identificatore mediante la stringa *#id*, è lo stesso che viene usato dai selettori CSS. Questa è una precisa scelta progettuale degli autori di *jQuery*: si possono usare tutti i selettori CSS per selezionare gli elementi del DOM.

Il linguaggio JavaScript permette di selezionare gli elementi del DOM utilizzando i selettori standard CSS e il metodo *querySelectorAll*.

ESEMPIO

```
// recupera tutti gli elementi di classe Classe
var x = document.querySelectorAll(".Classe");

// recupera l'elemento con id ID
var y = document.querySelectorAll("#ID");
```

e, analogamente, un corrispettivo esempio d'uso dei selettori di *jQuery*:

```
$(".Classe").css('color', 'red');
$("#ID").css('color', 'red');
```

A differenza dei selettori CSS, che permettono di agire solo sullo stile di visualizzazione, *jQuery* permette di agire sugli elementi selezionati per modificarne lo stile, il contenuto (compresa l'aggiunta o la rimozione di elementi figli), ma anche il comportamento, cioè la possibilità di rispondere e gestire gli eventi.

Le funzionalità della libreria *jQuery*, rispetto all'impiego del solo linguaggio JavaScript, rendono la programmazione di applicazioni web molto

Migliorare la velocità di esecuzione

I selettori *jQuery* che possono essere attivati mediante l'uso di *querySelectorAll* hanno prestazioni migliori degli altri. L'accesso al DOM per selezioni, ricerche o modifiche, è particolarmente costoso in termini di tempo di esecuzione: per questo motivo è consigliato minimizzare il numero di accessi al DOM da parte del codice di un'applicazione web.

flessibile e versatile. La tabella che segue riporta alcuni dei selettori della libreria *jQuery*.

Selettore (esempio)	Descrizione
`$("p")`	Seleziona gli elementi con *tag* **<p>**.
`$("#ID")`	Seleziona l'elemento con identificatore *ID*.
`$(".Classe")`	Seleziona gli elementi la cui classe è *Classe*.
`$("p.Classe")`	Seleziona tra tutti gli elementi il cui *tag* è **<p>** solo quelli la cui classe è *Classe*.
`$("span p")`	Seleziona gli elementi con *tag* **<p>** che sono all'interno di un *tag* ****.
`$("form[id^=a]")`	Seleziona i *form* i cui identificatori iniziano per «a».
`$("form[name$=a]")`	Seleziona i *form* i cui nomi finiscono con «a».
`$("a[target!=_blank]")`	Seleziona i *link* il cui *target* non è *_blank*.
`$("a[target=_blank]")`	Seleziona i *link* il cui *target* è *_blank*.
`$("tr:odd")`	Seleziona le righe dispari di una tabella.
`$("li:first")`	Seleziona il primo elemento di una lista.
`$(":button")`	Seleziona gli elementi di un particolare tipo, in questo caso pulsanti.

L'importanza dei selettori è evidente: prima di operare su un qualsiasi elemento, infatti, esso deve essere selezionato. Poi, applicando i metodi a «catena», si possono applicare le azioni desiderate: la «catena» è data dalla possibilità di usare i metodi in cascata ad altri; per esempio:

```
$("p").addClass("Classe").show();
```

2.2 *Script* eseguiti al caricamento della pagina

Gli *script* JavaScript possono essere inseriti in qualunque parte della pagina HTML, ma talvolta è desiderabile che alcune funzioni vengano eseguite non appena la pagina viene caricata dal browser. Ma se il codice modifica e/o interagisce con elementi della pagina, bisogna essere certi che nel momento in cui lo *script* viene eseguito, tali elementi siano a disposizione per essere manipolati.

jQuery mette a disposizione un metodo che permette di eseguire il codice solo quando il DOM è pronto per essere manipolato

```
.ready(funzione);
```

dove la funzione passata come parametro contiene il codice da eseguire.

Utilizzare *ready* è molto simile al ricorso del gestore dell'evento *load* della pagina web, ma, a differenza di quest'ultimo, non è necessario che le risorse esterne (per esempio le immagini) siano state caricate: è sufficiente che sia stato costruito l'intero DOM.

Nelle prime versioni della libreria si usava questa sintassi per eseguire una funzione collegata al metodo *ready*:

```
$(document).ready(function() {
    // codice da eseguire
  });
```

jQuery e i *plug-in*

jQuery è una libreria che può essere estesa mediante *plug-in*. Esistono numerosi *plug-in*, ciascuno dei quali risolve una problematica specifica. Uno tra i *plug-in* più diffusi è senz'altro *jQuery UI* (*User Interface*) per la creazione di interfacce grafiche avanzate. È molto usata anche la versione mobile di *jQuery*, ottimizzata per lo sviluppo di applicazioni su dispositivi mobili: permette una semplice gestione di interfacce fondata su HTML5 e incentrata su un'interazione utente che si basa su tocchi e gesti.

Anche se tale sintassi è ancora valida, a partire dalla versione 3 di *jQuery* è stata deprecata in favore della seguente, molto più compatta:

```
$(function() {
    // codice da eseguire
  });
```

2.3 *jQuery* e l'accesso al DOM

L'accesso e la modifica del DOM sono realizzati con *jQuery* invocando opportuni metodi; in particolare l'accesso al codice HTML di un elemento avviene invocando il metodo *html*, l'accesso agli attributi di un elemento avviene invece, sia in lettura sia in scrittura, invocando il metodo *attr*, mentre per accedere alle proprietà CSS si impiega il metodo *css*; infine si usa il metodo *val* per gestire i valori di un campo di un *form*.

I metodi *html*, *css*, *attr* e *val* si applicano sempre al primo degli elementi di un eventuale insieme di elementi selezionati da un selettore. Per applicare il metodo a tutti gli elementi selezionati è necessario iterare gli elementi stessi invocando il metodo *each*.

ESEMPIO

Il metodo *html* è usato per recuperare il contenuto HTML di un elemento:

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        alert($("div").html());
      });
    </script>
  </head>
  <body>
    <div>
      <b>Un esempio</b> d'uso di <em>jQuery</em>
    </div>
    <div>
      <b>Altro testo</b>
    </div>
  </body>
</html>
```

Anche nel caso in cui il selettore restituisca più di un elemento, il metodo *html* si applica solo al primo elemento selezionato; per applicarlo a ciascun elemento del risultato, è necessario scorrere gli elementi invocando il metodo *each*.

La funzione *each* permette in generale di iterare sia su oggetti sia su vettori: nel caso di oggetti l'iterazione avviene sulle proprietà, mentre nel caso di vettori avviene sugli indici; il risultato dell'iterazione non è restituito come valore, ma viene fornito come argomento alla funzione passata come parametro (*callback*).

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        $("div").each(
          function() {
            alert($(this).html());
          }
        );
      });
    </script>
  </head>
  <body>
    <div>
      <b>Un esempio</b> d'uso di <em>jQuery</em>
    </div>
    <div>
      <b>Altro testo</b>
    </div>
  </body>
</html>
```

Il metodo *html* può essere usato anche per impostare il contenuto degli elementi: si deve specificare un parametro che è, o una stringa contenente il codice HTML, o una funzione che restituisce la stringa di codice HTML.

ESEMPIO

A differenza del metodo privo di parametri usato per leggere il contenuto di un elemento HTML, il metodo di impostazione si applica a tutti gli elementi restituiti dal selettore:

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        $("div").html(
          function(index, oldHtml) {
            return oldHtml + "<p>in coda</p>";
          }
        );
      });
    </script>
  </head>
  <body>
    <div>
      <b>Un esempio</b> d'uso di <em>jQuery</em>
    </div>
    <div>
      <b>Altro testo</b>
    </div>
  </body>
</html>
```

Per ottenere il valore di una proprietà CSS si utilizza il metodo *css* a cui viene passato come argomento il nome della proprietà.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        alert($("b").css('color'));
      });
    </script>
  </head>
  <body>
    <div>
      <b>Un esempio</b> d'uso di <em>jQuery</em>
    </div>
    <div>
      <b>Altro testo</b>
    </div>
  </body>
</html>
```

Il metodo *css* gestisce diverse notazioni per il nome della proprietà. È infatti possibile specificare il nome sia secondo la notazione CSS, che per i nomi composti utilizza un simbolo «-» di separazione (esempio: *font-weight*), sia specificando la notazione JavaScript per le proprietà, che prevede l'eliminazione del simbolo «-» e l'uso del maiuscolo per il carattere successivo (esempio: *fontWeight*). Se viene specificato un nome con il simbolo «-», il nome deve essere racchiuso tra apici (singoli o doppi), mentre un nome di proprietà in stile JavaScript non ha questo vincolo.

Il metodo *css* si fa carico anche di gestire eventuali differenze e incompatibilità tra browser diversi.

ESEMPIO

Il metodo *css* può anche essere usato per impostare i valori delle proprietà:

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        $("em").css('color', 'green');
        $("b").css('color', function() { return 'blue'});
        $("div").css( { fontWeight:'bold', color:'red'});
      });
    </script>
  </head>
  <body>
    <div>
      <b>Un esempio</b> d'uso di <em>jQuery</em>
    </div>
    <div>
```

```
        <b>Altro testo</b>
      </div>
    </body>
  </html>
```

Il metodo *attr* permette la lettura dei valori di un attributo di un elemento: il nome dell'attributo di interesse viene fornito come argomento di tipo stringa al metodo stesso.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        alert($("a").attr('href'));
      });
    </script>
  </head>
  <body>
    <a href="test.html">Collegamento</a>
  </body>
</html>
```

ESEMPIO

Il metodo *attr* permette di impostare il valore degli attributi:

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        $("a").attr('href', 'altro.html') ;
        $("a").attr('target', function() {return '_blank'});
        $("table").attr( {border:1, align: 'right'});
      });
    </script>
  </head>
  <body>
    <table>
      <tr><td><a href="test.html">Collegamento</a></td></tr>
    </table>
  </body>
</html>
```

Il metodo *attr* può essere usato per impostare più valori con una singola invocazione fornendo un vettore come parametro della funzione. Per esempio per impostare le dimensioni di un'immagine si può scrivere:

```
$("img").attr({width: "150", height: "100"})
```

Il metodo *val* può essere usato per recuperare il valore di un elemento di un *form*: se l'elemento può assumere solo un valore si ottiene come risultato una stringa, quando ne può restituire più di uno, il risultato è un *array* di stringhe.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        alert(
          $("#a").val() +
          $("#b").val() +
          $("#c").val() +
          $("#d").val()
        );
      });
    </script>
  </head>
  <body>
    <input type="text" id="a" value="a">
    <input type="radio" checked="checked" id="b" value="b">
    <input type="checkbox" checked="checked" id="c" value="c">
    <select id="d">
      <option value="d" selected="selected">D</option>
    </select>
  </body>
</html>
```

Si noti nel codice precedente l'enorme semplificazione che si ha nell'utilizzare il metodo *val* anziché i diversi metodi per accedere ai diversi tipi di elementi di un *form*.

2.4 *jQuery* e altre librerie

jQuery può essere utilizzato senza problemi con altre librerie: le funzioni *jQuery* iniziano con il simbolo «$» o, in alternativa, con la stringa *jQuery*. Nel caso in cui potrebbe crearsi un conflitto con notazioni simili usate da altre librerie, è possibile utilizzare un nome alternativo per riferirsi alle funzioni della libreria *jQuery*. Per esempio, volendo usare la stringa *$jQueryAlias* al posto del carattere «$» è possibile definire la seguente variabile:

```
var $jQueryAlias = jQuery.noConflict();
```

Successivamente alla dichiarazione è possibile utilizzare la stringa *$jQueryAlias* per invocare le funzioni della libreria *jQuery*:

```
$jQueryAlias( function() {
  $jQueryAlias("p").text("Testo");
});
```

2.5 Considerazioni sulle prestazioni

In generale, per ragioni di performance degli *script*, è bene fare un uso oculato dei selettori ed è preferibile riutilizzare i selettori precedenti per applicare catene di metodi anziché ripetere un selettore. Inoltre si dovrebbe accedere il meno possibile al DOM.

