

Chapter 4

Stylesheets

from

Web Design

A Complete Introduction

Nigel Chapman and Jenny Chapman © 2006

PDF published by MacAvon Media © 2009

This PDF document contains one chapter from the book ***Web Design: A Complete Introduction***.

Free teaching and learning materials are available at the book's supporting Web site www.webdesignbook.org. Further chapters from ***Web Design: A Complete Introduction*** are available from MacAvon Media at www.macavonmedia.com.

Contents

Rules in CSS	153
Selectors	155
Inheritance	160
Typography	162
Fonts	162
Font Styling	167
Type Size	170
Text Decorations	173
Formatting Paragraphs	174
Layout	176
Layout Algorithm for Normal Flow	177
Padding, Borders and Margins	183
Backgrounds	188
Positioning	191
Floats	191
Absolute Positioning	194
Special Cases	199
Links	199
Lists	201
Tables	202
Forms	207
Advanced Features of CSS	208
Generated Content	208
Specifying Box Types	210
Media-Specific Stylesheets	212

Chapter 4 from “Web Design: A Complete Introduction” by Nigel and Jenny Chapman

Copyright © 2006 Nigel Chapman and Jenny Chapman
All figures © MacAvon Media

Nigel Chapman and Jenny Chapman have asserted their right under the Copyright, Designs and Patents Act 1988 to be identified as the authors of this work.

This PDF version published in 2009 by MacAvon Media: www.macavonmedia.com

This material comprises part of the book “Web Design: A Complete Introduction”
Published in print (ISBN-13 978-0-470-06089-6 (PB) ISBN-10 0-470-06089-1 (PB) by
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England
Telephone (+44) 1243 779777
Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit Wiley’s Home Page at www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be emailed to requests@macavonmedia.com.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this chapter are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher and Authors are not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.



Stylesheets



As the examples in the previous chapter demonstrate, the default way in which browsers display the elements that make up Web pages varies from the uninspiring to the dreadful. Thanks to the efforts and ingenuity of Web designers, most real Web pages don't look that bad. Stylesheets are the modern tool that allows designers to control the appearance of pages and improve on the browser's default efforts. Other more tortuous methods have been used in the past, and earlier versions of HTML provided some support for styling using attributes. Stylesheets are to be preferred, however, for two reasons. First, they provide more extensive control over all aspects of a page's layout and appearance than HTML's presentational attributes ever did. Second, they allow the specification of appearance to be kept independent of the markup that expresses structure. The structure is always available in the markup, irrespective of what a page looks like. It is also available to user agents that are not concerned with appearance at all, such as search engine spiders or screen readers. The appearance can be altered without affecting the structure. Different stylesheets can be applied to the same document for different purposes, and stylesheets can be used to control presentation in any medium, not just for display on a screen.

In principle, another advantage of separating stylesheets from markup is that any stylesheet language can be used with XHTML. At the present time, however, browsers only understand stylesheets written in one of two languages: **CSS** (the Cascading Stylesheets language) and **XSLT** (the Extensible Stylesheet Language for Transformations). CSS provides extensive control over layout, typography and other aspects of the presentation of XML, XHTML and HTML documents in graphical browsers. XSLT is a more powerful language, which can be used for radically restructuring a document for the purposes of presentation. One fairly common use of XSLT is for transforming XML documents into HTML, as a means of



preparing them for display in browsers that cannot display XML documents directly. We will not describe XSLT at all, since CSS is more than adequate for most Web designers' needs.

In this chapter, we discuss the mechanics of using stylesheets in conjunction with XHTML markup to control many aspects of the appearance of Web pages. In Chapter 10 we will return to this subject and consider the more important question of how to use these techniques to create usable and good-looking pages.

