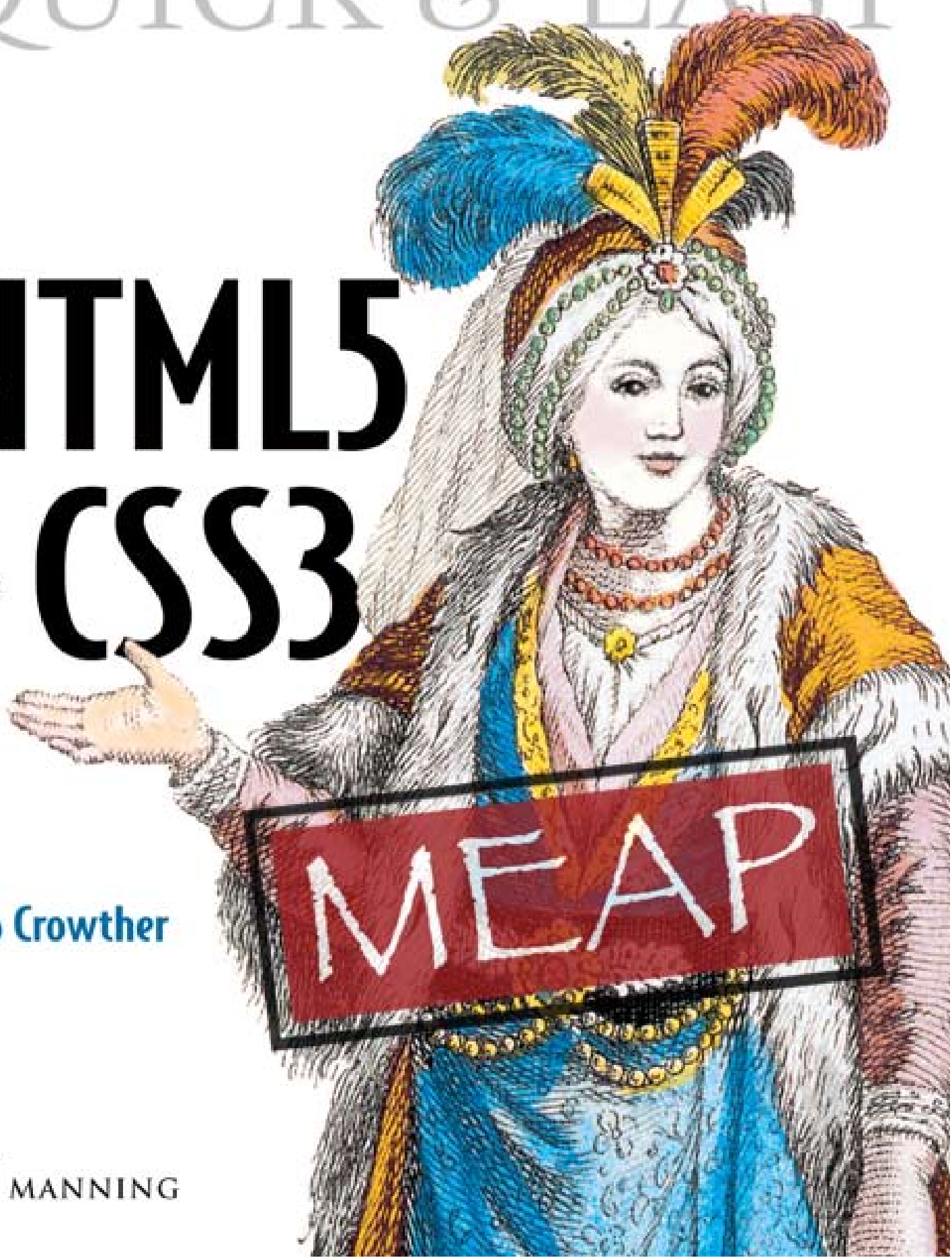
QUICK & EASY

HTML5 & CSS3



Rob Crowther

MANNING

MEAP Edition
Manning Early Access Program
Quick and Easy HTML5 & CSS3 version 6

For more information on this and other Manning titles go to
www.manning.com

1

How HTML5 and CSS3 revolutionize web development

Most web pages can be viewed easily on your desktop computer, your mobile phone, and perhaps even your game console, MP3 player or television, because they share common languages – not English, but Hyper Text Markup Language (HTML) and Cascading Style Sheets (CSS). The web is the largest shared information resource humanity has ever built, HTML and CSS are the technologies to know if you want to make your own contribution.

The HTML describes the content of the pages – what are the headings, the paragraphs and the lists. CSS describes what the content should look like, blue text on a white background, a menu on the right, and so on. In this book you're going to learn how to use the latest standards for building web pages – HTML5 and CSS3. In the next few chapters

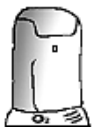
you will get an accelerated introduction to the old standards, but the majority of the book will be devoted to the new features in HTML5 and CSS3. To start with, in this chapter, you're going to learn the basics of what HTML and CSS are, why new versions of them are needed, and what problems they can solve. By the end of this chapter you will know:

- What HTML and CSS are and their role in the World Wide Web
- Why it's time for new standards to keep up with current uses of the web
- Some common features web authors are having to implement today using custom code, workarounds and plug-ins that are part of the web platform in HTML5 and CSS3

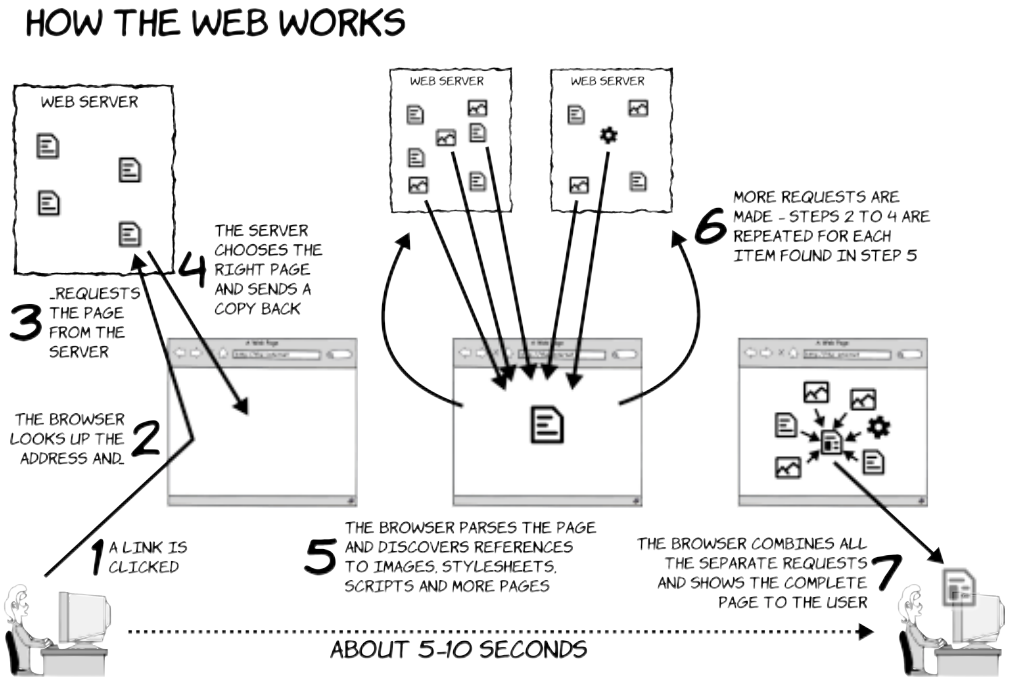
Before you get started on the benefits HTML5 and CSS3 bring, we'll briefly review what HTML and CSS are. In the next section you'll learn just enough about what HTML and CSS are to understand the rest of this chapter.

1.1 What are HTML and CSS?

As you learned above, HTML and CSS are languages used to make web pages. The web itself is based on a client-server model, the HTML and CSS files are stored on web servers, and the web browser clients ask the servers for the documents they want. The whole process is described in the diagram below.

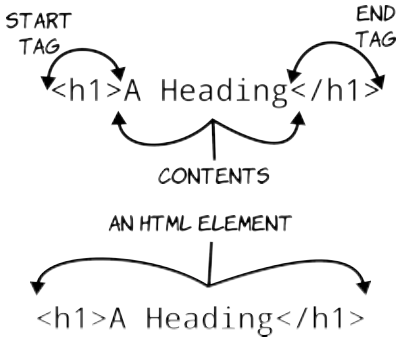


If you have past experience with HTML and CSS you can skip ahead to the next section. If you've never seen HTML and CSS syntax before, you can learn more in appendices B and C.



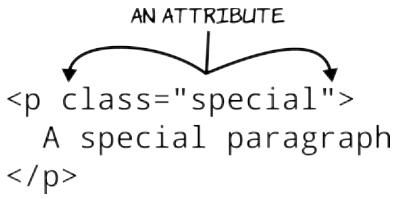
HTML is a text based markup format. Some characters in text the have special meaning, they are the markup, notably the angle brackets < and >.

Some letters within an angle bracket creates a tag. You can see in the image above there are two tags - <h1> and </h1>, a start tag and and end tag. An end tag always looks like a start tag except with a slash after the opening angle bracket.



The combination of a start tag and an end tag defines an element. Everything between the two tags is referred to as the contents of the element

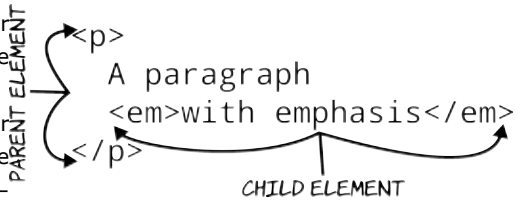
Start tags can also have attributes, a name optionally followed by a value. An attribute is used to select between different options of element function or to provide extra information about what the element means.



Some elements need to have at least one attribute to be any use, for instance a link element has an attribute which contains the address of the HTML page it links to.

Elements can have as their content other elements, we say, "the p element contains the em element".

Any element which contains other elements is said to be the parent of those other elements, those are in turn its children – the idea is the elements form a tree structure, like a family tree.



Now you have enough understanding about what HTML looks like to be able to understand a more formal definition of what it is. HTML is a definition of what words can appear within the angle brackets, it defines:

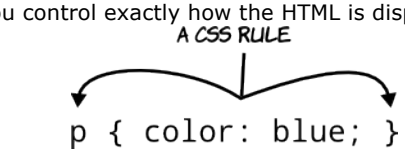
- What the valid tag names are.
- What the valid attributes are on each of those elements.
- How those tags can be composed into a tree of elements to make a document.
- What each of those elements means.

In the examples above we have a level one heading element, h1, a paragraph element, p, and an emphasis element, em. Other common elements include links, lists, blockquotes, tables, buttons and form controls.

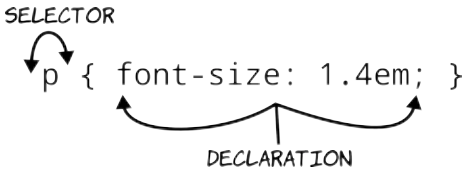
In HTML the ultimate parent element of the tree is always the <html> element. The <html> element is the ultimate daddy!

A lot of the power of HTML comes from the fact that it can easily be turned into a visual representation in a web browser. CSS is how you control exactly how the HTML is displayed.

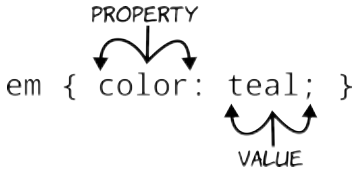
Like HTML, a CSS file is also plain text, it can be created with a tool as simple as Notepad. CSS is made up of rules like the one shown here. This rule sets all p elements to be drawn blue.



A CSS rule consists of a selector, which determines which elements the rule will apply to, and a semi-colon separated set of declarations inside curly brackets. For simplicity, the rule shown here has just one declaration, but any number are allowed.



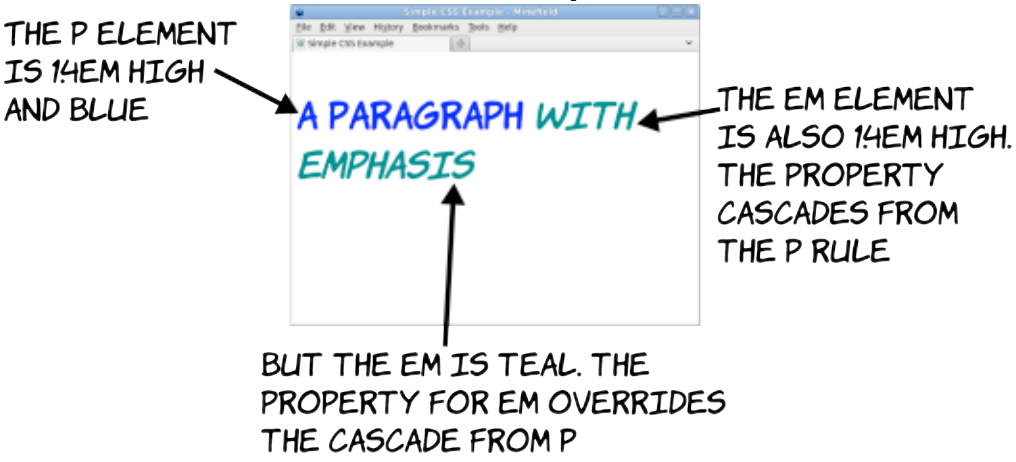
Each declaration has a property and a value, separated by a colon. Here you see a rule which declares that all elements will have teal colored text.



Other common declarations set backgrounds, borders and position. A key factor in CSS is that the rules cascade, hence the name 'Cascading Style Sheets'. The details of how this work are quite intricate, but the basics are quite easy to grasp with a short example. You can apply the CSS examples used above to the last HTML example. Here are the two listings again, the two declarations for p have been combined into a single rule:

```
p { color: blue; font-size: 1.4em; }
em { color: teal; }
```

```
<p>A paragraph
  <em>with emphasis</em>
</p>
```

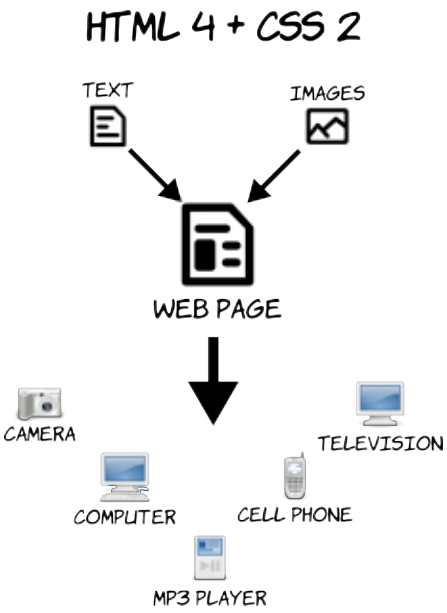


You should now have a basic grasp of what HTML and CSS are which will equip you for understanding the rest of the chapter. Everything shown in the coming sections can be created out of HTML and CSS with just a text editor. If you need to know more before continuing, take a look at Appendix B for more HTML and Appendix C for more CSS.

1.2 Why we need new standards

Our current web was built on the back of HTML version 4, first introduced in 1997, and CSS version 2, first introduced in 1998 when there were fewer than 20 million websites. Back then web pages consisted mostly of text and images and operated largely independently of the users own files and folders, and the standards of the time reflect that.

HTML4 has many elements suitable for text, and also has an image element to embed graphics directly in the rendered page, so it was satisfactory for most things people wanted to do with web pages.



These days people expect much more from the web – they want to take pictures and share them with their friends, listen to music and watch television and movies. They want not just information, but to check their email, chat with their friends and collaborate on work projects using Word Processors and Spreadsheets, and they want these web applications to more closely resemble the ones they are used to using on their desktop PCs.

Terminology

Plug-in: A generic extension method for HTML allows the page author to indicate some embedded content which is to be rendered by an external program. The web browser hands over control of that region of the web page to the external program. This external program is referred to as a plug-in.

JavaScript: A programming language which can be embedded within HTML. It can modify web pages in the browser, after they are loaded from the server. The combination of JavaScript and HTML was initially referred to as Dynamic HTML (DHTML), in recent years DHTML has been encompassed by a new hybrid, AJAX (Asynchronous JavaScript And XML).

DOM: The Document Object Model, or DOM, the representation of the HTML Element tree that is available to JavaScript for manipulation.

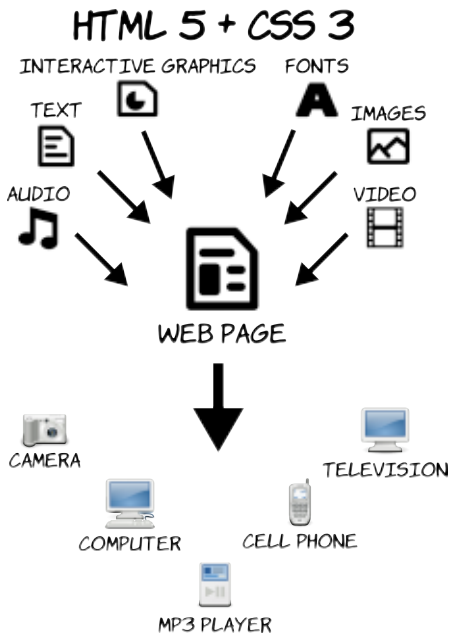
To satisfy the user demands for more desktop like web apps, web developers are relying on third party plug-ins and non-standard hacks using JavaScript. This can lead to a number of issues:

- It is unlikely developers can test their hacks and workarounds on the wide variety of devices which can be used to browse the web. This doesn't just mean old or unusual devices, it can also apply to new devices which are fully compatible with the standards, for example the iPhone when it first came out.
- Web authors are individually writing custom code to solve common problems, not only does this duplication of effort result in inconsistency across the web for users but it takes away from time spent improving applications.
- If common behaviors are implemented by the browser they will be more efficient that if they are created in JavaScript by web developers. There is the possibility of having direct access to the hardware (e.g. for graphics acceleration).
- Plug-ins create another layer of compatibility which web authors have to deal with – not only do they have to think about users who have might have older browser versions, they have to worry about the version of the plug-in too.

Browser vendors are aware of the limitations of HTML4 and CSS2 and so they seek to add functionality to their products to make more things possible. This has been a common feature in the development of the web, but was particularly acute during the 'browser wars' of the late nineties.

Back then standards lagged far behind the demands of web users, so the makers of the two most popular browsers, Netscape and Microsoft, competed by adding new features. Unfortunately, with no standard, the features they both added were largely incompatible.

Developers ended up having to support two different versions of HTML and CSS if they wanted their sites to have the latest features in both major browsers. Users of other browsers were often excluded completely, counter to the ideals of knowledge sharing upon which the web was initially built.



For the first time since HTML 4.0 was released, back in 1999, we have serious competition in the browser market, and this is fueling great innovation. However we also want to avoid the chaos of incompatibility that marked the last browser wars, and to do this we need new standards. HTML5 and CSS3 add standards for audio, video, interactive graphics and custom fonts in web pages

It's best for everyone if browser vendors can agree on what the specs should be, then concentrate on providing the best implementation of those specs.

The benefit of having an agreed standard can be seen by looking at some areas that are poorly covered in the existing standards. For example, every browser has a method for making key presses available to your JavaScript code. However, the standard specifies little more than this – that the browser should make these events available, not how or in what order. As a result, every browser has a different implementation, the same browser often differ in behavior when run on different operating systems or keyboard layouts. This is a real problem for things such as online games which require keyboard input.

THE GREAT STANDARDS DEBATE

I don't see the problem with the proprietary extensions added during the browser wars

Proprietary extensions tie you to particular browsers and operating systems, the web was built on choice.

Why shouldn't vendors be free to implement whatever features their users want, and developers just choose what to use? Standards hold back progress!

If you wrote a web app that depended on Netscape only features, you would have to stick with that even if a better browser came along. Or re-write every 6 months..

Even today there are companies who can't upgrade their user's browsers because the "Web Apps" they bought or built rely on non-standard technology. They're stuck using an obsolete browser, no fixes or updates., for everything.

They might as well have built desktop apps

But the browser wars were the source of huge innovation. Java-Script, AJAX, SVG, - all of these started out as vendor extensions.

We again have healthy competition in the browser market again today,

but all the vendors realize the key to getting their new features used in the real world is to get other browsers to implement it.

And the key to getting other browsers on board is to agree standards.

Make character heads face the text, feel free to re-arrange the sides the characters appear for better balance graphically, currently they're arranged on ideological grounds

HTML5 and CSS3 brings the web forward by providing a new baseline for web browser functionality. It does this in a way which is backwards compatible with existing web content, and also much more rigorous and complete than the standards they supersede.

With HTML5 and CSS3 web authors can then devote less time to detecting and mitigating browser differences and more time writing their applications.

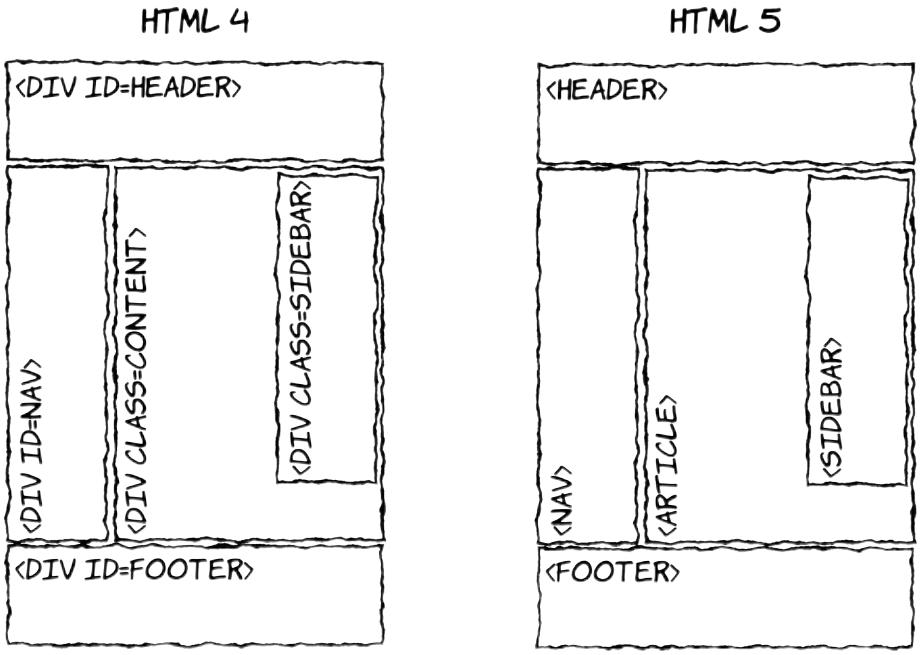
So far you've learned what HTML and CSS are and why having agreed standards for them is important. Now you will learn about some of the specific benefits HTML5 and CSS3 bring compared to the previous versions.

1.3 HTML5 and CSS3 In Action

HTML5 and CSS3 solve a lot of problems commonly faced by web developers as they try to build web sites and web applications. Below you will see several specific examples of these problems, a typical HTML4 or CSS2 hack or work-around and the improved, standardized way of solving the problem in HTML5 or CSS3.

1.3.1 Page structure

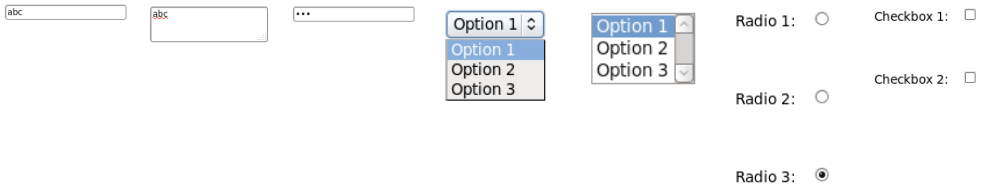
There are very few HTML4 elements for structuring sections of a page. You have to resort to the semantically neutral DIV element, differentiated only by appropriate ID or CLASS attributes. The editor of the HTML5 specification took advantage of Google's vast library of web content to identify the most commonly used attributes and used that information to suggest new elements.



HTML5 provides semantically meaningful elements for common areas of the document. You can now describe the elements of your layout in an unambiguous way, so that information is available to browsers, search engines and assistive technology. You will learn more about new elements for page structure in Chapter 2.

1.3.2 Forms and validation

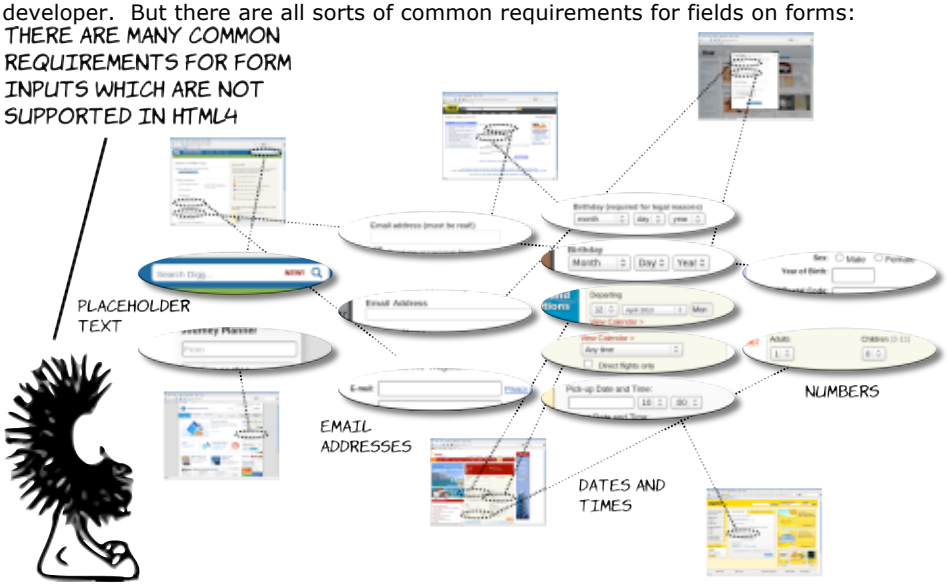
There are two basic HTML4 input types: select from a list (drop down or check boxes) or the input element. This is the basic set of options:



Single line text | Multiple line textarea | Password | Drop down select list | Scrolling select list | Radio buttons | Check boxes

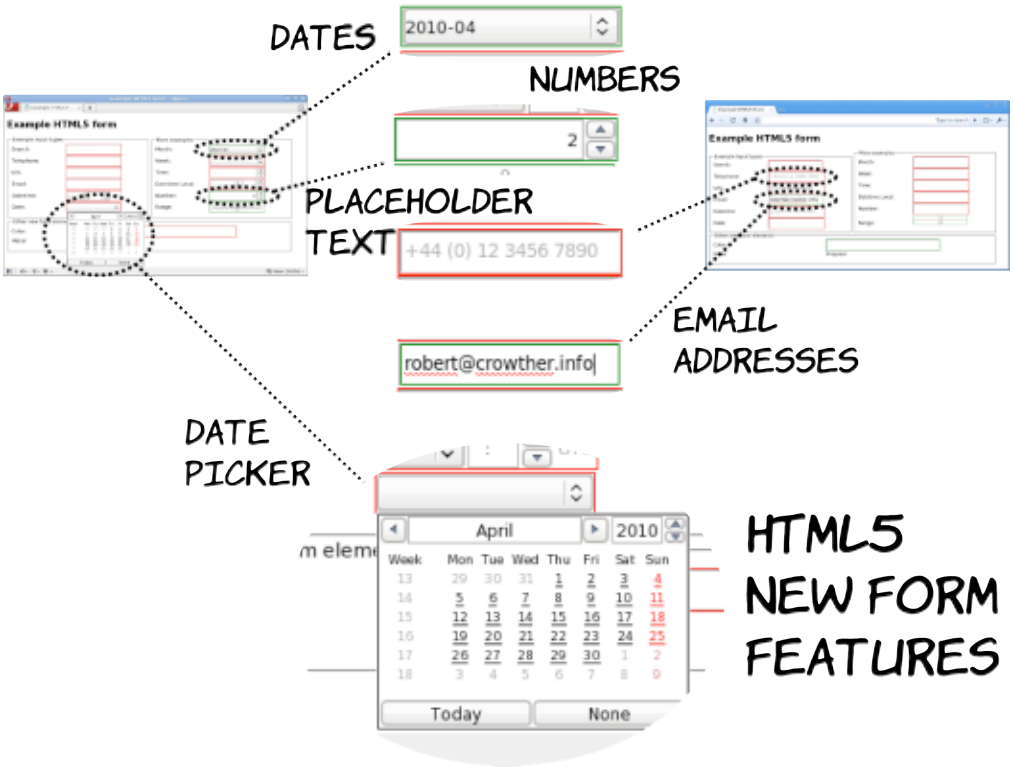
Please put labels in 'comic' font with arrows pointing to the input area

HTML4 forms offer few options for flexible input, text input needs to be validated by the developer. But there are all sorts of common requirements for fields on forms:



If this diagram can be rearranged to be more legible, please do so

If you want a particular field to be a number or a date, you will need to write custom logic in JavaScript to enforce it. If you want to present a calendar style date picker then you will also have to add custom code. HTML5 adds progress and meter as well as several additional input types: search; tel; url; email; datetime; date; month; week; time; datetime-local; number; range; color:



In addition, HTML5 allows you to declare required fields and valid ranges in your markup, and adds a CSS pseudo class so you can target styles at valid and invalid fields. The new form elements will be covered in Chapter 3.

CSS Pseudo Classes

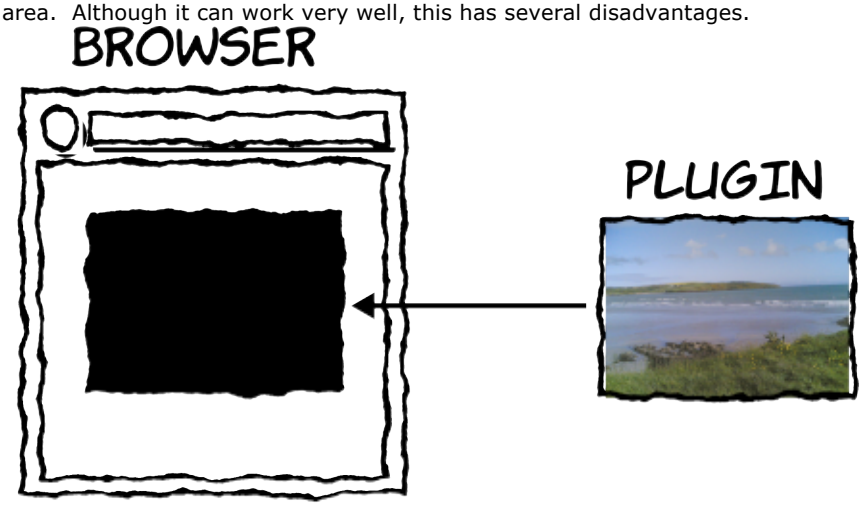
Pseudo classes are similar to regular CSS element selectors, allowing you to style more abstract items than the elements themselves. These can be behaviors or states, or specific properties, or relationships with other content. A pseudo class is indicated by a colon in the selector, for example:

```
p:hover { /* Applies only when the pointer is over the element */ }
input:invalid { /* Applies only to inputs with invalid data */ }
```

For more information, see Appendix C..

1.3.3 Plug-ins for interactive graphics audio and video

When you need to do things that are not accounted for by the standards, you have to resort to plug-ins. For example, you may want to display a graph which updates as a user adjusts some variables on the page, or play some music, or a video. The browser surrenders a portion of it's display to an external application, which then takes care of rendering content in that area. Although it can work very well, this has several disadvantages.



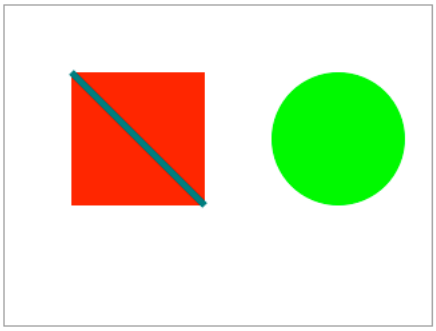
The content rendered by the plug-in is like a 'black hole' to the browser. This has some undesirable consequences:

- Functionality the user takes for granted within the browser – for example their preferences, keyboard shortcuts and in page search – will not be available if they happen to be interacting with the plug-in content.
- The browser cannot apply styles to anything within the plug-in content and other page elements can sometimes have problems if they need to extend over the area in control of the plug-in, for example a menu drop down might not be able to drop 'over' a playing video.

HTML5 adds three new elements: video, audio and canvas. Shown here is an audio element in Firefox.

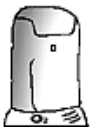
The video and audio elements are intended to make embedding these things in web pages as easy as it is to embed images today. No special plug-ins are required, the ability to view movies and listen to music is built in to the browser.

The canvas element allows you to draw directly into the browser window using JavaScript, for example a bar chart in response to user input.



In addition, HTML5 allows the embedding of SVG – an XML dialect for describing two dimensional graphics.

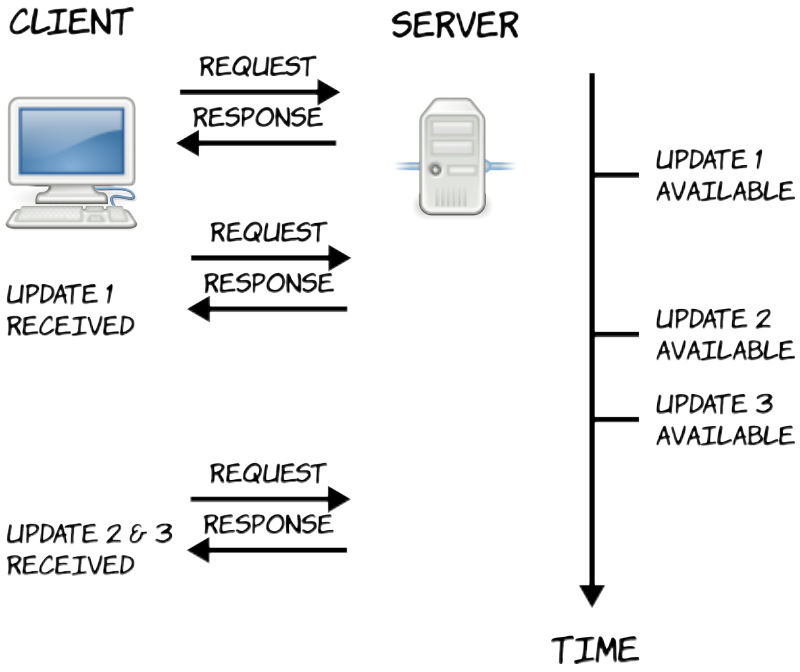
The SVG elements are available in the DOM in the same way as regular HTML elements so they can also be manipulated by JavaScript. All of this happens within the browser, so it is always subject your browser preferences and interacts well with other content on the page.



