

HTML5

CON CSS E JAVASCRIPT



Daniele Bochicchio, Stefano Mostarda

 **aspitalia.com**
The ASP.NET Community

HOEPLI
INFORMATICA

HTML5

con CSS e JavaScript

Daniele Bochicchio Stefano Mostarda

HTML5

con CSS e JavaScript



EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2015

via Hoepli 5, 20121 Milano (Italy)

tel. +39 02 864871 – fax +39 02 8052886

e-mail hoepli@hoepli.it

www.hoepli.it

Seguici su Twitter: [@Hoepli_1870](https://twitter.com/Hoepli_1870)

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

ISBN EBOOK 978-88-203-6983-5

Progetto editoriale: Maurizio Vedovati – Servizi editoriali
Impaginazione e copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Contenuti del libro

A chi si rivolge questo libro
Convenzioni
Materiale di supporto ed esempi
Requisiti software per gli esempi
Contatti con l'editore
Contatti e domande agli autori

ASPItalia.com Network

Gli autori

Ringraziamenti

Capitolo 1: Introduzione a HTML5

Nascita di HTML5
Un po' di storia
 Gli albori: da HTML 1 a HTML 3
 La prima infanzia: da HTML4 a XHTML e ritorno
 La nascita di HTML5
Anatomia di una pagina HTML
Le novità di HTML5
 I nuovi tag HTML
 Le novità dei CSS
 Le novità di JavaScript
Un primo esempio di pagina HTML5
Browser detection o feature detection?
 Dispositivi mobili
Conclusioni

Capitolo 2: Nuovi elementi del markup

Gli elementi supportati da HTML5

Gli elementi non più supportati da HTML5

I nuovi elementi di formattazione della pagina

- L'elemento section

- L'elemento hgroup

- L'elemento article

- L'elemento aside

- L'elemento nav

- L'elemento header

- L'elemento footer

- Gli elementi figure e figcaption

I nuovi elementi di formattazione del testo

- L'elemento mark

- Gli elementi ruby, rt e rp

- L'elemento time

- L'elemento wbr

I nuovi elementi interattivi

- Gli elementi details e summary

- L'elemento menu

I nuovi attributi

ARIA e gli attributi per l'accessibilità

- L'attributo tabindex

- L'attributo role

- Gli stati e le proprietà ARIA

- Le live region

Conclusioni

Capitolo 3: Le form

Principi di form

- Inviare il contenuto

- Gestire scelte multiple

- Elenchi a discesa

- Inserire testo su più righe

Uso avanzato delle form

- Inserire numeri di telefono, e-mail e URL

- Inserire numeri e orari

- Un campo specifico per la ricerca

- Altri elementi

- Mostrare un segnaposto nei campi
- Gestire l'autocomplete
- Uso del tag datalist
- Focus automatico sugli elementi

Validazione delle form

- Rendere obbligatorio un campo
- Specificare pattern complessi
- Indicare un messaggio di errore
- Bloccare la validazione
- Le novità associate alla form

Conclusioni

Capitolo 4: Introduzione ai CSS

Basi di CSS

- Gestire la registrazione dei CSS

Applicare gli stili a un documento

- Stile per i tag
- Stile con classe
- Stile con ID

I selettori CSS

- Le pseudo-classi
- Gli pseudo-elementi
- Il selettore :not
- I selettori di attributi
- I selettori combinati
- I selettori di gerarchia

Gestire il posizionamento

- Gestire la disposizione degli elementi
- Gestire il posizionamento assoluto

Conclusioni

Capitolo 5: Layout ed effetti con CSS3

Il box model e la gestione di margini, bordi e spazio

- Gestire il bordo
- Gestire il padding
- Gestire il margine
- Capire il box model

Personalizzare il box model con il box sizing

Outline

Capire la disposizione di un elemento

Impostare la proprietà CSS visibility

Impostare la proprietà CSS display

Gestire il cursore

Le media query

Utilizzare un CSS in base al device

Registrare un file CSS in base al device

Utilizzare un CSS in base alle caratteristiche del device

Applicare trasformazioni agli elementi

Gestire le transizioni

Combinare trasformazioni e transizioni

Conclusioni

Capitolo 6: Gestione tipografica

Gestione del testo

Impostazione del font

Impostazione della dimensione del carattere

Impostazione del colore

Applicare effetti al testo

Gestione del layout di pagina

Sfondi e bordi

Conclusioni

Capitolo 7: Introduzione a JavaScript

Hello World

Tipi di base

Dichiarazione di variabile

Operatori

Istruzioni di selezione

Istruzione if

Istruzione switch

Istruzioni di iterazione

Istruzione for

Istruzione while

Istruzione do ... while

- Istruzioni di salto
- Funzioni e procedure
- Array
- Classi e oggetti
 - Lavorare con gli oggetti
- Conclusioni

Capitolo 8: JavaScript avanzato

- Capire il Document Object Model (DOM)
- Utilizzare i selettori per navigare il DOM
 - Utilizzare i selettori CSS per navigare il DOM
- Manipolare gli oggetti del DOM
- Gestire gli eventi degli oggetti
- AJAX con XMLHttpRequest
 - Gestione dello stato di avanzamento
 - Trasferimento di file
 - Chiamate cross-domain
- History API
- Geolocalizzazione
- Web Storage
- WebSocket
- ... e molto altro ancora
- Conclusioni

Capitolo 9: Audio e video

- I tag audio e video
- Definire più sorgenti multimediali
- Controllare la riproduzione da JavaScript
 - La gestione dello stato e del caricamento
 - Controllare la riproduzione
 - Utilizzare player multimediali già pronti
- Applicare sottotitoli e traduzioni
- Combinare i CSS con gli elementi multimediali
- Le limitazioni dei tag audio e video
- Conclusioni

Capitolo 10: Grafici con Canvas e SVG

Grafica vettoriale con SVG

- Disegnare con gli shape
- Inserire elementi testuali
- Raggruppare e riutilizzare gli elementi
- Colorare con gradienti e pattern
- Trasformare gli elementi
- Applicare i CSS
- Manipolazione DOM e scripting

Grafica bitmap con il canvas

- Implementare il game loop: la logica di un gioco
- Le API di disegno
- Disegnare con il path
- Scrivere testo sulla bitmap

Conclusioni

Capitolo 11: I Framework per il Web

jQuery

- Effettuare ricerche nel DOM
- Manipolare gli oggetti
- Gestire gli eventi
- Effettuare chiamate AJAX
- jQueryUI

Bootstrap

- Creare un layout
- Creare una form con Bootstrap
- Altre funzionalità

Generare grafici

Templating e binding con KnockoutJS

Single Page Application

Conclusioni

Informazioni sul Libro

Contenuti del libro

HTML5 è un insieme di nuovi standard, che è stato approvato nel corso del 2014 e accompagna da qualche anno gli sviluppatori web. Porta con sé diverse novità, che vanno verso l'adozione di una serie di **specifiche standard**, condivise da browser e piattaforme diverse, che semplificano la vita degli sviluppatori, sempre alle prese con le tante difficoltà legate ai vari browser, e consentono di implementare più facilmente in nativo nei browser una serie di funzionalità che in passato necessitavano di plug-in esterni.

In realtà, HTML5 è molto di più che un aggiornamento dell'HTML e include una serie di nuove specifiche, che abbracciano anche **CSS** e **JavaScript**: districarsi tra le nuove specifiche potrebbe non risultare un'impresa semplice.

Con uno stile chiaro, pratico e ricco di esempi, questo libro si pone come una guida ideale, sia per chi inizia a sviluppare sia per chi vuole conoscere tutte le novità di HTML5 e delle tecnologie correlate.

Le tecniche introdotte nel libro sono frutto dell'esperienza del lavoro degli autori, tecnici e formatori che seguono lo sviluppo web sin dagli albori.

Daniele Bochicchio e Stefano Mostarda fanno parte di ASPIItalia.com Network, storica community all'interno della quale trova posto HTML5Italia.com, un nuovo sito interamente dedicato ai temi di HTML5.

Il libro è suddiviso in cinque parti, organizzate in modo da trattare al meglio i contenuti di tipo omogeneo.

La prima parte (Fondamenti di HTML5) introduce i fondamenti di HTML5, partendo dalle motivazioni storiche che hanno segnato l'introduzione di HTML, arrivando ad analizzare tutte le novità dal punto di vista del markup e soffermandosi in modo particolare sulle nuove caratteristiche legate alle form.

Con la seconda parte (Layout e aspetto grafico con CSS3), nell'arco di tre capitoli vengono analizzate le basi di CSS e le novità della terza versione, partendo dai fondamentali e arrivando a trattare tutte le nuove funzionalità.

Nella terza parte (JavaScript con HTML5), dedicata a come integrare JavaScript nelle proprie applicazioni, il manuale arriva fino ad analizzare molte delle API che ruotano attorno a HTML5 (geolocalizzazione, storage

API, web socket ecc.).

L'ultima parte (Scenari avanzati), che sposta l'accento sulle caratteristiche delle specifiche che consentono di aggiungere un tocco multimediale alle applicazioni. In questa parte trovano spazio SVG, Canvas e gestione nativa di audio e video, oltre che un ultimo capitolo dedicato ai framework che rendono più produttivo lo sviluppatore web.

Tutti i capitoli contengono molti esempi pratici, che mettono in luce l'immediatezza di utilizzo di HTML5, con un occhio di riguardo all'integrazione e al supporto anche da parte di browser non ancora allineati per quanto concerne il supporto allo standard.

A chi si rivolge questo libro

I contenuti di questo libro sono pensati per **sviluppatori** e **interactive designer** che lavorano con il Web. Idealmente, ciascuna delle parti di cui è composto può essere letta in maniera separata, così da poter approfondire ciascuno degli specifici aspetti che vengono trattati.

I contenuti del libro sono neutri rispetto a piattaforme server o client, perché si soffermano sulle specifiche che ruotano attorno a HTML5.

Il libro è indicato sia per chi è totalmente digiuno di HTML sia per chi è interessato ad approfondire tutte le novità di questa nuova versione.

Convenzioni

All'interno di questo volume abbiamo utilizzato stili differenti secondo il significato del testo, così da rendere più netta la distinzione tra tipologie di contenuti diversi.

I termini importanti sono spesso indicati in **grassetto**, così da essere più facilmente riconoscibili.

nota

Il testo contenuto nelle note è scritto in questo formato. Le note contengono informazioni aggiuntive relativamente a un argomento o ad aspetti particolari ai quali vogliamo dare una certa rilevanza.

Gli esempi contenenti codice o markup sono rappresentati secondo lo schema

riportato qui di seguito. Ciascun esempio è numerato in modo da poter essere referenziato più facilmente nel testo e recuperato negli esempi a corredo.

Esempio 1.1 - Linguaggio

Codice

Codice importante, su cui si vuole porre l'accento

Altro codice

Per namespace, classi, proprietà, metodi ed eventi viene utilizzato questo font.

Per attirare l'attenzione su uno di questi elementi, per esempio perché è la prima volta che viene menzionato, lo stile è **questo**.

Materiale di supporto ed esempi

Questo libro include una nutrita quantità di esempi, che riprendono sia gli argomenti trattati sia quelli non approfonditi. Il codice presentato nei capitoli può essere scaricato all'indirizzo <http://books.asptalia.com/HTML5/>, dove saranno anche disponibili gli aggiornamenti e il materiale collegato al libro.

Requisiti software per gli esempi

Questo è un libro dedicato a HTML5. Con l'eccezione di casi particolari, comunque evidenziati, per visionare e testare gli esempi è sufficiente un browser.

Per lo sviluppo, abbiamo generalmente utilizzato un editor di testo generico, come il *Notepad* di Windows.

I principali tool di sviluppo sul mercato supportano HTML5 nelle ultime versioni: vi consigliamo comunque di verificare che quello di vostra preferenza sia in grado di farlo.

Contatti con l'editore

Per qualsiasi necessità, potete contattare direttamente l'editore attraverso il sito www.hoeplieditore.it/.

Contatti e domande agli autori

Per rendere più agevole il contatto con gli autori, abbiamo predisposto un *forum* specifico, raggiungibile all'indirizzo <http://forum.aspitalia.com/>, in cui saremo a vostra disposizione per chiarimenti, approfondimenti e domande legate al libro.

Potete partecipare, previa registrazione gratuita, alla community di [ASPItalia.com](http://aspitalia.com) Network, di cui fa parte anche [HTML5Italia.com](http://html5italia.com), dedicato in maniera specifica ai temi dello sviluppo di soluzioni basate su HTML5. Vi aspettiamo!

Detto questo, non ci resta che augurarvi una buona e proficua lettura.

ASPItalia.com Network

[ASPItalia.com](#) Network, nata dalla passione dello staff per la tecnologia, è supportata da oltre quindici anni di esperienza con [ASPItalia.com](#) per garantirvi lo stesso livello di approfondimento, aggiornamento e qualità dei contenuti su tutte le tecnologie di sviluppo del mondo Microsoft.

Con oltre 65.000 iscritti alla community, i forum rappresentano il miglior luogo in cui porre le vostre domande riguardanti tutti gli argomenti trattati!



[ASPItalia.com](#) si occupa principalmente di tecnologie dedicate al Web, da [ASP.NET](#) a IIS, con un'aggiornata e nutrita serie di contenuti pubblicati nei dieci anni di attività, che spaziano da ASP a Windows Server, passando per security e XML. Il network comprende:

- [HTML5Italia.com](#) con HTML5, CSS3, ECMAScript 5 e tutto quello che ruota intorno agli standard web per costruire applicazioni che sfruttino al massimo il client e le specifiche web.
- [LINQItalia.com](#), con le sue pubblicazioni, approfondisce tutti gli aspetti di LINQ, passando per i vari flavour LINQ to SQL, LINQ to Objects, LINQ to XML oltre a Entity Framework.
- [SilverlightItalia.com](#) pubblica script, risorse, tutorial e articoli dedicati alla tecnologia di Microsoft per la creazione di RIA (Rich Internet Application).
- [WindowsAzureItalia.com](#), che si occupa di tutto quello che è cloud e Microsoft Azure.
- [WinFXItalia.com](#), in cui sono presenti contenuti su tutte le tecnologie legate allo sviluppo per Windows e il .NET Framework.
- [WinPhoneItalia.com](#) è un sito completamente dedicato a Windows Phone e allo sviluppo di applicazioni mobili su piattaforma Microsoft.
- [WinRTItalia.com](#) copre gli aspetti legati alla creazione di applicazioni per Windows, dall'UX fino allo sviluppo.



Daniele Bochicchio

E-mail: daniele@aspitalia.com

Twitter: <http://twitter.com/dbochicchio>

Blog: <http://blogs.aspitalia.com/daniele/>

Daniele Bochicchio è **Chief Digital Officer** in **iCubed** (<http://www.icubed.it>), dove segue progetti legati al Web e al cloud, al mobile e a Windows e Windows Phone. Daniele ha una passione per lo sviluppo web e mobile e si è occupato, sin dalle primissime versioni, di **ASP.NET**, XAML, Windows Phone, Windows, Azure e HTML5. Nel 1998 ha ideato e sviluppato **ASPItalia.com**, di cui coordina ancora le attività.

È **Microsoft Regional Director** per l'Italia, un ruolo che fa da tramite fra le community e Microsoft stessa. È inoltre **Microsoft MVP per ASP.NET dal 2002** e ideatore dei Community Days, la conferenza di riferimento in Italia per le piattaforme Microsoft.

Potete incontrarlo abitualmente ai più importanti eventi e alle conferenze tecniche italiane e internazionali.



Stefano Mostarda

E-mail: stefano@aspitalia.com

Twitter: <http://twitter.com/sm15455>

Blog: <http://blogs.aspitalia.com/sm15455/>

Stefano Mostarda è **Senior Service Architect** presso **Soluzioni4D** dove si occupa di progettazione e sviluppo di applicazioni che vanno dalla piattaforma web a quella Windows passando anche per il cloud. Da sempre appassionato di sviluppo, Stefano segue continuamente le evoluzioni in questo campo, passando per **ASP.NET**, HTML5, Windows e Windows Phone.

Dal 2004 è membro dello staff del network **ASPItalia.com** e content manager del sito **LINQItalia.com** dedicato all'accesso e alla fruibilità dei dati. **ASP.NET** MVP, è autore di diversi libri di questa collana e di altri volumi in lingua inglese, sempre dedicati allo sviluppo .NET, oltre che speaker nelle maggiori conferenze italiane.

Ringraziamenti

Daniele ringrazia ancora una volta Noemi, Alessio e Matteo (che sono la sua guida e la sua gioia), tutta la sua famiglia, il team editoriale Hoepli e Stefano, con cui ormai collabora da oltre dieci anni e che, nel tempo, è diventato anche uno dei suoi più grandi amici. Un ringraziamento va anche ai suoi soci in iCubed, che gli lasciano il tempo per dedicarsi a queste attività.

Stefano ringrazia tutta la sua famiglia e i suoi amici per il loro supporto. Un ringraziamento particolare va a Daniele, suo compagno in quest'avventura: "grazie Amico mio, di tutto". Infine, Stefano vuole ringraziare tutti quelli che gli sono stati vicini in un anno molto difficile. Questo libro è anche per loro.

Daniele e Stefano tengono particolarmente a ringraziare la community di ASPIItalia.com Network, a cui anche quest'ultimo libro è, come sempre, idealmente dedicato!

Un grazie anche a Sir Tim Berners-Lee (l'ideatore del World Wide Web), perché è grazie a lui che oggi possiamo avere il Web, grande veicolo di condivisione e conoscenza.

Grazie anche a Matteo Casati, Cristian Civera e Riccardo Golia, autori della prima edizione di questo libro.

Introduzione a HTML5

HTML5, acronimo di **HyperText Markup Language**, è ben più che la quinta versione del linguaggio di markup con cui sono costruite le pagine web. Sebbene all'inizio HTML fosse stato pensato per rappresentare principalmente documenti a sfondo scientifico, si è evoluto nel corso degli anni fino ai giorni nostri ed è diventato il linguaggio ufficiale del World Wide Web.

HTML5 rappresenta solo l'ultimo passo di un'evoluzione che ha visto il linguaggio HTML cambiare nel corso del tempo, per adattarsi alle sempre nuove esigenze di comunicazione e pubblicazione all'interno di Internet, arrivando a includere al proprio interno specifiche non strettamente collegate al markup, ma anche alla definizione degli stili (CSS), di un linguaggio client side (ECMAScript, nella sua implementazione più famosa, JavaScript) e protocolli.

Al momento della pubblicazione di questo libro, HTML5 è stato rilasciato come uno standard definitivo da parte del W3C (acronimo di **World Wide Web Consortium**). Pur essendo fresche di ratifica, in realtà le specifiche sono già supportate da diversi anni (fin dalla loro prima introduzione) da tutti i browser presenti sul mercato. Questo rende utilizzabile HTML5 (con alcuni distinguo) praticamente già da subito e con un'altissima compatibilità in fatto di supporto disponibile.

nota

Il W3C (World Wide Web Consortium) è un'organizzazione internazionale che sovrintende alla definizione delle specifiche, dei protocolli e delle linee guida che riguardano il World Wide Web. Nato nell'ottobre del 1994, oggi conta tra le sue fila più di 300 membri provenienti da tutto il mondo, tra cui Adobe, Apple, CERN, Google, IBM, Mozilla Foundation, Microsoft, Opera Software, Oracle, alcune università e un nutrito numero di produttori di

Nascita di HTML5

Un primo obiettivo del gruppo di lavoro nella stesura delle specifiche di HTML5 è stato quello di proporre una serie di nuove caratteristiche in grado di estendere in modo mirato le possibilità offerte dalla versione precedente del linguaggio, ovvero HTML4. Abbandonata l'idea di prendere una strada differente, iniziata con l'introduzione di XHTML, che non ha mai avuto una reale adozione nel mercato, perché molto rigido e non improntato a rendere più moderno il Web, ciò che ha mosso il W3C e i suoi membri a sviluppare HTML5 è stata la necessità di fornire direttamente quelle funzionalità che, in precedenza, erano ottenibili tramite l'impiego di estensioni proprietarie all'interno dei browser come, per esempio, Adobe Flash o affini. Un secondo obiettivo del gruppo di lavoro è stato quello di garantire una maggiore compatibilità tra i diversi browser, indipendentemente dalle piattaforme software utilizzate e dalle differenti tipologie di dispositivi presenti sul mercato, in particolare quelli **mobile**. In quest'ottica, HTML5 favorisce ulteriormente il disaccoppiamento tra la struttura delle pagine, definita dal markup, e la loro rappresentazione, gestita tramite gli stili CSS. Questo aspetto non solo garantisce una maggiore standardizzazione nella visualizzazione delle pagine da parte dei browser, ma anche un miglior approccio in fase realizzativa da parte dei designer e dei programmatori di applicazioni e siti web.

I contenuti presenti in questo libro si basano sulla versione chiamata Recommendation (cioè sulle specifiche finali) del 28 ottobre 2014. Ogni modifica successiva alla pubblicazione del manuale sarà direttamente reperibile nel sito di riferimento curato dal W3C, che è disponibile su <http://aspit.co/a0v>.

Un po' di storia

Prima di cominciare a parlare in modo specifico di HTML5, vediamo di capire come il linguaggio HTML è nato e quale è stata la sua evoluzione nel corso degli anni. Per comprendere meglio l'evoluzione del Web e di questi linguaggi, è necessario suddividere in differenti fasi storiche la trattazione.

Gli albori: da HTML 1 a HTML 3

Il linguaggio HTML nacque nei primissimi anni '90 dal lavoro di **Tim Berners-Lee** (considerato il papà del Web presso il CERN di Ginevra, in Svizzera. Oggi Tim Berners-Lee dirige il World Wide Web Consortium insieme al CEO Jeffrey Jaffe.

Nel 1989 Tim Berners-Lee propose un progetto riguardante la pubblicazione di ipertesti, noto con il nome di “World Wide Web”. All'interno di questo progetto presero vita sia il server web “httpd” (HyperText Transfer Protocol Daemon), sia il client “World WideWeb”, il primo browser della storia.

nota

Gli ipertesti sono documenti elettronici il cui contenuto non deve essere letto obbligatoriamente in modo lineare e sequenziale. Essi sono creati in maniera tale da poter offrire un collegamento tra un punto del testo e altri documenti correlati o altri punti del testo all'interno dello stesso documento, che diventa navigabile. Oggi questa definizione sembra scontata, ma negli anni '90 rappresentò una vera innovazione.

La realizzazione del client partì nell'ottobre del 1990 e questo fu reso disponibile dapprima internamente al CERN e successivamente in Internet nel corso del 1991.

Per la stesura di ipertesti, assistito dai suoi colleghi del CERN, Tim Berners-Lee concorse anche alla definizione della prima versione di HTML, che fu ufficialmente resa pubblica nel giugno del 1993.

Per la definizione di HTML, Tim Berners-Lee ricorse a SGML (Standard Generalized Markup Language), un metalinguaggio finalizzato alla definizione di linguaggi utilizzabili per la stesura di documenti destinati a essere trasmessi in forma elettronica. L'idea centrale di SGML consiste nel definire linguaggi basati su marcatori testuali, che permettano di descrivere le caratteristiche strutturali dei documenti. In HTML questi marcatori prendono il nome di **tag**.

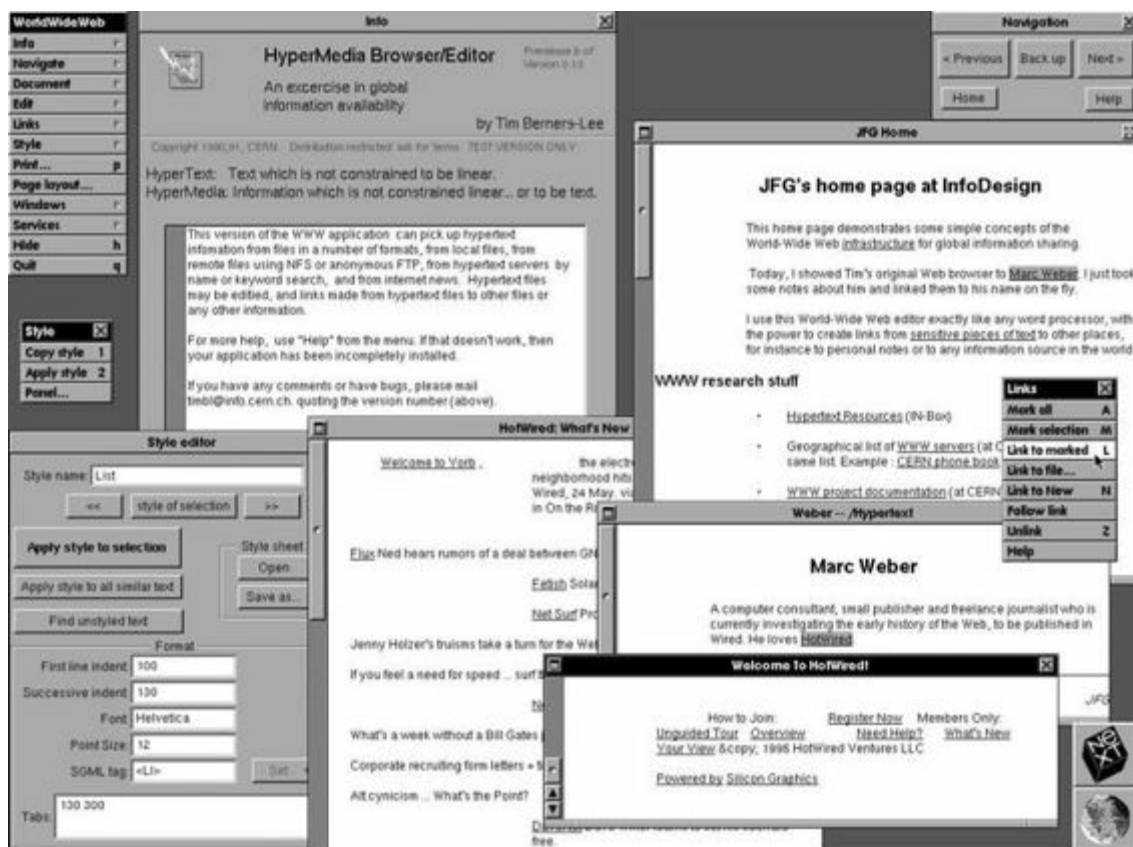


Figura 1.1 – WorldWideWeb, il primo browser della storia.

Nel 1994 nacque il W3C e, da quel momento in poi, lo sviluppo di HTML divenne prerogativa del World Wide Web Consortium. Nel 1995 il W3C definì la versione 3.0 di HTML, a cui seguì la versione 3.2 nel 1997. Infine, nel 1998, furono rilasciate le specifiche di HTML4.

La prima infanzia: da HTML4 a XHTML e ritorno

A quel punto, i membri del W3C decisero di fermare lo sviluppo di HTML, per dare vita a una versione equivalente del linguaggio basata su XML, ovvero XHTML. Questo progetto partì con la riformulazione di HTML4 in formato XML, fino alla definizione di XHTML 1.0 nell'anno 2000.

Nel corso del nuovo millennio, il W3C cominciò a lavorare a due nuovi progetti, uno orientato a estendere XHTML, l'altro finalizzato a definire un nuovo linguaggio non compatibile con le versioni precedenti di HTML e XHTML, noto col nome di XHTML2. Quest'ultimo progetto è stato dichiarato ufficialmente fallito alla fine del 2010, in favore di un approccio meno rigido. La sintassi di HTML5 segue infatti regole più flessibili rispetto a XHTML, dal momento che il formato XML rappresenta un'opzione non più obbligatoria. In altre parole, le specifiche di HTML5 non obbligano più a utilizzare una sintassi

basata su XML, come invece avviene nel caso di XHTML.

Quando creiamo un documento o una pagina in HTML5, possiamo considerare due sintassi: HTML oppure XHTML. La sintassi HTML è più flessibile e questo la rende compatibile anche con i browser più obsoleti. La sintassi XHTML è più rigida, dato che obbliga a una formattazione dei contenuti secondo le regole di XML.

Se un documento viene trasmesso con un MIME type HTML come text/html, nei browser esso viene processato secondo la sintassi HTML. Se un documento viene trasmesso, invece, con un MIME type XML come application/xhtml+xml, nei browser viene trattato come un documento XML a tutti gli effetti, con tutti i limiti del caso.

Il modo in cui i browser fanno il parsing di un documento o di una pagina differisce nel caso delle due sintassi menzionate. In generale, se un documento viene processato come XML, ogni singolo errore sintattico produce una visualizzazione parziale dei contenuti. Nel caso della sintassi HTML, il parser è più flessibile e trascura gli errori.

La nascita di HTML5

Nel corso del 2004, a seguito di un workshop, alcuni membri del W3C decisero di far ripartire l'evoluzione di HTML, dando vita a un nuovo progetto finalizzato a definire le specifiche della versione successiva di HTML: erano le prime fasi di quello che sarebbe diventato HTML5.

Nacque così il Web Hypertext Application Technology Working Group (WHATWG), un gruppo di lavoro staccato dal W3C, composto unicamente da Apple, Mozilla Foundation e Opera Software.

Due anni più tardi, più precisamente nel 2006, il W3C decise di prendere parte allo sviluppo di HTML5 e nel 2007 si unì al WHATWG per partecipare alla definizione delle specifiche della nuova versione del linguaggio. Da quel momento in poi, è stato il W3C a occuparsi di HTML5 e il WHATWG ha perso di importanza. HTML5, quindi, è supportato praticamente da tutti i vendor di sistemi operativi, browser e dispositivi mobile, raggiungendo quella che era l'idea iniziale del suo creatore, Tim Berners-Lee, di creare un linguaggio che consentisse l'interscambio di documenti ipertestuali in maniera indipendente dalle piattaforme.

Anatomia di una pagina HTML

Molte caratteristiche presenti fin dalla prima versione di HTML sono rimaste invariate nel corso del tempo e molti degli aspetti presenti in HTML4 continuano a rimanere validi anche in HTML5. Del resto, come abbiamo appena visto, HTML è nato come un linguaggio basato su tag per rappresentare ipertesti e questa sua caratteristica non è certamente destinata a cambiare. Vediamo allora i concetti fondamentali di HTML e diamo un po' di definizioni di base.

Un **documento HTML**, ovvero una pagina web nella maggior parte dei casi, è una struttura ad albero composta da elementi e testo, come mostrato [nell'esempio 1.1](#). Un **elemento** è formato da una coppia di tag, comprendente sia testo, sia elementi di varia natura. Il tag di apertura, come, per esempio, <head>, definisce l'inizio dell'elemento e, in genere, a esso corrisponde un tag di chiusura, come </head>. L'insieme di elementi e testo di cui una pagina si compone viene detto **markup**.

Esempio 1.1

```
<!DOCTYPE html>
<html>

<head>
  <title>Pagina di esempio</title>
  <link rel="stylesheet" href="stili.css" type="text/css" />
</head>

<body>

  <!-- Inizio della pagina -->

  <h1>Pagina di esempio</h1>

  <p>
    Questo è un esempio molto semplice di
    <a href="pagina.html">pagina HTML</a>.
  </p>

  <!-- Fine della pagina -->

</body>

</html>
```

Ciascun tag rappresenta un'informazione di natura strutturale con un

significato semantico particolare, che denota in qualche modo il testo a esso collegato. [Nell'esempio 1.1](#), il tag `a` permette di definire un link ipertestuale che punta a un'altra pagina, il tag `p` rappresenta un paragrafo del testo, il tag `h1` identifica un titolo, mentre i tag `<!-- -->` rappresentano i commenti, ovvero quei contenuti testuali che non devono essere presi in considerazione nella fase di rendering della pagina.

I vari tag vengono annidati gli uni con gli altri, in maniera tale da comporre una struttura complessa in grado di formattare il contenuto in modo opportuno. All'interno di questa struttura ogni elemento deve essere posizionato secondo determinate regole di precedenza, definite dalle specifiche di HTML, e secondo un ordine gerarchico che dipende dal significato semantico di ciascun elemento.

Il tag di apertura di un elemento può essere dotato di **attributi**, ovvero di proprietà composte da coppie nome/valore, il cui scopo è quello di fornire informazioni aggiuntive in grado di denotare in modo completo l'elemento a cui si riferiscono. [Nell'esempio 1.1](#), il tag `a` include l'attributo `href` che permette di indicare l'indirizzo della pagina a cui il link ipertestuale punta.

Quando un browser interpreta una pagina, esegue il parsing del markup e genera una rappresentazione ad albero della struttura del documento HTML. Questa rappresentazione si chiama **Document Object Model (DOM)** e si compone di tutti gli elementi presenti nel markup, a partire dall'elemento root, ovvero il tag `html`. Questo elemento contiene a sua volta due tag particolari:

- `head`, che serve per aggregare i metadati relativi alla pagina (tra queste informazioni figurano anche i meta-tag utilizzati dai motori di ricerca per l'indicizzazione). Si tratta di informazioni che non sono visualizzate direttamente, ma sono soprattutto impostazioni;
- `body`, che rappresenta il corpo vero e proprio della pagina, comprensivo di tutto il contenuto che il browser deve mostrare in fase di rendering.

I tag `html`, `head` e `body` sono presenti in ogni pagina (e nel relativo DOM), dal momento che essi rappresentano gli elementi base di cui ogni documento HTML si compone. [Nell'esempio 1.1](#), l'elemento `head` include il tag `title`, che permette di dare un titolo alla pagina, e il tag `link` che, nello specifico, consente di indicare la posizione del file contenente gli stili CSS. L'elemento `body` include gli altri elementi descritti in precedenza, che producono il risultato mostrato nella [figura 1.2](#).



Figura 1.2 – Pagina di esempio nel browser e relativo DOM.

Come possiamo notare, ciò che viene preso in considerazione per essere mostrato è solo la parte di markup inclusa nel tag `body`, ad eccezione del contenuto del tag `title`, che solitamente viene riportato nell'intestazione della finestra o nel nome del tab del browser. Il contenuto rimanente del tag `head` e i commenti non sono renderizzati.

nota

Secondo le specifiche di HTML5, ogni documento HTML si apre con il tag di intestazione `<!DOCTYPE html>`, che, diversamente dal passato, esprime in modo sintetico e semplificato la tipologia del documento stesso. A questo tag non corrisponde alcun tag di chiusura e, all'interno della pagina, è subito seguito dall'elemento `html`. Questo tag particolare è chiamato semplicemente DocType.

Oltre ai tag già menzionati nel corso del paragrafo, HTML include molti altri elementi, che avremo modo di elencare nel prossimo capitolo, quando cominceremo a parlare di HTML5 nel dettaglio. Tuttavia, alcuni dei tag di HTML rappresentano elementi di formattazione di base, presenti fin dalle

primissime versioni del linguaggio, e vale la pena darne una descrizione già in questo contesto.

A tale scopo, la [tabella 1.1](#) elenca i tag fondamentali di HTML e, per ciascuno di essi, fornisce una descrizione del loro significato semantico. In ogni caso, la tabella non riporta tutti gli elementi presenti in HTML, ma prende in considerazione semplicemente quelli più significativi e di uso più frequente, come utile indicazione per chi tra i lettori non dovesse conoscere nulla (o quasi) di HTML.

<i>Elemento/i</i>	<i>Descrizione</i>
<code>a</code>	Permette di definire un link ipertestuale verso una pagina o un punto della pagina corrente. L'attributo <code>href</code> serve per specificare l'indirizzo della pagina a cui il link deve puntare.
<code>body</code>	Rappresenta il corpo della pagina.
<code>br</code>	È il carattere “a capo” per l'interruzione di linea.
<code>button</code>	Consente di definire un pulsante.
<code>div</code>	Rappresenta un contenitore generico per associare un riferimento o uno stile a una porzione del markup.
<code>form</code>	Identifica un modulo per l'inserimento di dati. Le form sono trattate nel dettaglio nel capitolo 3 .
<code>h1...h6</code>	Rappresenta un titolo o un sottotitolo, in ordine di importanza decrescente, da 1 a 6.
<code>head</code>	Rappresenta la parte iniziale di un documento HTML, contenente i metadati e i riferimenti alle risorse usate nel markup.
<code>hr</code>	Rappresenta una riga orizzontale di separazione.
<code>html</code>	È l'elemento radice di un documento HTML.
<code>iframe</code>	Rappresenta un riquadro interno alla pagina, il cui contenuto è distinto da essa e caricato da un'altra locazione. L'attributo <code>src</code> serve per specificare l'indirizzo della pagina esterna.
<code>img</code>	Permette di mostrare un'immagine all'interno della pagina. L'attributo <code>src</code> serve per indicare il percorso dove è situato il file da mostrare.

<code>input</code>	Consente di definire un elemento per l'inserimento di dati. Questo tag è usato all'interno delle form.
<code>label</code>	Rappresenta l'etichetta associata a un elemento per l'inserimento di dati.
<code>link</code>	Consente di specificare un riferimento a una risorsa esterna (per esempio, un file CSS) all'interno dell'elemento <code>head</code> della pagina.
<code>meta</code>	È il tag usato per inserire metadati all'interno dell'elemento <code>head</code> della pagina. L'attributo <code>name</code> serve per indicare il nome associato al metadato, mentre l'attributo <code>content</code> permette di specificare il suo valore.
<code>p</code>	Rappresenta un paragrafo del testo.
<code>ol</code> , <code>ul</code> , <code>li</code>	Il tag <code>ol</code> consente di definire un elenco numerato, composto da una serie di elementi <code>li</code> al proprio interno. Il tag <code>ul</code> consente, con la stessa struttura, di inserire un elenco puntato.
<code>script</code>	Permette di includere direttamente all'interno della pagina una porzione di codice lato client o, in alternativa, consente di specificare un riferimento a un file esterno attraverso l'attributo <code>src</code> .
<code>select</code> , <code>option</code>	Rappresenta una dropdownlist per la selezione di un valore a partire da una lista di opzioni. L'elemento <code>select</code> include una serie di tag <code>option</code> , uno per ciascuna voce presente nell'elenco.
<code>span</code>	Rappresenta un contenitore generico per associare un riferimento o uno stile a una porzione di testo.
<code>style</code>	Permette di aggiungere in modo diretto la definizione degli stili CSS all'interno dell'elemento <code>head</code> della pagina, senza dover fare riferimento a un file esterno.
<code>table</code> , <code>th</code> , <code>tr</code> , <code>td</code>	Consente di definire una tabella composta da righe, colonne e celle. Il tag <code>tr</code> rappresenta una riga, mentre l'elemento <code>td</code> identifica una cella. Il tag <code>th</code> rappresenta le celle nell'intestazione della tabella.
<code>textarea</code>	Rappresenta una casella per l'inserimento di testo. Questo tag è

usato all'interno delle form.

title	Permette di indicare il titolo della pagina all'interno dell'elemento head.
-------	---

Tabella 1.1 – Elementi fondamentali di HTML.

A partire dalla versione 5, HTML ingloba una serie di nuovi elementi. Questi tag vengono presentati nel prossimo paragrafo insieme alle altre novità del linguaggio e saranno descritti in dettaglio nel corso del libro.

Le novità di HTML5

Le novità di HTML5 sono numerose e riguardano non solo gli elementi del markup, in particolar modo da un punto di vista semantico, ma anche la gestione dello stato di navigazione e gli aspetti generali legati all'interazione utente. Per esempio, fanno parte delle specifiche il supporto per la memorizzazione di grosse quantità di dati in locale all'interno del browser, in modo da consentire, tra l'altro, l'utilizzo di applicazioni web anche in assenza di un collegamento a Internet, situazione abbastanza frequente nel caso dei dispositivi mobili.

Come anticipato, HTML5 è un insieme di centinaia di specifiche, non tutte legate al markup, che introducono una serie di caratteristiche nuove e raggruppate sotto un solo nome. Come avremo modo di vedere, gruppi di specifiche vengono in realtà trattate singolarmente e spesso ci si riferisce a loro con il nome delle specifiche stesse.

I nuovi tag HTML

In generale, HTML5 permette un approccio migliore nella strutturazione dei contenuti delle pagine, grazie a una serie di caratteristiche innovative, che possiamo riassumere come segue:

- vengono introdotte regole più stringenti per quanto riguarda la strutturazione del testo, con l'aggiunta di una serie di tag semantici (per esempio, article, section, header e footer), orientati a connotare in modo mirato i contenuti in base al loro significato. Parallelamente, vengono eliminati i tag obsoleti o di scarso interesse e vengono estesi a tutti gli elementi del markup alcuni attributi, principalmente

finalizzati all'accessibilità, finora previsti solo per un numero limitato di tag. Parleremo di queste novità nel [capitolo 2](#);

- vengono estesi gli elementi presenti nelle form, per migliorare l'interazione utente nell'inserimento di dati e per supportare al meglio i diversi browser e dispositivi presenti sul mercato. Avremo modo di presentare le novità relative alle form nel [capitolo 3](#);
- viene introdotto il supporto alle tecnologie SVG e Canvas, che consentono di utilizzare JavaScript per creare animazioni e grafica vettoriale. Mostreremo come utilizzare queste tecnologie nel [capitolo 10](#);
- vengono introdotti i tag video e audio per il controllo diretto di contenuti multimediali, senza la necessità di dover ricorrere a tecnologie proprietarie. Parleremo nello specifico di questi tag nel [capitolo 9](#).

Le novità dei CSS

Alle caratteristiche appena elencate, proprie del linguaggio HTML, si accompagnano le novità che riguardano i fogli di stile CSS (**Cascading Style Sheets**) grazie all'avvento di CSS3, di cui avremo modo di parlare nei [capitoli 5, 6 e 7](#).

CSS3 include una vasta gamma di stili ed effetti applicabili in modo diretto ai contenuti senza dover adattare il markup, rendendo possibile una personalizzazione grafica senza precedenti e un controllo tipografico flessibile, anche grazie all'adozione del formato **WOFF** (Web Open Font Format). In CSS3 possiamo applicare gli stili agli elementi del DOM utilizzando query di selezione complesse, per esempio, in base alla posizione, alla gerarchia, in funzione della presenza o del valore di un attributo del tag, usando logiche come la negazione ecc.

Abbiamo inoltre la possibilità di applicare trasparenze, gradienti, ombreggiature, immagini di sfondo multiple, di creare bordi arrotondati e di utilizzare font personalizzati, il che ci permette di gestire gli effetti grafici più comuni senza dover ricorrere a immagini aggiuntive. Le media query di CSS3 consentono di differenziare lo stile applicato in base alle caratteristiche peculiari di un particolare dispositivo (come dimensioni, orientamento e risoluzione dello schermo), semplificando così lo sviluppo multi-piattaforma

e la realizzazione di siti pensati per essere visualizzati sui dispositivi mobili, adottando il cosiddetto **responsive design**. Infine, il supporto alle transizioni di CSS3 ci permette di gestire semplici animazioni e trasformazioni, come slideshow e menù dinamici, il tutto senza dover far ricorso allo scripting.

Le novità di JavaScript

Parallelamente a CSS3, anche JavaScript presenta non poche novità. Con HTML5 vengono infatti introdotte nuove API, allo scopo di colmare molte delle lacune presenti fino a oggi nel linguaggio. Queste ultime sono orientate a consentire un migliore accesso al DOM, favorire l'impiego delle risorse del client, semplificare i processi di interazione tra utente e applicazione, aumentare la performance e, infine, fornire un supporto programmatico negli scenari disconnessi e un'ottimizzazione della comunicazione client-server. Negli ultimi tre capitoli del libro parleremo di queste nuove API JavaScript e, in particolare, delle funzionalità che riguardano:

- l'accesso agli elementi del DOM tramite query basate sui selettori CSS3, in modo analogo a quanto possiamo fare con jQuery (**Selectors API**);
- il supporto alla geolocalizzazione, per determinare la posizione geografica del client. Si tratta di una funzionalità motivata principalmente dalla forte diffusione dei dispositivi mobili dotati di GPS (**Geolocation API**);
- un sistema di memorizzazione alternativo ai normali cookie, in grado di consentire la persistenza dei dati sul client (**Session Storage** e **Local Storage**), con un notevole risparmio di banda nella comunicazione col server (**Web Storage API**);
- la standardizzazione dei programmi JavaScript, con la possibilità di utilizzare le applicazioni web anche in modalità offline (**Application Cache**) e di gestire l'esecuzione del codice in thread separati, garantendo un certo livello di isolamento (**Web Workers**);
- la possibilità di modificare la cronologia di navigazione del browser (**History API**);
- lo scambio online di informazioni cross-domain (**Messaging API**);

- il supporto alla comunicazione basata su socket tramite un canale dedicato tra client e server (**WebSocket**).

Quando parliamo di HTML5, intendiamo un insieme di tecnologie tra loro complementari e caratterizzate da finalità diverse, ciascuna delle quali è in grado di fornire una risposta adeguata alle molteplici esigenze legate alla realizzazione di siti e applicazioni web.

Pertanto, nel corso del libro, parleremo di **markup HTML**, per la definizione dei contenuti da un punto di vista semantico, di **ECMAScript5 (JavaScript)**, per la gestione dei comportamenti, dell'interazione e dei dinamismi presenti nelle pagine, di **CSS3 (Cascading Style Sheets)**, per la User Interface, e del supporto alla **multimedialità**, che permette di gestire audio e video in maniera nativa all'interno del browser.

Un primo esempio di pagina HTML5

Giunti a questo punto, siamo pronti per dare uno sguardo più da vicino a HTML5. Per avere una prima idea di come si presenta una pagina HTML5, consideriamo [l'esempio 1.2](#). In questo esempio abbiamo inserito alcune delle novità di cui abbiamo parlato nel corso del paragrafo precedente: i tag semantici (in questo caso, header, article, section e footer), l'elemento canvas, per la generazione dinamica di immagini grazie all'uso dell'API JavaScript a questo associata, e il tag video, utile per la riproduzione di filmati senza la necessità di dover utilizzare un player di qualche tipo.

Esempio 1.2

```
<!DOCTYPE html>
<html>

<head>
  <title>Pagina di esempio</title>
  <link rel="stylesheet" href="stili.css" type="text/css" />
  <script type="text/javascript">
    window.onload = function() {
      var canvas = document.getElementById('html5_canvas');
      var context = canvas.getContext('2d');
      context.fillStyle = 'black';
      context.fillRect(0, 0, 200, 100);
      context.textBaseline = 'top';
      context.font = 'bold 16pt Tahoma';
```



```

        context.fillStyle = 'orange';
        context.fillText('HTML5 Espresso', 15, 35);
    }
</script>
</head>

<body>
    <header>
        <h1>HTML5 Espresso</h1>
    </header>
    <article>
        <header>
            <h1>Pagina di esempio</h1>
        </header>
        <section>
            <h3>Il nuovo elemento <canvas></h3>
            <canvas id="html5_canvas" width="200" height="100">
                ATTENZIONE: l'elemento <canvas> non è
                supportato.
            </canvas>
        </section>
        <section>
            <h3>Il nuovo elemento <video></h3>
            <video id="html5_video" src="video.ogg" controls>
                ATTENZIONE: l'elemento <video> non è
                supportato.
            </video>
        </section>
    </article>
    <footer>
        <a href="http://books.asptalia.com/HTML5/">HTML5
        Espresso</a>
    </footer>
</body>

</html>

```

La [figura 1.3](#) mostra come la pagina di esempio viene renderizzata da Google Chrome. Come possiamo notare, la funzione JavaScript presente [nell'esempio 1.2](#) sfrutta la tecnologia Canvas per la generazione dinamica di bitmap e produce una figura rettangolare di sfondo nero contenente la scritta arancione “HTML5 Espresso”. Parallelamente, il tag video permette di visualizzare un filmato in formato MPEG4 all'interno della pagina, fornendo automaticamente i controlli necessari per la gestione della riproduzione.



Figura 1.3 – Pagina di esempio in Google Chrome.

Dato che HTML5 è composto da differenti specifiche racchiuse da un nome commerciale, è bene ricordare che il risultato del rendering di una pagina come quella mostrata nella [figura 1.3](#) dipende fortemente dal browser utilizzato, anche se possiamo dire che, nel caso in questione, le specifiche utilizzate sono ormai supportate correttamente da tutti i browser più recenti, aggiornati negli ultimi 3-4 anni.

Browser detection o feature detection?

Per i motivi introdotti nel paragrafo precedente, contrariamente a quanto avveniva in passato (browser detection), oggi si tende a preferire un approccio più moderno e in grado di adattarsi alle reali caratteristiche offerte da un browser (feature detection).

Intercettare la versione di un browser o, peggio ancora, la tipologia di browser, porta a comportamenti errati: una certa versione, infatti, potrebbe

non supportare una particolare specifica, che viene aggiunta in quelle successive. Piuttosto che cercare di differenziare il comportamento in base a questo particolare, si tende a preferire un approccio per il quale viene definito, attraverso una libreria di appoggio, se quella particolare specifica è supportata a runtime.

In tal senso, la libreria più diffusa è **Modernizr** (<http://modernizr.com/>), che è composta da un file JavaScript da includere nella pagina.

Questa libreria funziona in maniera molto semplice: aggiungerà come classe CSS al tag `html` della pagina le feature che supportate. [Nell'esempio 1.3](#) viene mostrato come fare, in modo che possano essere sfruttati i nuovi background multipli, se presenti, e di fornire un background unico come opzione predefinita.

Esempio 1.3

```
#myDiv {
    background: url(background-full.png) top left repeat-x;
}
.multiplebgs #myDiv {
    background: url(background.png) top left repeat-x,
    url(background.png) bottom left repeat-x;
}
```

Se il browser supporta i background multipli, infatti, Modernizr aggiungerà la classe `multiplebgs` al tag `html`, consentendoci di gestire al meglio i browser moderni, senza penalizzare quelli più recenti.

Oltre che attraverso il markup, è possibile gestire la feature detection anche via JavaScript, come [nell'esempio 1.4](#), in cui testiamo se il browser supporta le funzionalità di GeoLocation API.

Esempio 1.4

```
if (Modernizr.geolocation) {
    // abbiamo il supporto per la geolocation
}
```

Dispositivi mobili

Quando parliamo di dispositivi mobili, ci riferiamo principalmente agli smartphone o ai tablet di ultima generazione, dotati di sistemi operativi come

Apple iOS (usato su iPhone e iPad), Android, Windows e BlackBerry, tanto per citare i casi più significativi.

In questo tipo di dispositivi, molte delle funzionalità introdotte con HTML5 si rivelano fondamentali:

- il supporto per le applicazioni offline, che rende possibile salvare sul dispositivo informazioni utili al corretto funzionamento dei programmi, per sopperire alle possibili interruzioni della connettività dovute alla mancanza della rete;
- Canvas e la riproduzione di filmati attraverso il tag video, che permettono di non dover installare plugin di terze parti come Adobe Flash;
- il supporto alle GeoLocation API, che, come già detto in precedenza, permette di sfruttare il GPS interno per localizzare la posizione del dispositivo;
- il supporto avanzato per le form, che facilita l’inserimento e la validazione dei dati, soprattutto per i dispositivi non provvisti di tastiera fisica, ma solo di touchscreen.

Conclusioni

È indubbio che il motivo scatenante per cui si è cominciato a parlare di HTML5 con così largo anticipo rispetto al rilascio delle sue specifiche, avvenuto a ottobre 2014, è stata la possibilità di gestire in modo diretto i contenuti multimediali all’interno dei browser. In più, la scelta fatta da Apple di non supportare Adobe Flash nei suoi dispositivi mobili di punta, ovvero iPhone e iPad, ha creato un certo interesse nei confronti di HTML5 anche per i “non addetti ai lavori”.

Peraltro, come abbiamo spiegato nel corso del capitolo, HTML5 non si limita solamente al tag video. Le novità sono numerose e riguardano il markup HTML, i fogli di stile, in virtù dell’avvento di CSS3, e il linguaggio JavaScript con le sue nuove API.

Nel corso del libro avremo modo di illustrare tutte le caratteristiche di HTML5 nel dettaglio, partendo da quelle che riguardano il markup, oggetto del prossimo capitolo, fino ad arrivare alle novità legate a CSS e JavaScript.

Nuovi elementi del markup

HTML5 rappresenta l'ultima versione di un linguaggio che è molto cambiato nel corso del tempo, cercando di evolvere in funzione delle tecnologie esistenti e delle sempre nuove esigenze di pubblicazione statiche e dinamiche delle pagine web. Molti degli elementi inclusi nella versione 4 di HTML rimangono validi anche in HTML5, con nuovi tag e attributi aggiuntivi che fanno la loro comparsa, allo scopo di supportare al meglio la formattazione e la strutturazione di pagine e testi, per gestire nuove tipologie di contenuti, per lo più multimediali, e per permettere una più agevole interazione da parte dell'utente in caso di inserimento di dati.

Nel corso di questo capitolo e in quelli successivi, prenderemo in esame i nuovi elementi di HTML5, fornendo per ciascuno una descrizione delle caratteristiche peculiari e uno o più esempi di utilizzo. In particolare, in questo capitolo ci focalizzeremo principalmente sui nuovi tag e attributi che riguardano la strutturazione delle pagine e la formattazione dei contenuti testuali, non prima, peraltro, di aver visto quali sono gli elementi supportati da HTML5 nel suo complesso.

Gli elementi supportati da HTML5

HTML5 include più di un centinaio di elementi di markup, di cui una buona parte proviene dalle versioni precedenti. Sebbene il numero di tag derivanti dal passato sia abbastanza corposo, non tutto il pregresso rimane ancora valido: alcuni tag sono stati rimossi (e quindi non sono più supportati) perché ritenuti obsoleti o inutili.

La [tabella 2.1](#) riporta in ordine alfabetico l'elenco di tutti gli elementi supportati dal linguaggio. I tag contrassegnati con l'asterisco rappresentano le novità di HTML5.

<!-->	<!DOCTYPE>	<a>	<abbr>
<address>	<area>	<article> (*)	<aside> (*)
<audio> (*)		<base>	<bdi> (*)
<bdo>	<blockquote>	<body>	
<button>	<canvas> (*)	<caption>	<cite>
<code>	<col>	<colgroup>	<command> (*)
<datalist> (*)	<dd>		<details> (*)
<dfn>	<div>	<dl>	<dt>
	<embed> (*)	<fieldset>	<figcaption> (*)
<figure> (*)	<footer> (*)	<form>	<h1>
<h2>	<h3>	<h4>	<h5>
<h6>	<head>	<header> (*)	<hgroup> (*)
<hr>	<html>	<i>	<iframe>
	<input>	<ins>	<keygen> (*)
<kbd>	<label>	<legend>	
<link>	<map>	<mark> (*)	<menu>
<meta>	<meter> (*)	<nav> (*)	<noscript>
<object>		<optgroup>	<option>
<output> (*)	<p>	<param>	<pre>
<progress> (*)	<q>	<rp> (*)	<rt> (*)
<ruby> (*)	<samp>	<script>	<section> (*)
<select>	<small>	<source>	
	<style>	<sub>	<summary> (*)
<sup>	<table>	<tbody>	<td>
<textarea>	<tfoot>	<th>	<thead>
<time> (*)	<title>	<tr>	
<var>	<video> (*)	<wbr> (*)	

Tabella 2.1 – Elenco degli elementi supportati da HTML5.

Come possiamo notare guardando l’elenco riportato nella [tabella 2.1](#), i nuovi elementi di HTML5 attualmente documentati sono in tutto ventotto. Alcuni di questi tag saranno descritti nel corso di questo capitolo:

- `section`, `hgroup`, `article`, `aside`, `nav`, `header`, `footer`, `figure` e `figcaption` sono tag semantici che sono stati introdotti al fine di fornire un approccio migliore nella strutturazione delle pagine;
- gli elementi `bdi`, `mark`, `ruby` con `rt` e `rp`, `time` e `wbr` sono stati concepiti specificamente per la formattazione dei contenuti testuali;
- `command`, `details` e `summary` rappresentano nuovi elementi interattivi.

Gli elementi rimanenti hanno altre finalità: alcuni servono per la gestione dei contenuti multimediali (`audio` e `video`) o per il rendering dinamico di immagini (`canvas`), altri sono destinati a rendere più agevole l'inserimento di dati all'interno di un elemento `form` (per esempio, `datalist`, `meter` oppure `progress`). Alcuni di questi elementi saranno trattati nel dettaglio nel corso dei prossimi capitoli, a cui potrete fare riferimento per un approfondimento delle loro caratteristiche di funzionamento e utilizzo.

Accanto agli elementi aggiuntivi che abbiamo appena elencato, HTML5 continua a contenere tag già di uso frequente nelle versioni precedenti come, per esempio, `img`, `a`, `input`, `ul` e `ol`. Alcuni di questi elementi sono stati modificati, al fine di includere nuovi attributi che ne estendano le funzionalità. Verso la fine del capitolo avremo modo di parlare anche di queste novità.

Gli elementi non più supportati da HTML5

Prima di cominciare a vedere più da vicino i nuovi tag di HTML5, vale la pena prendere brevemente in considerazione ciò che non è più presente nel linguaggio rispetto alle versioni precedenti.

Gli elementi `basefont`, `big`, `center`, `font`, `strike`, `tt` e `u` non sono più supportati, dal momento che il loro scopo è puramente legato alla rappresentazione del testo e gli stessi risultati sono ottenibili tramite CSS.

Anche i `frame` non sono più contemplati, in quanto rappresentano elementi potenzialmente dannosi in termini di sicurezza, usabilità e accessibilità. Di conseguenza, i tag `frame`, `frameset` e `noframes` non sono più disponibili.

Altri tag come `acronym`, `applet`, `isindex` e `dir` sono stati eliminati in quanto di uso non frequente o semplicemente perché esistono elementi alternativi funzionalmente simili. Per esempio, `dir` è stato dichiarato obsoleto a tutto vantaggio del tag `ul`.

Oltre ai tag, anche una serie di attributi sono stati rimossi in quanto ritenuti obsoleti, ridondanti o perché facilmente riproducibili attraverso CSS. Per esempio, l'attributo `align`, dapprima presente in moltissimi elementi di HTML, perde la sua utilità e non è più disponibile, dato che l'allineamento può essere gestito tramite gli stili. Per lo stesso motivo, gli attributi usati nella formattazione delle tabelle come `bgcolor`, `border`, `cellpadding`, `cellspacing`, `valign`, `width` e `height` non sono più presenti.

Come regola generale, in HTML5 tutto quello che riguarda la rappresentazione stilistica e la resa visiva diventa un problema che può e deve

essere gestito unicamente tramite gli stili, in particolare CSS, trattato nello specifico più avanti, nei [capitoli 4, 5 e 6](#). Secondo quest'ottica, tutte le numerose sovrapposizioni esistenti in HTML4 e in CSS2 sono state rimosse, al fine di produrre un linguaggio di markup più snello, orientato unicamente a definire gli aspetti legati alla formattazione e alla strutturazione dei contenuti.

I nuovi elementi di formattazione della pagina

Man mano che le esigenze di pubblicazione dei contenuti HTML sono mutate nel tempo, la formattazione delle pagine è diventata un aspetto via via sempre più complicato. Alla formattazione semplice e lineare dei primi siti, è seguita successivamente la necessità di posizionare i contenuti in modo più vario e complesso.

Inizialmente, le tabelle hanno rappresentato la soluzione al problema, fornendo un meccanismo per creare una struttura statica dove posizionare i contenuti all'interno delle pagine. In seguito, data la molteplicità di browser e di dispositivi da supportare, la staticità propria delle tabelle si è rivelata un limite troppo forte. Pertanto, nei siti più moderni, si è cominciato a usare gli elementi flottanti, ottenibili tramite l'impiego del tag `div` unitamente agli stili `float` e `clear`, dal momento che essi garantiscono un livello di flessibilità molto maggiore rispetto alle tabelle. Il tag `div` rappresenta peraltro un elemento contenitore generale, senza una particolare connotazione stilistica, che può essere impiegato secondo modalità e finalità diverse, raggruppando altri elementi o contenuti in ogni punto all'interno del corpo della pagina.

HTML5 affronta il problema della formattazione delle pagine in modo radicale e, come abbiamo già detto nel paragrafo precedente, presenta una serie di nuovi elementi di markup, destinati a fornire una soluzione alle esigenze di pubblicazione più comuni e a dare una connotazione semantica alle diverse parti della pagina.

Diversamente dal tag `div`, ciascuno di questi nuovi elementi nasce con un preciso scopo. La [tabella 2.2](#) riassume brevemente le finalità di ciascun tag.

<i>Elemento</i>	<i>Descrizione</i>
<code><section></code>	Rappresenta una sezione generica della pagina, senza una connotazione specifica.
<code><hgroup></code>	Rappresenta l'intestazione di una sezione e raggruppa elementi <code>h1</code> , <code>h2</code> , <code>h3</code> , <code>h4</code> , <code>h5</code> e <code>h6</code> .

<code><article></code>	Rappresenta una sezione indipendente, contenente principalmente un contenuto testuale, ma non solo.
<code><aside></code>	Rappresenta una sezione non indipendente, collegata in qualche modo al resto della pagina.
<code><nav></code>	Rappresenta una sezione della pagina pensata per la navigazione.
<code><header></code>	Rappresenta un blocco di intestazione all'interno della pagina o di una sezione (elementi <code>section</code> , <code>article</code> , <code>aside</code> e <code>nav</code>).
<code><footer></code>	Rappresenta un blocco di chiusura all'interno della pagina o di una sezione (elementi <code>section</code> , <code>article</code> , <code>aside</code> e <code>nav</code>).
<code><figure></code>	Rappresenta un blocco distinto dal testo principale, pensato per contenere immagini, diagrammi, esempi ecc.
<code><figcaption></code>	Rappresenta la didascalia per un elemento <code>figure</code> ed è opzionale.

Tabella 2.2 – Nuovi elementi di formattazione della pagina di HTML5.

Possiamo usare gli elementi descritti nella [tabella 2.2](#) in combinazione tra di loro, al fine di formare una pagina contenente diverse tipologie di sezioni e blocchi di markup. [L'esempio 2.1](#) mostra un semplice caso di formattazione di una pagina con HTML5.