A minor but important matter is the way in which a stylesheet becomes associated with an XHTML document. The easiest way is to place the stylesheet in its own file (typically this will have an extension `.css`), and refer to it from the document using a `link` element in the head, as described in Chapter 3:

```
<link href="Resources/styles.css" rel="stylesheet" type="text/css"/>
```

This arrangement means that a single stylesheet can be shared by many documents – often an entire Web site – to ensure a consistent style. Changes to the stylesheet will affect all the documents, making maintenance much easier. Occasionally, a stylesheet will only be applicable to a single document. In that case, you might prefer to embed the stylesheet in the document itself. This can be done using the `style` element, which appears in the head. It has a `type` attribute, which should be set to `text/css`. The content of the element consists of the CSS code which would otherwise be in an external file.

```
<style type="text/css">
  stylesheet
</style>
```

Various expedients are commonly used to conceal embedded stylesheets from old browsers that don't understand them, and to avoid the need for entity references within the stylesheet. HTML editors and Web authoring programs will normally insert these automatically. Linking to external stylesheet files is generally trouble-free and is to be preferred.

Rules in CSS

A CSS stylesheet consists of a collection of **rules**, describing how the browser should display elements of an XHTML document. Each rule consists of two parts: a **selector**, which determines which elements the rule applies to, and some **declarations**, which describe how those elements should be displayed. Here is an example of a simple CSS rule:

Technical Detail

The style Attribute

There is a third way of adding stylesheet information to a document. Any XHTML element may have a `style` attribute, whose value is a string consisting of a sequence of declarations. The effect is the same as a rule with a selector that matches just that element and a body containing the declarations. For instance, to style a level 1 heading in the same way as it would be using the example rule given in the main text, the following start tag could be used:

```
<h1 style="font-family: 'Lydian MT', Futura, Georgia, serif;  
font-size: 200%;text-align: center;">
```

Unlike the stylesheet rule, this attribute would only affect the particular heading to which it was attached.

Although not actually deprecated in XHTML 1 or 2, the use of the `style` attribute is discouraged, because it fails to separate the styling and content cleanly. Web authoring programs routinely generate `style` attributes, though.

```
h1 {  
  font-family: "Lydian MT", Futura, Georgia, serif;  
  font-size: 200%;  
  text-align: center;  
}
```

This example shows the general form of a CSS rule. The selector comes first. In this case, it is the element name `h1`. A selector like this, consisting of an element name, matches all elements of the corresponding type, so the styling specified in the declarations of this rule will be applied to all `h1` elements.

The declarations follow the selector; they are enclosed in curly brackets, which mark the extent of this rule. (The delimiters can never be omitted, even if there is only one declaration.) Declarations are separated by a semicolon – almost everyone also adds a semicolon after the last declaration in a rule, although this is optional, because it makes editing easier. Declarations do not need to be written on separate lines, but they often are, to make rules more readable.

Each declaration consists of two parts. First comes a *property*, which is the name of some aspect of the element's appearance. This is followed by a colon, and then the value for the property. Thus, the second declaration in our example gives the value 200% to the property called `font-size`. As you see, unlike attribute values in XHTML, most property values in CSS do not have to be enclosed in quotes, unless the value includes a space (or other whitespace character, such as a tab).

More than one hundred different properties are available in CSS2.1, the version we will be concerned with. They are the semantic heart of CSS, since they determine what can be controlled using a stylesheet. The language's syntax is trivial in comparison – we have already described most of it. Much of the remainder is concerned with different forms of selector. In order to provide concrete examples of rules while describing the syntax of selectors, we will use a single property, `color`[†], which controls the colour of text. It can take values that specify a colour in various ways. We will just use names, which hopefully will be self-explanatory. CSS allows sixteen colours to be named in this way. (In Chapter 5 we will describe how to specify a much wider and more useful range of colours.) A simple example of a rule that sets the `color` property is:

```
h1 { color: aqua; }
```

which causes all `h1` elements to be set in aqua-coloured text.

Selectors

We have illustrated the simplest form of selector, an element name. Rules that use element names as selectors define a consistent appearance for every element with that name in a document. They replace the browser's default rules, but still only allow for as many different styles as there are element types in XHTML, which may not be adequate. It is common practice to format the opening paragraph of an article slightly differently from the rest, for example. Rules which have `p` as the selector will apply to all paragraphs, though. In XML you could define a separate element – `p1`, say – for first paragraphs (though you might question whether it would really represent a structural unit), but in XHTML you must stick with the repertoire of available element types. The `class` attribute provides a way of distinguishing between subsets of an element type.