You will learn more about the audio and video elements in Chapter 5, canvas and SVG will be covered in Chapter 4.

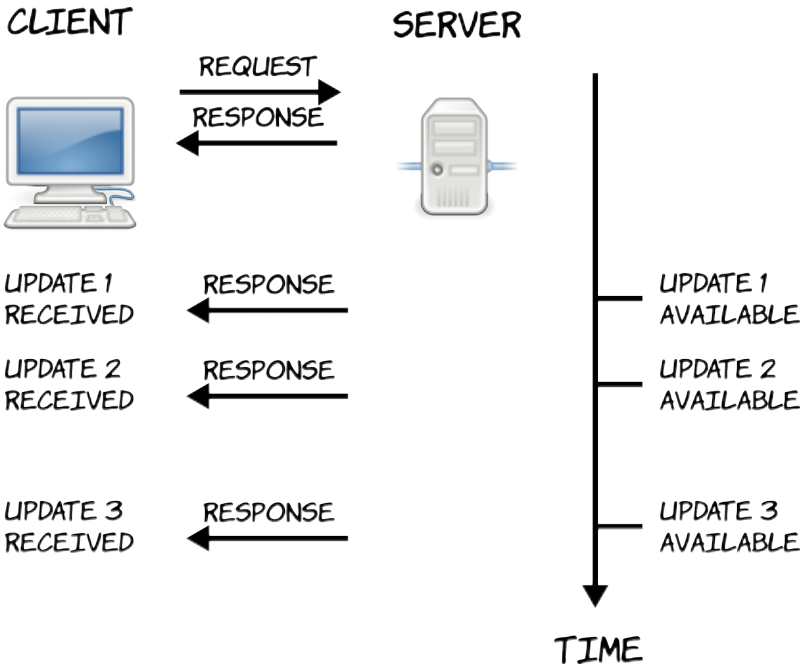
1.3.4 Real time communications

In HTML4 everything is driven by a request from the browser. The server has no way to send an update to the browser unless the browser first requests one.



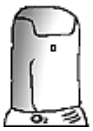
This is a very common problem: email, chat, multi-player games, weather or stock ticker applications all need to send regular updates to the user.

You would typically work around this by reloading a page on a schedule, or by performing an AJAX request back to the server at regular intervals. HTML5 provides WebSockets, a new protocol for lightweight server to client communication.



With HTML5 a web application can create a socket which allows the server to communicate with the browser. Once the connection is established, the server can send an update to the browser at any time. The browsers will get the update immediately, no polling is required.

Sockets are a networking concept that allow different services on a single computer to have logically independent communication with the outside world. The idea is similar to a large office which has a single phone number, but each desk is reachable by a unique extension.



Web Sockets, along with some other communication features of HTML5, will be covered in more detail in Chapter 6.

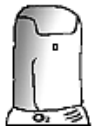
1.3.5 Rich Internet Applications

In addition to solving common problems, HTML5 extends the reach of the browser into new areas. Rich Internet Applications was a term coined in 2002 by Macromedia (now Adobe), developers of the Flash plug-in.

The idea of RIAs is to take advantage of the simple cross platform deployment model and standards based approach of the web, but enable closer integration with your computer, like a desktop application.

RIAs threaten to take the web back into custom applications, rather than a general purpose web browser. To counter this threat the HTML5 spec adds a number of features which make a web browser more suitable for desktop apps:

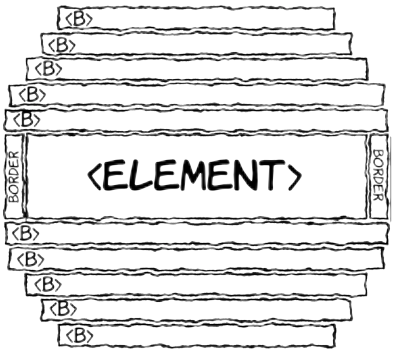
- Offline support: web applications can signal to the browser that certain pages and resources should be available when the user has no internet connection.
- Operating system integration: in HTML5 web applications can integrate seamlessly with your desktop, allowing you to 'drag and drop' your files on to the app.
- Storage: as a complement to the offline support, HTML5 adds several APIs for local storage – web apps can store several megabytes of your data, and that data will still be there even if you restart the browser.



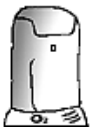
You will learn more about the APIs for offline applications, drag and drop and web storage will be covered in Chapter 7.

1.3.6 Rounded corners

With CSS2 you had a number of options if you wanted rounded corners on particular elements in your design. A common approach is shown below, a series of elements of one pixel height and successively decreasing width is stacked above and below your element to give the impression of rounding.



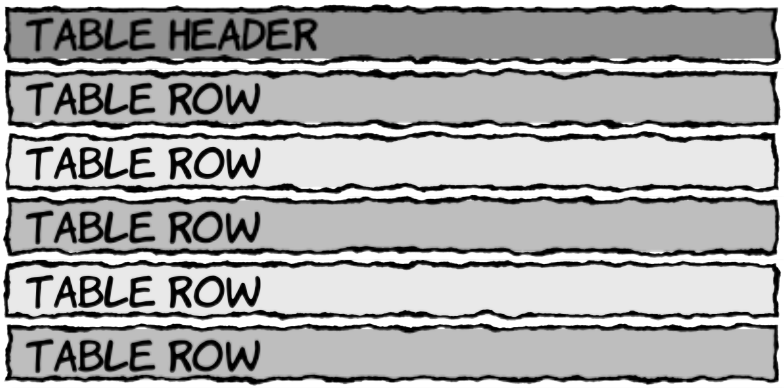
This approach adds a lot of pointless, empty elements to your document. These are elements the browser has to keep track of even if you are adding them with a script. With CSS3 you can apply rounding directly to the border of the element – no extra code is required in the page.



You will learn more about rounded corners and other border and background effects with CSS in Chapter 10.

1.3.7 Striped tables

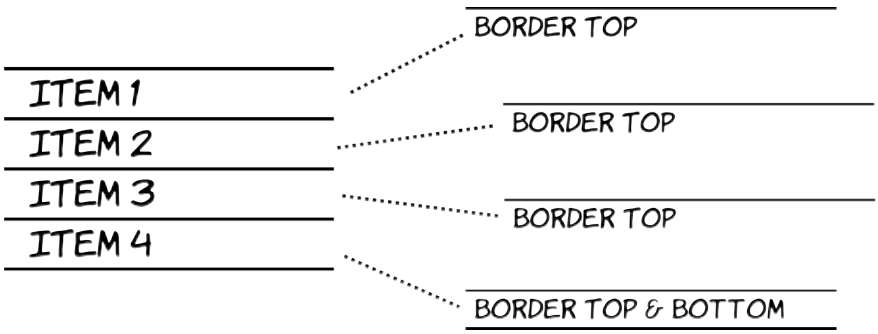
A common design element in the web 2.0 look is to use background color to differentiate table rows instead of borders. In the past you'd have been stuck manually adding class names to alternate table rows in your markup. This can be complicated if you're building your table from several different sources, or if you need to add a new row in the middle of the table.



CSS3 helps web designers keep their markup separate from their presentation by providing new selectors. The nth-child selector allows you to specify a rule which only applies to the even elements, or the odd elements, or every third or fourth or fifth element. If you remove or add a row to the table the browser will take care of which rows have which style.

1.3.8 Special styling for last and first of a set

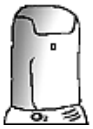
When styling a set of elements you often want to apply a special style to the first or last element of a set. For example you may want to have a top border on every element, but have an extra bottom border on the last element to enclose the set as in the diagram below.



With CSS2 you have to add a class name the last element in your markup, or use JavaScript to determine which is the last element once the page has loaded and apply the style.

As with the striped tables above, this approach can be fragile. If you insert a new element with JavaScript you have to add extra code to handle the situation where it's the last element.

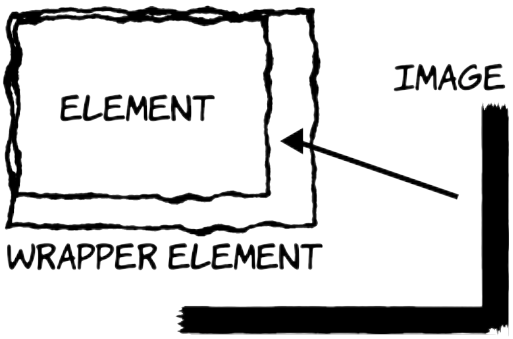
In CSS3 there are four new selectors for this situation: first-child; last-child; first-of-type; and last-of-type. Now the browser has enough information to apply the styles you want without your help. You can add and remove elements without worrying about it because the browser always apply the styles correctly.



To apply a style to particular element out of a set, like you've seen in 1.3.7 and 1.3.8, you need to be able to construct a selector that will target those elements. CSS selectors be covered in more detail in Chapter 7.

1.3.9 Applying drop shadows

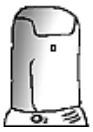
Drop shadows are a very common design element in 21st century web design. A drop shadow gives an element a 3D look, helping it to stand out from the page. Pages look more interesting as a result, and interesting looking pages are what designers are after.



A common technique is to add a wrapper around your content, then place a shadow image in that. Careful positioning of the wrapper allows you to make the element appear like it has a drop shadow.

Like many of the hacks already described, this approach adds unnecessary markup to your page, it can also cause problems if the shadow itself has to overlay other content.

CSS3 has built in support for shadows. No extra elements are required, just a rule in your CSS file. This means the shadow will always match the size of the element with no extra work on your part, and it will work well with other browser applied styles.

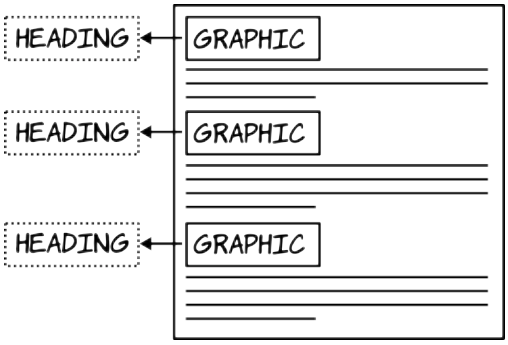


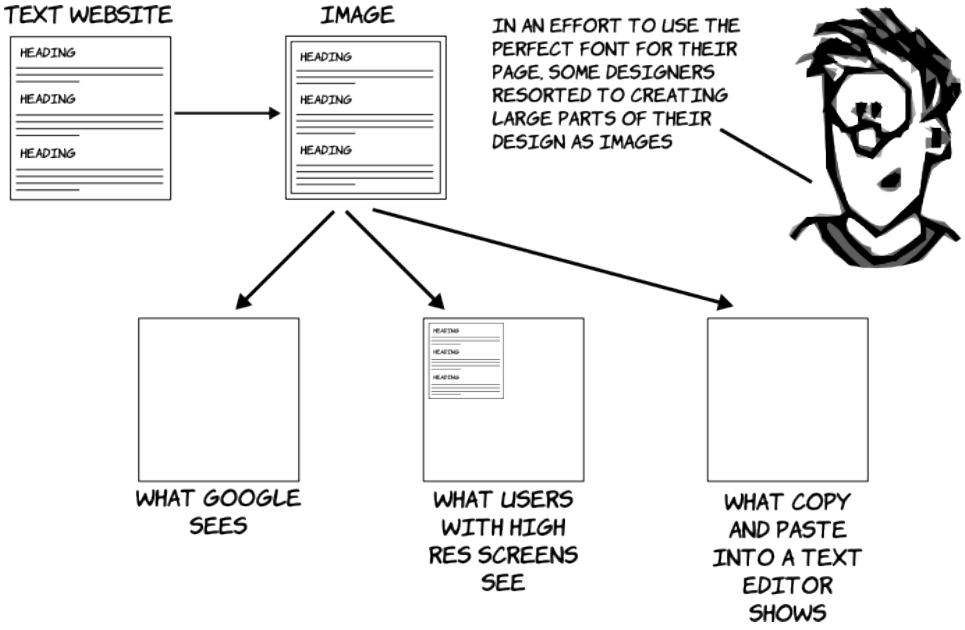
You will learn more about drop shadows with CSS in Chapter 10.

1.3.10 Designing with custom fonts

Normally a web page uses fonts installed on the user's computer, which are selected based on a list of recommendations provided in CSS by the designer. There are very few fonts which are available on all operating systems, so pages can end up looking very different if the font the designer intended doesn't exist on the user's computer. These two screenshots demonstrate how different a web page can look if the font the designer intended isn't available:

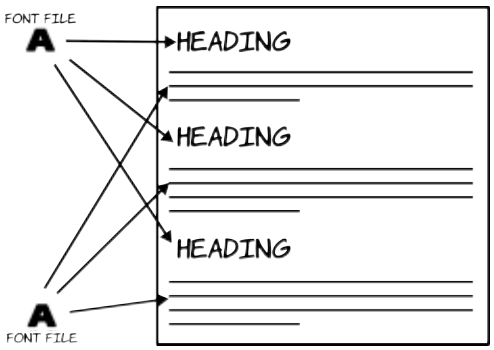
Although there were some mechanisms for working with custom fonts in CSS2 they were not well supported. Whenever a designer wanted to use a custom font, for branding reasons or perhaps just to fit a theme, their only option was to use the font on their own computer to make an image and replace the text on the page with the image.



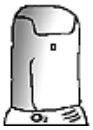


Although more sophisticated approaches have arisen over recent years, using Flash to generate the text dynamically for example, they still suffer from one or more of the basic issues that arise once you start replacing text with images of text. It also means that to update the content of the website you need graphics software rather than a simple text editor. If your website was to be served in several languages you'd need images of the text in each one.

Despite the difficulties surrounding text replacement techniques, extensive use has been made of them. Their popularity has led to a reexamination of the font-face rule which had initially been implemented in Internet Explorer 4. Using font-face, designers can specify fonts in the same way they link to background images in their stylesheets.



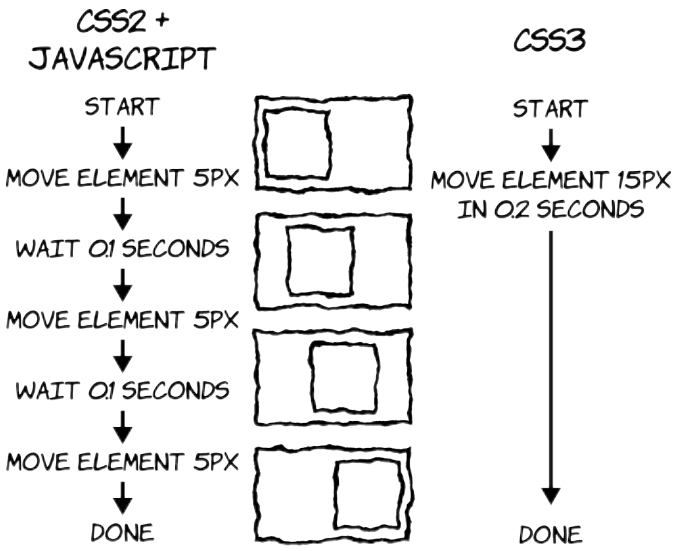
Now designers have the ability to make their text appear in the font of their choosing for every user, yet have it still be text. It can be treated by the browser as any other text, all the built in browser functionality for selecting and sizing text works, and search engines see all the content.



Fonts and other advanced text formatting features will be covered in detail in Chapter 11.

1.3.11 Easy transformations and animations

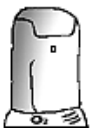
When users interact with elements it is often a better experience for the elements to animate slightly – instead of a menu springing immediately into view, it 'drops in' to view. For these sorts of effects in CSS2 you have to resort to JavaScript. If the menu will end up 100 pixels high, you set a script which runs every few milliseconds and adds 10 pixels to the height of the menu until it reaches its full height.



CSS3 allows you to declare a transition for CSS properties such as width, height, margin, border and position, in response to the user moving the mouse pointer over an element, or that element being selected for keyboard focus.

By taking advantage of CSS3 you can spend less time writing JavaScript to provide superficial animations and more time concentrating on your application logic.

Your animations are now described explicitly in CSS rather than implicitly in your JavaScript. The browser can use special purpose graphics algorithms to render the entire animation which is much faster than doing it step by step with JavaScript. It also opens up the possibility of using the computer's graphics hardware for animation without involving the CPU at all.



You will learn about CSS transitions, transforms and animations in Chapter 10.

1.4 Summary

You've covered a lot of ground at a high level in this chapter. After a brief look at what HTML and CSS actually are and why the time is ripe for a new version of the standards you've seen several specific issues which HTML5 and CSS3 solve for web authors:

- HTML5 allows you to describe your content with more appropriate elements
- New form input types mean you'll spend less time writing validation functions
- You no longer need to depend on plugins for playing audio, displaying video or drawing interactive graphics
- APIs allow desktop-like applications which can receive real time updates from the server
- Simple visual effects like rounded corners and drop shadows can be applied with a few lines of CSS3 rather than ugly markup or JavaScript hacks
- Applying alternating styles to consecutive elements or picking out the first and last elements of a set for special styling is easy with CSS3 selectors
- No more need for image based hacks just to get a particular font used on your web page
- Less time wasted writing JavaScript for simple animations in response to user interaction

So it seems like the web is in constant conflict,
everyone wants standards to ensure everyone has

access, but everyone is also trying to get an edge by innovating.

That's right, the web evolves and adapts as people discover more and more uses.

Standardization may be a task of Sisyphean proportions.

The standards establish a baseline for web functionality. In 1997 everyone was happy with text and images, but now that we all expect interactivity and multimedia it's time to standardize those things.

So people will look to compete and innovate in the areas outside the standard? I can't wait to see what the web throws up next!

Character images should face each other with talk in between

2

New markup elements

In this chapter you will learn about the new elements and attributes of HTML5, including:

- Semantic elements for common page elements, and old elements updated
- New global attributes which can be applied to any element
- How all these elements fit together thanks to the HTML5 content model
- Getting all these elements to work in browsers today

2.1 Document structure

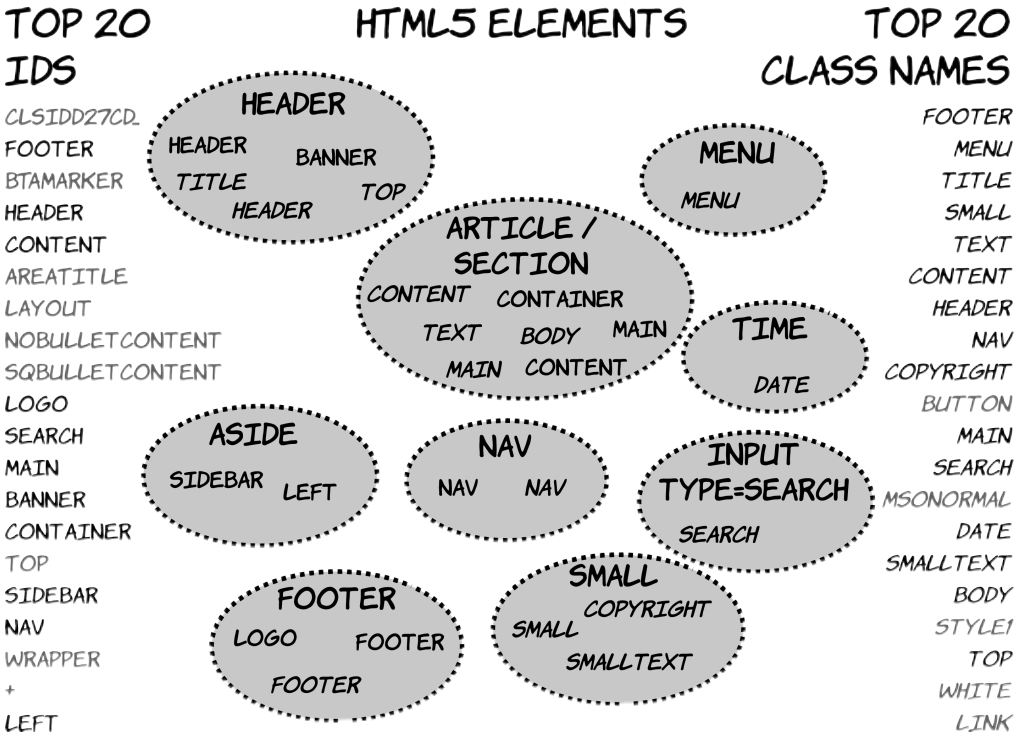
HTML, at heart, is a way of describing hyperlinked documents. Although HTML5 extends the language more formally into a language which can also describe applications, and there's certainly a large gray area between the two, the majority of existing web pages are documents rather than applications. It may come as a surprise to you that a language which has been used so pervasively for document description is lacking markup to describe several

very commonly used document elements, but HTML5 seeks to fill the gaps in the HTML document description vocabulary.

In 2005, several studies were done which attempted to analyze how authors were using markup on the web. Two of these are of particular interest to us:

- In November a study of 1315 websites counted how often different values for the ID attribute were used
- In December 2005 a study of slightly over a billion web pages analyzed, among other things, how often particular class names appeared

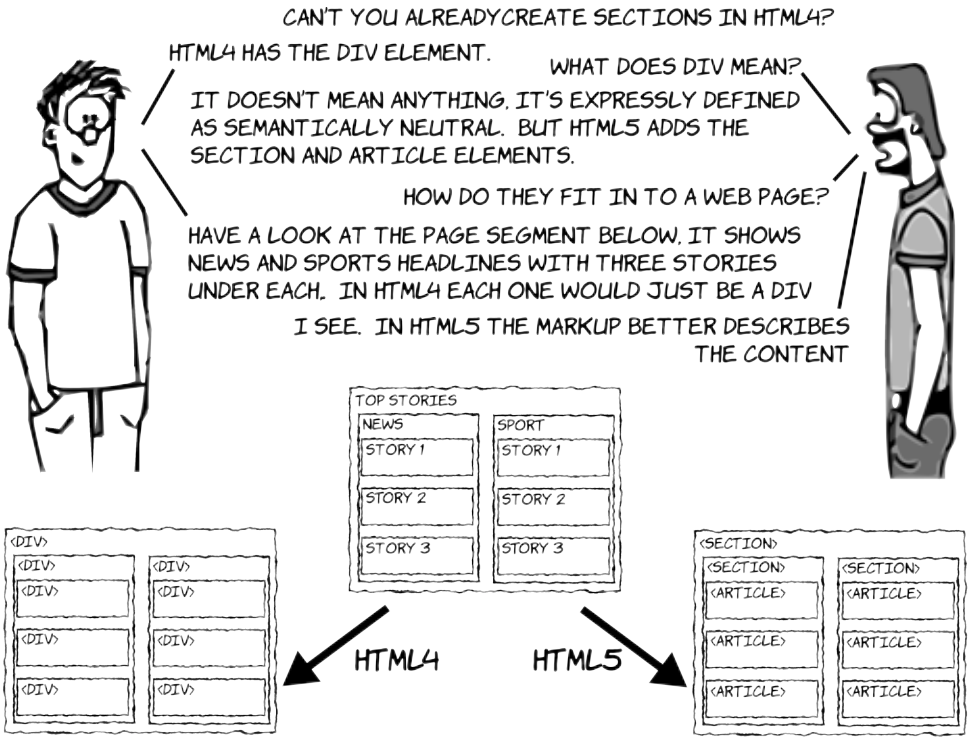
The diagram below shows the top twenty results in each category and the corresponding new elements in HTML5.



Many of the top IDs, things like `btamarker` and `nobulletcontent`, are automatically generated by software such as Microsoft FrontPage and other office products. Their popularity is therefore more of indication of the market penetration of the products than author requirements or intentions.

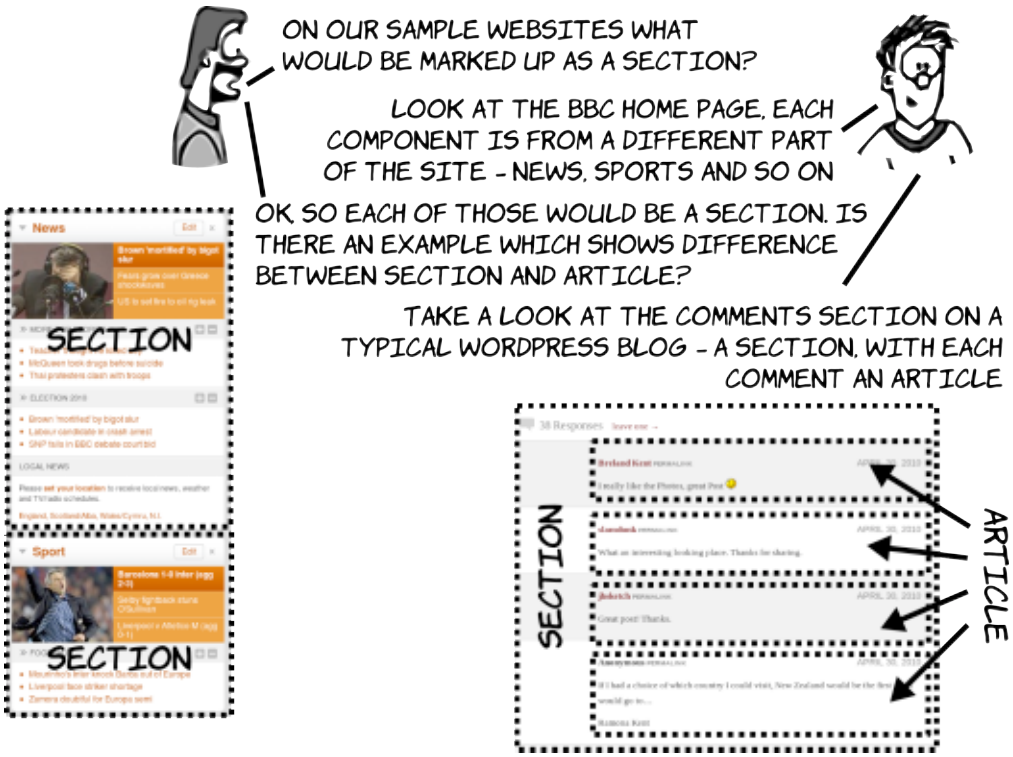
2.1.1 Sectioning Content

It's common for web pages to have many distinct sections. A blog home page will usually have several blog posts, each a section by itself, and each blog post may have a comments section or a 'related posts' section. HTML4 offers only one type of element for this common need, the div. HTML5 adds two new elements: section and article.



The `<section>` and `<article>` elements are conceptually very similar. Articles and sections can be quite interchangeable - articles can exist happily within sections, but articles could quite happily be broken down into sections, and there has been a lot of discussion about whether HTML5 really needs both of them. For now, though, we have both, and the question you are probably asking yourself is how to decide which one to use? The key parts of the spec to focus on when choosing one or the other is:

- An article is intended to be independently distributable or reusable.
- A section is a thematic grouping of content.



2.1.2 Headings, headers and the outlining algorithm

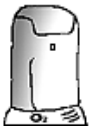
Heading elements provide an implicit structure for documents. A heading indicates the start of a new section and briefly describes the topic of the text that follows.

For headings HTML5 has the six elements from HTML4, `<h1>` through to `<h6>` – an `<h1>` is the most important and `<h6>` the least. In HTML4, although this is not mandated by the specification, it was best practice to have only a single `<h1>` element per document. It was also best practice to travel up and down the headings as you would in a book – so an `<h2>` would be followed by an `<h3>` or another `<h2>`, but not directly by an `<h4>`.

In HTML5 there are also two new elements for dealing with headers - `<header>` and `<hgroup>` - as well as a new way of organizing them. A header element will appear near the top of a document, section or article and will usually contain the main heading plus probably some navigation and search tools. Here's an example from the BBC website:

Here's how that might be marked up in HTML 5:

```
<header>
  <h1>BBC</h1>
  <nav>
    <ul>
      <li><a href="/options">Display Options</a></li>
      <li><a href="/access">Accessibility Help</a></li>
      <li><a href="/mobile">Mobiles</a></li>
    </ul>
  </nav>
  <form target="/search">
    <input name="q" type="search">
    <input type="submit">
  </form>
</header>
```



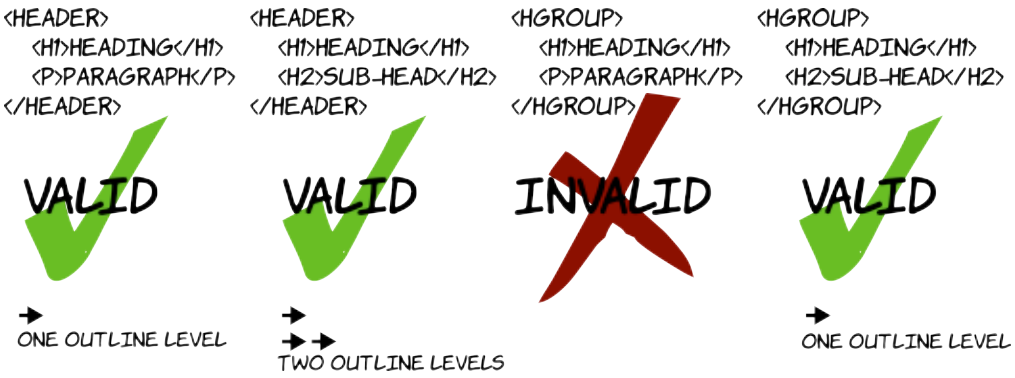
You'll learn more about the `<nav>` element later in this chapter, HTML5's new form elements will be covered in depth in chapter 3

`Hgroup` should be used where you want to have a main heading with one or more sub headings. For an example of this we can look at the HTML5 spec itself:

```
<hgroup>
  <h1>HTML5
    (including next generation
     additions still in development)
   </h1>
  <h2>Draft Standard &mdash;
      12 May 2010</h2>
</hgroup>
```

HTML5 (including next generation additions still in development)
Draft Standard — 12 May 2010

The header element can contain any content, but the `hgroup` element can only contain other headers - that is `h1` to `h6`, plus `hgroup` itself. The following diagram demonstrates the differences.



The outlining algorithm is new to HTML5. It is the outlining algorithm which gives a semantic definition to the implicit outlining generated by heading elements. The idea behind it is that it should automatically generate a table of contents for your entire document based on the section and heading markup you've used.

I've had a look at the outlining algorithm, do I really need to understand that in order to write HTML documents?

You can, if you want to, carry on authoring documents in complete ignorance of the outlining algorithm.

I can hear the reprimand on your voice, so if I'm the sort of conscientious web author who takes document semantics seriously?

The outline algorithm formalizes the well

structured semantic approach you've been using for years, and makes incorporating different sections into your document easier.

OK, but it still looks a little complicated?

It's not as complicated as it looks, a few examples will probably make things clear.

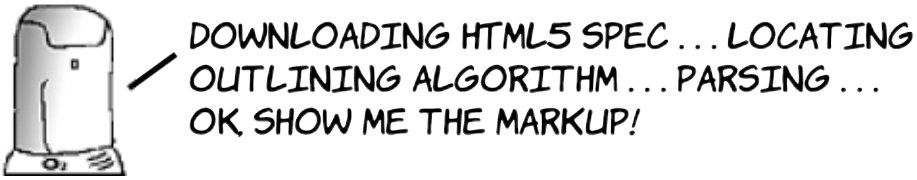
But in order to look at some examples to learn the algorithm, don't we need to implement the algorithm?.

Fortunately, I know someone who can help.

What manner of person is this, some sort of superhuman?

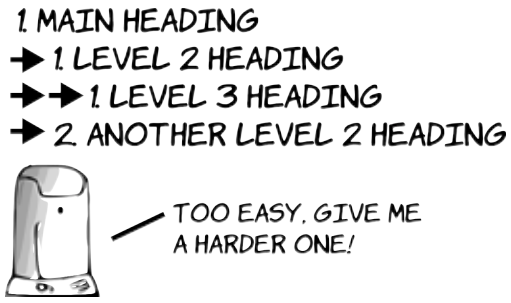
Well, not human, exactly. . .

Character images should face each other with talk in between



So you can get the idea, we'll look at several sample documents and Erwin will generate the document outline according to the HTML5 spec. You will see how the outline is impacted both by headings and heading groups as well as the articles and sections we discussed in the last section.

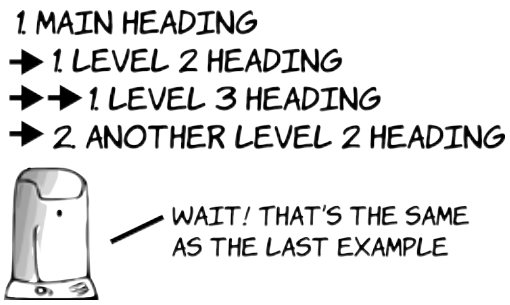
```
<body>
    <h1>Main heading</h1>
    <p>Some text</p>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</body>
```



In a plain document with no other sectioning content, the outline will match the heading levels. This is very similar to the way a table of contents on a Wikipedia is generated (left).

Headings can also be grouped using the `<hgroup>` element, let's see how they affect the document outline.

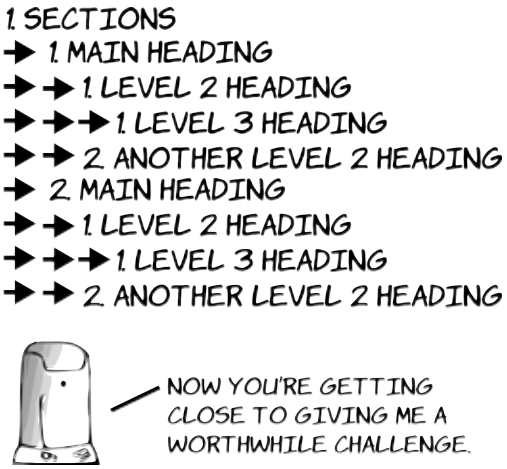
```
<hgroup>
    <h1>Main heading</h1>
    <h2>
      Sub heading to main heading
    </h2>
</hgroup>
<p>Some text</p>
<h2>Level 2 heading</h2>
<p>Some more text</p>
<h3>Level 3 heading</h3>
<p>A bit more text</p>
<hgroup>
    <h2>Another level 2 heading</h2>
    <h3>
      Sub heading to level 2 heading
    </h3>
</hgroup>
<p>The last bit of text</p>
```



The outline will only show the highest level heading from any `<hgroup>`, you can see the headings 'Sub heading to main heading' and 'Sub heading to level 2 heading' do not appear in the outline. The `<hgroup>` element can contain any number of subheadings, but it can only contain other heading elements.