Esempio 2.1

```
<!DOCTYPE html>
<html>

<head>
  <title>Capitolo 2 - Nuovi elementi del markup</title>
  <link rel="stylesheet" href="stili.css" type="text/css" />
</head>

<body>
  <header>
    <hgroup>
      <h1>CAPITOLO 2</h1>
      <h2>Nuovi elementi del markup</h2>
    </hgroup>
```

```

</header>
<article>
  <section>
    <p>Sebbene le sue specifiche non siano...</p>
  </section>
  <section>
    <h3>Gli elementi supportati da HTML5</h3>
    <p>HTML5 include più di un centinaio di elementi di
      markup...</p>
  </section>
  <section>
    <h3>I nuovi elementi di formattazione della pagina</h3>
    <p>Man mano che le esigenze di pubblicazione dei...</p>
  </section>
  <aside>
    <h3>Fonti</h3>
    <ul>
      <li>World Wide Web Consortium</li>
      <li>Web Hypertext Application Technology Working
        Group</li>
    </ul>
  </aside>
</article>
<footer>
  <p>
    HTML5 con CSS e JavaScript<br />
    D. Bochicchio, S. Mostarda<br />
    Copyright &copy; 2015 HOEPLI
  </p>
</footer>
</body>

</html>

```

Per avere un'idea più concreta di come una pagina formattata utilizzando i tag semantici di HTML5 possa essere strutturata all'interno di un browser, prendiamo in considerazione la [figura 2.1](#). In essa abbiamo rappresentato a titolo esemplificativo una situazione di distribuzione dei contenuti abbastanza comune all'interno della pagina. All'intestazione in alto (*header*) segue un corpo centrale suddiviso in tre parti (menù di navigazione sulla sinistra, area principale al centro e colonna laterale a destra), seguito a sua volta da un blocco di chiusura in basso (*footer*).



Figura 2.1 – Classica struttura di una pagina con i tag semantici di HTML5.

L'uso di questi nuovi tag è anche retrocompatibile, nel senso che i browser, quando non riconoscono un tag, tendono a ignorarlo completamente.

Come possiamo notare, i nuovi tag di HTML5 ci permettono di rappresentare in modo mirato le diverse parti di cui la pagina si compone, aiutandola a raggiungere il miglior significato a livello semantico.

Inoltre essi favoriscono la “lettura” del markup sia da parte dei browser in grado di supportare la nuova versione del linguaggio, sia da parte di sistemi automatici come, per esempio, un bot vocale o il crawler di un motore di ricerca. In HTML5, infatti, la struttura della pagina è meno anonima e il markup diventa più descrittivo, dato che ogni elemento denota il proprio contenuto in modo chiaro, a tutto vantaggio dei motori di ricerca, in grado di interpretare e indicizzare al meglio il testo in funzione dei tag presenti nel markup.

L'elemento `section`

L'elemento `section` permette di definire una sezione all'interno di una pagina. Una sezione rappresenta di fatto una porzione omogenea di contenuto,

generalmente dotata di intestazione ([esempio 2.2](#)). Esempi significativi di una sezione possono essere una parte di un articolo composta da una serie di paragrafi, la descrizione di un prodotto in un catalogo oppure l'insieme delle informazioni di contatto.

Esempio 2.2

```
<section>
  <h1>Capitolo 2 - Nuovi elementi del markup</h1>
  <p>Sebbene le sue specifiche non...</p>
</section>
```

L'elemento `section` non è pensato per essere un generico contenitore di markup come il tag `div`, perché denota una sezione di contenuto dal punto di vista semantico. Pertanto, la regola generale prevede che dobbiamo utilizzare questo tag per includere contenuti della pagina che sono effettivamente visualizzati dal browser. Se la necessità è quella di definire un blocco unicamente per applicare un particolare stile o per esigenze di scripting, allora l'elemento `div` rappresenta ancora la scelta corretta.

L'elemento `hgroup`

Internamente a una sezione possiamo opzionalmente definire un'intestazione. Essa può essere composta da uno o più tag `h1...h6`, che solitamente permettono di definire titoli e sottotitoli (più basso è il numero, più grande è la dimensione del testo). In HTML5 questi elementi vanno raggruppati insieme all'interno del tag `hgroup` ([esempio 2.3](#)).

Esempio 2.3

```
<section>
  <hgroup>
    <h1>Capitolo 2</h1>
    <h2>Nuovi elementi del markup</h2>
  </hgroup>
  <p>Sebbene le sue specifiche non siano...</p>
</section>
```

Possiamo utilizzare l'elemento `hgroup` non solo internamente al tag `section`, ma anche dentro al tag `header` e agli altri elementi che avremo modo di vedere a breve, ovvero `article`, `aside` e `nav`, che di fatto rappresentano

tipologie di sezioni più specifiche e destinate a uno scopo ben definito.

L'elemento `article`

L'elemento `article` è una sezione di contenuto con una connotazione semantica ben definita che, in linea di principio, può essere distribuita o utilizzata in modo indipendente. Esempi significativi di una sezione di questo tipo possono essere il post di un blog, l'articolo di una rivista o di un giornale online, una news o il commento a un contenuto. Come mostrato nell'esempio 2.4, il tag `article` può contenere al suo interno altri elementi, come, per esempio, `section` o `header`, oppure anche altri tag `article`.

Esempio 2.4

```
<article>
  <hgroup>
    <h1>CAPITOLO 2</h1>
    <h2>Nuovi elementi del markup</h2>
  </hgroup>
  <section>
    <p>Sebbene le sue specifiche non siano...</p>
  </section>
  <section>
    <h3>Gli elementi supportati da HTML5</h3>
    <p>HTML5 include più di un centinaio di elementi di markup...
    </p>
  </section>
  <section>
    <h3>I nuovi elementi di formattazione della pagina</h3>
    <p>Man mano che le esigenze di pubblicazione dei...</p>
  </section>
</article>
```

Quando due elementi `article` sono annidati, l'elemento interno rappresenta un contenuto che deve essere inevitabilmente collegato all'elemento contenitore. Un possibile esempio di una situazione come questa può essere il post di un blog: i suoi commenti possono essere rappresentati attraverso altrettanti tag `article` annidati dentro all'elemento che contiene il testo del post. In alternativa, possiamo comunque accorpare i vari commenti in una sezione separata rispetto al post, in modo tale che l'elemento `article` sia uno solo.

L'elemento `aside`

L'elemento `aside` rappresenta una sezione che include un contenuto che è collegato a quanto trattato nella pagina, ma che è comunque distinto da esso. Un esempio significativo di una sezione di questo tipo è rappresentato dalle barre laterali presenti nelle pagine di molti siti, che, pur riportando informazioni correlate al testo principale, di fatto forniscono contenuti che non sono essenziali.

L'esempio 2.5 mostra un caso significativo di utilizzo del tag.

Esempio 2.5

```
<article>
  <hgroup>
    <h1>Capitolo 2</h1>
    <h2>Nuovi elementi del markup</h2>
  </hgroup>
  <section>
    <p>Sebbene le sue specifiche non siano...</p>
  </section>
  <aside>
    <h3>Fonti</h3>
    <ul>
      <li>World Wide Web Consortium</li>
      <li>Web Hypertext Application Technology Working Group</li>
    </ul>
  </aside>
</article>
```

È importante sottolineare che l'elemento `aside` è pensato per includere un contenuto che è nettamente distinto da quello principale. Pertanto, questo tipo di tag si presta in modo perfetto per la pubblicazione nei siti di informazioni pubblicitarie, banner, elenchi di link, note, riferimenti o approfondimenti, solitamente posti a lato della sezione principale della pagina.

L'elemento `nav`

Come il nome suggerisce, l'elemento `nav` serve per racchiudere una serie di informazioni legate alla navigazione delle pagine di un sito. Esso rappresenta una sezione che contiene una serie di link che permettono di accedere ad altre pagine o ad altre sezioni della pagina corrente (esempio 2.6).

Esempio 2.6

```
<nav>
  <h1>Menù</h1>
  <ul>
    <li><a href="indice.html">Indice</a></li>
    <li><a href="introduzione.html">Introduzione</a></li>
    <li><a href="capitolo1.html">Capitolo 1</a></li>
    <li><a href="capitolo2.html">Capitolo 2</a></li>
  </ul>
</nav>
```

Non è comunque necessario che tutti i link di una pagina appartengano a una sezione di tipo `nav`. In genere, ha senso includere in un tag `nav` solo i link che appartengono alla parte della pagina che permette di gestire in modo centralizzato la navigazione, come, per esempio, un menù posto lateralmente o posizionato sotto il logo del sito. I link che di solito si trovano nel footer della pagina, come quelli relativi alla privacy, alle condizioni di utilizzo o al copyright, possono opzionalmente essere inclusi direttamente nel tag `footer` (che vedremo a breve), senza la necessità di dover definire per forza un elemento `nav`.

L'elemento header

L'elemento `header` rappresenta un blocco di intestazione e non è una sezione. Esso può essere inserito direttamente nel corpo della pagina oppure all'interno di una sezione (elementi `section`, `article`, `aside` e `nav`). Nel primo caso avremo l'intestazione di pagina, mentre nel secondo quello del blocco a cui fa riferimento.

Per capire meglio come possiamo utilizzare questo tag all'interno del markup, prendiamo in considerazione [l'esempio 2.7](#).

Esempio 2.7

```
<header>
  <hgroup>
    <h1>HTML5</h1>
    <h2>D. Bochicchio, S. Mostarda</h2>
  </hgroup>
  <p>Con uno stile chiaro, pratico e ricco di esempi...</p>
</header>
```

Come possiamo notare, il tag `header` è pensato per includere per lo più elementi di intestazione, come titoli e sottotitoli. Per questo motivo si presta principalmente a contenere elementi come `h1...h6` e `hgroup`. Tuttavia possiamo usare questo tag anche per includere altre tipologie di contenuti, come un testo introduttivo, un indice, un menù o un form di ricerca.

L'elemento footer

L'elemento `footer` rappresenta un blocco di chiusura e non è una sezione. Può essere inserito direttamente nel corpo della pagina oppure all'interno di una sezione, in genere come ultimo elemento (anche se questo non è obbligatorio).

L'esempio 2.8 mostra un caso significativo di utilizzo del tag.

Esempio 2.8

```
<footer>
  <p>
    HTML5<br />
    D. Bochicchio, S. Mostarda<br />
    Copyright &copy; 2015 HOEPLI
  </p>
</footer>
```

L'elemento `footer` nasce con lo scopo di includere principalmente contenuti come le informazioni di copyright, il nome dell'autore o del proprietario di una pagina o di un testo, un insieme di link ecc. Di conseguenza questo tag si presta a contenere diverse tipologie di elementi, tra cui anche sezioni di tipo nav.

Gli elementi figure e figcaption

L'elemento `figure` rappresenta un blocco distinto dal testo principale, pensato per contenere immagini, diagrammi, grafici, tabelle, esempi di codice, citazioni o estratti ecc. A `figure` possiamo opzionalmente associare una didascalia o una descrizione tramite il tag `figcaption` (esempio 2.9).

Esempio 2.9

```
<article>
  <p>Sebbene le sue specifiche non siano...</p>
  <p>HTML5 include più di un centinaio di elementi di markup...
```



```

</p>
<figure>
  <figcaption>Figura 2.1</figcaption>
  
</figure>
<p>Man mano che le esigenze di pubblicazione dei...</p>
<figure>
  <figcaption>Esempio 2.1</figcaption>
  <pre><code><!-- esempio di codice --></code></pre>
</figure>
<p>Oltre agli elementi di markup visti finora...</p>
</article>

```

L'elemento `figure` si rivela utile qualora vogliamo inserire all'interno di un testo un contenuto di completamento o approfondimento, senza alterarne o interromperne il flusso e senza rimandare a pagine esterne o a un altro punto della pagina corrente.

I nuovi elementi di formattazione del testo

Oltre agli elementi di markup visti finora, HTML5 presenta una serie di nuovi elementi aggiuntivi, il cui scopo è quello di connotare semanticamente alcune porzioni di testo. La [tabella 2.3](#) riassume brevemente la finalità di ciascun tag.

<i>Elemento</i>	<i>Descrizione</i>
<code><bdi></code>	Rappresenta una porzione di testo che deve essere isolata dal resto del contenuto per essere formattato in modo bidirezionale.
<code><mark></code>	Rappresenta una porzione di testo evidenziato.
<code><ruby></code>	Rappresenta una porzione di testo espresso tramite ideogrammi, utilizzati principalmente nelle lingue orientali.
<code><rt></code>	Usato all'interno del tag <code>ruby</code> , denota la componente testuale associata a un ideogramma.
<code><rp></code>	Usato all'interno del tag <code>ruby</code> , permette di isolare le parentesi poste attorno al testo associato a un ideogramma dal testo stesso.
<code><time></code>	Rappresenta un'orario (dalle 0:00 alle 23:59) oppure una data del calendario Gregoriano, opzionalmente con ora e fuso.

Tabella 2.3 – Nuovi elementi di formattazione del testo di HTML5.

Questi nuovi tag affiancano i numerosi elementi dello stesso tipo già esistenti in HTML4, tra cui ricordiamo `span`, `br`, `a`, `code`, `strong`, tanto per citarne alcuni. I nuovi elementi si limitano a completare le possibilità offerte dal linguaggio HTML nella formattazione del testo.

L'elemento `mark`

L'elemento `mark` permette di mettere in risalto una porzione di testo rispetto al resto del contenuto. L'effetto visivo sul testo è lo stesso che otteniamo quando usiamo un evidenziatore su un foglio di carta.

L'esempio 2.10 mostra un semplice caso di utilizzo del tag, nel quale la parola “evidenziato” viene messa in risalto rispetto al resto della frase.

Esempio 2.10

```
<p>
  Questo testo è <mark>evidenziato</mark>.
</p>
<p>
  Questo testo è <mark class="verde">evidenziato in verde</mark>.
</p>
```

Come possiamo vedere nella [figura 2.2](#), in fase di rendering, lo sfondo dietro alla porzione evidenziata cambia colore (di default diventa giallo), mentre altrove rimane invariato.

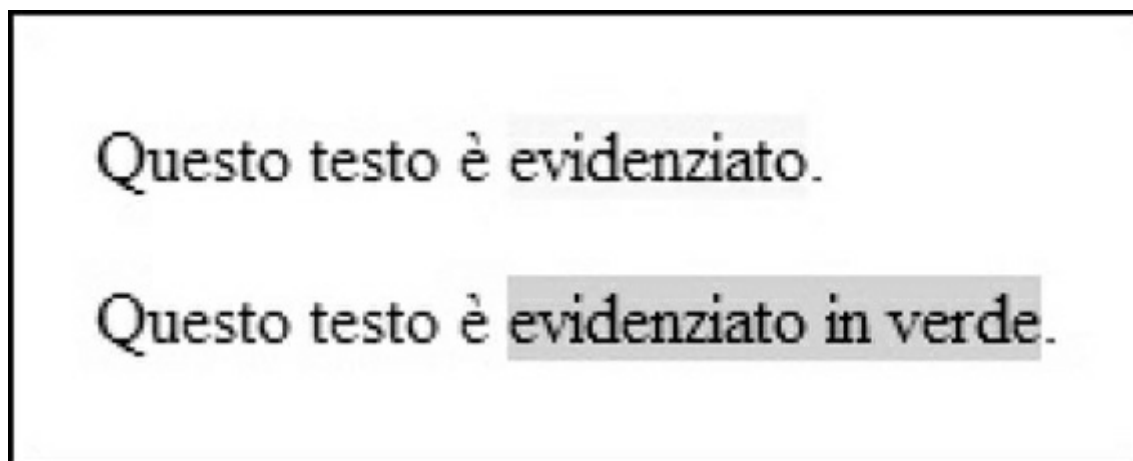


Figura 2.2 – Evidenziazione del testo tramite il tag mark.

Possiamo comunque variare il colore di evidenziazione a piacimento tramite CSS. Nell'esempio proposto, abbiamo applicato uno stile agendo sulla proprietà CSS `background`, al fine di evidenziare il testo con uno sfondo verde.

Gli elementi ruby, rt e rp

L'elemento `ruby`, insieme ai tag `rt` e `rp`, permette di formattare un contenuto espresso tramite gli ideogrammi, ovvero i simboli grafici che sono usati nelle lingue orientali come il giapponese o il cinese.

L'esempio 2.11 riporta alcuni casi d'uso del tag.

Esempio 2.11

```
<!-- Giapponese -->
<ruby>漢<rt>かん</rt>字<rt>じ</rt></ruby>

<!-- Cinese tradizionale -->
<ruby>漢<rt>`</rt>字<rt>`</rt></ruby>

<!-- Cinese semplificato -->
<ruby><rt>hàn</rt>字<rt>zì</rt></ruby>

<ruby>
  漢 <rp>(</rp><rt>かん</rt><rp>)</rp>
  字 <rp>(</rp><rt>じ</rt><rp>)</rp>
</ruby>
```

Come possiamo notare, all'interno dell'elemento `ruby`, tramite il tag figlio `rt` (*ruby text*), possiamo associare a ciascun ideogramma un testo esplicativo che eventualmente può essere separato dal resto del contenuto tramite parentesi incluse in altrettanti tag `rp` (*ruby parenthesis*).

La figura 2.3 mostra come il markup presente nell'esempio 2.11 dovrebbe essere renderizzato.

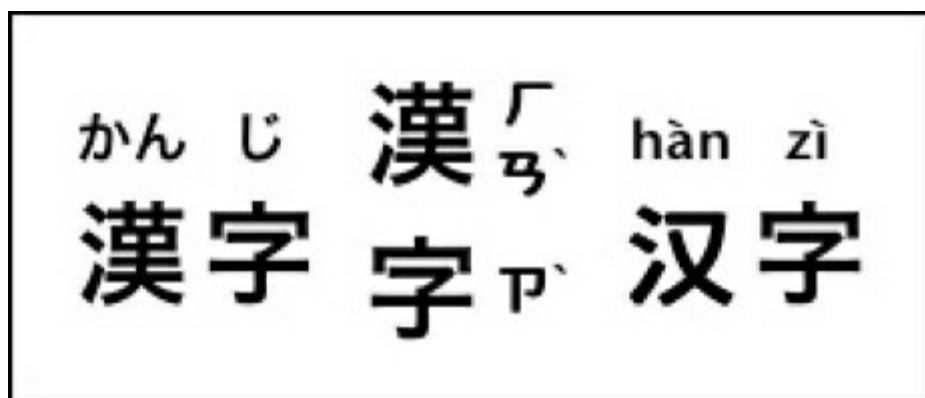


Figura 2.3 – Rendering del tag ruby.

Nei browser attualmente in circolazione, il supporto a questi tre tag è parziale (in particolare, per quanto riguarda gli elementi `rt` o `rp`) e, in alcuni casi, addirittura del tutto assente.

L'elemento time

L'elemento `time` permette di rappresentare un orario (dalle 0:00 alle 23:59) oppure una data del calendario Gregoriano, opzionalmente comprensiva di ora e fuso, o una durata. Lo scopo di questo tag è quello di fornire un modo per connotare una porzione di testo come un'informazione temporale, in modo tale che possa essere interpretata in modo univoco dal browser o da un bot come il crawler di un motore di ricerca.

L'esempio 2.12 mostra tre casistiche di impiego significative.

Esempio 2.12

```
<p>
  Orario di apertura: dalle <time>15:30</time> alle
  <time>19:30</time>
</p>
<p>
  <time datetime="2015-12-25T00:00Z">Natale 2015</time>
</p>
<p>
  Pubblicato il:<br />
  <time datetime="2015-04-01">1 aprile 2015</time>
</p>
```

Il primo caso illustrato nell'esempio rappresenta la situazione d'uso più semplice, dove il tag non presenta attributi.

Nei due casi rimanenti, l'elemento `time` viene usato per associare a un testo una data di riferimento. L'attributo `datetime` ci permette infatti di specificare la data da associare al testo, con ora e fuso opzionali e secondo il formato UTC, oppure direttamente un orario, senza data.

Un utilizzo particolare del tag `time` è quello per esprimere la durata temporale, come riportato [nell'esempio 2.13](#).

Esempio 2.13

```
<p>
  Durata delle ferie:
  <time datetime="P 4 D">4 giorni</time>
</p>
```

L'ultimo caso preso in esame rappresenta una durata, grazie al prefisso `P`: nell'esempio indichiamo che la durata che esprimiamo è di 4 giorni. Gli spazi sono opzionali, pertanto sia `P 4 D`, sia `P4D` sono considerati entrambi valori accettati.

L'elemento `wbr`

L'elemento `wbr` consente di indicare un'interruzione di riga opzionale. A differenza del tag `br`, che inserisce un'interruzione di riga in ogni caso, l'elemento `wbr` spezza il testo su due linee solo se necessario, in base al layout della pagina e alla larghezza del tag contenitore.

Per capire come possiamo usare il tag all'interno di un testo, prendiamo in considerazione [l'esempio 2.14](#).

Esempio 2.14

```
<p>
  HTML5 include più di un centinaio<wbr />
  di elementi di markup, di cui una<wbr />
  buona parte proviene dalle versioni<wbr />
  precedenti.
</p>
```

Sebbene il tag `wbr` appaia diverse volte, il contenuto viene comunque riportato su un'unica linea nel caso in cui la larghezza del paragrafo all'interno della pagina sia superiore allo spazio occupato dal testo. In caso contrario, il testo viene suddiviso su più righe in corrispondenza di uno o più tag `wbr`.

I nuovi elementi interattivi

Oltre ai tag visti finora per la formattazione di pagine e testi, HTML5 presenta una serie di elementi orientati a fornire un certo livello di interattività. Questi elementi consentono di raggruppare controlli che permettono di eseguire azioni sulla pagina, senza la necessità di eseguire un round-trip verso il server, come avviene nel caso delle form, oggetto del prossimo capitolo. La [tabella 2.4](#) riassume brevemente le finalità di ciascuno di questi elementi.

<i>Elemento</i>	<i>Descrizione</i>
<code><details></code>	Rappresenta un insieme di informazioni o controlli aggiuntivi che possono essere mostrati a richiesta nella pagina. Elemento non presente in HTML4.
<code><summary></code>	Rappresenta l'informazione riepilogativa di un elemento <code>details</code> . Elemento non presente in HTML4.
<code><menu></code>	Rappresenta un menù contestuale, una toolbar o un elenco di comandi. Elemento non nuovo, ma ridefinito rispetto a HTML4.
<code><command></code>	Rappresenta un comando che può essere invocato dall'utente, all'interno di un elemento <code>menu</code> o della pagina. Elemento non presente in HTML4.

Tabella 2.4 – Nuovi elementi interattivi di HTML5.

Come indicato nella [tabella 2.4](#), insieme ai nuovi elementi, è presente anche il tag `menu`, che, sebbene in HTML4 sia stato dichiarato deprecato, in HTML5 assume una nuova connotazione e permette di definire una lista o un menù di comandi, in modo analogo a quanto possiamo fare con gli elenchi puntati e numerati tramite i tag `ul` e `ol`. Come avremo modo di vedere a breve, possiamo usare l'elemento `menu` per raggruppare insieme diversi tag `command` al fine di creare toolbar e menù contestuali.

Gli elementi `details` e `summary`

L'elemento `details` permette di raggruppare una serie di informazioni (ed eventualmente controlli), in modo che possano essere mostrate a richiesta

all'interno della pagina. Esso può opzionalmente includere come primo elemento figlio un tag `summary`, il cui scopo è quello di riportare un'informazione riepilogativa (una didascalia, una legenda ecc.), che descriva in qualche modo il contenuto complessivo.

L'esempio 2.15 si riferisce all'utilizzo dei due tag per mostrare un box informativo relativo al download di un file. In esso sono elencate le informazioni relative al file in fase di scaricamento, precedute dall'elemento `summary`.

Esempio 2.15

```
<section>
  <hgroup>
    <h1>HTML5</h1>
    <h2>Capitolo 2 - Nuovi elementi del markup</h2>
  </hgroup>
  <details open="open">
    <summary>Download del file in corso...</summary>
    <ul>
      <li>Nome file: capitolo2.pdf</li>
      <li>Tipologia: PDF</li>
      <li>Dimensione: 200 KB</li>
    </ul>
  </details>
</section>
```

Come possiamo notare nell'esempio 2.15, l'elemento `details` prevede un attributo booleano `open`, che, se presente, rende il contenuto del tag visibile. Chiaramente possiamo impostare l'attributo `open` anche in modo programmatico.

Una delle più interessanti modalità di impiego del tag `details` consiste nel mascherare alcuni controlli all'interno della pagina. Infatti, se non specifichiamo l'attributo booleano `open`, possiamo usare questo elemento come contenitore per una serie di controlli che, di default, non devono essere mostrati all'utente. In questo modo, possiamo rendere questi controlli visibili solamente in talune circostanze, per esempio, in risposta a certe azioni eseguite dall'utente o in base allo stato di navigazione, sostituendo in questo l'uso del tag `div`, con un conseguente aumento del livello di interattività, dell'usabilità generale e, ancora una volta, della semantica della pagina.

L'elemento menu

L'elemento `menu` consente di realizzare toolbar e menu contestuali all'interno delle pagine e va usato come un contenitore per raggruppare in modo strutturato diversi comandi. Il tag presenta due attributi, ovvero `label`, che consente di indicare un'etichetta testuale, e `type`, che permette di definire la tipologia di menù. Quest'ultimo attributo prevede tre valori possibili:

- `context`: l'elemento `menu` rappresenta un menù contestuale e l'utente può interagire solo con i comandi in esso contenuti, ma solo se il menù è attivo;
- `toolbar`: l'elemento `menu` rappresenta una toolbar (barra degli strumenti), ovvero un elenco di comandi con cui l'utente può interagire direttamente;
- `list` (default): l'elemento `menu` rappresenta una semplice lista non ordinata di elementi, ciascuno dei quali corrisponde a un comando specifico che un utente può eseguire o attivare.

In HTML5 possiamo strutturare l'insieme dei comandi contenuti in un elemento `menu` secondo modalità diverse. Possiamo usare una serie di tag `li`, in modo analogo a quanto siamo abituati a fare con gli elenchi numerati e puntati, e creare una gerarchia di menù e sottomenù, come mostrato nell'esempio 2.16.

Esempio 2.16

```
<menu type="toolbar">
  <li>
    <menu label="File" type="list">
      <button type="button" onclick="nuovo()">Nuovo</button>
      <button type="button" onclick="apri()">Apri</button>
      <button type="button" onclick="salva()">Salva</button>
    </menu>
  </li>
  <li>
    <menu label="Modifica" type="list">
      <button type="button" onclick="taglia()">Taglia</button>
      <button type="button" onclick="copia()">Copia</button>
      <button type="button" onclick="incolla()">Incolla</button>
    </menu>
  </li>
  <li>
    <menu label="Aiuto" type="list">
      <li><a href="aiuto.html">Aiuto</a></li>
      <li><a href="info.html">Informazioni</a></li>
```



```
</menu>
</li>
</menu>
```

I nuovi attributi

Oltre ai tag visti finora e a quelli descritti nei prossimi capitoli, HTML5 introduce una serie di attributi aggiuntivi per gli elementi già esistenti in HTML4. Questi attributi estendono gli elementi a cui sono associati affinché possano supportare le nuove caratteristiche introdotte nel linguaggio.

La maggior parte delle novità riguardano i tag che sono utilizzati all'interno delle form. In particolare, l'elemento `input` presenta un numero notevole di nuovi attributi orientati a migliorare le funzionalità legate all'inserimento di dati. Un discorso analogo vale anche per gli altri tag della stessa famiglia, come `textarea` o `button`. Nel prossimo capitolo avremo modo di descrivere nel dettaglio le modalità di utilizzo di questi attributi.

nota

Nella sintassi HTML, un attributo *booleano* è un tipo particolare di attributo che non prevede un valore. La semplice presenza dell'attributo all'interno dell'elemento rappresenta il valore *true* (vero). Di conseguenza, l'assenza dell'attributo dentro al tag rappresenta il valore *false* (falso).

Nella sintassi basata su XML, mutuata da XHTML e possibile anche in HTML5, all'attributo va aggiunto un valore uguale al nome stesso dell'attributo, allo scopo di rispettare le regole imposte da XML. Anche in questo caso, la presenza dell'attributo nel tag rappresenta il valore *true* (vero).

La [tabella 2.5](#) mostra l'elenco dei nuovi attributi di HTML5 in ordine alfabetico. Tutti questi attributi sono associati a uno o più tag e hanno una valenza che è specifica per ognuno degli elementi a cui sono associati.

Attributo	Elemento/i
async	<script>
autocomplete	<input>

autofocus	<input>, <select>, <textarea>, <button>
charset	<meta>
disabled	<fieldset>
placeholder	<input>, <textarea>
dirname	<input>, <textarea>
form	<input>, <select>, <textarea>, <button>, <fieldset>
formaction	<input>, <button>
formenctype	<input>, <button>
formmethod	<input>, <button>
formnovalidate	<input>, <button>
hreflang	<a>
label	<menu>
list	<input>
manifest	<html>
max	<input>
media	<a>, <area>
min	<input>
multiple	<input>
novalidate	<form>
pattern	<input>
rel	<area>
required	<input>, <textarea>
reserved	
sandbox	<iframe>
scoped	<style>
seamless	<iframe>
sizes	<link>
srcdoc	<iframe>
start	
step	<input>
target	<a>, <area>, <base>

type	<menu>
value	
start	

Tabella 2.5 – Nuovi attributi di HTML5.

In aggiunta agli attributi riportati nella [tabella 2.5](#), HTML5 definisce una classe di attributi particolari, alcuni dei quali provenienti da HTML4, dando loro una valenza del tutto generale. Questi attributi sono chiamati *attributi globali*.

Gli attributi globali sono implicitamente associati a tutti gli elementi del linguaggio e, diversamente dai precedenti, possono essere utilizzati in tutti i contesti.

Tra questi figurano alcune “vecchie conoscenze” come `accesskey`, `class`, `dir`, `id`, `lang`, `style`, `tabindex` e `title`. Agli attributi appena menzionati si aggiungono alcune novità, tra cui `contenteditable` (per indicare che un elemento si trova in un’area editabile), `contextmenu` (per associare a un elemento un menù contestuale), `draggable` e `dropzone` (per la gestione del drag&drop), `hidden` (per indicare se un elemento è ancora rilevante nel suo contesto di impiego) e `spellcheck` (per il controllo ortografico).

ARIA e gli attributi per l’accessibilità

Accanto agli attributi descritti nel paragrafo precedente, HTML5 include una serie di attributi orientati a favorire l’accessibilità delle pagine, aspetto totalmente rivalutato nella nuova versione del linguaggio.

Il numero limitato di elementi di interfaccia presenti in HTML e la natura stateless del Web, che costringe a un modello di comunicazione sequenziale tra il client e il server, hanno da sempre rappresentato un forte limite in termini di interazione utente. Per aggirare questi problemi, in questi anni gli sviluppatori di applicazioni web hanno sfruttato JavaScript e tecnologie come AJAX, in modo tale da creare un modello di interazione più evoluto, molto simile a quello presente nelle normali applicazioni desktop. Tuttavia, le tecniche impiegate per elevare il livello di interazione utente hanno introdotto non pochi problemi dal punto di vista dell’accessibilità. Per fare un esempio, un elemento dell’interfaccia utente (widget) basato su AJAX usa un modello di interazione asincrono tra il client e il server, che non consente a una tecnologia di supporto (in inglese, Assistive Technology), come uno screen

reader, di accorgersi dei cambiamenti di stato del widget. Inoltre l'elemento non è accessibile da tastiera e il suo ruolo all'interno della pagina non è determinato.

nota

Il *widget* è un componente grafico dell'interfaccia di un'applicazione, il cui scopo è quello di facilitare all'utente l'interazione con l'applicazione stessa. Esempi tipici di widget sono i pulsanti, le caselle di spunta, i menù, le combo, gli slider, le scrollbar ecc.

HTML5 affronta questo genere di problemi attraverso WAI-ARIA (o, più semplicemente, ARIA), acronimo di **Web Accessibility Initiative - Accessible Rich Internet Applications**. ARIA è un insieme di specifiche che fornisce un metodo per indicare il ruolo, lo stato e le proprietà di ogni parte di una pagina, in modo che possa essere interpretata da una tecnologia di supporto. In tal senso, ARIA definisce una serie di attributi in grado di rendere evidenti, anche per gli utenti di sistemi come gli screen reader, i cambiamenti di stato e il ruolo degli elementi presenti nelle pagine.

L'attributo `tabindex`

Introdotta in HTML4 per un numero limitato di tag, allo scopo di consentire l'accesso da tastiera agli elementi di input della pagina, in HTML5 l'attributo `tabindex` è stato del tutto rivisto. Infatti, ARIA estende questo attributo, rendendolo utilizzabile per tutti gli elementi visibili nella pagina (come detto nel paragrafo precedente, `tabindex` è un attributo globale), e per questo attributo consente di specificare non solo valori positivi, come in passato, ma anche il valore negativo “-1”. In questo modo, l'ordine di accesso ai controlli può essere gestito in due modi:

- un valore da 0 a 32767 indica che l'elemento, a cui l'attributo è associato, può essere accessibile in base all'ordine indicato dal valore stesso;
- il valore negativo “-1” indica che l'elemento può essere attivato tramite la funzione JavaScript `focus()`, ma non può essere accessibile

attraverso l'utilizzo della tastiera.

Il valore predefinito pari allo zero rappresenta l'ordine naturale di tabulazione. Gli elementi con valore nullo vengono visitati in base all'ordine di apparizione all'interno della pagina.

L'attributo role

ARIA introduce l'attributo `role` per definire cosa esattamente un elemento (e, in particolare, un widget) rappresenta all'interno della pagina.

Gli elementi HTML hanno già un ruolo di base definito. Lo stesso non si può dire per i widget creati appositamente per gestire l'interazione utente in modo più evoluto. Per esempio, se utilizziamo un'icona denominata "indicatore" per creare un widget di tipo slider, questa perde il significato di immagine e assume un ruolo diverso, ovvero quello di indicare un valore numerico. Pertanto, se specifichiamo il ruolo `slider`, uno screen reader interpreterà l'elemento come uno slider con un certo valore e non come una semplice immagine di nome "indicatore".

Pertanto, l'attributo `role` serve per assegnare a un tag un ruolo diverso da quello di base, in modo che, da un punto di vista semantico, l'elemento possa essere interpretato in modo opportuno dalle tecnologie di supporto.

All'interno delle specifiche di ARIA, è definito un certo numero di ruoli, alcuni specifici per i widget come `slider`, `scrollbar` oppure `treeitem`, altri pensati per aiutare a comprendere meglio la struttura della pagina. Questi ultimi, detti *document landmark*, consentono agli screen reader di capire il ruolo di ciascuna sezione della pagina (per esempio, `banner`, `search` oppure `navigation`) e di agire di conseguenza, al fine di migliorare l'accesso ai contenuti.

Gli stati e le proprietà ARIA

Gli attributi di tipo `aria-xyz` rappresentano gli stati e le proprietà ARIA relativi a un widget e consentono di fornire alle tecnologie di supporto una serie di informazioni utili per gestire al meglio l'accessibilità. Gli stati e le proprietà ARIA sono elencati nella [tabella 2.6](#). Gli elementi contrassegnati con l'asterisco corrispondono agli stati.

<code>aria-autocomplete</code>	<code>aria-checked (*)</code>	<code>aria-disabled (*)</code>
--------------------------------	-------------------------------	--------------------------------

aria-expanded (*)	aria-haspopup	aria-hidden (*)
aria-invalid (*)	aria-label	aria-level
aria-multiline	aria-multiselectable	aria-orientation
aria-pressed (*)	aria-readonly	aria-required
aria-selected (*)	aria-sort	aria-valuemax
aria-valuemin	aria-valuenow	aria-valuetext

Tabella 2.6 – Stati e proprietà ARIA relativi ai widget.

Per capire meglio l’uso di questi attributi particolari, prendiamo in considerazione [l’esempio 2.17](#).

Esempio 2.17

```
<input type="image"
      src="indicatore.png"
      alt="Indicatore"
      role="slider"
      aria-valuemin="0"
      aria-valuemax="100"
      aria-valuenow="50"
      aria-valuetext="50%" />
```

L’elemento presente [nell’esempio 2.17](#) rappresenta un widget in cui sono stati utilizzati alcuni attributi per l’accessibilità. L’elemento `input` di tipo `image`, che appare come un’immagine, in realtà assume il ruolo `slider` e le quattro proprietà ARIA presenti all’interno del tag forniscono l’indicazione del valore corrente assunto dal widget nell’intervallo che va da 0 a 100. Il valore corrente è 50, rappresentato dalla stringa “50%”.

Le live region

Oltre a quanto visto finora, ARIA permette di definire le cosiddette *live region*, ossia consente a ciascun elemento presente all’interno di una pagina di notificare i suoi cambiamenti di stato, senza che l’utente perda il focus corrente. Un esempio di live region è rappresentato da una sezione in continuo aggiornamento, in cui sono mostrate informazioni in tempo reale, come i titoli azionari o le informazioni sul traffico.

Gli attributi che riguardano le live region sono quattro, ovvero le proprietà `aria-atomic`, `aria-live` e `aria-relevant` e lo stato `aria-busy`.

Ulteriori dettagli su come utilizzare i vari attributi di ARIA descritti in questo paragrafo sono disponibili online presso il sito di W3C, all'indirizzo Internet: <http://aspit.co/a0w>.

Conclusioni

Il problema della formattazione delle pagine e dei contenuti in HTML5 è stato affrontato in modo molto diverso rispetto al passato.

Da una parte sono stati introdotti una serie di elementi nuovi, orientati a fornire al contenuto delle pagine una più precisa connotazione, rendendo la struttura del markup totalmente autodescrittiva, a tutto vantaggio dei motori di ricerca. Dall'altra parte sono stati eliminati tutti quegli elementi e attributi il cui scopo, in passato, era strettamente legato alla rappresentazione e alla resa visiva dei contenuti. Con HTML5 tutto quello che riguarda l'estetica delle pagine deve essere gestito unicamente tramite CSS.

Il risultato è un linguaggio di markup più snello, meno caotico e orientato principalmente a definire la struttura dei contenuti e a garantire un livello di interazione senza precedenti, favorendo l'aspetto semantico del Web, così come gli inventori dell'HTML avrebbero voluto fin dall'inizio.

Come abbiamo visto nel corso del capitolo, molti elementi presenti in HTML4 sono stati ripensati o modificati. Molti di questi cambiamenti riguardano i tag che solitamente sono usati all'interno delle form per gestire l'inserimento di dati, aspetto cruciale nella realizzazione delle applicazioni web e dei siti moderni.

Nel prossimo capitolo andiamo a vedere le numerose novità che riguardano proprio le form, illustrando le modalità tramite cui possiamo trarre il massimo beneficio dalle nuove caratteristiche di HTML5, al fine di aumentare l'interattività all'interno delle pagine e sfruttare al meglio i diversi tipi di browser in circolazione, con un occhio di riguardo in maniera specifica al mobile.

Le form

Dopo aver esaminato i concetti fondamentali di HTML5 e le novità che riguardano gli aspetti semantici delle pagine, è arrivato il momento di dedicarci a come implementare le funzionalità legate all'interazione tra gli utenti e le applicazioni web attraverso le form. Le form sono utilizzate nell'HTML per fornire la possibilità all'utente di inviare contenuti al server: quest'ultimo, poi, generalmente le immagazzina perché vengano elaborate. Da un punto di vista pratico, la parte che risiede sul server prende in input i dati e li processa, rispedendoli all'utente, così che quest'ultimo li possa visualizzare a video in una forma probabilmente diversa dall'originale.

In quasi tutti i casi, la parte che si trova sul client, quella fatta di HTML, è composta da una serie di tag che aggiungono i componenti tipici legati all'inserimento dei dati: textbox (caselle di testo), liste a selezione multipla ecc.

HTML5 introduce una serie di novità in questo ambito, che consentono di sviluppare form che abbiano maggiore interattività con l'utente e sfruttino al meglio i diversi tipi di browser presenti sul mercato. Prima di addentrarci in maniera specifica nelle novità, diamo un'occhiata a come strutturare la pagina, in maniera da poter trarre il massimo beneficio dall'interazione tra client e server.

Principi di form

Nella sua incarnazione più semplice, una form si presenta attraverso una serie di tag, tra cui spicca senza dubbio quello di nome `form`, che prende lo stesso nome della funzionalità. [Nell'esempio 3.1](#) viene riportata una semplice form, con un campo che accetta l'input.

Esempio 3.1

```
<form>
  <label>Inserisci il tuo nome:
  <input name="name" />
```



```
</label>  
</form>
```

Come si può notare, abbiamo impiegato tre tag:

- `form`: racchiude all'interno la form stessa;
- `label`: indica una descrizione associata al campo;
- `input`: consente l'inserimento di testo.

Particolarmente interessante è il tag `input`, che, attraverso la proprietà `type`, consente di assegnare un comportamento visuale differente al tag stesso. Di default, questo valore è impostato su `text` e avremo modo di analizzarne meglio le caratteristiche nel resto del capitolo. Possiamo specificare un valore predefinito, impostando la proprietà `value`, mentre quella `name` è utilizzata, lato server, per ricavarne il rispettivo valore specificato nel browser. Così come per tutti i tag, poi, possiamo specificare una proprietà `id`, per l'interazione con la parte di JavaScript.

Il tag `form` consente di aggiungere una proprietà `action`, che serve per specificare la pagina a cui verrà inviato il contenuto della form, e una proprietà `method`, che invece indica il *verb* dell'HTTP (*GET* o *POST*). Se queste proprietà vengono omesse, il contenuto della form sarà inviato all'URL corrente attraverso il metodo *POST*. Il metodo *GET* è indicato per URL rapidi e generalmente non è sfruttato con le form, perché prevede l'invio dei dati in *querystring* (visibile nell'URL del browser) ed è limitato in quanto a dimensione. Il metodo *POST*, viceversa, invia i dati all'interno del corpo della richiesta ed è, generalmente, preferibile quando ci sono molte quantità di dati da inviare, oppure non si vuole semplicemente sfruttare il metodo *GET* per questioni di opportunità.

Le caratteristiche analizzate finora ci consentono di costruire una semplice form, ma come funziona l'invio dei dati al server?

Inviare il contenuto

Generalmente, una form è inviata sfruttando un pulsante, che nell'HTML corrisponde al tag `input` con l'attributo `type` impostato sul valore `submit`. [L'esempio 3.2](#) mostra una form che fa uso di questa funzionalità.

L'attributo `value`, in questo caso, riveste un ruolo importante, perché indica il testo che sarà visualizzato dal pulsante stesso, come è visibile nella

figura 3.1.

Esempio 3.2

```
<form>
  <input type="submit" value="Invia form" />
</form>
```

Demo



Figura 3.1 – Ecco come appare una semplice form quando è visualizzata all'interno del browser.

Si possono inserire più pulsanti all'interno di una form, per simulare azioni differenti. In questo caso dobbiamo inserire l'attributo `name` sul tag, così da poter poi recuperare questa informazione lato server, per capire quale dei pulsanti è stato premuto.

nota

Oltre all'uso del valore `submit`, possiamo specificare anche il valore `button`. In questo caso, il pulsante non effettua l'invio della form, ma può essere utilizzato per associare eventuale codice lato client alla relativa pressione.

Come comportamento di default, ogni browser (a seconda anche del sistema operativo) visualizza in maniera differente il pulsante, per adattarsi all'aspetto del sistema su cui la pagina viene visualizzata. Questa visualizzazione può essere variata semplicemente, agendo attraverso i CSS, che verranno illustrati nel [capitolo 4](#).

Gestire scelte multiple

HTML prevede un meccanismo semplice per inserire elenchi con scelte multiple, agendo sulla proprietà `type` del tag `input`.

L'esempio 3.3 introduce l'argomento.

Esempio 3.3

`<form>`

```
<fieldset>
  <legend>Le tue preferenze: </legend>
  <p><label><input type="checkbox" value="1" name="pref"/>
    1</label></p>
  <p><label><input type="checkbox" value="2" name="pref"/>
    2</label></p>
  <p><label><input type="checkbox" value="3" name="pref"/>
    3</label></p>
</fieldset>

<fieldset>
  <legend>La tua nazione: </legend>
  <p><label><input type="radio" value="1" name="country"
    checked="checked" /> Italia</label></p>
  <p><label><input type="radio" value="1" name="country " />
    USA</label></p>
  <p><label><input type="radio" value="1" name="country " />
    Canada</label></p>
</fieldset>
```

`</form>`

Possiamo notare, a questo punto, che il tag `fieldset`, insieme a quello `legend`, serve al browser per visualizzare una descrizione da associare ai due elenchi. A seconda che utilizziamo il valore `radio`, piuttosto che `checkbox`, il tag `input` è in grado di consentire o meno una scelta multipla tra le opzioni specificate. L'attributo `name`, in tal senso, serve a garantire che le diverse opzioni vengano unite tra loro e, pertanto, deve avere lo stesso valore.

La [figura 3.2](#) mostra come sono renderizzate da un browser queste due liste differenti.

The image shows a web form with a header bar labeled 'Forms'. Below the header, there are two sections. The first section, 'Le tue preferenze:', contains three checkboxes labeled 1, 2, and 3. The second section, 'La tua nazione:', contains three radio buttons labeled Italia, USA, and Canada. The 'Italia' radio button is selected. At the bottom of the form, there is a footer that reads 'HTML5Italia.com | HTML5'.

Figura 3.2 – La differenza legata all’uso dei tipi di tag appare evidente in questa immagine.

L’attributo booleano `checked`, se presente, indica che l’opzione è selezionata.

Benché questi tipi di oggetto che consentono scelte multiple o guidate siano utili in molti scenari, presentano alcuni limiti quando è necessario visualizzare un elenco di voci composto da molte opzioni. In questi casi è preferibile utilizzare i cosiddetti elenchi a discesa.

Elenchi a discesa

La possibilità di inserire un numero elevato di valori è gestita attraverso il tag `select`. Questo tag è in grado di consentire sia la sezione multipla sia quella singola, all’interno di una lista di valori, specificati attraverso il tag `option`. L’esempio 3.4 mostra un primo esempio di come creare la lista.

Esempio 3.4

```
<select name="group" size="3">
  <optgroup label="Gruppo A">
    <option value="A.1">valore 1</option>
    <option selected="selected" value="A.2">valore 2</option>
    <option value="A.3">valore 3</option>
  </optgroup>
  <optgroup label="Gruppo B">
```

```
<option value="B.1">valore 1</option>
<option value="B.2">valore 2</option>
<option value="B.3">valore 3</option>
</optgroup>
</select>
```

Un browser visualizzerà il tag come mostrato nella [figura 3.3](#).

Demo



Figura 3.3 – Il tag `select`, unito a `optgroup` e `option`, consente di specificare facilmente elenchi a discesa.

Possiamo subito cogliere un paio di spunti:

- l'attributo `size` del tag `select` (di default è impostato su 1) regola l'altezza della lista;
- possiamo racchiudere le opzioni all'interno di un tag `optgroup`, che provvede a fornire una visualizzazione più ordinata, comoda quando abbiamo liste con opzioni che possono essere raggruppate, per esempio, per categoria;
- per selezionare un valore nella lista, è sufficiente inserire l'attributo booleano `selected`.

Questo tag supporta la selezione multipla, quando l'attributo `multiple` viene specificato. In questo caso, diventa possibile utilizzare il tasto `CONTROL` per selezionare più di un elemento alla volta.

A questo punto, abbiamo analizzato le tipologie principali per specificare informazioni attraverso una lista: continuiamo il discorso analizzando le altre opzioni possibili quando si lavora con le form.

Inserire testo su più righe

Un caso particolare è rappresentato dal controllo `textarea`, che consente di inserire il testo su più righe, come mostrato nell'esempio 3.5.

Esempio 3.5

```
<textarea name="description" maxlength="200"
          cols="100" rows="5">testo</textarea>
```

Con questo tag, il testo va inserito all'interno dell'elemento stesso, mentre per recuperare il valore scritto dall'utente, nella parte server-side, dovremo implementare lo stesso codice associato a qualsiasi altro tipo di tag che venga utilizzato nella form.

L'attributo `maxlength` (disponibile anche sul tag `input`) indica la lunghezza massima del testo inseribile dall'utente. Gli attributi `cols` e `rows`, invece, indicano rispettivamente il numero di righe e colonne occupati dall'elemento visuale. Possiamo anche specificare la dimensione utilizzando uno stile dentro un CSS. Il risultato è visibile nella figura 3.4.

L'uso di questo tag è generalmente consigliato quando abbiamo blocchi lunghi di testo. In altri scenari, invece, possono tornare utili alcuni dei tag che introdurremo di seguito.

Demo



The screenshot shows a web browser window with a title bar. Inside, there's a header area with a grey background and the word "Forms" in a white, italicized serif font. Below the header is a large, multi-line text input field. The field has a light grey border and contains the text "testo" in a small, dark font. At the bottom of the browser window, there's a status bar with the text "HTML5Italia.com | HTML5".

Figura 3.4 – Il tag `textarea` può consentire l'inserimento di più righe di testo, facilitando gli scenari in cui è necessario specificare grandi quantità di testo.

Uso avanzato delle form

Prima di HTML5, il tag `input`, attraverso il suo attributo `type`, era utilizzato in queste modalità, oltre che in quelle già citate:

- `password`: mostra un campo per l'inserimento di password;
- `file`: consente l'invio di file in remoto (necessita di un opportuno script lato server);
- `hidden`: consente l'invio di campi nascosti (utili per inviare informazioni calcolate o che, comunque, l'utente non ha interesse a visualizzare).

Storicamente, questo è stato uno dei problemi più sentiti in HTML, perché le interfacce si sono evolute molto dalle specifiche iniziali: oggi giorno è del tutto normale inserire email, URL, date o orari e trovare aiuto (per esempio, attraverso tastiere virtuali) mentre digitiamo le informazioni. Questa caratteristica è data per scontata nei device mobili, molto in voga in questo periodo, come **smartphone** o **tablet**. In questi scenari, appunto, viene mostrata una **tastiera virtuale**, perché è assente quella fisica.

Le tastiere virtuali sono ottimizzate per tipologia di contenuto che viene inserito e, per questo motivo, i nuovi tipi di input sono utili soprattutto quando si devono sfruttare browser legati agli ambiti appena descritti. Iniziamo a dare un'occhiata a questi nuovi elementi.

Inserire numeri di telefono, e-mail e URL

Un ambito in cui può tornare comodo l'utilizzo di un tipo specifico per inserire i corrispondenti valori attraverso una tastiera virtuale è quello rappresentato da numeri di telefono, da e-mail o da URL. In questi casi, infatti, la tastiera deve mostrare un numero specifico di tasti, anziché quelli generici, così da agevolare l'utente durante l'inserimento. L'uso di questi valori è mostrato [nell'esempio 3.6](#).

Esempio 3.6

```
<p><label>Telefono: <input name="phone" type="tel" /></label></p>
<p><label>E-mail: <input name="email" type="email" /></label></p>
<p><label>Sito: <input name="website" type="url" /></label></p>
```

Su un browser per desktop, a meno di non utilizzare particolari plug-in,

questo markup produrrà una normale serie di campi in cui inserire valori. Tra l'altro, per come funzionano i browser, questo codice è compatibile anche con browser che non abbiano il supporto per queste specifiche: in questi casi l'attributo `type` viene ignorato e al suo posto viene visualizzato un normale box per l'inserimento di testo. La [figura 3.5](#) riporta un esempio di visualizzazione di questi tag in Safari per iPhone.

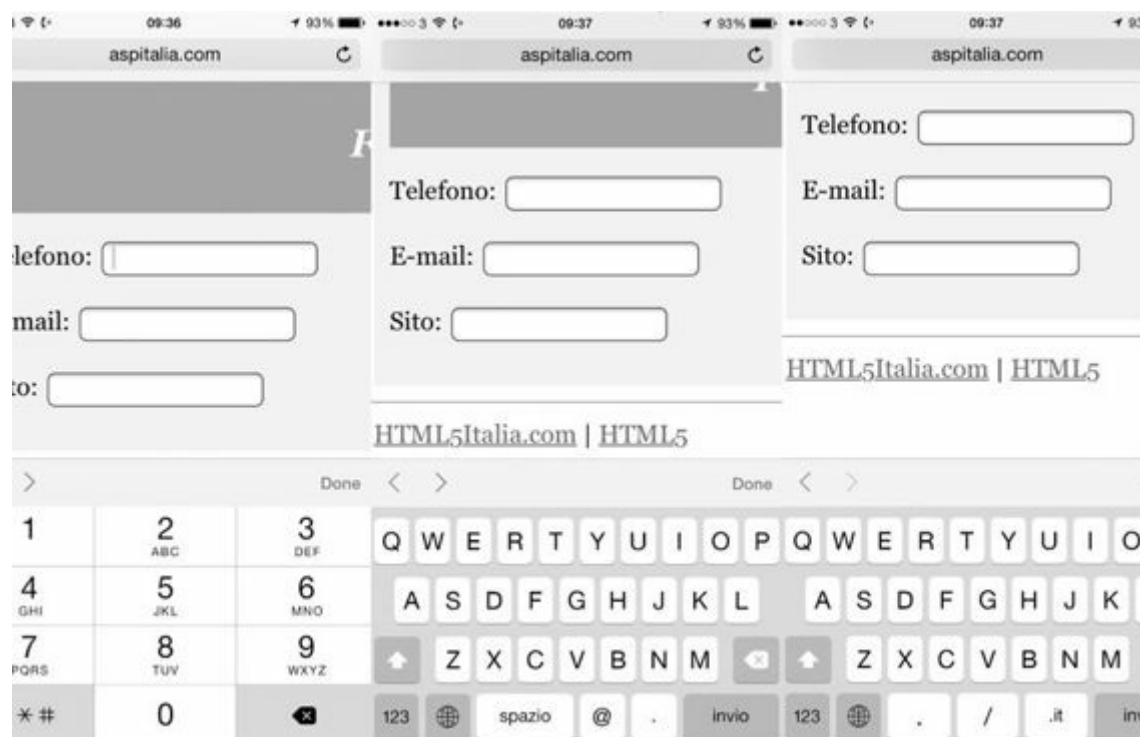


Figura 3.5 – Da destra verso sinistra, si possono notare le tastiere virtuali associate a un campo URL, E-mail e Telefono.

Una particolarità dell'uso della variante `email` consiste nella possibilità di specificare più indirizzi e-mail, attraverso l'inserimento dell'attributo `multiple` (già visto per il tag `select`), avendo cura di separare gli indirizzi con una virgola.

nota

Quasi tutti i browser applicano dei meccanismi di validazione automatica dei valori, inseriti in questi campi. Questo aspetto non è espressamente previsto dalle specifiche e consigliamo di provvedere comunque a una validazione dei dati lato client e, soprattutto, lato server. Maggiori informazioni su questo aspetto sono disponibili alla

fine di questo capitolo.

Il supporto per queste funzionalità sui browser desktop è al momento abbastanza scarso e si limita spesso solo a una validazione formale dei dati inseriti, ma questo non rappresenta un problema, dato che danno un valore aggiunto soprattutto nell'ambito dei browser per dispositivi mobili.

Inserire numeri e orari

Per gli stessi motivi analizzati in precedenza, in HTML5 sono disponibili delle varianti specifiche per gestire orari, date e numeri.

Partendo dai numeri, specificando `number` come valore dell'attributo `type` su un tag `input`, possiamo aggiungere un elemento visuale per agevolare l'inserimento di numeri.

Esempio 3.7

```
<p><label>Indica la misura:
  <input name="size"
    type="number" min="10" max="60" step="2" value="42" /></label>
</p>
```

Il tag visualizzerà un elemento visuale, detto *spin box*, che consentirà di inserire un valore numerico. Attraverso gli attributi `min`, `max` e `step` possiamo rispettivamente indicare un valore minimo, massimo e l'incremento. Nel nostro caso, potremo specificare un valore compreso tra 10 e 60, multiplo di 2. La [figura 3.6](#) mostra il risultato.

Demo



The screenshot shows a web browser window with a title bar. The page has a header area with the word "Forms" in a stylized font. Below the header, there is a form with a label "Indica la misura:" followed by a numeric input field (spin box) containing the number "46". The input field has small up and down arrows on its right side. At the bottom of the browser window, the address bar shows "HTML5Italia.com | HTML5".

Figura 3.6 – Attraverso questa variante, diventa possibile far inserire valori sia

a mano sia utilizzando i pulsanti per incrementare o decrementare il risultato.

Invocando i metodi `stepUp` e `stepDown` da JavaScript, diventa possibile incrementare o decrementare il valore del campo anche in maniera programmatica, cioè da codice client-side.

Discorso analogo può essere fatto per i cosiddetti slider, che possiamo attivare impostando l'attributo `type` dell'esempio 3.7 sul valore `range`. Come possiamo notare nella figura 3.7, il risultato visuale cambierà.

Demo



The image shows a web browser window with a title bar. Inside, there's a header area with the word "Forms" in a stylized font. Below the header, there's a form with a label "Indica la misura:" followed by a range slider. The slider has a circular handle and a numerical value of 0. At the bottom of the browser window, there's a footer that reads "HTML5Italia.com | HTML5".

Figura 3.7 – Attraverso uno slider, in talune circostanze, l'utente può inserire i valori utilizzando un'interfaccia più usabile, specie se non rappresenta un vantaggio visualizzare l'effettivo valore.

Infine, merita un discorso a parte la gestione di orari e date. Attualmente, solo Opera e (parzialmente) Chrome gestiscono bene questa parte delle specifiche, aggiungendo, come possiamo aspettarci in questo caso, un comodo calendario per inserire le date, senza ricorrere a plug-in esterni per farlo. In questo caso, occorre impostare l'attributo `type` su uno dei valori che seguono:

- `date`: consente di scegliere una data;
- `month`: per specificare il mese;
- `week`: per selezionare la settimana;
- `time`: per inserire un orario;
- `datetime`: per inserire data e ora;
- `datetime-local`: per inserire data e ora locali del browser.

Non ci soffermeremo oltre su questo aspetto, poiché attualmente il relativo supporto è molto limitato. Occorre notare che Chrome si aspetta sempre una data nel formato `YYYY-MM-DD` e questo è un problema per gli altri browser. Un approccio che spiega come risolvere il problema, grazie a Modernizr (introdotto nel [capitolo 1](#)) e un po' di JavaScript, è disponibile su <http://aspit.co/aod>.

Un campo specifico per la ricerca

Esiste la possibilità di impostare l'attributo `type` su `search`, per supportare al meglio le ricerche. In realtà, si tratta di una feature attualmente implementata solo da WebKit (l'engine su cui sono basati Chrome e Safari), che aggiunge un pulsante per resettare il contenuto della ricerca rapidamente.

Più realisticamente, questa funzionalità potrebbe essere utile per dare maggiore significato semantico alla pagina perché, quanto meno a livello funzionale, attualmente nessun browser si contraddistingue per un uso particolare.

Altri elementi

Ci sono alcuni nuovi tag che aggiungono il supporto a funzionalità interessanti ma che, come per altri casi che abbiamo analizzato finora, non sono supportati da tutti i browser e sui quali non ci soffermeremo troppo.

In particolare, occorre menzionare il tag `progress`, che consente di inserire nativamente una progress bar, come possiamo vedere [nell'esempio 3.8](#).

Esempio 3.8

```
<progress value="25" max="100">25%</progress>
```

Il risultato è visibile nella [figura 3.8](#).



Figura 3.8 – Il tag `progress` consente di visualizzare facilmente una progress bar all'interno della pagina.

L'uso ottimale di questo tag è in unione con JavaScript, dove il valore è cambiato dinamicamente.

Una menzione la meritano anche i seguenti nuovi tag:

- `meter`: simile a `progress` ma utilizzabile per inserire misurazioni, come per esempio l'uso dell'hard disk, la votazione di un articolo ecc;
- `keygen`: consente di inserire un campo che genera una chiave pubblica, salvata sul client, e una privata, inviata al server.

Eventuali approfondimenti su questi tag sono comunque disponibili nella documentazione ufficiale del W3C su <http://w3.org/html/>.

Mostrare un segnaposto nei campi

Quante volte vi siete trovati nella situazione in cui avreste dovuto mostrare un valore all'interno di un campo, per poi nascondere al click dell'utente per far digitare l'effettivo valore? Questo task, seppur non complesso, richiede un po' di codice JavaScript, necessario a intercettare il focus sull'elemento e la relativa perdita, per controllare se un testo è stato inserito oppure no.

HTML5 introduce un nuovo attributo, chiamato `placeholder`, che consente di specificare facilmente un segnaposto, come nell'esempio 3.9.


Esempio 3.9

```
<p><label>La tua e-mail:
```

```
  <input type="email" placeholder="daniele@aspitalia.com" />
</label></p>
```

Questo markup produrrà un effetto come quello mostrato nella [figura 3.9](#), senza necessità di aggiungere JavaScript.

Demo



The screenshot shows a web page with a header area containing the word "Forms" in a stylized font. Below the header is a form with a label "La tua e-mail:" followed by a text input field. The input field contains the text "daniele@email.mail", which serves as a placeholder. At the bottom of the page, there is a footer with the text "HTML5Italia.com | HTML5".

Figura 3.9 – L'attributo placeholder può fornire un aiuto all'utente nel compilare correttamente il contenuto della form.

Come per altri casi, se vogliamo che funzioni anche con browser vecchi, occorre continuare a utilizzare un meccanismo di fallback basato su JavaScript.

Gestire l'autocomplete

Un caso decisamente analogo a quello del placeholder è rappresentato dall'autocomplete, cioè dalla possibilità di disabilitare l'aiuto nel completare un modulo mentre scriviamo.

Potremmo, per esempio, voler inibire questo comportamento per gestire la privacy dell'utente, in campi che contengono dati sensibili, come lo username o il numero di carta di credito. Basta agire sull'attributo `autocomplete`, come mostrato [nell'esempio 3.10](#).

Esempio 3.10

```
<input type="text" autocomplete="off" />
```

Questo tag è inseribile anche all'interno del tag `form`, con l'effetto che tutti gli input al suo interno avranno questo comportamento. Di default il valore di questo attributo è impostato su `on`. Questo attributo non è una novità di HTML5.

Uso del tag `datalist`

Un elemento particolarmente interessante, introdotto dalle specifiche, è `datalist`. Si tratta di un elemento che consente di aggiungere un elenco di valori per un certo campo, che rappresentano le possibilità di inserimento di un valore a mò di autocomplete, lasciando però all'utente la possibilità di inserire un valore non incluso nella lista, a differenza di quanto avviene con gli altri tag che offrono funzionalità di questo tipo, come `select`.

L'esempio 3.11 mostra come fare.

Esempio 3.11

```
<label>La tua provincia:
  <input type="text" name="province" list="provinces" />
  <datalist id="provinces">
    <option value="AG">Agrigento</option>
    <option value="AL">Alessandria</option>
    <option value="AN">Ancona</option>
    ...
  </datalist>
</label>
```

La figura 3.10 contiene il risultato.



Figura 3.10 – L'uso del `datalist` consente di migliorare l'usabilità dei campi, aiutando l'utente durante l'inserimento. Da sinistra a destra: IE, Chrome e Firefox.

Come si può notare, il risultato finale del rendering varia in funzione del browser. Questo approccio è indicato per liste non molto lunghe, poiché i dati devono essere renderizzati subito lato client.