As we explained in Chapter 3, any element may have a `class` attribute, whose value is a string consisting of one or more names, separated by spaces. If the same name appears among

[†] North American spelling is always used for property names and keywords.

the `class` values of two elements, they are considered to belong to the same class, which has that name. For instance, the tag `<h1 class="red">` assigns an `h1` element to the class `red`, and `<h1 class="red underlined">` assigns it to two classes, `red` and `underlined`. The primary purpose of this classification is to support different styling for different classes.

Suppose we assign the value `first` to the `class` attribute of the first paragraph of every section of a page, by using the start tag `<p class="first">` to open such paragraphs. (There is, of course, no `section` element in XHTML 1, so suppose that we assign any paragraph that follows an `h1` element to this class.) Then the selector `p.first` will match all first paragraphs, and can be used in a rule to define any special formatting that seems appropriate. To set the opening paragraph in grey, you would use the following rule:

```
p.first { color: gray; }
```

In the general case, a selector consisting of an element type followed by a dot and a name matches all elements in that class (i.e. all elements of that type whose `class` attribute includes the name). A rule with a selector of this form will take priority over one that selects all elements of a type, so if there is a rule with selector `p` as well as one with selector `p.first`, it will be used for all paragraphs, except those with `class="first"` (or any other value for which there is a specific rule). Thus, if in addition to the rule above, we had

```
p { color: black; }
```

all paragraph text except for paragraphs belonging to the class `first` would be set in black. Where an element belongs to more than one class, the rules for all of its classes are applied. If this causes a conflict, later rules take precedence over earlier ones. For instance, if a paragraph belongs to two classes, `first` and `changed`, and there are two rules:

```
p.first { color: gray; }  
p.changed { color: purple; }
```

the last rule will be used: any paragraph with `class="first changed"` will be set in purple. Conflicts between CSS rules are always resolved in favour of the later rule.

Actually, matters are slightly more complicated – although the complications correspond to what you would probably expect. A rule with a selector for a class within an element type takes precedence over a general rule with a selector for that element type, only in respect

of those properties that are explicitly declared in the class's rule. Suppose, for example, we define a rule of the form:

```
p {
    prop1: value1;
    prop2: value2;
    prop3: value3;
    prop4: value4;
    prop5: value5;
}
```

(where *prop1* and so on stand for some CSS property names, and *value1* and so on for appropriate values) and another:

```
p.first {
    prop3: value31;
    prop5: value51;
}
```

Then a paragraph element of class `first` would have the values *value1*, *value2*, *value31*, *value4* and *value51* for these properties.

The selector `*` matches any element. Hence, `*.c` matches any element belonging to the class `c`. This is a commonly used kind of selector, because classes may be used to apply aspects of styling to several different element types. For instance, you might decide that certain links (a elements), headings (h1, h2,...) and also arbitrary runs of text (span elements) should be purple, so elements of any of these types might be assigned to the class `purple`. Rather than use separate selectors for all the possible cases, you could simply write a single rule:

```
*.purple { color: purple; }
```

This would allow you to assign other types of element to this class later, without changing the stylesheet. Because this is a common pattern, a shorthand is provided, whereby you may omit the `*`, so that the selector just comprises a dot followed by a class name, for example,

```
.purple { color: purple; }
```

If you wish to apply special styling to one individual element, you could invent a class and assign just that element to it, but you can also use the value of its `id` attribute in a selector.

A selector consisting of an element name followed by a `#` and a name matches the element whose `id` is equal to the name, so `p#summary` would match the paragraph with `id="summary"`. So would `*#summary`, since `id` values must be unique. As with classes, `*` can be omitted, giving the shorthand form `#summary` for selectors based on `id` values.

Careful assignment of elements to classes can be used to produce any styling scheme you might wish, but some patterns can be dealt with more conveniently using special-purpose **contextual selectors**. These selectors are for using on those occasions when you wish to apply certain formatting to elements of a particular type only when it occurs in a certain context. A good example is provided by `li` elements. Recall from Chapter 3 that these may appear either inside an `ol` element, as the items in an ordered list, or a `ul` element, as those of an unordered (bulleted) list. You might very well want to apply different formatting to the two cases. In reality, it isn't very likely that you would want to set them in different colours, but let's suppose that you do: items in a bullet list will be blue, those in a numbered list grey. If a selector consists of two element names separated by a space, it matches elements with the second name if and only if they are found in the content of an element with the first name. For instance, the selector `ul li` matches `li` elements within a `ul`, so we could achieve our aim of colour-coded list items with these rules:

```
ul li { color: blue; }
ol li { color: grey; }
```