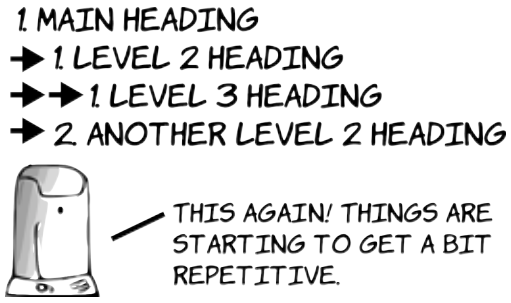
Next, let's have a look at how sections affect the outline.

```
<h1>Sections</h1>
<section>
    <h1>Main heading</h1>
    <p>Some text</p>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</section>
<section>
    <h1>Main heading</h1>
    <p>Some text</p>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</section>
```



As you can see, there are now multiple `<h1>` elements in the document, but they do not all sit at the same level of the document outline. In fact, you can do without any heading element other than `<h1>`. Let's have a look at another example.

```
<h1>Main heading</h1>
<p>Some text</p>
<section>
    <h1>Level 2 heading</h1>
    <p>Some more text</p>
    <article>
        <h1>Level 3 heading</h1>
        <p>A bit more text</p>
    </article>
</section>
<section>
    <h1>Another level 2 heading</h1>
    <p>The last bit of text</p>
</section>
```



We have achieved the exact same outline as our original example but using only level one headings. Earlier we discussed the similarity between `<section>` and `<article>`, if we replace one for the other in listing above we can see how similar they are.

```
<h1>Articles</h1>
<article>
    <h1>Main heading</h1>
    <p>Some text</p>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</article>
<article>
    <h1>Main heading</h1>
    <p>Some text</p>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</article>
```

1. ARTICLES
   - 1. MAIN HEADING
     - 1. LEVEL 2 HEADING
       - 1. LEVEL 3 HEADING
     - 2. ANOTHER LEVEL 2 HEADING
   - 2. MAIN HEADING
     - 1. LEVEL 2 HEADING
       - 1. LEVEL 3 HEADING
     - 2. ANOTHER LEVEL 2 HEADING

THIS IS IDENTICAL TO THE SECTION EXAMPLE. SECTION AND ARTICLE ARE INTERCHANGEABLE FOR OUTLINING

Now let's consider the `<header>` element. It represents the header of a document, section or article, typically containing headings and other metadata about the section. You will frequently have content which you don't want to be part of the heading element itself, but does not fit in with the following content. Examples would be subheadings, author bylines or publishing date information.

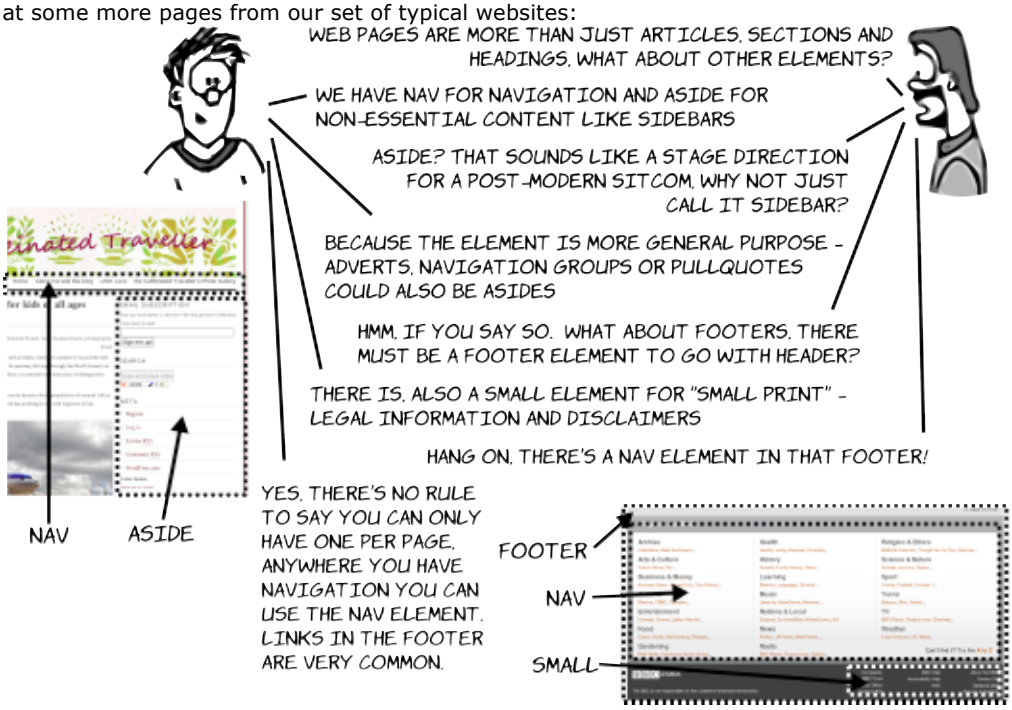
```
<h1>Articles</h1>
<article>
    <header>
        <h1>Main heading</h1>
        <p>Some text</p>
    </header>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</article>
<article>
    <header>
        <h1>Main heading</h1>
        <p>Some text</p>
    </header>
    <h2>Level 2 heading</h2>
    <p>Some more text</p>
    <h3>Level 3 heading</h3>
    <p>A bit more text</p>
    <h2>Another level 2 heading</h2>
    <p>The last bit of text</p>
</article>
```

1. ARTICLES
   - 1. MAIN HEADING
     - 1. LEVEL 2 HEADING
       - 1. LEVEL 3 HEADING
     - 2. ANOTHER LEVEL 2 HEADING
   - 2. MAIN HEADING
     - 1. LEVEL 2 HEADING
       - 1. LEVEL 3 HEADING
     - 2. ANOTHER LEVEL 2 HEADING

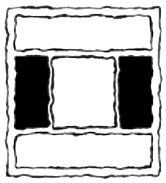
THE <HEADER> ELEMENT DOES NOT HAVE ANY IMPACT ON THE DOCUMENT OUTLINE. IT'S AS IF IT'S NOT THERE.

2.1.3 Common page elements

There are more new elements than `article`, `section`, `header` and `hgroup`. Let's have a look at some more pages from our set of typical websites:



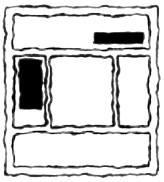
- The `<aside>` element is intended for content which is not part of the flow of text in which it appears, but still related in some way. In many books, including this one, you will see sidebars for things such as terminology definitions or historical background, like the one below – these would be marked up as `<aside>` if the book was HTML5. Sidebars are also quite common in website design, though the meaning is slightly different, often they will contain navigation or related links.



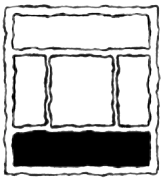
Sidebar

This is an example sidebar, if this were an HTML5 document it should be marked up with the `<aside>` element.

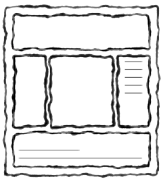
- The `<nav>` element is intended for navigation, both within the page itself, like in the Wikipedia table of contents, and through the rest of the website. You can have any number of `<nav>` elements on a page, on large sites it is common to have global navigation across the top (in the `<header>`) and local navigation in a sidebar (in an `<aside>` element).



- The `<footer>` element would generally appear at the end of a document, section or article. Like the `<header>` element, it's content will generally be meta-information – author details, legal information or links to related information. However, it is valid to include `<section>` elements within a footer, when marking up appendixes for example.



- The `<small>` element will often appear within a footer or aside element - it should contain copyright information, legal disclaimers and other 'small print'. Note that it is not intended to make text smaller. You may choose to style its contents smaller than your main text, but, as with other elements, its role is to describe its contents, not prescribe presentation.



2.1.4 The HTML Doctype

The DOCTYPE declaration is somewhat archaic; it comes from SGML, which was used to define previous versions of HTML in terms of the language syntax. It appears as the first thing in an HTML document and serves two practical functions:

- It is used by HTML validation services to determine which version the document uses
- Browsers use the DOCTYPE to determine whether to use standards, 'almost standards' or quirks mode to render the document (see Appendix C for a discussion of these issues – the short version is: standards mode is what you want)

HTML5 is defined in terms of its DOM representation after parsing, so it doesn't need a DOCTYPE for validation, but we still want legacy browsers to render pages in standards compliant mode. With this in mind the authors of the HTML5 spec worked out the minimal amount of markup required to trigger standards mode in browsers:

```
<!DOCTYPE html>
```

Compare this with similar declarations for HTML4 and XHTML1

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
                "http://www.w3.org/TR/html4/strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

You can see that the HTML5 doctype is much shorter, easier to type and easier to remember.

2.2 Other new elements

There are several other new or redefined elements in HTML5 and in this section you will learn about some of these. HTML5 includes dedicated elements for dates as well as figures and captions, it also rehabilitates the b and i elements which were deprecated in HTML4. Let's take a closer look.

2.2.1 Time

Web pages frequently contain information about upcoming and past events. Below are some examples from the three websites we've been looking at.

This Wordpress blog is advertising an upcoming event. You can see the key components are all present here:

- An event title
- A time
- A place

The BBC website has a page for each for each program, and this contains information about when it will next be broadcast. Although the title isn't shown here, you can see the key components are still here: a time and a place (though in this case the 'place' is a bit more abstract).

Finally, Wikipedia has events on many of pages – in this example the 'event' is the death of Edward I. You may not have considered this as the same sort of thing as the previous two examples, but you can see it shares the same basic characteristics.

This pattern is so common that the Microformats movement established a standard way of marking it up called *hCalendar*.

Microformats

Microformats are an effort to extend the expressive power of standard HTML by using certain attributes, mostly the class attribute, in a standardized way. Popular Microformats include *hCard*, for describing contact information, *adr* for addresses and *hCalendar*, for describing events. Similar technologies include the more formal RDFa and HTML5's own Microdata (see 2.3.2).

One of the goals of Microformats is to render common information like events easily parseable by computers. This enables a number of useful applications: search engines which can tell you about nearby upcoming events and browsers which can automatically add the events to your calendar.

Addresses, being naturally plain text information with some internal structure (house number, street name, city etc.) are relatively easy to deal with, as long as there is some way to demarcate the components. Microformats manages this by adding a class of location to the containing element, or alternatively using another Microformat, adr, to describe the address in detail.

Dates and times are altogether more complicated. Take the simple example "1/6/2011" - if you're reading this in the US you probably interpret that as 6th January, whereas in the UK the date would be 1st June. Or have another look at the BBC example above, in that one the date is "today" - I took that screenshot some time ago, how useful is 'today' now? You may think this is no more or less ambiguous than the addresses, but the frustrating thing is that

we know there's an absolute date and time which underlies the more ambiguous human expression of it we see more commonly.

Computers would like dates and times in an unambiguous format, people often find the the unambiguous format hard to digest but can easily understand ambiguous dates from the context. To serve both, our web pages need to provide dates in two formats.

HTML5 solves this by providing a `<time>` element, let's have a look at an example:

```
<time datetime="2011-06-01">1/6/2011</time>
```

We can be more specific:

```
<time datetime="2011-06-01T18:00:00+01:00">6 o'clock on 1/6/2011</time>
```

Humans get a readable time which they can disambiguate through the context in the normal way, computers can read the ISO6801 date and see the date, time and timezone.

2.2.2 Images and diagrams with figure and figcaption

Putting an image in a web page is easy enough, the <img> element has existed since the early days of the web. It was somewhat controversial at the time, several alternatives were put forward, but the most popular browser (Mosaic) implemented it so it became a de facto standard. The ability to add images was one of the main things which catapulted the web from being an academic document sharing network into a worldwide phenomenon in the mid nineties, but since that early take up not much has changed.

I remember back in early '93 when Marc

Andreessen invented the img tag, it seems to have been working fine for 17 years.

You're right that we're unlikely to see a replacement at this stage, but the img tag is limited from a semantic standpoint – there is no visible way to associate explanatory text with the image.

Surely it's quite straightforward to add a caption or description in your HTML4 markup?

In HTML4 the only way to mark this up is to wrap the picture and its caption in another element – you might be able to use a definition list, but most likely it would be a or a <table> – in other words a semantically neutral wrapper.

Can't web authors use the alt and longdesc attributes?

They can, but because neither are visible by default both have tended to be ignored or misused in the real world.

So there's a new solution in HTML5?

Yes, HTML5 retains the <img> element but adds two new elements, <figure> for wrapping the whole thing and <figcaption> for describing the image. I'll show you how they work.

Character images should face each other with talk in between

This is what the markup for the above screenshot looks like:

```
<figure>
    <img src="scenery.jpg" alt="Picture of the Irish south coast">
    <figcaption>Looking out into the Atlantic Ocean
     from south west Ireland</figcaption>
</figure>
```

Note that `<figure>` doesn't have to contain an `<img>` element, it might instead contain an SVG drawing or a `<canvas>` element, or even ASCII art in a `pre` element. Whatever type of graphic is contained, the `<figure>` element links the graphic to the caption.

2.2.3 Emphasizing words and phrases

The `<b>` and `<i>` elements have a long history in HTML, they were listed, along with the and elements, in the character highlighting section of the 1993 IETF draft proposal for Hypertext Markup Language. The `<b>` and `<i>` elements are listed in the sub-section 'Physical Styles' (along with `<tt>`) - that is their purpose was entirely presentational. Meanwhile `<em>` and `<strong>` (along with several others) are in the sub-section 'Logical Styles' – elements with semantic meaning. This early distinction highlights the problem `<b>` and `<i>` would later run into.

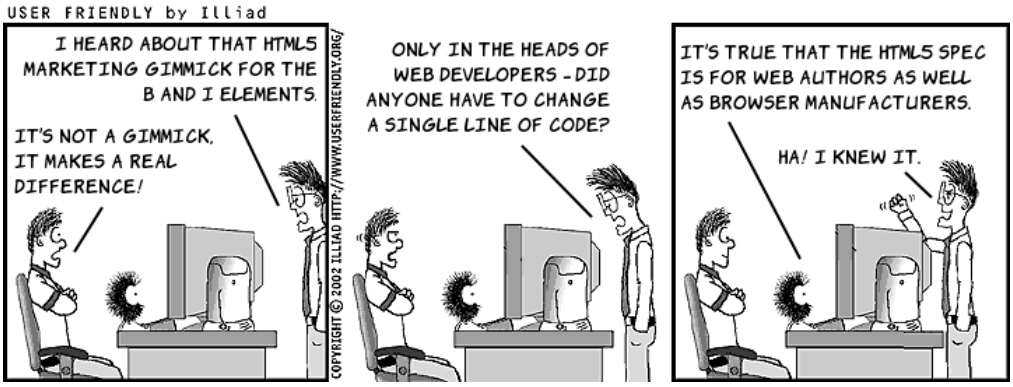
We saw, right at the start of this chapter, that separation of concerns is the holy grail of web authoring – HTML for content, CSS for presentation and JavaScript for behavior. Since `<b>` and `<i>` are entirely presentational their use has long been frowned upon, with several serious proposals to remove them from HTML altogether. Meanwhile `<strong>` and `<em>` have always had a semantic definition while appearing identical to `<b>` and `<i>`, respectively, in most browsers.

Ever pragmatic, the HTML5 spec recognizes that, with millions of pages of legacy content out there, browsers are not going to be dropping support for <b> and <i> any time soon. On the other hand, blindly using instead of <i> and instead of <b>, or using a element just to apply a bold or italic style to a word is also not good practice semantically.

So, instead of removing either `<b>` or `<i>`, HTML5 redefines and rehabilitates them:

Element	HTML4 Definition	HTML5 Definition (taken from the spec on 12/05/2010)
<i>	Renders as italic text style.	"The i element represents a span of text in an alternate voice or mood, or otherwise offset from the normal prose, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, a thought, a ship name, or some other prose whose typical typographic presentation is italicized."
<b>	Renders as bold text style.	"The b element represents a span of text to be stylistically offset from the normal prose without conveying any extra importance, such as key words in a document abstract, product names in a review, or other spans of text whose typical typographic presentation is boldened."

As you can see, the HTML4 definition is entirely presentational, whereas the HTML5 definition goes to great lengths to give a semantic meaning while remaining compatible with the purely presentational uses of the two elements for backwards compatibility.



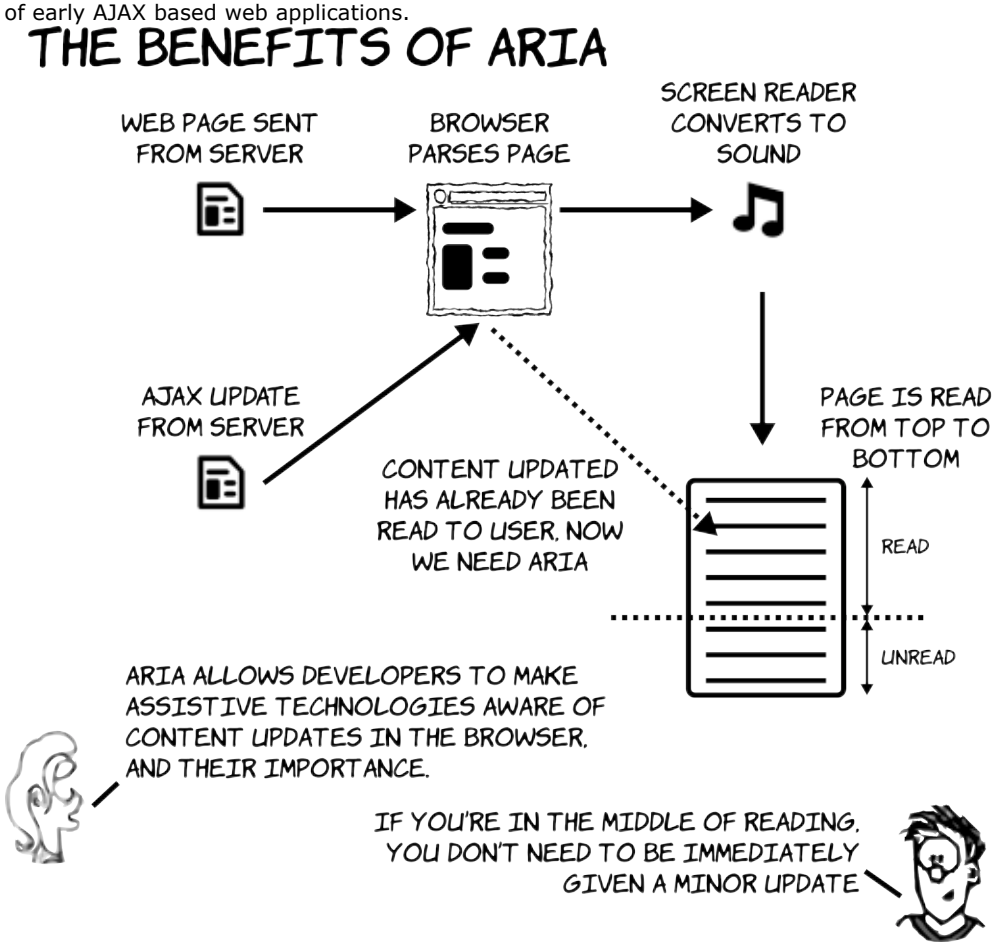
2.3 HTML5's new global attributes

HTML5 introduces several new global attributes that can be applied to all elements. Some of these are brand new, some formalize existing conventions or non-standard extensions, and

some have been taken from other W3C standards. In the latter category, HTML5 extends HTML4 with attributes taken from the Accessibility for Rich Internet Applications (ARIA) specification. Existing conventions are formalized by the set of data-* attributes for storing custom data on elements. Three new global attributes have been added for Microdata which provides a route for advanced semantics to be added to your markup.

2.3.1 Accessibility with ARIA

ARIA is a standard developed at the W3C in response to the the generally poor accessibility of early AJAX based web applications.



Notifying users of AJAX updates is not the only benefit ARIA can give us. ARIA consists of a set of attributes and values which can describe to assistive technology the role of various page elements and their status. In other words, they add semantic value to HTML elements

so that you can say this element is a header, this element is navigation, this element is a toolbar and so on. Let's have a look at an example:

```
<body role="document">
  <div role="note" aria-live="polite"
   aria-relevant="additions removals">
    An update added by JavaScript
  </div>
  <div role="banner">
    <h1 role="heading" aria-level="1">The heading</h1>
  </div>
  <div role="navigation">
    <a role="link" href="/home">Home Page</a>
    <a role="link" href="/inbox">Inbox</a>
  </div>
  <div role="main">
    A very interesting article goes here.
  </div>
  <div role="footer">
    All rights reserved.
  </div>
</body>
```

Wait a minute, this all sounds a bit familiar, aren't these all the same things which HTML5 is doing?

Yes, what HTML5 aims to accomplish through additions such as the <header> and <nav> elements is very similar to what ARIA is trying to accomplish in providing better semantics to assistive technology.

Why bother with ARIA at all then?

There was some debate about that, but ARIA has a wider and more far reaching vocabulary than HTML5 for describing the components of web applications and already has wide support among vendors of browsers, operating systems and assistive technology.

But, you have to admit, some of the ARIA stuff seems redundant in HTML5 – look at the nav element and the navigation role.

Yes, the HTML5 spec has a long list of elements to which user agents should automatically assign particular ARIA roles. These elements are said to have 'strong native semantics'.

So, if you use HTML5 correctly you will get a certain amount of accessibility for free compared to what HTML4 offered?

Once the browsers and assistive technologies implement support, yes.

And you can take advantage of the extra expressive ability of ARIA when you need to. Is there a risk the ARIA role will be in conflict with the HTML5 semantics?

There is, but the HTML5 spec explicitly lists the allowed ARIA roles for those elements where this might happen – the 'implied native semantics'. Validation tools can then flag up inappropriate combinations.

Character images should face each other with talk in between

By using HTML5, we can cut down on the amount of markup required to provide an accessible user experience. This listing updates the previous one but takes advantage of the strong and implied native semantics in place of several of the ARIA attributes:

```
<body>
  <aside aria-live="polite" aria-relevant="additions removals">
    An update added by JavaScript
  </aside>
  <header role="banner">
    <h1>The heading</h1>
  </header>
```

```
  <nav>
    <a href="/home">Home Page</a>
    <a href="/inbox">Inbox</a>
  </nav>
  <article role="main">
    A very interesting article goes here.
  </article>
  <footer>
    All rights reserved.
  </footer>
</body>
```

Although you do not have to use the implied ARIA roles on those elements with strong semantics, such as link and nav, at present no assistive technologies recognize the HTML5 elements, so you should specify both for backwards compatibility.

2.3.2 Extending HTML with custom attributes and microdata

Custom data attributes allow authors to add arbitrary data to elements for their own private use. The idea behind them is that there is data is not directly relevant to the user but which does have some meaning to the JavaScript on the page that cannot be expressed in HTML semantics. It is a standardization of an approach taken by several JavaScript 'widget libraries', for example Dijit, the Dojo toolkit. These libraries, like HTML5, set out to enhance and extend the application abilities of HTML4 – adding things such as combo boxes and date pickers, which HTML5 also provides, but also more complex UI elements such as tree views, dijit widget and dijit widget.

Using one of these libraries, you declare an element to be a tab control like this:

```
<div dojoType="dijit.layout.TabContainer">
     <div dojoType="dijit.layout.ContentPane" title="My first tab">
         Lorem ipsum and all around...
     </div>
     <div dojoType="dijit.layout.ContentPane" title="My second tab">
         Lorem ipsum and all around - second...
     </div>
     <div dojoType="dijit.layout.ContentPane" title="My last tab">
         Lorem ipsum and all around - last...
     </div>
</div>
```

A browser, as with HTML elements, will parse the attribute, even though it doesn't recognize it, and add it to the DOM. The Dijit library will run when the page has loaded, search for these attributes, and run the appropriate JavaScript to enable the advanced control.

Seems like everyone has been getting along just fine with their own attributes, why add this to HTML5?

Well, for one thing, it'll stop your markup validating.

Surely we're past the stage of obsessively validating all our HTML in order to earn a badge?

Validation is a tool. Failing validation may not bother you too much, but if you're looking for that one unintended mistake having to sift through many intended ones should be unnecessary.

OK, I can understand that, but is it the only reason?

There's a risk that the attribute names chosen by the widget libraries will get used in future versions of HTML.

Is that really a big risk?

It may seem unlikely, but remember the WHATWG is trying to add common use cases into standard HTML5.

Who's WHATWG? I thought it was the W3C for web standards?

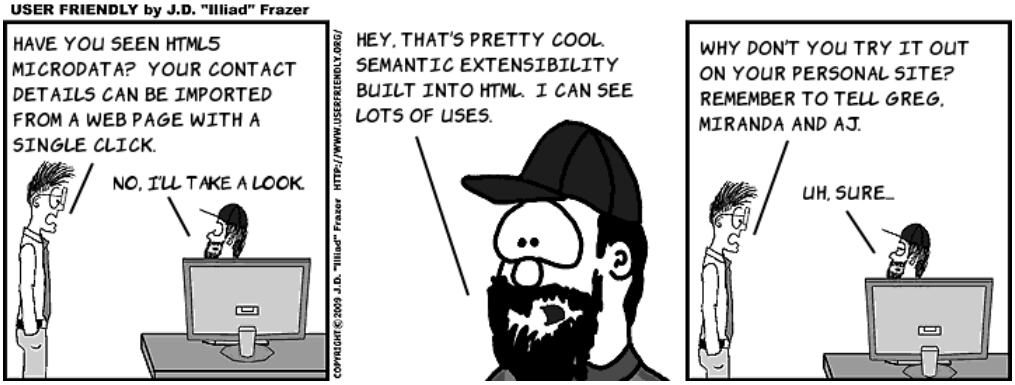
The HTML5 spec has an interesting history, check out Appendix A for the full background.

Character images should face each other with talk in between

The HTML5 solution to both the validation and potential name clash issue is the data-* attribute collection. The * is a wildcard, that is, it can be whatever you want it to be. But anything starting with data- will be allowed through the validator, and you are guaranteed that no data-* attributes will be made part of HTML.

The data-* attributes allow you to add information to your page for your own personal use, if your goal is to share information with other websites then you should instead use Microdata.

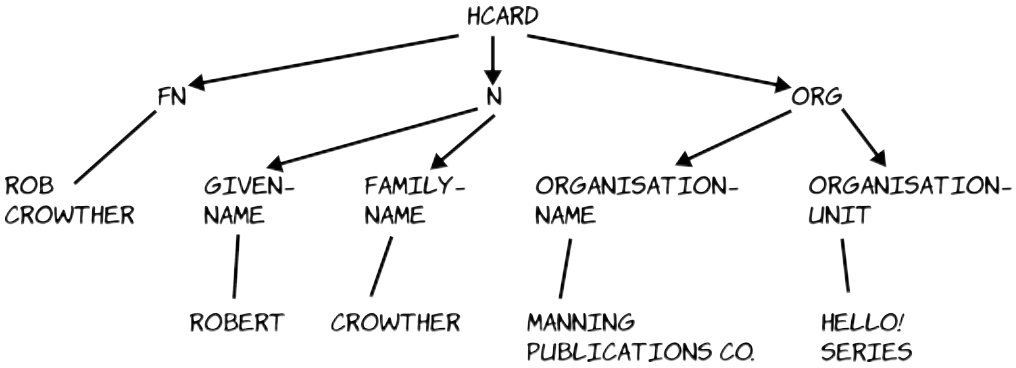
Microdata extends the expressive power of HTML to cover things which are not strictly markup, you can use Microdata to designate a portion of your page as describing contact information, a calendar event or licensing information.



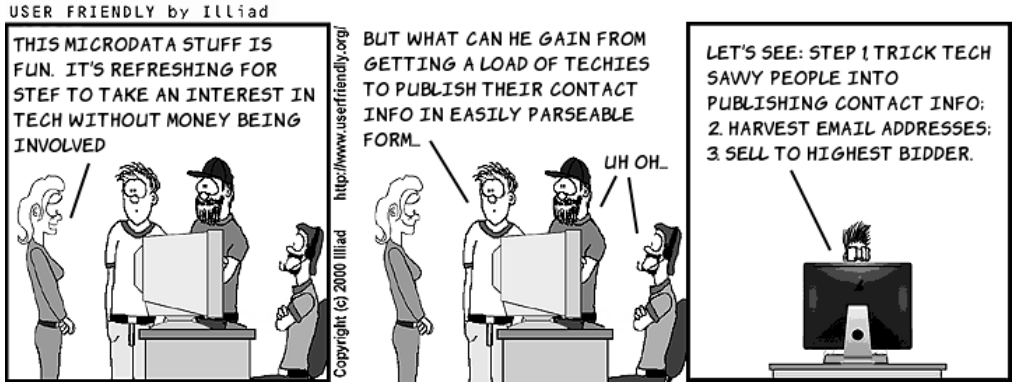
Microdata uses three global attributes: item, itemtype and itemprop. All three can be seen in action in this short example which describes contact information:

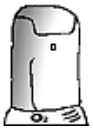
```
<section id="rob" itemscope
itemtype="http://microformats.org/profile/hcard">
 <h1 itemprop="fn">Rob Crowther</h1>
 <div itemprop="n">
  <meta itemprop="given-name" content="Robert">
  <meta itemprop="family-name" content="Crowther">
 </div>
 <p itemprop="org" itemscope>
  <span itemprop="organization-name">Manning Publications Co.</span>
  (<span itemprop="organization-unit">Hello! Series</span>)
 </p>
</section>
```

This code, because of the itemtype attribute on the parent element referencing the hcard vocabulary, describes a person – me! The itemprop attributes are extracted as a set of name-value pairs into a tree like data structure, following the markup, like this:



This information could then be recovered from the page in a usable format by a web browser or a search engine. Of course, you may not want the information to be more easily usable by computers, normal rules of internet publishing apply.





You will learn more about Microdata when we look at creating a contact manager application in chapter 12.

2.4 The HTML5 Content Model

In HTML5 elements are split into several categories. One element can be the member of several categories; it can also be a member of a category only in particular circumstances, such as when an attribute is given a certain value.

Element	Metadata Content	Flow Content	Phrasing Content	Interactive Content	Embedded Content	Heading Content	Sectioning Content
a, button, input, keygen, label,select, textarea		✓	✓	✓			
abbr, area, b, bdo, br, cite, code, datalist, del, dfn, em, i, ins, kbd, map, mark, meter, output, progress, q, ruby, samp, small, span, strong, sub, sup, time, var, wbr			✓	✓			
address, blockquote, div, dl, fieldset, figure, footer, form, header, hr, ol, p, pre, table, ul, Text		✓					
article, aside, nav, section		✓					✓
audio, embed, iframe, img*, object, video		✓	✓	✓	✓		
base, title	✓						
canvas, math, svg		✓	✓		✓		
command, link, meta, noscript, script	✓	✓	✓				
details, menu		✓		✓			
h1,h2,h3,h4,h5,h6, hgroup		✓				✓	
style	✓	✓					

Headings in the left seven columns of the above table should be rotated to vertical

The content model is important because it is the main way of determining if it is valid to use a certain element at a particular place within your document. For instance, the reason we knew that an <hgroup> element can contain only other headings is that <hgroup> is a member of heading content and heading content is only allowed to have other heading content as a direct ancestor.

Flow content	Most elements are categorized as flow content, it is the default content type for elements visible on the page.
Sectioning content	Sectioning content defines the scope of headers and footers and feeds into the document outline
Heading content	Heading content, as you might expect, is just for headings and hgroup.
Phrasing content	Phrasing content is mostly used to describe the text of a document. In most cases, phrasing content can only contain other phrasing content.
Embedded content	Embedded content is used to put an external resource into the web page, for example an image or video.
Interactive content	Interactive content is elements that are specifically intended for user interaction, mostly form controls. Note that other elements can be made responsive to user input through the use of JavaScript, but elements categorized as interactive content have default functionality in the browser.
Metadata content	Metadata content is content that sets up the presentation or behavior of the rest of the content, or that sets up the relationship of the document with other documents, or that conveys other "out of band" information

In the above table, the headings run down the left side, line should be to the right rather than under the headings

2.5 Browser support

So, do the new elements we've discussed in this chapter work in today's browsers? The short answer is yes (with a couple of exceptions), the long answer is a little more complex. Let's ask ourselves a question, what does it mean to say that a browser supports the <p> element?

For a text element like <p>, which is not required to do much except appear on the page, there are two principal requirements:

- It shows up in the DOM with a standard set of element properties
- It shows up on the user's browser with some sort of default presentation

It turns out that the first requirement is easy to satisfy – as long as you follow simple tag naming rules you can put any tags at all in your HTML and all browsers will put them in the DOM with a default set of properties.

Where things can fall down is that second requirement, having a default presentation. Browsers have only recently started providing any default presentation for the new elements in HTML5, for instance Firefox 3.6 doesn't but Firefox 4.0 does. But this is not really much of a problem. As you know, we web authors define our content in HTML and our presentation in CSS – and browsers work exactly the same way. The default presentation for the 'supported' elements is defined in CSS. If you use Firefox you can even find this file on your hard drive – it'll be called html.css.

So using these new elements is simply a matter of us taking on the responsibility of providing some default CSS rules for them. As, in most cases, we would want to write CSS for these elements anyway this doesn't seem like too much effort. Let's see how it works with an example.