Focus automatico sugli elementi

Una variante sui temi precedenti è rappresentata dal focus automatico su un elemento. Anche in questo caso, è possibile arrivare allo stesso risultato con JavaScript, che resta la via da preferire quando vogliamo supportare anche browser non aggiornati a HTML5. In tutti gli altri casi, per dare automaticamente il focus a un elemento, basterà aggiungere l'attributo `autofocus`.

La nostra panoramica sugli elementi disponibili nelle form è quasi completata. Il prossimo e ultimo argomento è molto gettonato tra gli sviluppatori: vedremo come gestire la validazione delle form con HTML5.

Validazione delle form

La convalida (o validazione) dei dati delle form rappresenta uno degli aspetti più noiosi per uno sviluppatore, ma ugualmente centrale per l'usabilità di una pagina. Dare un feedback visuale all'utente, relativo allo stato dei dati inseriti nella pagina, infatti, lo aiuta a riempire correttamente i dati, migliorandone la sensazione d'uso.

Senza queste specifiche (che, lo ripetiamo ancora una volta, non sono definitive al momento di andare in stampa), è necessario sviluppare codice client-side in JavaScript, che effettui le operazioni di convalida. Peraltro, sia questa nuova modalità sia quella con codice client-side non devono assolutamente sostituire un controllo server side, perché l'unico modo certo di garantire che i dati siano conformi è quello di controllarli in un ambiente dove l'utente non possa alterarli.

Fermo restando che stiamo parlando di convalida lato client e solo a fini di usabilità, le novità in tal senso sono interessanti. Partiamo ad analizzarle una per volta.

Rendere obbligatorio un campo

La casistica più diffusa resta quella relativa all'obbligatorietà di un campo. In tal senso, è sufficiente specificare l'attributo `required` su uno dei tag all'interno della form. [Nell'esempio 3.12](#) viene mostrato come rendere obbligatori due campi.

Esempio 3.12

```
<p><label>La tua e-mail:
    <input type="email" required="required" /></label></p>
<p><label>Il tuo telefono:
    <input type="tel" required /></label></p>
```

Grazie al fatto che le specifiche di HTML5 non obbligano più a utilizzare una sintassi XML, entrambe le varianti sono funzionanti.

L'effetto prodotto varia da browser a browser, ma possiamo intervenire facilmente grazie ai CSS (introdotti con maggior dettaglio nel capitolo che segue), che nella nuova versione hanno un selettore ad hoc, mostrato [nell'esempio 3.13](#).

Esempio 3.13

```
<style type="text/css"><!--
    :focus:invalid { background-color:red; color:white;}
    :valid { background-color:green; }
    :required {border: 2px solid red;}
    :optional {border: 2px solid green;}
--></style>
```

Attraverso i selettori riportati nell'esempio precedente, possiamo fornire uno sfondo colorato che aiuti l'utente a capire quando un campo è valido (:valid), quando è obbligatorio (:required), opzionale (:optional) e non valido con il focus dell'utente (:focus:invalid). La [figura 3.11](#) mostra il risultato che otteniamo applicando queste nuove funzionalità.

L'aggiunta di questa proprietà fa scattare il meccanismo di validazione sull'intera form. In caso contrario, la form invia normalmente i dati che contiene. Seppure interessante, questa tecnica ha il limite che non possiamo specificare il formato dei valori accettati dal campo. Vediamo, quindi, come fare un salto avanti in tal senso.

Demo



The image shows a web form titled "Forms". It contains three input fields: "La tua e-mail:" with a grey background, "Il tuo telefono:" with a white background, and "Un campo non obbligatorio" with a grey background. Below these fields is a "Invia dati" button.

Figura 3.11 – Possiamo notare i diversi colori applicati ai vari stati in cui si trovano gli elementi della form.

Specificare pattern complessi

In molti casi non basta assolutamente richiedere che un campo sia obbligatorio perché, per esempio, ci aspettiamo che il valore che contiene rispetti criteri più complessi, come può essere il caso di una data o di uno username. Per questo motivo, è possibile specificare una proprietà `pattern`, che accetta una *regular expression* come argomento. Nell'esempio 3.14, il nostro campo viene controllato con un pattern complesso, che richiede almeno 6 caratteri e non più di 9.

Esempio 3.14

```
<input type="text" required="required"
      placeholder="Username con minimo 6 e massimo 9 caratteri"
      pattern="\w{6,9}" />
```

Se applichiamo lo stesso CSS introdotto nel caso precedente, l'effetto sarà quello di visualizzare in maniera diversa il controllo quando la condizione non è rispettata.

Indicare un messaggio di errore

Purtroppo (e questo è un attuale limite), non è possibile associare un messaggio di errore specifico senza scrivere codice JavaScript. L'esempio 3.15

mostra come intercettare gli eventi `oninvalid` e `oninput`, che si verificano rispettivamente quando il campo non è valido e quando viene inserito l'input, per agire, grazie all'uso della funzione `setCustomValidity`.

Esempio 3.15

```
<input type="text" id="name" required  
  placeholder="Inserisci un nome"  
  oninvalid="this.setCustomValidity('Il nome va sempre  
  inserito')"  
  oninput="setCustomValidity('')"/> />
```

Il risultato è visibile nell'immagine 3.12.

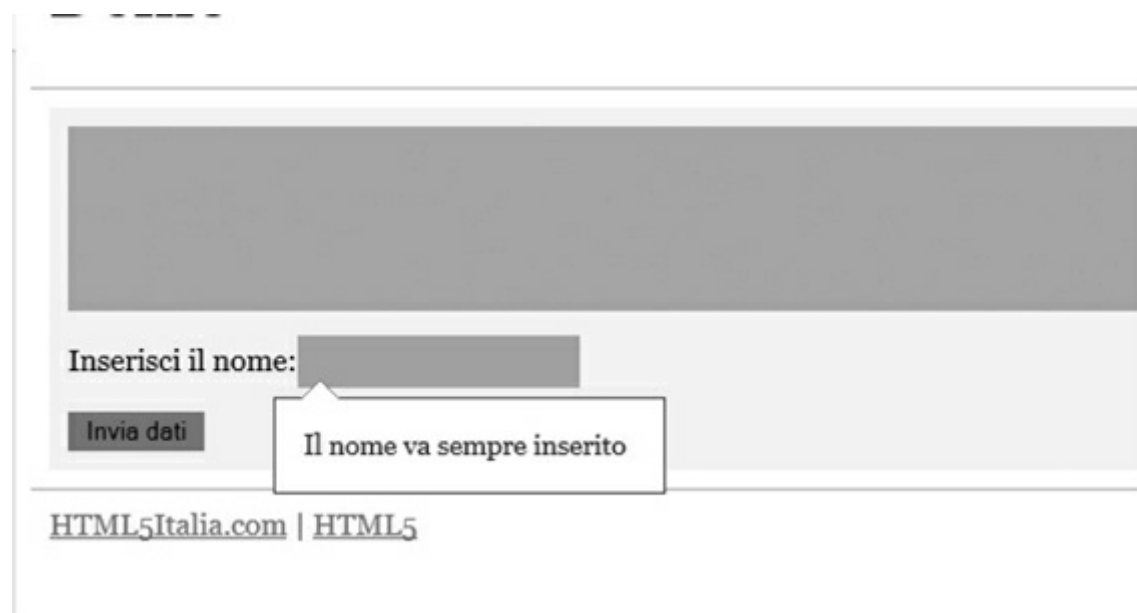


Figura 3.12 – Personalizzare il messaggio di errore è possibile ricorrendo a qualche istruzione JavaScript.

Bloccare la validazione

In alcuni casi può essere necessario bloccare la validazione. Per arrivare a questo scopo, è sufficiente inserire l'attributo `novalidate` all'interno della form.

È anche possibile associarlo a un tag `input`, per poter inibire in un particolare frangente la validazione, nella variante `formnovalidate`. Generalmente, questo attributo si aggiunge a un pulsante, quando si vuole evitare che quest'ultimo scateni la validazione (per esempio, perché il pulsante è associato a un'azione, come l'eliminazione di un elemento, che non

necessita dei dati inseriti nella form).

Le novità associate alla form

Fanno da contorno alle novità introdotte in precedenza, alcune specifiche migliorie che vanno a innestarsi direttamente nella relazione che esiste tra la form e i tag che contiene. In maniera specifica, attraverso il tag `input`, quando l'attributo `type` è impostato su `submit` o `image` (che ha la stessa funzione di `submit` ma consente di specificare un'immagine) possiamo ora impostare alcune funzionalità relative alla form.

Per esempio, possiamo cambiare l'URL di destinazione attraverso l'attributo `formaction`, il tipo di encoding con `formenctype`, il metodo utilizzato con `formmethod` e la finestra a cui inviare il risultato con `formtarget`. Si tratta di migliorie che rendono più semplice l'utilizzo di una sola form e l'assegnazione a un pulsante di un comportamento leggermente diverso, come accade nel caso dell'eliminazione dell'elemento, che è anche visualizzato nella form e, con l'uso di un altro pulsante, include anche la funzionalità di modifica.

Conclusioni

Le form hanno giocato, sin dalle primissime versioni di HTML, un ruolo importantissimo, perché consentono di far interagire il client (quindi l'utente) con il server. Nel corso delle versioni sono state introdotte diverse migliorie, ma con HTML5 abbiamo a disposizione una serie di nuove, interessanti novità, che possono dare una marcia in più alle nostre form. Come abbiamo visto nel capitolo, purtroppo il supporto in questo particolare ambito è ancora acerbo, con alcuni browser a buon punto e altri che invece hanno deciso di attendere fino a quando le specifiche saranno più mature.

Ad ogni modo, in questo capitolo abbiamo imparato come strutturare una form e come sfruttarne le caratteristiche.

Chiuso il discorso sul markup, passiamo ora al prossimo capitolo, nel quale inizieremo ad affrontare i CSS, un linguaggio già menzionato in questo capitolo, che consente di dare un aspetto grafico alle nostre pagine. Partendo dalle basi, arriveremo poi ad analizzare le novità della versione 3.

Introduzione ai CSS

Nei capitoli precedenti abbiamo visto come HTML5 porti a un livello ancora superiore il concetto di separazione tra struttura, che ha un significato e un utilizzo semantico, e sua rappresentazione.

In HTML i tag sono indicati, infatti, per dare un significato strutturale alla pagina, così che diventi possibile anche farne uso in assenza di un browser visuale: non dobbiamo dimenticare che l'accesso ai nostri documenti può essere fatto anche da browser non convenzionali, come quelli di motori di ricerca o browser vocali.

In tutto questo, all'interno di HTML5 trova spazio anche una serie di specifiche che prende il nome di **CSS3**. Siamo alla terza versione dei Cascade Style Sheet, una serie di specifiche che consentono di impaginare il markup, così da dargli una rappresentazione migliore. Questa nuova versione include molte migliorie, su cui ci soffermeremo all'interno di questo e dei prossimi due capitoli, non prima di aver iniziato a dare un'occhiata a come sfruttare al meglio i CSS, fin dalle basi.

Per dovere di cronaca, le specifiche di CSS3 sono suddivise in moduli e non tutte hanno lo stesso grado di maturità. Per questo motivo, alcune delle funzionalità presentate in questo e nei prossimi capitoli dedicati a CSS potrebbero non avere lo stesso grado di maturità e supporto all'interno dei browser: come sempre, cercheremo di porre l'attenzione su quelle specifiche che hanno già un certo grado di maturazione e il cui utilizzo valga la pena analizzare.

Le trovate tutte ricapitolate su <http://aspit.co/a0x>.

Iniziamo subito a partire alla scoperta delle basi di CSS.

Basi di CSS

Per convenzione, i file CSS vengono serviti all'interno di un file dotato di questa estensione (.css) e content type "text/css", referenziando il percorso

all'interno del documento, come [nell'esempio 4.1](#).

Esempio 4.1

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet"
          href="style.css"
          type="text/css"
          media="screen" />
  </head>
  <body>
    ...
  </body>
</html>
```

Dobbiamo notare che ciascun attributo riveste un ruolo fondamentale:

- **rel**: indica che il tag si riferisce a un foglio di stile;
- **href**: punta al file, con un percorso assoluto o relativo;
- **type**: indica il content type del documento, che, come detto, è `text/css`. In HTML5 è comunque considerato superfluo, ma si tiene per compatibilità;
- **media**: può assumere diversi valori, che indicano cosa serve il CSS; in questo caso, è specifico per la visualizzazione su schermo.

L'attributo `media` riveste un ruolo molto importante, perché consente di specificare CSS specifici per i vari aspetti. I valori al suo interno possono essere anche separati da una virgola, qualora con un dato CSS volessimo essere in grado di gestire più tipologie di browser. Anche se è possibile specificare diverse tipologie di valori per l'attributo `media`, all'atto pratico gli unici valori utilizzati sono `all`, `screen` (browser con schermo) e `print` (per la stampa).

Gestire la registrazione dei CSS

CSS3 introduce delle specifiche dedicate in maniera specifica a quest'area, per cui possiamo combinare delle istruzioni particolari in grado di abbinare un CSS a scenari avanzati, per esempio, in base alla risoluzione del browser o

alla disponibilità di una particolare feature. Le specifiche **CSS3 Media Queries** sono disponibili su <http://aspit.co/a0r> e saranno approfondite nel corso del prossimo capitolo.

Possiamo quindi specificare un CSS a uso e consumo di un browser il cui schermo abbia una risoluzione minore di 500px utilizzando [l'esempio 4.2](#).

Esempio 4.2

```
<link rel="stylesheet"
      href="style.css"
      type="text/css"
      media="screen and (min-device-width: 500px)" />
```

Ci sono diversi tipi di operatori e sorgenti che possono essere interrogati, che regolano l'aspect-ratio (4:3, 16/9), la risoluzione, il colore. Tutti questi valori sono approfonditi in maniera specifica nel [capitolo 5](#).

Tra l'altro, è possibile specificare gli stili anche direttamente nella pagina, benché questo approccio presenti lo svantaggio che ogni pagina invia ogni volta lo stesso contenuto, anziché fare riferimento a un file esterno condiviso, come fatto finora. Tuttavia, in alcuni contesti può essere comodo specificare degli stili localmente alla pagina ed è sufficiente che utilizziamo il codice [dell'esempio 4.3](#).

Esempio 4.3

```
<style>
  @screen {...}
  @print {...}
</style>
```

Anche in questo caso è possibile specificare degli stili ad hoc per i vari tipi di browser, ma si sfrutta una sintassi leggermente diversa. Il valore dell'attributo `media` del tag `link`, infatti, ora viene inserito direttamente all'interno del documento, antepoendo lo stesso valore con il simbolo `@`.

Resta inoltre possibile registrare un CSS esterno all'interno di un altro CSS, sia in una pagina sia, a propria volta, in un file esterno, usando la sintassi [dell'esempio 4.4](#).

Esempio 4.4

```
@import url("external.css") screen;
```

La keyword `screen` inserita in fondo è opzionale e, se omessa, indica che il CSS è valido per tutti i tipi di browser. Tra l'altro, lo sottolineiamo, questo è il comportamento predefinito. Dobbiamo notare che, inoltre, l'URL è specificato tramite un apposito attributo `url`, specificato dopo la keyword `@import`. Il percorso del file, come per tutte le risorse di un documento, può essere tanto relativo, quanto assoluto.

nota

Tra le tante specifiche che vengono raccolte dal termine CSS3, ne esiste una denominata **CSS Namespaces**. Si tratta di un modulo che specifica la sintassi da utilizzare per inserire namespace (cioè un modo logico per raggruppare delle regole) all'interno di un documento CSS. Viene definita una keyword `@namespace` e una sintassi per far riferimento al percorso completo degli elementi con namespace.

Maggiori informazioni sono disponibili su:
<http://aspit.co/a0y>.

Ora che abbiamo visto come registrare un file con i nostri stili, iniziamo a dare un'occhiata a come crearli all'interno della nostra applicazione.

Applicare gli stili a un documento

Anche se può sembrare complesso per chi è a digiuno di sviluppo web, in realtà il concetto di separazione netta tra markup e formattazione grafica è l'aspetto più potente dell'accoppiata HTML e CSS. Iniziamo a dare un'occhiata a come si può comporre il proprio foglio di stile, analizzandone le differenti tipologie di sintassi.

Stile per i tag

Applicare uno stile a un tag è abbastanza semplice. Dopo aver registrato il CSS all'interno della pagina, utilizzando uno dei metodi descritti in precedenza, dobbiamo procedere con la definizione della regola di stile.

Nel caso dei tag, ci basta specificare lo stile all'interno del CSS, come nell'esempio 4.5.

Esempio 4.5

```
body {  
    font-family: arial;  
    font-size: 15px;  
}  
p, div {  
    color: red;  
    background: white;  
}  
* {  
    color:green;  
}
```

Nell'esempio precedente, impostiamo il colore del testo di tutti i tag `p` e `div` sul rosso. Si possono specificare più tag a cui applicare un determinato stile semplicemente separandoli con una virgola. Un caso a parte merita il carattere `*`, che applica lo stile specificato a tutti i tag.

Possiamo specificare i colori nel formato con nome, come abbiamo fatto nel nostro esempio (generalmente si tratta del rispettivo colore in inglese e in minuscolo), oppure usando la forma `#RRGGBB`. In questo caso, ciascuna coppia equivale a un valore esadecimale che indica, rispettivamente, il valore di rosso, verde e blu: per esempio, un grigio avrà valore `#cccccc`, il rosso `#ff0000` e così via.

Benché comporre i colori usando questa sintassi sia comune, non avrete necessità di calcolare il rispettivo valore ogni volta manualmente, perché la maggior parte dei tool di manipolazione delle immagini è in grado di darvi il valore `#RRGGBB` a partire da un qualsiasi colore scelto in modo visuale. Nella [figura 4.1](#) viene mostrato un esempio del nostro CSS applicato alla pagina.

Demo

CSS

Questo è un contenuto normale all'interno di un paragrafo.

Questo è un all'interno di un div.

Figura 4.1 – Ecco come appare all'interno di un browser un semplice documento HTML a cui viene applicato il nostro CSS.

Per quanto riguarda gli altri stili utilizzati, che impostano dimensione e tipologia di font, vi rimandiamo al [capitolo 8](#), dove saranno approfonditi in maniera specifica, insieme alla gestione dei colori, che abbiamo appena accennato.

Stile con classe

Applicare uno stile sui tag è generalmente sconsigliato, anche se in molti casi può essere utile per specificare comportamenti comuni, per esempio font, margini o colori. Resta comunque facilmente implementabile, perché basta ripetere il tag all'interno del foglio di stile. Occorre soltanto prestare attenzione che le definizioni all'interno del foglio di stile siano case-sensitive.

Molto più diffusa, invece, è la notazione che fa uso della cosiddetta classe di stile, cioè di un valore ad hoc specificato attraverso l'attributo `class`, che ogni tag supporta. In questo caso c'è maggiore flessibilità, perché un certo comportamento può essere applicato a tipologie di tag differenti, in un solo colpo. In questo caso, il valore specificato nell'attributo `class` all'interno del tag deve avere un carattere `.` (punto) come prefisso. [Nell'esempio 4.6](#), viene mostrato come.

Esempio 4.6 – HTML

```
<p class="warning">Paragrafo in primo piano</p>
<p>Paragrafo normale</p>
<div class="warning small">Div in primo piano</div>
```

Esempio 4.6 – CSS

```
.warning {
  color: red;
  background: white;
}
.small {
  font-size: 70%;
}
```

La versatilità di questo approccio è notevole, poiché consente di applicare, come abbiamo fatto nell'esempio, lo stesso stile a tag differenti, risparmiandoci ogni volta l'onere di specificare per ciascun tag una regola. Separando gli stili con uno spazio all'interno dell'attributo `class`, poi, è possibile specificare più stili da applicare a un elemento, evitando duplicazione di stili.

Il vero vantaggio derivante dall'uso degli stili con classe, comunque, è quello di poter definire agevolmente una regola da applicare in casi speciali e, al tempo stesso, lasciare che altri tag si comportino in modo differente. La [figura 4.2](#) mostra l'esempio all'opera.

Demo

CSS

Paragrafo in primo piano

Paragrafo normale

Div in primo piano

[HTML5Italia.com](#) | [HTML5](#)

Figura 4.2 – Applicando gli stili con classe, diventa più facile riutilizzare in diversi contesti lo stesso stile.

Possiamo specificare più stili semplicemente separandoli con una virgola, oppure mischiare definizione di tag e stili con classe. [Nell'esempio 4.7](#) viene mostrato come.

Esempio 4.7

```
strong, .bold {  
    ...  
}
```

In questo caso, la regola sarà applicata a tutti i tag `strong` e a tutti quelli che hanno come `class` il valore “bold”.

È anche possibile specificare dei criteri di selezione più complessi, che

sono generalmente indicati con il termine **selettori** e che verranno specificati tra qualche paragrafo in questo stesso capitolo.

Stile con ID

Un ulteriore modo per specificare gli stili è quello di fare riferimento alla proprietà `id` che ciascun tag presenta. Attraverso questa proprietà possiamo definire un determinato tag in maniera univoca all'interno del documento. Generalmente questa proprietà è utilizzata con JavaScript, ma resta possibile farlo anche nei CSS. [L'esempio 4.8](#) ci introduce alla sintassi da utilizzare.

Esempio 4.8 – HTML

```
<div id="header">
  <p>Questa è l'intestazione</p>
</div>
```

Esempio 4.8 – CSS

```
#header {
  color: black;
  background: red;
}
```

In questo caso, il valore va fatto precedere dal carattere `#`, mentre restano valide tutte le altre considerazioni già fatte negli altri casi.

Questo approccio è, in genere, utilizzato in presenza degli elementi strutturali della pagina. Molto spesso, infatti, si tende a dare un ID a questi elementi, che rappresentano la struttura portante della pagina, così da poterli manipolare più facilmente. Non sarebbe sbagliato, comunque, dare direttamente una classe a tali elementi, perché funzionerebbero allo stesso modo. Tuttavia, specificare gli ID consente di fare in modo che il markup e il relativo CSS siano più ordinati, grazie al fatto che si ha la certezza che quello stile sarà applicato solo al tag a cui si riferisce. Nel corso del prossimo capitolo, impareremo l'importanza di questo aspetto, quando parleremo di come comporre i layout utilizzando i CSS.

I selettori CSS

Comunemente, insieme ad altre varianti che introdurremo tra qualche istante,

le sintassi analizzate finora vanno a formare i cosiddetti **selettori**. Si tratta di una grammatica particolare che viene utilizzata all'interno dei file CSS per definire regole, che possono essere complesse. Include la possibilità di definire delle regole di stile che si applichino a stati particolari degli elementi, piuttosto che alla gerarchia degli stessi all'interno del markup. I selettori rivestono un ruolo importante, per cui buona parte di questo capitolo è dedicato a questa caratteristica. Dove si tratti di una novità di CSS3, questa viene evidenziata, così da agevolarvi qualora conosciate già le caratteristiche delle versioni precedenti.

Le pseudo-classi

Le specifiche relative ai CSS prevedono che si possano specificare delle regole ad hoc da applicare a uno stile quando questo si trova in un particolare stato. Il caso più diffuso è quello rappresentato dai link, che sono inseriti usando il tag `a`, in cui il link può presentare diversi stati, legati all'interazione dell'utente con la pagina:

- `link`: quando il link viene visualizzato nella pagina;
- `hover`: quando l'utente passa con il mouse sopra il link;
- `active`: quando il link è quello attivo;
- `visited`: quando il link è già stato visitato.

Per consentire allo sviluppatore di personalizzare l'aspetto di tutta la pagina, le specifiche CSS consentono di definire degli stili per questi stati, che prendono il nome di pseudoclassi. L'origine del nome è da ricercare nel fatto che non si tratta di vere e proprie classi, quanto di classi legate allo stato degli elementi, che il browser crea in maniera trasparente e non sono ricavate dal markup.

In maniera particolare, queste regole prendono il nome di pseudo-classi e si definiscono come [nell'esempio 4.9](#).

Esempio 4.9 – HTML

```
<a href="http://www.html5italia.com/">HTML5Italia.com</a>
```

Esempio 4.9 - CSS

```
a {
```

```
color: black;
}
a:hover {
color: red;
}
a:visited{
color:gray;
}
a:active{
color:green;
}
```

In questo caso abbiamo definito uno stile specifico per ognuno degli stati, andando a cambiarne il colore. Proviamo questo esempio in un browser, interagendo con la pagina, e noteremo come, effettivamente, siamo stati in grado di cambiare lo stile applicato al tag utilizzando esclusivamente i CSS.

Le pseudo-classi si possono applicare anche in presenza di classi, oltre che dei tag, con la sintassi `tag.classe:pseudoclasse` illustrata [nell'esempio 4.10](#).

Esempio 4.10

```
a.redLink:hover {
color: red;
}
```

In questo caso la regola sarà applicata solo ed esclusivamente ai tag `a` che abbiano l'attributo `class` impostato su “redLink”.

Oltre alle già citate pseudo classi, ce ne sono alcune specifiche, che sono tutte ricapitolate nella [tabella 4.1](#).

<i>Pseudo-classe</i>	<i>Descrizione</i>
<code>:active</code>	Consente di gestire lo stato dei link quando sono attivi.
<code>:hover</code>	Rappresenta il selettore che indica lo stile da applicare a un element quando ci si passa sopra con il mouse.
<code>:link</code>	Rappresenta lo stile applicato a un elemento di tipo link quando non è stato ancora visitato.
<code>:visited</code>	Da indicare per specificare uno stile da associare quando un element di tipo link è stato visitato.

:root	Quando viene applicato a un dato elemento, lo stile specificato è valido solo se questo è l'elemento principale (root) del documento.
:nth-child(n)	Indica l'ennesimo elemento figlio di un altro elemento.
:nth-last-child(n)	Indica l'ennesimo elemento figlio di un altro elemento, partendo dall'ultimo.
:nth-of-type(n)	Indica l'ennesimo elemento contenuto all'interno dell'albero di un altro elemento (<i>sibling</i>), del tipo specificato.
:nth-last-of-type(n)	Indica l'ennesimo elemento del <i>sibling</i> , del tipo specificato, partendo dall'ultimo.
:first-child	Indica il primo elemento figlio.
:last-child	Indica l'ultimo elemento figlio.
:first-of-type	Indica il primo elemento del <i>sibling</i> , del tipo specificato.
:last-of-type	Indica l'ultimo elemento del <i>sibling</i> , del tipo specificato.
:only-child	Indica un elemento esclusivamente figlio di quello specificato.
:only-of-type	Indica un elemento del <i>sibling</i> , del tipo specificato.
:empty	Consente di specificare uno stile applicato a un elemento vuoto (senza tag all'interno né testo).
:enabled	Consente di individuare un elemento che non sia stato disabilitato.
:disabled	Al contrario di :enabled, indica un elemento che è stato disabilitato.
:focus	Specifico per gestire il focus sugli elementi della form (input ecc.), consente di gestire lo stato relativo al focus su un elemento.
:checked	Consente di applicare uno stile particolare quando un elemento è selezionato. Si applica a <code>checkbox</code> e <code>radio</code> .
:lang(it)	Consente di specificare una formattazione da applicare, solo in

:lang(it) presenza di determinate lingue.

Tabella 4.1 – Le pseudo-classi disponibili in CSS3.

Iniziamo a elaborare un esempio, così da capire la potenza applicata ai selettori CSS, quando utilizzati in maniera estesa. Per esempio, possiamo utilizzare un semplice selettore per applicare un colore alternato alle righe di una tabella. Questa cosa non era possibile prima di CSS3 senza intervenire sul markup generato, applicando alternativamente un colore di sfondo. [L'esempio 4.11](#) contiene, invece, il codice necessario per raggiungere lo stesso scopo con CSS3.

Esempio 4.11 – HTML

```
<table>
  <tr><td>Riga #1</td></tr>
  <tr><td>Riga #2</td></tr>
  <tr><td>Riga #3</td></tr>
  <tr><td>Riga #4</td></tr>
</table>
```

Esempio 4.11 – CSS

```
tr:nth-child(odd) {
  background: #c0c0c0;
}
tr:nth-child(even) {
  background: #999999;
}
```

Quando lanciato nel browser, l'effetto di questo CSS sarà quello visibile nella [figura 4.3](#).

Demo



Figura 4.3 – Creare righe alternate con CSS3 diventa molto semplice.

All'interno di questi selettori, oltre a `odd` e `even`, si possono specificare anche numeri o formule, come `-n+3`, che indica le ultime 3 righe di un dato elemento.

Combinando insieme i selettori appena introdotti, diventa possibile dare maggiore funzionalità alle pagine, sfruttando in maniera più semplice la potenza che offrono i CSS ed evitando, al tempo stesso, di fare uso di JavaScript in queste circostanze, necessità che, prima di CSS3, era praticamente impossibile evitare.

Gli pseudo-elementi

Esiste una famiglia di selettori che prende il nome di pseudo-elementi, perché, un po' come le pseudo-classi, sono in realtà degli elementi creati dal browser per consentirci una formattazione più semplice. La [tabella 4.2](#) riepiloga i selettori: i primi due sono stati introdotti con CSS3, mentre gli ultimi due con CSS2.1.

<i>Pseudo-elemento</i>	<i>Descrizione</i>
<code>::first-line</code>	Applica uno stile alla prima riga di un elemento.
<code>::first-letter</code>	Applica uno stile alla prima lettera di un elemento.

<code>::before</code>	Consente di inserire del contenuto prima dell'elemento.
<code>::after</code>	Consente di inserire del contenuto dopo l'elemento.

Tabella 4.2 – Gli pseudo-elementi disponibili in CSS3.

Questi selettori consentono di far riferimento a un elemento che in realtà non esiste, ma è creato per noi dal browser. Rispetto ai selettori per le pseudo-classi, sono necessari due caratteri “:” davanti al nome del selettore stesso, così che possano differenziarsi facilmente. Analizziamo per un attimo il codice presente [nell'esempio 4.12](#).

Esempio 4.12 – HTML

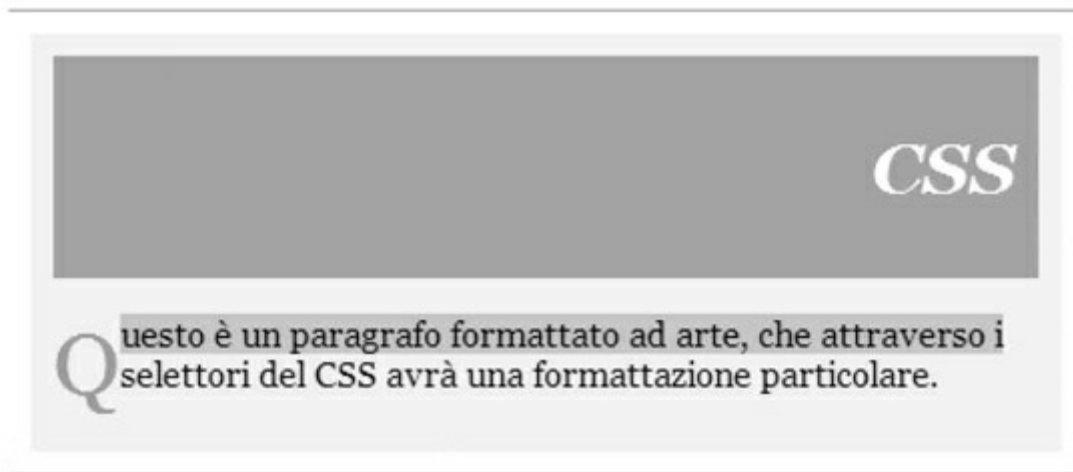
```
<p>Questo è un paragrafo formattato ad arte, che attraverso i  
selettori  
del CSS avrà una formattazione particolare.</p>
```

Esempio 4.12 – CSS

```
p::first-line {  
    background: #c0c0c0;  
}  
p::first-letter {  
    color: red;  
    font-size:250%;  
    float:left;  
}
```

Eseguendo questo esempio nel browser, potremo notare come le nostre regole vengano applicate, dando un'occhiata alla [figura 4.4](#).

Demo



HTML5Italia.com | [HTML5](#)

Figura 4.4 – Attraverso i selettori si possono creare effetti tipografici in maniera molto rapida.

Possiamo notare che la prima lettera è stata colorata di rosso e ha una dimensione del 250% più grande, mentre la prima linea ha sfondo grigio e il resto del paragrafo ha una formattazione normale. Questo effetto, spesso riprodotto sui libri stampati, prima di CSS3 era possibile solo applicando una buona dose di JavaScript, oppure con del markup scritto ad hoc, mentre ora è supportato in maniera nativa.

Il selettore `:not`

Un discorso a parte lo merita il selettore `:not`, introdotto con CSS3. La sua unica funzione, infatti, è quella di negare l'espressione specificata. [L'esempio 4.13](#) mostra come sfruttare la pseudo-classe di negazione.

Esempio 4.13 – HTML

```
<p class="redP">Paragrafo rosso.</p>
<p>Altro paragrafo</p>
```

Esempio 4.13 – CSS

```
p:not(.redP) {
    color: green;
}
```

```
p.redP {  
    color:red;  
}
```

Occorre prestare attenzione al fatto che le negazioni non possono essere innestate. Questo pseudo-elemento ritorna molto comodo quando si vuole applicare una determinata regola all'interno del CSS, negando una regola, così da poter applicare in maniera più specifica la regola stessa a un determinato insieme di elementi.

I selettori di attributi

Un altro caso interessante è rappresentato dai selettori che si applicano ad attributi presenti nel markup. Nel [capitolo 3](#) abbiamo visto come vengano introdotti alcuni elementi aggiuntivi e come, per questo motivo, si possano formattare più agevolmente attraverso CSS.

In particolare, prima di CSS3 non esisteva un modo semplice per distinguere tra elementi di una form, per cui molto spesso, per evitare di applicare uno stile alle checkbox, era necessario specificare manualmente uno stile. Grazie ai selettori di CSS3, invece, diventa possibile applicare semplicemente una formattazione specifica, come riportato [nell'esempio 4.14](#).

Esempio 4.14

```
input, textarea, select {  
    color:black;  
    background:red;  
}  
input[type="button"] {  
    color:red;  
    background:black;  
}  
input[type="checkbox"], input[type="radio"] {  
    color: black;  
    background: transparent;  
}  
input[disabled], input[readonly] {  
    background: gray;  
}
```

Tutti questi selettori vengono applicati ai tag `input`, agendo sul valore della

proprietà `type`, oppure, se presenti, di quelle `readonly` o `disabled`. Con un apposito markup che ne faccia uso, il risultato è come quello visibile nella [figura 4.5](#). L'uso dei selettori sugli attributi può consentire di personalizzare in maniera molto interessante un numero elevato di controlli che, a seconda dei valori degli attributi, mostrano comportamenti diversi, come quelli delle form.

Demo



Figura 4.5 – Una form con i selettori CSS all’opera.

Questi selettori possono essere applicati a qualsiasi tipo di tag e sono molto potenti, perché consentono di identificare facilmente un elemento, associando uno stile al cambio del relativo stato, evitando, ancora una volta, di fare uso di JavaScript per applicare una formattazione.

I selettori combinati

La vera potenza dei selettori risiede nella possibilità di combinarli tra di loro, così da creare complesse catene. Se avete dimestichezza con **jQuery**, sappiate che quest’ultimo utilizza i selettori di CSS, quindi troverete molte affinità tra quanto avete eventualmente già fatto e quello che andremo a introdurre in questa parte del capitolo. Molti dei concetti qui espressi non sono novità assolute di CSS3, ma è con CSS3, e i browser che ne supportano le specifiche, che si ha una compatibilità più diffusa di queste specifiche, tale da poter parlare tranquillamente di queste caratteristiche.

I selettori di gerarchia

Iniziamo a dare uno sguardo al più semplice dei selettori, quello che consente di identificare un tag utilizzando una gerarchia. Diamo un'occhiata al codice dell'esempio 4.15.

Esempio 4.15 – HTML

```
<p>Esempio di un paragrafo
  <span>con dentro
    <strong>un altro tag</strong></span></p>
<div>
  <section>
    <p>Contenuto</p>
  </section>
</div>
```

Esempio 4.15 – CSS

```
p strong {
  color: red;
}
div * p {
  background: yellow;
}
```

Il primo selettore ha l'effetto di recuperare tutti i tag `strong` all'interno del tag `p`, anche annidati eventualmente in altri tag. Attraverso la seconda sintassi, invece, indichiamo che vogliamo recuperare un tag `p` che sia contenuto all'interno di un altro (o più) tag posto all'interno del tag `div`. La combinazione di questi elementi consente di identificare in maniera precisa dei tag contenuti all'interno di altri tag e, in combinazione con i selettori di pseudo-classi e pseudo-elementi, consente di specificare con una precisione molto elevata come formattare un dato elemento, senza necessità di intervenire all'interno del markup. Questa caratteristica è molto importante quando il contenuto è generato in automatico, per esempio, perché siamo in presenza di un contenuto inserito con un CMS (Content Management System, sistema di gestione dei contenuti).

Un altro caso particolare è quello dato dai selettori all'interno dell'esempio 4.16.

Esempio 4.16 – HTML

```
<p>Questo è un contenuto <strong>importante</strong>.</p>
```

```
<div>
  <ul>
    <li>
      <div>Contenuto generico</div>
      <p>Contenuto dentro un paragrafo</p>
    </li>
  </ul>
</div>
```

Esempio 4.16 – CSS

```
p > strong {
  color: green;
}
div ul>li p {
  background: aqua;
}
```

In questo caso, il primo selettore indica che vogliamo recuperare i tag `strong` posti direttamente al di sotto di un tag `p`. A differenza del primo selettore [dell'esempio 4.15](#), quest'ultimo non funzionerebbe in presenza di un tag innestato.

Il secondo settore [dell'esempio 4.16](#) indica una gerarchia ancora più precisa: solo gli elementi `p` posti all'interno di un tag `li`, inserito esclusivamente sotto un tag `ul`, contenuto in un tag `div`, avranno lo stile CSS applicato. È da notare, insomma, che è il carattere `>` che vincola la presenza del tag direttamente a quello che contiene.

Come possiamo notare, lo spazio prima e dopo il carattere `>` è opzionale e, in genere, è mantenuto per una questione di leggibilità.

Una variante di questi selettori, che consente di gestire altri casi, è visibile [nell'esempio 4.17](#).

Esempio 4.17 – HTML

```
<label>
  <input type="checkbox" checked="true" /> Valore 1
  <span class="error">Errore!</span>
</label>
```

```
<p>Paragrafo</p>
<div>Div</p>
<pre>Preformattato</p>
```

Esempio 4.17 – CSS

```
input[type="checked"] + .error {  
    font-weight:bold;  
}  
p ~ pre {  
    background: yellow;  
}
```

Il primo selettore recupera un tag con class `error` posto direttamente dopo un tag `input` di tipo `checkbox`, mentre il secondo agisce su un tag `pre` posto dopo un tag `p`, ma non immediatamente.

I selettori di CSS sono molto complessi e possiamo utilizzare una grammatica ad hoc per gestire le regole. Maggiori informazioni sulle specifiche sono disponibili su <http://aspit.co/a0z>.

Gestire il posizionamento

Le specifiche CSS consentono di indicare un posizionamento relativo o assoluto dell'elemento rispetto al contenitore, oltre alla possibilità di far “galleggiare” gli elementi all'interno dello spazio.

Alcuni elementi, come `img`, `span` o `a`, hanno la caratteristica di essere posti in linea rispetto ai tag che li circondano, così che sia possibile mettere sulla stessa riga un'immagine e un testo, per esempio. Molti altri tag seguono questo stesso approccio, ma ce ne sono alcuni, detti contenitori, che invece delimitano lo spazio: è il caso di `p` o `div`, per esempio. In questo caso, si parla di elementi **block**, mentre, nel caso precedente, di elementi **inline**. Queste informazioni saranno approfondite in maniera specifica nel prossimo capitolo, dedicato alla gestione del layout, ma erano comunque utili da introdurre in questo contesto.

Iniziamo dunque ad analizzare come poter gestire un semplice ancoraggio degli elementi all'interno dello spazio.

Gestire la disposizione degli elementi

Finchè si tratta di mettere in verticale una serie di elementi di tipo `block`, i problemi probabilmente saranno minimi, perché è il comportamento predefinito di questi tag. Vi basta inserire una serie di paragrafi per esser certi che questi vengano messi uno dopo l'altro. Cosa succede se, invece, volessimo iniziare a comporre un layout disponendo gli elementi come nella

figura 4.6.

In questo caso, in passato si è fatto uso del tag `table`, che consente di creare una tabella e quindi si presta bene al compito che abbiamo appena deciso di affrontare.

In realtà, dal punto di vista semantico questa scelta non è il massimo, perché le tabelle nascono per rappresentare dati da mostrare a video, non per indicare la struttura di un documento.

In questi casi, dunque, bisogna utilizzare un comando particolare, da inserire nel CSS, attraverso la parola chiave `float`. Sfruttando questa chiave, diventa possibile rendere gli elementi block in grado di “fluttuare” nello spazio, adeguandosi alla dimensione degli altri blocchi.

Questa caratteristica è essenziale per comporre layout sfruttando HTML e CSS, di cui una prima versione è riportata [nell'esempio 4.18](#).

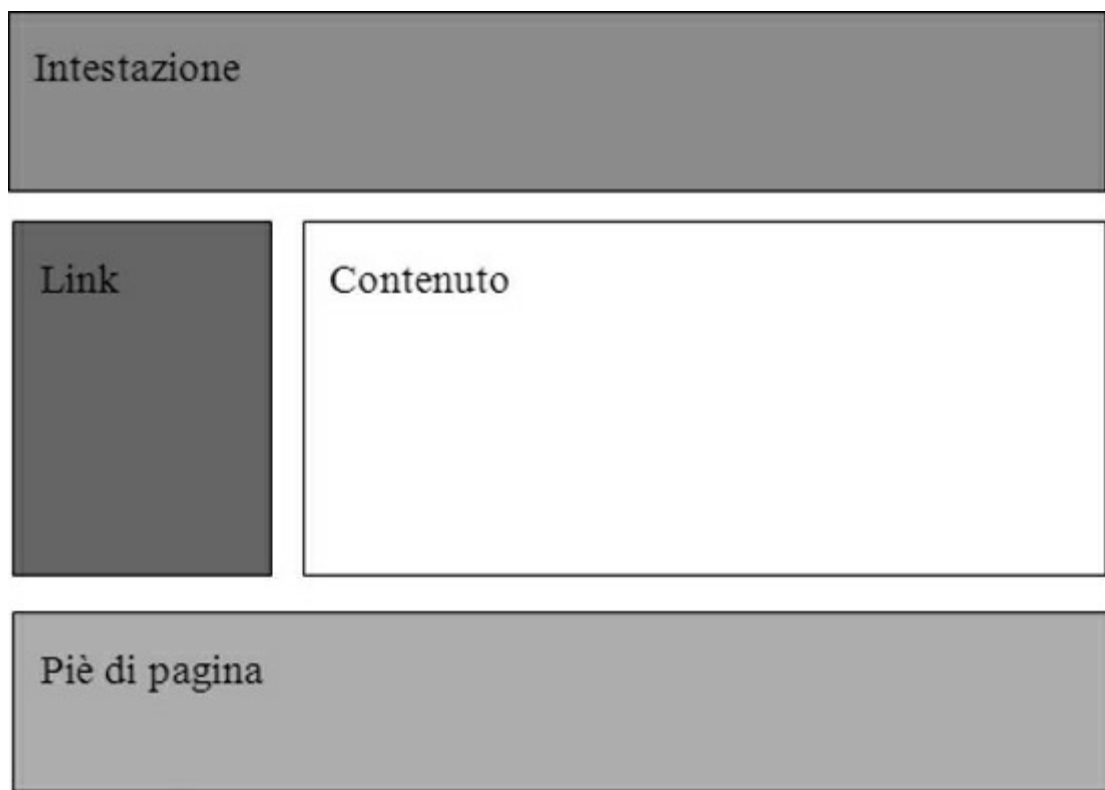


Figura 4.6 – Il layout che vogliamo riprodurre con i CSS.

Esempio 4.18 – HTML

```
<!DOCTYPE html><html>
...
<body>
  <header id="header">Header</header>

  <section id="content">
```



```

    <section id="navbar">Barra di navigazione</section>
    <div id="innerContent">Contenuto del sito</section>
</section>

<footer id="footer">Header</footer>
</body>
</html>

```

Esempio 4.18 – CSS

```

#header, #footer, #content {
    clear:both;
    background: #c0c0c0;
}
#navbar {
    float:left;
    width:30%;
    background: #f0f0f0;
}
#innerContent{
    float:right;
    width:70%;
    background: #d0d0d0;
}

```

Il markup che abbiamo utilizzato non funzionerebbe benissimo in presenza di browser che non abbiano il supporto a HTML5, quindi, in genere, si preferisce sfruttare questi nuovi tag insieme ad altrettanti tag `div`, per mantenere la compatibilità.

Nell'esempio precedente possiamo notare l'uso di `float` e `clear`, che indicano, rispettivamente, che l'elemento deve fare spazio ad altri, oppure deve ripulire tutto lo spazio che gli sta attorno. Comporre layout utilizzando queste proprietà è molto semplice e consente di ottenere risultati molto complessi, quando si combinano in maniera opportuna blocchi di questo tipo. Il risultato del layout è visibile nella [figura 4.7](#).

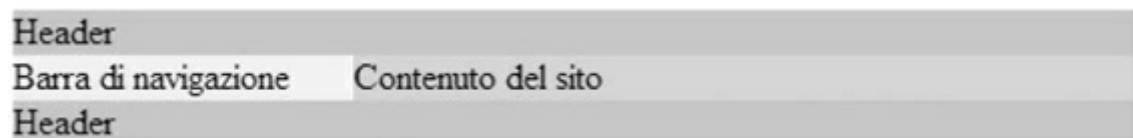


Figura 4.7 – Il CSS applicato al nostro markup produce una struttura simile a quella che avevamo stilizzato nell'immagine precedente.

È importante notare che la dimensione data agli elementi può influenzare, se omessa, il modo in cui gli elementi stessi vengono renderizzati dal browser. Nel corso del prossimo capitolo, verrà introdotto, inoltre, il concetto di box model, che è essenziale per capire come funzionino l'allocazione dello spazio, dei margini e del bordo da parte di CSS.

Gestire il posizionamento assoluto

Quanto abbiamo visto finora rientra nell'ambito del posizionamento relativo degli elementi. A parte casi particolari, il posizionamento assoluto resta il migliore degli approcci, perché in HTML è anche importante, ai fini di browser non convenzionali (motori di ricerca, browser vocali) che il contenuto sia posizionato nella pagina in base all'importanza dello stesso.

Ad ogni modo, in alcuni scenari, soprattutto quando ci sono da comporre layout molto complessi (oppure con elementi animati), la possibilità di posizionare in maniera assoluta gli elementi sulla superficie ritorna molto comoda. In particolare, è necessario prestare attenzione al fatto che, in questi casi, gli elementi potrebbero trovarsi sovrapposti e, quindi, nascondersi a vicenda. Diamo subito un'occhiata al codice [dell'esempio 4.19](#).

Esempio 4.19

```
.edit {  
  position: absolute;  
  z-index: 99;  
  top: 1px;  
  left: 1px;  
  background: red;  
}
```

Dobbiamo prestare attenzione alle prime quattro istruzioni:

- `position`: di default assume come valore `relative`, ma con `absolute` siamo in grado di dare all'elemento la possibilità di essere posizionato in maniera assoluta;
- `z-index`: gestisce l'indice dell'elemento sull'ipotetico piano `z`, aiutandoci a gestire elementi in primo piano e in secondo piano;
- `top`, `bottom`, `left` e `right`: ci consentono di specificare l'allineamento dell'elemento all'interno della pagina.

In genere, queste tecniche sono utilizzate per fare in modo che un dato elemento si sovrapponga a un altro, per esempio per creare una barra con dei link per modificare un certo contenuto, come nell'esempio che abbiamo appena creato, visibile nella [figura 4.8](#).

[Modifica questo contenuto]

Demo



Figura 4.8 – Come si può notare, il contenuto è stato posizionato in alto, in maniera assoluta rispetto agli altri.

Il posizionamento assoluto è particolarmente utile, anche se il suo uso è limitato a scenari ben particolari.

Con l'analisi di quest'ultima caratteristica, abbiamo esaminato un po' tutte le caratteristiche di base legate ai CSS.

Conclusioni

I CSS rappresentano il sistema attraverso il quale diamo una rappresentazione grafica al markup che scriviamo attraverso l'HTML.

Da un punto di vista pratico, i CSS sono uno degli elementi più complessi da padroneggiare, perché hanno molte caratteristiche e, soprattutto, da che sono stati creati, sono stati implementati in maniera molto diversa dai vari browser.

HTML5 e CSS3, cercando di porre fine a queste differenze, introducono una serie di nuove funzionalità specifiche e una serie di test condivisi tra i vari browser, che dovrebbero fare in modo che tutti implementino delle specifiche nello stesso modo.

In questo capitolo ci siamo soffermati sulle caratteristiche di base, introducendo i concetti fondamentali. Siamo poi passati al linguaggio

attraverso il quale costruire i fogli di stile, analizzando la semantica legata ai selettori. I selettori rivestono un ruolo fondamentale e avremo modo di trovarli anche nei prossimi capitoli, applicati anche insieme a JavaScript. Infine, abbiamo iniziato a dare un'occhiata a come comporre il layout, argomento che approfondiremo proprio nel prossimo capitolo, insieme ad altre tecniche necessarie per comporre al meglio il foglio di stile nelle nostre pagine.

Layout ed effetti con CSS3

Ora che abbiamo appreso le basi di CSS, possiamo fare un passo avanti e scoprire le funzionalità avanzate che i CSS ci mettono a disposizione.

Nella prima parte di questo capitolo ci soffermeremo sul box model, cioè la tecnica con cui i nostri elementi vengono dimensionati in base ai margini, il padding e i bordi. Successivamente vedremo le varie modalità di posizionamento degli elementi HTML, in modo che, alla fine della lettura di questa parte del volume, saremo in grado di scrivere un CSS che posiziona gli elementi sulla pagina in maniera efficace.

Nella seconda parte del capitolo vedremo come utilizzare le media query per selezionare i CSS da utilizzare a seconda delle caratteristiche del device che visualizza la pagina.

Infine, vedremo come trasformare gli elementi HTML e come creare animazioni. Grazie a queste tecniche possiamo creare effetti grafici gradevoli e accattivanti per l'utente finale, senza la necessità di ricorrere a librerie JavaScript.

Alla fine di questo capitolo saremo in grado di creare un sito con effetti grafici che migliorano l'interattività con l'utente.

Il box model e la gestione di margini, bordi e spazio

Ogni elemento HTML che creiamo nelle nostre pagine (un `div`, uno `span` ecc.) ha una dimensione. La dimensione è calcolata sommando tra loro i diversi parametri CSS che influenzano il dimensionamento dell'elemento stesso. Questi parametri sono:

- `border`: specifica il bordo di un elemento;
- `padding`: specifica lo spazio tra il contenuto di un elemento e il bordo dell'elemento stesso;

- **margin**: specifica lo spazio tra il bordo di un elemento e gli elementi circostanti.

Entriamo ora nel dettaglio di questi parametri.

Gestire il bordo

Come detto nella precedente sezione, il parametro CSS `border` rappresenta il bordo di un elemento. Del bordo possiamo impostare la dimensione, lo stile, il colore e l'arrotondamento degli angoli.

Nella [tabella 5.1](#) sono riassunte le principali proprietà CSS che gestiscono lo stile di un bordo.

Proprietà	Descrizione
<code>border-width</code>	Imposta la dimensione del bordo su tutti e quattro i lati dell'elemento. Possiamo assegnare dimensioni differenti per ogni lato sfruttando le proprietà <code>border-top-width</code> , <code>border-left-width</code> , <code>border-bottom-width</code> , <code>border-right-width</code> .
<code>border-style</code>	Imposta lo stile del bordo. Una trattazione completa di tutti gli stili possibili esula dagli scopi di questo libro, per cui riportiamo qui solo gli stili più usati che sono: <code>none</code> (nessun bordo), <code>solid</code> (linea continua), <code>dotted</code> (linea punteggiata), <code>dashed</code> (linea tratteggiata). Anche in questo caso, possiamo impostare lo stile del bordo per ogni lato dell'elemento, usando le proprietà <code>border-top-style</code> , <code>border-left-style</code> , <code>border-bottom-style</code> , <code>border-right-style</code> .
<code>border-color</code>	Imposta il colore del bordo. Anche in questo caso, possiamo impostare il colore del bordo per ogni lato dell'elemento, usando le proprietà <code>border-top-color</code> , <code>border-left-color</code> , <code>border-bottom-color</code> , <code>border-right-color</code> .
<code>border-radius</code>	Imposta la curvatura per creare gli angoli arrotondati. Possiamo impostare valori diversi per ogni angolo del bordo, usando <code>border-top-left-radius</code> , <code>border-top-right-radius</code> , <code>border-bottom-left-radius</code> , <code>border-bottom-right-radius</code> . Parleremo in maniera più dettagliata di questa caratteristica più avanti nel libro.

Tabella 5.1 – Elenco delle proprietà CSS per gestire il bordo.

[Nell'esempio 5.1](#) vediamo uno stile CSS che definisce il bordo di un elemento.

Esempio 5.1

```
div
{
    border-width: 1px;
    border-style: solid;
    border-color: #000;
}
```

Come si evince da questo esempio, le proprietà CSS che impostano il bordo sono estremamente semplici. In alternativa alle singole proprietà CSS, possiamo utilizzare la proprietà CSS `border` che ci permette di specificare rispettivamente dimensioni, stile e colore del bordo in una sola riga, così come mostrato nel seguente codice.

Esempio 5.2

```
div
{
    border: 1px solid #000;
}
```

In questo esempio, abbiamo impostato lo spessore del bordo a un pixel, lo stile a solido e il colore su nero. Se confrontiamo questo codice con quello [dell'esempio 5.1](#), ci rendiamo conto che il codice da scrivere è minore e quindi più semplice da mantenere.

Ora che abbiamo visto come utilizzare i bordi, passiamo alla prossima sezione, che si occupa del padding.

Gestire il padding

Come detto in precedenza, il parametro CSS `padding` rappresenta lo spazio tra il bordo di un elemento e il suo contenuto. Tutto ciò che possiamo impostare per questa proprietà CSS è la dimensione, così come [nell'esempio 5.3](#).

Esempio 5.3

```
<!--il padding è lo stesso su tutti i lati-->
div
{
    padding: 5px;
}
```

```
<!--il primo valore rappresenta il padding superiore e inferiore,
il
secondo rappresenta il padding destro e sinistro -->
div
{
    padding: 5px 10px;
}
```

```
<!--i valori rispecchiano rispettivamente il padding superiore,
destro, inferiore e sinistro -->
div
{
    padding: 5px 10px 6px 8px;
}
```

Come si evince da questo esempio, impostare il padding è estremamente semplice. Volendo, possiamo anche impostare un valore personalizzato per ogni singolo lato dell'elemento tramite le proprietà `padding-top`, `padding-left`, `padding-bottom`, `padding-right`, che specificano rispettivamente il padding rispetto al bordo superiore, sinistro, inferiore e destro.

Sul padding non c'è altro da aggiungere quindi possiamo passare ad analizzare il margine, che è il terzo parametro CSS che influisce sulle dimensioni di un elemento.

Gestire il margine

Il margine rappresenta la distanza tra il bordo di un elemento e gli elementi circostanti. Come per il padding, del margine possiamo impostare solamente la dimensione, così come [nell'esempio 5.4](#).

Esempio 5.4

```
<!--il margine è lo stesso su tutti i lati-->
div
{
    margin: 5px;
}
```



```
<!--il primo valore rappresenta il margine superiore e inferiore,
il
secondo rappresenta il margine destro e sinistro -->
div
{
    margin: 5px 10px;
}

<!--i valori rispecchiano rispettivamente il margine superiore,
destro, inferiore e sinistro -->
div
{
    margin: 5px 10px 6px 8px;
}
```

Così come il padding, possiamo impostare un margine personalizzato per ogni lato dell'elemento tramite le proprietà `margin-top`, `margin-left`, `margin-bottom` e `margin-right`, che specificano rispettivamente il margine rispetto al bordo superiore, sinistro, inferiore e destro.

Ora che abbiamo analizzato le proprietà CSS che influiscono sul dimensionamento di un elemento, possiamo discutere più in dettaglio come i browser calcolano la dimensione di un elemento in base a queste proprietà CSS. Questo modello di calcolo viene definito **box model**.

Capire il box model

Quando renderizza un elemento (per esempio un `div`), il browser genera internamente un rettangolo che contiene l'elemento stesso. La dimensione del rettangolo è pari alla dimensione dell'elemento contenuto. Il rettangolo è mostrato nella [figura 5.1](#).

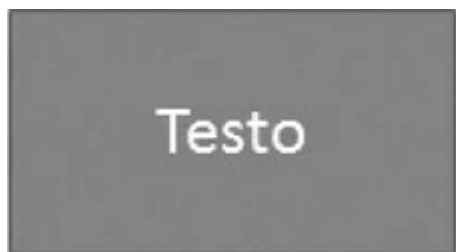


Figura 5.1 – Secondo il box model, il contenuto di un tag viene messo in un rettangolo.

A questo punto il browser prende in considerazione il padding. Aggiungendo questo spazio ai quattro lati del rettangolo, ne otteniamo uno nuovo più grande del precedente, o delle stesse dimensioni se il padding è 0.

A questo punto del processo, il box model ha due rettangoli, così come mostrato nella [figura 5.2](#) e le dimensioni dell'elemento non sono più solo quelle del testo contenuto, ma quelle del testo più il padding.



Figura 5.2 – Secondo il box model, il secondo rettangolo comprende l'area del primo rettangolo con l'aggiunta del padding.

Secondo le specifiche del box model, ora è il momento di creare un nuovo rettangolo, la cui dimensione è data da quella dei rettangoli precedenti con l'aggiunta del parametro CSS `border`. Ovviamente, se il bordo è 0, la dimensione di questo nuovo rettangolo è la stessa di quelli inclusi. Questo nuovo rettangolo è visibile nella [figura 5.3](#).



Figura 5.3 – Secondo il box model, il terzo rettangolo comprende l'area dei primi due rettangoli con l'aggiunta del bordo.

L'ultimo passo nel calcolo delle dimensioni di un elemento consiste nel creare un quarto rettangolo (che ovviamente include i precedenti), la cui dimensione è data dalla dimensione del precedente rettangolo più il parametro CSS `margin`. La dimensione di questo rettangolo rappresenta la dimensione finale dell'elemento. Questo nuovo rettangolo è visibile nella [figura 5.4](#).



Figura 5.4 – Secondo il box model, il quarto rettangolo comprende l'area dei primi tre rettangoli con l'aggiunta del margine esterno.

Ora che abbiamo capito come funziona il box model, facciamo un esempio per mettere in pratica quanto abbiamo appreso. Prendiamo come esempio il CSS nel prossimo codice.

Esempio 5.5

```
div
{
  width: 300px;
  height: 80px;
  padding: 5px;
  border: solid 1px black;
  margin: 10px;
```

}

Calcolare la dimensione di questo div è estremamente semplice. La larghezza è data dalla seguente formula: $\text{width} + (\text{padding-left} + \text{padding-right}) + (\text{border-left-width} + \text{border-right-width}) + (\text{margin-left} + \text{margin-right})$. Allo stesso modo, la formula per calcolare l'altezza di un elemento è: $\text{height} + (\text{padding-top} + \text{padding-bottom}) + (\text{border-top-width} + \text{border-bottom-width}) + (\text{margin-top} + \text{margin-bottom})$.

Il risultato finale è che la larghezza totale dell'elemento è 332 pixel e l'altezza totale è 112 pixel. La [figura 5.5](#) mostra in maniera visuale la formula.



Figura 5.5 – Il modo di calcolare la dimensione reale di un elemento è dato dalle sue dimensioni più padding, border e margin.

Il box model è tutto qui. Si tratta di un semplice pattern per il calcolo delle dimensioni di un elemento. Questo calcolo è fondamentale quando dobbiamo posizionare i nostri elementi sulla pagina e dare le corrette dimensioni.

Questo modello di calcolo delle dimensioni può essere modificato attraverso una tecnica denominata **box sizing** di cui parliamo nella prossima

sezione.

Personalizzare il box model con il box sizing

Il box model è un concetto molto semplice, ma non sempre conosciuto, almeno a livello teorico, da tutti coloro che sviluppano sul Web. Per semplificare la vita agli sviluppatori di ogni livello, è stata introdotta la proprietà CSS `box-sizing` attraverso la quale possiamo influenzare il box model.

Quando la proprietà CSS `box-sizing` non è specificata o viene impostata al valore `content-box`, il box model si comporta in maniera standard. Se invece la proprietà CSS `box-sizing` viene impostata a `border-box`, la dimensione di un elemento comprende anche il bordo e il padding, ma non il margine.

In questo caso la larghezza di un elemento è calcolata seguendo la formula `width + margin-left + margin-right`, mentre per l'altezza si applica la formula `height + margin-top + margin-bottom`.

Riprendendo l'esempio nella precedente sezione, il risultato finale è che la larghezza totale dell'elemento è 320 pixel e l'altezza totale è 100 pixel. La [figura 5.6](#) mostra in maniera visuale la formula.

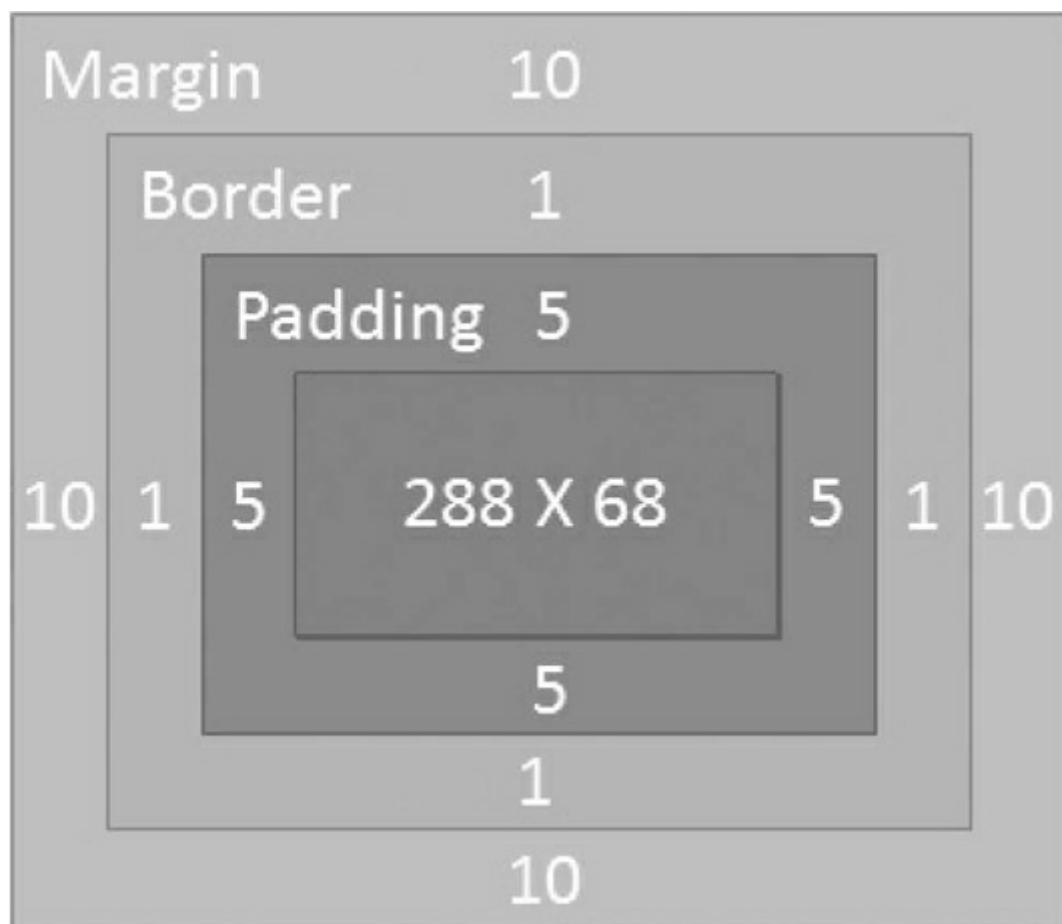


Figura 5.6 – Quando la proprietà `box-sizing` è impostata a `border-box`, la dimensione reale di un elemento si ottiene senza prendere in considerazione il margine.

Come si vede dalla [figura 5.6](#), il border e il padding esistono ancora. E sono inclusi nei 300 pixel della larghezza dell'elemento. Questo significa che l'area all'interno dell'elemento ha uno spazio ridotto rispetto allo spazio a disposizione quando il box model si comporta in maniera standard.

Ora che il box model non ha più segreti, possiamo analizzare un'altra proprietà CSS, che influenza la renderizzazione dell'oggetto ma non ha effetti sul box model: si tratta di `outline`.

Outline

La proprietà CSS `outline` permette di disegnare un secondo bordo intorno all'elemento esattamente come la proprietà CSS `border`. La proprietà `outline` viene renderizzata dopo il bordo reale ma, a differenza di quest'ultimo, non influisce sulla dimensione dell'elemento. Inoltre, se l'elemento ha un margine, l'`outline` viene disegnato sopra il margine stesso.

La proprietà CSS `outline` può contenere gli stessi valori della proprietà CSS `border`, quindi si possono specificare stile, colore, dimensioni e arrotondamento sia attraverso le singole proprietà `outline-style`, `outline-color` e `outline-width` sia attraverso la proprietà semplificata `outline`. [Nell'esempio 5.6](#) possiamo vedere un esempio di utilizzo della proprietà `outline`.

Esempio 5.6