Contextual selectors like this can be used to alter the styling of any element in circumstances where the appearance of a surrounding element makes the usual styling inappropriate. It is not quite adequate for all cases, though. Considering list items again, the following XHTML markup in conjunction with the rules above does not produce the desired result.

```
<ol>
  <li>One</li>
  <li>Two</li>
  <li>
    <ul>
      <li>Un</li>
      <li>Deux</li>
      <li>Trois</li>
    </ul>
  </li>
</ol>
```



Technical Detail

Attribute Selectors

Some additional forms of selector, based on attribute values, are available. These are more useful for stylesheets used with XML documents, and are rarely needed with XHTML. If E is an element name, and a is an attribute of elements of that type, $E[a]$ matches all elements of type E that have a defined, $E[a=v]$ matches all elements of type E whose a attribute is equal to v , and $E[a\sim v]$ matches all elements of type E whose a attribute consists of a sequence of names separated by white space, one of which is equal to v . You should be able to see that $E.c$ is an abbreviation for $E[class\sim=c]$ and $E\#i$ is an abbreviation for $E[id=i]$. The full forms of these selectors are required in XML, because XML documents do not have a fixed repertoire of elements and attributes; in particular, there is no reason to suppose that `class` and `id` will be attributes of every element.

A final form of attribute selector is $E[a|=v]$. This only matches on elements whose attribute a consists of a series of words separated by hyphens, which begins with v . To see why this might be useful, consider the form of language codes that might be the value of the `xml:lang` attribute of any element.

The numbered items in the inner list appear in grey, because they are inside an `ol` element, and the CSS rule for `ol li` takes precedence, since it appears later in the stylesheet. Reversing the order of the rules fixes this problem, but will fail if the lists are nested the other way out. To cope with such contingencies, if `>` replaces the space in a contextual selector, the rule is only applied to elements of the second type if they are children of the first, that is, they are immediately contained in them, as the `li` elements with French numbers are with respect to the `ul` element, but not the `ol` element. If the stylesheet rules are changed to

```
ul>li { color: blue; }
ol>li { color: grey; }
```

all the list items are coloured as intended.

The formatting of an element might depend not just on the element it is contained in, but on the element that precedes it. For instance, we have already noted that the first paragraph following a heading is often given special typographical treatment. We have shown how

this can be done using a class, but a more direct way is by using the selector `h1 + p`, which matches a `p` element if and only if it follows an `h1` element, so the rule with `p.first` as its selector given previously could be replaced by

```
h1 + p { color: gray; }
```

In this context, an element follows another if it comes right after its end tag; in terms of the document tree, the two elements must be siblings.

A similar special case occurs with elements that are the first child of their parents. For example, the first item in an unordered list would be the first child of the enclosing `ul` element; if an `h1` element was the first thing in a document, it would be the first child of the body element, and so on. Such first children may benefit from special styling. A selector consisting of an element name followed by `:first-child` matches elements in this position. To set an `h1` at the beginning of a document's body in purple, we could use the following rule:

```
h1:first-child { color: purple; }
```

This colon notation is used for selectors that match **pseudo-classes** of element. These are classes that don't depend on the presence of a name in the `class` attribute, but depend on properties of the document. We will see more examples of pseudo-classes later.

With all the possibilities available for writing selectors, it is quite possible that you will end up writing several rules with different selectors but the same declarations. These can be combined: several selectors, separated by commas may appear in a rule. The single rule behaves as if it was a sequence of rules, one for each of the selectors, with identical declarations. Thus, if you wanted to set headings at all levels in a document in blue, instead of writing six rules for each of the heading element types, you could write a single rule:

```
h1,h2,h3,h4,h5,h6 { color: blue; }
```

(Perl and Awk programmers will probably expect to be able to use regular expressions, such as `h[1-6]`, as selectors, but you can't.)

Inheritance

If a value is assigned to some property for an element in a document, the value is inherited by that element's children, unless a rule causes a different value to be explicitly assigned.