ESEMPIO

Nel codice che segue vengono valorizzate due liste: nel primo caso si aggiunge un singolo elemento alla volta al DOM; nel secondo caso si costruisce una stringa con tutti gli elementi e poi, in una sola invocazione, si va a modificare il DOM. Quest'ultima strategia risulta la più conveniente in termini di performance.

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        var frutta = ["Mele", "Pere", "Mirtilli"];
        for (var i=0; i<frutta.length; i++) {
          var lista = "<li>" + frutta[i] + "</li>";
          $("#frutta").append(lista);
        }
      });
      $(function() {
        var verdura = ["Insalata", "Cicoria", "Spinaci"];
        var lista="";
        for (var i=0; i<verdura.length; i++) {
          lista = lista + "<li>" + verdura[i] + "</li>";
        }
        $("#verdura").append(lista);
      });
    </script>
  </head>
  <body>
    <ol id="frutta"></ol>
    <ol id="verdura"></ol>
  </body>
</html>
```

Nel caso, per esempio, di aggiunta dinamica di un certo numero di righe a una tabella HTML, l'aggiunta implica l'accesso al DOM per ogni singolo elemento. Si può ricorrere al metodo *detach* per rimuovere temporaneamente l'elemento dal DOM, modificarlo e, quindi, reintrodurlo: queste tre operazioni risultano più performanti rispetto alla modifica diretta che comporta, ogni volta, l'accesso al DOM.

ESEMPIO

```
<!doctype html>
<html>
  <head>
    <script src="jquery.js"></script>
    <script>
      $(function() {
        var table = $("#tabella");
        var parent = table.parent();
        table.detach();
        for (var r=0; r<100; r++) // 100 righe
          table.append(
            "<tr><td>" + r + "</td><td>" +
            Math.random() + "</td></tr>");
        parent.append(table);
      });
```

```
    </script>
  </head>
  <body>
    <table id="tabella">
      <tr><td>Progressivo</td><td>Casuale</td></tr>
    </table>
  </body>
</html>
```

3 JavaScript e i *thread*

Le recenti versioni del linguaggio JavaScript sono dotate di strumenti per permettere alle funzioni di essere eseguite in modo concorrente e asincrono.

Normalmente uno *script* JavaScript blocca il browser fino al termine della sua esecuzione: questo perché il browser, non sapendo se lo *script* modifica o meno i contenuti della pagina, evita di eseguire contemporaneamente altri *script*, cosa che potrebbe portare risultati indesiderati. Se lo *script* ha un tempo di esecuzione elevato, l'utente lo percepisce come bloccato.

ESEMPIO

Il seguente *script* esegue un ciclo che non ha fine:

```
<!doctype html>
<html>
  <body>
    <script>
      for (var i=0; i<10; i++)
        i--;
    </script>
  </body>
</html>
```

I browser moderni considerano anomala la situazione in cui l'esecuzione di uno *script* si protrae a lungo e chiedono all'utente se continuarne l'esecuzione o bloccarla (FIGURA 4).

FIGURA 4

Anche la visualizzazione della finestra *pop-up* di messaggistica effettuata invocando la funzione *alert* blocca l'esecuzione dello *script* e, finché l'utente non chiude la finestra, il resto del documento non viene caricato.

Normalmente esiste un unico *thread* del browser che effettua sia il *rendering* della pagina sia l'esecuzione degli *script*.

Per l'esecuzione di *script* che impiegano molto tempo, o che attendono dati da una fonte esterna, è desiderabile non dover attendere la fine dell'esecuzione dello *script*.

Nel caso di una fonte di dati esterna esistono oggetti esposti dal browser che permettono di inviare una richiesta a un sito remoto indicando una funzione di *callback* che dovrà essere invocata quando la risposta è disponibile: in questo modo il programma può continuare la sua esecuzione senza preoccuparsi dei tempi di risposta.

OSSERVAZIONE Il meccanismo di invocazione asincrona di fonti di dati esterne rende possibile sviluppare applicazioni che fanno uso del cosiddetto AJAX (*Asynchronous JavaScript And XML*) che prevede una comunicazione dati asincrona in formato XML per aggiornare i contenuti delle pagine web, senza la necessità da parte dell'utente di aggiornare la pagina per visualizzare i nuovi dati.

Dato che l'esecuzione degli *script* JavaScript avviene a carico di un singolo *thread*, se ci sono più file di codice esterni la loro inclusione ed esecuzione avviene in sequenza, un file dopo l'altro: nel caso che alcuni *script* abbiano un caricamento, o un'esecuzione di lunga durata, essi bloccano gli altri *script* e il caricamento della pagina.

ESEMPIO Predisponendo tre file, denominati *file1.js*, *file2.js* e *file3.js*, contenenti il codice che segue e modificando opportunamente il messaggio di *debug* visualizzato sulla *console*

```
(function() {
  var c = 0;
  for (var a=0; a<10000*random(); a++)
    for (var b=a; b<100000; b++)
      c = a * b - a + b;
  console.log('Fine esecuzione script esterno #1');
}()
);
```

è possibile simulare un'esecuzione avente una durata di alcuni secondi.
Il codice è costituito da una funzione anonima con sole variabili locali in modo da evitare conflitti; la dichiarazione della funzione è seguita dalla sua invocazione.
Il caricamento della seguente pagina HTML che include i tre file

```
<!doctype html>
<html>
  <head>
    <script src="file1.js"></script>
    <script src="file2.js"></script>
    <script src="file3.js"></script>
    <script>
      console.log('Esecuzione script interno');
    </script>
  </head>
```

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

si ferma per alcuni secondi, poi viene visualizzato il messaggio di *file1.js*, poi quello di *file2.js*, seguito da quello di *file3.js* e, infine, l'esecuzione dello *script* interno alla pagina che visualizza il relativo messaggio.

È possibile eseguire file esterni di codice JavaScript in maniera asincrona specificando l'attributo *async* nel *tag* `<script>` che ne effettua l'inclusione e il caricamento: in questo modo il codice contenuto nel file verrà caricato ed eseguito senza bloccare il caricamento del resto della pagina e l'esecuzione di eventuali *script* interni a essa.
In alcuni contesti è necessario che lo *script* sia eseguito solo a caricamento avvenuto dell'intera pagina: in questo caso è necessario specificare l'attributo *defer* nel *tag* che include il codice JavaScript.

ESEMPIO

Il codice della pagina dell'esempio precedente deve essere così modificato per fare in modo che i file di codice esterni vengano caricati in maniera asincrona:

```
<!doctype html>
<html>
  <head>
    <script async src="file1.js"></script>
    <script async src="file2.js"></script>
    <script async src="file3.js"></script>
    <script>
      console.log('Esecuzione script interno');
    </script>
  </head>
  <body>
  </body>
</html>
```

Caricando in un browser più volte questa pagina si può notare che non è garantito l'ordine con cui sono visualizzati i messaggi contenuti nei vari file e nel codice della pagina stessa; l'esecuzione del codice dei singoli file è concorrente e l'esecuzione complessiva è molto più veloce, perché non è necessario attendere la fine degli *script* per completare la visualizzazione (*rendering*) della pagina.

4 Google Maps

Google ha reso disponibile un servizio denominato ***Google Maps*** che permette di visualizzare mappe geografiche e di mostrarle attraverso viste diversificate.
Per gli sviluppatori queste mappe sono rese disponibili per essere incluse nei propri siti e applicazioni web; esse sono personalizzabili in vari modi, ciascuno dei quali è reso fruibile attraverso opportune API, tra le quali vi è una libreria specifica per il linguaggio JavaScript.

Per poter usare le *Google Maps* nelle proprie applicazioni web realizzate con JavaScript è necessario ottenere un'API *key*[2].

2. L'API *key* si ottiene accedendo alla *Google API Console*; l'uso non commerciale delle mappe è gratuito se l'applicazione è pubblica: esiste un limite sul numero di richieste giornaliere che è più che sufficiente per scopi didattici (dell'ordine di alcune migliaia).

Le mappe sono generate dinamicamente utilizzando le API, ed è dinamico anche l'aggiornamento del contenuto in base alle azioni dell'utente. La pagina HTML che comprenderà la mappa deve includere la libreria delle API:

```
<script
  src="https://maps.googleapis.com/maps/api/js?key=...">
</script>
```

Inoltre deve essere prevista un'area dove la mappa verrà visualizzata: quasi sempre l'area viene definita utilizzando un *tag* `<div>` privo di contenuti. Dato che la visualizzazione dell'immagine della mappa sarà successiva al caricamento della pagina, è importante specificare la dimensione dell'area; la mappa riempirà tutta l'area così assegnata:

```
<body>
  <h1>La prima mappa</h1>
  <div id="mappa" style="width: 640px; height: 350px"></div>
</body>
```

Se il tag `<div>` specificato non ha una dimensione fissa, ma ha una dimensione espressa in percentuale rispetto alla finestra che lo contiene, quando si ridimensiona la finestra viene ridimensionata anche la mappa: essa viene nuovamente ricaricata dal server e va a occupare la nuova dimensione, come indicato dal valore percentuale.

Se viene specificato che la dimensione della mappa è il 100% del contenitore, è raccomandabile impostare tale dimensione anche ai contenitori che racchiudono la mappa; nel caso non ci siano contenitori particolari, è bene specificare la dimensione almeno sui *tag* `<body>` e `<html>`.

ESEMPIO

Ecco come impostare la dimensione della mappa al 100% dell'altezza:

```
<style>
  #mappa {
    height: 100%;
  }
  html, body {
    height: 100%;
    margin: 0;
    padding: 0;
  }
</style>
```

La mappa deve essere creata specificando le caratteristiche che deve avere come proprietà di un oggetto fornito come parametro al costruttore *google.maps.Map*; le principali proprietà sono illustrate nella TABELLA 4.

Documentazione *Google Maps*

Google rende disponibile un'esauriente documentazione delle proprie API. Tra i documenti disponibili per *Google Maps* si trovano il *reference*, un *tutorial* e una *developer's guide*. Il *tutorial* è fatto in modo che, eseguendo passo per passo quanto scritto, si possono realizzare semplici esempi funzionanti. La *developer's guide* affronta i contenuti in modo più approfondito perché è orientata agli sviluppatori. Infine il *reference* presenta i dettagli di tutti i metodi esposti dai vari oggetti della libreria.

TABELLA 4

Proprietà	Descrizione
mapTypeId	Indica il tipo di visualizzazione che è obbligatorio specificare; tra i valori ammessi quelli di uso più frequente sono: • *roadmap*, per le mappe riportanti le informazioni cartografiche; • *satellite*, per le immagini satellitari; • *hybrid*, per le immagini satellitari più le principali informazioni cartografiche; • *terrain*, per le informazioni altimetriche.

Proprietà	Descrizione
center	Le coordinate del centro della mappa che è obbligatorio specificare. Il valore è un oggetto di tipo *google.maps.LatLng* che può essere rappresentato come un oggetto dove sono valorizzate le proprietà *lat* e *lng*; la creazione dell'oggetto avviene invocando un costruttore con due parametri che sono, rispettivamente, la latitudine e la longitudine del punto di interesse, in questo caso il centro della mappa.
zoom	Livello di zoom dove 0 è «nessuno» e corrisponde all'immagine dell'intero globo, mentre numeri maggiori corrispondono a zoom via via maggiori, visualizzando mappe sempre più dettagliate.

ESEMPIO

La seguente funzione JavaScript crea una mappa centrata sul Colosseo di Roma (latitudine 41,8905° Nord, longitudine 12,4921° Est) con un valore di zoom pari a 8:

```
<script>
  var map;
  function inizializzaMappa() {
    var myOptions = {
      zoom: 18,
      center: { lat: 41.8905, lng: 12.4921 },
      mapTypeId: 'satellite'
    };
    map = new google.maps.Map(
      document.getElementById('mappa'),
      myOptions
    );
  }
</script>
```

La funzione *inizializzaMappa* deve essere invocata perché visualizzi effettivamente la mappa (FIGURA 5): un modo per farlo è quello di attendere il caricamento della pagina, ovvero associarlo all'evento *load*:

```
<!doctype html>
<html>
  <head>
    <script src="https://maps.googleapis.com/maps/api/
                 js?key-...">
    </script>
    <script>
      var map;
      function inizializzaMappa() {
        var myOptions = {
        zoom: 18,
        center: { lat: 41.8905, lng: 12.4921 },
        mapTypeId: 'satellite'
        };
      map = new google.maps.Map(
        document.getElementById('mappa'),
        myOptions
      );
    }
    </script>
  </head>
  <body onload="inizializzaMappa()">
    <h1>La prima mappa</h1>
    <div id="mappa" style="width: 640px; height: 350px"></div>
  </body>
</html>
```

FIGURA 5

Per evitare l'uso del gestore di eventi *onload* del *tag* **<body>** e, al tempo stesso, velocizzare il caricamento della pagina, è possibile invocare la libreria *Google Maps* in questo modo:

```
<script async defer
  src="https://maps.googleapis.com/maps/api/js?key=...&
       callback=inizializzaMappa">
</script>
```

L'uso congiunto di *async* e *defer* come attributi fa sì che lo *script* sia caricato in maniera asincrona rispetto al resto della pagina ed eseguito solo quando la pagina è stata caricata; il parametro *callback* fornito all'URL specifica la funzione che viene invocata dal sistema di gestione delle mappe per inizializzare la mappa.

4.1 Georeferenziare un indirizzo

L'esempio precedente mostra che per poter creare una mappa di un luogo è necessario conoscerne le coordinate geografiche. Google rende disponibili ulteriori API – note come *Google geocoding API* – che permettono, dato l'indirizzo di un qualsiasi luogo, di conoscerne le coordinate geografiche.
Utilizzando *Google Maps* non è necessario ricorrere a questo servizio, ma si può far uso di una classe che lo utilizza indirettamente: la classe *Geocoder* il cui metodo *geocode* richiede come parametro un oggetto con impostata la proprietà *address* che contiene l'indirizzo da georeferenziare.

Il risultato dell'operazione di georeferenziazione è un *array* (possono infatti corrispondere più luoghi a un unico indirizzo se esso non è univoco) di oggetti aventi la seguente struttura passato come argomento a una funzione di *callback* fornita come parametro al metodo *geocode* stesso:

```
{
  types[]: string,
  formatted_address: string,
  address_components[]: {
    short_name: string,
    long_name: string,
```

```
    types[]: string
  },
  geometry: {
    location: LatLng,
    location_type: GeocoderLocationType
    viewport: LatLngBounds,
    bounds: LatLngBounds
  }
}
```

Le proprietà del risultato sono fondamentali per interpretarlo correttamente: la proprietà *types* è un *array* di stringhe ciascuna delle quali identifica se il risultato è una città, un'entità politica, ...; la proprietà *formatted_address* è l'indirizzo georeferenziato in forma normalizzata, espansa nei componenti che la costituiscono nell'*array address_components*; la proprietà *geometry* è un oggetto che contiene la localizzazione vera e propria e di cui la proprietà *location* contiene le coordinate geografiche (latitudine e longitudine).

ESEMPIO La pagina web che segue permette all'utente di specificare un indirizzo e georeferenziarlo: il risultato della georeferenziazione viene visualizzato nei campi di un *form*, inoltre viene creata una mappa dettagliata corrispondente all'indirizzo inserito in sostituzione di quella del Colosseo di Roma visualizzata inizialmente (FIGURA **6**).

```
<!doctype html>
<html>
  <head>
    <script src="https://maps.googleapis.com/maps/api/js?key=...">
    </script>
    <script>
      var map;
      function inizializzaMappa() {
        var myOptions = {
          zoom: 18,
          // coordinate geografiche del Colosseo di Roma
          center: { lat: 41.8905, lng: 12.4921 },
          mapTypeId: 'satellite'
        };
        map = new
          google.maps.Map(document.getElementById('mappa'),myOptions);
      }
      function georeferenzia() {
        var indirizzo = document.getElementById('indirizzo').value;
        var geocoder = new google.maps.Geocoder();
        geocoder.geocode(
          { 'address': indirizzo }, funzioneCallback );
      }
      function funzioneCallback(risultati, status) {
        if (status == google.maps.GeocoderStatus.OK) {
          var coord = risultati[0].geometry.location;
          map.setCenter(coord);
          document.getElementById('latitudine').value = coord.lat();
          document.getElementById('longitudine').value = coord.lng();
        } else
        alert("Errore"+ status);
      }
```

HUFD

```
    </script>
  </head>
  <body onload="inizializzaMappa()">
    <h1>Cerca per indirizzo</h1>
    Indirizzo
    <input type="text" id="indirizzo" name="indirizzo">
    <input type="button" value="GeoReferenzia"
           onclick="georeferenzia()">
    <div id="mappa" style="width: 640px; height: 350px"></div>
    Lat:<input type="text" id="latitudine" name="latitudine">
    Lng:<input type="text" id="longitudine" name="longitudine">
  </body>
</html>
```

FIGURA 6

4.2 Creare un marcatore

Una delle caratteristiche più flessibili e interessanti delle *Google Maps* è rappresentata dalla possibilità di creare oggetti, figure geometriche e altri elementi visualizzati sopra le mappe. Tra gli elementi più utilizzati a questo scopo ci sono i cosiddetti **marcatori**: icone posizionate su un punto che aiutano a indicare visivamente una posizione di interesse. I marcatori possono rispondere agli eventi come, per esempio, al click su di essi, mostrando fumetti o altri effetti grafici.

Un marcatore può essere creato usando il costruttore *Marker* a cui viene passato un oggetto le cui proprietà possono essere quelle mostrate nella seguente tabella:

Proprietà	Descrizione
map	La mappa su cui mostrare il marcatore.
position	Le coordinate geografiche dove mostrare il marcatore.
title	Descrizione che compare quando il cursore del mouse è sopra il marcatore.
zIndex	Quando due o più marcatori si sovrappongono viene mostrato in primo piano quello con valore *zIndex* più alto; in mancanza di questa proprietà i marcatori sono mostrati secondo l'ordine di posizionamento in verticale: quelli più in basso sono mostrati davanti a quelli più in alto.
clickable	Se `true`, ed è il valore di default, l'oggetto risponde ai click del mouse. ▶

Proprietà	Descrizione
draggable	Se **true** può essere spostato; il valore di default è **false**.
flat	Se **true** non viene mostrata l'ombra del marcatore.
cursor	Tipo di cursore da mostrare quando il mouse è sopra il marcatore.
icon	Icona da usare come primo piano.
shadow	Icona da usare come ombra.
animation	Eventuale animazione da associare all'aggiunta del marcatore a una mappa; le animazioni possibili sono le seguenti: • *google.maps.Animation.BOUNCE*, fa sì che il cursore «saltelli»; • *google.maps.Animation.DROP*, disegna il cursore facendolo cadere dall'alto della mappa. Una volta impostata un'animazione è possibile fermarla assegnando il valore **null** alla relativa proprietà invocando il metodo *setAnimation*.

ESEMPIO

Il codice JavaScript riportato nella seguente pagina HTML crea una mappa, con un livello di zoom appropriato, che mostra sei marcatori disposti a cerchio il cui centro è la posizione del Colosseo di Roma (FIGURA 7): per ogni marcatore sono calcolate le sue coordinate geografiche e viene specificata la mappa su cui deve essere visualizzato (il calcolo delle coordinate fa uso delle funzioni trigonometriche *Math.sin* e *Math.cos*):

```
<!doctype html>
<html>
  <head>
    <script src="https://maps.googleapis.com/maps/api/js?key=...">
    </script>
    <script>
      var map;
      function inizializzaMappa() {
        var lat = 41.8905;
        var lng = 12.4921;
        var myOptions = {
          zoom: 16,
          center: new google.maps.LatLng(lat, lng),
          mapTypeId: 'satellite'
        };
        map = new google.maps.Map(
          document.getElementById('mappa'), myOptions);

        for (var i=0; i<6; i++) {
          var a = lat + Math.sin(i)/650;
          var b = lng + Math.cos(i)/650;
          new google.maps.Marker(
            {
              position: { lat: a, lng: b },
              map: map
            }
          );
        }
      }
    </script>
  </head>
  <body onload="inizializzaMappa()">
    <div id="mappa" style="width: 640px; height: 350px"></div>
  </body>
</html>
```

FIGURA 7

4.3 Gestire gli eventi

Per le mappe è possibile gestire due tipi di eventi: alcuni, simili a quelli del DOM, si riferiscono a determinate azioni effettuate sulle mappe stesse, come il click del mouse, o il movimento del cursore del mouse; altri si riferiscono a proprietà che variano in seguito ad azioni dell'utente sull'interfaccia della mappa come, per esempio, il cambio di zoom o del tipo di mappa visualizzata.

Con il metodo *google.maps.event.addListener* vengono associate funzioni a determinati eventi sui diversi oggetti gestiti dalle API *Google Maps*:

```
google.maps.event.addListener(
  oggetto, evento,
  function(event) {
    ...
  }
);
```

Il primo argomento è l'oggetto – come una mappa o un marcatore – su cui registrare la funzione, il secondo è il nome dell'evento a cui rispondere e, infine, la funzione da invocare al verificarsi dell'evento. L'evento intercettato è un oggetto che viene passato come argomento alla funzione: tra le sue proprietà rientrano anche le coordinate del punto dove è stato registrato: *event.latLng*.

Il metodo *google.maps.event.addDomListener* ha gli stessi parametri di *addListener* e permette di gestire gli eventi del DOM.

OSSERVAZIONE Anche gli eventi DOM sono in realtà gestiti dalla libreria *Google Maps* e presentano caratteristiche specifiche; per esempio, è possibile conoscere le coordinate geografiche del punto di una mappa su cui l'utente ha fatto click con il mouse.

ESEMPIO

Il codice della seguente pagina web illustra come eseguire la funzione *inizializzaMappa* al termine del caricamento della pagina stessa; inoltre, per ogni click del mouse effettuato dall'utente sulla mappa visualizzata viene generato un *alert* che riporta le coordinate geografiche del punto:

```
<!doctype html>
<html>
  <head>
    <script src="https://maps.googleapis.com/maps/api/js?key=...">
    </script>
    <script>
    var map;
    function inizializzaMappa() {
      var myOptions = {
        zoom: 5,
        center: { lat: 40.9, lng: 10.5},
        mapTypeId: 'satellite'
      };
      map = new google.maps.Map(
        document.getElementById('mappa'),myOptions);
      google.maps.event.addListener(
        map, 'click',
        function(event) {
          alert('lat:'+ event.latLng.lat() +
            ', lng:'+ event.latLng.lng() );
        });
      }
      google.maps.event.addDomListener(
        window, 'load',
        inizializzaMappa);
    </script>
  </head>
  <body>
    <div id="mappa" style="width: 640px; height: 350px"></div>
  </body>
</html>
```

4.4 Mostrare una finestra di informazioni

Un oggetto di tipo *google.maps.InfoWindow* è una finestra a forma di fumetto che si apre all'interno di una mappa: è normalmente associata a un evento sui marcatori, ma è possibile ancorarla su una posizione qualsiasi della mappa.

Il costruttore riceve come argomento un oggetto per il quale possono essere definite diverse proprietà, le principali delle quali sono riportate nella seguente tabella:

Proprietà	Descrizione
content	È il contenuto da mostrare quando la finestra è aperta: si tratta di testo e/o codice HTML.
position	Contiene la posizione dove ancorare la finestra, se viene associata a un marcatore la eredita da esso.
maxWidth	Dimensione massima della finestra, se non viene specificata questa proprietà la finestra si ridimensiona in base alla dimensione del contenuto.

Una volta definita una finestra informativa essa deve essere visualizzata invocandone il metodo *open*.

ESEMPIO

La pagina web che segue visualizza una mappa del Lago di Garda, al cui centro è posizionato un marcatore che reagisce al click del mouse mostrando una finestra di informazioni (FIGURA 8).