```
<!--proprietà singole-->
div
{
    ...
    outline-color: #555;
    outline-style: solid;
    outline-width: 5px;
}

<!--proprietà unica-->
div
{
    ...
```

```
outline: solid 5px #555;  
}
```

Il risultato di un elemento `div` con questo CSS è visibile nella [figura 5.7](#).

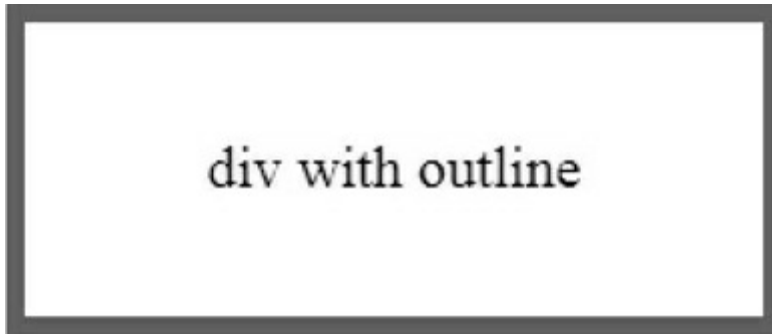


Figura 5.7 – Il bordo esterno dell’elemento rappresenta l’outline dell’elemento stesso.

Capire la disposizione di un elemento

Quando un elemento viene renderizzato sulla pagina, ci sono tre proprietà CSS che influenzano come questo viene disposto nella pagina: `float`, `visibility` e `display`. Il layout di un’intera pagina è fortemente influenzato dal valore che queste proprietà CSS possono assumere. Nel precedente capitolo abbiamo già visto la proprietà CSS `float`, quindi in questa sezione analizzeremo `visibility` e `display`, cominciando dalla prima.

Impostare la proprietà CSS `visibility`

Come si può facilmente intuire dal nome, la proprietà CSS `visibility` imposta la visibilità di un elemento. Di default, questa proprietà CSS è impostata su `visible`, il che significa che l’elemento è visibile nel browser.

Il secondo valore che possiamo impostare per la proprietà CSS `visibility` è `hidden`. Quando impostiamo la proprietà CSS `visibility` a `hidden`, l’elemento viene reso invisibile, ma continua a occupare il suo spazio nella pagina (spazio calcolato sempre secondo le regole del box model).

Per capire questo concetto, prendiamo come esempio tre elementi `span`, dove il primo e il terzo elemento sono visibili, mentre il secondo è invisibile, così come mostrato [nell’esempio 5.7](#).

Esempio 5.7

```
<span>Span 1</span>  
<span style="visibility:hidden">Span 2</span>  
<span>Span 3</span>
```

Il risultato è che il primo elemento `span` viene renderizzato ed è visibile nel browser, il secondo elemento `span` viene renderizzato e occupa il suo spazio (seppur invisibile) e il terzo elemento `span` è normalmente visibile. Il risultato dell'esempio 5.7 è visibile nella [figura 5.8](#).



Figura 5.8 – Il secondo elemento `span` viene renderizzato e occupa spazio nonostante sia invisibile.

Impostare la visibilità di un elemento mantenendo comunque lo spazio per l'elemento stesso è comodo in alcuni casi, ma in altri emerge la necessità di nascondere l'elemento e di eliminarne anche lo spazio occupato sulla pagina. In questi casi torna utile la proprietà CSS `display`.

Impostare la proprietà CSS `display`

La proprietà CSS `display` gestisce la renderizzazione di un elemento. Questa proprietà può avere diversi valori, ma quelli più importanti sono: `inline`, `block`, `inline-block` e `none`. Nelle prossime sezioni entreremo nel dettaglio di ogni valore, a cominciare da `inline`.

`inline`

Quando la proprietà CSS `display` viene impostata su `inline`, l'elemento viene renderizzato accanto a quello precedente. Nella [figura 5.9](#) possiamo vedere due elementi renderizzati inline.

Questo è il primo elemento con <code>display:inline</code>	Questo è il secondo elemento con <code>display:inline</code>
--	--

Figura 5.9 – I due elementi con `display` valorizzato con `inline` vengono renderizzati l'uno di seguito all'altro.

Un elemento renderizzato inline può essere separato in più blocchi. Supponiamo di avere un elemento contenitore che abbia una larghezza di 400 pixel. All'interno di questo contenitore ci sono due elementi visualizzati inline dove il primo ha una larghezza di 350 pixel e il secondo di 200 pixel. Quello che succede in questo caso è che i primi 50 pixel del secondo elemento vengono renderizzati accanto al primo elemento, mentre i restanti 150 pixel vengono visualizzati sotto il primo elemento, come se fossimo andati a capo con un tag `br`. La [figura 5.10](#) mostra visivamente questo esempio.

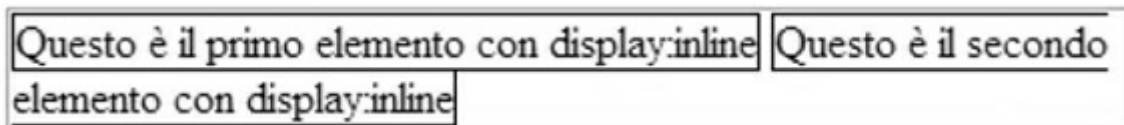


Figura 5.10 – L'elemento contenitore ha il bordo più scuro. Il secondo elemento inline viene visualizzato in parte sulla prima riga e in parte sulla seconda, andando a capo.

nota

Alcuni elementi vengono automaticamente renderizzati inline, quindi non abbiamo bisogno di impostare la proprietà `display` su `inline`. Tra questi elementi, i più usati sono i tag `input`, `select`, `span` e `label`.

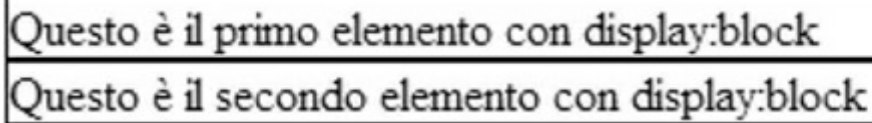
La modalità di rendering inline è sicuramente comoda, ma il fatto che un elemento possa essere suddiviso in più blocchi può rappresentare un problema. Vedremo nella prossima sezione come superare il problema.

block

Quando la proprietà CSS `display` viene impostata su `block`, l'elemento viene renderizzato come un blocco unico e non accetta altri elementi, né a destra, né a sinistra. Questa modalità di visualizzazione è in piena contrapposizione con la modalità inline, in quanto il fatto di non accettare elementi né a destra, né a

sinistra non permette di avere elementi uno accanto all'altro e quindi elimina anche eventuali problemi legati alla renderizzazione dell'elemento stesso in più blocchi.

Se, per esempio, abbiamo due elementi con la proprietà CSS `display` impostata a `block`, questi vengono mostrati uno sotto l'altro, così come mostrato nella [figura 5.11](#).



Questo è il primo elemento con `display: block`
Questo è il secondo elemento con `display: block`

Figura 5.11 – I due elementi con `display` valorizzato con `block` vengono renderizzati l'uno sotto l'altro.

nota

Alcuni elementi come `div`, `p`, `ul`, `ol` e `li` sono renderizzati automaticamente come blocchi, quindi per questi elementi non dobbiamo impostare la proprietà CSS `display` con `block`.

La modalità di visualizzazione a blocco è comoda quando vogliamo che l'elemento non venga diviso in più blocchi. Tuttavia, questa modalità presenta l'inconveniente di non permettere di avere due elementi uno di fianco all'altro. Nel capitolo precedente abbiamo visto come utilizzando la proprietà CSS `float` possiamo raggiungere questo scopo, ma a partire da CSS3, abbiamo a disposizione un nuovo valore della proprietà CSS `display` che offre una maggior versatilità: `inline-block`.

inline-block

Un elemento che ha la proprietà CSS `display` impostata a `inline-block` viene mostrato di seguito all'elemento precedente (come nella modalità `inline`), ma non può suddividersi in più blocchi (come nella modalità a blocchi). Se l'elemento supera il bordo del contenitore, allora viene visualizzato al di sotto degli elementi precedenti. La [figura 5.12](#) mostra chiaramente la disposizione degli elementi.



Figura 5.12 – L’elemento contenitore ha il bordo più scuro. Il primo e il secondo elemento vengono disposti inline. Il terzo elemento supera il bordo del contenitore, quindi viene disposto al di sotto degli elementi precedenti.

L’impostazione `inline-block` è molto versatile e in futuro sostituirà l’utilizzo di `float` che finora è stato l’unico parametro che ha permesso di disporre consecutivamente elementi renderizzati in blocco. Passiamo ora all’ultima modalità di renderizzazione: `none`.

none

Un elemento che ha la proprietà CSS `display` impostata su `none` non viene renderizzato sul browser ed è quindi invisibile all’utente. La differenza tra `display` valorizzato con `none` e `visibility` valorizzato con `hidden` è che nel primo caso l’elemento non viene renderizzato, quindi non occupa alcuno spazio sulla pagina.

Per esempio, se abbiamo tre elementi `span` uno dietro l’altro e il secondo ha la proprietà CSS `display` impostata a `none`, il primo e il terzo elemento vengono visualizzati uno a fianco all’altro, in quanto il secondo non occupa spazio nella pagina. Questo esempio è chiaramente visibile nella [figura 5.13](#).



Figura 5.13 – L’elemento con `display` a `none` non occupa spazio nella pagina.

Con quest’ultima modalità di visualizzazione abbiamo esaurito l’argomento relativo alla modalità di renderizzazione degli elementi, quindi possiamo passare al prossimo argomento che riguarda la gestione del cursore.

Gestire il cursore

Il cursore non è solamente il puntatore sul monitor che muoviamo attraverso il mouse. Spesso, il cursore è anche un veicolo di informazioni sullo stato della pagina. Per esempio, molte applicazioni renderizzano il cursore come una clessidra quando è in corso un’elaborazione oppure come una mano quando si passa su un oggetto su cui si può cliccare. In questa sezione analizzeremo i diversi stati in cui possiamo impostare il cursore, così da poter utilizzare questa importante risorsa nel modo migliore per l’utente. La proprietà CSS attraverso la quale possiamo impostare il cursore è `cursor`.

Questa proprietà CSS può avere una serie di valori predefiniti, che sono elencati nella [tabella 5.2](#).

Proprietà	Descrizione
<code>default</code>	Il cursore mostrato è quello di default per l’oggetto corrente. Questo significa che se il cursore si trova su un link, questo viene mostrato con una mano, se il cursore si trova su un testo, viene renderizzato come una linea verticale ecc.
<code>crosshair</code>	Il cursore viene mostrato come un simbolo +. Questa visualizzazione è generalmente utilizzata dai programmi di grafica per disegnare linee.
<code>n-resize</code> <code>s-resize</code> <code>w-resize</code> <code>e-resize</code>	Il cursore viene mostrato come una linea verticale per i primi due valori e come una linea orizzontale per il terzo e per il quarto valore. Agli estremi delle linee ci sono delle frecce. Queste modalità di visualizzazione del cursore vengono utilizzate quando si ridimensiona un oggetto in altezza per i primi due valori, e in larghezza per gli ultimi due valori.
<code>ne-resize</code> <code>nw-resize</code> <code>se-resize</code> <code>sw-resize</code>	Il cursore viene mostrato come una linea diagonale dall’alto, rispettivamente verso sinistra e verso destra, dove agli estremi ci sono delle frecce. Queste modalità di visualizzazione del cursore vengono utilizzate quando si ridimensiona un oggetto contemporaneamente sia in altezza sia in larghezza.
<code>move</code>	Il cursore viene mostrato come quando si esegue il drag di un oggetto.
	Il cursore viene mostrato come una mano. Questa è la

pointer	visualizzazione di default quando il cursore si trova sopra un link.
text	Imposta il cursore in modalità di selezione testo con una linea verticale. Questa è la visualizzazione di default quando si passa su un testo o su un campo di testo.
wait progress	Imposta il cursore in modalità di attesa. La differenza tra i valori varia a seconda del sistema operativo e delle versioni. Per esempio, in ambiente Windows il primo valore mostra il cursore come un cerchio animato che ruota, mentre il secondo valore mostra il cursore standard con accanto un piccolo cerchio animato che ruota.

Tabella 5.2 – Elenco dei valori della proprietà cursor.

Ora che abbiamo visto quali sono i valori che può avere il cursore, vediamo come impostare il valore da CSS [nell'esempio 5.8](#).

Esempio 5.8

```
div
{
    ...
    cursor: pointer;
}
```

Come si può notare, impostare il cursore è estremamente semplice. Il risultato di questa classe CSS è che quando si passa con il cursore sopra l'elemento `div`, il cursore diventa come nella [figura 5.14](#).

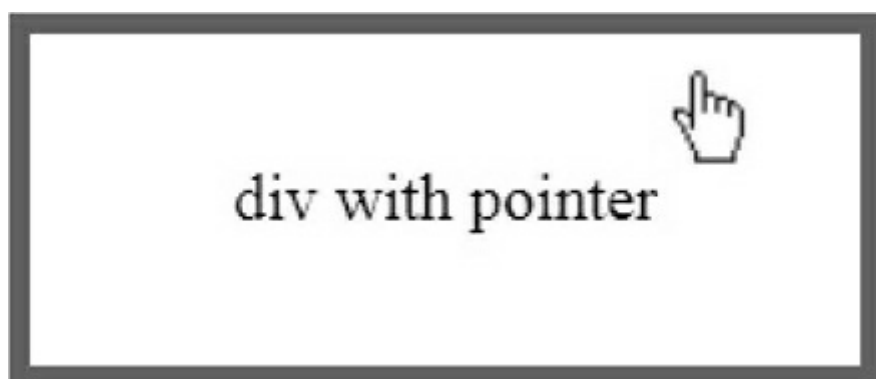


Figura 5.14 – Il cursore viene mostrato in stile `pointer` quando viene passato

sull'elemento `div`.

La gestione del cursore è uno di quegli aspetti che spesso vengono ignorati ma che in realtà risulta essere ottimale per offrire un'interfaccia gradevole all'utente: quindi è bene gestire sempre questa risorsa al meglio.

Ora che abbiamo visto come gestire il cursore, passiamo al prossimo paragrafo, che si occupa di come selezionare il CSS da utilizzare in base a determinate condizioni.

Le media query

Una delle caratteristiche più utili dei CSS risiede nel fatto che ci permettono di personalizzare completamente lo stile di una pagina semplicemente cambiando le proprietà CSS e senza toccare nulla nel codice HTML.

Così come possiamo cambiare lo stile di un sito, possiamo anche adattarne il contenuto in base al device che lo visualizza. Per esempio, se il device su cui il sito è visualizzato è un monitor in 16:9, potremmo ottimizzare la disposizione degli elementi in orizzontale; se il device è uno smartphone (quindi un monitor di dimensioni ridotte), potremmo nascondere alcuni elementi e ottimizzare la visualizzazione in verticale.

Creare un solo CSS che gestisce tutti i tipi di visualizzazioni è semplicemente impossibile per l'enorme complessità di manutenzione. Per questo motivo, la tecnica migliore consiste nel creare un CSS per ogni tipo di visualizzazione che intendiamo supportare e creare delle regole in base alle quali sarà il browser stesso a decidere quale CSS utilizzare.

Le specifiche CSS supportano la scelta del CSS da utilizzare in base a due caratteristiche: il tipo di device su cui viene visualizzato il sito e le sue caratteristiche.

Nel primo caso si distingue tra monitor, stampante e TV. Questa distinzione avviene quindi in base al device e non in base alle sue caratteristiche. Questo significa che se volessimo applicare un CSS diverso a seconda della risoluzione del monitor o della TV non potremmo farlo. Per questo la selezione del CSS in base alle caratteristiche è molto più utile anche se esistono casi in cui differenziare per device può tornare comodo, come nel caso si voglia usare un CSS differente per la stampa.

Grazie alle **media query** possiamo decidere quale CSS utilizzare, non solo in base al device, ma anche in base alle sue caratteristiche tecniche come dimensioni, supporto ai colori, risoluzione DPI e orientamento.

Cominciamo col vedere come supportare diversi CSS in base al tipo di device.

Utilizzare un CSS in base al device

Supponiamo di avere una pagina in cui vogliamo che la dimensione del carattere sia di 12 pixel sul monitor e di 10 pixel in stampa. In questo caso possiamo scrivere il seguente codice HTML.

Esempio 5.9

```
<html>
<head>
  <title></title>
  <style type="text/css">
    @media screen { body { font-size: 12px } }
    @media print { body { font-size: 10px } }
  </style>
</head>
<body>
  ...
</body>
</html>
```

La direttiva `media` accetta come parametro il tipo di device da utilizzare seguito dalla dichiarazione di un CSS all'interno del quale definiamo i nostri stili. Esistono diversi tipi di device, ma quelli usati nella stragrande maggioranza dei casi sono `screen` e `print` che non necessitano di spiegazioni. Se vogliamo specificare che un CSS vale per tutti i device, possiamo utilizzare la parola chiave `all`. Ora che siamo in possesso delle basi, vediamo come ottimizzare il processo attraverso l'uso dei file CSS.

Registrare un file CSS in base al device

Definire l'intero foglio di stile all'interno della pagina non è una cosa ottimale per il riutilizzo del codice. Per creare dei fogli di stile in un file esterno e registrarli all'interno della pagina in base al device, possiamo utilizzare la direttiva `import` o il tag HTML `link`.

Un esempio di queste tecniche è mostrato nel codice che segue.

Esempio 5.10

```
<!-- registrazione con direttiva import -->
<style type="text/css">
    @import url(stylesheetscreen.css) screen;
    @import url(stylesheetprint.css) print;
</style>

<!-- registrazione con tag link -->
<link rel="stylesheet" media="screen" href="stylesheetscreen.css"
/>
<link rel="stylesheet" media="print" href="stylesheetprint.css" />
```

Queste tecniche sono decisamente migliori di quelle viste [nell'esempio 5.9](#). Tuttavia presentano l'inconveniente di dover specificare in ogni pagina il file CSS da utilizzare in base al device. Possiamo migliorare ulteriormente la situazione creando un nuovo file CSS e inserendo al suo interno le direttive `import` viste nel precedente esempio. In questo modo, nella pagina faremo riferimento a un solo file CSS al quale demandiamo la gestione del tipo di device. Questa tecnica è mostrata [nell'esempio 5.11](#).

Esempio 5.11

```
<!-- registrazione del file CSS nella pagina -->
<link rel="stylesheet" href="stylesheetmain.css" />

<!-- codice del file stylesheetmain.css -->
@import url(stylesheetscreen.css) print;
@import url(stylesheetprint.css) screen;
```

A questo punto sappiamo come registrare un file CSS in base al device e come ottimizzare questa registrazione. Adesso vediamo come personalizzare ulteriormente il processo di scelta del CSS in base alle caratteristiche tecniche del device.

Utilizzare un CSS in base alle caratteristiche del device

Come detto in precedenza, le caratteristiche tecniche del device in base alle quali selezionare il CSS da utilizzare sono fisse e sono: dimensioni, orientamento e rapporto tra dimensioni e scala di colori. Partiamo col vedere come selezionare il CSS da utilizzare in base alle dimensioni del device.

Dimensioni

Sicuramente la discriminante più usata per selezionare un CSS è la dimensione del device, in quanto i nostri siti devono essere visualizzabili su browser di dimensioni eterogenee: dallo smartphone al monitor da 27 pollici alla TV. Nel prossimo esempio vediamo come specificare un CSS diverso in base alle dimensioni attraverso le clausole `device-width` e `device-height`.

Esempio 5.12

```
@import url(stylesheets/400x600.css)
(device-width: 400px) and (device-height: 600px);
```

In questo esempio possiamo vedere che le caratteristiche del device in base alle quali selezionare il CSS sono separate dalla parola chiave `and` e sono racchiuse tra parentesi tonde.

La parola chiave `and` indica che le caratteristiche specificate devono essere tutte vere affinché il browser usi il CSS specificato. Oltre alla parola chiave `and`, possiamo utilizzare altri operatori logici, come `not` per specificare una negazione e la virgola (,) per concatenare più condizioni di cui basta che una sola sia vera per selezionare il CSS. Un esempio di come usare la virgola è mostrato nel codice che segue.

Esempio 5.13

```
@import url(stylesheets.css)
(device-width: 200px) and (device-height: 300px),
(device-width: 300px) and (device-height: 600px);
```

In questo caso, vengono considerate le due condizioni in maniera separata e se almeno una è vera, il CSS viene selezionato.

L'ultima parola chiave da tenere a mente è `only`. Se la mettiamo all'inizio della media query, i vecchi browser non capiscono la media query e quindi la ignorano. I nuovi browser la tralasciano, prendendo in considerazione il resto della query. In questo modo possiamo fare in modo che browser obsoleti non usino un CSS studiato per browser moderni.

nota

Quando specifichiamo più caratteristiche, il browser utilizza il primo

CSS le cui caratteristiche sono **tutte** rispettate dal device. Se anche una sola caratteristica non è rispettata, il CSS non viene utilizzato.

Oltre alle dimensioni del device, possiamo anche selezionare un CSS in base alle dimensioni del browser (quando si usa un computer, il browser può essere di dimensioni diverse rispetto al monitor, in quanto possiamo ridimensionarlo come vogliamo), usando le clausole `width` e `height`. Volendo, possiamo anche lavorare con dei range, aggiungendo i prefissi `min-` e `max-` sia alle clausole `width` e `height` sia alle clausole `device-width` e `device-height`, così come mostrato nell'esempio 5.14.

Esempio 5.14

```
@import url(stylesheets1024xt68min.css)
(min-width: 1024px) and (min-height: 768px);
```

Selezionare il CSS in base alle dimensioni del device o del browser è molto comodo, ma spesso abbiamo bisogno di una tipologia di selezione più semplice, come l'orientamento.

Orientamento

L'orientamento di un device può essere una buona discriminante per scegliere il file CSS da utilizzare. Come detto in precedenza, se il device è in **landscape** (orientamento orizzontale), possiamo ottimizzare la visualizzazione del sito occupando la totale larghezza del device, mentre se il device è in **portrait** (orientamento verticale), abbiamo a disposizione meno pixel in larghezza, ma una maggiore altezza. Nell'esempio 5.15 vediamo come utilizzare la clausola `orientation` per selezionare il CSS in base all'orientamento del device.

Esempio 5.15

```
@import url(stylesheetsPortrait.css)
(orientation: portrait);
```

Oltre all'orientamento, anche il rapporto tra la larghezza e l'altezza del device può essere una discriminante.

Rapporto tra larghezza e altezza

I monitor attualmente in commercio hanno il formato 16:9 o 16:10. Tuttavia esistono ancora molti monitor datati che hanno il formato 4:3. Con le media query possiamo scegliere il CSS da utilizzare anche in base a questo formato, sfruttando le clausole `device-aspect-ratio` e `aspect-ratio`. La prima stabilisce il rapporto tra larghezza e altezza del device, mentre la seconda stabilisce il rapporto tra larghezza e altezza del browser.

Così come avviene per le clausole che specificano la dimensione, possiamo sfruttare un range di valori anteponendo i prefissi `min-` e `max-` alle clausole appena viste. [Nell'esempio 5.16](#) possiamo vedere delle dimostrazioni che mostrano come selezionare un CSS in base a queste clausole.

Esempio 5.16

```
@import url(stylesheetsheet16_9.css)
(device-aspect-ratio: 16/9)

@import url(stylesheetsheet4_3.css)
(aspect-ratio: 4/3)
```

Oltre a quelle già viste, ci sono altre clausole con cui selezionare il CSS da utilizzare, come la scala di colori e la risoluzione del device.

Ulteriori parametri di selezione del CSS

Per selezionare un CSS in base alla risoluzione del device, possiamo utilizzare la clausola `resolution`, che accetta anche i prefissi `min-` e `max-` per creare un range di valori.

Un'altra clausola di ricerca è la capacità del device di mostrare i colori e la quantità di colori a disposizione tramite le parole chiave `color` e `color-index`. Entrambe le clausole accettano i prefissi `min-` e `max-` per creare un range di valori.

Tutte queste clausole sono mostrate [nell'esempio 5.17](#).

Esempio 5.17

```
<!--Stampante con una risoluzione minima di 300dpi-->
@import url(stylesheetsheet300dpi.css)
print and (min-resolution: 300dpi)

<!--Tutti i device che supportano i colori-->
@import url(stylesheetsheetcolors.css)
all and (color)
```

```
<!--Tutti i device che supportano almeno 8 bit per colore-->
@import url(stylesheets8bitcolor.css)
    all and (min-color: 8)

<!--Tutti i device che supportano almeno 256 colori-->
@import url(stylesheets256colors.css)
    all and (min-color-index: 256)
```

Le media query sono molto potenti e ci permettono di eseguire selezioni da CSS che prima era possibile fare solo scrivendo codice. Questo è un altro esempio di come le specifiche attuali di CSS ci semplifichino notevolmente la vita.

nota

Per un ulteriore approfondimento delle specifiche delle media query, potete consultare direttamente il seguente link:
<http://aspit.co/a0r>.

Per semplificarci ulteriormente la vita, vediamo ora brevemente quali sono le media query più utilizzate per differenziare i vari tipi di device, dallo smartphone al monitor, passando per i tablet.

Media query più utilizzate

Esistono framework CSS come Bootstrap (di cui parleremo nel [capitolo 11](#)) che già offrono media query per diverse dimensioni del browser e quindi, implicitamente, ottimizzano anche la visualizzazione per tipo di device (smartphone, tablet, monitor). Tuttavia, se dobbiamo anche noi scrivere CSS ottimizzati per le dimensioni del browser, dobbiamo essere a conoscenza di questi formati

Esempio 5.18

```
<!--dimensioni del browser su smartphone-->
@import url(stylesheetsmartphones.css)
    (min-width: 768px)

<!--dimensioni del browser su tablet-->
@import url(stylesheettablet.css)
```

```
(min-width: 992px)

<!--dimensioni browser su monitor a full screen-->
@import url(stylesheets/monitor.css)
  (min-width: 1200px)

<!--retina display-->
@import url(stylesheets/retina.css)
  (-webkit-min-device-pixel-ratio: 2)
```

Ora possiamo passare al prossimo argomento del capitolo che sono le trasformazioni 2D degli elementi.

Applicare trasformazioni agli elementi

Storicamente, una delle mancanze di HTML e CSS è sempre stata l'impossibilità di eseguire **trasformazioni** sugli elementi. Semplici trasformazioni come il ruotare un elemento di 45 gradi o lo skew (un esempio di skew è la trasformazione da rettangolo a parallelepipedo) erano impossibili da ottenere se non lavorando con immagini, elementi `div`, CSS, JavaScript o con librerie JavaScript di terze parti.

Le specifiche di HTML e CSS contengono funzionalità che permettono di modificare gli elementi applicando trasformazioni 2D, semplicemente con l'utilizzo di alcune proprietà CSS.

Le proprietà CSS responsabili della trasformazione di un elemento sono `transform` e `transform-origin`. La prima indica il tipo di trasformazione che vogliamo effettuare, mentre la seconda indica da quale punto dell'elemento vogliamo applicare la trasformazione. I tipi di trasformazioni sono predefiniti e sono elencati nella [tabella 5.3](#) (nell'ordine in cui vengono applicati dal browser):

Proprietà	Descrizione
<code>matrix</code>	Specifica una trasformazione, fornendo i sei valori della matrice. Questa è la modalità che offre più versatilità, ma è anche quella più difficile da utilizzare, in quanto richiede conoscenze matematiche.
<code>translate</code> , <code>translateX</code> , <code>translateY</code>	Spostano l'elemento sull'asse orizzontale e/o verticale in base ai valori forniti.

<code>scale,</code> <code>scaleX,</code> <code>scaleY</code>	Ridimensionano la larghezza e/o l'altezza dell'elemento.
<code>rotate</code>	Ruota l'elemento dei gradi specificati.
<code>skewX,</code> <code>skewY, skew</code>	Specifica l'angolazione orizzontale e/o verticale dello skew di un elemento.

Tabella 5.3 – Elenco delle trasformazioni CSS.

Di default, la proprietà `transform-origin` prende come punto di partenza della trasformazione il centro dell'elemento. Volendo, possiamo modificarla specificando il punto di origine tramite la distanza (percentuale o assoluta) dal bordo destro e dal bordo superiore dell'elemento.

nota

Le specifiche per le trasformazioni 2D non sono ancora definitive. Per consultare le specifiche potete interrogare l'indirizzo:
<http://aspit.co/a0s>.

Nel prossimo esempio vediamo come applicare una trasformazione di tipo skew a un elemento `div`.

Esempio 5.19

```
div
{
  height: 100px;
  width: 100px;
  border: solid 1px black;
  transform: skew(20deg, 10deg);
  transform-origin: 50% 50%;
}
```

Il primo parametro della trasformazione specifica lo skew sull'asse delle X, mentre il secondo specifica lo skew sull'asse delle Y. Se il secondo parametro viene omissso, lo skew sull'asse delle Y è pari a zero. Come origine è stato impostato il centro dell'elemento (che è il valore di default, quindi può essere

omesso), ma cambiando le percentuali della proprietà `transform-origin` possiamo modificare questo comportamento. Nella [figura 5.15](#) possiamo vedere i vari passi della trasformazione.

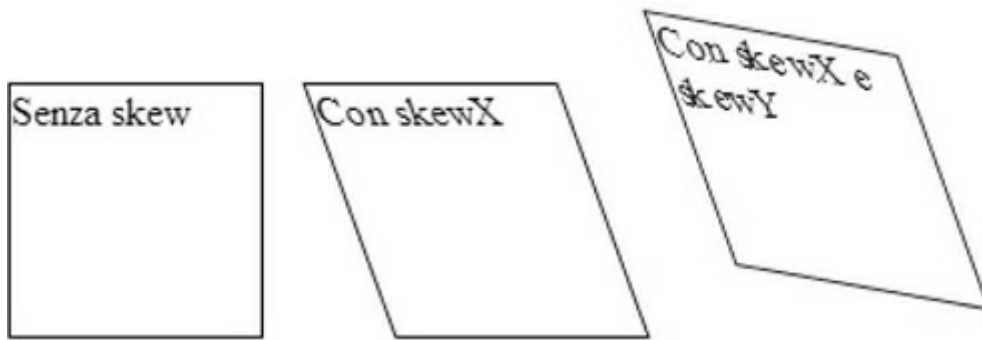


Figura 5.15 – Il primo elemento non ha alcuna trasformazione skew applicata. Al secondo elemento è stato applicato lo `skewX`, mentre al terzo elemento sono stati applicati sia lo `skewX` sia lo `skewY`.

Se volessimo ruotare un elemento potremmo utilizzare il seguente codice.

Esempio 5.20

```
div
{
  height: 100px;
  width: 100px;
  border: solid 1px black;
  transform: rotate(45deg);
}
```

La conseguenza di questa operazione è che l'elemento viene ruotato di 45 gradi in senso orario e il risultato è quello mostrato nella [figura 5.16](#).

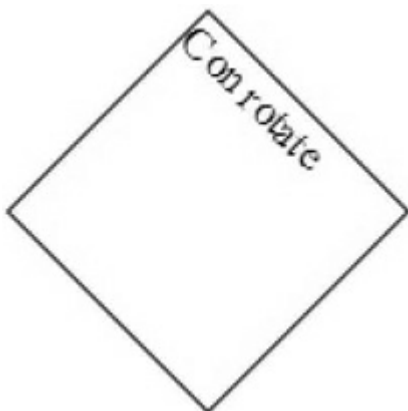


Figura 5.16 – Per via della trasformazione rotate, l'elemento viene ruotato di 45 gradi in senso orario.

Una delle cose più interessanti riguardo alle trasformazioni è che se ne può applicare più di una per ogni elemento. Per esempio, possiamo applicare le trasformazioni di tipo skew, rotate e scale semplicemente separandole con uno spazio. Il prossimo codice mostra il codice di questo esempio.

Esempio 5.21

```
div
{
    height: 100px;
    width: 100px;
    border: solid 1px black;
    transform: rotate(45deg) skew(20deg, 10deg) scale(1.5, 1.5);
}
```

Le trasformazioni sono molto semplici da capire e da applicare, fatta eccezione per la trasformazione di tipo matrix, che comporta calcoli matematici. Grazie a questa funzionalità, possiamo rendere le nostre pagine ancora più gradevoli per l'utente finale senza dover fare uno sforzo eccessivo.

Ora che le trasformazioni sono state trattate in maniera esaustiva, possiamo passare all'ultimo argomento del capitolo: le **transizioni**.

Gestire le transizioni

Quando un elemento viene renderizzato, gli stili CSS vengono applicati all'elemento e questo entra in un determinato stato. Mentre l'utente si trova sulla pagina, questo elemento può cambiare stile CSS, diventando invisibile, più grande, più piccolo e molto altro ancora.

Quando lo stile CSS di un elemento cambia, c'è una transizione dallo stato attuale al nuovo stato. Di default, questa transizione è immediata, ma possiamo impostarne la durata. In questo modo la transizione dal vecchio al nuovo stile avviene in maniera più gradevole.

nota

Così come per le trasformazioni 2D, le specifiche per le transizioni

non sono complete. Le specifiche possono essere consultate sul sito all'indirizzo: <http://aspit.co/a0q>.

Supponiamo di avere un elemento `div` di cui vogliamo modificare lo stile quando ci passiamo sopra con il mouse. Se volessimo modificare il colore di sfondo, il colore del testo e allargare l'elemento, dovremmo usare il seguente codice CSS.

Esempio 5.22

```
div
{
    width: 300px;
    height: 80px;
    border: solid 1px black;
    background-color:white;
    transition-property: background-color, width, color;
    transition-duration: 2s;
}

div:hover
{
    background-color: black;
    width: 600px;
    color: #fff;
}
```

La prima classe CSS specifica le caratteristiche iniziali dell'elemento `div`, quali proprietà CSS sono soggette a transizione e quanto dura la transizione dal vecchio al nuovo stile. Queste ultime due caratteristiche sono specificate rispettivamente attraverso le proprietà `transition-property` e `transition-duration`.

La proprietà `transition-property` accetta la lista delle proprietà soggette a transizione, separate dalla virgola. La proprietà `transition-duration` accetta una lista di durata delle transizioni. Se specifichiamo un solo valore, tutte le transizioni per le proprietà specificate in `transition-property` avranno la stessa durata. Se specifichiamo più valori (separandoli con la virgola) questi valori vengono ripetuti. Per esempio, se specifichiamo quattro proprietà e due durate, la durata della transizione per la prima e la terza proprietà sarà il primo valore specificato in `transition-duration`, mentre la durata della transizione per la seconda e la quarta proprietà sarà il valore della

seconda durata specificata in `transition-duration`.

nota

Nel nostro caso la durata è espressa in secondi attraverso il suffisso `s`. Possiamo specificare la durata in millisecondi, utilizzando il suffisso `ms`.

La seconda classe CSS [dell'esempio 5.22](#) specifica i nuovi valori che devono avere le proprietà soggette a transizione quando passiamo con il mouse sopra l'elemento.

La [figura 5.17](#) mostra la transizione specificata [nell'esempio 5.21](#).



Figura 5.17 – La prima immagine mostra l'elemento `div` nello stato iniziale. La seconda immagine mostra l'elemento `div` a metà della transizione. La terza immagine mostra l'elemento `div` alla fine della transizione.

Oltre a specificare la durata di una transizione, possiamo anche specificarne il ritardo iniziale attraverso la proprietà `transition-delay`. Riprendendo [l'esempio 5.22](#), possiamo stabilire che la transizione inizia solo se il mouse rimane all'interno dell'elemento `div` per almeno un secondo attraverso il seguente codice.

Esempio 5.23

```
div
{
```

```
...
transition-property: background-color, width, color;
transition-duration: 2s;
transition-delay: 1s;
}
```

Utilizzando la durata della transizione, il browser calcola in automatico i valori della transizione stessa. Riprendendo sempre [l'esempio 5.22](#), il browser sa che deve allargare l'elemento `div` di 300 pixel in due secondi e quindi esegue i calcoli matematici per garantire che la transizione avvenga in maniera graduale.

Se necessario, possiamo decidere come i valori intermedi della transizione vengono calcolati attraverso la proprietà CSS `transition-timing-function`. Questa proprietà può avere una serie di valori predefiniti (le cosiddette funzioni di **easing**), oppure un valore personalizzato. Sia le funzioni predefinite sia il valore personalizzato sono basati sulla **curva di bezier**. I valori che può assumere la proprietà `transition-timing-function` sono elencati di seguito:

- `ease`: equivale ai valori (0.25, 0.1, 0.25, 1.0) della curva di bezier;
- `linear`: equivale ai valori (0.0, 0.0, 1.0, 1.0) della curva di bezier;
- `ease-in`: equivale ai valori (0.42, 0, 1.0, 1.0) della curva di bezier;
- `ease-out`: equivale ai valori (0, 0, 0.58, 1.0) della curva di bezier;
- `ease-in-out`: equivale ai valori (0.42, 0, 0.58, 1.0) della curva di bezier;
- `cubic-bezier`: permette di specificare dei valori personalizzati seguendo la sintassi `transition-timing-function: cubic-bezier(x1, y1, x2, y2)`.

Le transizioni sono estremamente potenti e permettono di creare effetti grafici che possono migliorare notevolmente l'usabilità dell'interfaccia utente. Inoltre, le transizioni possono essere associate alle trasformazioni, così da creare vere e proprie animazioni.

Combinare trasformazioni e transizioni

Nella sezione dedicata alle trasformazioni abbiamo visto come applicare una

trasformazione di tipo skew a un elemento. Alla trasformazione possiamo combinare una transizione, così da creare una vera e propria animazione. Naturalmente, non siamo limitati alla trasformazione di tipo skew, ma possiamo animare qualunque tipo di trasformazione. Nell'esempio che segue, prendiamo un elemento `div` che facciamo ruotare di 360 gradi, traslare di 100 pixel sull'asse orizzontale e ingrandire del 50%. A tutte le trasformazioni assegniamo una transizione della durata di due secondi.

Esempio 5.24

```
div
{
    height: 100px;
    width: 100px;
    border: solid 1px black;
    transition-property: transform;
    transition-duration: 2s;
}

div.animate
{
    transform: rotate(360deg) scale(1.5, 1.5) translateX(180px);
}
```

Poiché la proprietà da animare è `transform`, quando all'elemento `div` viene applicato lo stile `animate` che imposta la proprietà `transform`, si scatena la transizione. La trasformazione e la transizione sono visibili nella [figura 5.18](#), che mostra i vari stati della transizione.

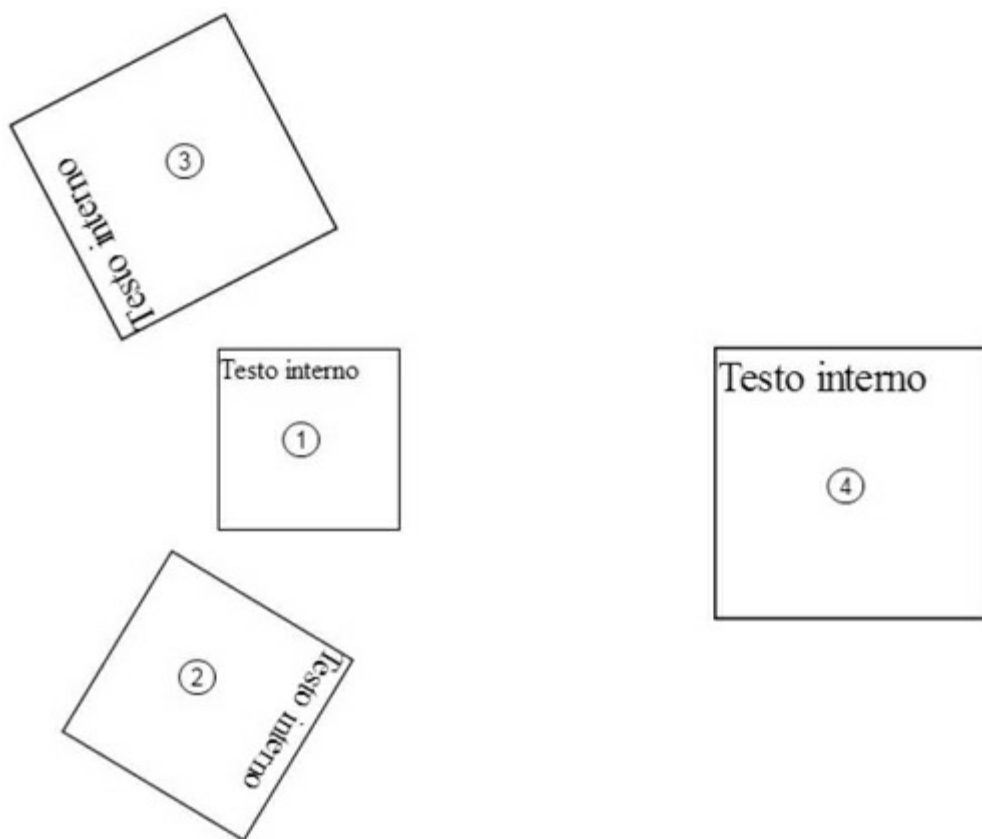


Figura 5.18 – L’elemento numero 1 è nello stato iniziale e l’elemento numero 4 è nello stato finale. Gli elementi con numero 2 e 3 rappresentano l’elemento durante la transizione.

L’esempio che abbiamo fatto in questa sezione è banale, ma rende l’idea di quello che possiamo fare combinando trasformazioni e transizioni. Inoltre, se combiniamo transizioni e trasformazioni su più elementi contemporaneamente, possiamo creare animazioni gradevoli e molto complesse, che prima dell’introduzione di questa funzionalità erano semplicemente impossibili da riprodurre se non usando pesantemente librerie JavaScript (e quindi complicando lo sviluppo) e perdendo comunque in termini di prestazioni (il browser passava più tempo a elaborare la grafica che a elaborare l’input dell’utente).

Conclusioni

In questo capitolo abbiamo analizzato molte funzionalità dei CSS che sono fondamentali se vogliamo realizzare un’interfaccia grafica moderna e gradevole per l’utente.

All’inizio del capitolo abbiamo analizzato il box model, illustrando come il browser calcola le dimensioni di un elemento, e includendo in queste anche il

marginale esterno, il bordo e il margine interno di un elemento. Inoltre abbiamo discusso come influenzare questo calcolo attraverso il box sizing.

Ci siamo poi occupati di come gestire un'importante risorsa come il cursore, analizzando come modificarne la visualizzazione a seconda dello stato della pagina.

Successivamente abbiamo visto come utilizzare un CSS diverso a seconda delle caratteristiche del device che visualizza la pagina, utilizzando le media query. Queste ci permettono di personalizzare il CSS a seconda che il nostro sito sia visualizzato su uno smartphone, un tablet o il monitor di un computer e in base ad alcune caratteristiche come le dimensioni, i colori supportati o l'orientamento.

Dopo le media query abbiamo visto come applicare trasformazioni 2D agli elementi HTML per farli ruotare, ingrandire, rimpicciolire e traslare scrivendo codice molto semplice.

Infine, abbiamo analizzato le transizioni che rendono animato il cambio di stile di un elemento come, per esempio, il ridimensionamento, lo spostamento, il cambio di colore di sfondo e altro ancora. Abbiamo inoltre visto come combinare trasformazioni e transizioni per creare animazioni che possono risultare vincenti in siti ad alta interattività.

Ora che abbiamo visto come creare effetti grafici di sicuro impatto per l'interfaccia utente, vediamo come utilizzare i CSS per la gestione tipografica.

Gestione tipografica

Dopo aver introdotto i fogli di stile e visto come usarli per gestire effetti, trasformazioni e transizioni, siamo pronti per capire come CSS consenta di controllare al meglio il layout per l'impaginazione del contenuto in modo tipografico. In questo capitolo vedremo infatti come definire il carattere da usare per la visualizzazione dei testi, come impostarne la dimensione, il colore e lo stile e come controllare la disposizione dei contenuti in colonne. Vedremo infine come personalizzare l'aspetto degli elementi, sfruttando le funzionalità offerte dai CSS relativamente alla gestione di sfondi e bordi, e come applicare questi ultimi per realizzare pagine dall'aspetto piacevole e accattivante.

Gestione del testo

Agli albori del Web e dell'HTML il carattere e lo stile del testo erano controllati esclusivamente dalle impostazioni del browser. Le pagine non potevano in alcun modo definire la modalità di visualizzazione del testo fino all'introduzione di HTML2 con il tag `font`, che rese possibile indicare quale carattere usare, purché presente sul client. Le prime specifiche di CSS (1996) non hanno di fatto aggiunto nulla per quanto riguarda il controllo del testo, se non la possibilità di separare formalmente l'impostazione dello stile dal contenuto vero e proprio (markup). Nel 1998 CSS2 ha introdotto il supporto per il download dei font ma questa tecnica, sebbene supportata da Internet Explorer fin dalla versione 4, non ha avuto particolare successo, tanto da venire rimossa dalle specifiche di CSS2.1. La [tabella 6.1](#) mostra alcune tra le principali impostazioni per la formattazione del testo mediante CSS.

<i>Proprietà CSS</i>	<i>Descrizione</i>
<code>font-family</code>	Imposta il tipo di carattere (font).

<code>font-size</code>	Imposta la dimensione del carattere.
<code>color</code>	Imposta il colore del testo.
<code>font-weight</code>	Definisce le impostazioni del grassetto da applicare al testo; i valori possibili sono <code>normal</code> , <code>bold</code> , <code>bolder</code> , <code>lighter</code> oppure un valore numerico tra 100 e 900, dove 400 corrisponde a <code>normal</code> e 700 a <code>bold</code> .
<code>font-style</code>	Imposta lo stile del font all'interno della stessa famiglia per l'uso del corsivo; i valori possibili sono <code>normal</code> , <code>italic</code> e <code>oblique</code> .
<code>font-variant</code>	Una variazione possibile del font è <code>small-caps</code> per cui le lettere minuscole vengono visualizzate con l'aspetto delle maiuscole ma più piccole e con proporzioni leggermente differenti.
<code>text-transform</code>	Trasforma il testo per esigenze stilistiche; per esempio, <code>capitalize</code> visualizza la prima lettera di ogni parola come maiuscola, <code>uppercase</code> visualizza tutto in maiuscolo e <code>lowercase</code> , al contrario, visualizza tutto in minuscolo.
<code>text-decoration</code>	Consente di impostare caratteristiche come la sottolineatura o il barrato.
<code>text-align</code>	Imposta l'allineamento del testo: <code>left</code> , <code>center</code> , <code>right</code> , <code>justify</code> .

Tabella 6.1 – Principali impostazioni del testo con CSS.

Oltre alle impostazioni di base presentate nella [tabella 6.1](#), CSS offre la possibilità di controllare numerose altre caratteristiche di visualizzazione del testo come l'interlinea, la distanza fra le lettere, l'indentazione, la gestione degli spazi, l'interruzione di linea ecc. Di seguito approfondiamo gli aspetti più comuni legati alla visualizzazione dei testi all'interno delle pagine web.

Impostazione del font

I caratteri usati per i testi sono uno degli aspetti che maggiormente caratterizzano un sito o un'applicazione web. Il numero di caratteri che è possibile impiegare è illimitato, ma si tende a suddividere i caratteri in due

categorie: “web safe” e “custom”.

Fanno parte della prima categoria i font che sono tendenzialmente presenti sulla maggior parte dei client. Questa lista include, per esempio, Arial/Helvetica, Times New Roman/Times, Courier New/Courier, Verdana, Georgia, Comic Sans MS, Trebuchet MS, Arial Black, Impact, Palatino, Garamond, Bookman e Avant Garde. L’elenco non è chiaramente completo e la disponibilità dipende dalla piattaforma: in alcuni casi, per esempio, ci sono caratteri molto simili ma con nomi diversi in Windows rispetto a Mac OS.

In generale è opportuno specificare come valore per la proprietà `font-family` un elenco di font (separati da virgola), così da avere un meccanismo di *fallback*: il browser, qualora il primo carattere non fosse disponibile, proverà ad adottare il secondo e così via. Se nessuno dei font specificati in elenco fosse presente sul client, verrà infine adottato quello predefinito dal browser.

L’esempio 6.1 mostra come indicare un carattere a larghezza fissa, specificando diversi nomi di font in modo da garantire una buona resa sui diversi sistemi operativi.

Esempio 6.1

```
div.codeSnippet { font-family: "Courier New", Courier, monospace;
}
```

La dichiarazione del carattere nell’esempio 6.1 definisce un gruppo di font simili, tanto da poter essere considerati equivalenti: `Courier New` per Windows, `Courier` per Mac e `monospace` per Linux/Unix. Con lo stesso principio possiamo creare diverse “famiglie” di font analoghi:

- `Arial, Helvetica, sans serif`
- `Arial Black, Gadget, sans serif`
- `Comic Sans MS, cursive`
- `Courier New, Courier, monospace`
- `Georgia, Georgia, serif`
- `Impact, Charcoal, sans serif`
- `Lucida Console, Monaco, monospace`

- Lucida Sans Unicode, Lucida Grande, sans serif
- Palatino Linotype, Book Antiqua, Palatino, serif
- Tahoma, Geneva, sans serif
- Times New Roman, Times, serif
- Trebuchet MS, Helvetica, sans serif
- Verdana, Geneva, sans serif

In questo modo siamo in grado di fornire al browser una tale serie di alternative da essere ragionevolmente sicuri che le nostre pagine verranno visualizzate in modo omogeneo su tutte le piattaforme.

Limitandoci ai font “web safe”, le possibilità di scelta sono piuttosto scarse, tanto che, da questo punto di vista, i diversi siti web tendono ad assomigliarsi. Queste limitazioni possono essere superate grazie alla seconda categoria di font cioè quelli “custom”. Grazie alle specifiche CSS, abbiamo la possibilità di incorporare ogni tipo di font (non solo quelli predefiniti) nel foglio di stile. Come abbiamo già detto in precedenza, non si tratta di una vera novità: Netscape Navigator 4 supportava l’incorporamento dei font attraverso un plug-in di Bitstream chiamato TrueDoc, così come anche Internet Explorer con la tecnologia Embedded Open Type (EOT).

Embedded Open Type consente di creare un file che può essere scaricato dal browser e che contiene le informazioni relative al carattere da usare nella pagina. Microsoft fornisce un programma gratuito per la creazione di file EOT partendo da un normale font, Microsoft Web Embedding Font Tool (WEFT, disponibile all’indirizzo <http://aspit.co/a05>).

EOT non è però un formato aperto, pertanto non è stato adottato dagli altri browser e dal W3C come standard per l’incorporamento dei font, a favore di altri formati come TrueType/OpenType, SVG e WOFF (Web Open Font Format). Quest’ultimo in particolare è un formato appositamente pensato per l’uso in pagine web ed è essenzialmente un “wrapper” che comprime al suo interno font in formato TrueType, OpenType o Open Font Format, consentendo di ridurre il peso del file di oltre il 40% rispetto a TTF e risultando pertanto ideale per il download.

Per incorporare un font in un CSS è sufficiente dichiarare una nuova famiglia di font collegata al carattere desiderato. [L’esempio 6.2](#) mostra la sintassi di base per includere un nuovo font di tipo TrueType e per utilizzarlo

come carattere per la visualizzazione dei titoli.

Esempio 6.2

```
@font-face
{
    font-family: MyFont;
    src: url(myfont.ttf) format('truetype');
}

h1 { font-family: MyFont, Arial, Helvetica, sans serif; }
```

La sintassi per l'incorporamento di un font mostrata [nell'esempio 6.2](#) è molto semplice: attraverso `font-family` si assegna un nome alla famiglia di caratteri mentre con `src` si specifica il percorso del file contenente il font vero e proprio e il relativo formato.

Poiché oggi non tutti i browser supportano tutti i formati, è opportuno fornire diverse alternative nella dichiarazione di `@font-face`, eventualmente comprendendo EOT per garantire la compatibilità anche con le vecchie versioni di Internet Explorer. [L'esempio 6.3](#) mostra appunto come, per lo stesso font, sia possibile indicare il collegamento ai sorgenti in diversi formati, per assicurare la maggior compatibilità possibile con i diversi dispositivi.

Esempio 6.3

```
@font-face
{
    font-family: 'MyFont';
    src: url('myfont.eot');
    src: url('myfont.eot?iefix') format('eot'),
        url('myfont.woff') format('woff'),
        url('myfont.ttf') format('truetype'),
        url('myfont.svg#myfont') format('svg');
}
```

[Nell'esempio 6.3](#) la prima dichiarazione `url('myfont.eot')` è usata per le vecchie versioni di Internet Explorer, così come l'indirizzo contenente il punto di domanda (“myfont.eot?iefix”) che, non venendo riconosciuto come valido, impedisce a IE di interpretare le linee successive. Il formato SVG viene utilizzato invece per compatibilità con le prime versioni di iOS (iPhone e iPad) e TTF è indicato per alcune versioni di browser per dispositivi mobile. Infine WOFF, che costituisce indubbiamente l'opzione migliore ed è

supportato da tutti i browser moderni.

Per ottenere i diversi formati di un stesso font, dato per esempio un file .ttf, possiamo usare programmi specifici oppure, più semplicemente, utilizzare servizi online come Font Squirrel, che mette a disposizione un generatore completo e gratuito. È sufficiente accedere all'indirizzo <http://aspit.co/a06>, caricare il file (o i file se desideriamo incorporare più font personalizzati all'interno del nostro sito), selezionare le opzioni di esportazione e scaricare il kit generato che, oltre al font nei diversi formati richiesti, contiene anche una pagina dimostrativa e il file CSS con le dichiarazioni necessarie.

nota

Per maggiori informazioni sull'uso dei font nelle pagine web, è possibile vedere la pagina “CSS Fonts Module Level 3” all'indirizzo <http://aspit.co/a07>.

È importante notare che, prima di incorporare un font nelle nostre pagine, ci dobbiamo assicurare di avere i diritti di licenza per l'utilizzo e la ridistribuzione.

Online ci sono numerosi siti e risorse dedicati ai font che, tra l'altro, consentono anche di effettuare ricerche in base al tipo di licenza (Free, Free for personal use, Shareware ecc.) come, per esempio, dafont, <http://aspit.co/a08>.

Dopo aver visto come definire il carattere da usare per la visualizzazione del testo, vediamo come impostarne la dimensione.

Impostazione della dimensione del carattere

Una volta selezionato il font, vogliamo controllare la dimensione con cui verrà visualizzato all'interno della pagina. Normalmente i browser assegnano una grandezza predefinita al testo, tipicamente di 12pt per il testo in paragrafi (è maggiore, per esempio, per elementi come i titoli H1, H2, H3 ecc.).

Possiamo impostare la dimensione del testo attraverso lo stile, specificando la proprietà `font-size`. La grandezza del carattere può essere espressa come assoluta attraverso le costanti `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large` e `xx-large` oppure come relativa con `larger` e

smaller. In questo modo, la misura viene impostata in proporzione rispetto alle impostazioni del browser.

In alternativa, possiamo specificare un valore preciso per la dimensione del testo, usando una delle unità di misura di lunghezza tra quelle disponibili ed elencate nella [tabella 6.2](#).

<i>Unità</i>	<i>Tipo</i>	<i>Descrizione</i>
%	Relativa	Rispetto alla dimensione dell'elemento contenitore (parent).
em	Relativa	Rispetto alla dimensione del font dell'elemento contenitore (o dell'elemento stesso, se non usata per esprimere il valore della proprietà font-size).
ex	Relativa	Rispetto a x-height, ovvero all'altezza delle lettere minuscole.
px	Relativa	Pixel; dipende dalla risoluzione del dispositivo.
gd	Relativa	Rispetto alla griglia definita dalla proprietà layout-grid.
rem	Relativa	Rispetto alla dimensione del font dell'elemento principale (root).
vw	Relativa	Rispetto alla larghezza dell'area di visualizzazione (viewport); la larghezza totale è pari a 100vw; quando l'area di visualizzazione cambia (per esempio, ridimensionando la finestra del browser) le dimensioni espresse con questa unità di misura vengono scalate in proporzione.
vh	Relativa	Rispetto all'altezza dell'area di visualizzazione (viewport); l'altezza totale è pari a 100vh; quando l'area di visualizzazione cambia (per esempio, ridimensionando la finestra del browser) le dimensioni espresse con questa unità di misura vengono scalate in proporzione.
vm	Relativa	Rispetto al valore più piccolo tra larghezza e altezza dell'area di visualizzazione.
ch	Relativa	Rispetto alla larghezza del carattere "0" (zero).
in	Assoluta	Misura in inch. (pollici); 1 inch = 2,54 mm.

cm	Assoluta	Misura in centimetri.
mm	Assoluta	Misura in millimetri.
pt	Assoluta	Punti; 1 pt = 1/72 inch.
pc	Assoluta	Picas; 1 pc = 12 pt.

Tabella 6.2 – Unità di misura di lunghezza CSS.

Osservando le unità di misura per le lunghezze disponibili presentate nella [tabella 6.2](#), notiamo che alcune sono *relative* (per esempio, a un valore precedentemente espresso o alle caratteristiche del dispositivo) mentre altre sono *assolute* (per esempio, se la dimensione viene espressa in centimetri).

Le unità di misura elencate nella [tabella 6.2](#) non valgono esclusivamente per l'impostazione della dimensione del font ma anche per tutte le larghezze e le altezze, come vedremo nel prosieguo di questo capitolo. Maggiori informazioni sulle unità di misura sono disponibili all'indirizzo: <http://aspit.co/a09>.

Impostazione del colore

Un altro aspetto importante per il controllo tipografico del testo è la scelta del colore, attraverso la proprietà `color`. Come per le unità di misura, anche per la definizione di una tonalità, CSS mette a disposizione diverse possibilità:

- Nome del colore, usando una delle costanti letterali come `white`, `black`, `blue`, `red`, `yellow`, `brown`, `purple` ecc.;
- Notazione esadecimale completa, nella forma `#RRGGBB` (rosso/verde/blu) dove, per esempio, il bianco è `#FFFFFF`, il nero `#000000`, il rosso `#FF0000`, il verde `#00FF00`, il blu `#0000FF`, il giallo `#FFFF00`, il viola `#FF00FF` ecc.
- Notazione esadecimale compatta, nella forma `#RGB`; rispetto alla notazione completa presuppone che le due cifre che determinano la “quantità” di ciascun colore siano uguali, per cui il bianco può essere espresso come `#FFF`, il nero come `#000`, il rosso come `#F00`, il verde come `#0F0`, il blu come `#00F`, il giallo come `#FF0`, il viola come `#F0F` ecc.

- RGB con i colori espressi in base 10 anziché 16; in questo caso bianco, nero, rosso, verde, blu, giallo e viola sono definiti rispettivamente con `rgb(255, 255, 255)`, `rgb(0, 0, 0)`, `rgb(255, 0, 0)`, `rgb(0, 255, 0)`, `rgb(0, 0, 255)`, `rgb(255, 255, 0)` e `rgb(255, 0, 255)`.
- RGBA, analogo a RGB ma con la gestione della trasparenza (alpha); il valore di alpha deve essere compreso tra 0 (completamente trasparente) e 1 (completamente opaco) quindi, per esempio, il nero con trasparenza al 50% è definito come `rgba(0, 0, 0, .5)`.
- HSL (hue, saturation, lightness) per definire il colore in modo “naturale”, in base a tonalità (da 0 a 360, corrispondente all’angolo della tinta desiderata nel cerchio dell’arcobaleno dei colori), saturazione e brillantezza (entrambe espresse in percentuale); per esempio `hsl(0, 100%, 50%)` esprime un colore rosso con luminosità massima e saturazione al 50%.
- HSLA, analogo a HSL ma con la gestione della trasparenza con la stessa modalità descritta per RGBA; in questo caso il colore rosso d’esempio può essere reso semi-trasparente indicando: `hsla(0, 100%, 50%, .5)`.

La gestione della trasparenza supportata dalle notazioni RGBA e HSLA ha effetto esclusivamente sul colore, a differenza dell’uso della proprietà `opacity` che consente di estendere l’effetto all’intero elemento, compresi gli eventuali figli in esso contenuti.

Le modalità per definire il colore descritte qui sopra vengono usate anche per indicare il colore di sfondo attraverso la proprietà `background-color`, come vedremo in seguito.

Applicare effetti al testo

Per completare il controllo della visualizzazione del testo, oltre alle proprietà base, come l’uso del grassetto, del corsivo, della sottolineatura e così via, presentate all’inizio del capitolo, e alla scelta di font, dimensione e colore, CSS mette a disposizione anche alcuni effetti aggiuntivi di varia natura.

Possiamo, per esempio, aggiungere un’ombreggiatura al testo, impostando la proprietà `text-shadow` e specificando nell’ordine:

- Scostamento orizzontale dell’ombreggiatura (horizontal offset, ovvero

sull'asse X); con valori positivi, l'ombra viene disegnata alla destra del testo mentre, con valori negativi, alla sua sinistra;

- Scostamento verticale dell'ombreggiatura (vertical offset, ovvero sull'asse Y); con valori positivi, l'ombra viene disegnata sotto il testo mentre, con valori negativi, sopra;
- Il raggio di sfumatura dell'ombreggiatura; più è grande il valore impostato più l'ombra risulterà “leggera” e sfocata; questo parametro è opzionale;
- Il colore dell'ombreggiatura.

L'esempio 6.4 mostra come impostare l'ombreggiatura a un titolo `h1`, usando la proprietà `text-shadow`.

Esempio 6.4