Browser Quirks

CSS Hacks

There are some cases where certain browsers' implementation of some CSS features is so badly flawed that pages using those features will be displayed too poorly to be acceptable. The chief offenders are Internet Explorer 5 (both Windows and Macintosh versions, although they exhibit different problems) and Netscape 4. All of these browsers are now considered obsolete, but they are still in use, especially IE5/Win. For modern browsers, adding a DTD to your documents to force them into strict mode is usually sufficient to get most CSS to display properly.

To work round the problems with defective browsers, certain tricks, which exploit additional bugs, have been developed to allow different CSS rules to be used for different browsers. In particular, IE5 does not implement the `>` operator in selectors, so any rule that uses the selector `html>body`, which should always match, will be ignored. Conversely, `* html` should always fail, since `html` is the root element, but IE5 behaves as if the root was enclosed in some other element, so rules that should only be used for this browser can use this selector.

On the whole, such hacks ought to be avoided, and you should keep an eye on published statistics on browser usage and your site logs to see whether you need to use them.

In other words, the value of a property is applied to the entire contents of an element. For instance, given the rules

```
.red { color:red; }
.purple { color:purple; }
```

and the markup

```
<p class="red">
A paragraph containing some <em>emphasized</em> text.
</p>
```

the word *emphasized* would be set in red, together with whatever styling had been specified for `em` elements, because the `color` property is inherited from its parent `p` element. On the other hand, if the markup was

```
<p class="red">
  A paragraph containing some <em class="purple">
    emphasized</em> text.
</p>
```

emphasized would be in purple, because the explicit assignment of a colour to text of class `purple`, to which this element belongs, overrides the inherited value. This rule is intuitive and usually causes no problems. For some properties whose values are numerical and may depend on context, it is occasionally unclear what the inherited value will be. As a general rule, any computation that is necessary to arrive at a value is performed before inheritance, so any values that are dependent on context will be computed in the context of the parent element. We will see some examples of how this operates later in the chapter.

As a consequence of the way inheritance works, default values of properties that affect an entire document can be set simply by using a rule with selector `body` or `html`. Any properties that are not set in this way will be set to default values, which may be chosen by the user, wired in to the browser, or determined by the CSS standard.

Typography

Typography – the style and appearance of text – is a central concern in design for print-based media. Centuries of experience in typesetting have led to the creation of thousands of type-faces that can be used – in conjunction with design skill and powerful software – to create attractive and effective pages in a huge variety of styles. It would be nice if the expertise of traditional typesetting could be transferred to the Web, but the new medium imposes some constraints which seriously limit the scope for fine typography.

Fonts

A **font** is a collection of character shapes, called **glyphs**, which are used for displaying text. All the glyphs in a font will share certain characteristics that convey an impression of uniformity. Whenever a character is to be displayed, the corresponding glyph – which is nothing more than a small image – is selected from the font and rendered on the screen. Although glyphs are usually very small, and each glyph for a particular character must be recognizable whatever the font, there is considerable scope for subtle differences in the appearance of the glyphs, with the result that the look of a piece of text can vary greatly, depending on the font

in which it has been displayed. The few samples shown in Figure 4.1 are sufficient to give some idea of the variations that are possible within the limits of font design.

Several fonts that share some fundamental characteristics may be grouped into a **font family**, consisting of several **variants**, such as a regular version, an italic and a boldface one. These variants will be noticeably different, but the members of the family will have a similar appearance, based on such things as the height of characters, the style of serifs (see below) and the ratio of thick to thin strokes in the glyphs. These similarities will ensure that fonts from the same family look well together on the page, whereas fonts from different families are unlikely to do so. Figure 4.2 shows samples of the members of two contrasting font families. There is no standard specifying how many members there must be in a font family, or what they should be. Some families consist of just a regular and an italic version, while others may have several different boldface versions or boldface italics as well.

The characteristics of fonts can be described in several ways. A fundamental distinction is drawn between **monospaced** – or **fixed width** – fonts, in which each character occupies exactly the same horizontal space, and **proportional** fonts, in which the space occupied depends on the width of the character. In Figure 4.1, Andale Mono and Courier are examples of monospaced fonts, all the rest are proportional. Text in a monospaced font looks as if it had been produced on a typewriter or teletype machine, and is often used for computer program code. On the Web, monospaced fonts are the default choice for pre elements.