Here's the contents of a simple HTML5 document for us to experiment with:

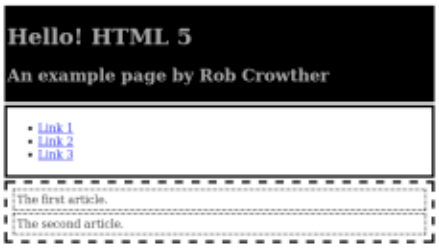
```
<header>
  <hgroup>
    <h1>Hello! HTML 5</h1>
    <h2>An example page by Rob Crowther</h2>
  </hgroup>
</header>
<nav>
  <ul>
    <li><a href="#">Link 1</a></li>
    <li><a href="#">Link 2</a></li>
    <li><a href="#">Link 3</a></li>
  </ul>
</nav>
<section>
  <article>The first article.</article>
  <article>The second article.</article>
</section>
```

Starting with the basic styles below, this screenshot shows what the page looks like in a browser which does not have any default HTML5 styles.

```
header, nav, section, article
 {padding: 4px; margin: 4px;}
header
 { background: #000; color: #999; }
nav
 { border: 4px solid #000; }
```

```
section
 { border: 4px dashed #000; }
article
 { border: 2px dotted #000; }
```

By making a single change to that CSS, we can make the page work in most older browsers. See if you can spot it:



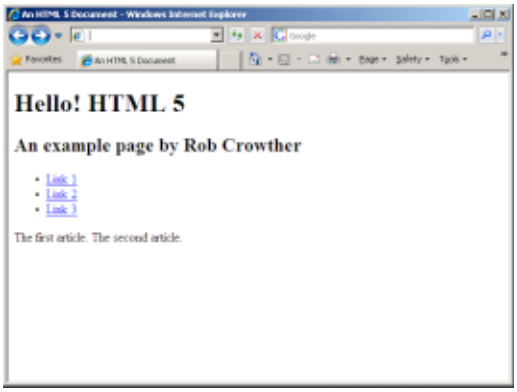

```
header, nav, section, article
 { padding: 4px; margin: 4px;
   display: block; }
header
 { background: #000; color: #999; }
nav
 { border: 4px solid #000; }
section
 { border: 4px dashed #000; }
article
 { border: 2px dotted #000; }
```

By specifying that the block level HTML5 elements header, nav, section and article should be display: block everything works as we want.

Most of the major browsers work identically in this regard. Unfortunately, there are two exceptions to this, one minor and one major. The minor one is Firefox 2.0, which, as Firefox users tend to upgrade regularly, is now used by a very small number of people so we won't worry about it for now. The larger problem is Internet Explorer, which is still the most commonly used browser on the web.

2.5.1 Supporting Internet Explorer

Internet Explorer will not apply CSS rules to any elements it doesn't recognize. Here is what our sample page looks like in IE7:



But all is not lost, you can trick it into recognizing elements with a bit of JavaScript. This code will persuade IE that the <section> elements really does exist and should have styles applied to it:

```
document.createElement("section");
```

Here's the final listing, with each element we want to use enabled in IE:

```
<script>
  document.createElement("header");
  document.createElement("nav");
  document.createElement("article");
  document.createElement("section");
</script>

<style>
  header, nav, section, article {
   padding: 4px; margin: 4px; display: block; }
```

```
    header { background: #000; color: #999; }
    nav { border: 4px solid #000; }
    section { border: 4px dashed #000; }
    article { border: 2px dotted #000; }
  </style>
```

2.5.2 Enable HTML5 support in Internet Explorer with html5.js

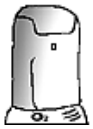
Rather than work out for yourself what elements you need to fix in Internet Explorer, you can use one of the freely available compatibility scripts. A simple one with a good following is html5.js, available here:

http://code.google.com/p/html5shiv/

Of course, the main drawback of these approaches is that they won't work if JavaScript is disabled in the browser, or if something blocks your JavaScript from being downloaded such as a corporate content filter or a personal firewall. Although this is likely to be a small percentage of users for most sites, you should do some analysis of your existing site visitors before embarking on an HTML5 re-design.

2.6 Summary

In this chapter you've learned about the new markup elements in HTML5 and the formal structure provided for them, and the elements inherited from HTML4, provided by the outlining algorithm and the content model. You've looked at several popular websites and seen how the content they're displaying fits naturally into the new semantic elements of HTML5, reducing the need for content authors to add semantic meaning to neutral div and span tags through the id and class attributes. You've also seen how the new global attributes in HTML5 allow you to extend the expressive power and accessibility of HTML documents.



Now that you've learned how HTML5 improves matters for those writing traditional HTML documents, it's time to move on to the main focus of HTML5: markup for applications. We'll start in the next chapter with a look at the enhanced support for forms.

3

HTML5 Forms

You'll start with a quick review of the limited options offered in HTML4, then look at the new form controls HTML5 adds. We'll investigate the built in validation features offered in HTML5 then look at some more new field types which take advantage of that functionality. After that we shall look at some other new features in HTML5 forms such as placeholder text and autofocus, before taking a look at the current state of browser support and learning how you can take advantage of these new features without leaving older browsers behind. [Forms are the basis of all web applications and now you've learned about the many ways HTML5 improves the lives of web authors and users when using forms on web pages. After a brief review of the paltry set of input types available in HTML4 and seeing that they're inadequate for even simple HTML forms, you've seen the range of new form input types and built in validation HTML5 provides.]

Frame1: uf012316 frame1; Frame2: uf012316 frame2; Frame3: uf012316 frame1. All dialogue changed.

3.1 The Limitations of HTML4 Forms

As you saw in chapter 1, HTML4 has a rather paltry selection of input types: basically, three ways of entering text and three ways of selecting from a predefined list of options. Let's review what's available in HTML4 before learning about the new features available in HTML5.

The text input is the workhorse of HTML4 Forms:


```
<input type="text" value="abc">
```

Most times when you can't predict what the user will want to enter but know it will be fairly short, you have to use an input of type text. This would include usernames, dates and times, search terms, email addresses, URLs, telephone numbers, currency, credit card numbers and any simple numeric values.

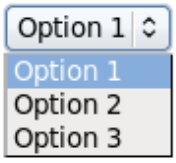
The password field is similar to the text input but without the characters entered being visible on screen.


```
<input type="password" value="abc">
```

A textarea is for larger amounts of free text, when you may be expecting paragraphs rather than a few words.

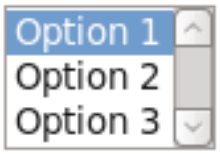

```
<textarea>abc</textarea>
```

If you already know there is a limited number of possible values the user could choose from then you can use a select element.




```
<select>
    <option selected>Option 1
    </option>
    <option>Option 2</option>
    <option>Option 3</option>
</select>
```

The select element allows the user to select from a number of predefined options you provide. It's normally a drop down list as shown here, but you can also use the size attribute so that more than one option shows:

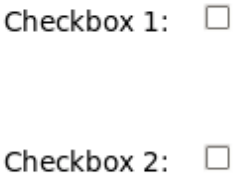



```
<select size="3">
    <option selected>Option 1
    </option>
    <option>Option 2</option>
    <option>Option 3</option>
</select>
```

An alternative to the select element are radio buttons. These are another type of input element but, in normal circumstances, there will be more than one of them in a set. What links them together is that they all have the same value for their name attribute.


```
<label for="exradio1">Radio 1:
</label>
<input type="radio" id="exradio1"
 name="exradio">
<label for="exradio2">Radio 2:
</label>
<input type="radio" id="exradio2"
 name="exradio">
<label for="exradio3">Radio 3:
</label>
<input type="radio" id="exradio3"
 name="exradio" selected>
```

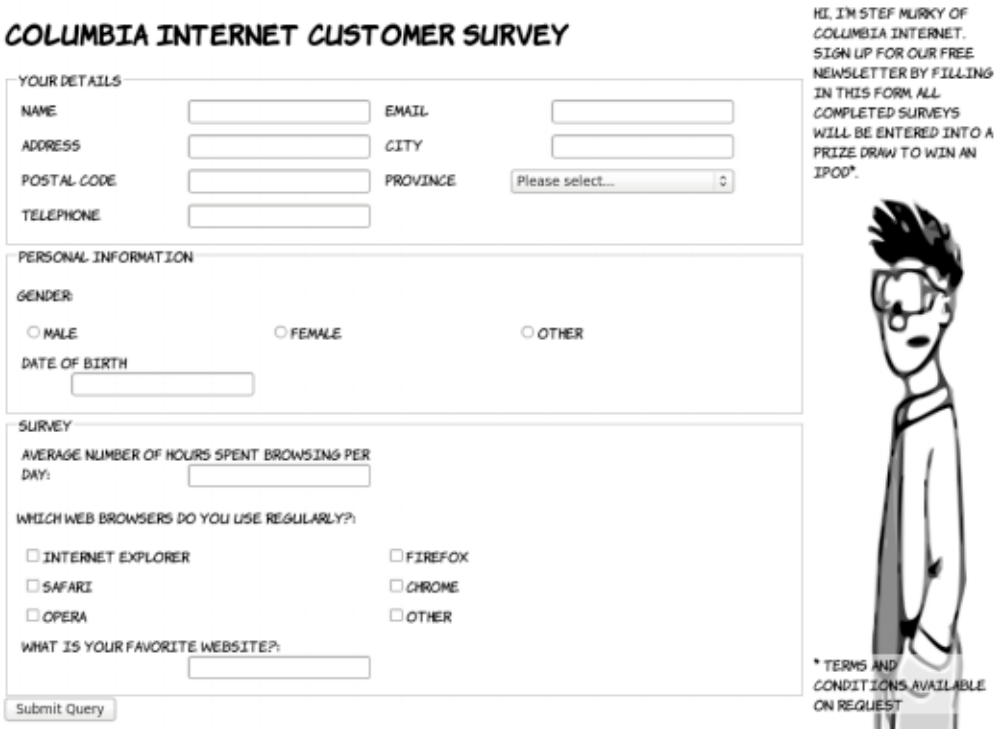
Within a set of radio buttons only one can be selected at a time. If you want multiple items to be selected instead you can either use the select element or a set of checkboxes.




```
<label for="excheckbox1">
  Checkbox 1:
</label>
<input type="checkbox"
  name="checkbox1" id="excheckbox1">
<label for="excheckbox2">
  Checkbox 2:
</label>
<input type="checkbox"
```

```
name="checkbox1" id="excheckbox2">
```

This set of controls has existed mostly unchanged since before forms were first added to the standard in 1996. We can build Stef's sign-up form out of these rudimentary controls:



Above you can see Stef's form implemented using HTML4 form controls. The full source code is available for download from the book's website. If you're in the US and wondering what a Postal Code is, remember Columbia Internet is a Canadian ISP.

A lot of the ideas for HTML5 forms were taken from the XForms2 proposal, a partner standard in what was to be the XML based future of the web with XHTML2 (see appendix A for more details).

3.2 Numbers, ranges, dates and times

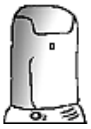
HTML5 introduces several new form controls that didn't exist in HTML4; they allow you to have more precise control over how you gather user input. In HTML4 all text inputs were just that, text. HTML5 significantly expands the range of controls available, not least by providing two ways of entering numbers and multiple controls for dates and times.

Form Submission

For a form to be practically useful in this scenario there would need to be some server side processing to deal with the form values the browser sends when the user hits submit. We don't want to get bogged down in backend issues in this book, so we assume one of the techies – Mike, Miranda or Pitr, will take care of that for us. If you want to test your own forms to see what values they are sending to the server then you can create a simple PHP file:

```
<?php
foreach ($_POST as $field_name => $field_value) {
    print "Field $field_name : $field_value <br />\n";
}
?>
```

A pre-built *collector.php* is available for download from the book's website.



Most of the screenshots in the following sections were taken in Opera 10.60, as that was the only browser at the time of writing to have added full support for the new controls. Exceptions will be mentioned in the text.

The basic number input provides a 'spinbox' as seen here.


```
<input type="number" value="4">
```

Normally the arrows will increment or decrement by 1, but you can adjust this by using the step attribute. This example will increment of decrement in steps of 2:

```
<input type="number" value="4"
  step="2">
```

If an exact number is not necessary then you can use a range control. In the browser this renders as a slider.

```
<input type="range" min="1"
  max="10" value="2">
```

As you can see, the exact value of the range input is not clearly visible. In practice you might use it for large numbers where accuracy is not important. Like the number control, you can specify a step value:

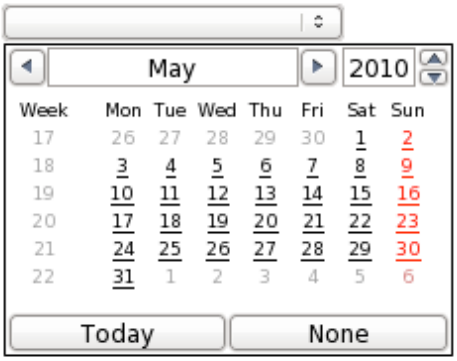
```
<input type="range" min="0" max="1000" value="20" step="20">
```

If you want to use `range` for a numeric input then the best approach would be to either label the high and low values in your HTML or to provide some other user feedback when it's adjusted, perhaps with an output element (see section 3.x.y), in the meantime let's move on to dates and times.

Create a simple date input like this:

```
<input type="date">
```

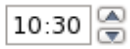
In its unexpanded state, as in the top picture, it looks similar to a select control. However if you activate the control you will see a date picker popup.



What the date picker looks like is left up to the browser, and currently there is no facility to style it through CSS.

The value returned from the date control, and any default value you want to set, will be in the format yyyy-mm-dd. Using this standard ordering saves any confusion between date formats in different countries.

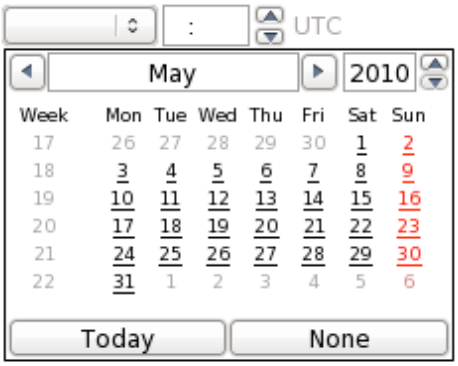
Next, the time input.




```
<input type="time" value="10:30">
```

Again, styling is determined by the browser.

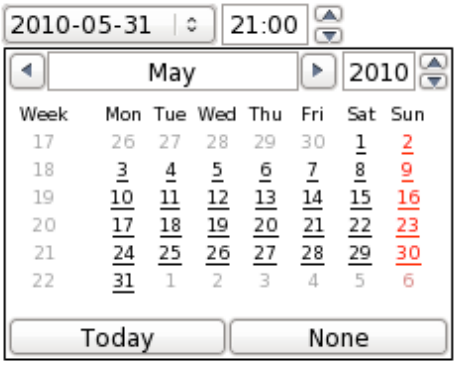
Time and date can be combined into a single input, the datetime type.




```
<input type="datetime">
```

As you can see from the picture, the datetime input is expecting a UTC (Coordinated Universal Time) value.

If you want the user's local time then you should use datetime-local.



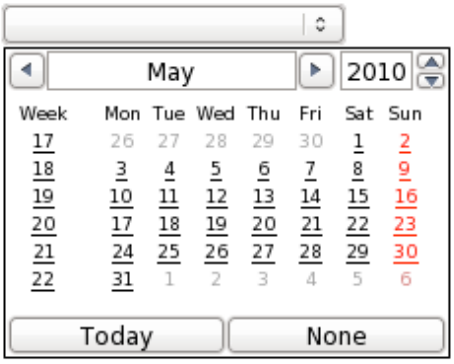

```
<input type="datetime-local"
 value="2010-05-31T21:00">
```

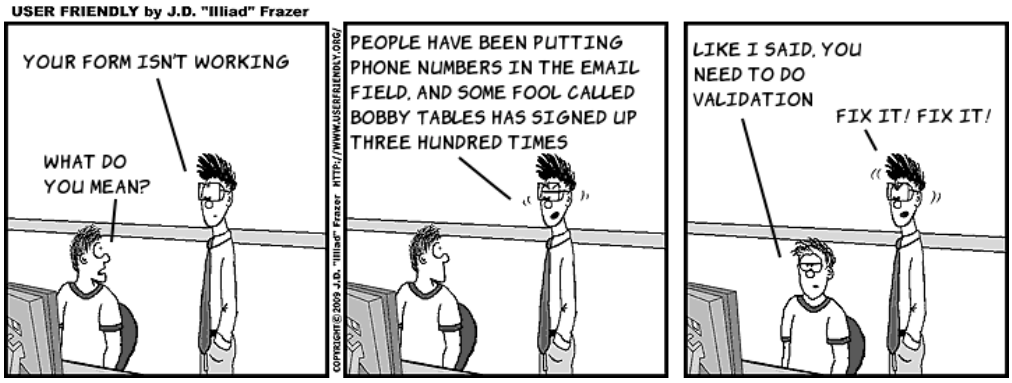
This looks the same as the datetime control but without the UTC annotation. In this example you can see how to specify a default value for the datetime and datetime-local input types: yyyy-mm-ddThh:mm

As well full dates you can also have months or weeks:


```
<input type="month"
 value="2010-05">

<input type="week">
```

In Opera these look identical to the full date picker, but some browsers may choose to implement a custom UI.





Cartoon: uf012317. All dialogue changed.

3.3 Validation

Validation is often a crucial issue on the web, both for security and general smooth operation of apps, but it's something that content authors very frequently get wrong. HTML5 has built in form validation features, in this section we'll look at how you can specify that filling in certain fields is required, delimit numeric inputs with maximum and minimum values and define arbitrary format requirements for any other text field with regular expressions. We'll

then go on to look at how these features interact with CSS and JavaScript to allow you to give useful feedback to your users.

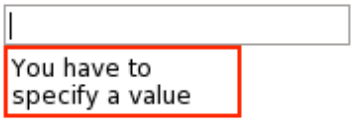
Cartoon: uf012314. All dialogue changed.

If you're accepting input from users through forms then there should always be validation going on at the server, but it provides a better experience to let users know they've made mistakes immediately rather than let them fill out the whole thing, submit, wait for the response and only then find out they made a mistake at the beginning.

3.3.1 The required attribute

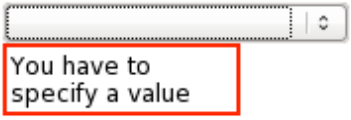
The simplest form of validation is to mark a field as required. In HTML5 this is done by adding the required attribute. If an input is marked as required the browser should not allow the form to be submitted until a value has been given by the user.

This image shows what happens when a text input is marked as required and the user tries to submit the form without entering a value.




```
<input type="text" required
  name="myrequiredfield">
```

The required attribute can be added to any type of input:




```
<input type="date" required
  name="myrequireddate">
```

As the image shows, the results of not entering a value are the same.

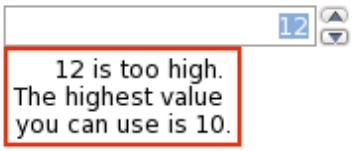
Note that a `name` attribute is included in the example, this is what will label the field's value when it is sent to the server. Without a value for `name` the field cannot be submitted, so the field will not be automatically validated.

3.3.2 The min, max and pattern attributes

The only native validation built into HTML4 for the text input is the `maxlength` attribute. This allows you to specify the maximum number of characters the user is allowed to enter, which is somewhat useful for things like dates and phone numbers, where there is a well defined length, but it's not much use for anything else.

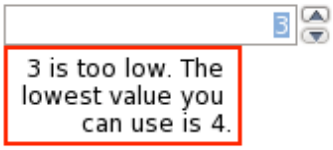
If you wanted a number between 1 and 99 you could set the `maxlength` to 2, this would stop people entering '100' but not '-1'. And if you were looking to limit people between 1 and 50 it's even more useless.

In HTML5 you can use the min and max attributes on the number input type. You already saw these attributes on the range control, where they specified the limits of the slider. On the number control, they will trigger an error when the user tries to submit the form.



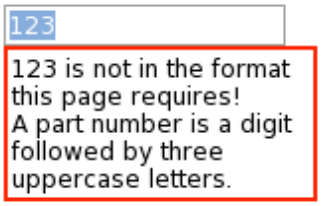

```
<input type="number" max="10"
  name="exnumber">
```

The min attribute works in exactly the same way:




```
<input type="number" min="4"
  name="exnumber">
```

There is also help if the format you require is somewhat more exotic. You can use the pattern attribute to supply a regular expression which will then be used to validate the field. This example is taken from the HTML5 spec:




```
<input type="text" name="partno"
 pattern="[0-9][A-Z]{3}"
 title="A part number is a digit
 followed by three uppercase
 letters.">
```

If the validation fails the browser will display the value of the title attribute, so you should include some information there that will help the user in filling out the form.

3.3.3 Taking advantage of validation with CSS

Besides the visible support in the browser for validation, the messages you've seen in the screenshots above, HTML5 also provides some behind the scenes hooks for CSS and JavaScript, allowing you to provide immediate visual feedback.

In CSS there are two pseudo classes which allow you to provide different styles based on whether they are current valid or invalid. Here is a simple pair of CSS rules to put a green outline round the valid controls, and a dotted red outline round the invalid controls:


```
input:valid {
    outline: 5px solid green;
}

input:invalid {
    outline: 5px dashed red;
}
```

The top screenshot shows the result of applying this CSS to these three number controls.

```
<input type="number" required>
<input type="number" min="4"
  value="4">
```

```
<input type="number" min="4"
  value="3">
```

In real life, of course, we wouldn't specify something invalid as the default value! Note that the validity state applies even though the elements have not been given names.

The same CSS works equally well with text controls using the pattern attribute. The three images shown here are all based on the example from section 3.2.2:


```
<input type="text"
 pattern="[0-9][A-Z]{3}">
<input type="text"
 pattern="[0-9][A-Z]{3}"
 value="1abc">
<input type="text"
 pattern="[0-9][A-Z]{3}"
 value="1ABC">
```

There's also CSS support for styling required controls differently through pseudo classes:

```
input:required {
    outline: 5px dashed blue;
}
input:optional {
    outline: 5px solid green;
}
```

The image shows the result of this CSS applied to these two inputs:

```
<input type="number" required>
<input type="number">
```

3.3.4 Turning off validation

Sometimes you want the user to be able to submit the form without triggering the validation, for example if it was a long form with many sections you might want to let the user save their progress and come back and complete the rest of it later. For this HTML5 provides two new attributes, `novalidate` and `formnovalidate`.

The `novalidate` attribute can be applied to the form element itself whereas the `formnovalidate` attribute affects the whole form but should be applied only to a submit button:

```
<input type="submit" value="Save for Later" formnovalidate>
```

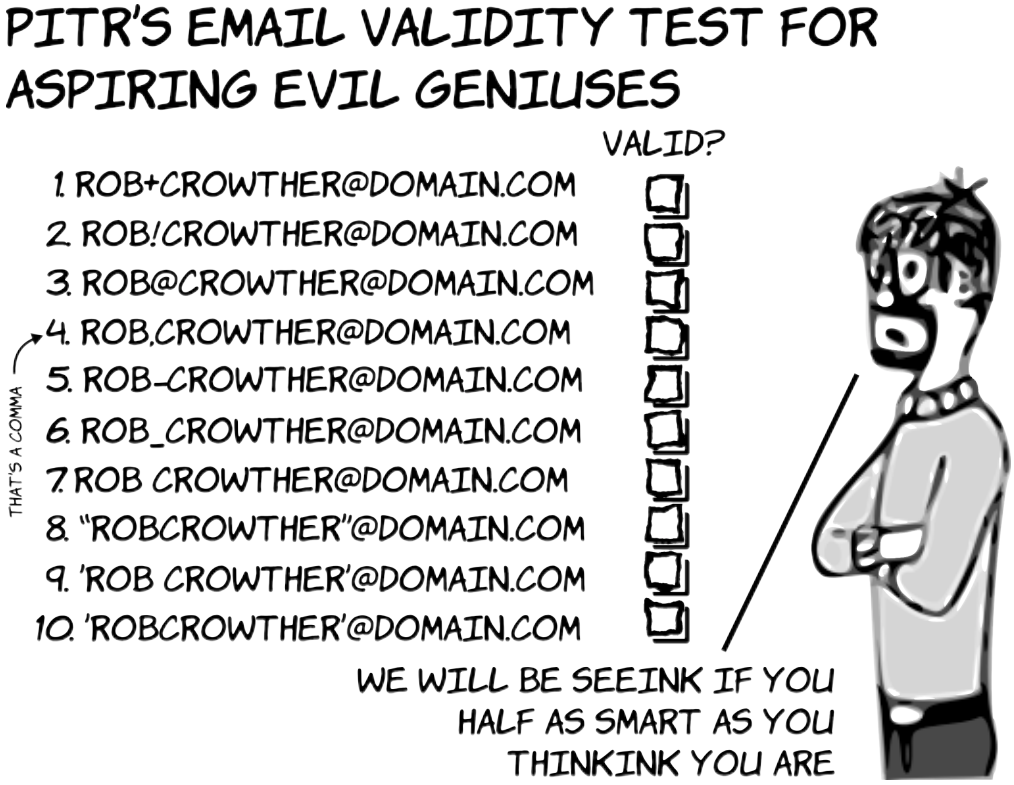
3.4 Emails and URLs

Now you've seen the validation features, let's consider two further new input types, email and url. We didn't look at them until now because, without HTML5's validation features, they look and behave identically to HTML4 text inputs.

On the web there are probably more forms where you need to enter an email address then there are where you don't.

3.4.1 Email addresses

Having seen the pattern attribute you're probably thinking it would be straightforward to either write your own regular expression or find one on the web and then implement your own email field. The problem is, you'd probably get it wrong. Email had reached the popular consciousness even before the web was born, and despite some confusion, most people would be able to recognize an email address, including you. So let's have one of our resident experts do a quick test to see if you really know what an email address looks like.



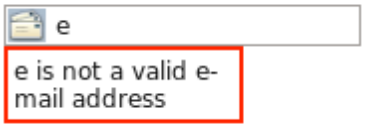
Easy? Half of them are valid and half of them invalid, Pitr will tell you which is which at the end of the section. Note by valid we mean they're constructed correctly, not that you'll be able to send email to them successfully. The reason you'd be almost certainly wrong if you implemented it yourself is that even the experts can't agree on what a valid email address looks like. The HTML5 spec itself is 'willfully violating' the standard:

> A valid e-mail address is a string that matches the ABNF production 1*(atext / ".") "@" ldh-str 1*("." ldh-str) where atext is defined in RFC 5322 section 3.2.3, and ldh-str is defined in RFC 1034 section 3.5.
>
> This requirement is a willful violation of RFC 5322, which defines a syntax for e-mail addresses that is simultaneously too strict (before the "@" character), too vague (after the "@" character), and too lax (allowing comments, white space characters, and quoted strings in manners unfamiliar to most users) to be of practical use here.

To the right is an example of an email input. Here is the code:

```
<input type="email">
```

Opera 11 displays a small email icon, in older versions the field looks like a normal text input. If you type in an invalid email address and hit submit you will get an error message similar to this one.



In HTML5 the invalid emails addresses are beink 3, 5, 7, 8 and 9. 1, 2, 4, 6, and 10 are beink valid. Be notink that even browsers sometime beink wronk, to be tryink rob%crowther@domain.com in Opera and Chrome to see incompatible.

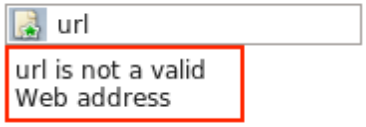
3.4.2 Web addresses

Forms where you enter a URL are also very common; think about the last time you posted a comment on a blog, it's likely there was a URL field there. Like email addresses valid URLs have some esoteric rules, HTML5 means you don't need to worry what they are.

Creating a URL control is as easy as changing the type:

```
<input type="url">
```

Again, Opera 11 presents an icon to indicate a URL is required, and when you attempt to submit the form you get a similar message.



3.5 Elements for user feedback

There are times where you may want to show the user a result, something calculated from the values on the rest of the form – think of a shopping cart which shows the running total of the user's expenditure. In HTML4 you could use JavaScript to insert the value into a read only field, or you might have simply written the value into the HTML content, but there was

no way of indicating that the field or the value had any sort of relationship to the form values, or even that it was part of the form at all. HTML5 changes all that with it's three built in form controls for user feedback: output, progress and meter.

3.5.1 The output element

The `output` element allows you to declare a relationship between one or more input elements and it's own value. The value of the output element can be anything you'd have been happy to put in an `input` element in HTML4, such as text, numbers and dates.

To see output in action, first create a couple of numeric form inputs:

```
<label for="one">Number: </label>
<input type="number" name="one">
<label for="two">Range: </label>
<input type="range" name="two"
 min="0" max="10">
```

Now add an output element:

```
<label for="out">Output: </label>
<output id="out" for="one two">
    0
</output>
```

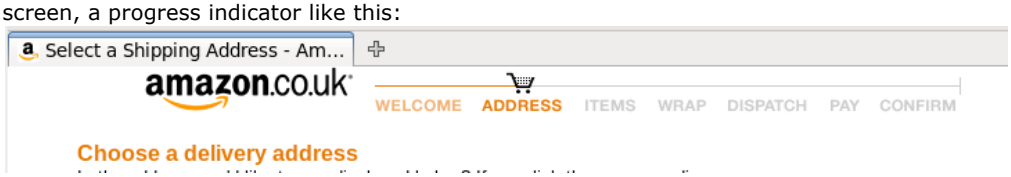
The value goes between the tags rather than in an attribute as with an `input` element. The `for` attribute indicates with which fields the `output` is associated. Finally we need some script to update the output element when there is input, we'll do this on the parent fieldset element:

```
<fieldset
  oninput="out.value =
    one.valueAsNumber +
    two.valueAsNumber;">
```

The exact relationship between the input elements listed in the `for` attribute and the `output` value has to be defined in code, HTML5 does not attempt to guess whether you want to add, subtract, multiply or calculate interest. We'll discuss the `oninput` event later in this chapter.

3.5.2 Progress

You often see complex processes broken into several steps. Buying a book at Amazon, for instance, usually proceeds through pages for entering your personal information, entering your credit card details, and entering the delivery details, among others, before finally confirming your purchase. If you go through this at Amazon you will notice, at the top of the screen, a progress indicator like this:



This follows user interface design best practice: when you're putting a user through a multi-step process always give them an indication of how far through they are. This is such a common requirement, HTML5 adds a special element to support it:

```
<progress value="5" min="0" max="9">5 out of 9</progress>
```

Progress 

WebKit has an initial implementation in their nightly builds:

Browser quick check

	Standard	Prefixed
IE	4.0	-
Firefox	3.5	-
Chrome	4.0	-

Safari	3.1	-
Opera	10.0	-

Please place above table in sidebar

The max attribute gives the value that represents 'completion' and the value which indicates how close to completion we are. The progress element can contain phrasing content – text, inline markup and images – but no other progress elements, so you could represent Amazon's progress bar above like this:

```
<progress max="7" value="2">
    <img src="order-progress-step-2.png" alt="Step 2 of 7">
</progress>
```

The progress element can also be used in applications to give feedback on long running operations like file uploads.

The progress element has strong native semantics for the purposes of ARIA. It should be reported automatically as being in the ARIA progressbar role by supporting agents. As we discussed in the previous chapter this improves the accessibility of your web apps with no extra effort on your part.

3.5.3 Meter

Meter is similar to the progress element but more general purpose in its semantic value. It should be used to indicate a scalar measurement within known bounds, for example disk space usage, progress through an audio track or share of a popular vote.

Chrome has support for the `meter` element in version 6, here's what it looks like.

```
<label for="exmeter">Meter</label>
<meter id="exmeter"
 value="3" min="1" max="6"
 high="5" low="2" optimum="3">
    3 out of 6
</meter>
```

Meter

Note the high and low attributes – if the value encroaches into them it has a visible effect:

```
<label for="exmeter">Meter</label>
```

Meter

```
<meter id="exmeter"
 value="5" min="1" max="6"
 high="5" low="2" optimum="3">
    5 out of 6
</meter>
```

As with the `progress` element, browsers which don't support the `meter` element will display the fallback content between the opening and closing tags. Although we used text here, you could include an image, or even some SVG, which more closely resembled the rendering of the browsers which do support `meter`.

3.6 Less common form controls

Numbers, dates and times, email addresses and URLs – these are all fields you're likely to have been needing in nearly every form, but HTML5 doesn't stop there. There are some additional form controls which you'll need either less regularly, or if you're building particular types of web application. These controls so far lack implementations or common use cases, but they may well see more uptake in the future as HTML is used for more desktop style (or mobile) applications.

3.6.1 Telephone numbers

You don't often need a telephone number in a web app, but they are a common requirement for things like credit card forms, so HTML5 has an input type for them:

```
<input type="tel">
```

However the format for a telephone number is unpredictable. Phones tend to just deal in strings of digits, but people tend to break them up into international dialing codes, area codes and local numbers with spaces, brackets and dashes. The HTML5 spec therefore does not specify and format, so the tel type is basically the same as a text input other than in it's semantic content. If you want to enforce a particular format on the tel field you'll have to use the pattern attribute we discussed in 3.2.2.

What value can these form fields have? The tel type seems particularly useless, it doesn't even offer validation.

It's an accessibility win - you can be told what type of information the field expects even if the page content doesn't make it clear.

Surely there's more to it than that?

The other place where it'll be helpful is in form auto-fill functionality, the browser can offer only email addresses you've previously used for email fields and phone numbers for telephone fields. Mobile browsers would be able to automatically insert your own phone number.

My browser seems to do pretty well with that already.

But it's based on guessing and heuristics, the extra information will make it more accurate.

Character images should face each other with talk in between

3.6.2 Color

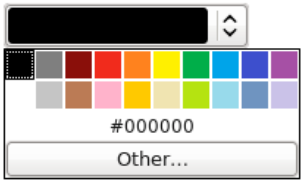
Color is not so common in today's web forms but, as one of the key focuses of HTML5 is to enable HTML applications, color pickers are likely to be a more common requirement in the future. The first implementation of the color input type is in Opera 11.

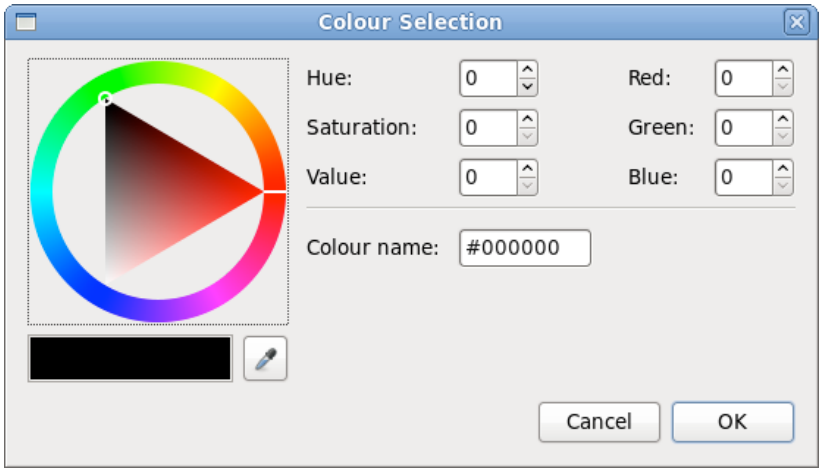
The HTML5 markup for a color picker is:

```
<input type="color">
```

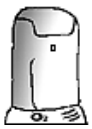
The default value is `#000000`, selected values will always be in #rrggbb hexadecimal shorthand.