```
h1
{
  font-family: Arial, Helvetica, sans serif;
  font-size: 48pt;
  font-weight: bold;
  color: rgb(0, 0, 0);
  text-shadow: -5pt 2pt 4pt #999;
}
```

Nell'esempio 6.4, dopo aver impostato le caratteristiche principali del testo come font, dimensione, grassetto e colore (nero), viene applicata un'ombreggiatura grigia spostata in basso a sinistra; il risultato finale è mostrato nella figura 6.1.



Figura 6.1 – Testo con ombreggiatura visualizzato nel browser.

Possiamo anche impostare ombreggiature multiple, immettendo come valore per la proprietà `text-shadow` diverse impostazioni (scostamento, sfocatura e

colore) separate da virgola.

nota

Per maggiori informazioni sulla gestione del testo con CSS, è possibile consultare la pagina “CSS Text Level 3” all’indirizzo <http://aspit.co/alc>.

Dopo aver visto come impostare al meglio le caratteristiche del testo, passiamo alla gestione del layout, ovvero all’impostazione e al posizionamento dei singoli elementi che costituiscono la pagina.

Gestione del layout di pagina

Da un punto di vista tipografico una pagina web non è molto diversa da una pagina “fisica” come, per esempio, la prima pagina di un quotidiano o di una rivista: ci sono diversi elementi comuni come la testata, i titoli, gli articoli, le immagini e ciascuno di essi deve avere una propria caratterizzazione stilistica specifica e occupare una determinata posizione nell’impaginazione, per ottenere un aspetto complessivamente chiaro, fruibile ed esteticamente piacevole.

In questo senso alcune delle regole basilari restano valide indipendentemente dal “medium” (cartaceo o digitale) come, per esempio, un corretto uso del colore per garantire la leggibilità del contenuto anche a utenti con difficoltà visive come il daltonismo o, più semplicemente, la presentazione di un testo suddiviso in più colonne anziché essere visualizzato come unico blocco con righe lunghissime. La [figura 6.2](#) mostra in modo eloquente come aumenti la leggibilità di un testo separato in colonne.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed viverra laoreet turpis. Donec rhoncus turpis quis diam sodales fermentum. Duis eu sem dolor. Nulla facilisi. Integer luctus lobortis felis, ut fringilla enim congue quis. Curabitur eget tortor vel nisi suscipit sagittis ac non magna. Ut fermentum sagittis nunc, eget pharetra sapien commodo sit amet. Suspendisse pharetra aliquet felis, auctor commodo metus porta eget. Fusce euismod magna eu diam tincidunt facilisis. Sed nisi libero, molestie eget condimentum id, consectetur at risus. Proin id elit eu erat congue suscipit. Aliquam vitae massa eget lacus dignissim luctus. Ut vehicula, orci sit amet porttitor varius, lorem purus eleifend odio, in mattis nunc ipsum sit amet turpis. Ut eleifend eros egestas lectus varius malesuada. Aliquam et pulvinar ipsum. Donec molestie cursus lacus et fringilla. In hac habitasse platea dictumst. Proin adipiscing facilisis felis vitae porta. Nulla gravida egestas pretium. In tincidunt porttitor nibh id varius. Phasellus facilisis tempor ultrices. Cras sagittis dolor faucibus nibh lobortis non venenatis lacus ullamcorper. Quisque id erat ut felis condimentum dignissim. Nullam vel interdum lectus. Aliquam pellentesque nulla eget lacus pulvinar placerat. Fusce consequat, lacus vel fringilla tempor, lorem ligula tempor magna, quis ullamcorper nisl massa at ligula. Integer sit amet vestibulum ipsum. Sed hendrerit ante vel nulla sollicitudin nec eleifend felis posuere. Maecenas pretium lorem et tortor malesuada ultrices. Mauris dignissim tristique vehicula. Morbi ac quam sapien. In ultricies dui nec magna adipiscing aliquam. Pellentesque hendrerit feugiat mattis. Integer erat velit, porta a mollis ut, tempor posuere mauris. Aliquam sed neque enim. Curabitur feugiat ante non elit scelerisque laoreet.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed viverra laoreet turpis. Donec rhoncus turpis quis diam sodales fermentum. Duis eu sem dolor. Nulla facilisi. Integer luctus lobortis felis, ut fringilla enim congue quis. Curabitur eget tortor vel nisi suscipit sagittis ac non magna. Ut fermentum sagittis nunc, eget pharetra sapien commodo sit amet. Suspendisse pharetra aliquet felis, auctor commodo metus porta eget. Fusce euismod magna eu diam tincidunt facilisis. Sed nisi libero, molestie eget condimentum id, consectetur at risus. Proin id elit eu erat congue suscipit. Aliquam vitae massa

eget lacus dignissim luctus. Ut vehicula, orci sit amet porttitor varius, lorem purus eleifend odio, in mattis nunc ipsum sit amet turpis. Ut eleifend eros egestas lectus varius malesuada. Aliquam et pulvinar ipsum. Donec molestie cursus lacus et fringilla. In hac habitasse platea dictumst. Proin adipiscing facilisis felis vitae porta. Nulla gravida egestas pretium. In tincidunt porttitor nibh id varius. Phasellus facilisis tempor ultrices. Cras sagittis dolor faucibus nibh lobortis non venenatis lacus ullamcorper. Quisque id erat ut felis condimentum dignissim. Nullam vel interdum lectus. Aliquam pellentesque nulla eget

lacus pulvinar placerat. Fusce consequat, lacus vel fringilla tempor, lorem ligula tempor magna, quis ullamcorper nisl massa at ligula. Integer sit amet vestibulum ipsum. Sed hendrerit ante vel nulla sollicitudin nec eleifend felis posuere. Maecenas pretium lorem et tortor malesuada ultrices. Mauris dignissim tristique vehicula. Morbi ac quam sapien. In ultricies dui nec magna adipiscing aliquam. Pellentesque hendrerit feugiat mattis. Integer erat velit, porta a mollis ut, tempor posuere mauris. Aliquam sed neque enim. Curabitur feugiat ante non elit scelerisque laoreet.

Figura 6.2 – La suddivisione del testo in colonne ne aumenta la leggibilità.

Con CSS possiamo decidere di suddividere il testo in più colonne senza ricorrere all'uso di tabelle, ma semplicemente impostando opportunamente alcune proprietà per lo stile dell'elemento contenitore, come mostrato nell'esempio 6.5.

Esempio 6.5

```
div
{
  column-count: 3;
  column-gap: 20pt;
  column-rule: 1px solid #aaa;
}
```

Nell'esempio 6.5 viene definito un layout a tre colonne, con una spaziatura di 20 punti tra le colonne e con un bordo di separazione verticale di un pixel grigio.

Il risultato finale è mostrato nella parte inferiore della figura 6.2. Possiamo inoltre controllare le dimensioni delle colonne con le proprietà `column-width` e `height` oppure definire elementi (come, per esempio, un'intestazione) da disporre trasversalmente su più colonne con `column-span` (in questo caso, specificando come valore `all`, l'elemento occuperà tutte le colonne, indipendentemente dal loro numero).

Per maggiori informazioni sulla gestione dei layout a colonne, è possibile vedere la pagina “CSS Multi-column Layout Module” all’indirizzo <http://aspit.co/alid>.

L’uso dei layout multi-colonna risulta molto pratico ed elastico perché tutto il contenuto si adatta automaticamente alle impostazioni definite, compresi eventuali immagini o sotto-elementi, senza dover intervenire sul markup della pagina come sarebbe invece necessario fare se i contenuti fossero strutturati usando tabelle.

In passato le tabelle venivano usate per controllare l’allineamento e il posizionamento di ogni elemento ma, da diversi anni, questo approccio è stato soppiantato in favore dei cosiddetti layout table-less, ovvero basati sull’uso di contenitori (tipicamente dei tag `div`) posizionati e dimensionati da CSS.

Come abbiamo visto anche nei precedenti capitoli, le possibilità di controllo dell’impaginazione con CSS sono veramente notevoli. Oltre al controllo di dimensioni e posizionamento, possiamo definire anche una serie di comportamenti specifici. Per esempio, mediante la proprietà `text-overflow: ellipsis`; il browser è in grado di applicare automaticamente dei puntini di sospensione alla fine del testo che eccede le dimensioni fissate dal contenitore.

Sfondi e bordi

Parlando del colore del testo, abbiamo anticipato che la stessa notazione può essere usata, attraverso la proprietà `background-color`, per indicare il riempimento dello sfondo di un elemento, definendo anche un’eventuale trasparenza, usando il formato HSLA o RGBA.

Oltre a un colore uniforme, possiamo impostare anche sfumature (gradient), sia lineari sia circolari, evitando di fatto la necessità di ricorrere a immagini per la creazione degli effetti grafici più comuni. [L’esempio 6.6](#) mostra lo stile CSS da impostare per ottenere un effetto di riempimento con gradiente verticale su diverse tonalità di grigio, da #999 a #eee.

Esempio 6.6

```
div
{
    width: 500pt;
    height: 250pt;
```

```
border: 1px solid #000;
background-color: #aaa;
background: #ddd;
background-image: linear-gradient(left, #999, #eee);
}
```

Il risultato del codice [nell'esempio 6.6](#) è visibile nella [figura 6.3](#).



Figura 6.3 – Contenitore con riempimento dello sfondo sfumato.

Oltre a un riempimento sfumato come quello mostrato nella [figura 6.3](#), attraverso la proprietà `background-image` è possibile impostare un'immagine come sfondo dell'elemento. Possiamo controllare la posizione (verticale e orizzontale, rispetto all'angolo superiore sinistro dell'elemento) in cui verrà visualizzata l'immagine con la proprietà `background-position` e se dovrà essere replicata come texture in orizzontale e/o in verticale con `background-repeat`.

Le specifiche CSS supportano background multipli, cioè la possibilità di specificare più di un'immagine di sfondo posizionando ciascuna risorsa su un livello diverso (layer). [L'esempio 6.7](#) presenta un esempio di background multipli per la costruzione di una scena complessa, usando singole immagini (un aeroplano, una nuvola e un sole) opportunamente posizionate come sfondo di un contenitore.

Esempio 6.7

```
div
{
```

```
width: 500pt;
height: 250pt;
border: 1px solid #000;
background:
    url(airplane.png) 230pt 100pt no-repeat,
    url(cloud.png) 280pt 65pt no-repeat,
    url(cloud.png) 330pt 25pt no-repeat,
    url(cloud.png) 80pt 25pt no-repeat,
    url(sun.png) 20pt 15pt no-repeat,
    #0667b5;
}
```

Nell'esempio 6.7 le immagini si sovrappongono parzialmente, per cui l'ordine di dichiarazione degli sfondi risulta fondamentale: al livello più alto (in primo piano) troviamo l'aeroplano, in quanto è la prima immagine dichiarata, e poi via via tutti gli altri. Il risultato finale è visibile nella [figura 6.4](#).

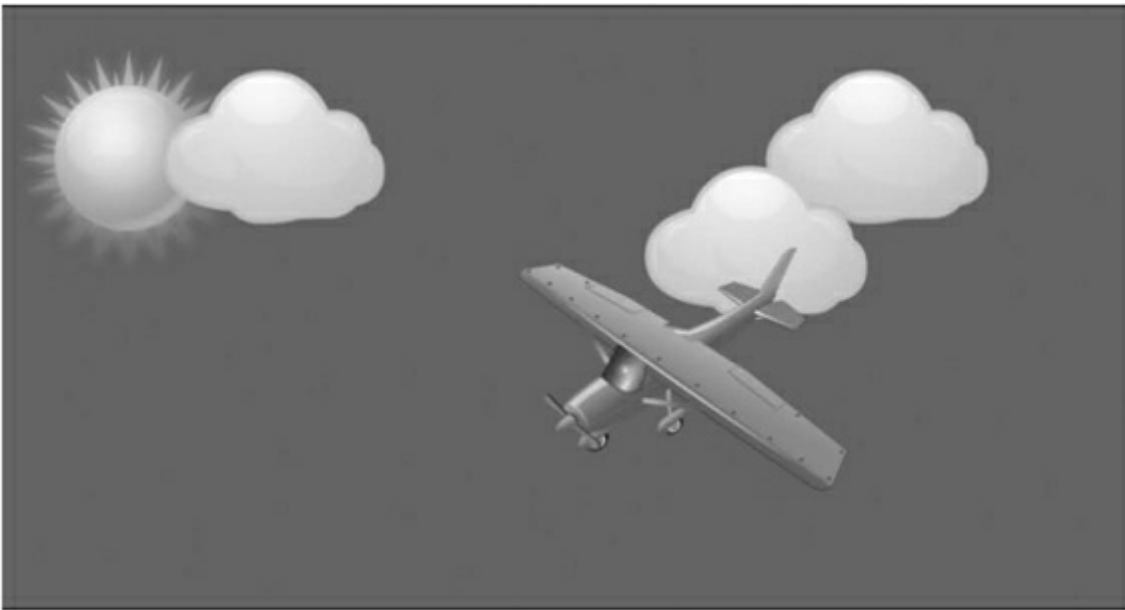
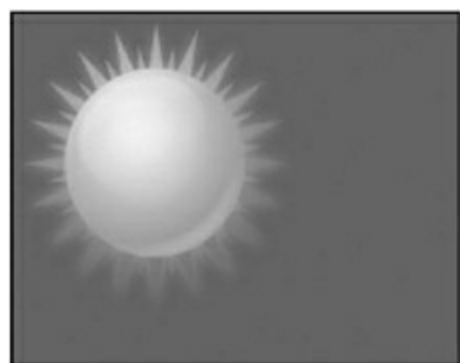


Figura 6.4 – Background multipli.

Ogni immagine di sfondo può inoltre essere ridimensionata usando la proprietà `background-size`; per indicare le dimensioni delle singole immagini, dobbiamo semplicemente inserire le coppie di valori (larghezza e altezza) separate da virgola, rispettando l'ordine usato nella definizione della proprietà `background-image`. Impostando `background-size: cover`; l'immagine verrà ridimensionata (mantenendo inalterate le proporzioni), in modo che il valore più *piccolo* tra larghezza e altezza sia pari alla dimensione del contenitore. Con `background-size: contain`; l'immagine verrà invece

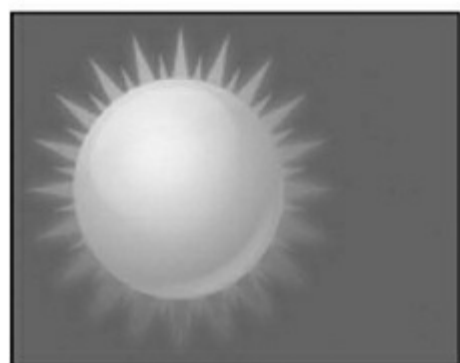
ridimensionata (sempre rispettandone le proporzioni), in modo che il valore più *grande* tra larghezza e altezza sia pari alla dimensione del contenitore. La [figura 6.5](#) mostra alcuni esempi di ridimensionamento dell'immagine di sfondo: il contenitore ha dimensioni fissate a 200x150 e l'immagine del sole originale è di 128x126.



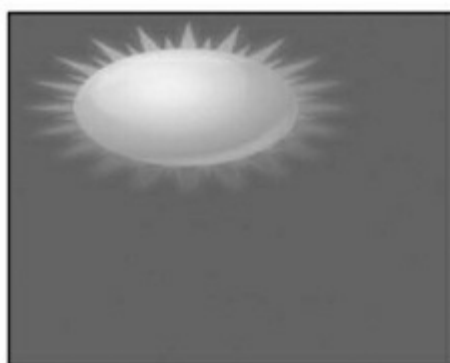
(default)



`background-size: cover;`



`background-size: contain;`



`background-size: 128px 80px;`

Figura 6.5 – Opzioni di ridimensionamento dello sfondo.

Attraverso le proprietà `background-clip` e `background-origin` possiamo anche definire come lo sfondo e le immagini di sfondo verranno posizionate rispetto ai bordi (`border-box`), ai margini interni (`padding-box`) o al contenuto (`content-box`) dell'elemento.

Per completare la personalizzazione dell'aspetto di un elemento, possiamo aggiungere un bordo. Come abbiamo visto nel capitolo precedente, è possibile impostare le caratteristiche di ogni bordo (superiore, inferiore, destro e sinistro) singolarmente, decidendo lo spessore, il colore, lo stile della linea e gli eventuali arrotondamenti agli angoli.

Oltre a queste caratteristiche, importanti per capire il box model, possiamo impostare immagini come bordo. Impostando la proprietà `border-image` possiamo definire l'immagine da utilizzare come “pattern” per il riempimento

dei bordi e degli angoli, indicare la porzione che dovrà essere usata per ogni bordo e decidere se l'immagine, per completare il riempimento del bordo, dovrà essere ripetuta (`round`) oppure se dovrà essere “stirata” (`stretch`). L'esempio 6.8 mostra due possibili implementazioni per utilizzare un'immagine come bordo.

Esempio 6.8

```
div.demo1
{
  width: 300px;
  height: 120px;
  border: 20px;
  border-image: url(border.png) 20 stretch round;
}

div.demo2
{
  width: 300px;
  height: 120px;
  border: 20px;
  border-image: url(border.png) 20 round round;
}
```

Il risultato finale è mostrato nella [figura 6.6](#): al centro è evidenziata l'immagine usata come sorgente (`border.png`), studiata appositamente per mettere in evidenza come il foglio di stile userà le singole porzioni (in questo caso creando quadrati di 20px per ogni lato) per il riempimento dei bordi e degli angoli.

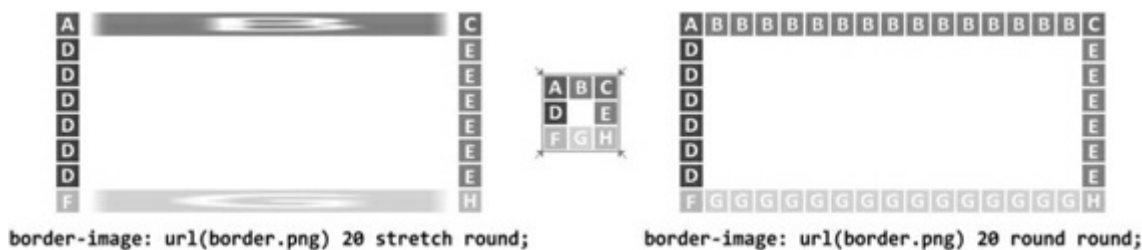


Figura 6.6 – Il bordo viene creato usando un'immagine.

A sinistra della [figura 6.6](#) possiamo vedere come si presenta il `div` con classe “demo1”, dove la porzione di immagine usata per i bordi orizzontali (superiore e inferiore) viene allungata (`stretch`). A destra, invece, troviamo la

rappresentazione del `div` con classe “demo2” che replica l’immagine per il riempimento di tutti i bordi (`round`).

nota

Per maggiori informazioni sulla gestione di bordi e sfondi con CSS, è possibile consultare la pagina all’indirizzo <http://aspit.co/ale>.

Possiamo infine aggiungere un ulteriore tocco di stile, applicando un’ombreggiatura al contenitore, in modo del tutto analogo a quanto visto per il testo. In questo caso, anziché `text-shadow`, dobbiamo usare la proprietà `box-shadow`, che prevede i seguenti valori:

- Scostamento orizzontale dell’ombreggiatura (horizontal offset, ovvero sull’asse X); con valori positivi l’ombra viene disegnata alla destra dell’elemento mentre con valori negativi alla sua sinistra;
- Scostamento verticale dell’ombreggiatura (vertical offset, ovvero sull’asse Y); con valori positivi l’ombra viene disegnata sotto l’elemento mentre con valori negativi sopra;
- Il raggio di sfumatura dell’ombreggiatura: più è grande il valore impostato più l’ombra risulterà “leggera” e sfocata; questo parametro è opzionale;
- Espansione dell’ombra per estenderla in tutte le direzioni, in base al valore specificato; con valori negativi l’ombra viene contratta;
- Il colore dell’ombreggiatura;
- `inset` per indicare che l’ombreggiatura avverrà all’interno dell’elemento anziché, quando omesso, all’esterno.

L’esempio 6.9 mostra come impostare l’ombreggiatura a un `div`, usando la proprietà `box-shadow`.

Esempio 6.9

```
div
{
    width: 500px;
```



```
height: 250px;
border: 2px solid #222;
background: #ddd;
box-shadow: 5px 5px 10px 7px #555;
}
```

Possiamo vedere il risultato finale [dell'esempio 6.9](#) nella [figura 6.7](#).



Figura 6.7 – Ombreggiatura di un elemento.

Nella [figura 6.7](#) possiamo notare che l'ombreggiatura del `div` si espande (di 7 pixel), risultando parzialmente visibile anche sopra e a sinistra.

Combinando opportunamente le diverse proprietà CSS, possiamo ottenere effetti interessanti per personalizzare in modo semplice l'aspetto delle nostre pagine senza dover intervenire sul markup esistente, per esempio per creare delle sovrastrutture. [L'esempio 6.10](#) mostra l'ipotetico sorgente HTML per la presentazione di un articolo, con un `header` contenente il titolo, una nota e un paragrafo di testo (il cui contenuto è stato troncato per brevità).

Esempio 6.10

```
<article>
  <header>
    <h1>HTML5 Espresso</h1>
    <p>di: Boichicchio, Mostarda</p>
  </header>
  <p>Lorem ipsum dolor sit amet, consectetur [...]</p>
</article>
```

Applicando le tecniche di controllo dell'impaginazione e di formattazione del testo, partendo dal markup [dell'esempio 6.10](#), possiamo facilmente ottenere un layout grafico come quello mostrato nella [figura 6.8](#).

Per ottenere il risultato visualizzato nella [figura 6.8](#), sono state usate molte delle tecniche e delle proprietà CSS che abbiamo visto nel corso di questo capitolo: uso di font personalizzati (per il titolo, mostrato con un carattere che ricorda la scrittura “a mano”), ombreggiatura del testo e del contenitore, impaginazione del contenuto in colonne, sfondi con gradiente di sfumatura, bordi arrotondati ecc. Si tratta di un esempio molto semplice, realizzato con un foglio stile di poche righe, ma comunque in grado di farci apprezzare le potenzialità di CSS per il controllo dell'aspetto della pagina.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed viverra laoreet turpis. Donec rhoncus turpis quis diam sodales fermentum. Duis eu sem dolor. Nulla facilisi. Integer luctus lobortis felis, ut fringilla enim congue quis. Curabitur eget tortor vel nisi suscipit sagittis ac non magna. Ut fermentum sagittis nunc, eget pharetra sapien commodo sit amet. Suspendisse pharetra aliquet felis, auctor commodo metus porta eget. Fusce euismod magna eu diam tincidunt facilisis. Proin id elit

eu erat congue suscipit. Aliquam vitae massa eget lacus dignissim luctus. Ut vehicula, orci sit amet porttitor varius, lorem purus eleifend odio, in mattis nunc ipsum sit amet turpis. Ut eleifend eros egestas lectus varius malesuada. Aliquam et pulvinar ipsum. Donec molestie cursus lacus et fringilla. In hac habitasse platea dictumst. Proin adipiscing facilisis felis vitae porta. Nulla gravida egestas pretium. In tincidunt porttitor nibh id varius.

Figura 6.8 – Layout tipografico con CSS3.

Conclusioni

Come abbiamo più volte ribadito nei precedenti capitoli, HTML consente di mantenere una netta separazione tra quello che è il contenuto vero e proprio (per esempio, il titolo o il corpo di un articolo), gestito tramite markup anche da un punto di vista semantico, e quella che è la sua rappresentazione visuale (che posizione occupa nella pagina, con che dimensioni, con quale carattere o

colore deve essere visualizzato ecc.), definita dallo stile CSS associato.

In questo capitolo abbiamo visto come è possibile utilizzare CSS per impostare le principali caratteristiche di visualizzazione dei testi e degli elementi della pagina, così da controllarne il layout in modo tipografico, come faremmo nell'impaginazione di un giornale o di una rivista.

Dopo aver trattato il markup, la grafica bitmap e vettoriale e introdotto CSS, è tempo di aggiungere dinamismo alle nostre pagine. I prossimi capitoli riguardano infatti Java-Script, il linguaggio di programmazione che consente di interagire con la pagina e con i suoi elementi, fino a creare vere e proprie applicazioni.

Introduzione a JavaScript

Dopo aver visto come sfruttare HTML e CSS per creare una pagina, aggiungere contenuto e renderla gradevole dal punto di vista visuale, cambiamo argomento e cominciamo a vedere come permettere all'utente di interagire con la nostra pagina.

Il linguaggio HTML definisce staticamente il contenuto di una pagina. Anche i CSS definiscono staticamente l'aspetto di una pagina, fatta eccezione per alcune funzionalità come transizioni o pseudo-elementi che se da un lato rendono la pagina animata e non statica, dall'altro applicano solo trasformazioni grafiche.

In questo capitolo cominciamo a vedere la parte dinamica del web: JavaScript. Questo linguaggio è quello che permette a una pagina di interagire con l'input dell'utente. Un tipico esempio di utilizzo del JavaScript è nella validazione delle form; quando le regole di validazione offerte nativamente da HTML non sono sufficienti (per esempio perché vogliamo validare un codice fiscale o una partita IVA), dobbiamo ricorrere a codice JavaScript per validare i dati e mostrare un feedback all'utente in caso di errore.

Il JavaScript è un linguaggio di scripting, dinamico e non è fortemente tipizzato. Queste tre caratteristiche lo rendono un linguaggio dal duplice aspetto; da un lato la dinamicità offre un'enorme versatilità, ma dall'altro la mancanza di tipizzazione rende lo sviluppo più lento, complesso e soggetto a errori. Questi aspetti verranno approfonditi nel corso del capitolo man mano che acquisiremo conoscenze su questo linguaggio.

Passiamo ora a parlare del JavaScript partendo dal classico Hello World.

Hello World

Come le prime quattro lettere di JavaScript lasciano intuire, questo linguaggio prende spunto dal Java. Tuttavia, i due linguaggi condividono solo alcune parti della sintassi come le istruzioni condizionali, i cicli iterativi e poco altro.

Per iniziare a parlare di JavaScript, facciamo un semplice esempio allo scopo di fissare i concetti di base.

Esempio 7.1

```
<script>
function HelloWorld(parameter) {
    var second, message;
    var currentDate = new Date();
    second = currentDate.getSeconds();
    if (second % 2 === 0) {
        message = "Numero pari";
    }
    else {
        message = "Numero dispari";
    }
    alert(message);
}
</script>
```

L'esempio 7.1 mostra una semplice funzione che esegue i seguenti passi:

- crea una funzione di nome `HelloWorld` che accetta un parametro di nome `parameter`;
- crea due variabili (`message` e `second`) senza assegnare né un tipo né un valore;
- crea una variabile (`currentDate`) e ne imposta il valore con un oggetto di tipo `Date` (che se istanziato senza parametri contiene la data corrente);
- invoca il metodo `getSeconds` sull'oggetto `currentDate` di tipo `Date` per recuperare i secondi;
- suddivide i secondi per due e recupera il resto. Se il resto è zero, imposta la variabile `message` con "Numero pari" altrimenti imposta la stessa variabile con "Numero dispari";
- mostra una finestra all'utente con il contenuto della variabile `message` tramite la funzione di sistema `alert`.

In questo piccolo esempio di codice abbiamo visto molte delle caratteristiche di base del JavaScript come la dichiarazione di una variabile, l'assegnazione di

un valore, l'invocazione di un metodo, il confronto tra valori e l'interazione con l'utente tramite una finestra di dialogo. Il tutto racchiuso tra tag `<script>`
`</script>` per dire al browser che si tratta di codice JavaScript. Oltre che a scrivere codice JavaScript nella pagina, possiamo anche creare un file esterno contenente codice JavaScript e importarlo nella pagina usando il tag `script` e impostandone la proprietà `src` all'indirizzo del file. In questo caso il codice nel file non deve essere racchiuso tra i tag `<script>`. Passiamo ora a vedere queste funzionalità in maggior dettaglio cominciando dai tipi di base.

Tipi di base

Come abbiamo detto in precedenza, il JavaScript è un linguaggio di scripting dinamico e non tipizzato. Questo significa che non ci sono controlli sui tipi (vedremo più avanti cosa questo significhi), ma non che i tipi non esistano. Tra i tipi di base del JavaScript più usati ci sono quelli inclusi nella [tabella 7.1](#).

<i>Tipo</i>	<i>Descrizione</i>
Object	Rappresenta il tipo di base di tutti gli oggetti.
Date	Rappresenta una data.
Boolean	Rappresenta un valore booleano.
Number	Rappresenta un numero.
String	Rappresenta una stringa.
Array	Rappresenta una lista di oggetti.
Error	Rappresenta un'eccezione.
Undefined	Rappresenta un valore che non esiste.

Tabella 7.1 – Tipi principali in JavaScript.

I tipi che vanno da `Object` a `Error` nella [tabella 7.1](#) sono comuni a molti altri linguaggi di programmazione e quindi non necessitano di ulteriori spiegazioni. Il tipo `Undefined`, invece, necessita di una spiegazione poiché è una peculiarità del JavaScript che approfondiamo nella prossima sezione.

nota

JavaScript non offre funzioni per convertire il valore di un tipo nello stesso valore di un altro tipo (per esempio, convertire il numero 5 in una stringa che ha come valore “5”), eccetto per i casi in cui si vuole convertire una stringa in un numero. Per questi casi esistono le funzioni `parseInt` e `parseFloat` che accettano appunto una stringa e restituiscono il valore numerico. La prima funzione restituisce un valore intero (eventuali decimali vengono ignorati), la seconda restituisce un valore decimale.

Passiamo ora a vedere come definire le variabili che contengono i tipi che abbiamo appena elencato.

Dichiarazione di variabile

Per definire una variabile, dobbiamo usare la parola chiave `var` seguita dal nome della variabile, come abbiamo già mostrato [nell'esempio 7.1](#). Se dobbiamo dichiarare più variabili nella stessa istruzione, possiamo farlo separando i nomi delle variabili con un carattere “,” (virgola), come abbiamo visto per le variabili `message` e `second` [nell'esempio 7.1](#).

nota

Una volta dichiarata, il valore della variabile è `Undefined`. `Undefined` è diverso da `null`: il primo identifica che la variabile non ha un valore, in quanto abbiamo dichiarato la variabile senza impostarlo, mentre il secondo è un valore vero e proprio.

Possiamo assegnare il valore a una variabile in fase di dichiarazione, aggiungendo un carattere “=” (uguale) dopo il nome della variabile seguito dal valore, come abbiamo visto [nell'esempio 7.1](#) per la variabile `currentDate`.

Quando assegniamo il valore a una variabile, il tipo della variabile viene impostato al tipo del valore assegnato. Tuttavia, data la natura dinamica di JavaScript, nulla ci vieta successivamente di assegnare un valore di un altro tipo alla stessa variabile, come [nell'esempio 7.2](#).

Esempio 7.2

```
var myValue = "test":  
myValue = 5;  
alert(myValue + 10); //mostra 15
```

La variabile `myValue` inizialmente è di tipo `String`, ma dopo che le viene assegnato il valore cinque, diventa di tipo `Number`, motivo per cui successivamente nella finestra di dialogo viene mostrato il valore 15.

Se da un lato questo comportamento ci offre una certa flessibilità nello sviluppo, dall'altro può facilmente causare errori specialmente quando si ha a che fare con notevoli quantità di codice JavaScript. Non solo, proprio per la mancanza di tipizzazione certa di una variabile, gli strumenti di sviluppo JavaScript non sono in grado di fornire un'esperienza di sviluppo completa.

Ora cambiamo argomento e vediamo quali sono gli operatori e come usarli in combinazione con variabili, costanti o altro ancora.

Operatori

Un operatore è un'istruzione che esegue una determinata operazione tra due operandi. Gli operatori sono suddivisibili in sei categorie: aritmetici, assegnazione, logici, comparazione, ternari e bitwise. Nella [tabella 7.2](#) possiamo vedere tutti gli operatori a disposizione.

Operatore	Descrizione	Categoria
<code>+</code> <code>-</code> <code>*</code> <code>/</code> <code>%</code>	Addizione, sottrazione, moltiplicazione, divisione, modulo (calcola il resto di una divisione).	Aritmetici
<code>++</code> <code>--</code>	Incrementa di uno, decrementa di uno	Aritmetici
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>	Assegnazione, somma e assegnazione, sottrazione e assegnazione, moltiplicazione e assegnazione, divisione e assegnazione, modulo e assegnazione.	Assegnazione
<code>+</code> <code>+=</code>	Se usati su stringhe le concatenano nel primo caso e le concatenano e assegnano nel secondo caso.	Assegnazione

== === != !==	Uguaglianza del valore, uguaglianza di tipo e valore, disuguaglianza del valore, disuguaglianza di tipo e valore.	Comparazione
> >= < <=	Maggiore di, maggiore o uguale a, minore di, minore o uguale a.	Comparazione
&& !	And, Or, Not.	Logici
?:	Ternario.	Ternari
& ~ ^ << >>	And, Or, Not, XOR, Left shift, Right shift.	Bitwise

Tabella 7.2 – Operatori JavaScript.

Come sempre, un esempio aiuta a capire meglio, quindi nel prossimo vediamo i principali operatori in azione.

Esempio 7.3

```
//operatori aritmetici
var x = 5, y = 2;
var z = x + y;           //z = 7
var z = x - y;           //z = 3
var z = x * y;           //z = 10
var z = x / y;           //z = 2.5
var z = x % y;           //z = 1
x++;                     //x = 6
x--;                     //x = 4

//operatori di assegnazione
x += y;                  //equivale a x = x + y; x = 7
x -= y;                  //equivale a x = x - y; x = 3

var s1 = "hello", s2 = "world";
var s3 = s1 + " " + s2;  //s3 = "hello world";
s1 += " " + s2;          //s1 = "hello world";
var s3 = s1 + x;         //s3 = "hello5";

//operatori di comparazione
var sx = "5";
var z = x == sx;         //z = true
var z = x === sx;        //z = false
var z = x != sx;         //z = false
```

```

var z = x !== sx;                //z = true

var z = x > y;                    //z = true
var z = x >= y;                   //z = true
var z = x < y;                    //z = false
var z = x <= y;                   //z = false

//operatori ternari
var z = x == 5 ? "uguale" : "diverso" //z = "diverso"

//operatori logici
var z = x == 5 && y > 0;           //return true
var z = x == 5 && y > 10;          //return false
var z = x == 5 || y > 10;         //return true
var z = x == 2 || y > 10;         //return false

```

Gli operatori aritmetici sono estremamente semplici in quanto eseguono un'operazione aritmetica tra due numeri. Gli operatori di assegnazione prendono il valore a destra dell'espressione e lo assegnano alla variabile di sinistra. In questa categoria torna spesso comoda la forma abbreviata di assegnazione “+=” quando vogliamo che il risultato dell'operazione sia assegnato non a una nuova variabile, ma a una di quelle coinvolte nell'operazione.

Gli operatori di comparazione necessitano di un approfondimento perché spesso generano confusione. Gli operatori “==” e “!=” eseguono una comparazione basata sul valore degli operandi. [Nell'esempio 7.3](#) mettiamo a confronto le variabili `x` e `sx` che hanno entrambe valore cinque, ma la prima è di tipo `Number` mentre la seconda è di tipo `String`. Nonostante i tipi siano diversi, gli operatori menzionati in precedenza ritornano `true` e `false`. Gli operatori “===” e “!==” eseguono una comparazione basata sul tipo e sul valore degli operandi. Per questo motivo, quando [nell'esempio 7.3](#) usiamo “===” per comparare le variabili `x` e `sx`, anche se il valore è lo stesso, il tipo delle variabili è diverso, quindi otteniamo `false`.

L'operatore ternario è molto comodo quando dobbiamo assegnare un valore in base a una condizione. [Nell'esempio 7.3](#) usiamo un operatore di comparazione per verificare che `x` equivalga a cinque. Se la comparazione torna `true`, viene restituito il valore dopo il carattere “?” (punto interrogativo), altrimenti viene restituito il valore dopo il carattere “:” (due punti).

Gli operatori logici servono per concatenare più comparazioni mentre gli operatori di bitwise sono poco usati, quindi non li tratteremo nel capitolo.

Passiamo ora a vedere come usare gli operatori condizionali per poter

gestire il flusso di esecuzione del codice.

Istruzioni di selezione

Le istruzioni di selezione servono per determinare l'esecuzione di un blocco di istruzioni in funzione di un'espressione booleana restituita da un operatore condizionale. Le istruzioni di selezione in JavaScript sono due: `if` e `switch`.

Istruzione `if`

L'istruzione `if` accetta un'espressione che ritorna un booleano e specifica il blocco di codice da eseguire nel caso l'espressione torni `true`. Opzionalmente, l'istruzione `if` può specificare anche il codice da eseguire nel caso il valore restituito dall'espressione sia `false`. [L'esempio 7.4](#) mostra alcuni esempi di utilizzo dell'istruzione `if`.

Esempio 7.4

```
var year = 2014;

if (year == 2014) {
    //codice eseguito nel caso la condizione sia vera
}

if (year == 2014) {
    //codice eseguito nel caso la condizione sia vera
}
else{
    //codice eseguito nel caso la condizione sia falsa
}

if (year == 2014) {
    //codice eseguito nel caso la condizione sia vera
}
else if (year == 2015) {
    //codice eseguito nel caso la condizione sia vera
}
else{
    //codice eseguito nel caso le precedenti condizioni siano false
}
```

Come si vede dal codice, l'operatore di comparazione deve essere racchiuso tra parentesi tonde e i blocchi da eseguire devono iniziare con l'apertura di un

carattere “{” e finire con un carattere “}”.

nota

Se un blocco è composto da una sola istruzione, le parentesi graffe possono essere omesse.

Nella terza istruzione `if` dell'esempio viene mostrato come applicare più istruzioni `if` una di seguito all'altra. In questi casi può risultare più leggibile l'istruzione `switch`.

Istruzione `switch`

L'istruzione `switch` permette di valutare il valore di una variabile e, in funzione di questo valore, eseguire un blocco di codice. Il blocco `switch` contiene a sua volta una serie di blocchi contrassegnati con la parola chiave `case`, seguita da un carattere “.” (due punti) e da un valore costante dello stesso tipo della variabile in esame. Un blocco viene eseguito in modo esclusivo rispetto agli altri fino alla prima istruzione di salto (`break`) se il valore della variabile combacia con quello della costante. La parola chiave `default` identifica il blocco che viene eseguito se nessun valore costante corrisponde al valore della variabile. Questo blocco va posto sempre per ultimo e non è obbligatorio.

L'esempio 7.5 mostra l'istruzione `switch` in azione.

Esempio 7.5

```
var periodo;
```

```
switch (new Date().getDay()) {  
    case 0:  
    case 6:  
        periodo = "Weekend";  
        break;  
    case 5:  
        periodo = "Venerdì";  
        break;  
    default:  
        periodo = "Altro";  
}
```

Ora che abbiamo visto come modificare l'esecuzione del codice in base a determinate condizioni, passiamo a vedere come eseguire più volte lo stesso codice.

Istruzioni di iterazione

Le istruzioni di iterazione servono per eseguire un blocco di istruzioni ripetutamente, in modo ciclico. In JavaScript esistono diverse istruzioni per eseguire iterazioni nel codice, ciascuna caratterizzata da una diversa modalità di uscita dal ciclo: `for`, `while` e `do ... while`. Vediamole in dettaglio, partendo dalla prima.

Istruzione `for`

L'istruzione `for` permette di definire un ciclo da ripetere tante volte finché un indice soddisfa una determinata condizione. Questa istruzione si compone di tre parti racchiuse in una coppia di parentesi tonde: l'inizializzazione dell'indice con l'eventuale dichiarazione, la condizione booleana di esecuzione del ciclo (generalmente basata sul controllo del valore dell'indice rispetto al suo valore massimo o minimo) e l'istruzione di incremento o decremento dell'indice. Nonostante le tre parti dell'istruzione `for` non siano obbligatorie, è vivamente consigliato mantenerle sempre. Anche in questo caso, qualora il ciclo sia composto da una sola istruzione, possiamo omettere le parentesi graffe che delimitano il blocco delle istruzioni.

Esempio 7.6

```
var result = "";
for (var i = 0; i < 5; i++) {
    result += "numero" + i;
}
```

In questo esempio, viene dichiarata e impostata la variabile `i` che agisce come indice, poi viene specificata la condizione di esecuzione del ciclo (l'indice deve essere minore di cinque), e la modalità di incremento dell'indice. Il risultato è che il codice tra le parentesi graffe viene eseguito cinque volte.

Istruzione `while`

L'istruzione `while` permette di definire un blocco di codice che deve essere eseguito in ciclo fintanto che una determinata condizione è soddisfatta. L'esempio 7.7 mostra come utilizzare questo ciclo.

Esempio 7.7

```
var result = "";
var i = 0
while (i < 5) {
    result += "numero" + i;
    i++;
}
```

Il risultato dell'esempio 7.7 è lo stesso del risultato 7.6. Come si vede, la quantità di codice è simile, quindi la scelta tra una tecnica e l'altra è una mera questione di gusti.

Istruzione `do ... while`

L'istruzione `do ... while` è simile all'istruzione `while` con la differenza che il blocco da eseguire in ciclo viene eseguito una volta prima di valutare l'espressione di entrata. L'esempio 7.8 mostra come utilizzare questo ciclo.

Esempio 7.8

```
var result = "";
var i = 0
do {
    result += "numero" + i;
    i++;
}
while (i < 0);
```

Il codice nel ciclo `do ... while` viene eseguito una sola volta in quanto quando si arriva alla valutazione dell'espressione di entrata, questa torna `false` e quindi il ciclo si interrompe.

Istruzioni di salto

A volte capita di dover interrompere un ciclo prematuramente in base a determinate condizioni. In questo caso dobbiamo usare l'istruzione `break` che arresta immediatamente l'esecuzione del ciclo trasferendo il controllo alla

prima istruzione successiva al ciclo stesso. [L'esempio 7.9](#) mostra come usare l'istruzione `break`.

Esempio 7.9

```
var condizione = true;
var result = "";
for (var i = 0; i < 5; i++) {
    result += "numero" + i;
    if (condizione)
        break;
}
```

L'istruzione `break` ci permette di uscire dal ciclo ma a volte dobbiamo semplicemente saltare una determinata iterazione e passare immediatamente alla successiva. In questi casi dobbiamo usare l'istruzione `continue`. Questa istruzione arresta il flusso di esecuzione del ciclo, riportandolo alla valutazione dell'espressione. [L'esempio 7.10](#) mostra come utilizzare questa istruzione.

Esempio 7.10

```
var result = "";
for (var i = 0; i < 5; i++) {
    if (i == 3)
        continue;

    result += "numero" + i;
}
```

In questo esempio, quando la variabile `i` ha valore tre, viene invocata l'istruzione `continue` e il codice rimanente del ciclo non viene eseguito. Il controllo viene rimandato al ciclo `for` quindi la variabile `i` viene incrementata di uno, viene rivalutata la condizione di entrata e il codice prosegue il suo normale flusso.

Tutte le istruzioni viste finora servono a decidere il flusso del codice. Ora cambiamo argomento e vediamo come creare funzioni per rendere il nostro codice più ordinato e leggibile.

Funzioni e procedure

In ogni linguaggio di programmazione, funzioni e procedure sono routine

(dette anche metodi) che possono accettare parametri e che rispettivamente ritornano o non ritornano un valore. Supponiamo di voler calcolare l’IVA di un acquisto. In questo caso possiamo creare una funzione che esegue il calcolo aritmetico e restituisce l’importo. Se invece vogliamo semplicemente mostrare l’IVA a video, possiamo scrivere una procedura che calcola l’IVA e poi la mostra senza restituire alcun valore al chiamante.

In JavaScript non esiste una distinzione sintattica tra procedura e funzione; se un metodo torna un valore è una funzione, se non torna alcun valore è una procedura. Vediamo [nell’esempio 7.11](#) come creare sia una funzione sia una procedura.

Esempio 7.11

```
//funzione
function CalcolaIVA(imponibile, percentuale){
    var iva = imponibile % percentuale / 100;
    return iva;
}

//Procedura
function MostraIVA(imponibile, percentuale){
    var iva = CalcolaIVA(imponibile, percentuale);
    alert(Iva);
}

//invoca metodo Calcola IVA
var iva = CalcolaIVA(100, 20);
```

Come si può notare nell’esempio, per dichiarare un metodo dobbiamo usare la parola chiave `function` seguita dal nome del metodo e dalla lista dei parametri racchiusi tra parentesi tonde e separati da virgola. Il tipo dei parametri non è ovviamente specificato data la dinamicità del JavaScript.

Il metodo `CalcolaIVA` è una funzione in quanto usa la parola chiave `return` per restituire un valore. Nel momento in cui l’istruzione `return` viene eseguita, il flusso di esecuzione abbandona il metodo e torna al chiamante, quindi qualunque codice sia scritto successivamente a questa istruzione all’interno della funzione non viene eseguito. Il metodo `MostraIVA` non contiene l’istruzione `return` quindi si tratta di una procedura.

nota

Se l'istruzione `return` non è seguita da un valore, l'esecuzione del metodo viene comunque arrestata, ma nessun valore viene restituito al chiamante, quindi il metodo è classificato come procedura.

Come mostrato nell'ultima riga [dell'esempio 7.11](#), per invocare un metodo basta specificarne il nome e passare i parametri tra parentesi tonde. Se passiamo meno parametri di quelli richiesti, il metodo viene comunque invocato e i parametri per i quali non è stato fornito un valore vengono impostati a `Undefined`.

Finora abbiamo parlato di come usare i tipi di base e di come organizzare il flusso del codice. Per completare le basi di JavaScript parliamo ora di come lavorare con i tipi complessi partendo dagli array.

Array

In JavaScript un array è un contenitore di oggetti. Non ci sono restrizioni al tipo di oggetti che un array può contenere tanto che all'interno dello stesso array possono esserci anche oggetti di diverso tipo.

Dichiarare una variabile di tipo `Array` non è diverso dal dichiarare una variabile di qualunque altro tipo; la differenza sta nell'inizializzazione che avviene semplicemente impostando come valore i caratteri “`[]`” (parentesi quadre). Possiamo anche inizializzare direttamente l'array con dei valori, inserendoli tra le parentesi quadre separati da virgola, così come mostrato nel prossimo esempio.

Esempio 7.12

```
var list = []; //inizializzazione di un array vuoto
```

```
var list = [1, 2, "stringa", new Date()]; //inizializzazione di un  
array con valori
```

Per aggiungere un elemento a un array dobbiamo usare il metodo `push` passando in input l'oggetto che vogliamo aggiungere.

Se invece vogliamo eliminare un elemento da un array dobbiamo usare il metodo `splice`. Questo metodo accetta come primo parametro l'indice dell'elemento da eliminare e come secondo il numero di elementi da eliminare partendo dall'indice passato come primo parametro.

nota

In JavaScript gli array partono da base zero. Questo significa che il primo elemento ha indice zero.

Se vogliamo invece modificare un elemento esistente, dobbiamo prima recuperarlo tramite `indexer` (passandogli l'indice dell'elemento tra parentesi quadre) e successivamente assegnargli un nuovo valore. Queste tecniche sono mostrate nel prossimo esempio.

Esempio 7.13

```
var list = [1, 2, 3, 4];

list.push(5); //aggiunge un elemento all'array
list.splice(4, 1); //rimuove l'elemento all'indice 4 (quindi in
posizione 5)
list[0] = 10; // recupera il primo elemento tramite indexer e ne
imposta il valore a 10
```

Oltre ai metodi appena visti, abbiamo a disposizione la proprietà `length` che restituisce il numero di elementi contenuti dall'array. Questa proprietà torna utile quando dobbiamo iterare tutti gli elementi dell'array, come [nell'esempio 7.14](#).

Esempio 7.14

```
var list = [1, 2, 3, 4];
var result = 0;
for (var i=0; i<list.length; i++){
    result += list[i];
}
```

I metodi e le proprietà che abbiamo visto sono quelli più importanti e che si usano maggiormente nello sviluppo. Tuttavia esistono altri metodi che possono essere utili a seconda dei casi:

- `sort`: converte gli elementi in stringhe e poi ne esegue un ordinamento;
- `reverse`: inverte l'ordine degli elementi;

- `join`: converte gli elementi in stringhe e poi concatena tutte le stringhe separandole con il carattere passato in input al metodo;
- `concat`: crea un nuovo array concatenandone due esistenti.

A differenza dei tipi semplici elencati nella [tabella 7.1](#), gli array sono un tipo complesso, cioè composto da più tipi semplici. Un altro esempio di tipo complesso sono gli oggetti.

Classi e oggetti

JavaScript non ha un supporto alle classi come invece hanno i linguaggi a oggetti. Tuttavia possiamo creare qualcosa di simile a una classe sfruttando una funzione, come [nell'esempio 7.15](#).

Esempio 7.15

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
    this.getFullName = function() {  
        return this.firstName + " " + this.lastName;  
    };  
}
```

In questo esempio creiamo una funzione che accetta due parametri. In JavaScript una funzione è anche un oggetto e tramite la parola chiave `this` accediamo all'oggetto stesso. Poiché JavaScript è un linguaggio dinamico e interpretato, possiamo creare nuove proprietà per l'oggetto semplicemente dichiarandole e assegnando loro un valore così come fatto per le proprietà `firstName` e `lastName`. Oltre alle proprietà, possiamo aggiungere anche metodi, come nel caso di `getFullName`.

Il risultato [dell'esempio 7.15](#) è la definizione di una classe `Person` che ha un costruttore, che accetta due parametri, due proprietà e un metodo. Nel prossimo esempio viene mostrato come utilizzare questa classe.

Esempio 7.16

```
var p = new Person("Stefano", "Mostarda"); // istanzia l'oggetto Person  
alert(p.getFullName()); // mostra a video il nome completo
```

Tramite la parola chiave `new` creiamo un'istanza della classe `Person` e la assegniamo a una variabile, sulla quale poi possiamo invocare metodi e utilizzare proprietà.

Creare classi permette di mantenere il codice pulito e ordinato, ma ci sono casi in cui abbiamo bisogno di creare un oggetto senza avere una classe. In questi casi la dinamicità del JavaScript torna utile, in quanto possiamo creare una classe semplicemente con la seguente riga di codice.

Esempio 7.17

```
var obj = { "firstName": "Stefano", "lastName": "Mostarda" };
```

In questo esempio viene usata la sintassi JSON (JavaScript Object Notation) per creare un oggetto. Aprendo le parentesi graffe si comincia a dichiarare un oggetto. L'oggetto è composto da coppie chiave/valore separate da un carattere “,” (virgola). La chiave è separata dal valore dal carattere “:” (due punti). Infine, si chiudono le parentesi graffe per terminare l'istanziamento dell'oggetto.

Il risultato dell'esempio è un oggetto che ha le proprietà `firstName` e `lastName`, il cui valore è rispettivamente “Stefano” e “Mostarda”.

Lavorare con gli oggetti

Lavorare con gli oggetti in JavaScript merita alcuni approfondimenti poiché ci sono alcune considerazioni da fare. La prima è che essendo JavaScript dinamico e non tipizzato, possiamo aggiungere proprietà e metodi a un oggetto a run time. Questo comportamento può aiutare ma spesso porta più problemi che vantaggi.

Supponiamo di aver creato un'istanza dell'oggetto `Person` (visto nell'esempio 7.15) e di voler cambiare il nome da “Stefano” a “Daniele”. Per fare questo dobbiamo semplicemente impostare la proprietà `firstName` con il nuovo nome. Se scriviamo in maniera errata il nome della proprietà (supponiamo `firstNme`), a run time non riceviamo un errore, bensì viene aggiunta all'oggetto una nuova proprietà `firstNme`, che ha come valore il valore impostato. Il prossimo esempio mostra meglio il problema.

Esempio 7.18

```
var p = new Person("Stefano", "Mostarda");  
alert(p.getFullName());           //mostra a video Stefano Mostarda
```

```
p.firstName = "Daniele"           //crea una nuova proprietà
nell'oggetto p
alert(p.getFullName());           //mostra a video Stefano Mostarda
```

Come si intuisce, nonostante un errore di scrittura del codice, non otteniamo alcun errore in fase di esecuzione e questo spesso è causa di forti perdite di tempo in fase di sviluppo.

Per i metodi vale invece un altro discorso in quanto, se proviamo a invocare un metodo che non esiste, riceviamo un errore e l'esecuzione del codice si arresta.

Un'altra cosa da notare è che internamente un oggetto è come se fosse un array. Infatti, possiamo accedere alle proprietà di un oggetto non solo tramite la sintassi vista finora, ma anche tramite la stessa sintassi vista per accedere a un elemento di un array, con la sola differenza che, invece di passare un indice, passiamo il nome della proprietà. Per capire meglio questo concetto studiamo [l'esempio 7.19](#).

Esempio 7.19

```
var p = new Person("Stefano", "Mostarda");
alert(p.firstName);                //mostra a video Stefano
alert(p["firstName"]);             //mostra a video Stefano
```

Il vantaggio di poter accedere alle proprietà in maniera dinamica (tramite stringa) e non statica (tramite sintassi classica) può aiutare quando si vuole scrivere codice generico. Rimane vero il fatto che se si passa all'indexer il nome di una proprietà che non esiste, questa viene aggiunta all'oggetto e nessun errore viene generato.

Conclusioni

Ogni sviluppatore web deve conoscere il JavaScript. Alla base di ogni singola interazione tra l'utente e la nostra pagina web c'è sempre un po' di codice JavaScript. Se all'inizio del Web questo linguaggio serviva solamente per pochi scopi, adesso è fondamentale sotto qualunque aspetto.

In questo capitolo abbiamo gettato le basi per la conoscenza del JavaScript, illustrando quali sono i tipi di base e come lavorare con questi e con tipi complessi come array, classi e oggetti. Abbiamo poi visto come

influenzare il flusso di esecuzione del codice attraverso le istruzioni di selezione e quelle di iterazione. Infine, abbiamo compreso come suddividere il codice in metodi, così da renderlo più ordinato.

Nel prossimo capitolo ci occuperemo di JavaScript in maniera più approfondita, mostrando come interagire con gli oggetti sulla pagina e con le altre API messe a disposizione dalla piattaforma HTML.

JavaScript avanzato

Nel capitolo precedente abbiamo presentato una panoramica introduttiva di JavaScript, il linguaggio di programmazione per le pagine web, mostrando quali siano le sue principali caratteristiche a livello di sintassi e utilizzo di base.

In questo capitolo mettiamo in azione le conoscenze apprese nel precedente. Cominceremo col vedere come accedere ai controlli della pagina tramite JavaScript e come manipolarli per trasformare la pagina in seguito alle interazioni dell'utente. Successivamente vedremo nel dettaglio le principali API HTML, come quelle per AJAX, per la gestione di dati in locale, per la geolocalizzazione, per la navigazione personalizzata tra pagine e per la programmazione multithread.

Queste funzionalità mettono a disposizione una piattaforma di sviluppo estremamente potente e permettono di creare applicazioni web sempre più ricche e coinvolgenti per l'utente finale, al punto da non riuscire quasi a distinguere un'applicazione web da una desktop.

Capire il Document Object Model (DOM)

JavaScript è un linguaggio che permette di manipolare gli oggetti presenti nella pagina, al fine di favorire l'interazione dell'utente con la pagina stessa. Per esempio, possiamo rendere o non rendere visibile una sezione della pagina per dare un messaggio all'utente, possiamo disabilitare il bottone di conferma finché l'utente non ha inserito correttamente tutti i dati, possiamo mostrare un menu quando l'utente passa con il mouse su un elemento ecc. L'unico limite alle interazioni che possiamo abilitare con JavaScript è la nostra immaginazione.

Per interagire con gli oggetti presenti sulla pagina, JavaScript deve avere a disposizione un contenitore dove andare a recuperare questi oggetti. Questo contenitore è il Document Object Model (DOM d'ora in poi). Quando una

pagina viene inviata al browser, il browser interpreta il codice HTML e lo renderizza. Allo stesso tempo, il browser crea un oggetto in memoria per ogni frammento HTML (sia esso un tag input, un testo, un'immagine o qualunque altra cosa). Alla fine, tutti gli oggetti creati vengono inseriti in una struttura unica: il DOM.

Il DOM ha una struttura ad albero che è equivalente a quella della pagina. Così come possiamo annidare tag, testi e altro ancora, anche nel DOM gli oggetti sono annidati e non hanno quindi una struttura piatta. La [figura 8.1](#) mostra il DOM generato da una pagina HTML.

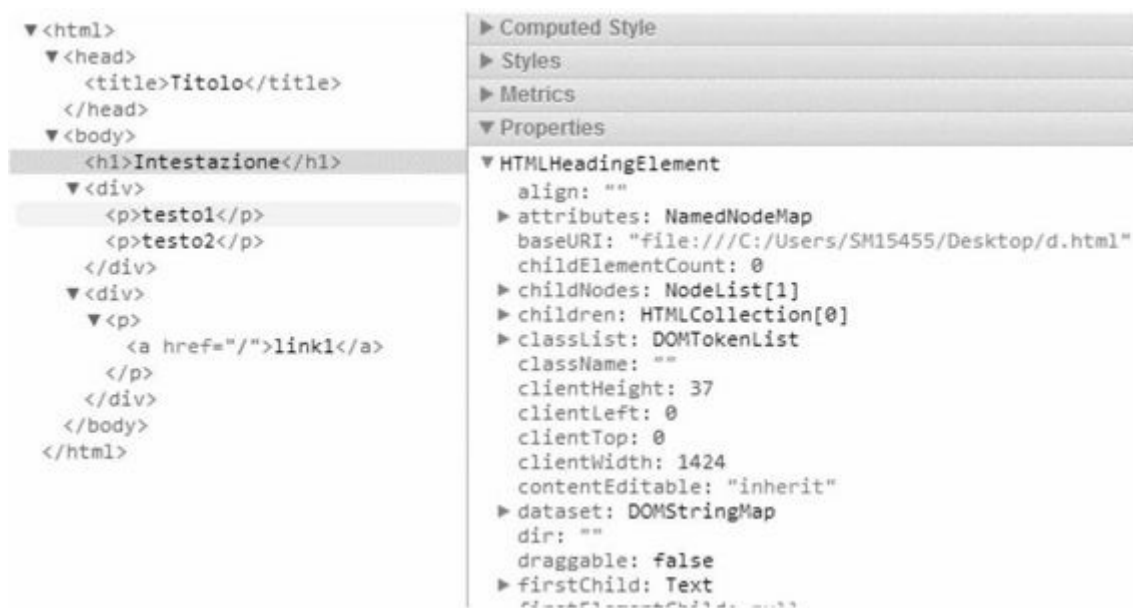


Figura 8.1 – La parte sinistra della figura mostra gli oggetti nel DOM (viene mostrato il codice HTML per comodità), mentre la parte destra mostra alcune delle proprietà dell'oggetto selezionato (l'oggetto corrispondente al tag H1, in questo caso).

Come la [figura 8.1](#) mostra chiaramente, il DOM introduce un livello di astrazione tra l'HTML e il nostro codice JavaScript. Infatti, senza il DOM, per recuperare gli oggetti nella pagina dovremmo interpretare il codice HTML, con conseguente lentezza e complessità del codice JavaScript. Queste operazioni vengono invece svolte dal browser, che poi inserisce gli oggetti nel DOM, semplificando notevolmente il nostro lavoro.

In JavaScript possiamo accedere al DOM tramite la variabile `document`. Questa variabile contiene un oggetto che espone tutti i metodi per navigare il DOM in base alle nostre necessità. `document` è una variabile creata dal browser, il che significa che non dobbiamo preoccuparci di dichiararla e

valorizzarla. Inoltre, `document` è anche una variabile globale e cioè una variabile che è sempre raggiungibile da qualunque punto del codice JavaScript. Nella prossima sezione ci occupiamo di come sfruttare `document` per navigare il DOM.

Utilizzare i selettori per navigare il DOM

Supponiamo di voler recuperare dal DOM un oggetto `span` che ha l'attributo `id` impostato su `text`. Per recuperare questo elemento dal DOM, possiamo utilizzare il metodo `getElementById`, dell'oggetto `document`, passandogli in input la stringa `"text"`, come mostrato nel prossimo esempio.

Esempio 8.1

```
var obj = document.getElementById("text");
```

Recuperare un oggetto dato il suo attributo `id` è molto semplice. Tuttavia ci sono situazioni in cui vogliamo recuperare più oggetti e in cui la ricerca non deve avvenire per nome.