```
<!doctype html>
<html>
  <head>
    <script src="https://maps.googleapis.com/maps/api/js?key-...">
    </script>
    <script>
      var map;
      function inizializzaMappa() {
        var coord = { lat: 45.605, lng: 10.63}; // Lago di Garda
        var myOptions = {
          zoom: 9,
          center: coord,
          mapTypeId: 'satellite'
        };
        map = new google.maps.Map(
          document.getElementById('mappa'),myOptions);
        var infowindow = new google.maps.InfoWindow( {
          content: "Questo è il <b>Lago di Garda</b>"
        });
        var marker = new google.maps.Marker( {
          position: coord,
          map: map,
          title: 'Lago'
        });
        marker.addListener('click', function() {
          infowindow.open(map, marker);
        });
        google.maps.event.addDomListener(
          window, 'load',
          inizializzaMappa);
      }
    </script>
  </head>
  <body>
    <div id="mappa" style="width: 640px; height: 350px"></div>
  </body>
</html>
```

FIGURA 8

I CONCETTI CHIAVE

DOM. Per poter modificare il contenuto di una pagina HTML è necessario che gli *script* possano accedere alla rappresentazione interna dei documenti. Oltre a questa caratteristica il browser offre agli *script* una collezione di oggetti che modellano vari aspetti di una pagina (DOM, *Document Object Model*) e del browser stesso.

OGGETTO *WINDOW*. Esiste un oggetto che è il genitore di tutti quelli messi a disposizione da un browser: l'oggetto *window*, che modella la finestra corrente.

OGGETTO *NAVIGATOR*. L'oggetto *navigator* permette di accedere alle informazioni relative al browser che sta eseguendo il codice.

OGGETTO *LOCATION*. L'oggetto *location* modella l'indirizzo del documento visualizzato nella finestra.
In particolare, la proprietà *href* è quella che contiene l'URL.

OGGETTO *HISTORY*. Per un'applicazione JavaScript è possibile navigare la storia delle pagine visitate precedentemente rispetto alla pagina dove viene eseguito lo *script*. Questo avviene grazie all'oggetto *history* e ai metodi che espone.

OGGETTO *SCREEN*. L'oggetto *screen* fornisce informazioni sullo schermo su cui viene visualizzato il browser e sulle sue caratteristiche in termini di risoluzione video e di colori supportati.

COLLEZIONI DI OGGETTI DI *DOCUMENT*. Il browser espone, come proprietà dell'oggetto *document*, collezioni di oggetti grazie alle quali è possibile accedere a determinati tipi di elementi di una pagina HTML; in particolare espone le collezioni *anchors* e *link* che contengono, rispettivamente, tutte le ancore e i *link* della pagina; *forms* che contiene tutti i *form* e *images* per le immagini.

ELEMENTI DI UN *FORM*. Un *form* può contenere vari elementi come aree di testo, caselle di testo, liste di scelta (singola o multipla), caselle di selezione (a selezione singola, *radio button*, o a selezione multipla, *checkbox*): ciascuno di essi espone proprietà per leggere e impostare i valori inseriti e/o selezionati.

ACCESSO AI CSS. Attraverso l'utilizzo del DOM è possibile specificare anche le proprietà normalmente gestite mediante CSS. È infatti possibile accedere alla proprietà *style* come a qualsiasi altra proprietà del DOM e, di conseguenza, si può accedere alle specifiche proprietà CSS come proprietà di *style*.

LIBRERIA *JQUERY*. *jQuery* è una libreria per il linguaggio JavaScript, scaricabile con licenza *open-source* MIT dal sito **https://jquery.com**, che ha come finalità principale quella di semplificare la manipolazione e la gestione di documenti HTML, in maniera *cross* browser, con codice fortemente ottimizzato, sia in termini di prestazioni sia di dimensioni della libreria stessa. Il nome stesso della libreria *jQuery* ricorda che una delle sue caratteristiche fondamentali è l'interrogazione condotta sugli elementi del DOM di una pagina web con una gestione il più possibile omogenea e semplificata degli stessi.

SELETTORI *JQUERY*. Una delle operazioni più comuni nella programmazione web lato client è quella di selezionare un elemento della pagina HTML; *jQuery* mette a disposizione una funzione, denominata *selettore*, invocabile utilizzando il solo carattere «$». Il modo di riferirsi a un elemento con uno specifico identificatore mediante la stringa *#id*, è lo stesso che viene usato dai selettori CSS. Questa è una precisa scelta progettuale degli autori di *jQuery*: si possono usare tutti i selettori CSS per selezionare gli elementi del DOM.
A differenza dei selettori CSS, che permettono di agire solo sullo stile di visualizzazione, *jQuery* permette di agire sugli elementi selezionati per modificarne lo stile, il contenuto (compresa l'aggiunta o la rimozione di elementi figli), ma anche il comportamento, cioè la possibilità di rispondere e gestire gli eventi.

***JQUERY* DOM**. L'accesso e la modifica del DOM sono realizzati con *jQuery* invocando opportuni metodi; in particolare l'accesso al codice HTML di un elemento avviene invocando il metodo *html*, l'accesso agli attributi di un elemento avviene invece, sia in lettura sia in scrittura, invocando il metodo *attr*, mentre per accedere alle proprietà CSS si impiega il metodo *css*; infine si usa il metodo *val* per gestire i valori di un campo di un *form*.

JAVASCRIPT E I *THREAD*. Normalmente esiste un unico *thread* del browser che effettua sia il *rendering* della pagina sia l'esecuzione degli *script*.

JAVASCRIPT ASINCRONO. È possibile eseguire file esterni di codice JavaScript in maniera asincrona specificando l'attributo *async* nel *tag* `<script>` che ne effettua l'inclusione e il caricamento: in questo modo il codice contenuto nel file verrà caricato ed eseguito senza bloccare il caricamento del resto della pagina e l'esecuzione di eventuali *script* interni a essa.
In alcuni contesti è necessario che lo *script* sia eseguito solo a caricamento avvenuto dell'intera pagina: in questo caso è necessario specificare l'attributo *defer* nel *tag* che include il codice JavaScript.

***GOOGLE MAPS*.** Google ha reso disponibile un servizio denominato *Google Maps* che permette di visualizzare

mappe geografiche e di mostrarle attraverso viste diversificate.
Per gli sviluppatori queste mappe sono rese disponibili per essere incluse nei propri siti e applicazioni web; esse sono personalizzabili in vari modi, ciascuno dei quali è reso fruibile attraverso opportune API, tra le quali vi è una libreria specifica per il linguaggio JavaScript.

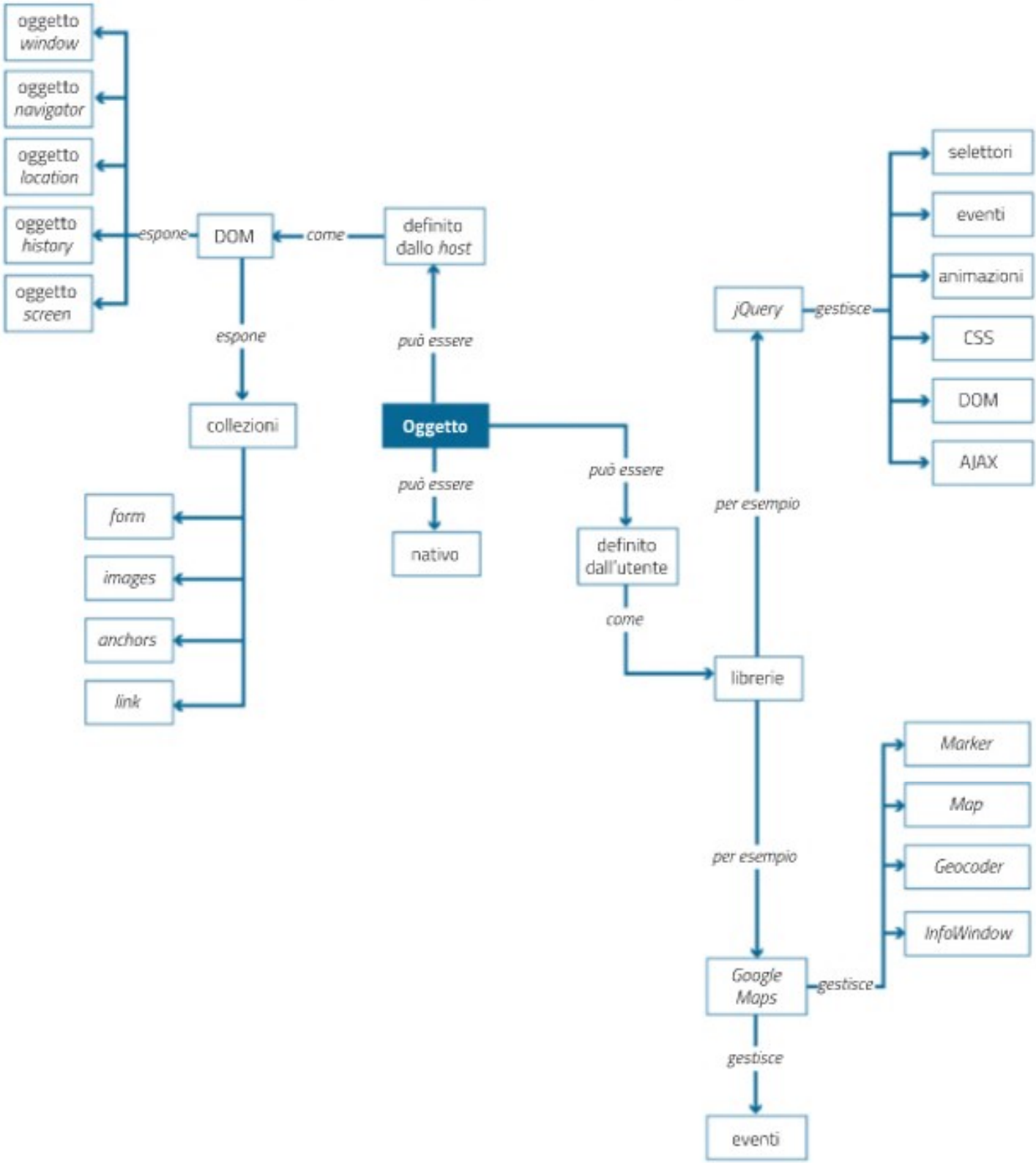
GEOREFERENZIAZIONE. Per poter creare una mappa di un luogo è necessario conoscerne le coordinate geografiche. Google rende disponibili ulteriori API – note come *Google geocoding API* – che permettono, dato l'indirizzo di un qualsiasi luogo, di conoscerne le coordinate geografiche.
Utilizzando *Google Maps* non è necessario ricorrere a questo servizio, ma si può far uso di una classe che lo utilizza indirettamente: la classe *Geocoder* il cui metodo *geocode* richiede come parametro un oggetto con impostata la proprietà *address* che contiene l'indirizzo da georeferenziare.

MARCATORI *GOOGLE MAPS*. Una delle caratteristiche più flessibili e interessanti delle *Google Maps* è rappresentata dalla possibilità di creare oggetti, figure geometriche e altri elementi visualizzati sopra le mappe. Tra gli elementi più utilizzati a questo scopo ci sono i cosiddetti *marcatori*: icone posizionate su un punto che aiutano a indicare visivamente una posizione di interesse. I marcatori possono rispondere agli eventi come, per esempio, al click su di essi, mostrando fumetti o altri effetti grafici.

EVENTI *GOOGLE MAPS*. Per le mappe è possibile gestire due tipi di eventi: alcuni, simili a quelli del DOM, si riferiscono a determinate azioni effettuate sulle mappe stesse, come il click del mouse, o il movimento del cursore del mouse; altri si riferiscono a proprietà che variano in seguito ad azioni dell'utente sull'interfaccia della mappa come, per esempio, il cambio di zoom o del tipo di mappa visualizzata.

FINESTRE INFORMATIVE *GOOGLE MAPS*. Un oggetto di tipo *google.maps.InfoWindow* è una finestra a forma di fumetto che si apre all'interno di una mappa: è normalmente associata a un evento sui marcatori, ma è possibile ancorarla su una posizione qualsiasi della mappa.

RIPASSA CON LA MAPPA



SPIEGA IN UNA PAGINA

1 Cosa sono gli oggetti JavaScript forniti dallo *host*? Illustrare in particolare i principali oggetti forniti da un browser e le relative funzionalità esposte.

2 Per quali motivi una libreria come *jQuery* risulta utile nello sviluppo di applicazioni web? Introdurre le caratteristiche e le funzionalità offerte da *jQuery*.

QUESITI A RISPOSTA APERTA

3 Perché non è possibile conoscere, utilizzando il DOM, gli URL delle pagine visitate in precedenza?

4 A cosa serve un selettore *jQuery*?

5 Qual è l'approccio corretto per rendere maggiormente performanti gli *script jQuery* che modificano parti del DOM?

6 Quali vantaggi si riscontrano nel rendere asincrone parti di codice JavaScript?

7 Perché il risultato restituito dal metodo *geocode* della classe *google.maps.Geocoder* è un *array*?

8 Cosa differenzia i metodi *addListener* e *addDomListener* delle API *Google Maps*?

TEST A RISPOSTA MULTIPLA

9 Per DOM si intende:

A un insieme di oggetti esposti da un browser per interagire con il browser stesso e le pagine visualizzate.

B una parte del linguaggio di programmazione JavaScript che permette di gestire collezioni di oggetti qualunque.

C una libreria di oggetti che va scaricata e inclusa nella propria pagina.

D un insieme di oggetti definiti dallo standard ECMAScript.

10 Il metodo *document.getElementById* serve a:

A recuperare tutti gli elementi che sono identificati da un certo tipo di *tag*.

B recuperare un elemento specificando il suo *id*.

C modificare l'*id* di un elemento.

D modificare il tipo di *tag*.

11 Indicare quale tra questi oggetti permette di ottenere informazioni sul browser che sta visualizzando la pagina.

A *window*.

B *history*.

C *navigator*.

D *location*.

12 Indicare quale tra questi oggetti permette di ottenere informazioni sulla risoluzione video del dispositivo in cui il browser sta visualizzando la pagina.

A *window*.

B *history*.

C *navigator*.

D *location*.

13 Utilizzando l'oggetto *location*, che differenza c'è tra usare il metodo *replace* o assegnare alla proprietà *href* un nuovo valore per cambiare la pagina visualizzata?