Another distinction is between **serifed** and **sans serif** fonts. Serifs are the little decorative strokes on the ends of letters, as shown in Figure 4.3. Glyphs in sans serif fonts lack these. Trebuchet, Arial and Verdana are all sans serif fonts; so is Andale Mono, although Courier has crude serifs. Proportional serifed fonts, such as Times Roman and Georgia, are the traditional choice for setting long passages of continuous text in books and other printed material. They work less well on screen, because the fine detail of the serifs is not rendered accurately at low resolutions. The bolder appearance of sans serif fonts is generally considered more readable on screen. Despite this, the default proportional font on most browsers when they are first installed is usually a serifed font, often Times Roman.

The last three fonts shown in Figure 4.1 are different from the rest. They are examples of what are called **display fonts**. The majority of existing fonts fall into this general category. They are fonts that are not suitable for continuous text, but have more eye-catching designs, suitable for headlines, advertising and other short pieces of text that need to have a high visual impact. This gives greater scope to the font designer, hence the vast number of display fonts. Some display fonts, such as Brush Script, are loosely based on handwriting styles;

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the
 lazy dog

Georgia

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the lazy
 dog

Times New Roman

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over
 the lazy dog

Verdana

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the
 lazy dog

Arial

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the
 lazy dog

Trebuchet

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps
 over the lazy dog

Andale Mono

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps
 over the lazy dog

Courier

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the lazy dog

Brush Script

**ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890**

**The quick brown fox jumps over the lazy
 dog**

Impact

ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
 1234567890

The quick brown fox jumps over the
 lazy dog

Comic Sans

Figure 4.1 Some common Web fonts

ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Georgia Regular</i>	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Arial Regular</i>
ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Georgia Italic</i>	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Arial Italic</i>
ABCDEFGHIJKLMNOPQRSTUVWXYZ- VWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Georgia Bold</i>	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Arial Bold</i>
ABCDEFGHIJKLMNOPQRSTUVWXYZ- VWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Georgia Bold Italic</i>	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 1234567890 The quick brown fox jumps over the lazy dog <i>Arial Bold Italic</i>

Figure 4.2 The members of two font families



Figure 4.3 Serifed and sans serif fonts

they are generically referred to as **calligraphic** or **cursive** fonts. Fonts which do not fit neatly into any category, are sometimes called **fantasy** fonts. Comic Sans is a fairly restrained example; there are many more fantastic fantasies.

The choice of a font in which to set a page makes a significant difference to its appearance. On the Web, though, you can never know for sure which font will be used to view the text on your pages. As we will explain shortly, CSS provides a way for a designer to *suggest* which fonts should be used for each type of element, but there is no guarantee that they will actually be used, for the simple reason that Web browsers can usually only use fonts that are installed on the system on which they are running, and there is no guarantee that any particular visitor to a page will have the designer's chosen fonts installed on their system. In fact, the fonts shown in Figure 4.1, with the exception of Brush Script, are the only ones likely to be installed on all recent Windows and MacOS X systems. It's particularly unlikely that an average domestic or business user will have the vast array of esoteric fonts that most graphic designers acquire, so there is little point in a designer spending a lot of time choosing the perfect font for a Web page that gives it just the right degree of style and distinction. The chances are that most people will see it in Times Roman.

Even if you choose a font that almost everybody is bound to have, it is still possible that some people will see the page rendered in a different one, because many browsers allow their users to specify a stylesheet of their own, which overrides any referred to in the page. It is probably the case that very few ordinary people do this, and those who do will generally have a good reason – for instance, they may have some vision defect, which makes it necessary to use highly readable fonts and colours. You can't even work backwards, assume that if you don't specify a font the browser's default will be used, and design around that. This is because browsers provide different defaults (some version of Times Roman is a popular choice, but it is not a universal one), and all browsers provide a simple means for the user to change the default. Many people switch to a sans serif font as their default.