Supponiamo, per esempio, di voler recuperare tutti i tag `input` di una pagina per eseguire un controllo di obbligatorietà. Se abbiamo 20 controlli `input` nella pagina, dovremmo utilizzare per 20 volte il metodo `getElementById`. Questo approccio è, come minimo, scomodo e soggetto a errori. Per semplificare il codice possiamo usare il metodo `getElementsByTagName` dell'oggetto `document` passando in input il tipo di tag che vogliamo recuperare. Per esempio, per recuperare tutti gli `input` di una pagina, dovremmo scrivere il seguente codice.

Esempio 8.2

```
var objs = document.getElementsByTagName("input");
```

Il metodo `getElementsByTagName` è molto semplice da utilizzare, così come il metodo `getElementById`. Oltre a questi due, esiste anche il metodo per ricercare gli oggetti tramite la loro classe CSS: `getElementsByClassName`. Il nome del metodo è auto esplicativo; nel nostro caso possiamo usare questo metodo per recuperare tutti i tag che hanno la classe CSS `text`, come viene mostrato nel prossimo esempio.

Esempio 8.3

```
var obj = document.getElementsByClassName("text");
```

Tutti i metodi che abbiamo esaminato finora offrono una variante molto interessante, cioè permettono di recuperare gli oggetti non solo cercando all'interno dell'intero DOM, ma anche all'interno di un solo sottoinsieme. Per esempio, potremmo prima ricercare il tag `div` con id `mainText` e poi cercare i tag `p` al suo interno. Per fare questo, dobbiamo usare il metodo `getElementsByTagName` non sull'oggetto `document` ma sull'oggetto restituito dal metodo `getElementById`. Nel seguente codice mettiamo in pratica l'esempio appena fatto.

Esempio 8.4

```
var container = document.getElementById("mainText");  
var children = container.getElementsByTagName("p");
```

Anche in questo caso il codice è estremamente semplice. La potenza di questa tecnica risiede nella possibilità di combinare i metodi per applicare filtri anche complessi sulla nostra ricerca nel DOM.

Una volta che abbiamo ottenuto gli oggetti che cerchiamo, possiamo partire da questi per navigare ulteriormente nel DOM alla ricerca di altri oggetti attraverso una serie di metodi elencati nella [tabella 8.1](#).

<i>Metodo</i>	<i>Descrizione</i>
<code>childNodes</code>	Restituisce i nodi figli dell'oggetto del DOM su cui applichiamo il metodo.
<code>firstChild</code>	Restituisce il primo nodo figlio dell'oggetto del DOM su cui applichiamo il metodo.
<code>lastChild</code>	Restituisce l'ultimo nodo figlio dell'oggetto del DOM su cui applichiamo il metodo.
<code>nextSibling</code>	Restituisce il nodo successivo allo stesso livello dell'oggetto del DOM su cui applichiamo il metodo.
<code>previousSibling</code>	Restituisce il nodo precedente allo stesso livello dell'oggetto del DOM su cui applichiamo il metodo.

parentNode	Restituisce il nodo padre dell'oggetto del DOM su cui applichiamo il metodo.
------------	--

Tabella 8.1 – Elenco dei metodi di navigazione del DOM di un oggetto.

L'utilizzo di questi metodi è molto semplice ed è mostrato nel seguente codice.

Esempio 8.5

```
//Recupera un div con id mainText
var obj = document.getElementById("mainText");

//Recupera tutti i nodi figli del nodo obj
var childNodes = obj.childNodes();

//Recupera il primo nodo figlio del nodo obj
var firstChild = obj.firstChild();

//Recupera l'ultimo nodo figlio del nodo obj
var lastChild = obj.lastChild();

//Recupera il nodo successivo al nodo obj sullo stesso livello
var nextSibling = obj.nextSibling();

//Recupera il nodo precedente al nodo obj sullo stesso livello
var previousSibling = obj.previousSibling();

//Recupera il nodo padre del nodo obj
var parent = obj.parentNode();
```

Questo esempio mostra che il codice da scrivere è estremamente semplice ed efficace. Tuttavia, si può fare di meglio e risparmiare ulteriori righe di codice attraverso due metodi (`querySelector` e `querySelectorAll`) che offrono molta più flessibilità nella ricerca degli oggetti nel DOM e che sono argomento del prossimo paragrafo.

Utilizzare i selettori CSS per navigare il DOM

Il metodo `querySelector` effettua una query sul DOM e restituisce il primo oggetto restituito dalla query. Se la query restituisce più oggetti, al codice viene restituito solo il primo. Se la query non restituisce nemmeno un oggetto, viene restituito `null`.

Il metodo `querySelectorAll` effettua una query sul DOM e restituisce tutti gli oggetti recuperati. Qualunque quantità di oggetti sia recuperata dalla query (nessuno, uno o più di uno) il metodo `querySelectorAll` restituisce una lista.

Quello che rende veramente speciali i metodi `querySelector` e `querySelectorAll` è il fatto che, per ricercare gli oggetti da recuperare, utilizzano i selettori CSS. Gli esempi che sono mostrati nel codice qui sotto danno una precisa idea di quanto questi metodi semplifichino la vita.

Esempio 8.6

```
var obj, list;
//1. Recupera il primo oggetto con ID mainText
obj = document.querySelector("#mainText");

//2. Recupera tutti div
list = document.querySelectorAll("div");

//3. Recupera tutti i tag con classe CSS "myCssClass"
list = document.querySelectorAll(".myCssClass");

//4. Recupera il primo tag div
obj = document.querySelector("div");

//5. Recupera tutti i tag p contenuti in un tag div
list = document.querySelectorAll("div p");

//6. Recupera tutti i tag p figli diretti di un tag div
list = document.querySelectorAll("div > p");

//7. Recupera le righe pari di una tabella
list = document.querySelectorAll("table tr:nth-child(even)");

//8. Recupera tutti gli elementi disabilitati
list = document.querySelectorAll("form input[disabled]");

//9. Recupera tutti i checkbox
list = document.querySelectorAll("form input[type='checkbox']");
```

I metodi mostrati in questo esempio dimostrano come i metodi `querySelector` e `querySelectorAll` coprano gran parte delle esigenze:

- il risultato ottenuto dall'esempio 1 è equivalente a quello ottenuto invocando `getElementById`;

- il risultato ottenuto dagli esempi 2 e 4 è equivalente a quello ottenuto invocando `getElementsByTagName`;
- il risultato ottenuto dall'esempio 3 è equivalente a quello ottenuto invocando `getElementsByClassName`;
- il risultato ottenuto dall'esempio 5 può essere ottenuto chiamando prima `getElementsByTagName` per recuperare tutti i tag `div`, poi ciclando sui tag ottenuti e invocando nuovamente `getElementsByTagName` su ognuno di essi. Non mostreremo questo codice, ma risulta chiaro come il codice da scrivere sia maggiore rispetto all'esempio 5;
- il risultato ottenuto dall'esempio 6 può essere ottenuto chiamando prima `getElementsByTagName`, per recuperare tutti i tag `div`, poi ciclando sui tag ottenuti e, per ognuno di essi, ciclare sui nodi figli (tramite la proprietà `childNodes`), prendendo solo quelli che rappresentano un tag `p`). In questo caso il codice da scrivere è ancora maggiore rispetto a quello necessario nell'esempio 5.

Questi esempi mostrano chiaramente che la versatilità di `querySelector` e `querySelectorAll` ci permette di ottenere ciò che vogliamo con una quantità di codice minima.

Una volta recuperati dal DOM gli oggetti che ci servono, possiamo modificarli come vogliamo. Per esempio, possiamo disabilitare il pulsante di conferma finché tutti i checkbox non sono selezionati oppure mostrare un tip all'utente quando imposta il focus su una textbox.

Manipolare gli oggetti del DOM

Supponiamo di dover realizzare una form di login per gli utenti del nostro sito. Uno dei requisiti è che il tasto login non deve essere abilitato finché username e password non sono inserite. Il codice HTML della pagina è il seguente.

Esempio 8.7

```
<html>
  <head>
    <title></title>
  </head>
```

```
<body>
  <form action="">
    <div>
      <span>Username</span>
      <input type="text"/>
    </div>
    <div>
      <span>Password</span>
      <input type="password"/>
    </div>
    <div>
      <input type="submit" value="Login" disabled/>
    </div>
  </div>
</body>
</html>
```

Questo codice produce l'output mostrato nella [figura 8.2](#).



The image shows a web form with two labels, 'Username' and 'Password', each followed by a text input field. Below these fields is a button labeled 'Login'. The button has a light gray background and a thin border, indicating it is disabled.

Figura 8.2 – La maschera di login ottenuta dal codice mostrato [nell'esempio 8.7](#).

Quando username e password sono entrambe valorizzate, possiamo abilitare il pulsante di login. Per fare questo, intercettiamo l'evento `change` di entrambe le textbox e verifichiamo che siano valorizzate così come viene mostrato nel prossimo esempio.

Esempio 8.8

```
<html>
  <head>
    <title>Page</title>
    <script type="text/javascript">
      function change () {
        var inputs = document.querySelectorAll(
```

```

        "input[type='text'], input[type='password']");
var button =
document.querySelector("input[type='submit']");
button.disabled = "";

    for (var i=0; i<inputs.length; i++){
        if (inputs[i].value == ""){
            button.disabled = "disabled";
            break;
        }
    }
}
</script>
</head>
<body>
    <form action="">
        <div>
            <span>Username</span>
            <input type="text" onchange="change"/>
        </div>
        <div>
            <span>Password</span>
            <input type="password" onchange="change"/>
        </div>
        <div>
            <input type="submit" value="Login" disabled="disabled"/>
        </div>
    </form>
</body>
</html>

```

Nel codice HTML l'unica modifica che dobbiamo fare è quella di specificare il metodo da chiamare quando i valori delle textbox cambiano (`onchange="change()"`). Nel codice JavaScript prima vengono recuperate le textbox e poi il pulsante. Successivamente, il pulsante viene abilitato impostando la proprietà `disabled` a stringa vuota. Infine, viene eseguito un ciclo sulle textbox per verificare che siano entrambe valorizzate (la proprietà `value` deve essere diversa da stringa vuota). Se anche solo una non è valorizzata, si disabilita il pulsante impostandone la proprietà `disabled` a `disabled`. Quando entrambe le textbox sono valorizzate, la form apparirà come nella [figura 8.3](#).

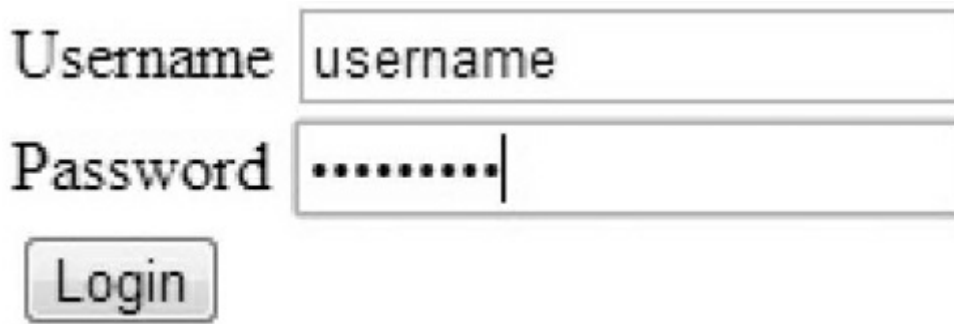


Figura 8.3 – La maschera di login una volta valorizzate entrambe le textbox.

Il recupero della proprietà `value` e l'impostazione della proprietà `disabled` è un esempio molto banale di manipolazione di un oggetto del DOM. Se invece che abilitare e disabilitare il pulsante decidessimo di renderlo visibile o invisibile, potremmo utilizzare il seguente codice JavaScript:

Esempio 8.9

```
//bottone visibile
button.style.display = "block";

//bottone invisibile
button.style.display = "none";
```

Tramite la proprietà `style` possiamo modificare tutti gli stili CSS di un oggetto e quindi modificare il modo in cui questo viene visualizzato.

Esistono tantissime proprietà e tantissimi metodi che possiamo utilizzare per recuperare e/o modificare gli elementi del DOM. Elencare e mostrare tutte queste proprietà è fuori dagli scopi di questo libro. Tuttavia, nella [tabella 8.2](#) potete trovare i metodi e le proprietà più comunemente usati.

Proprietà/metodo	Descrizione
<code>addEventListener</code>	Collega una funzione a un evento di un oggetto del DOM (il click di un pulsante, il focus di una textbox ecc).
<code>attributes</code>	Restituisce un <code>Array</code> con tutti gli attributi HTML dell'oggetto. È possibile aggiungere, modificare e rimuovere elementi a questa proprietà.
<code>blur</code>	Toglie il focus dal controllo.

<code>className</code>	Restituisce il valore dell'attributo <code>class</code> del nodo HTML. È possibile impostare questa proprietà per modificare le classi CSS associate all'oggetto.
<code>click</code>	Scatena il click del controllo.
<code>disabled</code>	Abilita e disabilita un controllo. Per disabilitare un controllo, la proprietà deve essere impostata a <code>disabled</code> . Per abilitare un controllo, la proprietà deve essere impostata a stringa vuota.
<code>focus</code>	Imposta il focus sul controllo.
<code>id</code>	Restituisce l'id del controllo HTML. Questa proprietà è modificabile.
<code>innerHTML</code>	Restituisce il codice HTML all'interno del controllo HTML. Possiamo modificare questa proprietà per modificare il contenuto del controllo HTML.
<code>offsetLeft</code>	Restituisce la distanza tra il bordo sinistro dell'oggetto e il bordo sinistro del contenitore dell'oggetto.
<code>offsetTop</code>	Restituisce la distanza tra il bordo superiore dell'oggetto e il bordo superiore del contenitore dell'oggetto.
<code>style</code>	Restituisce tutte le proprietà CSS del controllo. Questa proprietà è modificabile.
<code>tagName</code>	Restituisce il nome del tag HTML completamente in maiuscolo.

Tabella 8.2 – Principali proprietà e metodi necessari a manipolare un oggetto nel DOM.

Saper manipolare gli oggetti del DOM è essenziale per poter creare siti usabili e che siano velocemente reattivi all'input di un utente. Mostrare messaggi in popup, abilitare e disabilitare oggetti sono cose che ormai fanno parte dell'interfaccia di tutti i giorni e che sono ottenibili esclusivamente manipolando il DOM da JavaScript.

Gestire gli eventi degli oggetti

Nella precedente sezione abbiamo visto come gestire l'inserimento di testo in una textbox da parte dell'utente, attraverso l'evento `onchange`. Questo è solo uno dei tanti eventi messi a disposizione da HTML per reagire all'input dell'utente. Infatti abbiamo eventi per reagire quando l'utente passa con il mouse su un oggetto e quando esce, quando effettua click su un pulsante, quando un controllo prende il focus e quando lo perde, quando viene premuto un tasto, quando viene ridimensionata la finestra e altri ancora. Alcuni di questi eventi sono comuni a tutti gli oggetti, mentre altri sono dedicati a un oggetto. Nella [tabella 8.3](#) possiamo vedere un riepilogo degli eventi principali e dei controlli che li espongono.

<i>Evento</i>	<i>Oggetto</i>	<i>Descrizione</i>
<code>onerror</code>	<code>window</code>	Scatenato quando si solleva un'eccezione non gestita dal codice (per esempio, quando si invoca un metodo che non esiste).
<code>onload</code> <code>onunload</code>	<code>window</code>	Scatenati rispettivamente quando la pagina è caricata dal browser (e il DOM è pronto) e quando viene scaricata perché si naviga verso un'altra pagina o si chiude il browser.
<code>onresize</code>	<code>window</code>	Scatenato quando si ridimensiona il browser.
<code>onsubmit</code>	<code>form</code>	Scatenato quando si esegue il submit di una form.
<code>oninvalid</code>	<code>form</code> , controlli di input	Scatenato quando la form o il controllo di input non rispettano le regole di validazione.
<code>onfocus</code> <code>onblur</code>	Controlli di input	Scatenato quando un oggetto rispettivamente prende e perde il focus.
<code>onclick</code>	Tutti	Scatenato quando l'utente clicca su un controllo.
<code>onkeydown</code> <code>onkeypress</code> <code>onkeyup</code>	Tutti	Scatenati rispettivamente quando un tasto della tastiera viene tenuto premuto, premuto, rilasciato.

Tabella 8.3 – Principali eventi degli oggetti.

Qualunque sia l'evento che andiamo a intercettare, il codice da scrivere è sempre lo stesso: nel codice HTML aggiungiamo un attributo per l'evento e, al suo interno, invochiamo una funzione (detta anche delegato). Questa funzione non accetta parametri, ma al suo interno la variabile `this` corrisponde all'oggetto che ha scatenato l'evento. Poter accedere all'oggetto che ha scatenato l'evento può essere utile quando vogliamo che più controlli reagiscano allo stesso modo a un evento (per esempio quando vogliamo scatenare la validazione dei controlli nel momento in cui perdono il fuoco).

Collegare il delegato all'evento attraverso un attributo HTML è molto semplice, ma ha il problema di mischiare interfaccia e logica che è sempre bene tenere indipendenti per una maggior pulizia del codice. Quando vogliamo mantenere separati interfaccia e logica, dobbiamo sottoscriverci agli eventi da codice tramite il metodo `addEventListener`, come mostrato nell'esempio 8.10.

Esempio 8.10

```
window.addEventListener("load", function(){
    document.getElementById("btn").addEventListener("click",
        function(){
            alert('cliccato');
        });
});
```

Il metodo `addEventListener` accetta in input il nome dell'evento e la funzione da invocare quando viene scatenato l'evento. In questo caso ci siamo sottoscritti all'evento `load` della pagina e all'interno di questo metodo ci sottoscriviamo al `click` di un bottone. Grazie a questa tecnica possiamo eliminare gli eventi dal codice HTML e gestire tutto da JavaScript, garantendo una totale separazione tra interfaccia e logica.

nota

Così come il metodo `addEventListener` aggiunge un delegato a un evento, il metodo `removeEventListener`, che ha la stessa firma, lo rimuove.

Oltre a manipolare il DOM, una delle necessità più comuni dello sviluppo client è quella di accedere al server per recuperare dati: questa tecnica prende il nome di AJAX.

AJAX con XMLHttpRequest

AJAX è la tecnica tramite la quale, da JavaScript, possiamo invocare un url e recuperarne la risposta in maniera asincrona e senza che avvenga un ricaricamento della pagina da parte del browser. Questa tecnica è basata sulla classe JavaScript `XMLHttpRequest`.

L'esempio 8.11 mostra il codice necessario per effettuare una chiamata al server in modo asincrono.

Esempio 8.11

```
var url = "/getdata?id=1";
var request = return new XMLHttpRequest();
request.open("GET", url, true);
request.onreadystatechange = function ()
{
    if (request.readyState == 4)
    {
        alert(request.responseText);
    }
}
request.send(null);
```

Il codice dell'esempio 8.11 è semplice: viene creata un'istanza di `XMLHttpRequest` e viene aperta una connessione asincrona verso l'indirizzo desiderato, specificando il metodo HTTP da usare (nell'esempio, "GET") e, intercettando l'evento `onreadystatechange`, si definisce l'azione di callback da eseguire una volta ricevuta la risposta dal server (nell'esempio 8.11 viene semplicemente visualizzata la proprietà `responseText` in un `alert`). L'ultima riga invoca il metodo `send` per avviare la chiamata vera e propria.

`XMLHttpRequest` espone altre proprietà e metodi oltre quelli già visti per creare funzionalità comode quando si lavora con il server. Nelle prossime sezioni vedremo quali sono queste funzionalità, partendo dallo stato di avanzamento di una richiesta.

Gestione dello stato di avanzamento

Oltre al già menzionato `onreadystatechange`, `XMLHttpRequest` prevede una serie di eventi specifici per intercettare l'invio o la ricezione di blocchi di dati (chunk) nell'ambito di una richiesta HTTP: `onloadstart`, `onprogress`, `onabort`, `onerror`, `onload`, `onreadystatechange` e `onloadend`. Utilizzando opportunamente questi eventi, possiamo adattare l'interfaccia grafica della nostra applicazione per mostrare in tempo reale all'utente lo stato di avanzamento della richiesta al server: che percentuale è stata completata, quanto tempo stimiamo manchi al completamento dell'operazione e così via. L'esempio più tipico di questo tipo di utilizzo è la trasmissione o la ricezione di un file.

Trasferimento di file

Utilizzando le File API, l'applicazione può accedere al contenuto di un file locale selezionato da un utente attraverso un controllo di tipo `<input type="file"/>`. In più, utilizzando le API per il supporto al Drag&Drop e l'oggetto Data Transfer, abbiamo anche la possibilità di leggere un file trascinato direttamente dal desktop in un'area specifica della pagina.

Il metodo `send` di `XMLHttpRequest` può ricevere come parametro un oggetto di tipo `Blob`, `FormData` (coppie di tipo chiave/valore) o file e inviarne il contenuto in modo progressivo. [L'esempio 8.12](#) mostra come inviare un file e altre informazioni al server.

Esempio 8.12

```
var formData = new FormData();

formData.append("Id", 1);
formData.append("userfile", fileInput.files[0]);

var request = new XMLHttpRequest();
request.open("POST", "fileupload.handler");
request.send(formData);
```

Nel codice [dell'esempio 8.12](#), creiamo un oggetto di tipo `FormData` e poi lo popoliamo con una lista di coppie chiave valore, dove la prima coppia contiene un numero mentre la seconda contiene un file selezionato dall'utente. Successivamente invochiamo il metodo `send`, passando in input l'oggetto `FormData` creato.

Chiamate cross-domain

Usando le impostazioni predefinite `XMLHttpRequest` può inviare e ricevere dati esclusivamente da URL nello stesso dominio della pagina che ha istanziato l'oggetto. Questa restrizione è imposta per ragioni di sicurezza poiché, altrimenti, sarebbe possibile effettuare attacchi di tipo cross-site scripting (XSS).

Tutti i browser moderni supportano Cross-Origin Resource Sharing (CORS) quindi consentono di evitare questa restrizione, garantendo al contempo un buon livello di sicurezza. Si tratta di un meccanismo per definire le politiche di condivisione delle risorse tra più domini mediante dichiarazioni trasmesse come intestazioni della richiesta HTTP (per maggiori informazioni rimandiamo alla pagina <http://aspit.co/alm> sul sito del W3C) e che si riflette anche sul comportamento di `XMLHttpRequest`. Si tratta di una funzionalità importante perché ci consente di creare applicazioni AJAX in grado di utilizzare direttamente servizi forniti da terze parti, senza dover ricorrere a espedienti ingegnosi o a proxy lato server.

nota

Nel [capitolo 11](#) parleremo di un framework JavaScript chiamato `jQuery`. Questo framework espone API per astrarre `XMLHttpRequest` dal nostro codice, rendendolo quindi più semplice da utilizzare.

History API

La modalità di consultazione del Web più comune tra gli utenti consiste nel muoversi avanti e indietro, in rigoroso ordine cronologico, tra le pagine visitate durante una sessione di navigazione. Tipicamente, un utente, una volta effettuata una ricerca, fa click sul primo risultato, visualizza qualche pagina del sito trovato dal motore, quindi – usando gli appositi tasti del browser – torna ai risultati della ricerca per esplorare il collegamento successivo, e così via.

La cronologia di navigazione, oltre che con i pulsanti del browser, è accessibile in maniera programmatica attraverso l'oggetto esposto dalla proprietà `history` dell'oggetto `window`. I metodi `back()`, `forward()` e `go()`

permettono infatti di muoversi facilmente tra le pagine visitate, usando JavaScript. Oltre a questi metodi, abbiamo anche la possibilità di modificare l'elenco degli indirizzi memorizzati dal browser attraverso i metodi `pushState` e `replaceState`, e di essere notificati sui cambi di URL tramite l'evento `onpopstate`.

Questi ultimi metodi ed eventi sono stati introdotti con HTML5 per sopperire ai limiti di navigazione e ai problemi di usabilità che affliggono i siti che fanno forte uso del paradigma AJAX o comunque di una pesante manipolazione del DOM.

Con la diffusione di AJAX, infatti, la classica modalità di esplorazione descritta in precedenza è in buona parte decaduta: una sola pagina è in grado di mostrare diversi contenuti e intere applicazioni o siti web vengono esplorati senza cambiare contesto (URL), impedendo di fatto all'utente di "tornare sui propri passi" muovendosi avanti e indietro nella cronologia come è abituato a fare.

Supponiamo di voler gestire la consultazione di un catalogo prodotti utilizzando la classica visualizzazione master-details, dove i dettagli dei singoli prodotti siano ottenuti dal server via AJAX. La [figura 8.4](#) mostra la struttura della pagina ipotizzata.



Figura 8.4 – Catalogo prodotti con visualizzazione master-details in un'unica pagina.

Quando l'utente seleziona un prodotto dall'elenco (indicato con la lettera "M")

nella figura), l'applicazione richiede al server i dettagli del prodotto per visualizzarne la scheda (indicata con la lettera "D" nella figura) mediante una chiamata asincrona con XMLHttpRequest. Utilizzando le API per la navigazione, possiamo memorizzare i prodotti visualizzati dall'utente, consentendogli di muoversi all'interno della cronologia di navigazione come se avesse effettivamente visitato URL diversi. Il listato 8.13 presenta il codice JavaScript necessario.

Esempio 8.13 – Utilizzo delle History API

```
window.onload = function ()
{
    if (!Modernizr.history)
    {
        alert("Il tuo browser non supporta History API");
        return;
    }
    trackProductClick(document.getElementById("product1"));
    trackProductClick(document.getElementById("product2"));
    // omissis: intercettare i link di tutti i prodotti

    // intercetto la navigazione della cronologia:
    onpopstate = function (event) { showProductDetails(event.state);
    }
};

function trackProductClick(link)
{
    link.addEventListener("click", function (e)
    {
        // omissis: ottenere i dati del prodotto dal server via AJAX
        var product = { id: "...", url: "http://...",
            name: "...", description: "...", price: "..." };
        history.pushState(product, product.name, product.url);
        showProductDetails(product);
        e.preventDefault();

    }, false);
};

function showProductDetails(product)
{
    if (product != null)
    {
        // omissis: mostrare i dati di product nella scheda prodotto
    }
};
```

Come possiamo notare nell'esempio, risulta molto semplice rendere un'applicazione AJAX compatibile con la classica modalità di navigazione attraverso la cronologia: è sufficiente aggiungere all'oggetto `history`, tramite il metodo `pushState`, le informazioni del prodotto visualizzato e intercettare l'evento `onpopstate` per aggiornare la visualizzazione della scheda prodotto quando l'utente si muove avanti e indietro nella cronologia. Con poche semplici righe di codice aggiuntivo, grazie alle History API, possiamo dunque aumentare sensibilmente l'usabilità delle nostre applicazioni che usano AJAX.

Geolocalizzazione

Esistono molteplici scenari in cui un'applicazione ha l'esigenza di conoscere la posizione geografica del dispositivo connesso, come, per esempio, nella ricerca dei punti vendita più vicini, nel calcolo dei percorsi stradali per raggiungere la sede di un'azienda o per agevolare le interazioni tra gli utenti nell'ambito di community e social network.

Le API per la geolocalizzazione semplificano notevolmente la creazione di queste funzionalità da parte degli sviluppatori. Il W3C ha infatti formalizzato in una serie di specifiche le funzionalità per conoscere automaticamente la posizione del dispositivo connesso e le ultime versioni di tutti i principali browser supportano pienamente queste specifiche. La modalità per determinare la posizione del dispositivo si basa sul miglior sistema di tracciamento disponibile tra la localizzazione dell'indirizzo IP, del MAC address Wi-Fi o Bluetooth, della posizione della connessione RFID o Wi-Fi oppure, per esempio per gli smartphone, utilizzando il GPS integrato o basandosi sull'ID della cella GSM/CDMA.

nota

Le specifiche di Geolocation API sono gestite dal consorzio W3C e sono disponibili all'indirizzo: <http://aspit.co/aln>.

Le API per la geolocalizzazione espongono due semplici metodi per determinare la posizione dell'utente:

- `getCurrentPosition` per ricavare in modo asincrono le coordinate attuali;

- `watchPosition` analogo al precedente ma in grado di determinare la posizione, in modo da conoscere in tempo reale gli eventuali spostamenti di un dispositivo mobile; l'osservazione della posizione può essere interrotta invocando il metodo `clearWatch`.

Utilizzare le funzioni per la geolocalizzazione è molto semplice: ci basta invocare il metodo desiderato, specificando la funzione di callback a cui notificare la posizione calcolata, oltre a un eventuale metodo da richiamare in caso di errore e alle opzioni da utilizzare, così da poter specificare, per esempio, se privilegiare la precisione del risultato alla velocità di calcolo oppure se richiedere un timeout per l'esecuzione dell'operazione, come mostrato [nell'esempio 8.14](#).

Esempio 8.14 – Visualizzare la posizione del client su una mappa, usando Geolocation API.

```
function initPage()
{
    if (!Modernizr.geolocation)
    {
        alert("Il client non supporta la geolocalizzazione");
        return;
    }

    // ottengo la posizione del visitatore:
    navigator.geolocation.getCurrentPosition(showMap,
    function (error) { alert(error.message); },
    { enableHighAccuracy: true, timeout: 10000, maximumAge: 0 });
};

function showMap(position)
{
    // visualizzo le informazioni sulla posizione:
    document.getElementById("debug").innerHTML =
        "<p>latitude: " + position.coords.latitude + "<br>\n" +
        "longitude: " + position.coords.longitude + "<br>\n" +
        "accuracy: " + position.coords.accuracy + " metri</p>";

    // creo una mappa con Google Maps centrata sulla posizione:
    var location = new google.maps.LatLng(position.coords.latitude,
    position.coords.longitude);

    var map = new google.maps.Map(document.getElementById("map"), {
        zoom: 12, mapTypeId: google.maps.MapTypeId.ROADMAP, center:
location
    });
};
```

```
// aggiungo un marker alla mappa:  
var marker = new google.maps.Marker({ map: map, draggable: true,  
animation: google.maps.Animation.DROP, position: location });  
// visualizzo l'area con la precisione del calcolo:  
new google.maps.Circle({ map: map, center: clientLocation,  
radius: position.coords.accuracy, strokeColor: "#0080FF",  
strokeOpacity: 0.5, strokeWeight: 1, fillColor: "#0080FF",  
fillOpacity: 0.2 });  
};
```

Il codice [dell'esempio 8.14](#) mostra come visualizzare su una mappa l'attuale posizione del client. Al primo accesso, il browser richiede il consenso dell'utente al tracciamento. Quindi, se l'applicazione viene autorizzata, viene caricata la mappa e mostrato un segnaposto in corrispondenza della posizione calcolata (latitudine e longitudine) e viene evidenziata un'area corrispondente alla precisione con cui il calcolo è stato effettuato, come mostrato nella [figura 8.5](#).

Geolocation API



latitude: 45.6161505
longitude: 8.9420603
accuracy: 793 metri

Figura 8.5 – Le API per la geolocalizzazione in azione.

La [figura 8.5](#) presenta il risultato finale del codice d'esempio proposto: la pagina visualizza la mappa con la posizione del visitatore, evidenziando la precisione di calcolo raggiunta e, in forma testuale, le coordinate del punto localizzato.

Web Storage

È difficile immaginare un'applicazione che non richieda la persistenza di dati: parametri di configurazione, preferenze dell'utente, informazioni di accesso o relative alle entità trattate (catalogo prodotti, database utenti ecc.), i punteggi record ottenuti o i livelli sbloccati di un gioco e così via. Nell'ambito di

applicazioni web, i dati vengono tipicamente gestiti attraverso due modalità: o sono mantenuti sul server e scambiati con il client a ogni richiesta, oppure salvati nei cookie che però, date le numerose limitazioni che presentano, risultano inadeguati nella maggior parte dei casi.

Fortunatamente, grazie alle Web Storage API, abbiamo la possibilità di gestire con facilità grandi volumi di dati – generalmente il limite è impostato a ben 5 MB per ogni applicazione – direttamente all'interno del browser.

HTML5 ci mette a disposizione due diversi tipi di archivi, `localStorage` e `sessionStorage`; l'unica differenza tra i due è che il primo risulta persistente a tempo indeterminato mentre il secondo, come il nome stesso suggerisce, viene automaticamente svuotato al termine della sessione di navigazione.

Entrambi gli storage sono costituiti da insiemi di coppie chiave/valore, gestibili come un semplice array associativo o attraverso i seguenti metodi:

- `length`: per determinare il numero totale di elementi memorizzati;
- `key(index)` per ottenere una chiave in base alla sua posizione nell'indice;
- `getItem(key)` per ottenere un valore memorizzato data la chiave corrispondente;
- `setItem(key, value)` per salvare un nuovo elemento, fornendo la chiave e il relativo valore associato;
- `removeItem(key)` per eliminare un elemento;
- `clear()` per rimuovere tutti gli elementi precedentemente memorizzati.

nota

Le specifiche e la documentazione completa di Web Storage API sono disponibili all'indirizzo <http://aspit.co/a10>.

Come abbiamo detto in precedenza, le modalità di accesso allo storage locale e a quello di sessione sono identiche poiché entrambi implementano la stessa interfaccia; [l'esempio 8.15](#) mostra come sia possibile creare una funzione `getStorage()` per semplificare l'accesso ai due tipi di storage, astruendo dal tipo richiesto e riducendo così al minimo la dipendenza con il sistema di

persistenza scelto.

Esempio 8.15 – Usare il pattern Abstract Factory per accedere a Local o Session Storage.

```
function getStorage(useSessionStorage) // true|false
{
    var name = useSessionStorage ? "sessionStorage" :
        "localStorage";
    return (isStorageAvailable(name)) ? window[name] : null;
};
```

I dati (sia le chiavi sia i valori) vengono comunemente salvati nello storage come stringhe (sebbene in teoria la specifica ammetta qualsiasi tipo), per cui potrebbe risultare necessario effettuare dei cast in fase di estrazione. Serializzare in formato JSON i valori salvati nello storage potrebbe essere una soluzione ottimale in quanto, oltre a semplificare la conversione dei tipi primitivi come valori booleani, numeri e date, consente anche di gestire facilmente tipi complessi (object, array ecc.). [L'esempio 8.16](#) mostra un esempio di serializzazione di un oggetto in una stringa JSON, usando il metodo nativo `JSON.stringify` e la relativa deserializzazione mediante `JSON.parse`.

Esempio 8.16 – Scrivere e leggere dallo storage.

```
//recupera lo storage
var storage = getStorage();

//crea un utente, lo serializza in JSON e lo scrive nello storage
var user = { Id: 123, Name: "Stefano" };
var serializedUser = JSON.stringify(user);
storage.setItem("user", serializedUser);

//recupera l'utente dallo storage e lo deserializza
var storageItem = storage.getItem("user");
var user2 = JSON.parse(storageItem);
```

Se il quantitativo di dati memorizzati eccede la dimensione massima consentita dalle impostazioni dello Storage (generalmente di 5 MB, ma variabile a discrezione del browser o dell'utente), viene generato un errore di tipo `QUOTA_EXCEEDED_ERR` e l'operazione di scrittura fallisce. È interessante notare come in alcuni browser (come per esempio Chrome) la capacità reale di

archiviazione risulti dimezzata (2.5 MB), in quanto le stringhe vengono salvate con codifica UTF-16 (che richiede 2 byte per ogni carattere).

La possibilità di avere un sistema di persistenza lato client apre nuovi interessanti scenari nello sviluppo di applicazioni web: possiamo, per esempio, dotare un CMS (Content Management System) di una funzionalità per il salvataggio automatico dei dati immessi dall'utente, così da evitarne la perdita accidentale, oppure aggregare i dati da inviare al server, riducendo il numero di chiamate HTTP effettuate in un'applicazione che sfrutta fortemente AJAX, e molto altro ancora.

WebSocket

La possibilità di scambiare dati tra client e server senza dover ricaricare completamente la pagina è stata alla base del successo di AJAX. In molti casi viene effettuata puntualmente una chiamata al server a seguito di una specifica azione dell'utente, per esempio per l'invio asincrono di un form o di una richiesta per l'aggiornamento di una porzione di pagina. Altri scenari prevedono l'uso di tecniche di *polling* attraverso AJAX: la pagina, a intervalli regolari, effettua una richiesta automatica al server, per esempio, per sapere se è necessario notificare all'utente la presenza di una nuova email. In altri casi risulta invece necessario mantenere aperto un canale di comunicazione continua tra client e server, come nel caso di una chat in tempo reale. In questo tipo di applicazioni AJAX mostra i propri limiti e quindi è bene ricorrere a WebSocket, che consente la gestione di un canale di comunicazione bidirezionale tra il server e il client, costantemente aperto e gestito nativamente dal browser.

nota

Le specifiche complete di WebSocket API sono disponibili all'indirizzo <http://aspit.co/alp>.

Utilizzare WebSocket è molto semplice: è sufficiente, infatti, creare un'istanza di `WebSocket` specificando l'indirizzo del servizio con cui vogliamo comunicare attraverso il protocollo `ws://`. Una volta istanziato, l'oggetto apre automaticamente il canale con il server e una volta che questo è stabilito

scatena l'evento `onopen`. A questo punto possiamo inviare messaggi sfruttando il metodo `send` e intercettare i messaggi in arrivo attraverso la gestione dell'evento `onmessage`. Per chiudere la connessione dobbiamo invece invocare esplicitamente il metodo `close`, che genera l'evento `onclose` quando la chiusura del canale è effettiva. Questi metodi ed eventi sono mostrati nell'esempio 8.17.

Esempio 8.17 – Un servizio di chat con un WebSocket.

```
var output;
var websocket;

function onLoad()
{
    output = document.getElementById("output");

    websocket = new WebSocket("ws://echo.websocket.org/");
    websocket.onopen = function(evt) {
        write("CONNECTED");
    };
    websocket.onclose = function(evt) {
        write("DISCONNECTED");
    };

    websocket.onmessage = function(evt) {
        write("RESPONSE: " + evt.data +);
    };

    websocket.onerror = function(evt) {
        write("ERROR: " + evt.data);
    };
}

function doSend(message)
{
    write("SENT: " + message);
    websocket.send(message);
}

function doClose(){
    websocket.close();
}

function write(message)
{
    output.value += message + "\r\n";
}
```



```
window.addEventListener("load", onLoad);
```

L'esempio è molto semplice: al caricamento della pagina viene creato il WebSocket e viene aggiunto un delegato per ogni evento (i delegati scrivono in una `textarea` i dettagli dell'evento). Ci sono poi i metodi `doSend` e `doClose` che sono invocati dall'interfaccia utente quando dobbiamo rispettivamente inviare un messaggio e chiudere il canale.

... e molto altro ancora

Le API JavaScript non finiscono qui: HTML contiene infatti molte altre funzionalità che coprono diverse esigenze applicative. Oltre a quanto visto in questo capitolo, abbiamo interfacce per semplificare la gestione delle modalità di interazione tra utente e applicazione. Di seguito vi forniamo una lista delle altre API disponibili.

- drag and drop API;
- Speech API per il supporto di comandi vocali;
- Navigation Timing API per l'analisi delle performance;
- File API per permettere la lettura di file e cartelle in locale;
- Web Worker per permettere di eseguire codice multithread;
- Offline API per permettere all'applicazione di funzionare offline;
- IndexedDB per avere un motore di database locale.

Per brevità non ci è possibile parlare di tutte queste API in dettaglio ma questa lista fornisce un'idea di quanto siano enormi le potenzialità della piattaforma HTML. Quel che è importante considerare è che con HTML5 si riduce notevolmente la distanza tra le applicazioni native (desktop o mobile che siano) e quelle web.

Il futuro, analizzando gli investimenti e le ricerche di colossi come Google e Microsoft, sarà sempre più nel Cloud Computing e il browser smetterà di essere un semplice strumento di navigazione per diventare sempre più l'ambiente principale di lavoro.

Conclusioni

Le specifiche di HTML e JavaScript contengono un insieme di API volte a semplificare lo sviluppo di applicazioni web e a ridurre la necessità di plugin esterni per l'implementazione di funzionalità sempre più comuni o interessanti.

Nel corso di questo capitolo abbiamo visto come sia possibile usare JavaScript per manipolare il DOM, per realizzare applicazioni AJAX di nuova generazione, determinare la posizione di un utente, sfruttare i nuovi storage per la gestione dei dati in locale e creare pagine capaci di comunicare in modo semplice ed efficace con il server per lo scambio di informazioni.

Padroneggiare queste tecniche è importante quando si vogliono creare siti web moderni, accattivanti e sempre più usabili e potenti per l'utente. Nel prossimo capitolo ci occuperemo di un'altra funzionalità fondamentale per alcune categorie di applicazioni web: il multimedia.

Audio e video

Grazie alla sempre più ampia diffusione di Internet e alla disponibilità di banda, consumare contenuti multimediali è, ormai, all'ordine del giorno. In passato, questa operazione era possibile solo attraverso plug-in, come **Adobe Flash** o **Microsoft Silverlight**, che sono in grado di decodificare, anche con accelerazione hardware, i più diffusi formati audio e video disponibili. Con l'esplosione della navigazione da dispositivi mobile, su cui questi plug-in non sono disponibili, la necessità di riprodurre contenuti multimediali ha portato alla definizione di una serie di specifiche dedicate all'integrazione di contenuti multimediali direttamente all'interno delle pagine.

HTML5 viene incontro, quindi, a questa necessità, introducendo due nuovi tag, di nome `audio` e `video`, che possono essere usati per sviluppare un player multimediale, che possa sostituirsi quasi in tutto all'uso dei plug-in e che, soprattutto, funzioni su tutte le piattaforme, da quelle desktop a quelle mobile.

I tag audio e video

Come abbiamo anticipato, in HTML5 abbiamo a disposizione due nuovi tag molto simili nell'utilizzo, che differiscono per il fatto che uno, il tag `audio`, ha il compito di riprodurre un contenuto audio, mentre l'altro, il tag `video`, ha il compito di riprodurre video.

Contrariamente a quanto possiamo pensare, anche il tag `audio` presenta un'interfaccia, perché entrambi i tag possono avere la barra di controllo, per agire sul contenuto in ascolto o in visualizzazione.

Entrambi i tag vanno inseriti all'interno della pagina, nel punto in cui lo riteniamo opportuno, e rientrano nel DOM della stessa, interagendo e influenzando gli altri elementi presenti e potendo essere manipolati con JavaScript e stilizzati (nel limite e con differenze tra browser) con i CSS. [Nell'esempio 9.1](#) possiamo vedere come usare i due tag per inserire due

contenuti audio e video all'interno della nostra pagina.

Esempio 9.1

```
<h1>Audio</h1>
<audio src="test.mp3" controls />

<h1>Video</h1>
<video src="test.mp4" controls>
  Il tuo browser non supporta i video
</video>
```

L'attributo `src` ci permette di indicare con un URI (nell'esempio una risorsa locale) quale sia il contenuto multimediale da riprodurre. L'attributo `controls` indica di visualizzare i controlli standard offerti dal browser per la classica gestione di play, pause, seek e volume, di cui ogni contenuto multimediale normalmente necessita. Da notare, infine, che all'interno del tag possiamo indicare il markup da visualizzare nel caso il browser non dovesse supportare HTML5: in genere, viene inserito un comportamento di *fallback*, che include, per esempio, l'uso di Flash o Silverlight per quei (pochi) browser più vecchi che non dovessero avere il supporto per questi tag.

Per il tag `audio`, la mancata indicazione dell'attributo `controls` comporta generalmente l'avvio della traccia audio, mentre per il tag `video` la riproduzione non risulta controllabile, se non sfruttando il menu contestuale del browser. La barra di controllo è comunque visibile solo passando con il mouse sul video visualizzato, anche se questo comportamento cambia da browser a browser e non è facilmente controllabile.

Nella [figura 9.1](#) possiamo vedere il rendering standard offerto da Internet Explorer, utilizzato in questo caso perché supporta il formato MPEG-4.

Audio



Video



Figura 9.1 – Tag audio e video con i controlli predefiniti di IE.

Possiamo utilizzare altri attributi per controllare la riproduzione del contenuto: abbiamo a disposizione `autoplay`, per avviare automaticamente il contenuto non appena la pagina si avvia, o l'attributo `loop` per ripetere all'infinito il contenuto, che può ritornare utile, come [nell'esempio 9.2](#), per creare un suono di sottofondo continuo per la pagina.

Esempio 9.2

```
<audio src="test.mp3" autoplay loop />
```

Qualora l'attributo `autoplay` non fosse indicato, l'attributo `preload` istruisce il browser su come caricare il contenuto. Il valore predefinito è `auto` e indica che è a discrezione del browser caricare il contenuto in funzione del carico della rete o della previsione di prebuffering necessario per riprodurlo. Con `metadata`, invece, indichiamo che il browser deve limitarsi a caricare i primi byte del file, necessari a individuare le caratteristiche degli stream, la durata, la dimensione e il primo frame. Con `none`, infine, il browser capirà che non è necessario caricare immediatamente il file, così da ridurre il traffico da effettuare con il server.

Grazie ai metadata contenuti in ogni elemento multimediale, il browser è

in grado di ricavare, per esempio, la dimensione del video. Possiamo forzarla, impostando gli attributi `width` e `height`, anche se il browser manterrà, comunque, le proporzioni originali del video.

Quando l'attributo `autoplay` non è specificato, è compito del browser caricare il primo frame (se non diversamente specificato con l'attributo `preload`) e mostrarlo come rappresentazione dell'anteprima del video.

Nel caso in cui il file non sia ancora stato caricato sufficientemente per individuare il primo frame o l'attributo `preload` sia impostato su `none`, possiamo indicare un'immagine da caricare in sostituzione, come segnaposto del video che verrà caricato. [Nell'esempio 9.3](#) utilizziamo l'attributo `poster` per indicare l'immagine da applicare fino a quando l'utente non farà partire il video.

Esempio 9.3

```
<video src="test.mp4" poster="poster.png" preload="none" controls />
```

Ciò che otteniamo è visibile nella [figura 9.2](#), questa volta visualizzato in Google Chrome e con i relativi controlli.



Figura 9.2 – Poster applicato a un video in Google Chrome.

Infine, sempre sul tag `video`, abbiamo a disposizione l'attributo `audio` che, se impostato su `mute`, permette di azzerare il volume da applicare alla

riproduzione del video.

In generale, tutti gli aspetti classici legati a queste necessità possono essere facilmente modificati.

Ora che abbiamo presentato i due tag, possiamo analizzare quali sono i formati supportati e come possiamo fornire sorgenti in formati differenti.

Definire più sorgenti multimediali

I formati riproducibili con HTML5 sono il punto dolente dei nuovi tag, perché sono oggetto di disputa fra i produttori dei browser più diffusi presenti sul mercato. Lo standard, infatti, non ha l'obiettivo di definire quali sono i container e i codec ufficiali che i browser dovrebbero implementare e supportare, quanto il modo in cui questo tipo di funzionalità debba essere gestita.

La materia del contendere è, soprattutto, il formato da utilizzare nei video, dove a contendersi lo scettro ci sono **MP4** e **WebM** (nello specifico per i codec video H.264 e VP8). Il supporto dell'uno o dell'altro dipende dall'orientamento del produttore: WebM è spinto da vendor più orientati a codec open source (Google) mentre, viceversa, MP4 è preferito da vendor che preferiscono soluzioni più consolidate, ma coperte da brevetti (Microsoft, Apple e Firefox). In realtà, la questione è più complessa di così, perché Google Chrome su Windows, che ha già un decoder MPEG-4, supporta anche questo formato. In generale, MP4 è un formato che garantisce una buona diffusione, dal punto di vista dei video, così come lo è MP3 dal punto di vista dell'audio.

Ad ogni modo, abbiamo la possibilità di decidere quale formato supportare, oppure possiamo orientarci a supportare tutti i browser, preparando le risorse multimediali in più formati, che i browser scelgono in base a ciò che supportano. In alternativa all'uso dell'attributo `src`, visto negli esempi precedenti, possiamo usare l'elemento `source`, per definire una o più sorgenti, identificandole in base ai formati specificati: sarà il browser, poi, a decidere di riprodurre quello che è in grado di decodificare.

nota

Una risorsa multimediale viene generalmente inclusa in un contenitore, il quale viene identificato in base all'estensione del file, come .mp4, .mp3, .ogv, .mkv, .webm. Il contenitore comprende le

intestazioni e i dati riguardanti il numero dei flussi audio e video, le dimensioni, i sottotitoli e, più in generale, qualunque altra informazione che il contenitore supporti. Ogni flusso audio e video è memorizzato, a sua volta, secondo un codec, cioè un formato di compressione degli stream, che permette di risparmiare in termini di dimensione in byte del flusso e di ridurre le dimensioni del file finale.

Nell'esempio 9.4 abbiamo definito due sorgenti, una basata sul container OGG e l'altra basata su MP4: sarà il browser a scegliere quale riprodurre.

Esempio 9.4

```
<video controls>
  <source src="test.ogv"
    type='video/ogg; codecs="theora, vorbis"' />
  <source src="test.mp4"
    type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"' />
</video>
```

Come possiamo vedere, per ogni `source` indichiamo con l'attributo `src` l'URI al video e, facoltativamente, possiamo indicare con `type` il **MIME type** della sorgente, al fine di facilitare il browser nella scelta. Il MIME type, oltre a indicare il container, può essere seguito dall'informazione sui codec video e audio utilizzati nel file. Le sigle usate nell'esempio 9.4 rappresentano i profili di compressione (configurazioni di standard di compressione) che fanno variare la dimensione del file, la qualità di compressione e le capacità CPU/GPU richieste in fase di decoding.

Su device portatili o su PC ad alte prestazioni, specificare i profili di compressione usati permette al browser di determinare qual è il video più appropriato da scaricare e visualizzare per il dispositivo in uso. Per questo motivo, è caldamente raccomandato fornire anche queste informazioni. Nella [tabella 9.1](#) possiamo trovare alcuni esempi dei profili e dei relative MIME type che possiamo utilizzare.

<i>MIME type</i>	<i>Descrizione</i>
video/mp4; codecs="avc1.42E01E, mp4a.40.2"	H.264 Constrained baseline profile video (main and extended video compatible) level 3 and Low-

<code>mp4a.40.2"</code>	Complexity AAC audio in MP4 container.
<code>video/mp4; codecs="avc1.58A01E, mp4a.40.2"</code>	H.264 Extended profile video (baseline-compatible) level 3 and Low-Complexity AAC audio in MP4 container.
<code>video/mp4; codecs="avc1.4D401E, mp4a.40.2"</code>	H.264 Main profile video level 3 and Low-Complexity AAC audio in MP4 container.
<code>video/mp4; codecs="avc1.64001E, mp4a.40.2"</code>	H.264 'High' profile video (incompatible with main, baseline, or extended profiles) level 3 and Low-Complexity AAC audio in MP4 container.
<code>video/mp4; codecs="mp4v.20.8, mp4a.40.2"</code>	MPEG-4 Visual Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container.
<code>video/mp4; codecs="mp4v.20.240, mp4a.40.2"</code>	MPEG-4 Advanced Simple Profile Level 0 video and Low-Complexity AAC audio in MP4 container.
<code>video/webm; codecs="vp8, vorbis"</code>	VP8 video and Vorbis audio in WebM container.
<code>video/3gpp; codecs="mp4v.20.8, samr"</code>	MPEG-4 Visual Simple Profile Level 0 video and AMR audio in 3GPP container.
<code>video/ogg; codecs="theora, vorbis"</code>	Theora video and Vorbis audio in Ogg container.
<code>video/ogg; codecs="dirac, vorbis"</code>	Dirac video and Vorbis audio in Ogg container.

Tabella 9.1 – MIME type di esempio per una risorsa video.

Inoltre, la scelta della sorgente può essere suggerita con l'attributo `media`, da aggiungere sempre sul tag `source`. Attraverso le media query (utilizzate soprattutto con i CSS e introdotte nel [capitolo 5](#)) possiamo indicare quali condizioni devono essere soddisfatte per scegliere una sorgente piuttosto che un'altra. In aggiunta, all'indirizzo <http://aspit.co/a0r> possiamo trovare le specifiche e le numerose espressioni supportate da **CSS3** in maggior dettaglio.

Nell'esempio 9.5 utilizziamo la condizione `min-device-width` per impostare la scelta del file di nome `hi-res.mp4`, solo se la risoluzione del

dispositivo supera almeno 800 pixel in larghezza.

Esempio 9.5

```
<video controls>
  <source src="hi-res.mp4"
          media="screen AND (min-device-width: 800px)" />
  <source src="low-res.mp4" />
</video>
```

Quanto abbiamo visto finora è tutto basato su markup ma, essendo questi dei normali elementi della pagina, possiamo intervenire attraverso JavaScript per fornire funzionalità più avanzate.

Controllare la riproduzione da JavaScript

I tag `audio` e `video` presentano, oltre a specifiche di markup, anche le interfacce DOM che permettono ai linguaggi di scripting, come JavaScript, di manipolare e gestire la risorsa multimediale attraverso metodi, proprietà ed eventi.

Tra le ragioni che ci possono spingere all'uso dello scripting c'è sicuramente il fatto che la UI e l'esperienza dei controlli predefiniti dal browser non sono sempre sufficienti, oltre a variare da browser a browser.

Sicuramente, il fatto che il layout e i colori cambino a seconda del browser presenta un problema relativamente all'uniformità di rappresentazione del player multimediale, perciò ricorrere a JavaScript è quasi inevitabile.

Se poi vogliamo creare video che partono in base ad azioni dell'utente (come nei giochi, per esempio) la via dello scripting è l'unica percorribile.

Per creare un player, ad ogni modo, gli aspetti da controllare sono molteplici: in questo capitolo cerchiamo di affrontare quelli principali, così da raggiungere una buona personalizzazione.

L'obiettivo che vogliamo raggiungere è di fornire un pulsante di play, unito a un'etichetta che mostri la posizione corrente della riproduzione o gli eventuali stati di caricamento o errore. Cominciamo, quindi, con il definire il video, il pulsante e l'etichetta, come mostrato [nell'esempio 9.6](#).

Esempio 9.6

```
<video src="test.mp4" id="video" />
<div>
  <a href="#" id="play"></a> <span id="time" />
```

Il markup è essenziale, ma sufficiente per affrontare le tematiche principali. Abbiamo aggiunto un pulsante per far partire la riproduzione e un tag `span` per visualizzare la posizione del video.

La gestione dello stato e del caricamento

Il primo passo da effettuare consiste nel gestire gli stati e gli eventi relativi al contenuto multimediale, così da notificare all'utente se stiamo caricando il file o se ci sono problemi nella riproduzione.

Sull'interfaccia associata ai tag `audio` e `video` ci sono due proprietà fondamentali, di nome `networkState` e `readyState`, che ci danno, rispettivamente, le informazioni sullo stato di caricamento del file e sullo stato inerente al suo contenuto e alla quantità di informazioni che sono state caricate.

Definiamo, quindi, una funzione di inizializzazione di nome `initVideo`, che chiamiamo a caricamento del DOM avvenuto. In questa funzione, inoltre, intercettiamo con la funzione `addEventListener` di JavaScript gli eventi che influenzano le due proprietà prima menzionate, come [nell'esempio 9.7](#).

Esempio 9.7

```
function initVideo() {
    var video = document.getElementById('video');

    // Eventi relativi alla proprietà networkState
    video.addEventListener("loadstart", videoNetworkChanged, false);
    video.addEventListener("progress", videoNetworkChanged, false);
    video.addEventListener("suspend", videoNetworkChanged, false);
    video.addEventListener("emptied", videoNetworkChanged, false);
    video.addEventListener("stalled", videoNetworkChanged, false);

    // Eventi relativi alla proprietà readyState
    video.addEventListener("loadedmetadata", videoStateChanged,
false);
    video.addEventListener("loadeddata", videoStateChanged, false);
    video.addEventListener("waiting", videoStateChanged, false);
    video.addEventListener("playing", videoStateChanged, false);
    video.addEventListener("canplay", videoStateChanged, false);
    video.addEventListener("canplaythrough", videoStateChanged,
false);
}
```

Gli eventi sono molteplici, ma li gestiremo attraverso le due funzioni `videoNetworkChanged` e `videoStateChanged`, che troviamo definite nell'esempio 9.8.

Esempio 9.8

```
function videoNetworkChanged(e) {
    // NETWORK_EMPTY
    // NETWORK_IDLE
    // NETWORK_LOADING
    // NETWORK_NO_SOURCE
    if (e.target.networkState == e.target.NETWORK_LOADING)
        document.getElementById('time').innerHTML =
            "Caricamento...";
}

function videoStateChanged(e) {
    // HAVE_NOTHING
    // HAVE_METADATA
    // HAVE_CURRENT_DATA
    // HAVE_FUTURE_DATA
    // HAVE_ENOUGH_DATA
    if (e.target.readyState == e.target.HAVE_ENOUGH_DATA)
        document.getElementById('time').innerHTML = "Pronto";

    if (e.target.paused)
        document.getElementById("play").innerHTML = "Play";
    else
        document.getElementById("play").innerHTML = "Pausa";
}
```

Nelle due funzioni ci limitiamo a controllare le proprietà e a indicare, attraverso lo `span` con ID *time*, se è in corso il caricamento o se il video è pronto per la riproduzione. Dentro al codice, nei commenti, sono indicate le costanti che `networkState` e `readyState` possono assumere, così da poter gestire tutte le fasi del caricamento.

Un altro evento fondamentale da intercettare è `error`, scatenato nel caso ci siano problemi nel caricamento del contenuto multimediale, nella decodifica o nella riproduzione. Tramite la proprietà `error` possiamo, anche in questo caso, controllare le costanti e informare l'utente dell'errore. Nell'esempio 9.9 possiamo vedere la funzione che intercetta l'evento e mostra l'errore.

Esempio 9.9

```
// Evento di errore
```

```
video.addEventListener("error", videoError, false);

function videoError(e) {
    // MEDIA_ERR_ABORTED
    // MEDIA_ERR_NETWORK
    // MEDIA_ERR_DECODE
    // MEDIA_ERR_SRC_NOT_SUPPORTED
    document.getElementById("play").innerHTML =
        "Errore: " +
        e.target.error.code;
}
```

Gestita la fase di caricamento, non ci resta che occuparci della riproduzione del contenuto multimediale, vedendo come poter automatizzare anche questa parte attraverso l'uso di JavaScript.

Controllare la riproduzione

Avendo omesso i controlli predefiniti, l'anchor con ID *play* dell'esempio 9.6 è l'unico elemento che permette di avviare la riproduzione. Da JavaScript, i metodi `play` e `pause` permettono di avviare o sospendere la riproduzione dell'elemento multimediale, mentre la proprietà `paused` ci indica, tramite un boolean, se essa è in corso.

Prima di tutto occorre intercettare gli eventi `play` e `pause`, così da alterare la UI del pulsante di riproduzione. Nell'esempio 9.10 intercettiamo questi eventi, agganciandoli alla funzione `videoStateChanged`.

Esempio 9.10

```
// Eventi relativi alla proprietà pause
video.addEventListener("play", videoStateChanged, false);
video.addEventListener("pause", videoStateChanged, false);

function videoStateChanged(e) {
    var msg = '';
    if (e.target.readyState == e.target.HAVE_ENOUGH_DATA)
        msg = "Pronto";

    if (e.target.paused)
        msg = "Play";
    else
        msg = "Pausa";

    document.getElementById("play").innerHTML = msg;
}
```

}

Nell'esempio precedente abbiamo ridefinito, inoltre, la funzione `videoStateChanged`, per far sì che, in base alla proprietà `paused`, possiamo andare ad alterare il testo del pulsante: se è in corso la riproduzione, scriviamo "Pausa" mentre, se la riproduzione è ferma, scriviamo "Play".

Un altro aspetto che va gestito, solitamente, è la posizione e la durata della riproduzione corrente. Il nostro obiettivo è che il tag `span` con ID `time` contenga i secondi della posizione corrente nella timeline di riproduzione e i secondi totali del contenuto. Per raggiungere questo scopo, l'evento ideale è `timeupdate`, perché viene scatenato molte volte durante la riproduzione, dandoci la possibilità di aggiornare il testo da visualizzare. [Nell'esempio 9.11](#) possiamo vedere come, con poche righe, diventa possibile scrivere la posizione corrente.

Esempio 9.11

```
video.addEventListener("timeupdate", updateTime, false);

function updateTime(e) {
    var msg = e.target.currentTime.toFixed(1) +
        " | " + e.target.duration.toFixed(1) + " s";

    document.getElementById('time').innerHTML = msg;
}
```

Le proprietà `currentTime` e `duration` restituiscono, rispettivamente, la posizione corrente e la durata totale del contenuto, in secondi. Poiché non esiste una regola su quante volte l'evento dovrebbe essere scatenato e poiché ciò avviene anche più di una volta al secondo, ricorriamo alla funzione `toFixed` per arrotondare l'informazione al centesimo di secondo. Tra le altre proprietà che possono essere d'aiuto vale la pena citare:

- **seekable**: indica se il contenitore supporta lo spostamento della riproduzione, agendo su `currentTime`;
- **seeking**: indica se è in corso lo spostamento del punto di riproduzione;
- **muted**: restituisce o imposta un booleano per indicare se il volume è a zero;

- **volume**: restituisce o imposta il volume con valori che vanno da zero (minimo) a uno (massimo);
- **played**: restituisce il range dell'intervallo di tempo riprodotto;
- **playbackRate**: restituisce o imposta la velocità di riproduzione con valori `double`, dove uno rappresenta la velocità normale;
- **buffered**: restituisce l'intervallo di tempo precaricato.

A questo punto possiamo finalmente godere del risultato ottenuto, visibile nella [figura 9.3](#).



Figura 9.3 – Player multimediale basato su HTML5 e JavaScript.

Sebbene l'esempio sia semplificato e l'interfaccia non sia curata, possiamo capire che, grazie ai metodi, alle proprietà e agli eventi messi a disposizione, uniti alle potenzialità del markup, di SVG e di CSS, possiamo realizzare un player multimediale di tutto rispetto.

Utilizzare player multimediali già pronti

Il player alternativo sviluppato negli esempi precedenti non è certo completo al 100%. Mancano alcuni elementi, come la gestione del volume attraverso uno slider, il seek di riproduzione attraverso un altro slider oppure con

pulsanti di jump o, ancora, l'indicazione della percentuale di buffering del contenuto multimediale.

Queste funzioni non sono particolarmente difficili da implementare, ma spesso quello che necessitiamo è un player che soddisfi le esigenze più comuni, permettendoci di concentrarci semplicemente sugli aspetti grafici, così da adattarlo allo skin del sito che ospita il contenuto multimediale.