A Non c'è alcuna differenza.

B Si deve usare *replace* perché *href* è una proprietà in sola lettura.

C Usando *replace* non resta traccia della pagina precedente nella cronologia.

D *Replace* permette di ricaricare solo la pagina corrente.

14 Supponendo che esista un *form* denominato *test*, qual è il modo corretto per riferirsi a esso?

A `document.getElementById('test').`

B `document.forms['test'].`

C `window.location['test'].`

D `document.getElementById('form').`

15 È possibile modificare il valore delle proprietà CSS direttamente da JavaScript?

A No, sono due contesti indipendenti.

B Sì, ma solo se la proprietà non è stata precedentemente impostata in un foglio di stile.

C Sì, usando *oggetto.style.proprietà*.

D Sì, accedendo al DOM del foglio di stile CSS.

16 **Indicare quali delle seguenti funzionalità sono presenti nella libreria *jQuery*.**

A Selezione di tutti i *tag* di un determinato tipo.

B Ridefinizione di una proprietà di un elemento della pagina.

C Associazione di una nuova funzione al verificarsi di un evento.

D Tutte le funzionalità precedenti.

17 **Usando la libreria *jQuery* come si seleziona un elemento con *id ID*?**

A `$("!ID").`

B `$("#ID").`

C `$("@ID").`

D `$(id='ID').`

18 **A cosa serve il metodo *val* della libreria *jQuery*?**

A Per recuperare il valore del colore dell'elemento a cui si applica.

B Per recuperare il valore del campo del *form* a cui si applica.

C Per recuperare il valore della posizione all'interno della pagina dell'elemento a cui si applica.

D Per recuperare il valore di un elemento del DOM.

19 **Se in uno *script* di codice JavaScript è presente un *loop* infinito:**

A è una situazione non possibile: qualunque *loop* termina sicuramente.

B il browser si blocca, a meno che non abbia controlli sul tempo di esecuzione del codice JavaScript, e propone all'utente di terminare lo *script* se la sua esecuzione è troppo lunga.

C il browser continua il *rendering* del resto della pagina e l'esecuzione di eventuali altri *script*, in pratica non ci si accorge che una parte del codice resta in esecuzione senza terminare.

D il browser segnala l'errore in fase di caricamento del codice se contenuto in un file esterno, o della pagina se definito al suo interno.

20 **Quale dei seguenti è il costruttore corretto per creare una mappa *Google Maps*?**

A `new google.maps.Map(proprietà).`

B `new google.maps.Map(proprietà1, proprietà2, ...).`

C `new google.maps.Map().`

D Sono tutte invocazioni valide del costruttore di una mappa *Google Maps*.

21 **Gli eventi gestiti dalle API *Google Maps*:**

A sono esattamente gli stessi del DOM.

B sono simili a quelli del DOM, ma estesi con informazioni specifiche e con eventi collegati ai controlli delle mappe.

C non esistono, in quanto le API *Google Maps* non permettono di gestire gli eventi.

D sono collegati esclusivamente a controlli delle mappe.

PROBLEMI

22 **Realizzare una pagina web comprendente un campo di testo e un pulsante. La pressione sul pulsante deve visualizzare un *alert* con il messaggio «Vuoto» se il campo di testo non contiene alcun valore, altrimenti con il valore del campo.**

23 **Codificare una funzione JavaScript che legga tutti i *tag* `<p>` e inserisca, prima del contenuto del paragrafo, un numero progressivo tra parentesi quadre (il contenuto del primo paragrafo sarà preceduto da «[1]», del secondo da «[2]» e così via).**

24 **Creare uno *script* JavaScript che collezioni tutti i *link* di una pagina web e crei un elenco in fondo alla pagina.**

25 **Utilizzando la libreria *jQuery* realizzare uno *script* che ricerchi tutte le tabelle presenti in una pagina web impostandone la classe CSS *Tabella*.**

26 **Realizzare una pagina web che, utilizzando *Google Maps*, visualizzi la mappa di un luogo e intercetti il click effettuato con il mouse dell'utente per visualizzare un nuovo marcatore nel punto specifico.**

27 **Modificare lo *script* della pagina web realizzata nell'esercizio precedente affinché, oltre al marcatore, compaia anche un fumetto contenente le informazioni relative alle coordinate geografiche (latitudine e longitudine).**

CASI PRATICI

28 Realizzare una pagina web che, utilizzando *Google Maps*, visualizzi la mappa di un luogo e un'area di testo. Inserirvi uno *script* JavaScript che reagisca al click del mouse sulla mappa visualizzando un nuovo marcatore nel punto specifico e, contestualmente, nell'area di testo il codice JavaScript di creazione del marcatore stesso. In questo modo l'utente ha la possibilità di copiare il codice e inserirlo nel codice della pagina web al fine di rendere il marcatore persistente.

29 Scattare 10 fotografie che rappresentino 10 diverse fasi di realizzazione di una torta. Realizzare una pagina web con due pulsanti in modo che il click del mouse sul primo visualizzi la fotografia della fase precedente, il click del mouse sul secondo visualizzi la fotografia della fase successiva.

30 Utilizzare le funzionalità delle API *Google Maps* per richiedere all'utente di una pagina web la descrizione di un luogo e visualizzare 8 mappe, tutte centrate sul luogo descritto, ma con diversi livelli di zoom.

PROGETTI

31 Realizzare un'applicazione web che permetta di giocare al gioco dell'impiccato nel contesto di un browser, prestando particolare attenzione ai seguenti aspetti:

- permettere la scelta casuale della parola da indovinare partendo da un dizionario;
- gestire la visualizzazione della parola in modo che inizialmente compaiano solo la prima e l'ultima lettera, per le lettere intermedie utilizzare il simbolo «_»;
- gestire l'interfaccia affinché l'utente possa scegliere una delle lettere dell'alfabeto da inserire nella parola da indovinare con una tastiera visualizzata nella pagina web: la selezione di una lettera ne deve modificare l'aspetto grafico, per esempio il colore di sfondo;
- mostrare l'immagine dell'impiccato aggiornata dopo ogni errore del giocatore.

B3

Strumenti di sviluppo per linguaggio JavaScript

Come per qualsiasi linguaggio di programmazione, anche per JavaScript è utile avere a disposizione strumenti di sviluppo che agevolano la creazione di applicazioni web. L'attività di individuazione e correzione degli errori è parte integrante del processo di sviluppo e, come tale, può essere supportata da strumenti specifici.

Con riferimento al linguaggio JavaScript, esistono due grandi categorie di strumenti:

- IDE (*Integrated Development Environment*), cioè ambienti integrati di codifica, compilazione, distribuzione e condivisione del codice sorgente con interfaccia utente di tipo grafico e visuale;
- strumenti integrati all'interno dei browser moderni, spesso usati nella fase di ricerca e correzione degli errori.

I primi sono orientati a sviluppare applicazioni web complete composte sia dalla componente client sia da quella server; rientrano in questa categoria *NetBeans* ed *Eclipse*.

Tutti i browser moderni hanno un loro ambiente di sviluppo integrato: nel seguito sarà analizzato quello presente all'interno di Google Chrome. Il ricorso a editor di testo avanzati in cui sono presenti solo poche e limitate caratteristiche – come la colorazione delle diverse componenti del codice sorgente – può risultare più immediato dell'uso di un vero e proprio IDE. Esempi di tali editor, in ambiente Windows, sono *UltraEdit* e *NotePad++*: utilizzando questi strumenti ha senso ricorrere agli strumenti integrati nei browser per il *debug* e il *tuning* del codice JavaScript.

Quale IDE per JavaScript?

Per creare semplici pagine web è possibile fare a meno di un IDE, ma se si sviluppano applicazioni complesse è indispensabile capire quale linguaggio utilizzare lato server per creare applicazioni evolute. Infatti un'applicazione solo lato client è di scarsa o nessuna utilità se non supportata da un'applicazione lato server che rende persistenti i dati o li condivide fra più utenti. Nel caso che lato server siano usati i linguaggi Java o PHP *NetBeans* è un'ottima scelta, così come *Eclipse*.

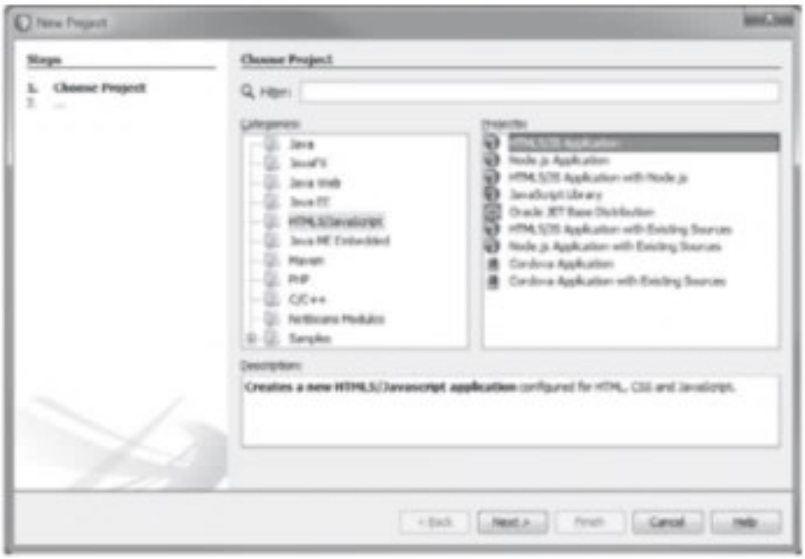
1 IDE *NetBeans* per *script* JavaScript

NetBeans rappresenta uno degli IDE storici per lo sviluppo di applicazioni Java che nel tempo ha integrato molti altri linguaggi di programmazione. Nel 2016 *NetBeans* è stato donato da Oracle alla Apache Software Foundation divenendo un progetto completamente *open-source*.

NetBeans permette lo sviluppo di applicazioni web lato client in HTML5, CSS e JavaScript. Per la parte server supporta i linguaggi Java, PHP e Node.js.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

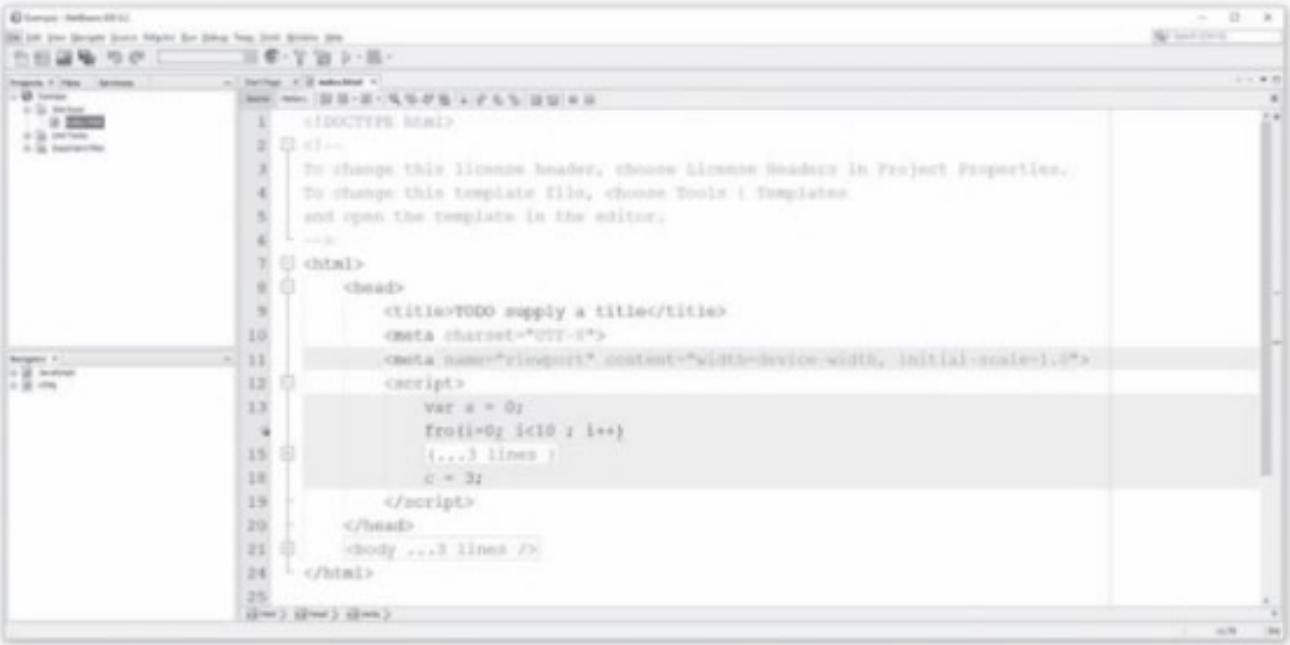
Per creare un nuovo progetto web in *NetBeans* è necessario usare il comando *New Project* dal menu *File*: per un progetto web la categoria è *HTML5/JavaScript* e il tipo di progetto *HTML5/JS Application*:



Nelle schede successive del *wizard* di creazione del progetto è possibile indicarne il nome e la cartella del progetto e l'eventuale *template* da usare, che risulta utile per creare una volta sola la struttura delle pagine web quando questa è comune. Una volta creato il progetto viene automaticamente generata una pagina iniziale, denominata *index.html*.

La figura che segue illustra alcune delle caratteristiche di *NetBeans*:

- l'evidenziazione degli errori di sintassi commessi in fase di digitazione;
- la colorazione del codice e l'individuazione dei *tag* HTML corrispondenti;
- la possibilità di collassare o espandere il codice compreso tra due *tag* HTML o tra le parentesi di un blocco JavaScript.



Utilizzando i comandi del menu *Run* si può verificare l'effetto dell'esecuzione della pagina: alla prima esecuzione viene richiesto di installare il com-

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

ponente *NetBeans connector* nel browser Chrome; utilizzando questo componente è possibile eseguire il *debug* dell'applicazione nel contesto dell'IDE *NetBeans*, ma utilizzando Chrome come ambiente di esecuzione degli *script*.

2 Strumenti per sviluppatori del browser Google Chrome

Per lo sviluppo in linguaggio JavaScript, in alternativa al ricorso a un IDE, è possibile impiegare gli strumenti integrati nei moderni browser che non richiedono alcuna installazione aggiuntiva.

ESEMPIO

Lo *script* seguente mostra un pulsante e una casella di testo non editabile: a ogni click sul pulsante dovrebbe incrementare di uno il numero visualizzato nella casella di testo.