When the user expands the control a selection of common colors is presented. Currently there is no way to configure this set of colors, but clicking the 'Other...' button brings up the full color picker you can see below.





3.6.3 Keygen



The purpose of `keygen` is to provide an API into your operating system's cryptography store. It allows public/private key exchange to take place between you and the server. If that doesn't make any sense to you, it's safe to skip ahead to the next section.

`Keygen` originated as a proprietary feature in Netscape Navigator. It was then reverse engineered by Opera and WebKit. As long as the element is useful, the HTML5 way is to document existing behavior so that everyone can implement in an inter-operable manner. We won't be using `keygen` in this book, as it depends on relatively complex server side code to be useful, it is mentioned here for completeness.

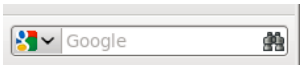
You've now learned about the many new input types and elements available in HTML5 forms, but it doesn't stop there. In the next section you'll investigate the many new form features which are not tied to specific elements.

3.7 New attributes for the input element

As well as all the new form controls in HTML5, existing HTML4 form controls have been extended with new attributes. You've already seen some of these in the section on validation, where we covered attributes such as `required` and `pattern`, but there are several others that can be applied to most input elements: `placeholder`; `autofocus`; and `autocomplete`.

3.7.1 Placeholder Text

A popular technique in recent years as been to put a suggestion for the required user input of a field in the field by default, this is called placeholder text. Here's a few examples of placeholder text on the Firefox search bar and on the Wordpress login



This approach is popular with designers because, if you use field name as placeholder, it doesn't take up any extra space yet still provides the user with useful information about the expected input. It's also a compact way of indicating the desired input format.

There are several common approaches to achieving this look with JavaScript, but these mostly boil down to two alternatives:

- Make the input transparent and place the label element behind it
- Hide the label element but copy the text of the label into the input

You then have to add JavaScript to remove the placeholder text when a user clicks on the control, and put it back again if the user leaves the control without entering a value. But there are a number of issues that can occur:

- Errors in the JavaScript can stop the placeholder text being removed when a user clicks into the element and if users have JavaScript disabled then the text won't work properly or at all
- It can interfere with browser 'form fill' functionality that remembers values for frequently used forms
- Assistive technology has no way of distinguishing between placeholder text and valid input

HTML5 has support for placeholders built in thanks to the new placeholder attribute. Simply add the attribute `placeholder` to the input element with the value you want to use.

```
<input type="email"
 placeholder="email@example.com">
```

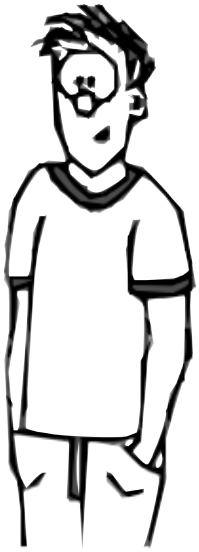
You are protected from JavaScript errors causing issues because the functionality no longer depends on your (or indeed, any) JavaScript. Browsers can handle the interaction with native functionality better because everything is now under their control, and for the same reason the browser can keep assistive technology better informed of what's going on.

3.7.2 Form autofocus

Many web forms, as a convenience, use JavaScript to put the focus on the first input element when the page loads. For instance, if you visit the Google home page and start typing the text will appear in the search field in the middle of the page. Rather than have everyone write their own JavaScript routine to achieve this, HTML5 adds support directly to HTML:

```
<input type="text" autofocus>
```

Am seeink autofocus attribute. Is not much shorter than JavaScript to do same thing. Why to be bothering?



Making it part of the browser functionality gives several important benefits – it means it won't interfere with other browser functionality, it'll work without JavaScript enabled and users will be able to turn it off if they don't like it by changing a browser setting.

Why be disablink this useful function?

Advanced users can find it annoying having the focus stolen when they've already started typing somewhere else, particularly on large and complex pages. It also presents a problem for users with screen readers – a screenreader reads a web page in a linear fashion, if the focus starts out at a form field half way through the document it's likely a lot of context, explaining the purpose of the form, has been skipped

Can already turn off thinks they don't like. Like me, can be disablink in source code and recompilink browser.

People who are still at the trainee stage of evil genius will probably find a browser setting a bit easier to manage.

Da. Supposink so. Standards of education droppink always.

Character images should face each other with talk in between

3.7.3 Protecting private information with the autocomplete attribute

The `autocomplete` attribute allows you to provide a hint to the browser that the values entered into the field should not be remembered for future use by the browser's auto form filling functionality. This could be because the field accepts information which is supposed to be secret, for instance a PIN number, or because the field is expecting a one off value where past values entered are likely to be irrelevant, such as one off password reset code.

```
<label>Account: <input type="text" name="ac" autocomplete="off"></label>
<label>PIN: <input type="password" name="pin" autocomplete="off"></label>
```

3.8 Extending forms with JavaScript

There are several other enhancements to forms in HTML5 over and above the new controls. These include ways to access the validity information through JavaScript as well as several convenience features that either completely replace the JavaScript you would commonly write for every form today, or make various scripting operations much easier.

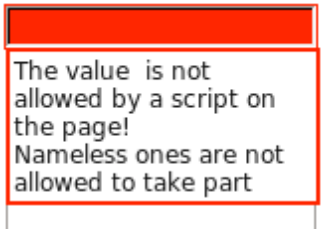
3.8.1 Customizing the validation messages

The default messages are a little bland and, while the pattern attribute allows you to include a custom message in the title attribute, there's no attribute you can use to provide a custom message to the other input types. However, you can supply a custom message in script.

To do this you need to use the setCustomValidity property of the input element in the DOM:

```
var fldName =
document.getElementById('fullname');
fldName.form.onsubmit =
 function () {
  fldName.setCustomValidity("");
  if (!fldName.validity.valid) {
```

```
    fldName.setCustomValidity(
    "Nameless ones are not allowed
     to take part");
  }
};
```

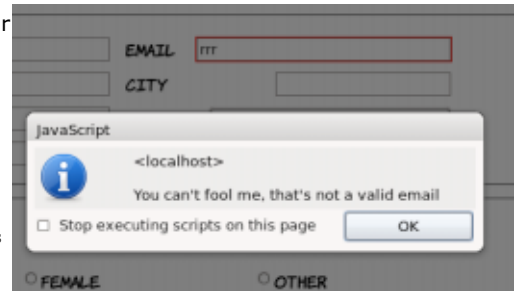


As you can see, your message is displayed but only beneath some standard boilerplate from the browser.

Note that the first step in the function in the listing above resets the custom validity message to an empty string. This is because setting the custom validity message forces the valid status to be false, it will be impossible to submit the form after the error if the message isn't reset.

You can take more complete control of the user experience while still taking advantage of the built in validation by making use of the validity property in the DOM.

To do this you override the `invalid` event on the form element in question, and add your own code to report the validity status:




```
var fldEmail =
  document.getElementById('email');
fldEmail.addEventListener(
  'invalid',
  function(event) {
    alert("You can't fool me, that's
           not a valid email");
    event.preventDefault();
  }
  ,false);
```

Note that the event handler function cancels the default event in order to prevent the message from the browser also appearing.

3.8.2 Triggering validation with JavaScript

Normally a supporting browser will only trigger the form validation when the form is submitted. This is very sensible, having the error messages pop up repeatedly while the user is trying to fill in the form would be very distracting.

There may be occasions where you want to trigger the validation without submitting the form – possibly the the allowed values for a later field in the form depend upon the user already having entered a valid email address, or perhaps the input isn't part of a form at all.

If you have an email input in your HTML like this:

```
<input id="email" type="email">
```

You can trigger the validation by calling the checkValidity() method:

```
document.getElementById('email').checkValidity();
```

In Opera the field will be validated as it would be when the form is submitted, so if the email is invalid the normal message will pop up, though this is not required by the spec. The method will also work in Firefox 4 and recent Chrome and Safari releases, returning `false` if the email is invalid. You can use the return value to implement your own notifications.

3.8.3 Responding to any changes in value

Before HTML5, when you had to write your own form validation code, it was a common technique to attach your JavaScript validation to the `onchange` event handler on each individual input. Even if you extract your validation code into a single function you still have to attach event handlers to every element, and remember to update every time you add a new field.

An event handler is a hook HTML provides to JavaScript to allow you to run code when particular things happen. You might want to know when the page finishes loading (`onload`), or when the user clicks on something (`onclick`) or, as in this case, when the user leaves an input field in which value had changed (`onchange`). When the event happens it is said to 'fire' Check Appendix D for more on event handlers.

HTML5 provides a new event – `oninput` – which is fired by any form element when the value changes, as the value changes. You've already seen this feature in action when we discussed the output element. Let's compare the code we wrote to power the output element earlier (on the left) with a version that relies on attaching to the event handler of each field (on the right):

```
<fieldset oninput="value =
    one.valueAsNumber +
    two.valueAsNumber">
```

```
<fieldset>
  <label for="one">Number: </label>
  <input type="number" name="one"
```

```
  <label for="one">Number: </label>
  <input type="number" name="one">

  <label for="two">Range: </label>
  <input type="range" name="two"
   min="0" max="10">

  <label for="out">Output: </label>
  <output id="out" for="one two">
    0
  </output>
</fieldset>
```

```
   onchange="out.value =
      one.valueAsNumber +
      two.valueAsNumber">

  <label for="two">Range: </label>
  <input type="range" name="two"
   onchange="exoutput1.value =
      one.valueAsNumber +
      two.valueAsNumber">

  <label for="out">Output: </label>
  <output id="out"
   for="one two">6</output>
</fieldset>
```

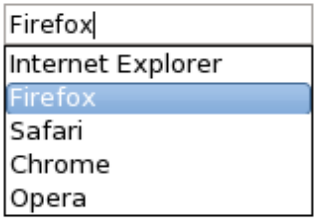
Having to handle changes on each input element individually can lead to extra code, but there's a more important advantage: The `onchange` event only fires after the user leaves the field – by tabbing out of it or clicking elsewhere on the page. The `oninput` event fires for any change.

3.8.4 Create combo-boxes with datalist

One type of form control which is common in desktop applications but not available in HTML4 is the combo box. This gets its name because it is the combination of a text box and a select list – the user can select from a list, but they can also free type a value which is not on the list. When Ajax was becoming popular one of the most common features of the early JavaScript libraries was support for creating combo box like features.

HTML5 adds support for this directly into the markup with the `datalist`. A `datalist` is a named list of options, similar to the list of options in a select element, which can then be associated with one or more input elements using the list attribute.

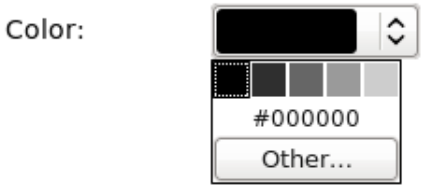
In this example, the input has been has been associated with a `datalist` with id browsers. When a user selects the input the list of options pops up, the user can pick one of the options using the cursor keys or time his own.

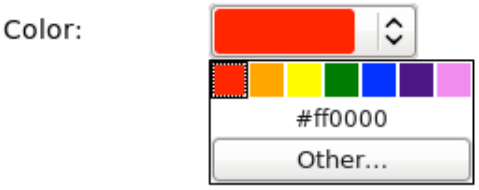



```
<input type="text" name="browser"
 list="browsers">
<datalist id="browsers">
    <option
     value="Internet Explorer">
    <option value="Firefox">
    <option value="Safari">
    <option value="Chrome">
    <option value="Opera">
```

```
</datalist>
```

Opera, at the time of writing the only browser to implement the `color` input type, allows for `datalist` to be attached to that. This sets the default colors available on the initial dropdown:






```
<input type="color"
       list="greyscale">

<datalist id="greyscale">
    <option value="#000000">
    <option value="#333333">
    <option value="#666666">
    <option value="#999999">
    <option value="#cccccc">
</datalist>
```

```
<input type="color"
       list="rainbow">

<datalist id="rainbow">
    <option value="#FF0000">
    <option value="#FFA500">
    <option value="#FFFF00">
    <option value="#008000">
    <option value="#0000FF">
    <option value="#4B0082">
    <option value="#EE82EE">
</datalist>
```

3.8.5 Easy ways to work with form values in JavaScript

This is another feature you've already seen in action. When dealing with forms in JavaScript the value is always a string, even if it represents a date or a number. Because JavaScript will automatically convert the types of any value involved in an expression this can easily lead to hard to spot errors.

This code looks very similar to the code we saw in 3. and 3.4.5, but doesn't have the results we might expect:

```
<label for="one">Number: </label>
<input type="number"
 value="5" name="one">

<label for="two">Range: </label>
<input type="range" name="two"
 min="0" max="10" value="3">

<label for="out">Output: </label>
<output id="out" for="one two"
```

```
onforminput="value =
 one.value +
 two.value">0</output>
```

Number: 5

Range:

Output: 53

Can you spot the difference? Compare the last two lines with what we used earlier (on the right):

```
one.value +
two.value">0</output>
```

```
one.valueAsNumber +
two.valueAsNumber">0</output>
```

We might expect 5 + 3 to be 8, but because the form values are strings we are not performing addition, we are performing concatenation. It is relatively straightforward to work around the issue in HTML4, but why should you have to? HTML5 provides the properties `valueAsDate` and `valueAsNumber` so that you can get directly at the values you need.

3.9 Browser support and detecting HTML5 features

Unlike the structural elements we looked at in chapter 2, the new form elements have more complex associated behavior and APIs. The structural elements just had to exist, these form elements have to do something for us to be able to say a browser supports them. With these more complex requirements it's not surprising that support isn't yet as far advanced as it might be. The table below shows the level of support in the current and, where known, upcoming versions of all the major browsers at the time of writing.

Key	Complete or nearly complete support:		Incomplete or partial support:		Little or no support:

Browser	Input Types	Validation API	Placeholder / Autofocus	Input UI	Range / Meter / Progress	Output

Browser	Input Types	Validation API	Placeholder / Autofocus	Input UI	Range / Meter / Progress	Output
Chrome 5						
Chrome 6						
Safari 4						

WebKit isn't just about showy CSS3 stuff you know, it's a rendering engine for getting stuff done!

WebKit was one of the first browsers to support the new input types, enabling keyboards tuned to the required input type on the iPhone. The latest versions support the validation API but you need to write your own code to take advantage of it. There is no support for the `output` element, but you can always access the value with `innerHTML` instead.

Browser	Input Types	Validation API	Placeholder / Autofocus	Input UI	Range / Meter / Progress	Output
Firefox 3.6						
Firefox 4						

The folks at Mozilla are keen to get this stuff implemented but they also want to get it right.

Firefox 4 has support for HTML5 forms in the beta release, including datalist. Firefox 4.1 will likely support everything but the UI for the new input types. Firefox 4 will also have default styling for the `:invalid` pseudo class, if you want to turn that off use the following in your CSS:

Browser	Input Types	Validation API	Placeholder / Autofocus	Input UI	Range / Meter / Progress	Output

```
:invalid { box-shadow: none; }
```

Firefox also adds an experimental attribute, `x-moz-errormessage`, to allow you to customize the error message:

```
<input type="email" name="email"
    x-moz-errormessage="Email please!">
```

IE 8

IE 9

This must be some sort of misprint! Either that or this stuff just isn't very important – IE is the best browser, I tell you!

Internet Explorer has no support for HTML5 forms in version 8, and none has yet been announced for the version 9 betas.

Opera 10.5

When it comes to web standards Opera is always leading the way.

Opera was the only browser to add support for the Web Forms 2 standard, which made it easy for them to support the HTML5 forms. Support for `color`, `tel`, `meter` and the `placeholder` attribute was added in Opera 11.

The table above is split into rows which indicate the support in various versions of browsers, each then followed by a 'colorful'

comment by the browser's character advocate and then a 'factual' comment from AJ.

3.9.1 Detecting supported features

As mentioned above, if a browser has no support for one of the new form input types it will convert it to an input of type text. This makes it quite easy to detect whether or not an input type is supported in JavaScript – just create an element of the desired type, then immediately look to see if it is a text input:

```
var el = document.createElement("input");
el.setAttribute("type", "date");
if (el.type == "text") {
    implementDateValidation();
}
```

In the listing above you create a date input and then check to see what type the browser thinks it is. If it's 'text', call a function `implementDateValidation` to deal with browsers which don't support the date input type.

For date inputs we have to go one step further than this – if you remember, WebKit implements the date input type but doesn't provide any UI for it. To detect if a UI is provided, set a value on the date element that isn't a date:

```
var el = document.createElement("input");
el.setAttribute("type", "date");
el.value = "text";
if (el. value == "text") {
    implementDateUI();
}
```

If the browser implements the date UI components then it will be impossible to set the value of it to the string 'text', therefore if the element reports it's value as 'text' after you've set it then the UI is not implemented by the browser and you should provide our own. You might also consider setting an appropriate pattern attribute at this point.

You may also want to check if the user's browser supports one of the new form attributes, such as `autofocus` or `placeholder`. Here's some code to do this:

```
var el = document.createElement("input");
if (!!('placeholder' in el)) {
```

```
    window.alert('Placeholder supported');
} else {
    window.alert('Placeholder not supported');
}
```

The easiest approach is to loop through the available properties on an element and see if one of them is the attribute you're looking for. This same approach can be used for any of the other new HTML5 attributes, not just form elements.

The final thing you might want to check is if the browser supports a particular event, such as the `oninvalid` event we discussed earlier:

```
var eventName = "oninvalid";
var isSupported = !!(eventName in el);
if (!isSupported && el.setAttribute) {
    el.setAttribute(eventName, 'return;');
    isSupported = typeof el[eventName] == 'function';
}
if (isSupported) {
    window.alert('oninvalid supported');
}
```

The above code tries two different approaches. First it looks to see if the `oninvalid` event exists in the element properties. If that fails it tries to set the event on the element and, similarly to the input type detection above, it looks to see if the the type of the attribute is a function.

You don't have to write all this detection code yourself, there's already a library which will do all the work for you; check out the modernizr library at http://www.modernizr.com/. If you don't fancy writing any of your own form validation code, you could try a different library which enables HTML5 forms support in all browsers: html5-now

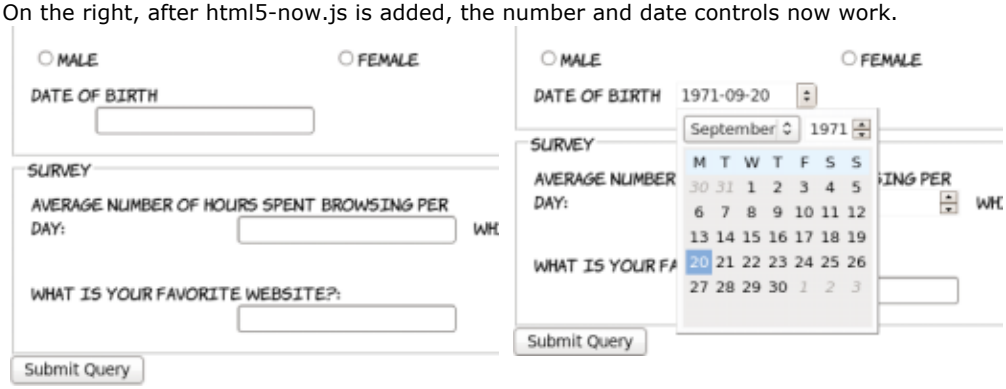
3.9.2 The html5-now library

HTML5-Now is an open source project started by Dean Edwards. Dean is famous for writing several drop-in scripts for old versions of Internet Explorer which made them behave in a standards compliant manner. The aim of html5-now.js is to provide a drop-in solution which patches the browser's holes in HTML5 support. It's in alpha currently, but already provides a

lot of support for HTML5 form controls. Download it from http://code.google.com/p/html5-now/ and then include it in your page like this:

```
<script src="html5-now/html5-now.js"></script>
```

The result of adding the script to our form can be seen in the screenshots below. On the left a screenshot of our HTML5 form in Firefox 3.6, all of the HTML5 controls render as text. On the right, after html5-now.js is added, the number and date controls now work.



HTML5-now is smart enough to work out if the browser already has support for particular HTML5 features, and won't interfere if that's the case, so it is safe to use across all browsers. But it is a heavyweight script, so if you are only intending to use a small number of HTML5 features you would be better off detecting them directly as per section 3.5.1.

3.10 Summary

In this chapter you've learned about:

- How the new form input types available in HTML5 greatly increase the range of options you had in HTML4
- How you can reduce the amount of JavaScript you have to write to validate input
- Other new features like autofocus and placeholder text
- What support is available in web browsers right now, and how to detect what support your user's browsers have

You should now be ready to take your forms to the next level with HTML5!

In the last two chapters you've learned about HTML5 features which are extensions of common usages of HTML4 markup. In the next few chapters you're going to learn about some of the completely new functionality in HTML5 for dealing with media and dynamic graphics. We'll start in the next chapter with a look at Canvas and SVG, the two HTML5 technologies for drawing graphics in the browser.

4

Dynamic Graphics with Canvas and SVG

In this chapter you're going to learn about HTML5's facilities for dynamic graphics – graphics which can change in response to user input, data or simply time passing. This could include charts representing network activity or the location of people on a map. The key areas you will cover are:

- Using the Canvas element to draw shapes, text and images
- Transforming existing images with Canvas
- Using SVG in your web pages
- The strengths and weaknesses of Canvas and SVG
- Cross browser support

This chapter, especially the parts to do with the Canvas element, will make a lot of use of JavaScript. If you're not familiar with JavaScript you should check out Appendix D before proceeding.

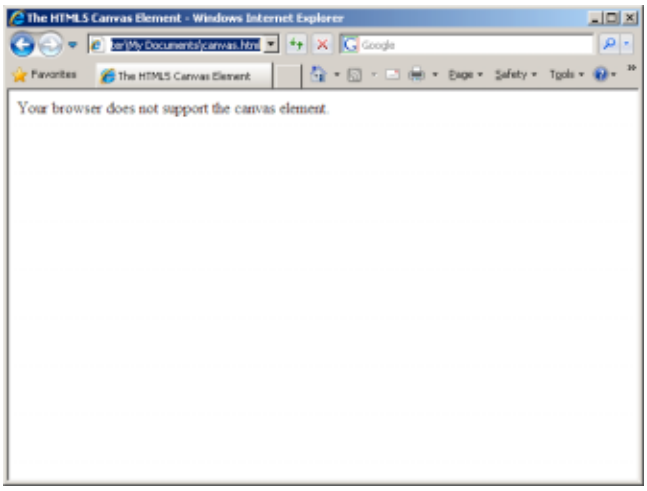
4.1 Getting Started with Canvas

The canvas element itself is very straightforward. It is similar to an image element in that you can specify a width and a height, but it has starting and closing tags which can enclose fallback content and doesn't reference an external source:

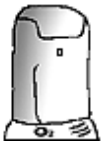
```
<canvas id="mycanvas" width="320" height="240"
```

```
        style="outline: 1px solid #999;">
    Your browser does not support the canvas element.
</canvas>
```

In a browser which doesn't support canvas the fallback content will be displayed, as in this screenshot:

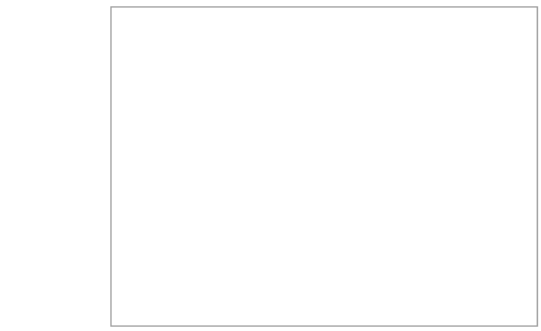


You might have a static image as the fallback if that could adequately present some of the information that would be displayed on the canvas supporting browsers or, if you were particularly ambitious, you might use an alternative rendering method such as Flash.



The examples in this section should work in all versions of Safari, Chrome and Firefox 2.0 and Opera 9 and later.

You may be more interested to see what it looks like in a browser which does support canvas:



Wait, there's nothing there! All I can see is an outline.

You told me this canvas thingy was for all sorts of whizzy graphics and games. Like on the Apple website!

So how do we draw the pictures?

We have to? You mean *you* have to, come and get me when there's something worth looking at.

Of course, what were you expecting?

Yes, but the element doesn't create the graphics by magic, you have to draw them yourself.

We have to write a JavaScript program to draw the pictures.

Before we get to drawing something there's a couple of things you need to understand. You need to know how to get a reference to your canvas object so that you can send it drawing commands and, since you will be telling the canvas to draw shapes on a grid, you need to know how the grid is defined. First, how to get a reference in JavaScript:

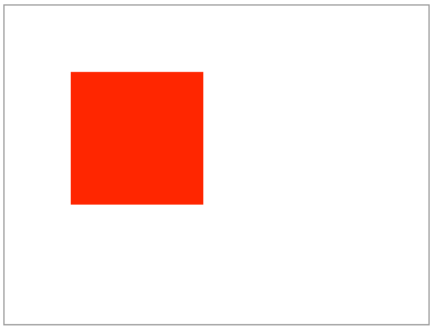
```
function draw(){
    var canvas = document.getElementById('mycanvas');
    if (canvas.getContext) {
        var ctx = canvas.getContext('2d');
        //do stuff
    }
}
window.addEventListener("load", draw, false);
```

You have to pass a parameter, '2d', to the getContext method. This gives you a two dimensional drawing context, currently this is the only parameter supported. Several browser vendors are already experimenting with a three dimensional drawing context with direct access to graphics hardware which will open up possibilities such as 3D games, virtual reality experiences and modeling tools.

4.1.1 Drawing Shapes

To draw on the canvas we need to get a drawing context. The context then gives us access to methods which allow the drawing of lines and shapes.

Basic shapes are easy, if we extend the code from section 4.1 we can draw a rectangle by using the fillRect method. The only prerequisite is that we first have to set the fill color using the fillStyle method.



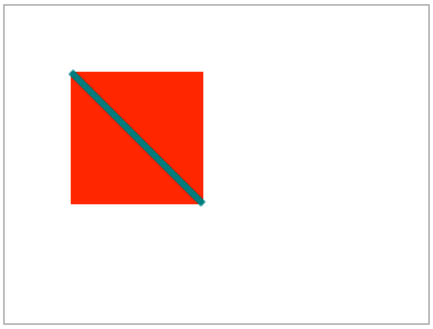
```
if (canvas.getContext) {
    var ctx =
           canvas.getContext('2d');
    ctx.fillStyle = 'rgb(255,0,0)';
    ctx.fillRect(50,50,100,100);
}
```

We call the fillRect method with four arguments, the x and y values of the top left corner and the width and height to fill.

Aside from fillRect there are also methods to clear an area of pixels and to draw an empty rectangle: clearRect and strokeRect respectively. They take the same for parameters as fillRect.

Let's extend the code to draw a line. Lines are a little more complex. You have to first draw a path, but the path doesn't appear until you apply a stroke.

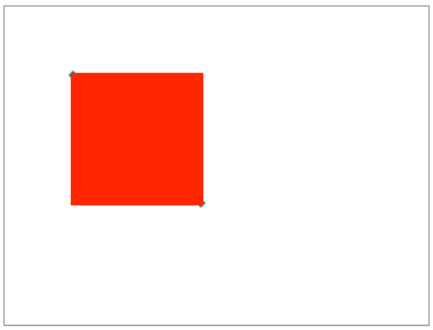
```
ctx.fillStyle = 'rgb(255,0,0)';
ctx.fillRect(50,50,100,100);
ctx.strokeStyle = 'rgb(0,127,127)';
ctx.moveTo(50,50);
ctx.lineTo(150,150);
ctx.lineWidth = 5;
ctx.stroke();
```



If you've ever used graphics software like Photoshop this process should be familiar to you. The moveTo method moves the 'pen' without recording a path, the lineTo method moves the pen and records a path.

Now for a little experiment. What happens if we draw the line first and then the box?

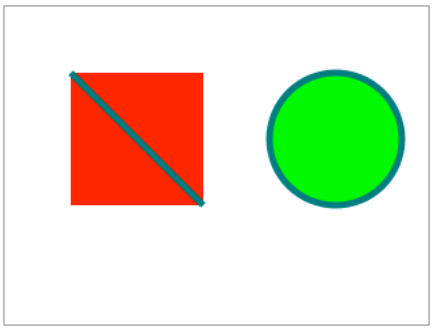
```
ctx.strokeStyle = 'rgb(0,127,127)';
ctx.moveTo(50,50);
ctx.lineTo(150,150);
ctx.lineWidth = 5;
ctx.stroke();
ctx.fillStyle = 'rgb(255,0,0)';
ctx.fillRect(50,50,100,100);
```



As you can see the line is mostly obscured by the rectangle. You might think that if you could remove the rectangle the line would still be there 'underneath', but in fact once you've drawn over it the line is gone.

The only way to get the line back is to erase both the rectangle and the line and then draw the line again. The canvas element doesn't store the elements drawn, only the resulting pixels.

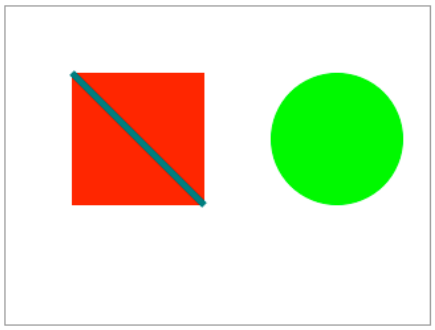
So, what about other shapes? The path then stroke approach is the way to do it. We can use the arc method to draw a circle and then fill it. The arc method accepts parameters for the location of the center, the radius, how far round, in radians, the arc should extend and whether that should be clockwise or anti-clockwise:



```
ctx.fillStyle = 'rgb(255,0,0)';
ctx.fillRect(50,50,100,100);
ctx.fillStyle = 'rgb(0,255,0)';
ctx.arc(250, 100, 50, 0,
        Math.PI*2, false);
ctx.fill();
ctx.strokeStyle = 'rgb(0,127,127)';
ctx.moveTo(50,50);
ctx.lineTo(150,150);
ctx.lineWidth = 5;
ctx.stroke();
```

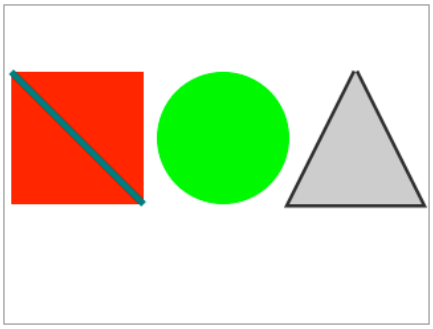
Note that the stroke we use to draw the line at the end also gets applied to the circle.

In order that the stroke for the line not apply to the circle we need to explicitly put them on different paths with the beginPath method:



```
ctx.fillStyle = 'rgb(255,0,0)';
ctx.fillRect(50,50,100,100);
ctx.beginPath();
ctx.fillStyle = 'rgb(0,255,0)';
ctx.arc(250, 100, 50, 0,
        Math.PI*2, false);
ctx.fill();
ctx.beginPath();
ctx.strokeStyle = 'rgb(0,127,127)';
ctx.moveTo(50,50);
ctx.lineTo(150,150);
ctx.lineWidth = 5;
ctx.stroke();
```

Other shapes are just a matter of creating a path and then stroking or filling, or both. If we move our first two shapes over a little, there's room to add a triangle:

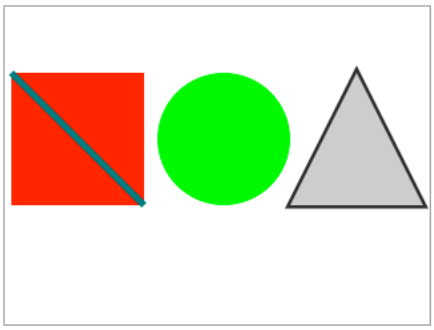


```
ctx.beginPath();
ctx.moveTo(265,50);
ctx.lineTo(315,150);
ctx.lineTo(215,150);
ctx.lineTo(265,50);
ctx.strokeStyle = 'rgb(51,51,51)';
ctx.fillStyle = 'rgb(204,204,204)';
ctx.stroke();
ctx.fill();
```

Notice that, even though our triangle started and ended at the same point there is a slight gap at the top.

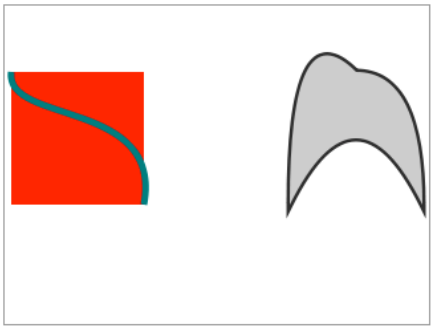
To prevent this we need to close the path using the closePath method:

```
ctx.beginPath();
ctx.moveTo(265,50);
ctx.lineTo(315,150);
ctx.lineTo(215,150);
ctx.lineTo(265,50);
ctx.closePath();
ctx.strokeStyle = 'rgb(51,51,51)';
ctx.fillStyle = 'rgb(204,204,204)';
ctx.stroke();
ctx.fill();
```



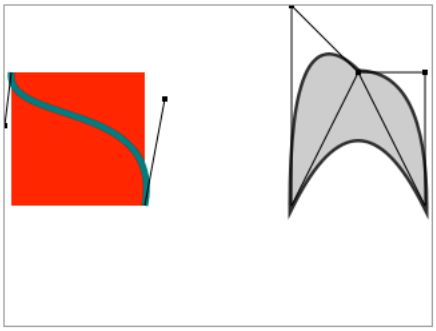
You don't have to restrict yourself to straight lines in paths, instead of lineTo you can use either of two types of curve: quadratic or bezier. Let's replace our first straight line with a bezier curve:

```
ctx.moveTo(5,50);
ctx.bezierCurveTo(0,90,  120,70,
                  105,150)
```



The last two numbers are the end point, preceding that are co-ordinates for two are control points. A quadratic curve is similar but only needs one control point, below is the code with which we drew a triangle using quadraticCurve instead of lineTo:

```
ctx.moveTo(265,50);
ctx.quadraticCurveTo(315,50,
                     315,150);
ctx.quadraticCurveTo(265,50,
                     215,150);
ctx.quadraticCurveTo(215,0,
                     265,50);
```



In the second picture above the control points have been drawn in to help you visualize what's going on. The drawn lines are distorted from their direct path so that they approach an imaginary line drawn between the start or end point and the control point.