Esistono diversi player multimediali preconfezionati, che richiedono l'inclusione di script JavaScript e l'invocazione a una chiamata di setup. Vi sono molteplici soluzioni, gratuite e a pagamento, che elenchiamo nella [tabella 9.2](#).

<i>Nome</i>	<i>URI</i>
JW Player for HTML5	http://aspit.co/v5
MediaElement.js	http://mediaelementjs.com/
SublimeVideo	http://sublimevideo.net/
Video JS	http://videojs.com/

Tabella 9.2 – Alcuni player multimediali già pronti.

Per capire come possono essere utilizzati questi player già pronti, prendiamo in considerazione **MediaElement.js**. Questo interessante player offre anche funzionalità di fullscreen, backlight o controllo dei colori e richiede il download del plug-in di jQuery e dei loro script di nome `mediaelement-and-player.min.js`. [Nell'esempio 9.12](#) includiamo quindi questi script, insieme al CSS predefinito che applica lo skin al player.

Esempio 9.12

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript"
      src="jquery.min.js"></script>
    <script type="text/javascript"
      src="mediaelement-and-player.min.js"></script>
    <link rel="stylesheet" href="mediaelementplayer.min.css"
type="text/css" />
  </head>
```


L'uso del file CSS ci fa intuire che, con questo particolare player, possiamo personalizzarne gli aspetti grafici, pur mantenendone le funzionalità. Il CSS predefinito richiede, inoltre, alcune immagini, che troviamo nel pacchetto e dobbiamo distribuire insieme ai file sopra menzionati.

A questo punto dobbiamo definire il nostro video come [nell'esempio 9.3](#), aggiungendo lo script di setup del player.

Esempio 9.13

```
<body>
<video src="test.mp4" id="video">
</video>

<script type="text/javascript">
    $("#video").mediaelementplayer();
</script>
```

La funzione `mediaelementplayer` è un plug-in per jQuery, che permette di applicare automaticamente tutte le funzionalità. Il player è così configurato e possiamo vederne il risultato nella [figura 9.4](#).



Figura 9.4 – Un esempio di player multimediale basato su `mediaelement.js`.

Un altro aspetto interessante di questo player è il fatto che mette a disposizione anche un filmato in formato Flash e/o un'applicazione per Silverlight che, in caso di fallback per il mancato supporto di HTML5 da parte del browser,

permettono comunque all'utente di visualizzare il video sfruttando i plug-in.

Un altro aspetto che in HTML5 riveste un ruolo centrale è l'accessibilità dei contenuti. Diamo un'occhiata a come, con i tag `audio` e `video`, possiamo applicare facilmente sottotitoli e traduzioni.

Applicare sottotitoli e traduzioni

Durante la riproduzione di un contenuto multimediale, l'audio è costituito da musica, dialoghi e rumori. Purtroppo, non tutti gli utenti sono in grado di udire, perciò, come spesso siamo abituati a fare con il markup HTML, è bene prevedere la visualizzazione di sottotitoli, che permettano a un non udente di comprendere il video che sta visualizzando.

In modo simile, anche chi non ha difficoltà di questo tipo può scontrarsi con la mancata conoscenza della lingua dei dialoghi: anche in questo caso, può essere comodo fornire le traduzioni testuali, da mostrare come accompagnamento al video.

Per supportare queste e altre esigenze, le specifiche prevedono la possibilità di inserire uno o più tag `track`, per indicare contenuti testuali o metadati relativi al file multimediale. Per capirne l'utilizzo, ci affidiamo all'esempio 9.14, che indica un file esterno di nome `test.vtt` per i sottotitoli in lingua italiana.

Esempio 9.14

```
<video src="test.mp4" id="video">
  <track src="test.vtt" srclang="it"
        kind="subtitles" label="Italiano" default />
</video>
```

Possiamo notare l'utilizzo di alcuni attributi:

- **src**: per indicare l'URI al file contenente la traccia;
- **srclang**: per specificare la lingua, secondo la specifica RFC 5646, della traccia;
- **kind**: per indicare la tipologia della traccia, scegliendo tra `subtitles`, `captions`, `descriptions`, `chapters` e `metadata`;
- **label**: per indicare l'etichetta da associare alla traccia.

La specifica che stiamo utilizzando in questo esempio è chiamata **WebVTT**. Acronimo di Web Video Text Tracks, è un formato che deriva da SRT, già in uso da molti anni, ma che concilia alcune caratteristiche di formattazione proprie del Web. Questa specifica non è ancora stata approvata ed è ancora in fase di definizione, **ma è supportata praticamente da tutti i browser moderni**.

nota

Internet Explorer ignora i file serviti senza MIME type corretto. Per provare correttamente gli esempi, è necessario impostare tutti i MIME type. Nel caso di WebVTT, sarà necessario aggiungere nel file `.htaccess` (Apache) o `web.config` (IIS) il MIME type `text/vtt` per l'estensione `.vtt`. Questi esempi non funzioneranno con IE in mancanza di un web server.

Questo file è testuale e ha un'intestazione `WEBVTT`, seguita da uno o più *cue*, rappresentativi delle descrizioni e dei dialoghi che devono accompagnare il contenuto multimediale. Ogni *cue* ha un identificativo, l'indicazione della fascia temporale durante la quale il testo dev'essere visualizzato e il testo. [Nell'esempio 9.15](#) possiamo vedere un estratto del file `test.vtt`.

Esempio 9.15

WEBVTT

00:00:00.000 --> 00:00:10.000

Esempio di sottotitolo sfruttando il tag `<track>`;

00:00:10.000 --> 00:00:20.000

Sottotitolo con HTML5

Come possiamo notare, sono presenti due *cue*, dei quali il primo dev'essere mostrato nei primi 10 secondi, mentre il successivo dev'essere mostrato da 10 a 20 secondi. Il testo è codificato in HTML e può contenere i tag `b` o `i`. Sono supportate anche le possibilità di posizionare il testo e di allinearlo, ma queste specifiche sono ancora in draft e necessitano di ulteriore sviluppo.

Molti dei player custom che abbiamo introdotto in precedenza, inoltre, offrono un supporto a questo formato in maniera uniforme rispetto ai

browser, consentendoci di personalizzarne l'aspetto in maniera che sia omogeneo su tutti i browser.



Figura 9.5 – I sottotitoli visualizzati da Internet Explorer 11.

Essendo un normale tag HTML5, anche il tag `track` prevede eventi come `load`, `cuechanged` e `error`, per gestire il caricamento, il cambio del *cue* o l'eventuale errore di caricamento, oltre a proprietà come `cues` e `activeCues`, per conoscere le *cue* attuali e quelle attive.

Viste le tracce, un'altra caratteristica che possiamo sfruttare con il tag `video` è la combinazione dell'elemento con i CSS, per ottenere player più gradevoli dal punto di vista grafico.

Combinare i CSS con gli elementi multimediali

Abbiamo già anticipato nei capitoli precedenti come a ogni elemento che componga il DOM di una pagina possa essere applicato uno stile in modo esplicito, inline o con fogli di stile. Il tag `video` non è da meno e, se lo combiniamo con i CSS, possiamo allineare o applicare effetti sul box del video.

Senza entrare troppo nel dettaglio, possiamo comprenderne la facilità di utilizzo e apprezzare il risultato, analizzando le poche righe [dell'esempio 9.16](#). In questo esempio definiamo uno stile per il tag `video` e applichiamo alcuni degli stili di CSS3, gestendo l'arrotondamento dei bordi.

Esempio 9.16

```
<head>
  <style type="text/css">
    video
    {
      box-shadow: 10px 10px 5px #888;
      border-top-left-radius: 60px 90px;
      border-bottom-right-radius: 60px 90px;
      border: 12px solid blue;
    }
  </style>
</head>
<body>
  <video src="test.mp4" controls>
  </video>
</body>
```

Nell'esempio precedente applichiamo un'ombreggiatura al `box video`, arrotondando gli angoli e contornando il tutto con una linea blu. Il risultato è visibile nella [figura 9.6](#).



Figura 9.6 – Il tag video con un CSS applicato, che ne cambia lo stile.

In generale, gli attributi che possiamo applicare sono gli stessi dei contenitori, come i `div`, ma purtroppo, anche in questo caso, il supporto dei browser non è sempre completo e in certe situazioni un comportamento standard non è ancora stato definito, costringendoci, come spesso avviene, ad avere la

certezza della resa di un particolare tag, semplicemente provando la pagina sui vari browser disponibili.

A questo punto è lecito chiederci se questi attributi siano sufficienti a sostituire l'uso di plug-in. Proviamo quindi a dare una risposta a questa domanda.

Le limitazioni dei tag audio e video

All'inizio del capitolo abbiamo introdotto questi due nuovi tag, anticipando che il loro obiettivo è quello di soddisfare una delle esigenze più diffuse sul Web. Ci sono, però, delle limitazioni che dobbiamo conoscere (che potrebbero essere superate in futuro, con la maturazione degli standard), che per il momento fanno pendere la bilancia, in certi scenari, ancora a favore dei plug-in, come Adobe Flash e Microsoft Silverlight.

Al di là dei formati e dei codec, su cui, come abbiamo visto, la disputa è ancora aperta, uno dei limiti più importanti è l'impossibilità di fare **live streaming**. I contenitori previsti supportano solo flussi audio e video limitati e devono essere, quindi, preparati prima di essere riprodotti. Di conseguenza, non è supportato anche lo scenario in cui sia richiesto un **adaptive streaming**, cioè la capacità del client e del server di adattarsi in funzione dei carichi della rete, diminuendo o alzando il bitrate del flusso, per evitare il più possibile l'interruzione della riproduzione. Resta possibile, comunque, usare tecnologie server che permettano di restituire i byte ai client in funzione della riproduzione, al fine di ridurre al minimo il precaricamento da parte del browser e ottimizzare la banda consumata.

In realtà, il formato Dynamic Adaptive Streaming over HTTP (meglio noto come **MPEG-DASH**) è stato ratificato e attualmente è supportato da molti browser. Il supporto a MPEG-DASH esiste su molti browser (IE, Chrome, Firefox e Opera), mentre Apple supporta una tecnologia di propria creazione, chiamata HTTP Live Streaming (HLS), che però ha il limite di non essere una soluzione standard e di non godere di supporto al di fuori delle piattaforme del vendor che le ha implementate.

In entrambi i casi, occorre avere un server che sia in grado di fornire contenuti in questo formato. Un esempio di uso si può trovare su <http://aspit.co/alt>.

Sempre in merito a questi scenari, per quanto riguarda il **Digital Rights Management** (DRM), che permette di proteggere i contenuti, abbiamo a disposizione le specifiche EME (Encrypted Media Extensions), che al

momento sono abbracciate da Microsoft, Google, Mozilla e Apple.

Senza DRM, i file possono essere liberamente riprodotti e scaricati, senza limiti. I browser stessi permettono di salvare il file o copiarne l'indirizzo tramite un menu contestuale.

Infine, per quanto concerne la riproduzione in fullscreen attraverso JavaScript, occorre affidarsi alle Fullscreen API, le cui specifiche sono disponibili su <http://aspit.co/alu>.

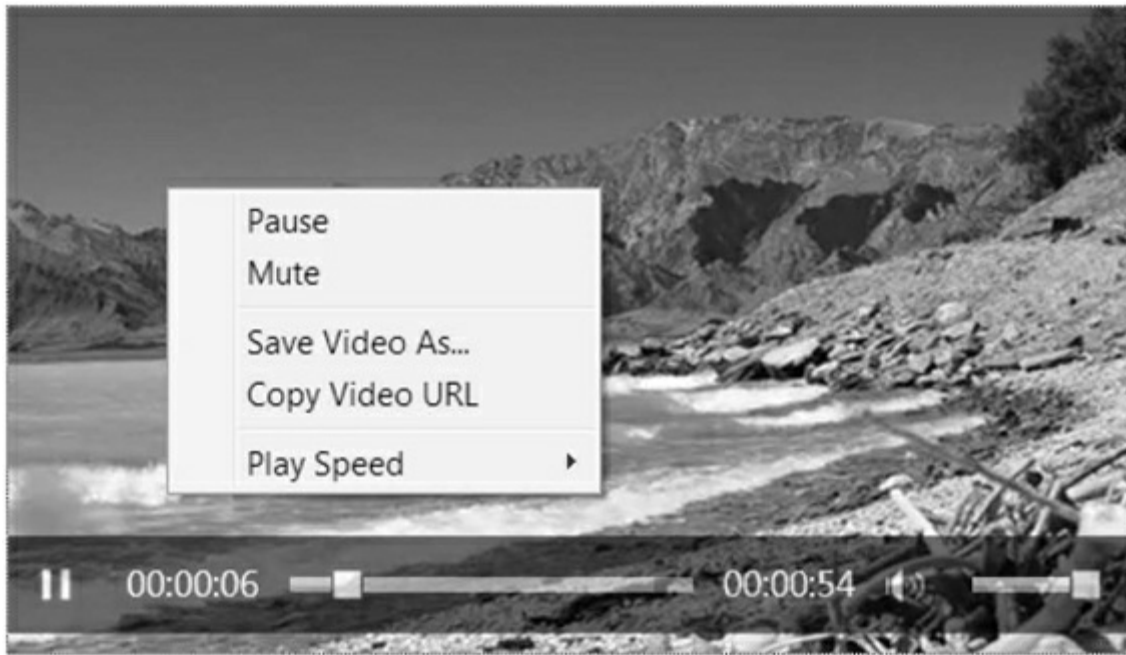


Figura 9.7 – Il menu contestuale sul tag video, che consente anche di salvare localmente il file riprodotto.

Conclusioni

In questo capitolo abbiamo affrontato, in modo specifico, due nuovi tag di HTML5: `audio` e `video`. Questi due nuovi elementi ci danno la possibilità, seppur con alcune limitazioni, di riprodurre contenuti multimediali, supportando più formati e risoluzioni video, a seconda del dispositivo che deve riprodurli, e garantendoci un buon supporto nell'ambito dei device di tipo mobile.

Abbiamo visto come, con i linguaggi di scripting, possiamo aggiungere funzionalità che vadano oltre la barra di controllo messa a disposizione dal browser e quale codice serva per conoscere le fasi di caricamento e della riproduzione, per alterare la posizione o il volume di riproduzione.

Grazie alle tracce, abbiamo analizzato la capacità di offrire contenuti

multimediali accessibili, che siano in grado di fornire sottotitoli e traduzioni attraverso un file esterno, che i player sanno leggere e mostrare nella pagina.

Infine, abbiamo affrontato la possibilità di conciliare il tag `video` con l'uso dei CSS e come superare alcuni limiti.

Continuando a trattare le novità legate ai contenuti grafici, nel prossimo capitolo daremo uno sguardo a SVG e Canvas, utili per aggiungere elementi di tipo grafico più avanzati alle nostre applicazioni.

Grafici con Canvas e SVG

Nei capitoli precedenti abbiamo affrontato sostanzialmente caratteristiche legate al markup e all'indicazione tramite tag delle tipologie di contenuti; abbiamo visto come, mediante stili e CSS, possiamo personalizzarne l'aspetto in molteplici modi per dare forma al layout che desideriamo per il nostro sito internet, e come JavaScript aggiunga l'interattività.

Tutto questo, però, può non bastare, perché il testo e il CSS possono non riuscire a fornire informazioni e allo stesso tempo dare un look piacevole o significativo. Per colmare questa mancanza si ricorre spesso all'uso di immagini in formato JPEG o PNG ma, con **HTML5**, sono state introdotte due nuove tecnologie che ci permettono di realizzare elementi grafici sia con un approccio vettoriale, tramite **SVG**, sia con un approccio orientato alle bitmap, tramite **canvas**.

L'introduzione di ulteriori elementi di markup si è resa quindi necessaria per eliminare i difetti delle immagini compresse. Con SVG possiamo infatti ottenere immagini che non perdono mai di qualità, mentre con il `canvas` disponiamo di una superficie di disegno dove lavorare liberamente attraverso scripting, godendo di ottime prestazioni.

In questo capitolo daremo quindi un'infarinatura su queste due tecnologie, applicandole a esempi pratici, così da capire quali possano essere la loro utilità e gli ambiti di applicazione.

Grafica vettoriale con SVG

Oltre alle immagini bitmap, rappresentate principalmente dai formati compressi come JPEG, GIF e PNG, ci sono le immagini vettoriali, la cui caratteristica principale è di non essere formate da un insieme di pixel, bensì da informazioni che ne determinano l'aspetto, una volta effettuato il rendering.

In pratica, se una linea rossa è in una bitmap un insieme di n pixel,

ognuno dei quali formato solitamente da 32bit per i rispettivi quattro canali di colore (alpha, red, blue, green), in un'immagine vettoriale è presente un'informazione (a seconda della tipologia di file) che indica una linea di colore rosso che va dalle coordinate x_1, y_1 a quelle x_2, y_2 .

Questa sostanziale differenza offre un importante beneficio: possiamo trasformare l'immagine **senza perdita di qualità**. Una bitmap, invece, richiede algoritmi più o meno efficienti per ricostruire i pixel e fornire un'immagine scalata: in genere, queste operazioni determinano sbavature e perdita di definizione.

In un'immagine vettoriale, quindi, non è la dimensione della figura a determinare il peso del file, ma la complessità della figura e l'efficienza del linguaggio utilizzato. In molti contesti, infatti, un'immagine vettoriale è più leggera rispetto a una bitmap.

Il formato SVG, acronimo di **Scalable Vector Graphics**, è stato quindi creato per fornire uno standard per la rappresentazione di immagini vettoriali, non solo per l'uso destinato al Web. La prima versione definita dal **W3C** è del 2001 e non ha subito significativi cambiamenti, ma piuttosto, come spesso succede con gli standard, un'integrazione della documentazione per chiarire meglio alcuni comportamenti che i viewer (visualizzatori in grado di supportare SVG) devono mantenere.

L'attuale versione è la **1.1 Second Edition** e al momento della stesura del libro è in draft la versione 2.0, ma è ancora acerba e non supportata da nessun browser.

Per iniziare, creiamo un file con estensione `.svg` e inseriamo il markup [dell'esempio 10.1](#), che rappresenta un rettangolo rosso.

Esempio 10.1

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
    version="1.1"
    width="200"
    height="100">
  <rect fill="red" x="10" y="10" width="180" height="80" />
</svg>
```

Dall'esempio possiamo subito notare alcune caratteristiche:

- il documento è basato sulla sintassi XML e ne osserva le regole;

- la root di un documento è il tag `svg`, che determina l'area dell'immagine attraverso gli attributi `width` e `height`;
- il documento SVG contiene gli elementi di disegno e di testo che rappresentano l'immagine;
- i tag, in modo simile all'HTML, sono auto descrittivi e rendono comprensibile il markup, anche aprendo il file con un editor di testo.

L'uso della sintassi XML fa sì che il documento possa essere trattato con molteplici strumenti, come designer dedicati alle immagini vettoriali, come Adobe Illustrator, Corel Draw o Open Office, oppure strumenti in grado di esportare in SVG, come Microsoft Visio.

Sempre grazie alla sintassi XML, un documento SVG gode degli stessi benefici di una pagina HTML, perché può essere indicizzato dai motori di ricerca (cosa che i più diffusi bot già fanno) e dà significato agli elementi, poiché questi possono avere un id, contenuti testuali e metadati.

Un file SVG può essere compresso attraverso **gzip**, adatto per stream testuali, e si identifica con l'estensione `.svgz`. Con HTML5, inoltre, un documento SVG può essere inserito all'interno di una pagina, permettendo di mischiare markup rappresentativo di informazioni con markup dedicato alle figure, senza che questo debba rispettare obbligatoriamente la sintassi XML, ma semplicemente le specifiche SGML, come in HTML5.

Nell'esempio 10.2 possiamo vedere due rettangoli inseriti all'interno di una pagina che contiene anche un tag `h1`.

Esempio 10.2

```
<!DOCTYPE html>
<html>
  <head>
    <title>Esempio</title>
  </head>
  <body>
    <h1>SVG</h1>
    <svg viewBox="0 0 100 100">
      <rect fill="red" width="50" height="50" />
      <rect fill="gray" x="25" y="25" width="50" height="50" />
    </svg>
  </body>
</html>
```

Il tag `svg` rappresenta quindi la figura ed è sottoposto alle regole di layout come qualsiasi altro elemento presente nella pagina. Può essere disposto come blocco, allineato, influenzato dagli altri elementi e, facoltativamente, possiamo ometterne le dimensioni. [Nell'esempio 10.2](#), infatti, specifichiamo il `viewBox`, cioè il rettangolo (x, y, w, h) rappresentativo della zona da visualizzare. I due rettangoli occupano un'area di 100x100 pixel e questo permette alla figura di non essere influenzata nelle sue misure dall'esterno. Impostando una dimensione con gli attributi `width` o `height`, otterremo un'immagine che, forzatamente, viene scalata proporzionalmente per riempire l'area. [Nell'esempio 10.2](#), omettendo le dimensioni, otteniamo una figura che va a riempire tutto lo spazio allocato dopo il titolo, cioè l'intera pagina, come è visibile nella [figura 10.1](#).

SVG



Figura 10.1 – Due rettangoli SVG all'interno di una pagina HTML.

Da notare, inoltre, che l'ordine della definizione degli elementi determina lo `z-index` degli stessi: nella [figura 10.1](#) possiamo notare che il secondo rettangolo si pone davanti al primo.

Per incorporare un file `.svg` esterno, come facciamo con immagini in formato JPEG e PNG, possiamo utilizzare il tag `object`, specificando per l'attributo `type` il valore `image/svg+xml`, come [nell'esempio 10.3](#).

Esempio 10.3

```
<body>
```

```
<h1>SVG</h1>
<object data="1.svg" type="image/svg+xml">
  SVG non supportato
</object>
</body>
```

Ora che sappiamo come dichiarare e scrivere un SVG, vediamo quali sono gli elementi più comuni che possiamo usare. Sebbene sia più produttivo impiegare strumenti di disegno e, per certi versi, potremmo ignorarne le specifiche, è utile conoscere (almeno in termini generici) cosa serve per generare immagini che non siano statiche, ma che possano essere dinamiche.

Infatti, attraverso elaborazioni server con strumenti come [ASP.NET](#), PHP o Java, possiamo generare del markup SVG secondo informazioni ottenute da servizi o da database, come già oggi facciamo con l'HTML.

Cominciamo quindi dal capire quali sono le primitive di disegno che possiamo utilizzare.

Disegnare con gli shape

Gli shape sono le primitive di disegno con le quali possiamo rappresentare una figura e, molto spesso, richiedono una combinazione tra di loro per arrivare a generare un'immagine che possa diventare abbastanza realistica da avvicinarsi a una bitmap. Ricordiamoci, comunque, che non è certamente l'intento di SVG quello di poter rappresentare una fotografia.

Abbiamo già conosciuto il tag `rect`, che mediante le proprietà `x`, `y`, `width` e `height` permette di indicare la posizione dell'angolo sinistro e le dimensioni del rettangolo. Le unità di misura supportate da SVG sono le medesime di CSS e HTML5, perciò un valore "5" significa 5 pixel, ma possiamo specificare molteplici unità di misura: `pt`, `pc`, `mm`, `cm`, `in` e `%`.

SVG supporta gli elementi di disegno classici, come ellissi, cerchi o poligoni. Nella [tabella 10.1](#) abbiamo riportato i principali tag, con alcune note sull'utilizzo degli attributi.

Elemento	Descrizione	Attributi
<code><rect></code>	Disegna un rettangolo, eventualmente con i vertici arrotondati.	<code>x</code> , <code>y</code> , <code>width</code> e <code>height</code> per dimensionare; <code>rx</code> e <code>ry</code> per arrotondare i vertici.

<code><circle></code>	Disegna un cerchio.	<code>cx</code> e <code>cy</code> per indicare il centro; <code>r</code> per indicare il raggio.
<code><ellipse></code>	Disegna un ellisse.	<code>cx</code> e <code>cy</code> per indicare il centro; <code>rx</code> e <code>ry</code> per indicare il raggio sugli assi delle <code>x</code> e delle <code>y</code> .
<code><line></code>	Disegna una linea.	<code>x1</code> e <code>y1</code> per il punto di inizio; <code>x2</code> e <code>y2</code> per il punto di fine.
<code><polyline></code>	Disegna un insieme di linee, tutte congiunte tra di loro.	<code>points</code> per indicare una lista di punti con la coppia (<code>x,y</code>), separata dallo spazio.
<code><polygon></code>	Disegna un insieme di linee, tutte congiunte tra di loro. Crea sempre una figura chiusa ricongiungendo l'ultimo punto con il primo.	<code>points</code> per indicare una lista di punti con la coppia (<code>x,y</code>), separata dallo spazio.
<code><path></code>	Disegna una figura complessa basandosi su elementi più semplici, eventualmente chiudendola.	<code>d</code> per indicare dei comandi di disegno separati dallo spazio.

Tabella 10.1 – Gli shape disponibili con SVG e i loro principali attributi.

Nell'esempio 10.4 possiamo vedere un caso d'uso di alcuni di questi elementi, in cui procediamo a definire un rettangolo con i vertici arrotondati, un cerchio e un'ellisse.

Esempio 10.4

```
<svg width="400" height="300">
  <rect fill="gray"
    x="2" y="2"
    rx="20" ry="50"
```

```

        width="200" height="100" />
<circle fill="red"
        cx="300" cy="50" r="50" />
<ellipse fill="yellow"
        cx="100" cy="180"
        rx="100" ry="50"/>
</svg>

```

Nella [figura 10.2](#) possiamo vedere ciò che otteniamo visualizzandolo nel browser.



Figura 10.2 – Rettangolo, cerchio ed ellisse in SVG.

Con [l'esempio 10.5](#) vediamo invece come usare gli elementi dedicati alle linee, creando una linea, un poligono che forma un triangolo e un multi linee.

Esempio 10.5

```

<svg width="400" height="300">
  <line stroke="red"
        stroke-width="3"
        x1="10" y1="5"
        x2="100" y2="50" />
  <polygon fill="red"
        stroke="blue"
        stroke-width="1"
        points="150,20 300,80 120,100"/>
  <polyline points="10,190 20,190 20,120 50,120 50,190 60,190"

```

```
stroke="green"  
stroke-width="4"  
fill="blue" />
```

```
</svg>
```

Nella [figura 10.3](#) possiamo vedere ciò che otteniamo.

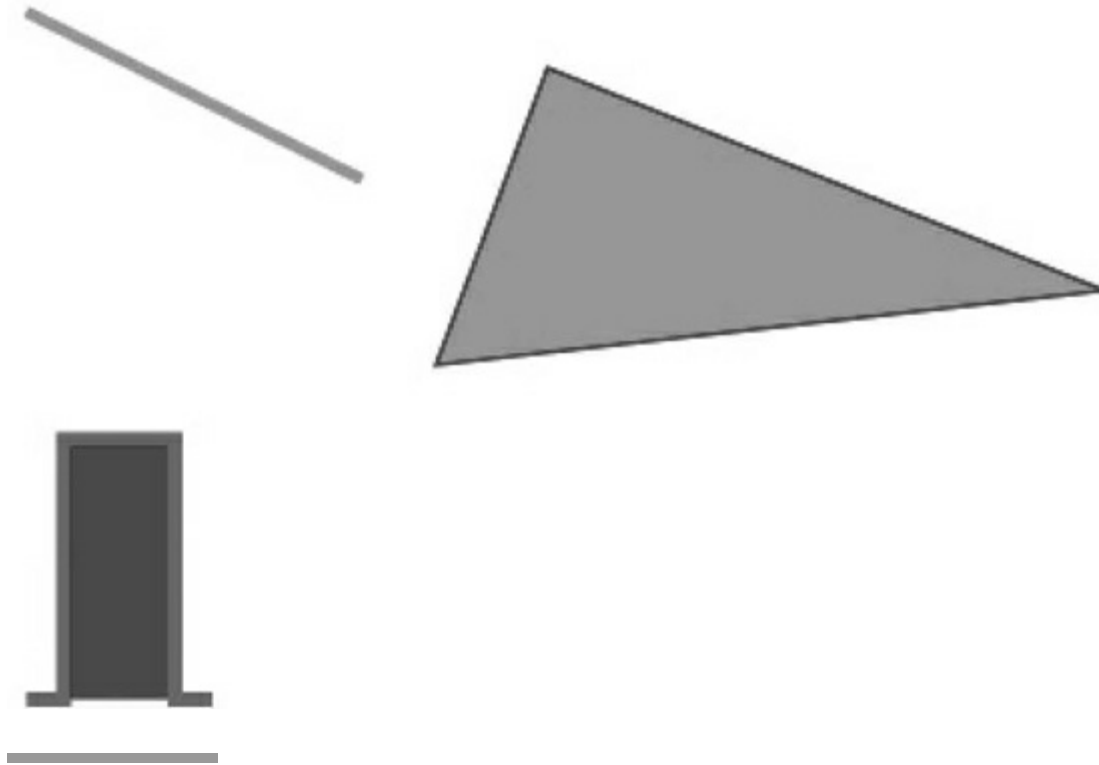


Figura 10.3 – Linea e poligoni ottenuti con SVG.

Dal markup contenuto negli esempi 4.4 e 4.5 possiamo subito notare l'uso degli attributi `fill`, `stroke` e `stroke-width`, che servono per indicare, rispettivamente, il colore di riempimento, il colore della linea o del contorno e lo spessore di quest'ultimo. Anche in questo caso, la sintassi si rifà a quella delle specifiche CSS, perciò possiamo valorizzare tali attributi con nomi dei colori, esadecimali o RGB.

L'elemento `path`, infine, è il tag più utilizzato, perché molto versatile: ci permette di disegnare una figura con dei comandi che indicano a un'immaginaria penna di spostarsi, disegnare una linea e produrre un risultato. L'attributo `d`, infatti, ci permette di indicare una serie di comandi, raccolti nella [tabella 10.2](#), ognuno dei quali lavora in funzione del comando precedente.

Comando	Descrizione	Parametri
---------	-------------	-----------

M	Sposta la penna in un punto, senza disegnare.	x, y
L	Disegna una linea dal punto attuale a quello indicato.	x, y
H	Disegna una linea orizzontale dal punto attuale alla x indicata.	x
V	Disegna una linea verticale dal punto attuale alla y indicata.	y
C	Disegna una curva di bezier verso il punto x,y dove i punti x1,y1 e x2,y2 sono il punto di inizio e di fine.	x1, y1 x2, y2 x, y
S	Disegna una curva di bezier verso il punto x,y dove x2,y2 è il punto di inizio e di fine.	x2, y2 x, y
Q	Disegna una bezier quadratica tra il punto x,y e x1,y1.	x1, y1 x, y
T	Disegna una bezier quadratica tra il punto x,y e la posizione attuale.	x, y
A	Disegna un arco dal punto attuale verso x,y.	x,y
Z	Chiude la figura con una linea dal punto attuale al primo punto.	

Tabella 10.2 – I comandi messi a disposizione dall'elemento path.

Per capire il suo utilizzo, [nell'esempio 10.6](#) vediamo come disegnare un grafico a torta formato da tre path, ognuno dei quali sposta la penna in un punto, disegna due linee, traccia una curva e chiude la figura.

Esempio 10.6

```
<svg width="500" height="450">
  <path fill="blue"
    d="M 310,402 L 213,214 L 326,35 C 454,111 468,319 310,402
      z" />
  <path fill="red"
    d="M 3,208 L 213,214 L 326,35 C 202,-46 8,30 3,208 z"/>
  <path fill="yellow"
    d="M 3,208 L 213,214 L 311,400 C 177,467 3,380 3,208 z" />
</svg>
```

Il risultato che otteniamo è una torta, visibile nella [figura 10.4](#)



Figura 10.4 – Grafico a torta ottenuto con SVG.

Se non avete ben chiaro quello che abbiamo appena visto, non temete: raramente dovrete procedere a ricostruire voi gli elementi e, molto più facilmente, potete contare su tool e librerie che vi aiuteranno in tal senso.

Ora che abbiamo visto come disegnare manualmente gli elementi, vediamo come aggiungere del testo al documento.

Inserire elementi testuali

All'interno di una figura, spesso, possiamo avere necessità di informazioni testuali: sebbene possano essere raffigurate con degli shape, non avrebbero significato dal punto di vista del markup, non potrebbero essere selezionate come testo dall'utente e non sarebbero indicizzate (come avviene di norma con le immagini bitmap).

Per questo motivo, SVG offre un tag `text`, che al suo interno può contenere del testo formattato e dimensionato. Rispetto all'HTML5, inoltre,

questo tag permette di cambiare le posizioni e la rotazione di ogni carattere.

Vediamo quindi come utilizzare questo elemento, usando alcune delle caratteristiche principali di questo tag.

Esempio 10.7

```
<svg width="200" height="150">
  <text x="10" y="20"
        font-family="Verdana"
        font-size="30">
    dx="2,0,0" dy="8,-6,-3"
    rotate="-25,0,25">
    SVG
  </text>
</svg>
```

Nell'esempio 10.7 ci sono molteplici attributi in uso su cui vale la pena soffermarci:

- `x` e `y` sono utilizzati per indicare le coordinate dell'angolo basso/sinistro da cui partire;
- `font-size` e `font-family` vengono sfruttati per indicare la dimensione e il tipo di carattere;
- `dx` e `dy` si usano per indicare a ogni lettera di quanti pixel spostarsi, in `x` e in `y`;
- `rotate` indica, infine, di quanto deve ruotare ogni lettera.

In pratica, con il markup visto in precedenza, indichiamo che la lettera "S" deve essere spostata rispetto al punto di inizio di 2,8 pixel e ruotata di -25 gradi; la "V", invece, viene alzata di 6 pixel, mentre la G alzata di 3 pixel e ruotata di 25 gradi. Così facendo otteniamo la scritta visibile nella [figura 10.5](#).



Figura 10.5 – Testo con manipolazione dei caratteri ottenuta con SVG.

L'aspetto interessante è che il testo è selezionabile, con tutte le operazioni che ne derivano. Molteplici sono gli aspetti che possiamo controllare, come la spaziatura e le decorazioni: purtroppo, per motivi di spazio, non possiamo trattarli tutti in questa sede, perciò vi invitiamo a visionare le specifiche complete alla pagina <http://aspit.co/a03>.

I colori si possono controllare con gli attributi (già visti) `fill`, `stroke`, e così via; lo `stroke`, in particolare, rappresenta il colore del contorno di ogni carattere.

Sebbene abbiamo la possibilità di inserire testo, questo non va confuso con le caratteristiche tipiche di un documento HTML5. Per esempio, non possiamo fare il wrap del testo, sebbene, con caratteristiche simili allo `span`, possiamo inserire come contenuto dell'elemento `text` una lista di `tspan` che ci permetta di raggruppare del testo, dando posizioni e aspetti differenti.

Nell'esempio 10.8 il testo viene diviso in tre parti consecutive ma allineate verticalmente più in basso.

Esempio 10.8

```
<text font-family="Verdana"
      font-size="30"
      y="30">
  <tspan>Esempio</tspan>
  <tspan dy="20">di</tspan>
  <tspan dy="20" font-weight="bold">TSPAN</tspan>
</text>
```

Come vediamo nella [figura 10.6](#), lo spostamento di un `tspan` in verticale determina che quello successivo continui sulla stessa riga.

Esempio
di
TSPAN

Figura 10.6 – Testo con `tspan` allineati su righe differenti.

In generale, il tag `text` è sottoposto alle stesse problematiche dell'HTML5. I font che possiamo utilizzare devono essere tra quelli previsti o già installati sulla macchina dell'utente. In alternativa, è presente una specifica di nome

SVG Font, che permette di creare glyph e, di fatto, creare font con SVG stesso, per poi utilizzarli. Questa specifica è comunque in disuso a favore di un altro standard, esclusivamente pensato per i font, di nome **Web Open Font Format** (WOFF), che permette di distribuire font e di renderli utilizzabili con HTML5, e che abbiamo approfondito in maniera specifica nei capitoli dedicati ai CSS. Visto come possiamo inserire elementi testuali, vediamo come SVG ci permette di organizzare la struttura del documento al fine di agevolare il disegno e la manutenzione.

Raggruppare e riutilizzare gli elementi

Abbiamo anticipato che i documenti SVG possono diventare complessi, perché possono richiedere molteplici shape per rappresentare immagini vettoriali che vadano oltre gli esempi affrontati finora.

Una delle tecniche più usate, soprattutto negli ambienti di disegno, è quella di raggruppare gli elementi. In SVG possiamo farlo attraverso l'elemento `g`, il quale rappresenta un insieme logico di elementi, ma permette anche di agire su di essi impostando posizioni, dimensioni e colori di riempimento da applicare a tutti gli elementi.

Con l'[esempio 10.9](#) possiamo capirne meglio l'utilità: abbiamo definito un ipotetico logo, con rettangolo e testo, senza specificare i colori di riempimento e delle linee. Queste due informazioni sono definite, invece, a livello di gruppo, così da agire automaticamente su tutti i figli.

Esempio 10.9

```
<svg width="300" height="200">
  <g id="SVGLogo" fill="red" stroke="black">
    <rect width="66" height="50" />
    <text font-family="Verdana" font-size="30"
      y="36" stroke-width="1">
      SVG
    </text>
  </g>
</svg>
```

L'altro aspetto interessante del raggruppamento è che, dandogli opportunamente un nome con l'attributo `id`, possiamo riutilizzarlo più volte attraverso l'elemento `use`.

Questo tag permette di riutilizzare e riposizionare gli elementi,

eventualmente applicando trasformazioni, il tutto facendo riferimento al nodo attraverso **XLink**.

Questo standard permette di collegare risorse (interne ed esterne) con l'attributo `href`, specificando anche informazioni come titolo e tipologia. [Nell'esempio 10.10](#) possiamo vedere come utilizzare questo standard per linkare il gruppo in base al suo `id`.

Esempio 10.10

```
<svg xmlns:xlink="http://www.w3.org/1999/xlink">
  <g id="SVGLogo" fill="red" stroke="black">
    <rect width="66" height="50" />
    <text font-family="Verdana"
          font-size="30" y="36" stroke-width="1">
      SVG
    </text>
  </g>
  <use x="66" y="50" xlink:href="#SVGLogo" />
</svg>
```

Così facendo, otteniamo due loghi, come possiamo vedere nella [figura 10.7](#).

In modo simile a `g`, l'elemento `defs` permette di definire degli elementi come se fossero delle risorse, visualizzabili solo usando il tag `use`. Sono utilizzati in modo particolare con i gradienti e, in generale, per quegli elementi che da soli non rappresentano qualcosa, ma devono obbligatoriamente essere usati da un altro elemento.

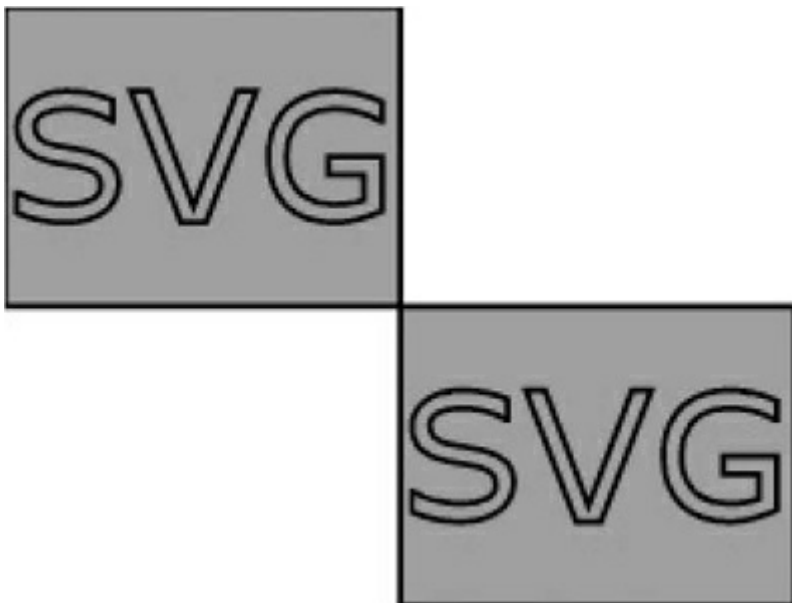


Figura 10.7 – Riutilizzo di un gruppo mediante l'elemento `use`.

Colorare con gradienti e pattern

In realtà possiamo usare colori di riempimento più evoluti, tramite gradienti e pattern. Attraverso gli elementi `linearGradient` e `radialGradient` possiamo definire dei gradienti lineari o radiali. La loro definizione è molto simile a quella che troverete in altri linguaggi, perché si basa sugli stop, cioè segnaposti di colore tra i quali effettuare una transizione.

Come anticipato nel paragrafo precedente, vanno definiti con `defs` per poter essere poi usati con gli attributi `fill` e `stroke`, sempre tramite XLink, come mostrato nell'esempio 10.11, che crea un gradient dal grigio al blu.

Esempio 10.11

```
<defs>
  <linearGradient id="grad" x1="0" y1="0" x2="1" y2="1">
    <stop offset="0" stop-color="gray" />
    <stop offset="1" stop-color="blue" />
  </linearGradient>
</defs>
<rect fill="url(#grad)" width="50" height="20" />
```

Gli attributi `x1,y1` e `x2,y2` indicano il punto di inizio e di fine di un ipotetico rettangolo 1x1 (oppure utilizzando una misura in percentuale) per stabilire la direzione del gradiente. Nell'esempio 10.11 tracciamo una diagonale tra l'angolo in alto a sinistra e quello in basso a destra. I tag `stop` definiscono con `offset` il punto con valori da 0 a 1 in cui passare al nuovo colore rispetto alla diagonale tracciata.

Da notare, inoltre, l'uso della funzione `url` di XLink, utilizzabile, per esempio, anche nei CSS. Riprendendo l'esempio 10.6 della torta, possiamo sfruttare il gradiente radiale per rendere più accattivante il nostro grafico. Nell'esempio 10.12 definiamo tre gradienti, ognuno dei quali utilizzato su ogni fetta della torta.

Esempio 10.12

```
<svg width="500" height="450">
  <defs>
    <radialGradient id="blueg" cx="0" cy="1" r="1">
      <stop offset="0" stop-color="black" />
      <stop offset="1" stop-color="blue" />
    </radialGradient>
    <radialGradient id="redg" cx="1" cy="1" r="1">
      <stop offset="0" stop-color="black" />
```

```

        <stop offset="1" stop-color="red" />
    </radialGradient>
    <radialGradient id="yellowg" cx="1" cy="0" r="1">
        <stop offset="0" stop-color="black" />
        <stop offset="1" stop-color="yellow" />
    </radialGradient>
</defs>
<path fill="url(#blueg)"
      d="M 310,402 L 213,214 L 326,35 C 454,111 468,319 310,402
        z" />
<path fill="url(#redg)"
      d="M 3,208 L 213,214 L 326,35 C 202,-46 8,30 3,208 z"/>
<path fill="url(#yellowg)"
      d="M 3,208 L 213,214 L 311,400 C 177,467 3,380 3,208 z" />
</svg>

```

In questo caso, essendo un radiale, dobbiamo specificare il centro $c_{x,xy}$ e il raggio, ottenendo la torta visibile nella [figura 10.8](#).

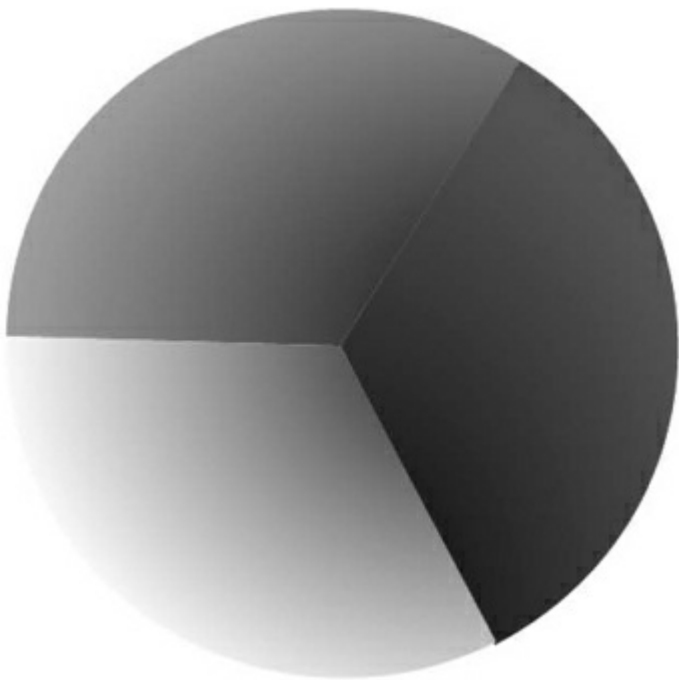


Figura 10.8 – Torta con gradienti radiali applicati a ogni fetta.

Oltre ai gradienti abbiamo, poi, la possibilità di creare `pattern`, cioè matrici di porzioni di SVG da utilizzare come colori di riempimento. Si definiscono anch'essi nella sezione `defs` e possono contenere qualsiasi shape, gruppo o immagine.

Nell'[esempio 10.13](#) possiamo vedere come definire due `pattern`, uno basato su uno shape e l'altro basato su un'immagine.

Esempio 10.13

```
<defs>
  <pattern id="blueg" x="0" y="0" width="4" height="4"
    patternUnits="userSpaceOnUse">
    <circle fill="blue" cx="2" cy="2" r="2" />
  </pattern>
  <pattern id="redg" x="0" y="0" width="200" height="133"
    patternUnits="userSpaceOnUse">
    <image xlink:href="crop.jpg" width="200" height="133" />
  </pattern>
</defs>
```

I pattern richiedono di specificare la dimensione della matrice di riempimento, perché ogni pattern deve adattarsi e ripetersi, così da riempire tutto l'oggetto in cui è applicato come sfondo. Se li applichiamo alla torta creata in precedenza, infatti, l'immagine viene duplicata, così come il cerchio blu.

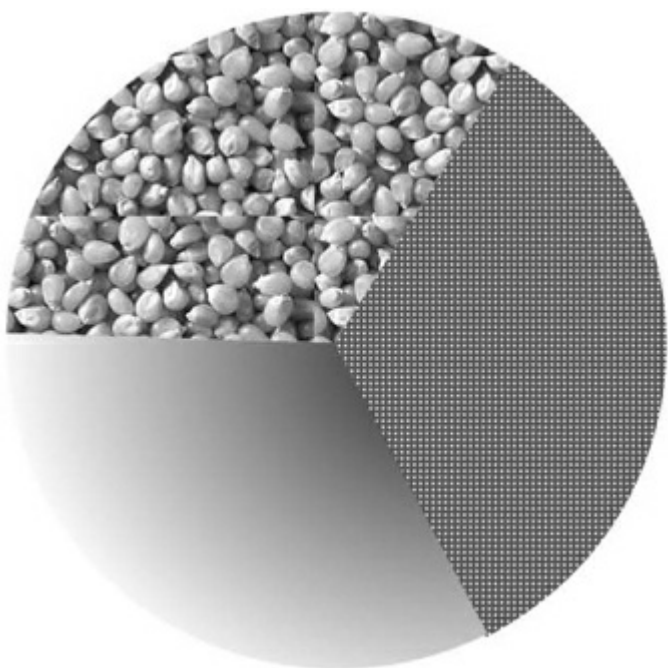


Figura 10.9 – Una torta in SVG con riempimenti basati su pattern.

Ora che abbiamo descritto le funzionalità avanzate di riempimento, possiamo vedere cosa SVG offre per allineare e trasformare gli elementi.

Trasformare gli elementi

Gli esempi mostrati finora sfruttano le funzionalità di posizionamento degli elementi, ma questo non è sempre sufficiente e, soprattutto, complica il lavoro

se vogliamo applicare effetti e manipolazioni, come vedremo più avanti nel capitolo.

Fortunatamente, ci viene in aiuto l'attributo `transform`, che permette di indicare le trasformazioni da applicare agli shape, agli `use` e alle immagini. Per capire come utilizzarle, riprendiamo la torta e stacciamo una fetta, come mostrato [nell'esempio 10.14](#).

Esempio 10.14

```
<path fill="url(#blueg)"  
      d="M 310,402 L 213,214 L 326,35 C 454,111 468,319 310,402 z"  
      transform="translate(10,0)" />
```

La funzione `translate` indica, attraverso i parametri `x` e `y`, di quanti pixel spostare l'intero elemento. Questo aspetto delle specifiche facilita moltissimo il lavoro, perché non dobbiamo rivedere tutte le coordinate del `path` e, soprattutto, è una tecnica applicabile anche ai gruppi.

La traslazione non è l'unica funzione disponibile: nella [tabella 10.3](#) possiamo trovare la lista completa delle funzioni di trasformazione, con i relativi parametri.

Funzione	Descrizione	Parametri
<code>translate</code>	Trasla l'elemento in tutte le direzioni.	<code>x, y</code>
<code>scale</code>	Scala l'elemento, rimpicciolendolo o ingrandendolo a seconda dei parametri.	<code>sx, sy</code>
<code>rotate</code>	Ruota l'elemento in base all'angolo e al centro.	<code>angle, cx, cy</code>
<code>skewX</code>	Inclina orizzontalmente l'elemento in base all'angolo.	<code>angle</code>
<code>skewY</code>	Inclina verticalmente l'elemento in base all'angolo.	<code>angle</code>
<code>matrix</code>	Permette di creare una trasformazione in base a una matrice di 6x6.	<code>a, b, c, d, e, f</code>

Tabella 10.3 – Le funzioni di trasformazione messe a disposizione da SVG.

Le trasformazioni, inoltre, possono essere combinate separandole dallo spazio, come mostrato [nell'esempio 10.15](#), che sposta e ingrandisce del 10% l'oggetto.

Esempio 10.15

```
<path fill="url(#blueg)"
      d="M 310,402 L 213,214 L 326,35 C 454,111 468,319 310,402
      z"
      transform="translate(10,-18) scale(1.1, 1.1)" />
```

A questo punto del capitolo abbiamo affrontato gli elementi necessari per disegnare, ma sicuramente uno degli aspetti più interessanti di SVG è la sua integrazione con CSS.

Applicare i CSS

Cascading Style Sheets (CSS) è un linguaggio per descrivere la presentazione delle pagine web. Viene utilizzato da HTML5 per separare le strutture di informazioni che HTML5 rappresenta dagli aspetti prettamente grafici e di presentazione.

CSS è molto comodo per definire gli aspetti grafici di un documento HTML, centralizzandone la dichiarazione e permettendone il riutilizzo una o più volte. L'aspetto interessante di SVG è che può lavorare in coppia con i CSS per definire gli attributi di riempimento, colorazione e font, appena visti nel corso del capitolo.

Anche per SVG possiamo quindi definire degli stili per tag, nominali o per id, all'interno della pagina stessa o in un file esterno. Riprendiamo ancora una volta la torta e creiamo uno stile CSS per impostare il colore e le informazioni del font delle etichette da porre a ogni fetta, come mostrato [nell'esempio 10.16](#).

Esempio 10.16

```
<style type="text/css">
  text
  {
    fill: white;
    font-family: Tahoma;
    font-size: 30px;
  }
</style>
```

Poiché impostiamo lo stile su tutti i tag `text`, non dobbiamo fare altro che definire le tre etichette all'interno dell'SVG.

Esempio 10.17

```
<svg>
  <!-- ... path omessi... -->
  <text x="320" y="230">Blue</text>
  <text x="120" y="120">Red</text>
  <text x="120" y="320">Yellow</text>
</svg>
```

Come possiamo verificare nella [figura 10.10](#), otteniamo di conseguenza le tre etichette con testo in bianco.

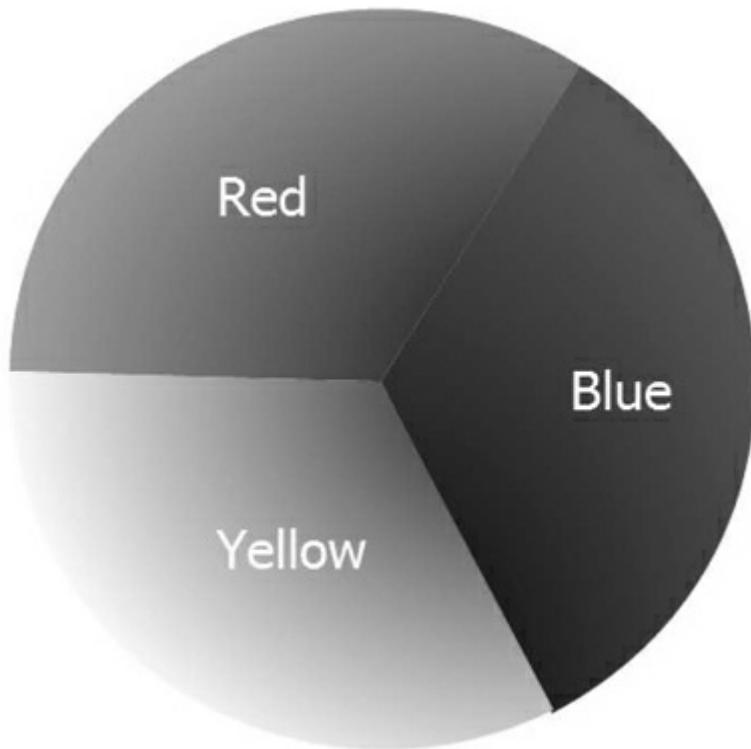


Figura 10.10 – I CSS applicati agli elementi SVG consentono di definire più facilmente gli stili.

Possiamo anche usare gli stili con un nome, specificando il CSS con l'attributo `class`. In alternativa all'uso degli stili definiti nella sezione `style`, possiamo definire gli stili in linea, attraverso l'attributo `style`: sebbene quest'ultima tecnica non porti apprezzabili benefici e dovrebbe essere evitata.

Molto interessante e utile è, invece, la possibilità di manipolare gli

elementi di un documento SVG attraverso lo scripting, cioè sfruttando codice JavaScript.

Manipolazione DOM e scripting

All'inizio del capitolo abbiamo accennato al fatto che SVG, oltre a offrire la possibilità di realizzare grafica vettoriale, sfrutta una sintassi basata su XML e, come tale, una volta incorporato in una pagina HTML (anche in un file esterno), rientra a far parte del **Document Object Model** (DOM) della pagina stessa, che poi approfondiremo meglio nel corso dei capitoli dedicati a JavaScript.

Questo fatto comporta che gli elementi SVG, come quelli di HTML5, formano una struttura ad albero, che possiamo consultare a runtime con i linguaggi di scripting. Per questo, diventa possibile intercettare eventi, manipolarne le proprietà, rimuovere o aggiungere elementi, anche con l'aiuto di librerie come **jQuery**, non solo direttamente con JavaScript. L'obiettivo che riusciamo a raggiungere è quello di dare maggiore interattività agli elementi, caratteristica che va ad aggiungersi alla lista dei vantaggi che offre SVG rispetto alle bitmap.

Per una dimostrazione pratica, riprendiamo il grafico a torta visto in precedenza e aggiungiamogli interattività: inseriamo nella pagina HTML un tag `script` e, con l'aiuto di JavaScript, cerchiamo gli elementi `g` che raggruppano i path e il testo di ogni fetta della torta.

Esempio 10.18

```
<script type="text/javascript">
  var elements = document.getElementsByTagName('div');

  for (i = 0; i < elements.length; i++)
  {
    var element = elements[i];

    element.addEventListener("mouseover", function () {
      this.setAttribute("transform", "scale(1.1)");
      document.getElementById("current").innerHTML =
                                                                    this.textContent
    }, false);

    element.addEventListener("mouseleave", function () {
      this.removeAttribute("transform");
    }, false);
  }
</script>
```

```
}  
</script>
```

Nell'esempio 10.18 possiamo vedere come allo startup cerchiamo ogni raggruppamento e intercettiamo gli eventi `mouseover` e `mouseleave`.

Con il primo evento, che si verifica quando l'utente porta il mouse sull'elemento, applichiamo una trasformazione che ingrandisce l'elemento, mentre nel secondo rimuoviamo la trasformazione; oltre a questo, valorizziamo un tag HTML5 con ID `current` con il valore dell'elemento su cui stiamo passando. Quello che otteniamo al passaggio del mouse sulla fetta gialla è visibile nella figura 10.11.

Yellow

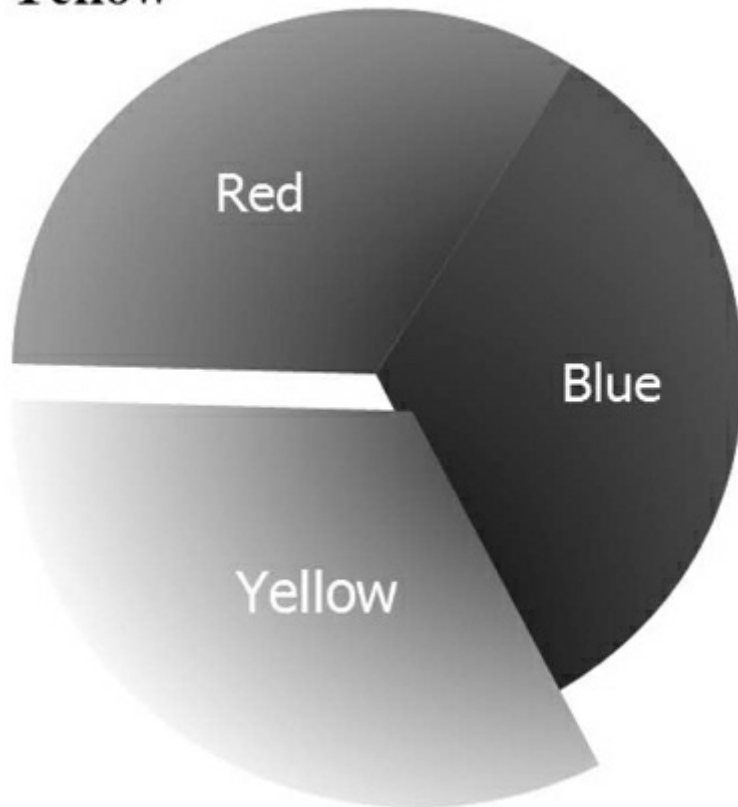


Figura 10.11 – Un documento SVG che reagisce alle azioni dell'utente, grazie al JavaScript.

Visto che con JavaScript possiamo manipolare gli oggetti, possiamo anche applicare animazioni agli stessi, utilizzando la funzione `setInterval`, che permette di ripetere a intervalli regolari una funzione. In essa possiamo incrementare posizioni, applicare trasformazioni o cambiare colori. In realtà, esiste nelle specifiche di SVG Animation la possibilità di usare tag specifici

per l'animazione degli elementi e delle loro proprietà, ma sono poco sfruttate, perché con JavaScript possiamo ottenere lo stesso risultato e avere il massimo della flessibilità.

Chi già conosce JavaScript e mastica il DOM non dovrebbe trovare difficoltà ad applicare queste tecnologie a SVG, poiché non ci sono particolari considerazioni da fare a riguardo, salvo il fatto che questo linguaggio è nato per figure statiche, definite appunto tramite il markup. Se invece quello di cui necessitiamo è una forte capacità di rendering, sia in termini di prestazioni sia di flessibilità, allora è più opportuno prendere in considerazione il `canvas`.

Grafica bitmap con il canvas

Il `canvas` è un nuovo elemento di HTML5 che ha come obiettivo rispondere all'esigenza di avere una superficie dove renderizzare al volo bitmap. È la scelta giusta se non ci sono altre possibilità di rendering e trova il massimo della sua utilità nei giochi, dove le scene cambiano più volte al secondo.

Si basa interamente sullo scripting, perché non esiste una sintassi di disegno ma delle API da chiamare per preparare la bitmap. Per capire come funziona, creiamo una pagina HTML5, inserendo il tag `canvas` nel punto in cui vogliamo renderizzare i contenuti.