```
<!doctype html>
<html>
  <head>
    <script>
      function incrementa() {
        var elem = document.getElementById("conta");
        a = a + 1;
        elem.value = a
      }
    </script>
  </head>
  <body>
      <input type="button" value="Incrementa"
             onclick="incrementa()" />
      <input type="text" disabled="disabled" id="count" />
  </body>
</html>
```

Ma quando viene caricata la pagina il click sul pulsante non produce alcun effetto: il codice JavaScript in presenza di errori interrompe il proprio funzionamento, ma non segnala nulla nella pagina visualizzata.

Gli strumenti per sviluppatori di Google Chrome (*Chrome Developer Tools* o *Chrome DevTools*) sono integrati nel browser Chrome per la profilazione, il *debug* e la manutenzione di *script* in linguaggio JavaScript inseriti nelle pagine HTML.

In particolare i *Chrome DevTools* permettono di:

- simulare le dimensioni ridotte degli schermi dei dispositivi mobili;
- analizzare gli elementi del codice HTML/CSS di una pagina web;
- visualizzare i messaggi diagnostici ed effettuare il *debug* del codice JavaScript;
- analizzare le prestazioni di rete e di esecuzione del codice da parte del browser;
- monitorare l'uso della memoria e delle risorse in genere;

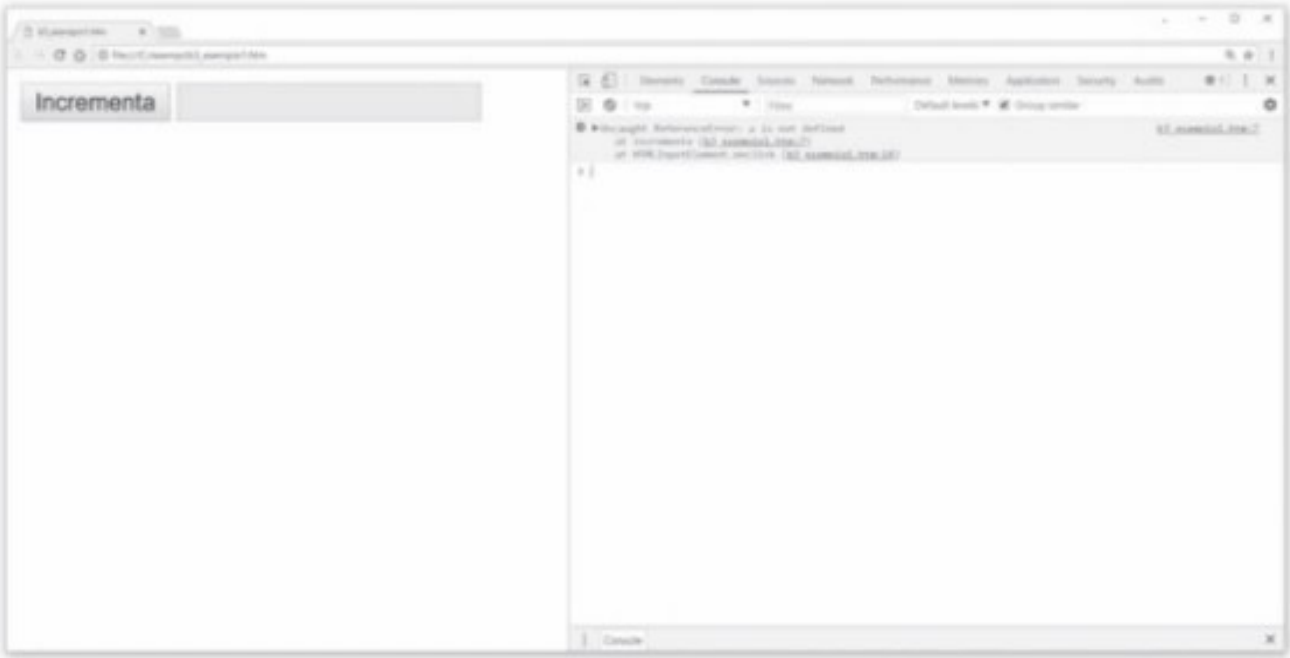
Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

- analizzare i rischi per la sicurezza del codice delle pagine web.

Per accedere a questi strumenti si utilizza il pulsante *Chrome menu* posizionato in alto a destra nel browser scegliendo la voce *Altri Strumenti* e infine *Strumenti per sviluppatori*: gli strumenti sono visualizzati sulla parte destra del browser.

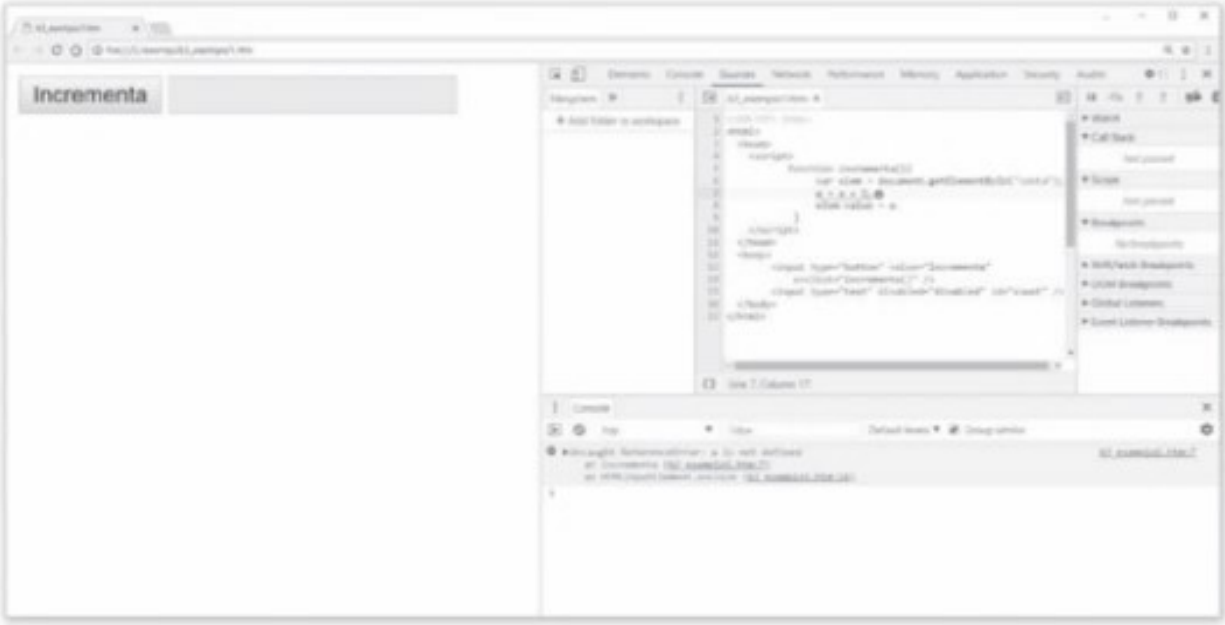
2.1 Strumento *Console*

Tra i diversi strumenti resi disponibili è possibile accedere alla *Console* in cui sono visualizzati eventuali errori presenti nella pagina corrente. Con riferimento all'esempio precedente risulta subito chiara l'origine del problema: la variabile *a* non è stata inizializzata anche se viene usata nella parte destra di un'istruzione di assegnamento:



Nella *Console* l'informazione relativa al problema è priva di contesto: con un click sul collegamento ipertestuale con cui è visualizzato l'errore viene aperta la scheda *Sources* con evidenziata la riga ove è presente l'errore.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018



2.2 Strumento *Sources*

L'editor integrato nella scheda *Sources* consente di modificare il codice dello *script* inserito in una pagina web.

ESEMPIO L'inserimento della dichiarazione della variabile *a* e la sua contestuale inizializzazione risolve il problema riscontrato nell'esempio precedente:

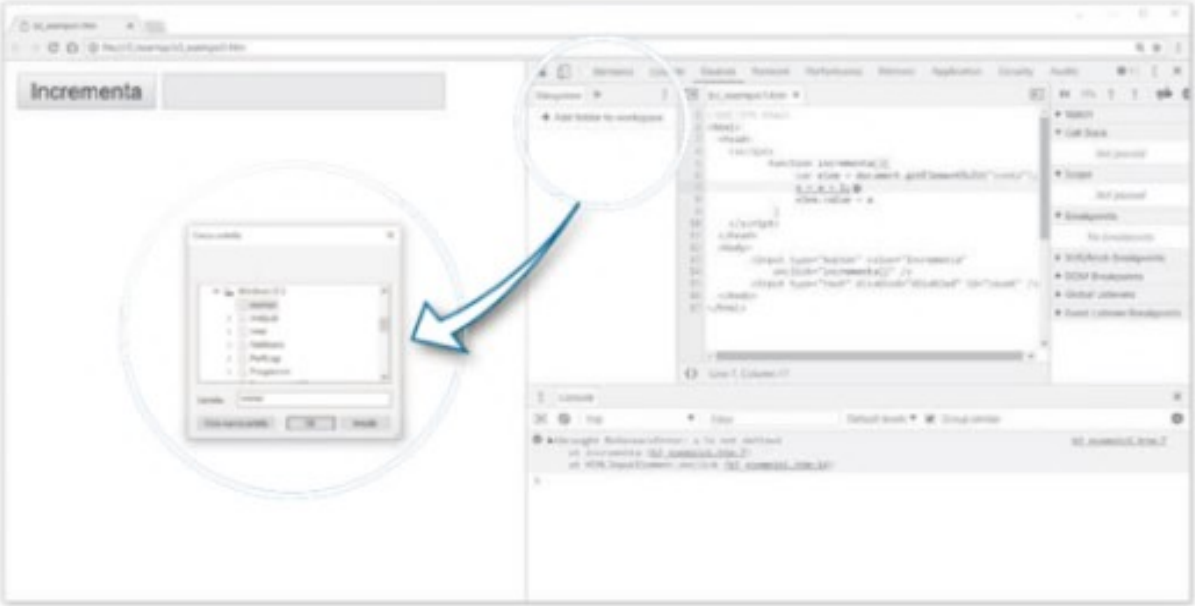
```
<!doctype html>
<html>
  <head>
    <script>
      var a = 0;
      function incrementa( ) {
        var elem = document.getElementById("conta");