As you can see, it is easy to create some interesting shapes but drawing curved lines can be a bit hit and miss, especially if you're trying to get the curve to line up with some other drawn object. The best approach is usually some trial and error.

4.1.2 Adding Images and Text

One of the great features of canvas is that you can use it to manipulate images and text content to achieve effects that are otherwise difficult to do with HTML and CSS. Here are a couple of examples of what can be achieved:

[image missing]

The canvas element cannot download images, you cannot simply give it a URL and expect it to fetch the image. Any image you want to use has to already be available in your page content. There are various ways to do this, but the easiest is just to include the element in the normal way, for now I will hide it:

```
<div style="display: none;">
    <img id="myimage" src="example.png" width="236" height="260">
</div>
```

For the next few examples we're going to take the image below and import it into the canvas element.

The simplest example is to import the image and place it on the canvas:

```
if (canvas.getContext) {
    var ctx =
canvas.getContext('2d');
    var img =
document.getElementById('myimage');
    ctx.drawImage(img, 10, 10);
}
```

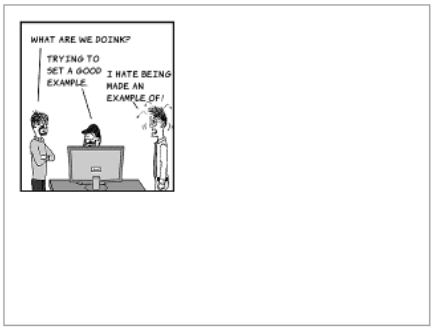
We call drawImage with three parameters – the img element and the x and y co-ordinates.

Our example image is too large to fit into the canvas frame. We can easily fix this by defining a width and height for the placed image:

```
if (canvas.getContext) {
    var ctx =
canvas.getContext('2d');
    var img =
document.getElementById('myimage');
    ctx.drawImage(img, 10, 10, 118,
130);
```

```
}
```

Now we're calling drawImage with five parameters, the additional two are the width and height of the placed image.



Finally, we may not event want all of the original image:

```
if (canvas.getContext) {
    var ctx =
canvas.getContext('2d');
    var img =
document.getElementById('myimage');
    ctx.drawImage(img,
           80, 100, 80,  160,
           10, 10,  160, 200);
}
```

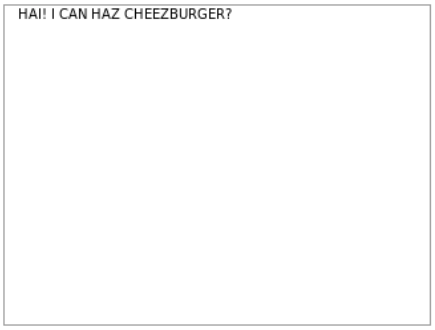
Here we call drawImage with nine parameters. The first one is still the img element. The next four define an x and y co-ordinate and a width and a height for a section of the source image, the final four are an x and y co-ordinate and a width and a height for the placed image.

Let's now turn our attention to text. Canvas has a limited ability to draw single lines of text on to the context, suitable for drawing labels and titles rather than rendering large blocks of text.

The text API has only recently been finalized, so browser support is not as advanced as with other areas of the canvas element, we'll look into that in more detail in section 4.4.

Drawing text on the canvas is easy with the fillText method:

```
ctx.fillText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,10);
```

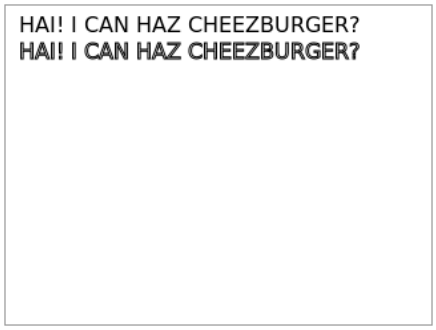


There are three required parameters: a string which is the text to be drawn and x and y co-ordinates to determine where it is to be drawn.

The text will be drawn in the current font, which is determined by setting the font property of the drawing context.

If we set the font size a little larger you can see an alternative method for drawing text. As with rectangles, you can draw the fill and the stroke separately:

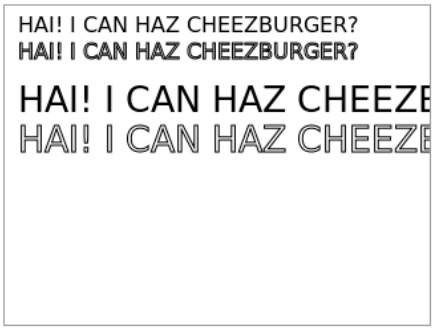
```
ctx.font = "12pt sans-serif";
ctx.fillText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,20);
ctx.strokeText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,40);
```



You can of course draw both fill and stroke on a single line of text if you want.

Let's have a look and see what happens if you increase the font size a bit more, remember our example canvas element is 320 pixels wide:

```
ctx.font = "20pt sans-serif";
ctx.fillText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,80);
ctx.strokeText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,110);
```

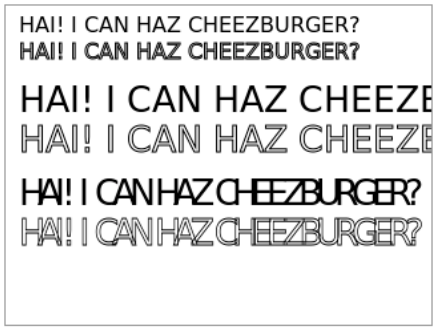


As you can see the text which doesn't fit

simply flows off the edge of the element without wrapping.

To work around this issue we can make use of the fourth, optional, parameter to the fillText and strokeText methods:

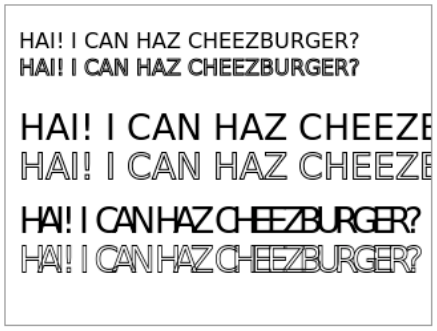
```
ctx.fillText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,150,300);
ctx.strokeText(
 'HAI! I CAN HAZ CHEEZBURGER?',
 10,180,300);
```



The fourth parameter sets a maximum width for the text, if the text is going to be wider than the value passed the browser will make the text fit either by narrowing the spacing between the letters or scaling down the font.

There are a few more properties and methods for manipulating text on the canvas. You can set the baseline of the text, which will adjust where it is drawn in relation to the co-ordinates you provide:

```
ctx.textBaseline = "top";
```



It's quite hard to spot the difference to the image above, but now the baseline is 'top' the text is further down the canvas.

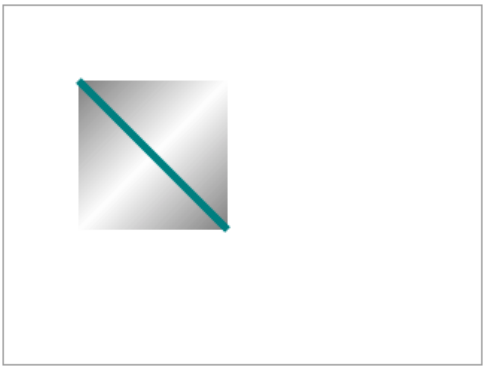
You can now draw images and text on your canvas and done some basic transformations. The simple examples we've covered here may not seem too much more exciting than what can be achieved with plain HTML and CSS, but of course we've barely scratched the surface yet. In the next section we'll learn some more advanced techniques.

4.1.3 Applying Colors, Transformation and Animation

You've already seen several examples of applying styles and colors, we used strokeStyle and fillStyle in section 4.2 to set the color of our lines and shapes. The two style properties can accept any color value you would expect to be able to use in CSS, including semi-transparent RGBA.

As well as colors, strokeStyle and fillStyle can accept a gradient object.

There are two steps. First, you create a gradient then you specify the color stops:



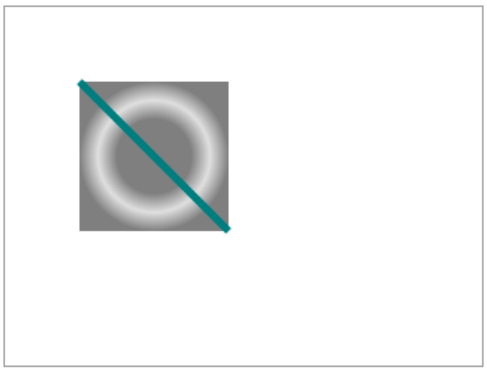
```
var lineargradient =
 ctx.createLinearGradient(
     50,50,150,150
 );
lineargradient.addColorStop(
    0,'rgb(127,127,127)'
);
lineargradient.addColorStop(
    0.5,'rgb(255,255,255)'
);
lineargradient.addColorStop(
    1,'rgb(127,127,127)'
);
ctx.fillStyle = lineargradient;
ctx.fillRect(50,50,100,100);
```

The createLinearGradient method takes four parameters which define the top left and bottom right corners, while the color stops are defined by a distance as a fraction of one and a color. Once the gradient is created it is assigned to the fill style just like we did with a color earlier.

Radial gradients are a little more complex:

```
var radialgradient =
 ctx.createRadialGradient(
    100,100,25,100,100,50
);
radialgradient.addColorStop(
    0, 'rgb(127,127,127)'
);
radialgradient.addColorStop(
    0.5, 'rgba(127,127,127,0.25)'
);
radialgradient.addColorStop(
    1, 'rgb(127,127,127)'
);
```

```
ctx.fillStyle = radialgradient;
```



Note that for both linear and radial gradients you define them with co-ordinates relative to the entire canvas context, not the shape you want to apply it to. You have to make sure you choose your co-ordinates so that the gradient appears in the shape you want to fill it with.

Though the gradient is not confined to the co-ordinates you specify, it will extend out across the canvas. Have a look at the three screenshots below which show our linear gradient created with three different sets of co-ordinates.

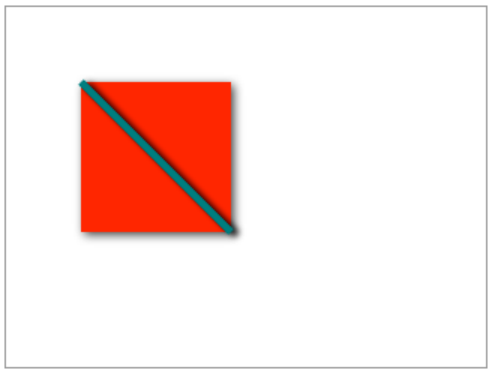
```
createLinearGradient(
  0,0,320,240
);
```

```
createLinearGradient(
  0,0,320,0
);
```

```
createLinearGradient(
  75,75,125,125
);
```

Canvas also has built in support for drop shadows:

```
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 8;
ctx.shadowColor =
    "rgba(0, 0, 0, 0.75)";
```



With shadows you can create effects such as 'cut out' text:

```
ctx.shadowOffsetX = 4;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 5;
ctx.shadowColor =
    "rgba(0, 0, 0, 0.9)";
ctx.fillStyle = 'rgb(0,0,0)';
ctx.fillText('HAI!',170,50);
ctx.fillStyle = 'rgb(255,255,255)';
ctx.fillText('I CAN HAZ',170,70);
ctx.strokeStyle = 'rgb(0,0,0)';
ctx.strokeText('CHEEZBURGER?'
    ,170,90);
```

The canvas 2d context supports a number of transformations. These work on the context itself, so you apply the transformation, then draw whatever you want to appear subject to that transform.

Let's start with a simple translate transformation. This simply moves the origin of the canvas according to the x and y offsets you pass in as arguments.

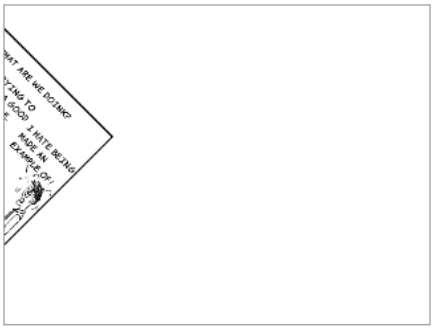
```
var img =
```

```
document.getElementById('myimage');
ctx.translate(120,20);
ctx.drawImage(img, 10, 10, 118,
130);
```

If you compare this example with the similar one in section 4.3 without the transformation you'll see we've basically moved the image down and to the right. Not particularly useful when we could have simply drawn the image there in the first place, but it would be useful if you wanted to move a collection of objects around while keeping their relative positions the same.

Next, let's try rotation.

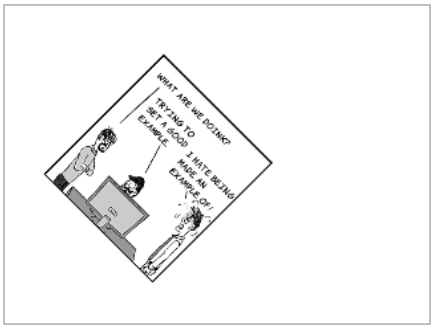
```
var img =
document.getElementById('myimage');
ctx.rotate(Math.PI/4);
ctx.drawImage(img, 10, 10, 118,
130);
```



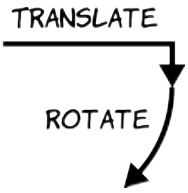
The rotate method takes a value in radians and rotates the drawing context by that angle. As with translate, the values we provide to the drawImage method are now relative to the transformation.

We don't really want the off the canvas like that, so let's translate it then rotate it:

```
var img =
document.getElementById('myimage');
ctx.translate(120,20);
ctx.rotate(Math.PI/4);
ctx.drawImage(img, 10, 10, 118,
130);
```

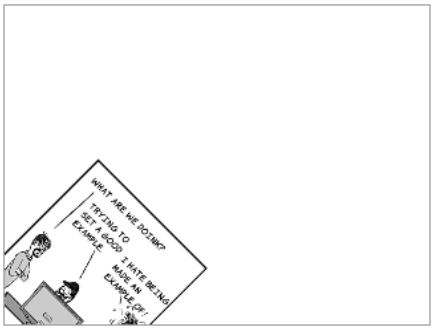


Because the transformations affect the whole context, the order you apply them in is important.

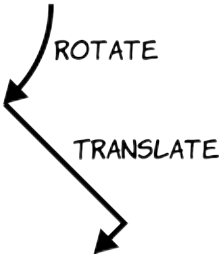


Let's try the opposite order:

```
var img =
document.getElementById('myimage');
ctx.rotate(Math.PI/4);
ctx.translate(120,20);
ctx.drawImage(img, 10, 10, 118,
130);
```



You can see that the rotate now changes the direction the translate goes in.



One potential use of the canvas element that has many developers excited is creating games. Already many arcade classics of the eighties and nineties have been recreated using canvas. In order to create games you need to have animation, let's have a look at how you can animate your canvas drawings.

Above is one of the world's least exciting animations implemented with the canvas element. Pac-man it isn't, but this simple demo is enough to demonstrate the general principles, here's the code used to generate it:

```
function init() {
    draw();                                                    1
    window.setInterval(draw,1000);                             2
}

function draw(){                                               3
    var now = new Date();
    document.getElementById('timestamp').innerHTML
     = now.toLocaleString();
    var canvas = document.getElementById('mycanvas');
    if (canvas.getContext) {
        var ctx = canvas.getContext('2d');
        ctx.clearRect(0,0,320,240);                            4
        ctx.fillStyle = 'rgb(255,0,0)';
        ctx.fillRect(now.getSeconds() * 4,50,100,100);         5
        ctx.beginPath();                                       6
        ctx.strokeStyle = 'rgb(0,127,127)';
        ctx.moveTo(now.getSeconds() * 4,50);
        ctx.lineTo(now.getSeconds() * 4 + 100,150);
        ctx.lineWidth = 5;
        ctx.stroke();
    }
}
```

1 Cueball
2 Cueball
3 Cueball
4 Cueball
5 Cueball
6 Cueball

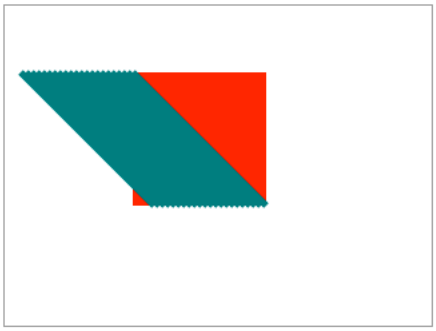
Replace #1, #2 , #3, #4, #5, #6 in the following three paragraphs with a cueball

On line #1 we start our animation by drawing the initial state. We use the draw function #3 for every step of the animation starting with the first one. Then on line #2 set an

interval to call the draw function every second. The progress is keyed against the current time #5 so we have no need to worry about storing additional parameters.

Notice on line #4 the first step is to clear the canvas. The pixels drawn on the canvas persist until they are overwritten by new drawing commands. The canvas is not cleared automatically.

Pixels are not the only thing which persists between calls to our draw function. On line #6 the beginPath method is called. Have a look below to see what happens if you forget to explicitly start a new path.



Mon 21 Jun 2010 15:04:24 BST

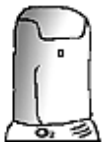
If you forget to close any paths you have open, they will be redrawn as you iterate through your animation steps along with any additions to the path. Clearing the pixels on the context does not reset the path.

[Comic strip here: Pitr has implemented Quake in canvas, Stef thinks he must be able find someone on the internet he can beat, Stef gets beaten by 5 year old girl]

Now you've learned about the canvas element it's time to look at the second technology available in HTML5 for drawing graphics, SVG.

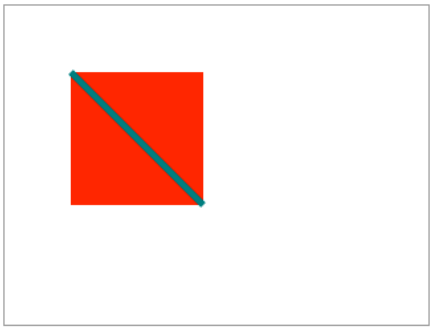
4.2 Getting started with SVG

SVG is an XML language for displaying vector graphics. It has long been possible to embed SVG within XML based XHTML documents but, as HTML5 leads us back to HTML based markup it adds the useful feature that SVG can be embedded directly.



The examples in this section will need Firefox 4.0 to work as is. Although other browsers support SVG they do not yet support direct embedding in HTML as per the HTML5 spec, see 4.4.2 for more information.

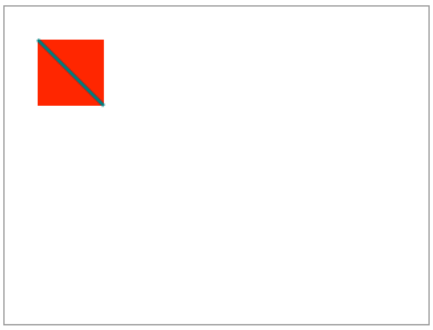
Let's create a simple SVG drawing and see what it looks like:



You're probably thinking this all looks very familiar, and you're right. Many of the things which can be achieved with canvas can also be easily achieved with SVG:

```
<svg id="mysvg" viewBox="0 0 320 240"
        style="outline: 1px solid #999; width: 320px; height: 240px;">
    <rect x="50" y="50" width="100" height="100"
          style="fill: rgb(255,0,0)"></rect>
    <line x1="50" y1="50" x2="150" y2="150"
          style="stroke: rgb(0,127,127); stroke-width: 5;"></line>
</svg>
```

There are several interesting things to be seen in this simple example. First, note that the size of the element on the page is determined by CSS (code annotate 2) but we also define a viewBox with the same values. Because SVG is a vector format pixels aren't as significant, we can use viewBox to define a mapping between the physical dimensions of the element, defined in CSS, and the logical co-ordinates of everything displayed within. Have a look what happens if we use these values: viewBox="0 0 640 480"



This is the same SVG graphic as before, but rendered into a larger viewPort

4.2.1 Applying styles to SVG

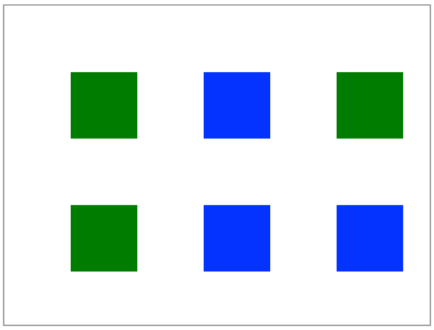
In the previous examples we used an inline style to apply colors and stroke thicknesses. Those properties can also be applied directly to the elements in question, like this:

```
<rect x="50" y="50" width="100" height="100"
      fill="rgb(255,0,0)"></rect>

<line x1="50" y1="50" x2="150" y2="150"
      stroke="rgb(0,127,127)" stroke-width="5"></line>
```

However, we could equally well have left the style and inline attributes off and used this in our CSS file and have the same results:

```
rect {
    fill: rgb(255,0,0);
}

line {
    stroke: rgb(0,127,127);
    stroke-width: 5;
}
```

It looks much like any other CSS, albeit with some unusual properties. As with regular HTML, CSS can make life much easier if you have a lot of similar objects because you can use a class to apply a set of styles to several elements.



Above there are three green squares and three blue squares. Rather than specify inline styles on each one we can declare their commonality with the class attribute:

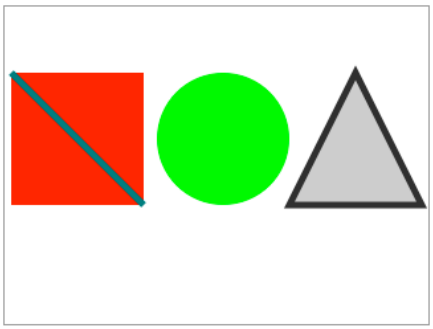
```
<rect x="50" y="50" width="50" height="50" class="green"></rect>
<rect x="150" y="50" width="50" height="50" class="blue"></rect>
<rect x="250" y="50" width="50" height="50" class="green"></rect>
<rect x="50" y="150" width="50" height="50" class="green"></rect>
<rect x="150" y="150" width="50" height="50" class="blue"></rect>
<rect x="250" y="150" width="50" height="50" class="blue"></rect>
```

Then we style the common elements with CSS:

```
rect.green {
    fill: rgb(0,127,0);
}

rect.blue {
    fill: rgb(0,0,255);
}
```

4.2.2 Drawing common shapes

Let's carry on and recreate the rest of our canvas example shapes in SVG. As well as the rectangle and line elements you've seen already, SVG has elements for circles and arbitrary polygons.



For a circle we simply need to provide the x and y co-ordinates of the center and the radius as appropriate attributes. A polygon is slightly more complex, it has an attribute points to which we supply a space separated list of x,y co-ordinates. This is the code which generates the image above:

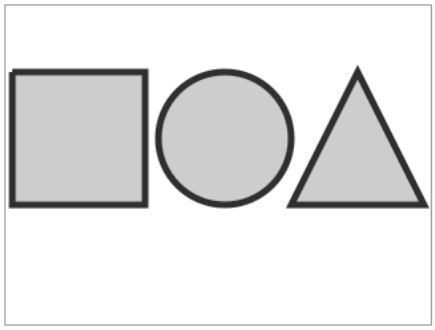
```
<rect x="5" y="50" width="100" height="100"
      style="fill: rgb(255,0,0);"></rect>
<line x1="5" y1="50" x2="105" y2="150"
      style="stroke: rgb(0,127,127); stroke-width: 5;"></line>
<circle cx="165" cy="100" r="50"
        style="fill: rgb(0,255,0);"></circle>
<polygon points="265,50  315,150  215,150"
         style="stroke: rgb(51,51,51); fill: rgb(204,204,204);
                stroke-width: 5;"></polygon>
```

With the polygon element you don't have to provide the starting point a second time, it will assume the shape is closed and the path drawn will return to the first point. If you want to draw an open shape you can use the polyline element instead, it uses an identical points attribute but does not close the path round the shape.

```
<polygon
  points="265,50  315,150  215,150"
  style="stroke: rgb(51,51,51);
         fill: rgb(204,204,204);
         stroke-width: 5;">
</polygon>
```

```
<polyline
  points="265,50  315,150  215,150"
  style="stroke: rgb(51,51,51);
         fill: rgb(204,204,204);
         stroke-width: 5;">
</polyline>
```

You've seen several elements make different shapes in a single SVG drawing, but there is also a way to draw several different shapes in a single SVG element, for this you use the path element. Let's have a look at an example.



Though you can see three shapes in the above image, they are just a single SVG element. The path element in SVG is very powerful, here's the code:

```
<path d="M5,50                                          1
        l0,100 l100,0 l0,-100 l-100,0
        M215,100
        a50,50 0 1 1 -100,0 50,50 0 1 1 100,0
        M265,50
        l50,100 l-100,0 l50,-100                        2
        z"                                              3
    style="stroke: rgb(51,51,51);
           fill: rgb(204,204,204);
           stroke-width: 5;"/>
```

1 Cueball
2 Cueball
3 Cueball

Replace #1, #2 , #3 in the following paragraph with a cueball

The path element works as if it was an imaginary pen, you then use the d attribute to pass a series of commands to the pen to tell it what to draw. For example, line #1 above issues the command 'Move to point 5,50 without drawing' and line #2 begins 'Draw a line from the current point 50 units across and 100 units down'. Each letter is a command,

usually followed by two or more parameters. The commands can be upper case or lower case, upper case commands expect absolute co-ordinates while lower case commands expect co-ordinates relative to the current pen position. At the end of the path #3 the z command closes the path, in much the same way as closePath method did for canvas.

It seems like a path can do anything, why even bother to use anything else?

The path element is difficult to understand and manipulate because of it's reliance on a single attribute value, also any style will apply to all shapes on the same path.

So all my shapes would have the same border and color?

Yes, there's nothing stopping you using more than one path, of course, with a different style applied to each.

Sometimes enlightenment requires following many paths?

...

4.2.3 Adding images, text and embedded content

Images are very easy to embed within your SVG drawing, the syntax is very similar to that of HTML, the only additional information you need to provide over and above the img

element are the co-ordinates of the top left corner:

```
<image x="10" y="10"
       width="236" height="260"
       xlink:href="example.png">
</image>
```

We use an xlink:href to link to the image. The xlink is a namespace, a legacy of SVG's XML heritage which leaks through to HTML5, more on that shortly.

Text is handled a little differently in SVG compared to HTML. In HTML, any text within the body is rendered to the screen, no special wrapping is required, whereas in SVG it has to be explicitly wrapped within an element for containing text.

```
<text x="10" y="20">
    HAI! I CAN HAZ CHEEZBURGER?
</text>
<text x="10" y="60">
    HAI
    CAN HAS STDIO?
    VISIBLE "HAI WORLD!"
    KTHXBYE
</text>
```

HAI! I CAN HAZ CHEEZBURGER?

HAI CAN HAS STDIO? VISIBLE "HAI

The previous example highlights another problem, text which won't fit in the view is not automatically wrapped. Line breaks also have to be explicitly coded using the tspan element:

```
<text x="10" y="20">
    HAI! I CAN HAZ CHEEZBURGER?
</text>
<text x="10" y="60">
    <tspan x="10">HAI</tspan>
    <tspan x="10" dy="20">
        CAN HAS STDIO?
    </tspan>
    <tspan x="10" dy="20">
        VISIBLE "HAI WORLD!"
    </tspan>
```

HAI! I CAN HAZ CHEEZBURGER?

HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE

```
    <tspan x="10" dy="20">
        KTHXBYE
    </tspan>
</text>
```

As with the canvas element, large blocks of text are somewhat cumbersome in SVG, the text elements are only really useful for labels and short descriptions. However, with SVG there is an alternative, we'll have a look at a different method for putting text in an SVG drawing after we've had a quick look at some of the things you can do with the text element.

A nice effect you can achieve on short runs of text is to make the text follow a path. If we extract the circle part of the path from the earlier example we can spread the text along it with the textpath element:


```
<defs>
    <path id="myTextPath"
           d="M215,100
              a50,50 0 1 1
               -100,0 50,50 0 1 1
               100,0">
    </path>
</defs>
<text>
    <textPath
        xlink:href="#myTextPath">
        HAI! I CAN HAZ CHEEZBURGER?
    </textPath>
</text>
```

The path is created in the defs element, then we link to it using an xlink:href like we used for the image earlier. The link works just like other web content, so we could refer to path in a separate file if we wanted to.

You can also apply gradient fills and any number of other SVG effects to the text, we'll cover this in detail in the next section, but here is a quick example which shows a gradient, from transparent light green to solid dark green, applied as a fill to slightly larger version of our circular text.

[cartoon – why can't I just use regular html elements within the SVG?? well, that brings me on to foreign object]

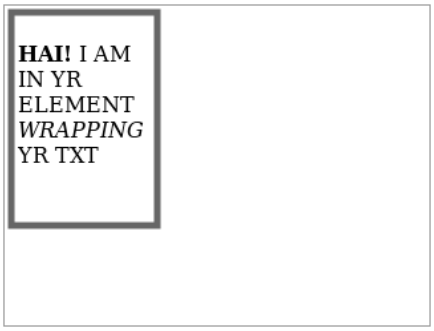


Figure 4.x Using the foreignObject element to embed HTML within SVG

```
<rect x="5" y="5" width="110" height="160"
      style="stroke-width: 5; stroke: rgb(102,102,102); fill: none;">
</rect>
<foreignObject x="10" y="10" width="100" height="150">
    <body>
      <p>
        <strong>HAI!</strong> I AM IN YR ELEMENT <em>WRAPPING</em> YR TXT
      </p>
    </body>
</foreignObject>
```

The inside the foreignObject element is HTML, and, unlike the SVG text element, HTML can cope with wrapping text just fine. It's important to remember that the browser is not rendering the contents of foreignObject as if they were HTML, the content inside is really HTML and can still be interacted with in the normal way. A second example will make this more clear.

On the right is a screenshot of the whole browser window, on the left it is zoomed in to just the content of the foreignObject element. The Google home page has been scaled down and rendered upside down inside the browser, but it is still possible to type search terms in and see results returned (even if they're too small to read!). This was achieved by wrapping an HTML document inside a foreignObject element in SVG, then applying some transforms:

```
<svg id="mysvg" viewBox="0 0 800 600">
    <g transform="rotate(180) translate(-800,-600)">
        <foreignObject x="10" y="10" width="800" height="600">
            <body>
                <iframe src="http://www.google.co.uk/"
                    style="width:780px;height:580px">
                </iframe>
            </body>
        </foreignObject>
    </g>
</svg>
```

This example shows a few things you've seen before – the viewBox is set to 800 by 600 pixels, even though our element is 320 by 240 pixels, this takes care of the scaling, and an iframe is used inside the foreignObject to fetch the Google page. New in this example are the g element for grouping SVG content and the transform attribute, both of which we'll look at in the next section.

4.2.4 Applying transforms, gradients, filters and declarative animation

SVG is a huge topic, worthy of a book by itself, so we're barely scratching the surface so far. In this section we'll finish of by taking a very quick look at some of the more advanced features.

When we want to apply a transform to a collection of elements we use the grouping element, g. Grouping is also useful for other purposes, for example if you wanted to move several elements at the same time.