Esempio 10.19

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/ecmascript">
      var context;

      function start() {
        context = canvas.getContext("2d");
      }
    </script>
  </head>
  <body onload="start()">
    <canvas id="canvas" width="400" height="300">
    </canvas>
  </body>
</html>
```

Nell'esempio 10.19 notiamo la dichiarazione della dimensione del `canvas` che, direttamente o indirettamente, deve essere fatta, così da delineare l'area di

rendering. Oltre a questo, al `load` della pagina otteniamo il contesto di disegno, chiamando la funzione `getContext`. Quest'ultima determina le API di disegno che possiamo chiamare sul contesto, di cui esiste una sola tipologia standard di nome `2d`. Esiste anche un'implementazione di nome **WebGL**, implementata solo da alcuni browser, che porta le API di rendering 3D sul contesto del `canvas`.

In questa sede ci occupiamo delle API 2d, al fine di capire come possiamo impiegare il `canvas` e trarne vantaggi.

Implementare il game loop: la logica di un gioco

Abbiamo accennato al fatto che il `canvas` trova soprattutto applicazioni nei giochi, perciò, per capire come sfruttarlo, svilupperemo una semplice superficie con una palla che rimbalza lungo i confini del `canvas`. Rispetto a SVG, non ci troviamo più a lavorare con elementi grafici, ma semplicemente a disegnare su una tavola di precise dimensioni. Questo porta maggiori benefici quando gli oggetti da gestire sono molteplici, perché il browser non deve gestirli, ma siamo noi a renderizzarli come un'unica fotografia. Si parla di fotografia, perché nei giochi la tecnica più usata per creare immagini in movimento è creare un infinito ciclo (chiamato **game loop**), che ha il compito di occuparsi della logica e del disegno del fotogramma. In JavaScript questo si traduce in una chiamata alla funzione `setInterval`, in modo da chiamare del codice a intervalli regolari. [Nell'esempio 10.20](#) aggiungiamo quindi questa chiamata, in modo che crei 60 frame al secondo, che è il frame rate tipico dei giochi.

Esempio 10.20

```
function start() {  
    context = canvas.getContext("2d");  
    // Ciclo del gioco, in millisecondi  
    setInterval(function () { update(); draw(); }, 1000 / 60);  
}
```

Le funzioni chiamate sono due: `update` per le logiche e `draw` per il disegno. Per lo scopo, nel primo metodo aggiorniamo a ogni frame la posizione della palla (variabili `ballx` e `bally`) e se questa supera i confini del `canvas` ne invertiamo il senso (`vx` e `vy` in base all'asse).

Esempio 10.21

```
function update() {  
  // Rimbalza in alto o in basso  
  if (bally > height - ballSize || bally < ballSize) {  
    vy = vy * -1;  
    count++;  
  }  
  // Rimbalza a sinistra o a destra  
  if (ballx > width - ballSize || ballx < ballSize) {  
    vx = vx * -1;  
    count++;  
  }  
  
  // Sposto la palla  
  ballx = ballx + (vx * speed);  
  bally = bally + (vy * speed);  
}
```

La variabile `speed` è un coefficiente che permette di accelerare lo spostamento della palla, poiché `vx` e `vy` danno solo il senso con valori `+1` o `-1`. A questo punto, abbiamo un motore che gira e aggiorna le variabili di posizionamento della palla e non ci resta che disegnarla sulla nostra superficie.

Implementiamo, quindi, il metodo `draw`, agendo completamente sull'oggetto `context`, creato allo start dell'applicazione. Su di esso possiamo chiamare una serie di funzioni per il disegno sulla superficie, con metodologie e concetti del tutto simili a quanto visto con SVG. La differenza risiede solo nel modo con cui disegniamo: non più da markup, ma programmaticamente, cioè da codice JavaScript.

Nella realizzazione di un gioco, l'implementazione della funzione `draw` comporta l'esecuzione delle attività di disegno del frame, che vengono ripetute più volte, annullando il frame precedente, semplicemente sovrascrivendolo. Per capire come fare, vediamo quali sono le primitive di disegno che abbiamo a disposizione.

Le API di disegno

Le API che abbiamo a disposizione sul contesto possono essere sostanzialmente raggruppate in tre categorie: shape, immagini e testo. Questi elementi hanno in comune il fatto che possono avere un colore di riempimento e di contorno, perciò il contesto implementa una logica di disegno che si basa su uno stile. Ogni funzione di disegno deve, infatti, essere

preceduta dall'impostazione delle proprietà che ne influenzano l'attività. [L'esempio 10.22](#) disegna un rettangolo rosso con un contorno di nero di due pixel e rende più semplice il concetto.

Esempio 10.22

```
context.fillStyle = "red";
context.strokeStyle = "black";
context.lineWidth = 2;
context.fillRect(0, 0, 400, 300);
context.strokeRect(0, 0, 400, 300);
```

Nel codice precedente, notiamo che le colorazioni possono essere valorizzate con la stessa sintassi dei CSS e possiamo capire che tutte le operazioni di riempimento sono influenzate dalle proprietà a essa relative, così come le operazioni per il contorno.

Questa tecnica è molto comoda perché non obbliga a ripetere numerosi parametri a ogni funzione di disegno, ma può portare a risultati inaspettati, perché su listati molto lunghi possiamo perdere il controllo sulla situazione corrente di disegno. Per semplificarci la vita vengono in aiuto le funzioni `save` e `restore` che, se invocate all'inizio e alla fine di un blocco di codice, garantiscono che al `restore` la situazione degli stili sia ripristinata a quella presente quando abbiamo invocato `save`.

Procediamo ancora con il gioco e per prima cosa disegniamo un'immagine da porre sullo sfondo: con questa tecnica azzeriamo il frame precedente con facilità. Per renderizzarla, abbiamo a disposizione la funzione `drawImage`, la quale accetta un'immagine (cioè un tag `` di HTML5) e le coordinate di posizionamento.

nota

L'immagine dev'essere un elemento già caricato dal browser, perciò è opportuno avviare il gioco, e quindi il suo rendering, solo dopo aver caricato tutti gli elementi necessari. L'evento `onload` del tag `body` garantisce che questo sia stato fatto.

[Nell'esempio 10.23](#) vediamo come abbiamo implementato la funzione `draw`.

Esempio 10.23

```
function draw() {  
    // Disegno lo sfondo  
    context.drawImage(background, 0, 0);  
  
    // Disegno la palla  
    drawBall();  
  
    // Scrivo il testo  
    drawText();  
}
```

A ogni frame, oltre a disegnare l'immagine, disegneremo anche la palla, grazie all'uso del path specificato.

Disegnare con il path

Nell'esempio 10.22 abbiamo visto l'uso delle funzioni `fillRect` e `strokeRect`, da usare per il riempimento e il contorno del rettangolo. In realtà, sono le uniche due funzioni di disegno, perché, per le altre figure primitive, l'unico strumento che possiamo usare è il path. La logica è simile a quanto già visto con SVG:

- iniziamo il disegno di una forma con la funzione `beginPath`;
- spostiamo il cursore o dichiariamo gli elementi con le funzioni `moveTo`, `lineTo`, `quadraticCurveTo`, `bezierCurveTo`, `arcTo` e `arc`;
- chiudiamo facoltativamente il disegno della forma con `closePath`;
- riempiamo o contorniamo la forma disegnata con `fill` o `stroke`.

Per il disegno della palla, il nostro obiettivo è riempire e contornare un cerchio: nell'esempio 10.24 c'è il codice necessario a implementare la funzione `drawBall`.

Esempio 10.24

```
function drawBall() {  
    context.save();  
  
    // Traccio la palla  
    context.beginPath();
```

```

context.arc(ballx, bally, ballSize * 2, 0, Math.PI * 2, true);

// Creo il gradiente della palla
var radial = context.createRadialGradient(
                                ballx, bally, 2, ballx, bally,
                                10);
radial.addColorStop(0, "#F06707");
radial.addColorStop(1, "#E34F12");

// Coloro la palla
context.fillStyle = radial;
context.fill();

// Disegno il contorno della palla
context.strokeStyle = "black";
context.stroke();

context.restore();
}

```

Nel codice ci sono molti spunti interessanti: innanzitutto la chiamata iniziale e finale delle funzioni `save` e `restore`, che serve a garantire il chiamante sul ripristino dello stato. Viene poi preparato il cerchio, con la chiamata a `arcTo`, che necessita di parametri per indicare il centro, il raggio, l'angolo iniziale e finale (espressi in radiale) e se il senso è antiorario. Segue la chiamata a `createRadialGradient` che, in un'unione a `createLinearGradient`, permette la creazione di un gradiente radiale o lineare.

Come in SVG, dobbiamo definire i due cerchi di inizio e di fine e i colori che compongono il gradiente. Molto importante è il fatto che le coordinate del gradiente siano globali per tutta la dimensione della clip corrente (il `canvas`).

La funzione, poi, prosegue assegnando il colore di riempimento con il gradiente, lo riempie, assegna il colore del contorno e lo disegna.

Entrambe le funzioni agiscono sul path corrente, che è il cerchio. Non ci resta, a questo punto, che scrivere del testo per segnare banalmente il numero dei rimbalzi effettuati.

Scrivere testo sulla bitmap

La scrittura del testo segue la stessa logica vista finora, perciò, prima di procedere con la scrittura, dobbiamo ricordarci di valorizzare quelle proprietà che ne determinano l'aspetto: `font`, `textAlign`, `textBaseline`.

Ricordiamoci per un attimo che vogliamo disegnare in alto a destra il

numero dei rimbalzi. Anche questa volta, vediamo prima la funzione `drawText` [nell'esempio 10.25](#), per poi commentarne il codice.

Esempio 10.25

```
function drawText() {  
    context.save();  
  
    context.fillStyle = "black";  
    context.textAlign = "right";  
    context.textBaseline = "top";  
    context.font = "bold 12px Tahoma";  
    context.fillText("Boing: " + count, 590, 10);  
  
    context.restore();  
}
```

Come anticipato nel codice, valorizziamo prima le proprietà, rendendo il testo nero, allineato a destra, con la base in alto e con il font formattato come in CSS. Infine, chiamiamo la funzione `fillText` che, in coppia con `strokeText`, permette di riempire e contornare del testo. Poiché abbiamo allineato in alto a destra, le coordinate 590x10 si riferiscono all'angolo in alto a destra della zona occupata dal testo.

Tutto questo ci porta finalmente a ottenere il risultato finale, visibile nella [figura 10.12](#).

Questo semplice esempio, in realtà, già ci permette di affrontare la maggior parte delle funzioni che il contesto 2d mette a disposizione. Le capacità di realizzazione di un gioco accattivante è, infatti, interamente demandata allo sviluppatore e alle sue capacità di programmazione, oltre che a una buona base grafica di partenza.



Figura 10.12 – Una palla che rimbalza dentro un contenitore, realizzato tramite canvas.

Rimangono esclusi dagli esempi presentati in questo capitolo solo argomenti più avanzati, ma di cui, in realtà, possiamo intuire facilmente l'utilizzo, perché identici a quanto visto con SVG: le trasformazioni per mezzo delle funzioni `scale`, `translate`, `rotate` e `transform`. Da segnalare, infine, la funzione `toDataURL`, che permette di avere uno screenshot del `canvas` sotto forma di URI contenente l'immagine PNG serializzata in base64. Questo può essere molto comodo per salvare l'immagine in maniera che sia trasportabile facilmente, per esempio in una mail o in un documento HTML che non abbia file esterni.

Conclusioni

In questo capitolo abbiamo affrontato un tema che va oltre la descrizione delle informazioni tipica dell'HTML e si spinge fino alla creazione di grafica avanzata.

SVG e `canvas` sono due strumenti simili nella logica di disegno, ma molto differenti nell'utilizzo e nelle potenzialità. Il primo permette di descrivere con markup figure vettoriali, che godono quindi di una definizione infinita superando i limiti delle bitmap, mentre il secondo fornisce una superficie di disegno sulla quale interagire con API di scripting. Quindi, con SVG, abbiamo visto le principali caratteristiche per il disegno degli shape e per l'inserimento di elementi testuali. Con il raggruppamento e le definizioni abbiamo illustrato come organizzare le figure e come possiamo anche definire gradienti o pattern di colorazione delle figure stesse.

Siamo poi passati a funzionalità più avanzate, come le trasformazioni, per vedere come SVG, essendo basato su markup, possa essere conciliato con l'uso di CSS e di linguaggi di scripting.

Per capire invece l'utilità del `canvas`, abbiamo realizzato un piccolo gioco che mette in campo la logica basata sui frame per il disegno delle animazioni. Infine, abbiamo visto come, col contesto 2d ottenibile dal `canvas`, possiamo disegnare attraverso le API e rappresentare gli elementi di cui possiamo aver bisogno: shape, immagini e testo.

Nel prossimo capitolo, l'ultimo, vedremo quali sono i framework per il Web e come possono aiutarci nello sviluppo.

I Framework per il Web

Nei capitoli precedenti abbiamo introdotto HTML, CSS e JavaScript e abbiamo visto le loro funzionalità di base quanto quelle avanzate. Questo rappresenta la base per lo sviluppo sul web. Tuttavia esistono dei framework che offrono già delle funzionalità di base, delle classi CSS pronte da utilizzare, in modo che possiamo costruire la nostra applicazione partendo da una serie di componenti già esistenti.

Questi framework offrono funzionalità come astrazione delle differenze tra browser, classi CSS già pronte per semplificare il layout, metodi di accesso e manipolazione del DOM più semplici da usare, oggetti visuali e non per lavorare con date, grafici, template HTML e altro ancora.

Creare applicazioni web senza usare questi framework è semplicemente impensabile, poiché la quantità di codice che dovremmo scrivere sarebbe enorme. Quasi tutte queste librerie sono **Open Source** e quindi ci si può scoraggiare per la difficoltà nel trovare il supporto. Tuttavia questo, nella maggior parte dei casi, non rappresenta un problema, in quanto si tratta di librerie oramai stabili e con una community affidabile alle spalle.

In questo capitolo ci occuperemo di questi framework, illustrando come ognuno di essi è fondamentale nello sviluppo e come ci permettano di risparmiare un'enorme quantità di righe di codice. Cominciamo da quello che è ormai lo standard JavaScript del Web: **jQuery**.

jQuery

jQuery è un framework JavaScript molto potente, che mette a disposizione degli sviluppatori funzionalità avanzate per la ricerca, la navigazione e la manipolazione del DOM, per la gestione degli eventi, per la gestione delle richieste AJAX, per l'astrazione delle differenze tra browser e molto altro ancora. Il modo in cui jQuery ha rivoluzionato lo sviluppo web grazie alla sua semplicità di utilizzo ha portato alla creazione del seguente modo di dire: "Io

non sviluppo scrivendo JavaScript, sviluppo scrivendo jQuery”.

nota

jQuery è scaricabile all'indirizzo <http://aspit.co/a10>.

Cominciamo a scoprire perché si è arrivati a questo modo di dire, analizzando le singole funzionalità di jQuery e partendo dalla capacità di eseguire query nel DOM.

Effettuare ricerche nel DOM

jQuery è nato per eseguire query nel DOM del browser al fine di recuperare oggetti. Queste query sono effettuate basandosi sulla sintassi CSS esattamente come avviene per il metodo `querySelectorAll` dell'oggetto `document`. `querySelectorAll` è un metodo introdotto da HTML5 e quindi relativamente giovane; jQuery permetteva di effettuare query già nel 2007.

Per sfruttare jQuery, il punto di entrata è l'oggetto `$`. Questo oggetto può essere usato sia come metodo sia come oggetto. Se vogliamo effettuare una query nel DOM, sfruttiamo questo oggetto come metodo passando in input la stringa di ricerca, così come nel prossimo esempio.

Esempio 11.1

```
//ricerca un elemento per id
var objects = $("#textBoxId");

//ricerca elementi per classe
var objects = $(".classe");

//ricerca tag p all'interno dei div con classe "contenitore"
var objects = $("div.contenitore p");
```

Ci sono due cose da notare in questo esempio. La prima è che, a prescindere da quanti oggetti possa tornare la query, il metodo da usare è sempre lo stesso. La seconda è che la variabile `objects` non contiene una lista di oggetti HTML, come ci si potrebbe aspettare, bensì un oggetto di jQuery (detto contenitore) che contiene la lista degli oggetti recuperati al suo interno. Questo oggetto dimostra la sua utilità proprio quando dobbiamo manipolare gli oggetti al suo

interno.

Manipolare gli oggetti

Supponiamo di voler aggiungere una classe CSS a tutti gli elementi restituiti da una query. Se non utilizziamo jQuery, una volta ottenuta la lista di oggetti dobbiamo eseguire un ciclo, aggiungendo la classe a tutti. Con jQuery possiamo fare tutto in una riga di codice, invocando il metodo `addClass` del contenitore passandogli in input il nome della classe, come viene mostrato nell'esempio 11.2.

Esempio 11.2

```
var objects = $(".myClass").addClass("selected").  
addClass("anotherClass");
```

Il metodo `addClass` scorre tutti gli oggetti nel contenitore e aggiunge a ognuno di essi la classe `selected`, permettendoci quindi di risparmiare codice. Non solo, il metodo `addClass` è una funzione che ritorna il contenitore stesso. Questo significa che possiamo sfruttare la tecnica *fluent* per invocare di nuovo il metodo `addClass` per aggiungere un'altra classe CSS agli oggetti. Il metodo `addClass` è solo uno dei tanti metodi per manipolare gli oggetti: parlare di tutti i metodi è impossibile per ovvie questioni di spazio, quindi vi offriamo un riepilogo di quelli più importanti nella [tabella 11.1](#).

Metodo	Descrizione
<code>append</code>	Prende in input un oggetto jQuery o una stringa HTML e li aggiunge al contenuto di tutti gli oggetti nel contenitore.
<code>html</code>	Prende in input una stringa HTML e la imposta come proprietà <code>innerHTML</code> di ogni oggetto nel contenitore.
<code>text</code>	Prende in input una stringa e la imposta come proprietà <code>innerText</code> di ogni oggetto nel contenitore.
<code>after</code> <code>before</code>	Accettano in input un oggetto jQuery o una stringa HTML, che vengono aggiunti rispettivamente prima o dopo ogni oggetto nel contenitore.
<code>empty</code>	Elimina il contenuto di ogni oggetto nel contenitore.

<code>remove</code>	Accetta in input una lista di oggetti da rimuovere dal DOM.
<code>attr</code>	Accetta in input il nome di un attributo HTML e restituisce il valore di quell'attributo per il primo oggetto nel contenitore. Se il contenitore contiene più oggetti, gli altri vengono ignorati. Questo metodo ha un overload che accetta un secondo valore. In questo caso, l'attributo HTML viene impostato con il secondo valore.
<code>removeAttr</code>	Accetta in input il nome dell'attributo HTML da eliminare dal primo oggetto nel contenitore.
<code>val</code>	Ritorna la proprietà <code>value</code> del primo oggetto nel contenitore. Questo metodo ha un overload che accetta un valore che viene poi usato per impostare la proprietà <code>value</code> del primo oggetto nel contenitore.
<code>addClass</code> <code>removeClass</code> <code>toggleClass</code>	I primi due metodi aggiungono e rimuovono rispettivamente una classe CSS dagli oggetti nel contenitore. Il terzo metodo esegue una verifica sull'esistenza della classe CSS sull'oggetto: se è già presente, allora la rimuove, altrimenti la aggiunge.
<code>hide</code> <code>show</code> <code>toggle</code>	I primi due metodi mostrano e nascondono rispettivamente gli oggetti nel contenitore. Il terzo metodo esegue una verifica sulla visibilità dell'oggetto: se è visibile, allora lo nasconde, altrimenti lo rende visibile.

Tabella 11.1 – Elenco dei metodi di navigazione del DOM di un oggetto.

Oltre a manipolare gli oggetti, jQuery permette anche di gestirne gli eventi.

Gestire gli eventi

Gestire gli eventi degli oggetti con jQuery è semplice come manipolare gli oggetti stessi. Una volta recuperati gli oggetti che vogliamo monitorare e manipolare, dobbiamo usare il metodo `on` passando in input il nome dell'evento e il delegato da invocare quando si scatena l'evento.

Se invece vogliamo eliminare la sottoscrizione all'evento, dobbiamo usare il metodo `off`, che accetta gli stessi parametri di `on`. L'utilizzo di `on` e `off` è

visibile nel prossimo esempio.

Esempio 11.3

```
//si sottoscrive all'evento click di un pulsante
$("#myButton").on("click", handler);

//si cancella dall'evento click di un pulsante
$("#myButton").off("click", handler);

function handler() {
  alert("bottone cliccato");
}
```

Oltre a sottoscriverci, possiamo anche scatenare programmaticamente un evento tramite il metodo `trigger`, che accetta in input il nome dell'evento da scatenare.

C'è un evento particolare in jQuery che merita attenzione, cioè l'evento scatenato quando la pagina è caricata dal browser. Per intercettare questo evento, ci basta passare a `$` una funzione invece che una stringa. In questo caso, il codice della funzione viene eseguito nel momento del caricamento della pagina.

Quella di lavorare con gli oggetti del DOM è stata la prima funzionalità di jQuery, ma col tempo se ne sono aggiunte altre, la più importante delle quali è senz'altro la possibilità di effettuare chiamate AJAX.

Effettuare chiamate AJAX

Eseguire una chiamata AJAX utilizzando direttamente `XMLHttpRequest` è semplice ma richiede una notevole quantità di codice rispetto a quello che dobbiamo scrivere usando jQuery.

jQuery mette a disposizione il metodo `ajax`, il quale accetta in input un oggetto che contiene le proprietà necessarie per effettuare la chiamata. Le proprietà fondamentali di questo oggetto sono elencate nella [tabella 11.2](#).

Proprietà	Descrizione
cache	Se impostato a <code>false</code> disabilita la cache del browser, appendendo un timestamp all'url.
	Rappresenta un oggetto contenente i dati da inviare al server. Se

<code>data</code>	la richiesta è di tipo GET, l'oggetto viene serializzato in <code>querystring</code> , altrimenti viene serializzato nel corpo della richiesta.
<code>headers</code>	Rappresenta un oggetto contenente le intestazioni da aggiungere alla richiesta.
<code>jsonp</code>	Se impostata, specifica che la chiamata deve sfruttare la tecnica JSONP e il valore della proprietà rappresenta il metodo da invocare a fine chiamata.
<code>timeout</code>	Specifica il timeout della richiesta in millisecondi.
<code>type</code>	Specifica il tipo della richiesta (POST, GET e così via).
<code>url</code>	Specifica l'url da invocare.

Tabella 11.2 – Elenco delle proprietà dell'oggetto in input al metodo `ajax`.

Poiché le chiamate AJAX sono asincrone, il metodo `ajax` ritorna una **promise**. Una promise è un'oggetto di tipo `Deferred` che rappresenta un'operazione in sospeso (in questo caso la chiamata) e che, una volta terminata, invoca determinati metodi per sfruttarne il risultato. I metodi sono:

- `done`: invocato quando la promise torna con successo;
- `fail`: invocato quando la promise torna un errore (per esempio, quando il server torna un errore);
- `always`: invocato a prescindere dal risultato della promise.

L'esempio 11.4 mostra come usare il metodo `ajax`.

Esempio 11.4

```
$.ajax({
  url: "/userhandler"
  data: { userId: 1 }
})
.done(function(result) {
  alert(result);
})
.fail(function() {
  alert("errore");
})
```

```
.always(function() {  
    alert("chiamata terminata");  
});
```

Il metodo `ajax` è banale ma jQuery offre anche dei metodi, basati su `ajax`, che semplificano ancora di più il codice. Questi metodi sono: `getJSON`, `post` e `load`.

- `get`: esegue una chiamata GET;
- `getJSON`: esegue una chiamata GET che accetta una risposta in JSON;
- `post`: esegue una chiamata POST;
- `load`: esegue una chiamata GET caricando il risultato in un oggetto del DOM.

Tutti i metodi accettano in input l'url da invocare e i parametri da passare, e ritornano una promise. Il loro utilizzo è visibile [nell'esempio 11.5](#).

Esempio 11.5

```
$.get("/userhandlerget", { userId: 1 });  
  
$.getJSON("/userhandlerjson", { userId: 1 });  
  
$.post("/userhandlerpost", { userId: 1 });  
  
$("#divid").load("/userhandlerload", { userId: 1 });
```

Non solo jQuery semplifica il codice JavaScript, ma mette a disposizione anche una libreria di controlli visuali: **jQueryUI**.

jQueryUI

Per sfruttare al massimo la semplicità con cui jQuery permette di lavorare con il DOM, il team di jQuery ha creato una libreria di componenti visuali chiamata jQueryUI. Allo stato attuale, la libreria comprende dodici controlli visuali, ma quelli più usati sono i seguenti:

- `accordion`: racchiude un insieme di pannelli mostrandone solo le intestazioni; il contenuto di un solo pannello per volta per volta può

essere visibile;

- **autocomplete**: permette l'autocomplete sulle textbox;
- **datepicker**: mostra un calendario associato a una textbox;
- **dialog**: mostra una finestra in overlay;
- **tabs**: racchiude pannelli in una visualizzazione a tab.

Non parleremo in dettaglio di questi componenti, ma nel prossimo esempio mostreremo quanto sia semplice il loro uso.

Esempio 11.6

```
//html
<input type="text" id="datepicker">

//JavaScript
$(function() {
    $( "#datepicker" ).datepicker();
});
```

In questo esempio dichiariamo una textbox nel codice HTML e, successivamente al caricamento della pagina, le associamo il controllo `datepicker`. Il risultato è che quando la textbox prende il fuoco, viene visualizzato il calendario come nella [figura 11.1](#).



Figura 11.1 – Il controllo datepicker in azione.

Tutti i controlli di jQueryUI lavorano con codice HTML strutturato in maniera precisa, quindi è bene leggere la relativa documentazione sul sito, disponibile all'indirizzo: <http://aspit.co/ain>.

nota

jQuery mette a disposizione anche delle animazioni che scattano quando si mostra o nasconde un oggetto o quando si aggiunge o rimuove una classe CSS. Inoltre, mette a disposizione anche dei comportamenti per arricchire la nostra applicazione, come il drag&drop, l'ordinamento di oggetti e così via.

jQuery è oramai la base del Web. La quasi totalità dei framework per lo sviluppo web si basa su jQuery. Uno di questi framework è **Bootstrap**, che illustreremo nella prossima sezione.

Bootstrap

Bootstrap è un framework web che ci offre una serie di classi CSS e di controlli visuali JavaScript già pronti per l'uso.

Le classi CSS, in combinazione con il codice HTML impostato secondo il loro scopo, coprono gran parte delle necessità di un'applicazione web, come il layout responsive, gli stili di intestazioni, paragrafi, testo, bottoni, icone e molto altro ancora. Grazie all'estrema semplicità con cui si possono creare layout, Bootstrap è al momento il miglior framework HTML/CSS in circolazione.

I controlli visuali utilizzano JavaScript per creare tab, tooltip, finestre modali e altro ancora. Questi controlli sono basati su jQuery e sono un'alternativa (meno completa) a jQueryUI.

nota

Bootstrap è un progetto nato all'interno di Twitter, che lo ha reso disponibile all'universo web. Può essere scaricato all'indirizzo

Come abbiamo detto, il successo di Bootstrap è dovuto alla semplicità con cui permette di creare layout che siano anche responsivi. Cominciamo a vedere come funziona il sistema di layout.

Creare un layout

La semplicità nel creare layout di Bootstrap è dovuta al **grid system**. Il grid system prevede che la pagina sia suddivisa in righe e colonne. Ogni riga è suddivisa in dodici colonne “virtuali” della stessa dimensione e sta a noi creare colonne “reali” che occupino un certo numero di colonne virtuali.

Per chiarire meglio i concetti appena espressi, osserviamo la [figura 11.2](#).

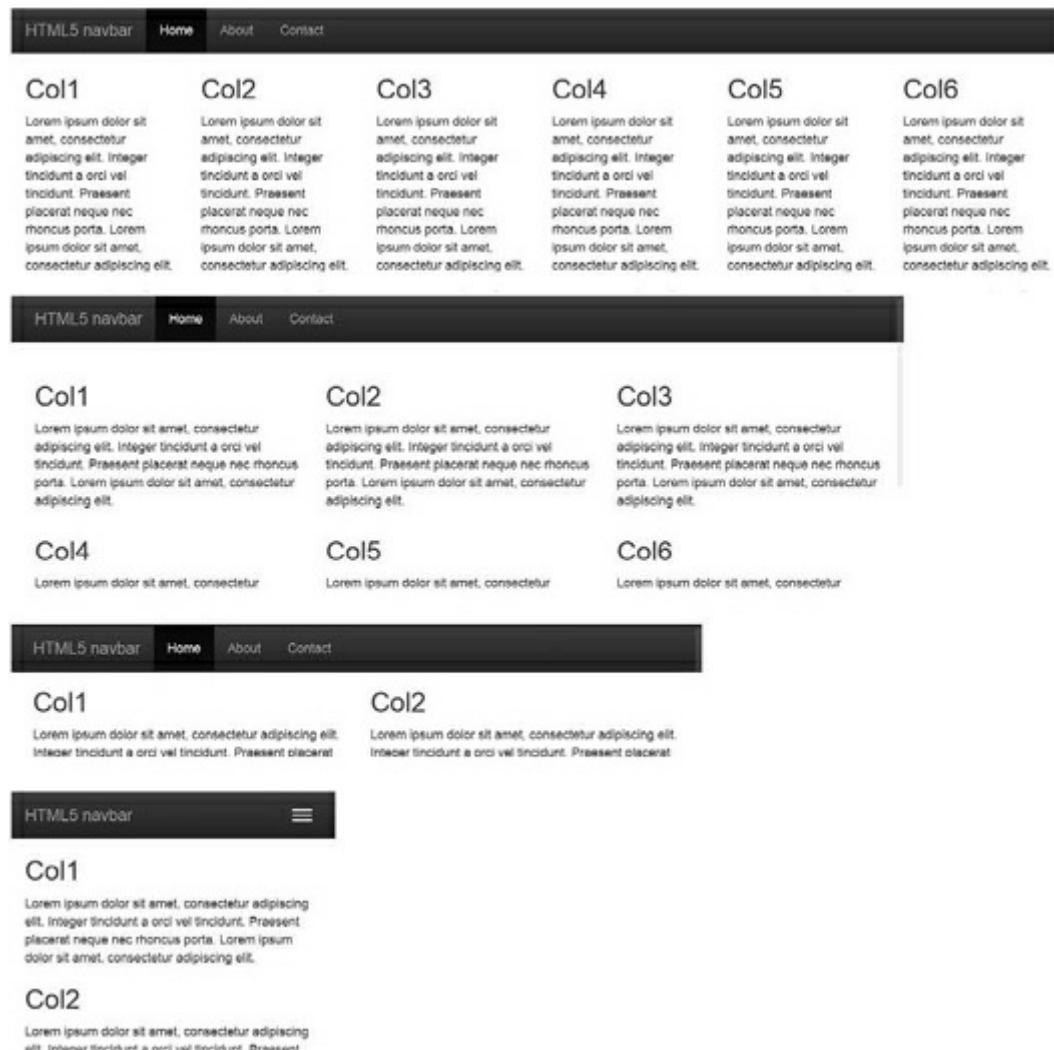


Figura 11.2 – Un layout responsive con Bootstrap.

Se analizziamo il primo layout della figura, abbiamo una riga che contiene sei colonne di uguali dimensioni. Questo significa che ogni colonna reale occupa lo spazio di due colonne virtuali. Se invece prendiamo il secondo layout, abbiamo tre colonne reali, ognuna delle quali occupa quindi quattro colonne virtuali. Il terzo layout ha due colonne reali, quindi ognuna ne occupa sei virtuali e, infine, il quarto layout ha una sola colonna che occupa tutte quelle virtuali. Questo è un tipico esempio di responsive layout.

Il codice HTML necessario a creare questo layout è visibile nel prossimo esempio.

Esempio 11.7

```
<div class="row">
  <div class="col-sm-6 col-md-4 col-lg-2">
    <h2>Col1</h2>
    <p>Contenuto</p>
  </div>
  <div class=" col-sm-6 col-md-4 col-lg-2">
    <h2>Col2</h2>
    <p>Contenuto</p>
  </div>

  <br class="clearfix visible-sm-block" />

  <div class=" col-lg-2">
    <h2>Col3</h2>
    <p>Contenuto</p>
  </div>

  <br class="clearfix visible-md-block" />

  <div class="col-sm-6 col-md-4 col-lg-2">
    <h2>Col4</h2>
    <p>Contenuto</p>
  </div>

  <br class="clearfix visible-sm-block" />

  <div class="col-sm-6 col-md-4 col-lg-2">
    <h2>Col5</h2>
    <p>Contenuto</p>
  </div>
  <div class="col-sm-6 col-md-4 col-lg-2">
    <h2>Col6</h2>
    <p>Contenuto</p>
  </div>
```

Anche se questo codice HTML è apparentemente semplice, ci sono molti dettagli tecnici che meritano di essere approfonditi. Prima di tutto viene creato un `div` con classe CSS `row`, la quale rappresenta una riga. All'interno di questo tag `div` vengono creati sei `div`, tanti quante sono le colonne che vogliamo inserire. Infine, vengono inseriti dei tag `br` per far andare a capo le colonne in base al tipo di layout. Da un punto di vista del codice HTML, questo layout non dice molto; la vera magia consiste nelle classi CSS applicate: `col-*`. Bootstrap suddivide il layout in quattro categorie, in base a delle media query:

- **Large:** riconoscibile dalla sigla `lg` nelle classi CSS. Questo layout entra in gioco quando la larghezza del browser è superiore a 1200px (tipicamente quella dei monitor 16:9 e dei grandi schermi).
- **Medium:** riconoscibile dalla sigla `md` nelle classi CSS. Questo layout entra in gioco quando la larghezza del browser è superiore a 992px e inferiore a 1200px (tipicamente quella dei monitor più vecchi).
- **Small:** riconoscibile dalla sigla `sm` nelle classi CSS. Questo layout entra in gioco quando la larghezza del browser è superiore a 768px e inferiore a 992px (tipicamente quella dei tablet).
- **Extra-small:** riconoscibile dalla sigla `xs` nelle classi CSS. Questo layout entra in gioco quando la larghezza del browser è inferiore a 768px (tipicamente quella degli smartphone).

Grazie a questa suddivisione, possiamo decidere come organizzare le colonne in base al layout. Nel nostro esempio:

- quando le dimensioni sono `large` (primo layout della [figura 11.2](#)), entra in gioco la classe `col-lg-2`, la quale specifica che il `div` occupa due colonne virtuali;
- quando le dimensioni sono `medium` (secondo layout della [figura 11.2](#)), entra in gioco la classe `col-md-4`, quindi il `div` occupa quattro colonne virtuali;
- quando le dimensioni sono `small` (terzo layout della [figura 11.2](#)), entra

in gioco la classe `col-sm-6`, quindi il `div` occupa sei colonne virtuali;

- quando le dimensioni sono `extra-small` (quarto layout della [figura 11.2](#)), entra in gioco la classe `col-xs-12`, quindi il `div` occupa tutte le colonne virtuali (questa classe non è specificata, in quanto è inutile specificarla poiché per default tutta la riga viene occupata).

Nel codice HTML viene dichiarata solo una riga ma, fatta eccezione per il layout `large`, tutti gli altri mostrano le colonne su più righe. Questo avviene perché sugli altri layout lo spazio occupato dalle colonne è maggiore delle dodici virtuali, quindi il browser le manda a capo.

Per garantire un corretto allineamento quando si va a capo, vengono usati i tag `br`. A questi tag applichiamo la classe `clearfix`, per mandare correttamente a capo, e la classe `visible-{abbreviazione}-block`, per visualizzare o nascondere questo tag a seconda del tipo di layout. In questo modo, quando il layout è `small` il tag `br` diventa attivo dopo la seconda e quarta colonna, visto che mostriamo solo due colonne per riga. Finora abbiamo parlato delle colonne e di come renderle responsive, ma al lettore più attento non sarà di certo sfuggito il fatto che anche il menù è responsivo. Infatti, mentre nei primi tre layout il menu è in orizzontale, nell'ultimo scompare, lasciando il posto alle classiche tre righe orizzontali che, se cliccate, fanno apparire il menu a discesa. Per ottenere questo risultato, abbiamo utilizzato il seguente codice.

Esempio 11.8

```
<nav id="navbar"
  class="navbar navbar-default navbar-inverse navbarfixed-top">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle"
        datatoggle="collapse" data-target="#navbarCollapse">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="#">HTML5 navbar</a>
    </div>

    <div class="collapse navbar-collapse" id="navbarCollapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="/">Home</a></li>
        <li><a href="http://www.aspitalia.com">About</a></li>
```

```
        <li><a href="http://www.aspitalia.com">Contact</a></li>
    </ul>
</div>
</div>
</nav>
```

Il menu di navigazione viene creato attraverso il tag `nav`, al quale si applicano gli stili `navbar*` per stilizzarlo e specificare che il menu è sempre visibile in alto (`navbar-fixed-top`). Dopodiché si usa il tag `div` con classe `navbar-header`, per specificare il logo (tag `a` con classe `navbar-brand`) e il pulsante da visualizzare quando il layout diventa `extra-small` (tag `button`). Questo bottone ha l'attributo `data-target`, all'interno del quale specifichiamo il nome del `div` con il menu da mostrare quando il bottone viene cliccato.

Successivamente, creiamo il `div` con il menu da mostrare (tag `div` con id `navbarCollapse`). All'interno del `div` usiamo i tag `ul` e `li` per aggiungere le voci di menu, impostando la classe CSS `active` sulla voce che vogliamo selezionare. Grazie a questa piccola porzione di codice, abbiamo un menu responsivo a costo praticamente zero. Questo dovrebbe essere sufficiente a far capire perché Bootstrap è il framework HTML/CSS per eccellenza.

Tuttavia, come abbiamo spiegato in precedenza, Bootstrap non permette solo di creare layout in maniera semplice ma mette a disposizione anche altre classi per disegnare le nostre applicazioni web, come quelle per disegnare una form.

Creare una form con Bootstrap

Qualunque applicazione web necessita di form per l'inserimento di dati. Ne sono un esempio le form semplici come quelle di registrazione, di login e di ricerca o quelle più complesse in cui si inserisce una notevole mole di dati. Qualunque sia il tipo di form che si deve realizzare, Bootstrap ci aiuta a mantenere il codice HTML pulito e semplice. Prendiamo come esempio la [figura 11.3](#), che mostra una form di login.

Username

Password

☐ Ricordami

Figura 11.3 – Una form di login con Bootstrap.

Quando creiamo una form con Bootstrap, dobbiamo racchiudere ogni oggetto di input in un tag `div`, all'interno del quale aggiungiamo un tag `label` e un tag `input`. Le checkbox fanno eccezione, in quanto il tag `input` va impostato come figlio del tag `label` invece che del `div`. Questa teoria viene messa in pratica [nell'esempio 11.9](#).

Esempio 11.9

```
<form>
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" class="form-control" id="username"
placeholder="Username">
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" class="form-control" id="password"
placeholder="Password">
  </div>
  <div class="checkbox">
    <label>
      <input type="checkbox"> Ricordami
    </label>
  </div>
  <button type="submit" class="btn btn-default">Login</button>
</form>
```

Il codice HTML è abbastanza semplice e quindi, come al solito, sono le classi

CSS a fare il lavoro. La classe `form-group` imposta i margini tra gli oggetti, mentre la classe `form-control` specifica che il controllo di input si deve vedere sotto la label. La classe `checkbox`, invece, imposta semplicemente i margini, in quanto i tag `label` e `input` sono renderizzati inline per default e quindi sono visualizzati uno di fianco all'altro.

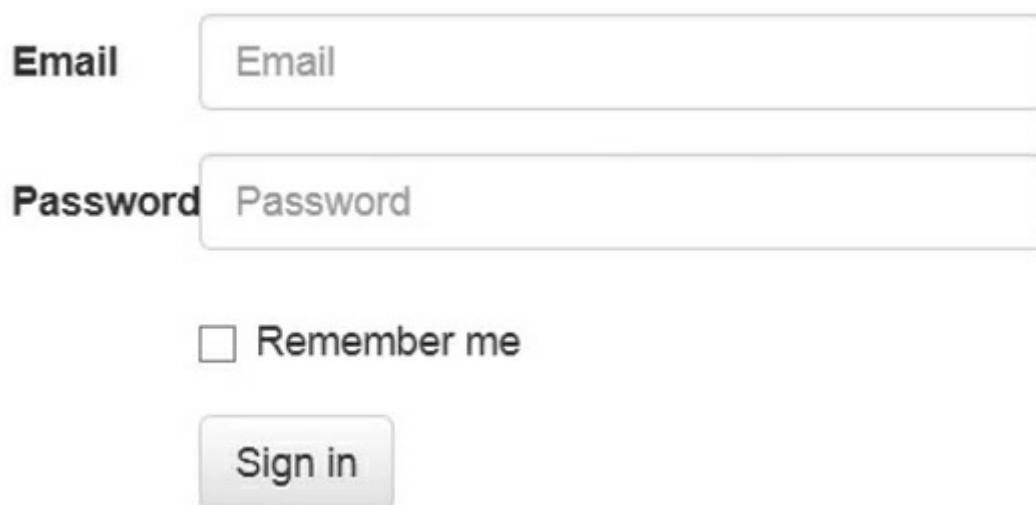
Gli oggetti di una form possono essere orientati anche in modo orizzontale, aggiungendo al tag `form` la classe CSS `form-inline`. Il risultato è visibile nella [figura 11.4](#).



A horizontal login form. It consists of a label 'Username' followed by an input field, a label 'Password' followed by an input field, a checkbox labeled 'Ricordami', and a 'Login' button.

Figura 11.4 – Una form di login orizzontale con Bootstrap.

Esiste un terzo tipo di form che è un ibrido dei due tipi visti finora: i campi sono uno sotto l'altro e le label sono di fianco al campo, così come nella [figura 11.5](#).



A vertical login form. It consists of a label 'Email' followed by an input field, a label 'Password' followed by an input field, a checkbox labeled 'Remember me', and a 'Sign in' button.

Figura 11.5 – Una form di login verticale, con label e controllo in orizzontale.

Per ottenere questo tipo di form, dobbiamo usare il codice [dell'esempio 11.10](#).

Esempio 11.10

```
<form class="form-horizontal">
  <div class="form-group">
    <label for="username" class="col-lg-2 control-label">Username</label>
    <div class="col-lg-10">
```

```

        <input type="text" class="form-control" id="username"
            placeholder="Username">
    </div>
</div>
<div class="form-group">
    <label for="password" class="col-lg-2 control-label">
        Password
    </label>
    <div class="col-lg-10">
        <input type="password" class="form-control" id="password"
            placeholder="Password">
    </div>
</div>
<div class="form-group">
    <div class="col-lg-offset-2 col-lg-10">
        <div class="checkbox">
            <label>
                <input type="checkbox">Ricordami
            </label>
        </div>
    </div>
</div>
<div class="form-group">
    <div class="col-lg-offset-2 col-lg-10">
        <button type="submit" class="btn btn-default">Sign
            in</button>
    </div>
</div>
</form>

```

Il primo passo consiste nell'aggiungere la classe CSS `form-horizontal` al tag `form`. Poi, all'interno di ogni gruppo, racchiudiamo il tag `input` in un tag `div`. Infine, ai tag `label` e `div` assegniamo le classi `col-*` (che possono essere utilizzate anche se non sono contenute in un `div` con classe `row`) per impostare quante colonne virtuali occupano.

nota

La classe `col-lg-offset-*` specifica il numero di colonne virtuali da lasciare vuote.

Una volta appreso come creare layout e form, abbiamo già compreso gran parte del potenziale di Bootstrap.

Tuttavia rimangono alcune funzionalità, che possiamo comunque menzionare anche se non possiamo approfondirle, per ovvi motivi di spazio.

Altre funzionalità

Bootstrap mette a disposizione molte classi per la cosiddetta **tipografia**. Abbiamo classi per impostare l'orientamento del testo, per trasformare il case del testo, per le immagini responsive, per colorare bottoni, per dare evidenza a dei `div`, per creare progress bar e per creare badge. Inoltre, alcuni oggetti sono già stilizzati di default come le intestazioni, i modificatori di stile del testo, i paragrafi, le liste, le tabelle e molto altro ancora.

Non solo, Bootstrap mette a disposizione anche 260 icone in formato font dalla libreria Glyphicon Halflings. Queste icone coprono le esigenze più comuni sul Web e sono disponibili tramite classi CSS apposite. Alcuni esempi sono le classi `glyphicon-plus` e `glyphicon-minus` per mostrare un'icona con i caratteri + e -, oppure la classe `glyphicon-star` per mostrare una stella e così via.

Infine, Bootstrap mette a disposizione componenti JavaScript come finestre modali, tab, tooltip, popup, carousel e altro ancora.

Tutte queste funzionalità sono perfettamente documentate sul sito di Bootstrap, quindi rimandiamo a quest'ultimo per ulteriori approfondimenti.

Con Bootstrap e jQuery risparmiamo una notevole quantità di codice ma possiamo aumentare ulteriormente il risparmio di codice usando anche altri framework che coprono necessità più specifiche, come la creazione di grafici.

Generare grafici

Un'esigenza fra le più comuni in un sito web è quella di mostrare grafici, ma HTML e CSS non prevedono nulla per la generazione di grafici. Per questo motivo dobbiamo ricorrere a librerie JavaScript di terze parti. Per semplicità di utilizzo e funzionalità, una delle migliori è senza dubbio **chart.js**, scaricabile all'indirizzo <http://aspit.co/a12>. Questa libreria mette a disposizione alcuni tipi di grafici, tra cui i tre tipi più comuni: grafico a barre, grafico a torta e grafico a linee.

Qualunque sia il tipo di grafico, la prima cosa da fare è includere il file JavaScript che contiene la libreria. In seguito, dobbiamo includere nel codice HTML un tag `canvas`, impostandone le dimensioni. A questo punto, nella parte JavaScript dobbiamo recuperare il contesto 2d del canvas e passarlo in

input al costruttore della classe `Chart`.

Ora che abbiamo un'istanza della classe `Chart`, dobbiamo solamente chiamare il metodo per la creazione del tipo di grafico. Per creare un grafico a barre dobbiamo usare il metodo `Bar`, per creare un grafico a torta dobbiamo usare il metodo `Pie` mentre, per creare un grafico a linee, dobbiamo usare il metodo `Line`.

Qualunque sia il metodo che invochiamo, i parametri da passare sono sempre gli stessi: un oggetto con i dati del grafico e uno con le opzioni di visualizzazione. Il formato di questi oggetti cambia da metodo a metodo, in quanto sia il modello di dati sia le opzioni cambiano in base al grafico.

Per quanto riguarda l'oggetto contenente i dati, nel caso di grafico a barre e a linee, dobbiamo impostare due proprietà:

- `labels`: si tratta di un array di stringhe che specifica le label sull'asse delle x;
- `datasets`: si tratta di un array di oggetti, in cui ogni oggetto rappresenta una linea e contiene i suoi valori da rappresentare sull'asse delle y, più alcune opzioni di visualizzazione.

Nel caso, invece, di grafico a torta, l'oggetto con i dati è composto da un array di oggetti dove ogni oggetto rappresenta il valore da riportare sul grafico, il colore con cui deve essere rappresentato e la label. [L'esempio 11.11](#) mostra come creare un grafico a linee.

Esempio 11.11

```
//Grafico a linee
var ctxline = $("#linechart").get(0).getContext("2d");
var data = {
  labels: ["Gennaio", "Febbraio", "Marzo", "Aprile", "Maggio",
    "Giugno", ],
  datasets: [
    {
      fillColor: "rgba(220,220,220,0)",
      strokeColor: "rgba(220,220,220,1)",
      pointColor: "rgba(220,220,220,1)",
      pointStrokeColor: "#fff",
      pointHighlightFill: "#fff",
      pointHighlightStroke: "rgba(220,220,220,1)",
      data: [20, 30, 50, 40, 10, 60]
    }
  ]
};
```

```

var linechart = new Chart(ctxline).Line(data, {});

//grafico a barre
var ctxbar = $("#barchart").get(0).getContext("2d");
var data = {
  labels: ["Gennaio", "Febbraio", "Marzo", "Aprile", "Maggio",
    "Giugno", ],
  datasets: [
    {
      fillColor: "rgba(220,220,220,0.5)",
      strokeColor: "rgba(220,220,220,0.8)",
      highlightFill: "rgba(220,220,220,0.75)",
      highlightStroke: "rgba(220,220,220,1)",
      data: [20, 30, 50, 40, 10, 60]
    }
  ]
};
var barChart = new Chart(ctxbar).Bar(data, {});

//grafico a torta
var ctxpie = $("#piechart").get(0).getContext("2d");
var data = [
  {
    value: 20,
    color: "#F7464A",
    label: "Gennaio"
  },
  {
    value: 30,
    color: "#46BFBD",
    label: "Febbraio"
  }
];
var piechart = new Chart(ctxpie).Pie(data, {});

```

Il risultato di questo codice sono i grafici della [figura 11.6](#).

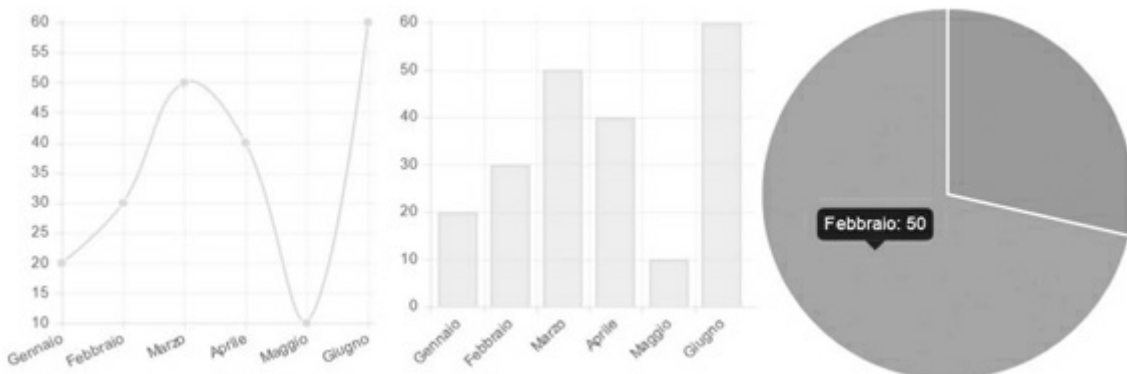


Figura 11.6 – I grafici a linea, a barre e a torta renderizzati con Chart.js

Come si intuisce dal codice [dell'esempio 11.11](#) e dalla [figura 11.6](#), ottenere grafici potenti con Chart.js è tutt'altro che complicato, quindi è comprensibile il motivo per cui suggeriamo l'uso di questa libreria.

Tuttavia questa non è l'unica disponibile sul mercato. In particolare, esistono due librerie che sono molto usate nel Web: **flot** e **Google Charts**.

La libreria **flot**, scaricabile all'indirizzo <http://aspit.co/a14>, offre meno tipi di grafici e ha delle API leggermente più complesse da utilizzare rispetto a quelle di chart.js.

Google Charts, raggiungibile all'indirizzo <http://aspit.co/a15>, è senza dubbio la miglior libreria in circolazione per quanto riguarda semplicità d'uso e ricchezza di funzionalità. Infatti, oltre ai grafici a linee, a barre e a torta, mette a disposizione grafici geografici, scatter, timeline, gauge e molto altro ancora, tutti ben documentati sul sito. Inoltre, molte tipologie di grafici sono interattive e animabili, il che rende la libreria un vero gioiello. Il tallone di questa libreria consiste nelle performance e nel codice HTML generato, che a volte ne rende l'utilizzo estremamente complicato anche in scenari molto semplici. Il nostro consiglio è quello di usare Chart.js laddove possibile e ripiegare su Google Charts laddove si vogliano utilizzare tipologie di grafici che solo questa supporta.

Templating e binding con KnockoutJS

Con l'introduzione del paradigma AJAX, c'è stato sempre un maggior sviluppo di funzionalità sul client e quindi una sempre maggior necessità di codice JavaScript da scrivere. Per ridurre la quantità di codice, si possono utilizzare framework che introducono il concetto di **templating** e **binding**. Quello più diffuso e completo è senza dubbio **KnockoutJS**. Questo framework permette di legare gli oggetti nella pagina con le proprietà di una classe JavaScript (definita ViewModel) e di mantenere i valori aggiornati tra loro. Supponiamo di avere un campo di testo legato a una proprietà del ViewModel. Tramite KnockoutJS possiamo modificare la proprietà e far sì che il campo di testo nella pagina sia aggiornato e, viceversa, fare in modo che se l'utente modifichi il campo di testo e il valore della proprietà sia aggiornato: questo sistema di collegamento tra ViewModel e oggetti nella pagina viene definito binding e il pattern di sviluppo che lo sfrutta viene

definito Model-View-ViewModel (MVVM).

Il primo passo per stabilire il binding consiste nel creare il ViewModel che espone le proprietà che vogliamo collegare alla UI. In questo caso creiamo una proprietà per il nome e una per il cognome, più una proprietà calcolata per il nome completo. Il codice della classe è visibile [nell'esempio 11.12](#).

Esempio 11.12

```
function TestViewModel() {  
    this.firstName = ko.observable("Stefano");  
    this.lastName = ko.observable("Mostarda");  
  
    this.fullName = ko.computed(function() {  
        return this.firstName() + " " + this.lastName();  
    }, this);  
}
```

Le proprietà esposte dalla classe `TestViewModel` non sono semplici proprietà bensì dei cosiddetti **observable**, ovvero delle funzioni che espongono il valore della proprietà e che permettono di impostarlo. Gli observable sono parte della magia che sta alla base di KnockoutJS. La proprietà `fullName` è un tipo particolare di observable, in quanto è in sola lettura perché calcolata.

Ora che abbiamo la classe, creiamo il codice HTML e vediamo come legare gli oggetti alle proprietà della classe [nell'esempio 11.13](#).

Esempio 11.13

```
<p>Nome: <input data-bind="value: firstName" /></p>  
<p>Cognome: <input data-bind="value: lastName" /></p>  
<p><strong data-bind="text: fullName"></strong></p>
```

La parte da notare nel codice è l'attributo `data-bind` e il suo contenuto. In questo attributo, specifichiamo una coppia chiave/valore dove la chiave è il nome di un custom binding (di cui parleremo a breve) e il valore è il nome della proprietà a cui il custom binding è legato. In questo modo, gli oggetti HTML e la proprietà observable sono collegati. I custom binding sono oggetti di KnockoutJS che stanno in ascolto sull'observable a cui sono collegati e che reagiscono al cambiamento di valore aggiornando la UI mentre, viceversa, aggiornano l'observable quando cambia la UI.

Ora che tutto è collegato, manca solo il collante, ovvero manca la parte di JavaScript che collega un'istanza del ViewModel al codice HTML.

Quest'ultimo pezzo di magia è realizzato dal codice [dell'esempio 11.14](#).

Esempio 11.14

```
ko.applyBindings(new TestViewModel());
```

Il metodo `applyBindings` di KnockoutJS prende in input un'istanza della classe `TestViewModel` e scorre il DOM alla ricerca degli oggetti con l'attributo `data-bind`. A ognuno di essi applica i custom binding specificati nell'attributo, collegandoli alle proprietà del ViewModel. A questo punto, la nostra UI e i dati nel ViewModel sono sempre sincronizzati grazie agli observable e ai custom binding.

Esistono custom binding di diversi tipi per manipolare il DOM:

- `value`: imposta e legge la proprietà `value` dell'oggetto HTML;
- `text` e `html`: impostano la proprietà `innerText` e `innerHTML` dell'oggetto HTML;
- `text` e `html`: impostano la proprietà `innerText` e `innerHTML` dell'oggetto HTML;
- `visible`: imposta la visibilità di un oggetto HTML tramite la proprietà CSS `display`;
- `css`: imposta una classe CSS se il valore dell'observable collegato è `true`;
- `enable` e `disable`: impostano se il campo è editabile o no a seconda del valore booleano dell'observable collegato.

Oltre a questi custom binding ce ne sono altri per gestire gli eventi degli oggetti, come `submit`, `click` ed `event`. Vediamo come usare `click` per i nostri scopi.

Supponiamo di voler aggiungere un bottone che mostra un messaggio con il nome completo dell'utente e un saluto. Il primo passo consiste nell'aggiungere una funzione al ViewModel che mostra il messaggio. Il secondo consiste nell'aggiungere un pulsante nel codice HTML e, nel suo attributo `data-bind`, usare il custom binding `click` per invocare il metodo. Tutto questo viene mostrato [nell'esempio 11.15](#).

Esempio 11.15

```
//JavaScript
function TestViewModel() {
    ...
    this.sayHello = function() {
        alert("ciao " + this.fullName());
    };
}

//HTML
<button data-bind=""click: sayHello">Saluta</button>
```

Come si evince da questi esempi, grazie al binding, KnockoutJS ci permette di eliminare molto codice legato alla manipolazione degli oggetti, permettendoci di lavorare esclusivamente con i dati. Questa è una freccia molto potente nel nostro arco.

Grazie al binding, KnockoutJS permette anche di mantenere legate liste di oggetti JavaScript a una lista di elementi nell'HTML. Questo torna utile quando vogliamo permettere all'utente di visualizzare e modificare i valori in una tabella o in una lista.

Supponiamo che l'utente debba inserire una lista di nomi. Nel ViewModel creiamo un tipo di observable specializzato per le liste: **observableArray**. Nel codice HTML usiamo il custom binding `foreach` per specificare che il codice HTML, al suo interno, deve essere ripetuto per ogni elemento dell'observable. Premendo il tasto aggiungi, l'utente aggiunge un elemento all'observable.

A questo punto, visto che nell'HTML abbiamo legato la lista e l'observable, KnockoutJS genera in automatico una nuova riga nella lista mentre, se eliminiamo un nome dall'observable, la relativa riga nella lista viene cancellata.

Questo è possibile grazie al binding e al templating cioè alla possibilità di creare dinamicamente frammenti di codice HTML (nel nostro caso le righe della lista dei nomi). [Nell'esempio 11.16](#) possiamo sperimentare il caso in questione.

Esempio 11.16

```
//Javascript
function TestViewModel() {
    var self = this;

    self.names = ko.observableArray([
```

```

    { name: "Stefano Mostarda" },
    { name: "Daniele Bochicchio" }
  ]);

  self.addName = function() {
    self.names.push({
      name: "Nuovo Nome " + new Date().getSeconds()
    });
  };

  self.removeName = function() {
    self.names.remove(this);
  }
}

//HTML
<h4>People</h4>
<ul data-bind="foreach: names">
  <li>
    <span data-bind="text: name"> </span>
    <a href="javascript:;"
      data-bind="click: $root.removeName">Elimina</a>
  </li>
</ul>
<button data-bind="click: addName">Add</button>

```

Il ViewModel è abbastanza semplice; il metodo `addName` aggiunge un nome randomico alla lista, mentre il metodo `removeName` rimuove l'elemento selezionato dall'utente dalla lista stessa.

Il codice HTML, al contrario, merita un approfondimento. Innanzitutto vediamo che al custom binding `foreach` passiamo in input l'observable con i nomi. Successivamente, tutto quello che è contenuto all'interno del tag `ul` (cioè il template) viene ripetuto per ogni nome. Quando viene fatto il binding nel template, il contesto è l'oggetto corrente dell'array, quindi le proprietà in binding (`name` nel nostro caso) si riferiscono a questo. Poiché il contesto è l'oggetto dell'array, se legassimo il custom binding `click` al nome del metodo per cancellare un nome, otterremmo un errore. Per questo motivo usiamo la parola chiave `$root` che si riferisce al ViewModel.

nota

La proprietà `name` è in binding nonostante non sia un observable. Questo significa che se cambiassimo il valore dell'elemento dal

ViewModel, la UI non verrebbe aggiornata. Inoltre, il metodo `removeName` passa in automatico l'elemento a cui è legato alla funzione, quando questa viene invocata.

KnockoutJS e il pattern MVVM permettono di sviluppare applicazioni web concentrandosi più sulla logica di business e meno sulla logica della UI. Grazie a binding e a templating, possiamo creare funzionalità interessanti con poche righe di codice. Queste funzionalità, comunque, non sono offerte solo da KnockoutJS, ma anche da altri framework, che spesso spingono l'utilizzo di JavaScript e HTML a un livello ancora più avanzato. Ne sono un esempio i framework per creare Single Page Application.

Single Page Application

Grazie alle nuove funzionalità offerte da JavaScript, HTML e CSS, negli ultimi anni è nato un nuovo paradigma di sviluppo sul Web: Single Page Application (SPA d'ora in poi). Una SPA è un'applicazione che gira interamente sul client e sfrutta il server solo per recuperare i file JavaScript, CSS e HTML e per recuperare i dati.

Il funzionamento di una SPA è relativamente semplice. All'inizio viene scaricata sul browser una semplice pagina HTML (definiamola **masterpage**) e i file CSS e JavaScript necessari a inizializzare l'applicazione. Successivamente, il motore JavaScript scarica il frammento di HTML relativo alla homepage dal server e scarica i relativi dati. Il frammento di HTML viene impostato come contenuto della masterpage e legato tramite binding (per esempio con KnockoutJS, come abbiamo visto in precedenza) ai dati scaricati dal server. In questo modo otteniamo una normale pagina che interagisce con l'utente.

Quando dobbiamo navigare verso un'altra pagina dell'applicazione, il JavaScript intercetta la navigazione, scarica la pagina HTML relativa all'url verso cui abbiamo navigato, la imposta come contenuto della masterpage (rimpiazzando quindi la vecchia pagina) e la lega a nuovi dati che scarica dal server.

Una volta assimilato questo meccanismo, è facile intuire quali siano i punti di forza e i punti deboli di questo paradigma di sviluppo.

Dal lato dei punti di forza, ci sono sicuramente le performance e l'usabilità. Essendo l'applicazione eseguita principalmente sul client, il server

è scaricato di molte responsabilità. Tutto quello che deve fare è fornire frammenti di HTML e dati: un bel risparmio rispetto all'elaborare le pagine a ogni navigazione. Inoltre, l'applicazione è molto più veloce perché si riduce la latenza di rete. Non è un caso se stiamo assistendo alla creazione di applicazioni web sempre più simili ad applicazioni desktop grazie alle SPA.

Il vero punto debole delle SPA è sicuramente l'enorme quantità di codice JavaScript da scrivere e mantenere. Se non si ha molta dimestichezza, questo ostacolo può sembrare di poco conto all'inizio ma, una volta che l'applicazione cresce, diventa insormontabile se non si architettano bene le cose sin dall'inizio.

Per ridurre i problemi legati all'enorme quantità di codice JavaScript, possiamo utilizzare framework per SPA esistenti, come AngularJS, DurandalJS e il nuovo Aurelia. I primi due framework sono **legacy**, ovvero girano anche su browser meno recenti, come Internet Explorer 8. La nuova versione di AngularJS (ancora non uscita al momento della scrittura di questo libro) e Aurelia (che è il nome della nuova versione di DurandalJS ed è in versione alpha), invece, supportano solamente i browser più moderni. A prescindere dalla scelta, creare una SPA senza partire da un framework esistente sarebbe un errore grave. Parlare di questi framework richiederebbe un libro intero, quindi non ci addentreremo in una spiegazione di come funzionano. L'importante è capire che questo modello di sviluppo delle applicazioni web sta prendendo sempre più piede e che abbiamo a disposizione armi notevoli per creare queste applicazioni nel migliore dei modi.

Conclusioni

In questo capitolo abbiamo visto che in tanti anni di sviluppo web sono stati creati framework che semplificano lo sviluppo sotto diversi punti di vista. L'unione di jQuery e Bootstrap ci permette di partire da una situazione molto favorevole per quanto riguarda la creazione di una nuova applicazione web.

Inoltre, framework come quelli per le SPA e KnockoutJS ci permettono di spostare sempre più funzionalità sul client, mantenendo al minimo il codice JavaScript da scrivere per l'infrastruttura e permettendoci di concentrarci solo sul codice di business.

Il corretto utilizzo di tutte queste librerie può spesso risultare vincente per la riuscita di un progetto.

Siamo giunti alla fine della nostra esplorazione alla scoperta delle principali caratteristiche di HTML e JavaScript.

Ci auguriamo che abbiate gradito la lettura di questo libro e che, come noi, crediate che l'evoluzione del Web sarà positiva per tutti, sviluppatori e utenti finali. Speriamo di aver stimolato la vostra curiosità e persino convinto qualcuno di voi a iniziare a utilizzare le potenzialità di HTML e a seguirne le evoluzioni.

Il futuro di Internet è qui e noi siamo già pronti a viverlo.

Vi ringraziamo per aver letto questo libro e... ci vediamo in rete!

Informazioni sul Libro

HTML5 è un insieme di nuovi standard, appena approvato, che consente di sviluppare applicazioni web moderne. Porta con sé diverse novità, che vanno verso l'adozione di una serie di specifiche, condivise da browser e piattaforme tra loro diversi, che semplificano la vita degli sviluppatori web, sempre alle prese con le difficoltà legate ai vari browser.

In realtà, HTML5 è molto di più e include una serie di nuove caratteristiche, che abbracciano anche CSS e JavaScript e impattano sul mobile, una fetta sempre importante della navigazione mondiale: districarsi tra le nuove specifiche può non essere un'impresa semplice.

Con uno stile chiaro, pratico e ricco di esempi, questo libro si pone come una guida ideale, sia per principianti sia per chi vuole conoscere le novità di HTML5 e delle tecnologie correlate.