        a = a + 1;
        elem.value = a
      }
    </script>
  </head>
  <body>
      <input type="button" value="Incrementa"
             onclick="incrementa()" />
      <input type="text" disabled="disabled" id="count" />
  </body>
</html>
```

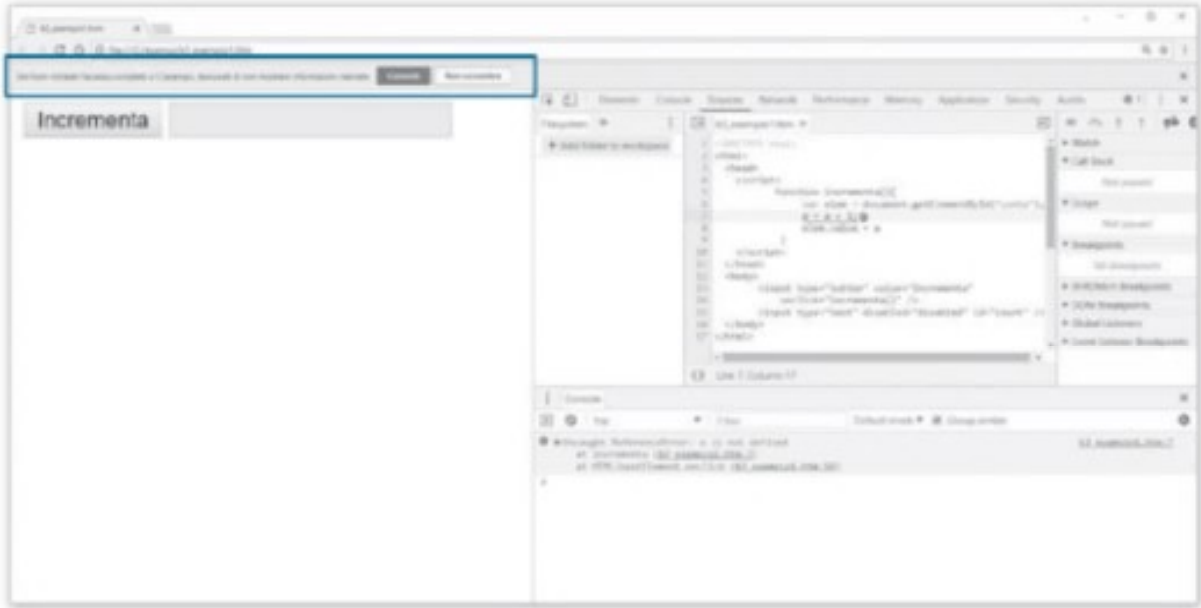
2.3 Editor integrato in Chrome *DevTools*

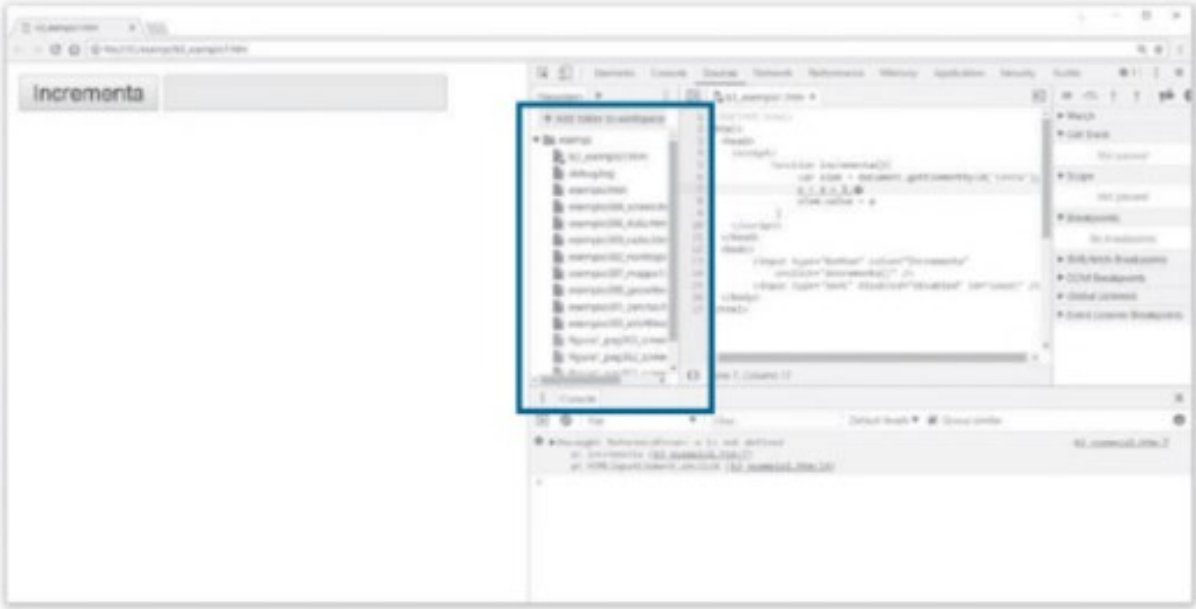
Per modificare il codice sorgente contenuto in una pagina HTML è possibile modificare il file sorgente direttamente nella scheda *Sources* del browser Chrome: è infatti possibile selezionare *Filesystem* e, per aggiungere la cartella che contiene i file HTML su cui si sta lavorando, fare click su *Add folder to workspace* (il browser richiede all'utente conferma del permesso di accesso in scrittura su disco). Dopo questa operazione è possibile inter-

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

venire sul codice direttamente nell'editor di Chrome *DevTools* e salvare le modifiche, senza la necessità di aprire un editor separato:



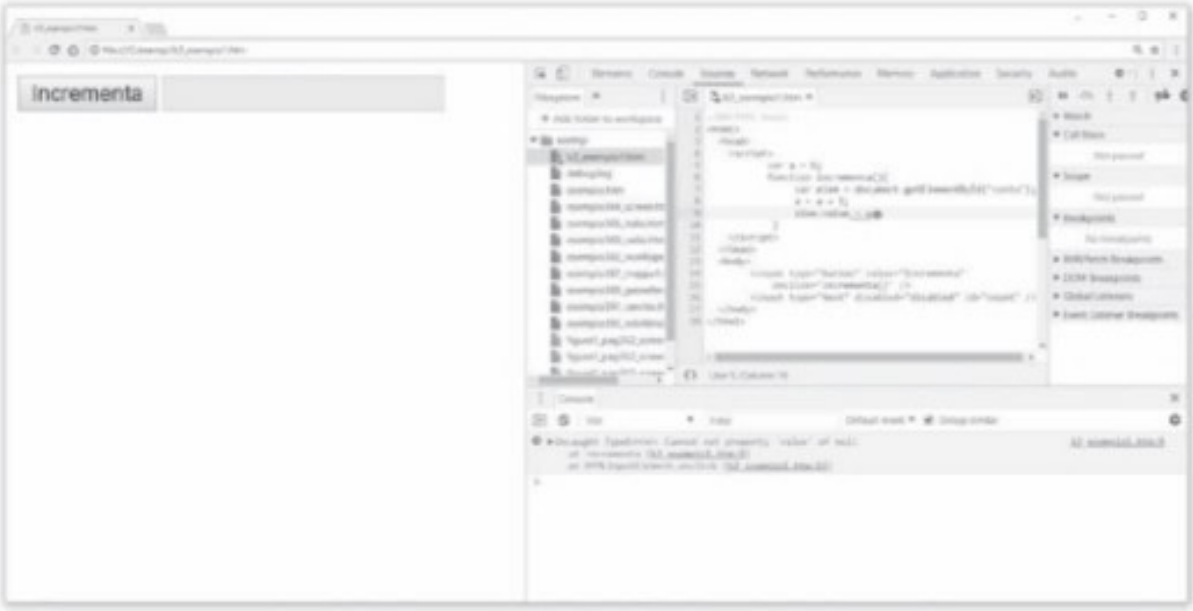




Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

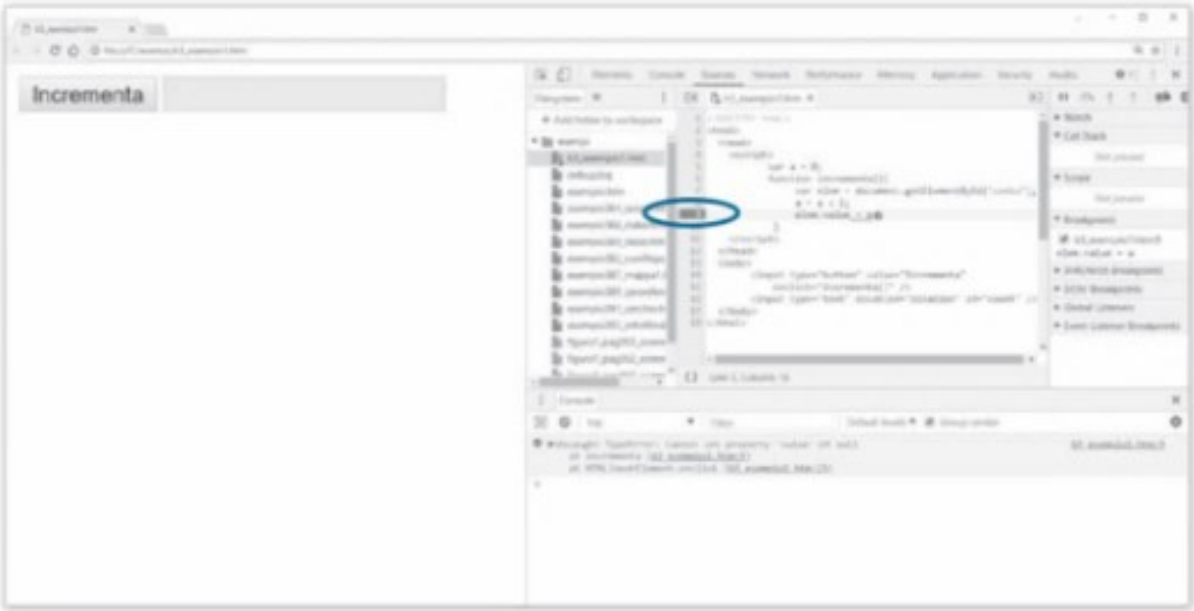
2.4 *Debug* del codice utilizzando *break-point*

Con riferimento all'esempio precedente il caricamento della pagina successivamente alla modifica del codice dello *script* evidenzia che il comportamento non è quello atteso. L'analisi del contenuto della *Console* permette di verificare che la variabile *elem* è `null`:



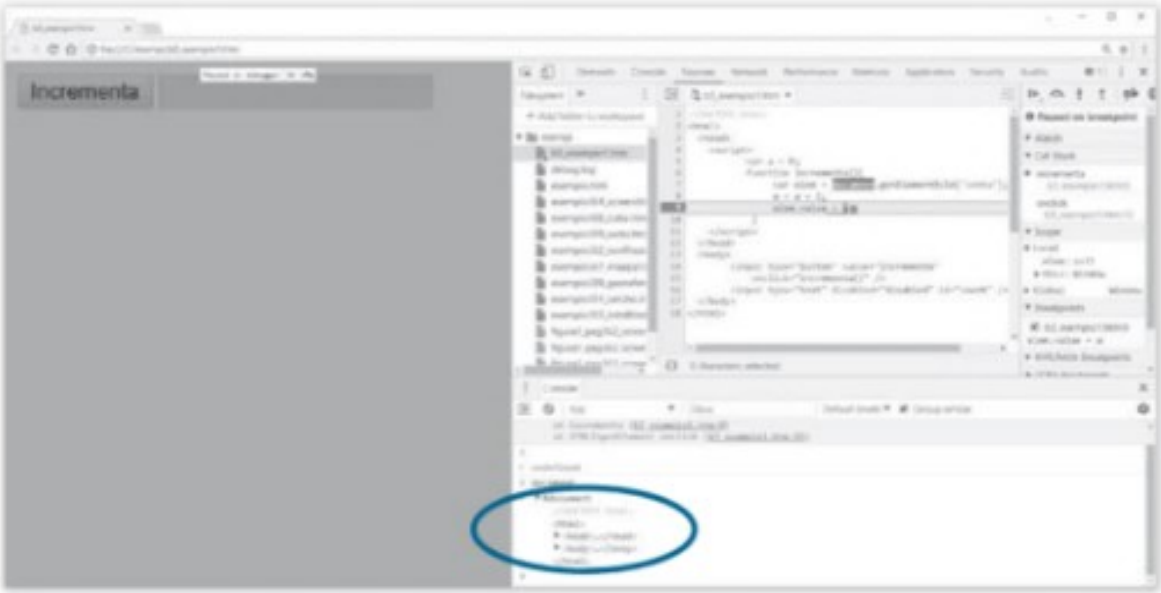
L'analisi del codice dello *script* visualizzato nella scheda *Sources* non evidenzia problemi per cui si deve effettuare una sessione di *debug* vera e propria.

Per sospendere l'esecuzione dello *script* a una determinata istruzione è possibile impostare un punto di interruzione (*break-point*). Nell'esempio di riferimento è utile impostarlo in corrispondenza dell'istruzione che recupera l'elemento in base all'*id* e lo assegna alla variabile *elem*: a questo scopo è sufficiente fare click a destra della linea di codice dove apparirà un marcatore:



Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

Ricaricando la pagina web e facendo click sul pulsante il codice della funzione *incrementa* viene eseguito, ma l'esecuzione dello *script* si sospende in corrispondenza del punto di interruzione impostato: si tratta di capire come mai l'istruzione fallisca. Un dubbio potrebbe essere relativo al fatto che l'oggetto *document* non rappresenti correttamente la pagina caricata: selezionando *document* nel codice JavaScript il click con il pulsante destro del mouse apre un menu contestuale che comprende la voce *Evaluate in console* la cui attivazione apre la *Console* con l'oggetto *document* navigabile, cosa che permette di verificare che il documento è rappresentato correttamente:



2.5 Ispezionare un elemento della pagina