You've just seen a transform in action in the last example of the previous section. Here it is again:

```
<g transform="rotate(180) translate(-800,-600)">
```

The transform attribute accepts a space separated list of commands which are applied in order. The element is rotated 180 degrees and then, because the rotation point by default is the top left corner, it is moved back into view with the translate transform. We could instead have passed a set of co-ordinates to the rotate transform and achieved the same result in a single step:

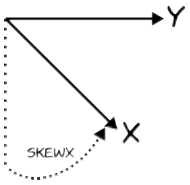
```
<g transform="rotate(180,400,300)">
```

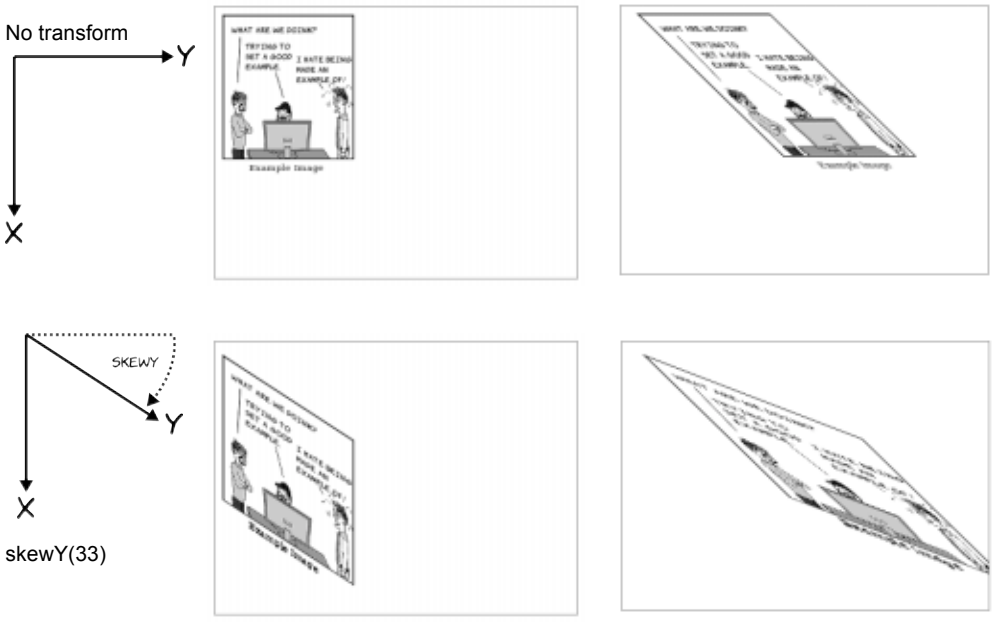
Aside from rotate and translate there are several other transformation commands:

- scale – you've seen examples of scaling already, in earlier examples we scaled the whole viewBox, this command allows you to control it for specific elements
- skewX and skewY – see the diagram below
- matrix – matrix is a very powerful transformation which allows you to emulate all the others in combination, if you understand the mathematics of matrix transformations. If, like me, you missed that particular part of the curriculum it's easiest to stick to the other ones

No transform

skewX(45)



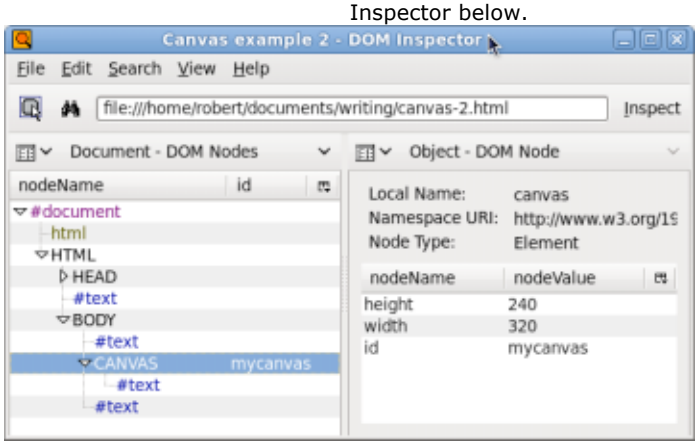


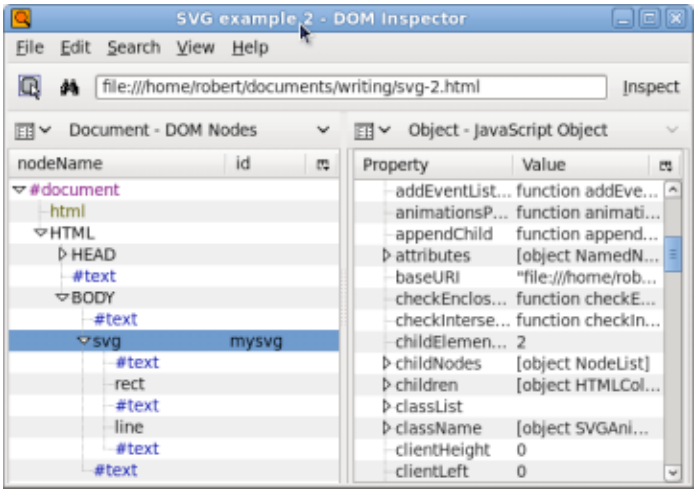
The transformation functions for SVG and Canvas look very similar, and they are. The main difference from a developer perspective is that SVG transformations expect angles in degrees while Canvas transformations expect angles in radians.

4.3 SVG vs Canvas

I am likink the canvas element, clean API makink happy developer.

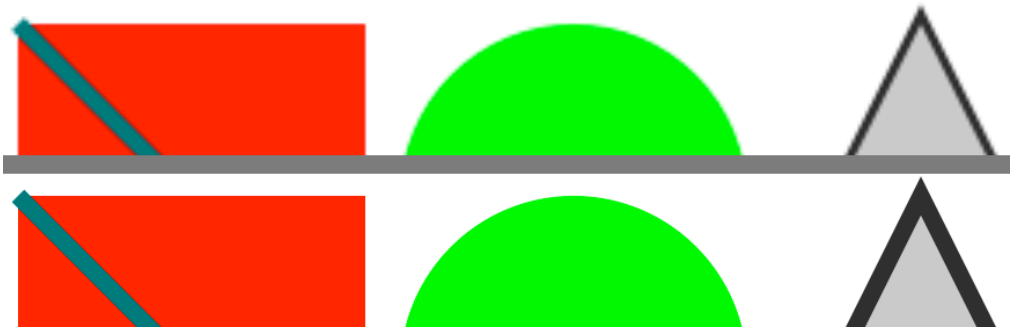
SVG has an API too, but it's through the browser DOM, which doubles as a persistent object model. With canvas you have to manage your own objects and it has no internal structure as you can see by comparing the two screenshots of DOM Inspector below.





Is true. But browser DOM is beink too heavyweight. Is good that I am choosing own object model in canvas when mappink several thousands of minions.

Well, there are issues once you hit a certain threshold in number of objects, but SVG has other advantages. The two screenshots below show the effect of zooming in eight times on a canvas element compared to and SVG element underneath.



Ya. Natural vectors is beink nice feature. But beink practical, can just be resizink and redrawink canvas element to be matchink page dimensions.

True, but why make extra work for yourself? Integrating other web content is straightforward in SVG. Also, the declarative style of SVG may be more comfortable for web authors who's strengths lie in HTML and CSS.

Have been discussink this before, am not believing markup monkeys is being the same thing as real developers.

Not all web authors need to be hardcore developers. There are other benefits to the object model and declarative markup – SVG will be much easier to make accessible.

Is true, but again beink practical, no browser is supportink accessibility features in SVG yet.

Sounds like a good time to look at browser support for Canvas and SVG.

4.4 Canvas and SVG in browsers

Both canvas and SVG have wide support in current browsers, with prospects for even better support in the respective next releases. Canvas support tends to be all or nothing but the situation with SVG is a lot more complex, more on this in the coming sections.

Key	Complete or nearly complete support:	Incomplete or partial support:	Little or no support:

Browser	Canvas	Canvas Text	SVG Score	SVG as Image	SVG in CSS	SVG as Object	SVG in XHTML	SVG in HTML
Chrome 5	✓	✓	88%	✓	✗	✓	✓	
Chrome 6	✓	✓	~	✓	✗	✓	✓	

Browser	Canvas	Canvas Text	SVG Score	SVG as Image	SVG in CSS	SVG as Object	SVG in XHTML	SVG in HTML
Safari 4	✓	✓	83%	✓	✗	✓	✓	

Hardly surprising, Apple invented the canvas element after all, but WebKit support for this stuff rocks.

WebKit has not yet implemented a full HTML5 parser, so embedding SVG in HTML is not yet possible. Also it's disappointing to note that Google didn't enable any SVG support in the Android version of the Chromw browser.

Firefox 3.6	✓	✓	62%				✓	✗
Firefox 4	✓	✓	78%	✓	✓	✓	✓	✓

Firefox will be the first major browser to be release with an HTML5 parser, this more than makes up for it's failure to support some of the more esoteric features of SVG.

The lack of support for SVG as an image or used from CSS has finally been rectified in the Firefox 4 release. Fortunately for us web authors Firefox users tend to upgrade promptly.

IE 8			0%					

Browser	Canvas	Canvas Text	SVG Score	SVG as Image	SVG in CSS	SVG as Object	SVG in XHTML	SVG in HTML
IE 9			31%					

This is a conversation between Stef and AJ

Stef: I'm starting to think there's some bias in these browser support tables you show me – IE never seems to be doing very well.

AJ: SVG is partly based on an old Microsoft technology called VML which came out in 1998

Stef: So in fact Microsoft has been leading the way for over a decade and it's only now everyone else is catching up? I knew these tables were just propoganda!

AJ: I don't think that's quite what I said...

Stef: You know the truth!

Opera 10.5			96%					

As per usual, Opera's standards support rocks! Opera embraced SVG early on because of it's potential as an ideal image format for mobile devices, which vary in screen resolution far more than desktops.

Although their support for SVG is excellent, Opera only added support for the Canvas element in version 10.5. In practical terms this isn't usually a problem as Opera users upgrade frequently.

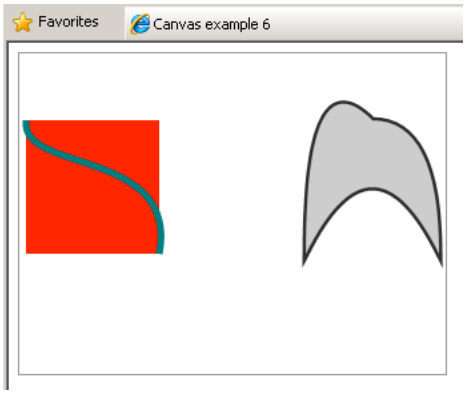
The table above is split into rows which indicate the support in various versions of browsers, each then followed by a 'colorful' comment by the browser's character advocate and then a 'factual' comment from AJ (apart from the IE section, see other note).

4.4.1 Supporting canvas in older versions of IE with explorercanvas

Internet Explorer is the only major browser which has no support for the canvas element, and currently, unlike SVG, there are no plans for support to be added in IE9. Since IE still represents about 50% of users this is a bit of an issue if you want to use canvas.

However all is not lost, as Internet Explorer uniquely has support for Vector Markup Language (VML). VML is a predecessor of SVG, and we've already seen that SVG and Canvas can do a lot of similar things. The explorercanvas library implements Canvas in IE using VML. Activating explorercanvas is as simple as including a script element in the head of your HTML document:

```
<head>
<!--[if IE]><script src="excanvas.js"></script><![endif]-->
</head>
```



4.4.2 SVG in XML vs SVG in HTML

We mentioned earlier that SVG support is not as clear cut as Canvas support. This is not just because the SVG specification is more complex but because there's more ways of making use of SVG from within a web page, and this is largely because SVG was originally envisioned as one of a family of XML based languages which would be used for web content.

In nearly all of the major browsers it has long been possible to embed SVG content in the XML version of HTML – XHTML. Unfortunately there has been one major obstacle to this happening.

Fully compliant XHTML should be delivered from the server as XML content. The server tells the browser the content type of the file in the header of the HTTP response. Unfortunately, if you try and send XML web page to any version of Internet Explorer it refuses to parse it. Since deploying SVG in XHTML requires breaking IE, few people have considered it practical.

4.4.3 Embedding SVG as an image

SVG can be used in the img element in the same way as any other image format:

```
<img src="svg-2.svg">
```

When used in this way, SVG still has the advantage of being scalable; you can set it to take up half the browser window and it will remain sharp no matter how high or low your user's screen resolution. But you lose the advantage of being able to manipulate the image from the JavaScript – the elements of the image are not present in the DOM.

4.4.4 Reference an SVG image from CSS

In the same way as it can be used as an image in HTML, SVG can be referenced as an image in CSS:

```
div {
    background: url(svg-2.svg) top right no-repeat;
}
```

4.4.5 Embedding SVG as an object

The object element is a general purpose method to embed any external content in your web page. To embed SVG with object you need to supply two parameters specifying the file name and the file type:

```
<object type="image/svg+xml" data="svg-2.svg"></object>
```

In browsers with native support for SVG the object embedding method has similar results to including the SVG inline: the SVG elements are available in the DOM and can be manipulated. It has the benefits that it works in every browser which has SVG support and also that, if you are using the same SVG image on different pages if your site it will be cached in the same way as a normal image would be, making your site slightly faster to load. The corollary of this, of course, is that if you only use the image once it will require a second request to the server, making your site slightly slower to load.

4.4.6 SVG Support in older browsers with SVGWeb and Raphaël

You do not have to rely on direct browser support for SVG, in older browsers and Internet Explorer there are a couple of JavaScript libraries which enable SVG support through alternative means.

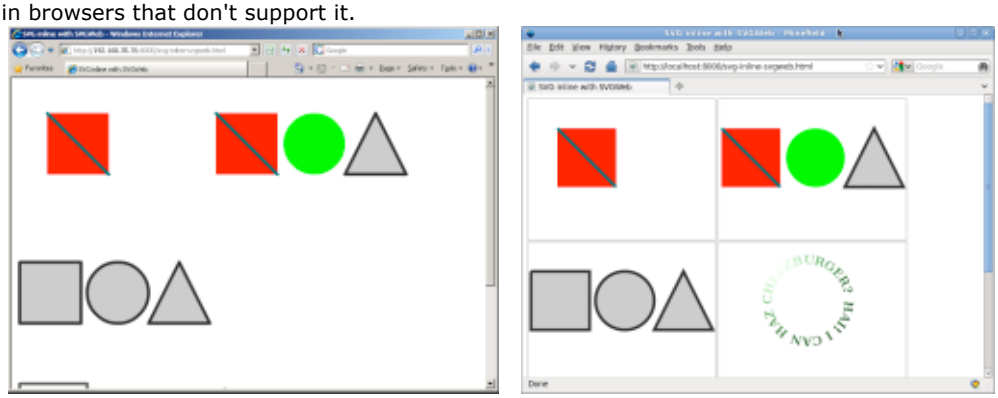
SVGWeb is a JavaScript library which, if it detects the browser has no native support for SVG, it will replace any SVG graphics it finds with a Flash movie. The Flash movie will then take care of rendering the SVG in the browser. You have to make some slight modifications to your web page in order to enable SVG web. The first is in the head of the document, where you reference the SVGWeb JavaScript library:

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

The you have to surround each of your SVG graphics with script tags:

```
<script type="image/svg+xml">
    <svg viewBox="0 0 320 240">
        <rect x="50" y="50" width="100" height="100"
              style="fill: rgb(255,0,0)"></rect>
        <line x1="50" y1="50" x2="150" y2="150"
              style="stroke: rgb(0,127,127); stroke-width: 5;"></line>
    </svg>
</script>
```

Your SVG graphics will then render as SVG in browsers that support it, and Flash movies in browsers that don't support it.



On the left you can see that SVGWeb allows Internet Explorer to render inline SVG, though it doesn't match the native support offered by browsers such as Firefox, on the right.

The Raphaël JavaScript library takes a very different approach. Instead of making existing SVG work in Internet Explorer, it presents an API for creating graphics. In Firefox, Chrome, Safari and Opera it creates SVG, in Internet Explorer it creates VML. The interface Raphaël provides looks very similar to the canvas API:

```
var paper = Raphael(10, 50, 320, 200);
var circle = paper.circle(50, 40, 10);
circle.attr("fill", "#f00");
circle.attr("stroke", "#fff");
```

Raphaël looks similar to canvas, but it is still SVG underneath. This means when you call the circle function it returns an object. This object can later be modified and the drawing will update to reflect the changes, you don not have to clear everything and redraw it like you do with canvas.

4.5 Summary

In this chapter you've learned how you can generate graphics in your web page on the fly using two different HTML5 technologies – Canvas and SVG. Because both can be created and updated dynamically they do not need the user to reload the page in order to present new information to the user.

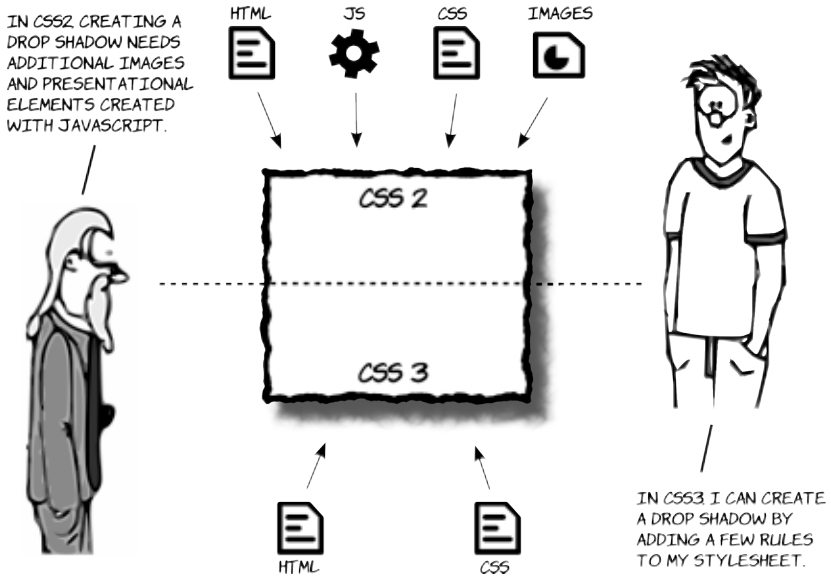
You've learned the basic techniques for drawing shapes and lines with both technologies, as well as how to import images and apply effects and transformations. With canvas you've looked at how to do simple animation while with SVG you saw how we can import whole web pages and applied transformations to them.

Now that you can create your own graphics on the fly it's time to complete your education on the multimedia possibilities of HTML5 with a look at the new audio and video elements.

10

Borders and backgrounds with CSS3

CSS3 makes the traditional background image approaches more flexible and provides declarative options for drop shadows, rounded corners and gradients. Solutions which have involved images, JavaScript and extra markup can be replaced with just simple HTML and CSS.



10.1 Drop shadows with CSS3

The pseudo 3D effect provided by drop shadows is a very popular design approach. In the past designers have gone to great lengths to add this visual effect, but CSS3 saves a lot of time and resources by having the functionality built in.

CSS3 defines two types of shadow – box and text. They use a similar syntax:

```
text-shadow: rgb(0,0,0) 3px 3px 3px;
box-shadow: rgb(0,0,0) 3px 3px 3px;
```

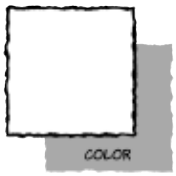
A basic shadow, in either case, is defined by four values:

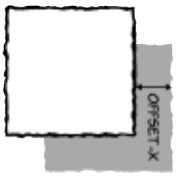
```
<color> <offset-x> <offset-y> <blur-radius>
```

In the next section we'll take a look at what each of the four values used to define a shadow does.

10.1.1 Box Shadows

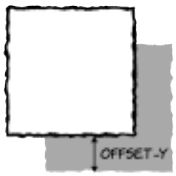
color is any valid CSS colour value, such as #6699cc, rgb(102,153,204) or rgba(102,153,204,255)

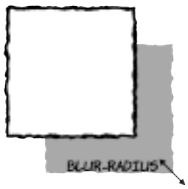




offset-x is a CSS length, such as 3px, 0.5em, negative values are allowed

offset-y is also a CSS length, again negative values are allowed





blur-radius is also a CSS length, negative values are not allowed but this value is optional

Browser quick check

	Standard	Prefixed
IE	9	-
Firefox	4.0	3.5
Chrome	10	5
Safari	-	3
Opera	10.5	-

Please place above table in sidebar

10.1.2 Simple Box Shadows

Without the optional blur-radius, box shadow is not much different from a border. Here's an example that only sets an offset-x:

```
box-shadow:
 rgb(0,0,0) 12px 0px;
```

And here's a plain offset-y:

```
box-shadow:
 rgb(0,0,0) 0px 12px;
```

Adding a blur-radius by itself creates a more shadow like effect, even without any offsets:

```
box-shadow:
 rgb(0,0,0) 0px 0px 12px;
```

By combining the blur-radius with the offsets you can set the apparent light source:

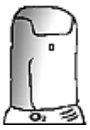
```
box-shadow:
 rgb(0,0,0) 0px 12px 12px;
```

10.1.3 Complex Box Shadows

For complex effects, you can add multiple shadows in a comma separated list, they can all use different colors and directions. In the example below there is a red/orange shadow down and to the right and a purple shadow up and to the left. Whether or not this is a good


```
box-shadow:rgb(255,0,0) 3px 3px 3px,
                rgb(255,102,0) 6px 6px 6px,
                rgb(255,204,0) 9px 9px 9px,
                rgb(255,0,204) -3px -3px 6px;
```

idea is up to you!

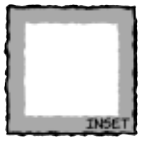


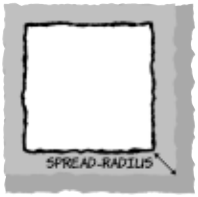
The examples in the next section will only work in Firefox and Opera, though you may need to add the vendor prefix (see chapter...) for your browser depending on which version you have.

In the full box-shadow definition there are two additional, optional elements:

<inset> <color> <offset-x> <offset-y> <blur-radius> **<spread-radius>**

inset, if present, puts the shadow inside the element instead of outside.





spread-radius is a CSS length, it causes the shadow to grow (positive values) or shrink (negative values) relative to the size of the element.

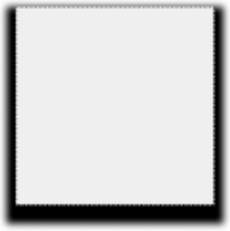
Here the shadow is grown by six pixels on every side:

```
box-shadow: rgb(0,0,0)
 0px 0px 12px 6px;
```

This creates a shadow which extends evenly all around the element.

This example shows a spread-radius combined with an offset-y:

```
box-shadow: rgb(0,0,0)
 0px 12px 12px 12px;
```



Using inset you can achieve some bevel-like effects:

```
box-shadow: inset rgb(0,0,0)
 0px 0px 12px 6px;
```

Or you can make an element appear to 'drop' into the page, perhaps by using the :hover pseudo class.

```
div:hover {
 box-shadow:
  inset rgb(0,0,0)
  3px 3px 5px;
}
```

10.1.4 Text Shadows

Text shadows work in exactly the same way as box shadows, they are defined by the same four values:

```
<color> <offset-x> <offset-y> <blur-radius>
```

These work in the exact same way as the values for box-shadow, so we won't look at them again, instead let's look at some examples.

Browser quick check

	Standard	Prefixed
IE	-	-
Firefox	3.5	-
Chrome	5	-
Safari	4	-
Opera	10	-

Please place above table in sidebar

A simple offset-x:

```
text-shadow:
 rgb(51,51,51) 6px 0px;
```

And an offset-y:

```
text-shadow:
 rgb(51,51,51) 0px 6px;
```

As with box shadows, things become more interesting when blur-radius is invoked:

```
text-shadow:
 rgb(51,51,51) 0px 0px 6px;
```

And combining that with an offset can create a 3D feel to match your box shadows:

```
text-shadow:
 rgb(51,51,51) 0px 6px 6px;
```

Multiple shadows work as well:

```
text-shadow:
 rgb(51,0,0) 6px 0px,
 rgb(0,0,51) 0px 6px;
```

But be careful, it is easy to create completely unreadable text.

```
text-shadow:
 rgb(51,0,0) 6px 0px 3px,
 rgb(0,0,51) 0px 6px 3px;
```

In most cases you will want to keep your text shadows small and subtle.

You can use negative values for the offset-x and offset-y, this allows for some interesting pseudo-3D effects if you set the text color to be the same as the background color:

```
color: rgb(255,255,255);
text-shadow:
 rgb(51,51,51) -1px -1px;
```

You've now learned how to achieve a number of useful effects with border-image, including beveled edges and rounded corners, by slicing up an image across a background. But rounded corners are such a useful effect that CSS3 provides a way to make them without using an image, let's have a look at that next.

10.2 Easy Rounded Corners

Perhaps even more common than the drop shadow in modern web design is the rounded corner. Even otherwise very simple website designs often make use of rounded corners in their design.

The lengths web authors have to go to get the rounded corner 'look' in CSS2 is extraordinary – one popular method involves adding hundreds of elements to a page just to get rounded corners on a single element, we'll look at that below.

Many of the rounded corners you see on the web are not strictly necessary. An engineer at Yahoo! once created a version of their home page without any rounded corners and discovered it reduced the amount of data a user had to download by more than 50%. When he showed the two different versions of the page to a designer they didn't spot the difference.

Browser quick check

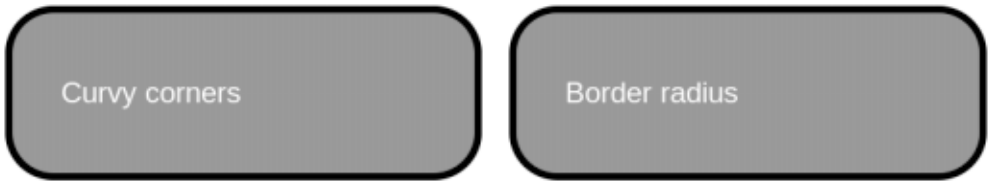
	Standard	Prefixed
IE	9	-
Firefox	4.0	3.0
Chrome	7	5
Safari	5	3
Opera	10.5	-

Please place above table in sidebar

CurvyCorners is a JavaScript library for creating rounded corners on elements, let's see how it compares to the new CSS3 techniques for creating rounded corners:

OK, I'm going to create an element with rounded corners using the CurvyCorners JavaScript library.

Meanwhile I'll create an identical looking element using CSS3.



So far it doesn't look any different whichever approach you take.

I agree, let's compare the code we had to write to get the rounded corner effect.

```
<style>
#myBox {
    background-color: #999;
    border: 6px solid #000;
}
</style>
<script
 src="curvycorners.js"></script>
<script>
```

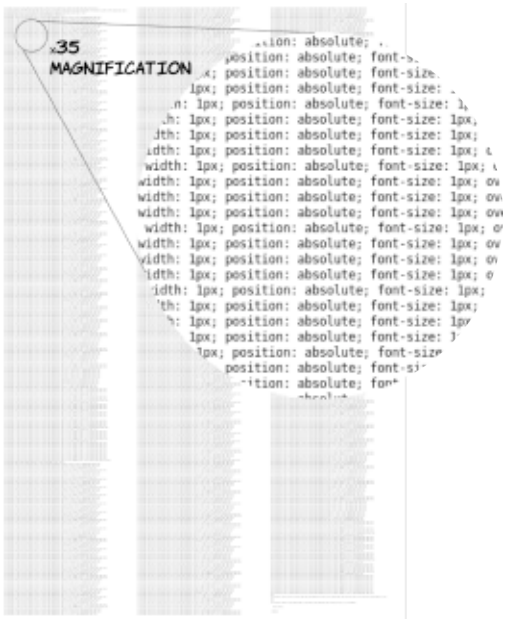
```
<style>
#myBox {
    background-color: #999;
    border: 6px solid #000;
    border-radius: 40px;
}
</style>
```

```
  addEvent(window,
     'load', initCorners);
  function initCorners() {
    var settings = {
      tl: { radius: 40 },
      tr: { radius: 40 },
      bl: { radius: 40 },
      br: { radius: 40 },
      antiAlias: true
    }
</script>
```

It's starting to look like the CSS approach is the winner. Even though the CurvyCorners code isn't complex there's twice as much of it. However, I think you've cheated – to get the widest browser support you need to add some browser specific rules to your code.

That's true, there would be four extra lines in my CSS if this was production code, I think I'd still be winning though!

This isn't a competition AJ, we're collaborating to try and find the best approach! Let's have a look at the client side markup for each.



Er, OK, there's a little extra markup involved with CurvyCorners, but it's not like I had to write all that HTML myself. Although it was a bit of a pain having to attach a script it wasn't that much effort on my part.

Still, there'll be a performance impact on the browser when scrolling or resizing just due to having thousands of extra elements in the DOM. With the CSS3 approach the browser takes care of it all in the background and may even hand off the processing to graphics hardware rather than render shadows itself.

On the other hand, the CurvyCorners approach will work on nearly every browser out there, including old versions of Internet Explorer. CSS3 isn't supported in every browser yet, and web authors will be dealing with old versions of some browsers for years.

That's true, but you can easily detect support for border-radius with JavaScript and only run the CurvyCorners script on browsers without support. Browsers that support border radius will get faster and more lightweight pages, everyone else will not see anything amiss.

As well as the single value form you saw above, it is also possible to have different values for each corner of an element:

Like the border, padding and margin properties, the border-radius property can accept up to four values. These apply starting from the top left corner and proceeding clockwise round the element:

```
border-radius: 40px 160px 80px 120px;
```

Later in this chapter you're going to learn about at several features in CSS3 which provide native support for effects web authors have previously tried to achieve with background images, but there are also new features for background images themselves. We'll look at these in the next section.

10.3 New features for background images

CSS3 offers four new features for the venerable background image: sizing; multiple backgrounds on a single element; positioning relative the border, padding or content; and clipping according the the border, padding or content. In this section you will learn about each of these new features.

Browser quick check

	Standard	Prefixed
IE	9	-
Firefox	4.0	3.6

Chrome	7	5
Safari	5	3
Opera	10.5	10.1

Please place above table in sidebar

10.3.1 Background size

Background images are intended to be used purely for decoration, whereas images placed in the HTML are supposed to mean something – this is the same separation of concerns principal that's been discussed in several previous chapters. However images placed in the markup have certain practical advantages which can discourage web authors from doing the right thing.

One of these advantages of images in markup is that it's easy to make an inline image scale according to the size of its container; simply set the width of the image to be 100%. But in CSS2 there is no way to make the image be anything other than its innate size. The following example shows the issue.

In this example an image has been set as a decorative background to a header:

```
h1 {
    background-image:
      url('head-banner.png');
    background-repeat: no-repeat;
    padding-top: 1.85em;
}
```

Top padding has been set to allow room for the background image, and the image itself is sized to match the width of the heading.

If the heading was to change at all though. the background image might look a little incongruous. If we translate 'Columbia Internet' into French, suddenly we have an image with some text sticking out underneath, the visual relationship is lost.

The new `background-size` property allows the image to be stretched in proportion to the dimensions of the element:


```
h1 {
    background-image:
      url('head-banner.png');
    background-repeat: no-repeat;
    background-size: 100% 1.85em;
    padding-top: 1.85em;
    display: inline-block;
}
```

The h1 is set to be inline-block so that its width will 'shrink wrap' it's content, then the background image is set to be a fixed height and full width. You can see that the image becomes distorted as the width of the element forces it to stretch. For this reason this approach is only suitable to allow for small changes in expected element width.

If the text will stay the same but will appear in different font sizes it is possible to avoid the aspect ratio issue. If you know how wide the text will be; simply specify the width and height in proportion to the aspect ratio:


```
h1 {
    background-image:
      url('head-banner.png');
    background-repeat: no-repeat;
    padding-top: 2.18em;
    display: inline-block;
    background-size: 11.1em 2.18em;
}
section:nth-of-type(1) h1 {
    font-size: 200%;
}
section:nth-of-type(2) h1 {
    font-size: 250%;
}
section:nth-of-type(3) h1 {
    font-size: 300%;
}
```

The ability to size backgrounds aligns well with vector graphics. In chapter 4 you learned that SVG graphics, because they are vectors, are smooth and sharp however much you stretch them, whereas bitmap graphics become blocky and blurry. Below are two examples,

one using a bitmap PNG format, the other SVG.


```
div {
    background-image: url('aj.png');
    background-repeat: no-repeat;
    background-size: 50% 100%;
    padding-left: 50%;
    display: inline-block;
}
```

```
div {
    background-image: url('aj.svg');
    background-repeat: no-repeat;
    background-size: 50% 100%;
    padding-left: 50%;
    display: inline-block;
}
```

Although SVG works well with background sizing, it is a bad fit for 'photographic' type images. You may want to set a single picture behind your content, similar to the effect achieved by setting a desktop background on your computer. In this case the detail of the image is less important so distortion is less of an issue.

This example shows the main content with a semi-transparent background overlaid over a background image set to fill the entire box. Here is the markup:

```
<section>
  <div>
    <p>In almost every...</p>
  </div>
</section>
```

The CSS for this is shown in the listing below. Unlike the previous examples this uses the shorthand syntax. The size appears with the position separated by a slash: `top / 100%`.

```
section {
    margin: 1em;
    padding: 5%;
    outline: 4px dashed black;
    background: url('10years.jpg') top / 100% no-repeat;
    display: inline-block;
    min-height: 342px;
    min-width: 300px;
}
div {
    background-color: rgba(255,255,255,0.66);
}
```

Miranda

Note that although several browsers have implemented background-size only Opera has implemented the shorthand syntax. For other browsers you will have to stick with a separate `background-size` declaration.

Scaling is not the only new backgrounds feature added in CSS3, you also are allowed to attach multiple backgrounds to a single element. In the last example above an element was added to the markup who's only role was to add a background to the text. In the next section you'll see how CSS3 allows you to do without additional markup for styling.

10.3.2 Multiple Backgrounds

In CSS2 you are only allowed one background image per element, but there are many situations where you might want more than one image:

- A header might have a background image spanning the width of the page as well as a company logo
- A decorative pullquote box could have opening and closing quotes on either side
- Bevelled buttons or tabs have images for the left and right sides
- Creating a rough edged 'paper scroll' effect needs a repeating image down both sides

Often web authors will use CSS tricks to size a child element to match its container so that their background images can overlap (this is known as the 'sliding doors' technique), but often they will be forced to introduce an extra element just to support the styling, or even add a purely decorative image inline in the markup.

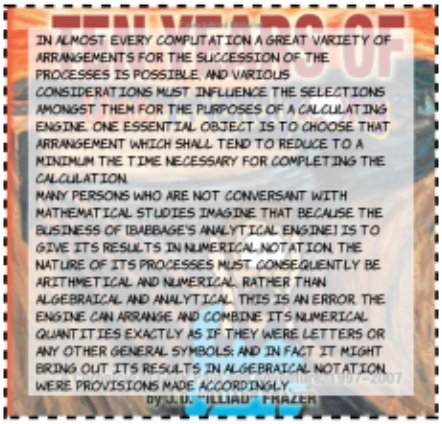
All these approaches end up adding purely presentational markup to the page or depending on elements being combined in a particular way. While this is usually not a big issue in these isolated instances, they indicate a lack of power in the presentation language, CSS. This lack of power is addressed in CSS3 as you'll see in the examples below.