Facendo click con il pulsante destro del mouse su un qualsiasi elemento della pagina web è possibile selezionare la voce *Ispeziona*; nel caso dell'esempio, con un click sulla casella di testo si ottengono due effetti: la visualizzazione di un riquadro di sintesi nella pagina web stessa e l'apertura di una scheda *Elements* nell'area dei *DevTools* che consente l'ispezione dell'elemento della pagina:

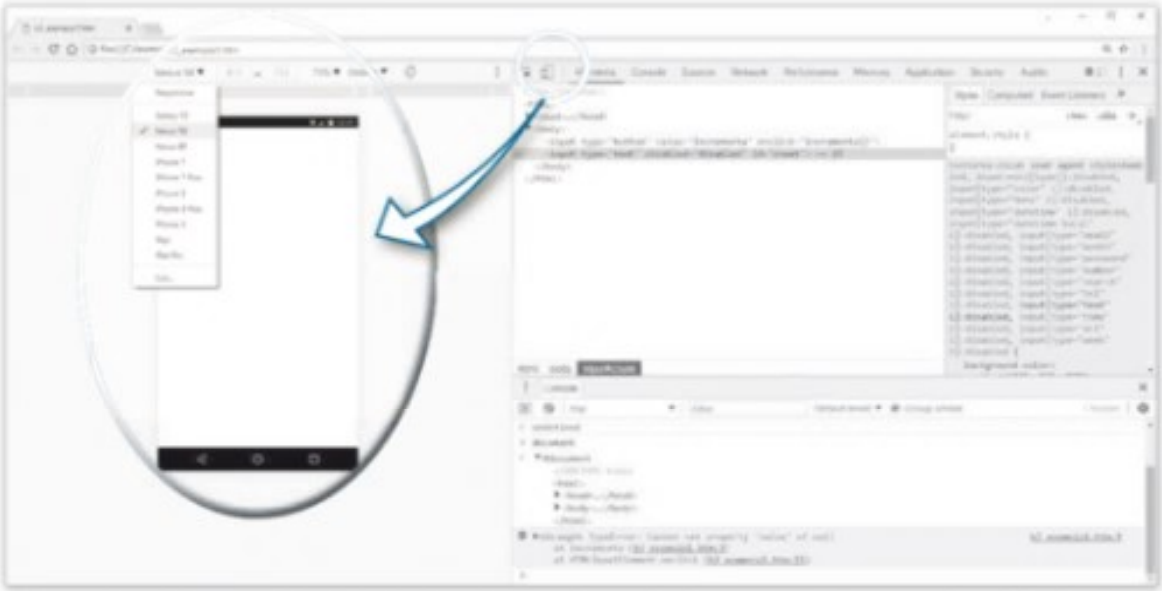
Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

L'ispezione dell'elemento permette di verificare che l'*id* è *count*, mentre lo *script* invoca *getElementById* con parametro *conta*: la correzione di questo errore rende lo *script* funzionante.

2.6 Non solo gestione degli errori

Uno degli aspetti critici nello sviluppo di pagine web per dispositivi mobili è capire come esse vengono visualizzate su schermi di diverse tipologie e dimensioni. Nei Chrome *DevTools* è disponibile un'opzione per variare la dimensione della finestra, avendo alcuni modelli di dispositivi mobili come riferimento.

Il pulsante in alto a sinistra *Toogle device toolbar* permette di mostrare la pagina web all'interno di un riquadro ridotto che può simulare per dimensioni uno dei modelli di *smartphone* più diffusi; nella modalità *Responsive* si possono impostare, attraverso la valorizzazione di due caselle di testo, le dimensioni personalizzate della finestra di visualizzazione della pagina:



Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

I CONCETTI CHIAVE

SVILUPPO DI APPLICAZIONI WEB. Lo sviluppo di applicazioni web complesse richiede strumenti adeguati per poter gestire sia il lato server sia il lato client.

NETBEANS. *NetBeans* è un IDE evoluto che permette una gestione di progetti HTML5 e JavaScript.

INTEGRAZIONE CON GOOGLE CHROME. *NetBeans* possiede un componente connettore che, installato in Chrome, integra i due strumenti.

CHROME *DEVTOOLS.* Chrome dispone di numerosi strumenti integrati utili per il *debug* delle applicazioni JavaScript; tali strumenti prendono il nome di Chrome *DevTools*.

STRUMENTO *CONSOLE.* Lo strumento *Console* permette di verificare la presenza di errori nel codice.

SALVATAGGIO DELLE MODIFICHE. È possibile modificare i sorgenti delle pagine direttamente nell'editor di Chrome *DevTools*.

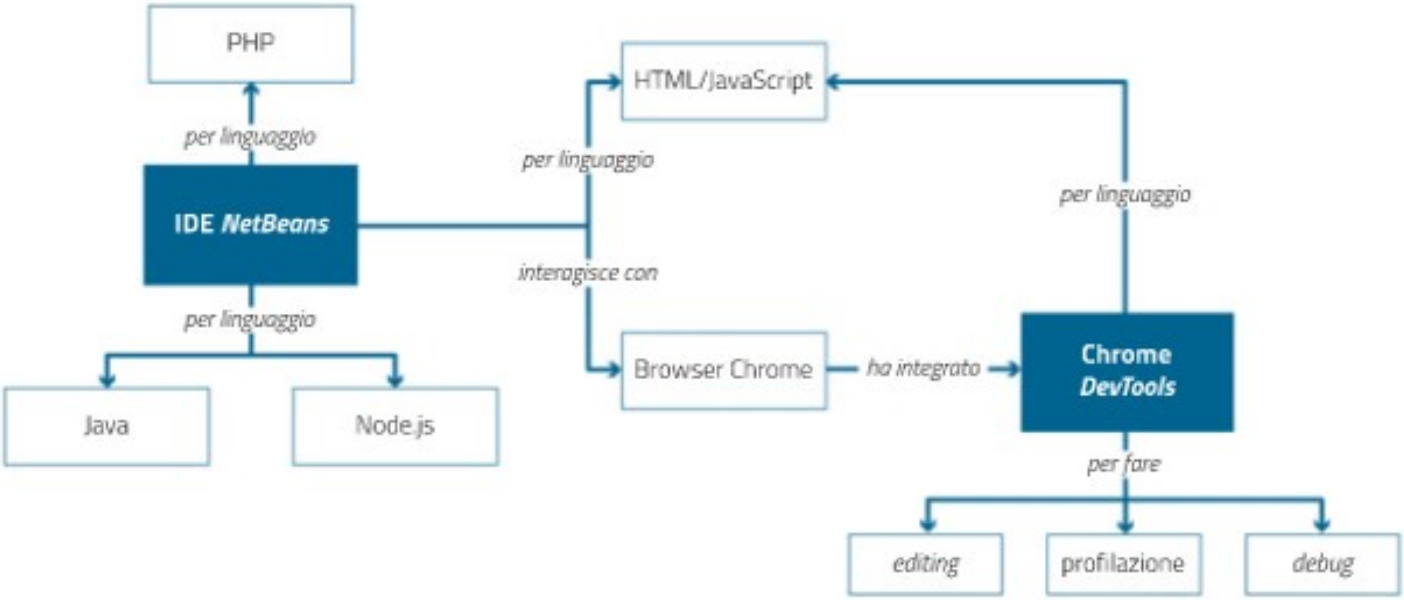
PUNTI DI INTERRUZIONE. È possibile sospendere l'esecuzione di uno *script* impostando uno o più punti di interruzione.

STRUMENTO *ELEMENTS.* Questo strumento permette di ispezionare uno qualunque degli elementi della pagina HTML.

SVILUPPO PER DISPOSITIVI MOBILI. Con Chrome *DevTools* è possibile verificare la visualizzazione delle pagine web su un dispositivo mobile.

Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018

RIPASSA CON LA MAPPA



Questo file costituisce un'estensione online dei corsi di Formichi, Meini, Venuti © Zanichelli editore 2018