Browser quick check

	Standard	Prefixed
IE	9	-
Firefox	3.6	-
Chrome	7	-
Safari	3	-
Opera	10.5	-

Please place above table in sidebar

Let's revisit the last example from the previous section, except this time without the additional `div` element for wrapping the content:




```
<section>
  <p>In almost every...</p>
</section>
```

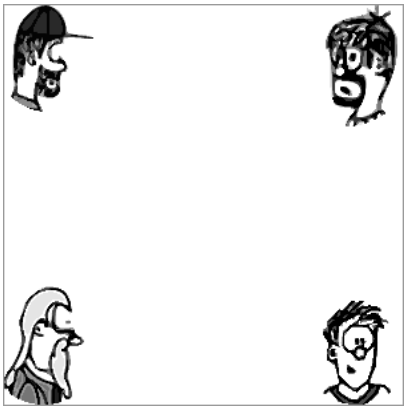
Despite losing the extra element the page still looks the same, this is because two backgrounds are applied to the `section` element:

```
section {
    margin: 1em;
    padding: 5%;
    outline: 4px dashed black;
    background: url('trans-66.png') 50% 50% no-repeat,
                url('10years.jpg') no-repeat;
    background-size: 90% 90%,
                     100%;
    display: inline-block;
```

```
}
```

Adding multiple background images is simply a matter of listing them, along with any relevant attributes, separated by commas in the background property:

```
background: top right
  url('pitr-head.png') no-repeat,
 bottom right
  url('aj-head.png') no-repeat,
 top left
  url('mike-head.png') no-repeat,
 bottom left
  url('sid-head.png') no-repeat;
```



All the other background properties also allow a comma separated list, so you could also write the above as:

```
background-image:
 url('pitr-head.png'), url('pitr-
head.png'),
 url('mike-head.png'), url('sid-
head.png');
background-position:
 top right, bottom right,
 top left, bottom left;
background-repeat:
 no-repeat, no-repeat,
 no-repeat, no-repeat;
```

The background image you list first will be the 'closest' to the viewer. If you put them all in the same place you can see the first image covers the rest:

```
background: center
  url('pitr-head.png') no-repeat,
 center
  url('aj-head.png') no-repeat,
 center
  url('mike-head.png') no-repeat,
 center
  url('sid-head.png') no-repeat;
```

You can use this to your advantage to create some interesting effects. Here we use a semi-transparent PNG image in between each of the other background images to create a progressive fading:

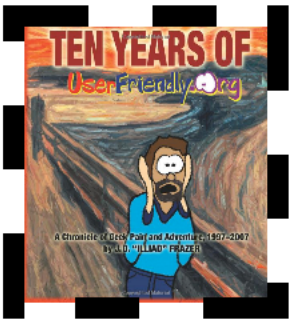
```
background: top right
  url('pitr-head.png') no-repeat,
 url('half-white.png'),
 bottom right
  url('aj-head.png') no-repeat,
 url('half-white.png'),
 top left
  url('mike-head.png') no-repeat,
 url('half-white.png'),
 bottom left
  url('sid-head.png') no-repeat;
```

10.3.3 Background origin and clipping

CSS2 has no control over what part of an element the background applies to. Since CSS2 doesn't allow background sizing most authors will have not come up against this limitation, but CSS3 introduces two new properties to give web authors fine grained control: `background-origin` and `background-clip`.

Erwin The next section will require an understanding of the CSS box model to get the most out of it. If you are not sure, please refer to the discussion in Appendix C or the diagrams in Chapter 8 before proceeding.

The default for background-origin is padding-box, this means the background applies to the area containing the padding but not to the are containing the border:

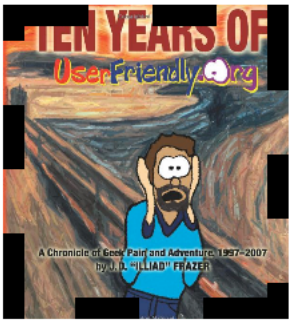



```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-origin: padding-box;
}
```

Remember this example image is scaled to fill its container and set to not repeat.

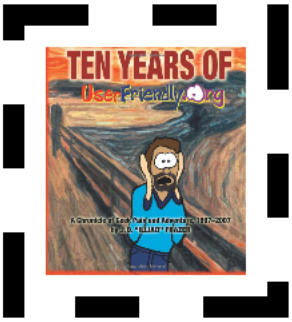
Setting the origin to `border-box` means that the background will now extend out under the border:

```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-origin: border-box;
}
```



Finally `content-box` limits the background to just the content area, inside the padding:

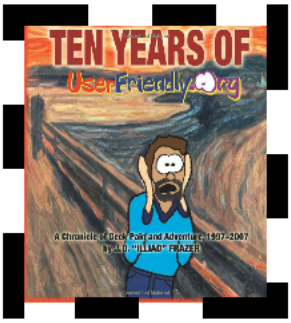
```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-origin: content-box;
}
```



The default value for `background-clip` is `border-box`:

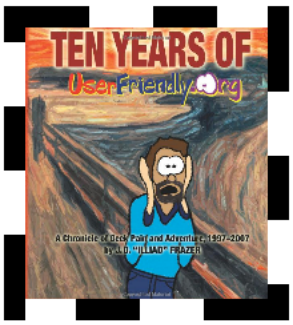
```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-clip: border-box;
}
```

Remember that our example is scaled and set to not repeat.



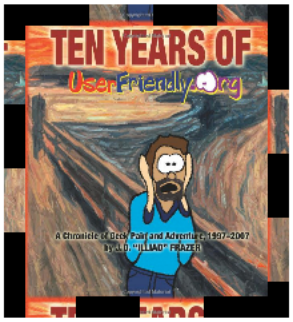
When applied to backgrounds which don't repeat this is indistinguishable from `padding-box`, because of the default value of `background-origin`:

```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-clip: padding-box;
}
```



However if the background is allowed to repeat the difference becomes apparent. A setting of `padding-box` will clip the image inside the border still, but for `border-box` we can see the repeating image under the border:

```
section {
    background-clip: border-box;
    background-repeat: repeat;
}
```



Finally, `content-box` will clip the background to the content area:

```
section {
    margin: 1em;
    padding: 1em;
    border: 1em dashed black;
    background-clip: content-box;
}
```

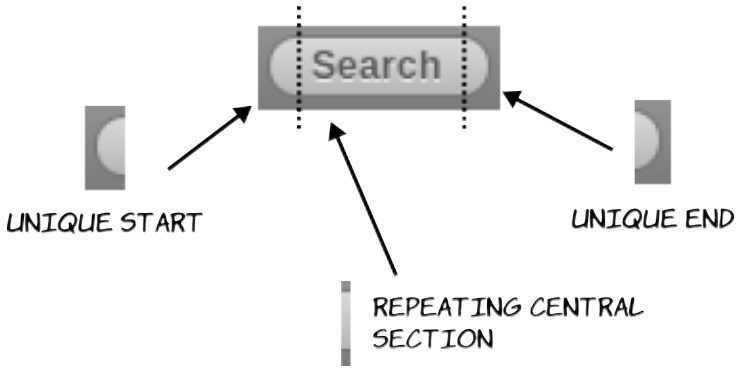
Note that even though the background is clipped, the image is still sized to the `padding-box`.

Dust puppy Scaling backgrounds uniformly may not always produce the effect you want. Although the sliding doors technique provides a work-around for that,

there is a more straightforward CSS3 approach to achieve the same effect: border-image. Let's look at that next.

10.4 Selective background scaling with border images

When you're trying to create flexible layouts you often want a background which looks the same for most of its length, but with a certain number of pixels at either end to be slightly different. This is especially true when you want to create a rounded element with a bevelled effect.



Here are some examples:

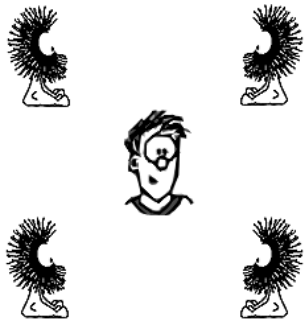
Rounded corners on a bevelled background from bbc.co.uk

Butt Log In rounded corn Search shading on wordpress.com.

Tabs with rounded corners and shading on yahoo.com

The border-image property. The border-image property allows you to 'slice up' an image and apply transformations selectively to each slice. It's more simple than it sounds, as you'll see after a few examples.

The ability to add an image to a border is one of the more powerful features of CSS3, unfortunately it's also one of the least intuitive. Despite being supported by Firefox since version 3.0 it's seen far less uptake than several other features in this chapter.



To start with we'll use the example image above. This image is 240 pixels square, and contains five smaller images each approximately 80 pixels square.

Browser quick check

	Standard	Prefixed
IE	-	-
Firefox	-	3.5
Chrome	-	7
Safari	-	3
Opera	-	10.5

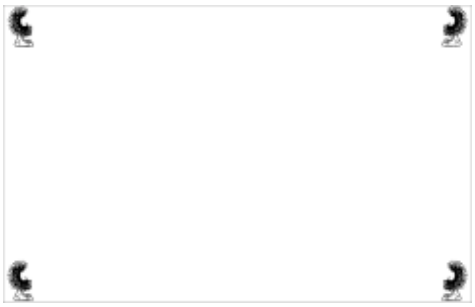
Please place above table in sidebar

Let's start with the simplest possible example. We'll apply our border image to an element 720 pixels wide and 400 pixels high:

```
height: 400px;
width: 720px;
border-width: 80px;
border-image: url('border1.png') 80;
```

As you can see, the center disappears entirely. It may be you want to retain the

center of the image, in which case use the fill keyword:

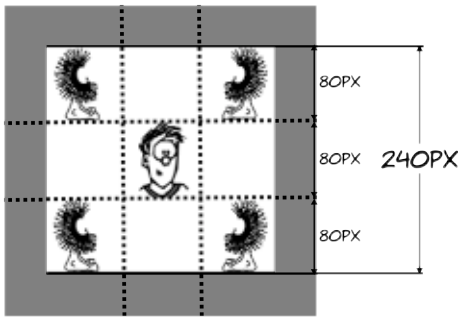


```
height: 400px;
width: 720px;
border-width: 80px;
border-image:
 url('border1.png') 80 fill;
```

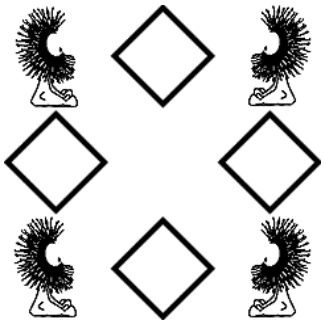
As you can see, the center of the image gets stretched to fill the space, but the corners stay where they are.

This diagram shows what's going on. The 80 in the border-image value above specifies a pixel length, the browser uses this length to slice the image into nine sections, each 80 pixels square.

The four corner sections are not adjusted, they remain in the corners of the elements. The center segment expands to fill the remaining space.

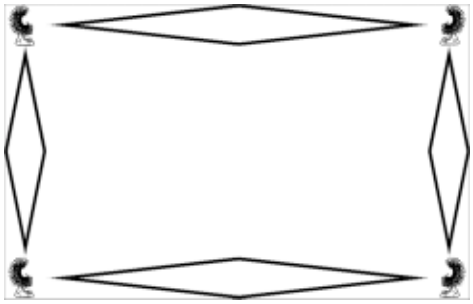


In the example above the middle segments on the sides are also stretched but you can't see that in this example as they are solid white. To see this, let's have a look at an example with a different image.



Let's apply the same rules with this new image:

```
height: 400px;
width: 720px;
border-width: 80px;
border-image: url('border2.png') 80;
```

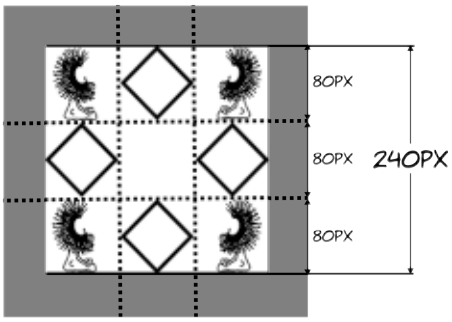


Now the stretching of the middle segments is more apparent.

The pattern fits neatly inside the middle segments, each 80 pixels square. As we saw before, the corners stay the same but the middle segments are stretched. This is because there is the third parameter to border-image, when we leave it off we get the default. The previous example is equivalent

to this:

```
border-image:
 url('border2.png') 80 stretch;
```



There are two other keywords that can be used instead of stretch. The first is repeat:

```
height: 400px;
width: 720px;
border-width: 80px;
border-image:
 url('border2.png') 80 repeat;
```

The image in the middle segment is repeated across the available width and height.

The second is round:

```
border-image:
 url('border2.png') 80 round;
```

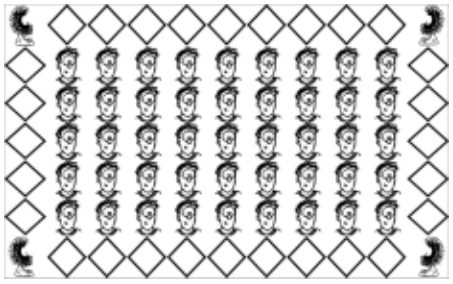
It looks like there's no difference between repeat and round, but that's due to a careful choice of element size to demonstrate the technique. If the size of the element is reduced slightly the difference is apparent. Have a look at the two screenshots below of the same two rules applied to a 680x360px element.

Repeat retains the width of the center segment – if it doesn't fit across the width of the box then parts of the segment will be cut off.

Round will adjust the width of the center segment to fit a whole number of times across the width.

When you have a middle segment that makes a repeated pattern then round is usually going to be what you want. If the middle segment is a solid color or gradient, like a beveled edge, then use repeat to avoid any distortion from the browser adjusting the image.

You may be curious about what happens to the middle segment when using repeat or round. This screenshot shows that the middle behaves in the same way as the middle segments on the border.



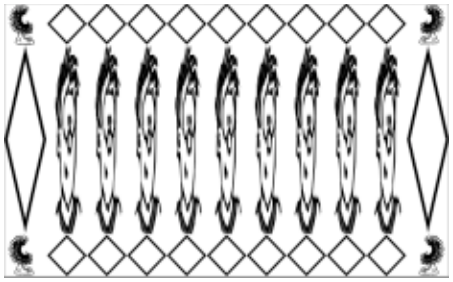
```
border-image:
 url('border3.png') 80 fill round;
```

If you ever need to create internet bank notes, this might be the way to go.

You can also use different approaches on the horizontal and vertical borders:

```
border-image:
 url('border3.png')
 80 fill round stretch
```

The first keyword applies to the horizontal borders, the second to the vertical borders.



Now you know enough to implement all three examples from the beginning of this section. We will take a single, fixed size image and use it to make elements which adapt in size to their contents yet retain a sharp image at the edges for rounded corner and bevel effects.

First we'll make some buttons and tabs. Here is a generic image we'll use to provide the background for both:

A generic image for creating panels, buttons and tabs

Using this image we can easily create an image like this.

The code for the border image on each of the above buttons is:

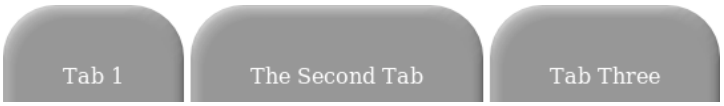
```
border-width: 45px;
```

```
border-image: url('border4-bevel.png') 45 fill repeat;
```

Each button has then had a text size set individually, and they have different amounts of text. Notice that the buttons are not even the same shape as our original image, but it has been adapted seamlessly.

[Slicing diagram in here]

Now let's use the same image to create some tabs:

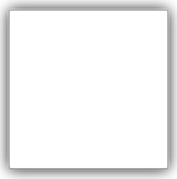


To achieve tabs we just have to adjust one line of CSS from the previous example, we set the bottom border width to be zero width:

```
border-width: 45px 45px 0px 45px;
border-image: url('border4-bevel.png') 45 fill repeat;
```

The border image, as with the buttons, adapts to the size of the content.

We've already seen the direct support for box shadows in CSS3, but they can also be achieved with border-image. If we combine the CSS below with the image on the right, you can achieve the results below.



```
border-image:
 url('drop-shadow.png') 70 repeat;
```

You're not likely to use this often, but there may be occasions where you want a more precise control over a shadow than box-shadow allows.

Border image is hard to override selectively, if you want to scale an image across a background with the built in CSS support (if available) and border image if not, you will need to make use of something like modernizr.js

10.5 Create gradients with CSS

Gradients - a smooth transition from one color to another - have always been popular with designers. In CSS2 the only way to implement a gradient is to create it as an image in a graphics program and attach it as a background to the element. This has problems and limitations, several of which you'll be familiar with from above:

- Images don't always scale well, which can create problems when using them as a background to content that is intended to scale
- If you decide to change your color scheme slightly, you may have to regenerate all your gradient images
- Every different gradient you want to use means an extra download from the server, increasing page load times for users and your bandwidth requirements
- If the element color is to change in response to user interaction, for example a menu item on mouseover, you need to make double the number of images, doubling all of the above problems[this can be a cartoon argument]

Browser quick check

	Standard	Prefixed
IE	-	-
Firefox	-	3.6
Chrome	-	7
Safari	-	4
Opera	-	11.1

Please place above table in sidebar

AJ A CSS gradient will be allowed anywhere you currently can specify an image. At this time browsers only have support for using them as

backgrounds, so our examples will concentrate on that.

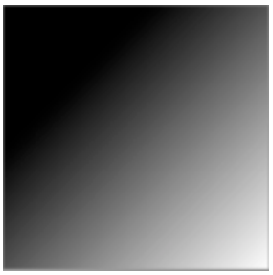
A simple gradient is very easy – simply specify a keyword for orientation and two colors. The browser will calculate a gradient across the whole background, treating the first color as the starting color and the second as the end color:

```
background: linear-gradient(
 top, #000, #fff
);
```

As well as up and down, gradients can go from one side to another:

```
background: linear-gradient(
 left, #000, #fff
);
```

It could be you want something other than up and down or left and right, a gradient can also have a rotation:

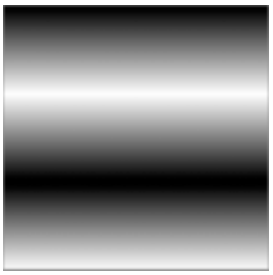


```
background: linear-gradient(
 left 315deg, #000, #fff
);
```

If you add more colors then the browser will treat them as equally spaced 'color stops' and calculate the gradient accordingly:

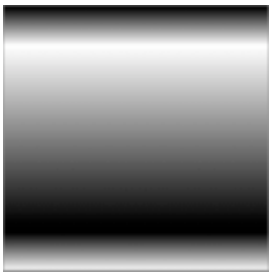
```
background: linear-gradient(
```

```
 top, #000, #fff, #000, #fff
);
```



Finally, if you don't want the color stops to be evenly spaces, you can give percentage values for each color stop:

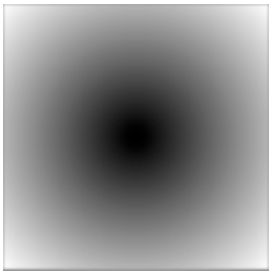
```
background: linear-gradient(
 top, #000, #fff 15%, #000 85%, #fff
);
```



AJ Linear gradients are all very well, but sometimes you want a more circular effect – for example a spot highlight on a 'glassy' button. CSS3 also allows you to create radial gradients, we'll look for them now.

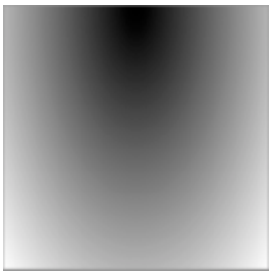
Radial gradients are just as simple as linear gradients, just supply a start color and an end color.

```
background: radial-gradient(
 #000, #fff
);
```



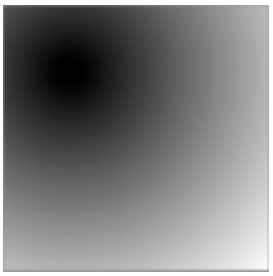
More interesting effects can be achieved by positioning the center of the gradient:

```
background: radial-gradient(
 top, #000, #fff
);
```



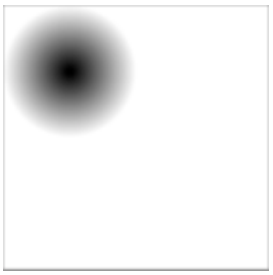
The gradient center can be positioned anywhere:

```
background: radial-gradient(
 25% 25%, #000, #fff
);
```



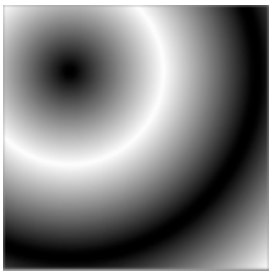
Using the contain keyword means that the gradient will stop when it touches the edges of the containing element:

```
background: radial-gradient(
 25% 25%, contain, #000, #fff
);
```



As with linear gradients, a radial gradient can have any number of color stops:

```
background: radial-gradient(
 25% 25%, #000, #fff, #000, #fff
);
```



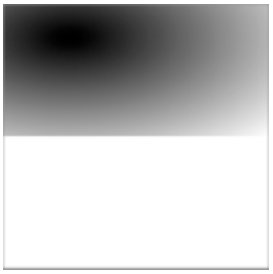
Because gradients replace background images, you can use the same background properties on them as you would use for background images. You can use this to your advantage to produce some useful effects, let's have a look.

If you want your gradient to only cover part of the background then you can use the background size property:

```
background: linear-gradient(
 top, #000, #fff
) no-repeat;
background-size: 100% 50%;
```

This might be most useful where you want to put a gradient in a fixed part of the background rather than scaling it across the whole thing.

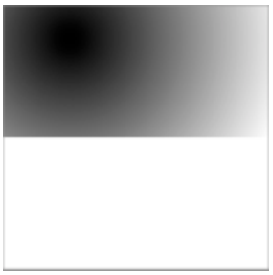
You can also size a radial gradient, though the effect isn't so pleasing:



```
background: radial-gradient(
 25% 25%, #000, #fff
) no-repeat;
background-size: 100% 50%;
```

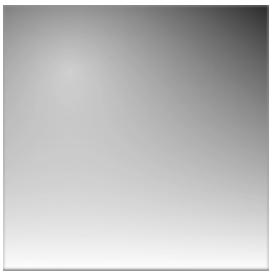
However it does allow us to see another property of radial gradients: by default they are ellipsoid rather than circular; we haven't noticed up until now because we've been applying them to square elements.

You can make the gradient circular using the circle keyword:



```
background: radial-gradient(
 25% 25%, circle, #000, #fff
) no-repeat;
background-size: 100% 50%;
```

It's also possible to layer multiple gradients if you use RGBA colors. Here is a radial gradient over a linear gradient to create a highlight effect:



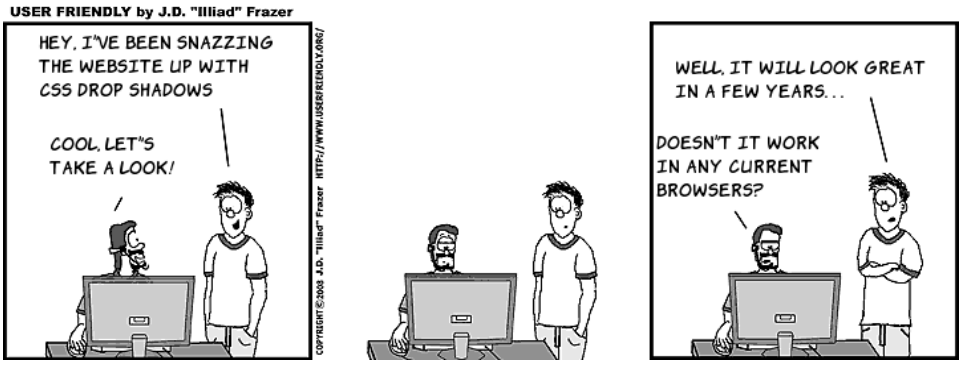
```
background: radial-gradient(
 25% 25%, circle,
 rgba(255,255,255,0.75),
 rgba(255,255,255,0)
) no-repeat,
linear-gradient(
 top, #000, #fff
) no-repeat;
```

There's no reason why your gradient can't have the same starting and ending colors. We can modify the final example from section 10.2.2 above to use a gradient instead of loading an extra image from the server:

```
background: top right
  url('pitr-head.png') no-repeat,
 linear-gradient(top,
  rgba(255,255,255,0.5),
  rgba(255,255,255,0.5)),
 bottom right
  url('aj-head.png') no-repeat,
 linear-gradient(
  top,rgba(255,255,255,0.5),
  rgba(255,255,255,0.5)),
 top left
  url('mike-head.png') no-repeat,
 linear-gradient(top,
  rgba(255,255,255,0.5),
  rgba(255,255,255,0.5)),
 bottom left
  url('sid-head-bg.png') no-repeat;
```

Dust puppy Now you've seen all the new features, let's take a detailed look at current browser support.

10.6 Browser Support



Key Complete or nearly complete support: Incomplete or alternative support: Little or no support:

Browser	Text / Box Shadow	Border Image	Border Radius	Multiple Backgrounds	Background Size	CSS Gradients
Chrome 7						
Safari 4						

Browser	Text / Box Shadow	Border Image	Border Radius	Multiple Backgrounds	Background Size	CSS Gradients

Ha! WebKit invented CSS drop shadows and CSS gradients.

Safari and Chrome have supported box-shadow since their respective version 3 releases though with some differences to the spec. Although they invented CSS gradients, their proposed syntax has been dropped in favor of the much simpler version in the current spec.

Firefox 3.6

Firefox 4

Firefox may have taken a bit longer to add support for some of these things, but at least they got it correct!

Firefox has support for box-shadow since version 3.5. Simply use the standard rules as illustrated above.

IE 8

IE 9

Browser	Text / Box Shadow	Border Image	Border Radius	Multiple Backgrounds	Background Size	CSS Gradients

Wait, none of this works in Internet Explorer! You can't just ignore two thirds of the browsing public! We need to sell stuff to everyone!

Stef

Actually, IE has had support for drop shadows and gradients in CSS since version 5.5.

AJ – alternate speakers from here

I knew it! Microsoft makes a superior product. That's why they dominate the browser marketplace.

Well, maybe IE9 will be OK...

Don't stand there arguing. Get implementing!
OK, we'll look in detail at IE in the next section

Opera 10.5

Just getting things right isn't enough, they need to be fast and lightweight.

Opera have complete support for text and box shadow in their 10.50 release. They have support for the 2008 draft of border-image and ... draft of border radius. Opera 10.10 and above support multiple backgrounds and background size.

The table above is split into rows which indicate the support in various versions of browsers, each then followed by a 'colorful' comment by the browser's character advocate and then a 'factual' comment from AJ, then an optional code example.

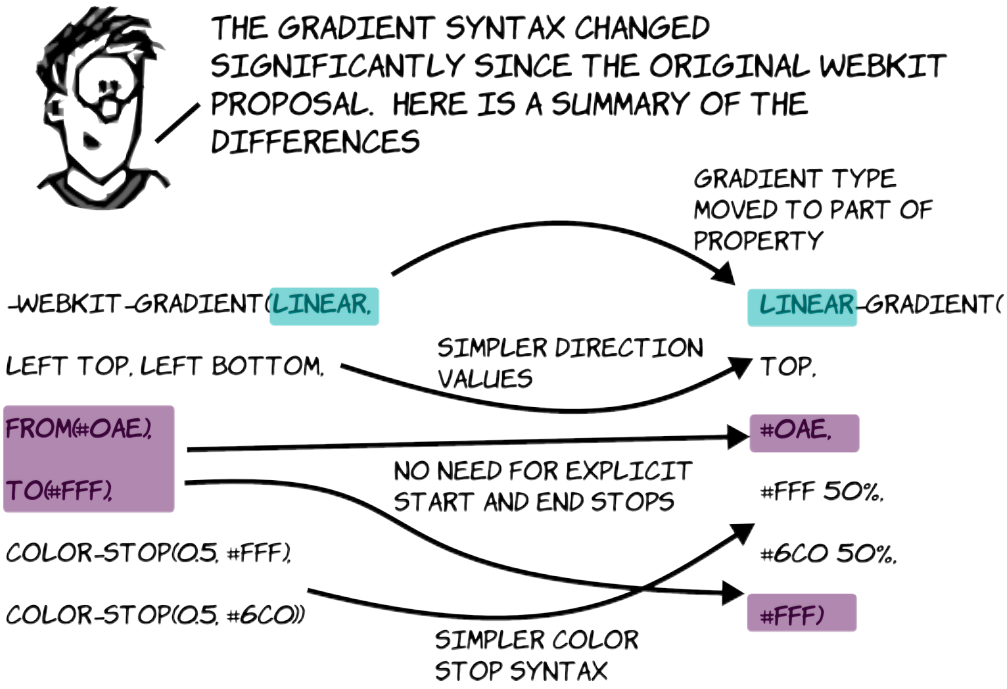
10.6.1 Borders and backgrounds in Chrome, Firefox and Opera

Chrome (before 10) and Safari, like Firefox use a vendor extension for box shadow, the vendor extension is the same for both:

```
-webkit-box-shadow: rgb(0,0,0) 3px 3px 3px;
```

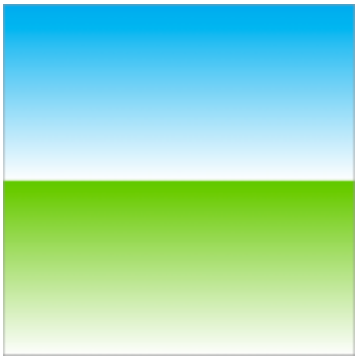
Neither of the WebKit browsers currently support inset, spread radius support was added in Chrome 8 and Safari 5 but isn't working reliably yet. WebKit also handles alpha transparency of elements with shadows differently to Firefox, we'll cover that later.

Text shadow is supported, using the standard syntax, in Safari 1.1 and Chrome 2.0, but support for multiple shadows was only added in the 4.0 versions.



For WebKit you need to use their original syntax, Firefox supports the standard syntax but with a vendor prefix and no other vendors have yet announced support. Your cross browser code should look something like this:

```
background:
 -webkit-gradient(linear,
  left top, left bottom,
  from(#00abeb),
  to(#fff),
  color-stop(0.5, #fff),
  color-stop(0.5, #66cc00));

background:
 -moz-linear-gradient(
  top,
  #0ae,
  #fff 50%,
  #6c0 50%,
  #fff);

background:
 linear-gradient(
  top,
  #0ae,
  #fff 50%,
  #6c0 50%,
```



```
#fff);
```

Firefox supports box-shadow through a vendor extension:

```
-moz-box-shadow: rgb(0,0,0) 3px 3px 3px;
```

The Gecko engine also supports inset and spread radius, though the color is optional (and defaults to black).

```
-moz-box-shadow: rgb(0,0,0) 3px 3px 3px 9px;
```

Text shadows are supported using the standard syntax since Firefox version 3.5. Firefox 3.6 supports all the properties discussed in this chapter, but with vendor extensions, Firefox 4.0 will use the standard properties. Opera and Microsoft are confident enough in the stability of the spec to skip the vendor extension stage and implement the box-shadow rule directly. To support all browsers you will need to issue multiple declarations like this:

```
-moz-box-shadow: rgb(0,0,0) 3px 3px 3px 9px;
-webkit-box-shadow: rgb(0,0,0) 3px 3px 3px, rgb(0,0,0) 3px 3px 9px;
-box-shadow: rgb(0,0,0) 3px 3px 3px 9px;
```

Note that, as the old WebKit based browsers don't support `spread-radius`, an extra shadow has been added in an effort to simulate the effect.

You say border-image is not fully supported, but it seems to me to work pretty

well cross browser.

Border image is one of the properties which has changed significantly since it was first introduced to the spec. As we speak, all implementations follow the shorthand property from the September 2008 Working Draft.

But what everyone's implemented isn't going to be in the standard?

Following that draft significant changes were made, adding specific properties for each component and also the fill keyword. Before border-image was like a sub property of border, now it's a stand alone property with sub-properties of its own.

But does that mean we can't use it now?

It is likely that vendors will move towards the new syntax once the spec reaches the Proposed Recommendation status. In the meantime it's possible to use it in a cross browser way while still being compatible with the current spec. Let me show you a quick example:

```
-moz-border-image: url('border1.png') 80;          1
-webkit-border-image: url('border1.png') 80;       2
border-image: url('border1.png') 80;               3
border-image: url('border1.png') 80 fill;          4
```

1 Legacy Firefox support
2 Legacy WebKit support
3 Support for 2008 draft implementations
4 Support for current draft implementations

Replace #1, #2 , #3 in the following paragraph with a cueball

To get border-image to work in current browsers you need to first specify the vendor specific properties for #1 Firefox and #2 Safari/Chrome. Next give the September 2008 version of the property for #3 Opera, finally the #4 current version of the property with the fill keyword. This ensures that future browsers, which fully implement the property, will render identically to current browsers, which treat fill as being the default.

Unfortunately if you don't want the behavior from the 2008 spec, where fill is the default, there is no way to override it in CSS. The easiest way to achieve this is to use an image that is blank or transparent in central the area.

10.6.2 Borders and backgrounds in Internet Explorer

Internet Explorer was the first browser to implement a method for specifying drop shadows in CSS, way back in the version 5.5 release. Microsoft implemented a method of calling an ActiveX control and applying it to an element which can be used either from CSS or Javascript. ActiveX has a pretty bad reputation in web developer circles which may partly explain why these techniques didn't start getting seriously explored until recently. The IE way to apply several effects is to use a filter:

[hedgehog diagram: -ms-filter property, single quoted value, filters]

There are two filters for shadows in IE8: DropShadow and Shadow.

DropShadow can accept a color value with Alpha transparency:

```
-ms-filter: "progid:
DXImageTransform.Microsoft.DropShadow
(
        color=#ff000000, offX=2, offY=2
)"
```

For the color parameter above we have used four hexadecimal pairs. The last three pairs are equivalent to how you would specify black in CSS: #000000. The first hex pair is a value between 0 and 255 for opacity.

The differences between the two are minimal in practice –, Shadow will apply a gradient to the edges

[redo screenshot above]

```
-ms-filter: "progid:
DXImageTransform.Microsoft.Shadow(
        color=#000000, direction=135, strength=2)"
```

In the example above there is a background color on the element the shadow applies to so that it will apply to the whole element rather than just the text. If you want a shadow on the text you need to leave the background transparent. As you may guess, this makes applying a shadow to both the text and the box a bit more interesting

There is also an IE filter for gradients, though compared to CSS gradients they are very limited. You can specify only a start and end color, and they can either be vertical or horizontal.

```
-ms-filter: "progid:
DXImageTransform.Microsoft.gradient(
GradientType=1,
startColorstr=#CC1C5B9B,
endColorstr=#E56CBFFF)";
)
```

[Simple left right gradient screenshot]

```
-ms-filter: "progid:
DXImageTransform.Microsoft.gradient(
GradientType=0,
startColorstr=#88FFFFFF,
endColorstr=#00FFFFFF)";
```

[Simple top bottom gradient screenshot]

Internet Explorer 9, when it's released, will have full support for the CSS standards for box and text shadow and rounded corners. Use conditional comments to provide a specific stylesheet to IE8 and below if you want to support all versions, or investigate a solution like CSS3Pie.

10.7 In a Nutshell

In this chapter you've learned about features of CSS3 for borders and backgrounds:

- Adding visual depth with drop shadows on text and boxes
- Creating striking effects with border images
- Making things smooth with rounded corners
- How multiple can backgrounds make your life easier

- Snazzing up your designs with gradients

Most of these features give web authors a way to accomplish effects that are commonly used but, with CSS2, have to be created in graphics packages as images or require JavaScript hacks and additional HTML markup