Given this situation, no typographical decisions made by a designer can be considered final, but CSS does allow you to specify your preferred fonts, using a collection of font properties. The first of these is **font-family**, which is used to select a font family. You could have guessed that, but perhaps not guessed that its value is a list of names. As our first example of a CSS rule illustrated, the names are separated by commas, and names with spaces in them must be enclosed in quotes. The collection of fonts in the list is called a **font set**. The browser is expected to look for each of the fonts in the set in turn, until it finds one it can use. In other words, the font set consists of those fonts that the designer would find acceptable, in decreasing order of preference. Thus, given the rule:



```
h1 { font-family: "Lydian MT", Futura, Georgia, serif; }
```

the browser would first look for fonts in the Lydian MT family. This is a bit of a long shot, so the search is likely to fail. Next, it would try Futura, which is more commonly found but still not part of most standard installations. If no Futura fonts are found, the browser will try Georgia. This time, it is very likely to succeed, because Georgia is present on most systems. The final name in the list, *serif*, is provided in case none of the named font families is available. It is one of CSS's **generic font families**. The others are *sans-serif* (note the hyphen), *cursive*, *fantasy* and *monospace*. Browsers must provide some font for each of these families. It is to be expected that they will correspond to the class of font suggested by each generic family's name, but there is no guarantee of this. Some browsers allow their users to choose the fonts which will be used for some or all of these generic families, so it is quite possible that a rule such as the one above could result in h1 elements being set in Verdana.

The example rule for h1 elements shows a useful pattern for font set values: start with the specific font you would like to use, follow it with more common ones that resemble it, and finish off with a generic font family. It is worth specifying the less common fonts, as long as your design does not depend on them too heavily, because there is no knowing what fonts any user may have: word processors, desktop publishing and vector drawing programs, among others, usually install a wide selection of fonts. It is, however, a good idea to test your page on systems without the fonts you have specified, to see how badly their absence affects its appearance. (Using a font management program to turn off each of the fonts in the set, until only the generic family is left will show you how it degrades. Figure 4.4 shows the effect of turning off each of the fonts in our example set. Notice the change of width, even though these fonts are all nominally the same size.) If a page looks truly awful without your preferred fonts, you should consider redesigning it.

Font Styling

Four other properties may be used to refine the choice of font: *font-style*, *font-variant*, *font-weight* and *font-stretch*. These set (somewhat arbitrary) independent characteristics, which together are enough to identify a font within a family. The *font-style* may be *normal*, *italic* or *oblique*. The first two values should be self-explanatory. An oblique font is slanted, as italic fonts invariably are, but does not necessarily have the handwritten quality of italics. Often, oblique fonts are created just by applying a skew transformation to a normal font. If the *font-style* is *italic*, but no actual italic font can be found, an oblique font may be substituted.

Technical Detail

Intelligent Font Matching in CSS2

The method described in the main text for choosing a font in response to a font-family declaration is not the full story. It is the method prescribed for CSS1, but CSS2 provides additional facilities which potentially can produce better results. The idea is that, if a browser cannot find a font that matches the name specified for a font-family, it may try to find an alternative that resembles the correct font. If that fails, it may try to download a suitable font from a server. If that also fails, it may synthesize a suitable font. Not all browsers will be capable of all these steps, and in practice most contemporary ones fall back on the CSS1 mechanism.

A CSS2 stylesheet can include font descriptions, which, as the name suggests, provide a description of the characteristics of a font. These descriptions take the form of special 'at-rules', which are a type of CSS rule that do not have a conventional selector, but instead use a keyword, prefixed by an @ character, and perform some function other than specifying the styling of elements of a document. For font descriptions, the rule has the form

```
@font-face { declarations }
```

The declarations set the values of various font-related properties, which provide a description of the font, sufficient for a browser to select a good approximation to a desired font, or to synthesize one.

Most interestingly, one of the properties that can be set in an @font-face rule is `src`, whose value is a URL from which the font can be obtained. This seems to be the perfect solution to the problems of Web typography: if browsers can download fonts, designers can use any font they wish to. Sadly, this state of affairs has not been achieved. Fonts are valuable intellectual property, and font foundries are reluctant to make them available for download. Until browsers implement a means of downloading fonts without potentially making them available to other applications, which is secure enough to satisfy the owners of font designs, font downloading will not take place. Given this, and the fact that font synthesis and intelligent matching are non-trivial to implement, most browsers simply ignore @font-face rules.

Examples of Font Substitution

Examples of Font Substitution

Examples of Font Substitution

Examples of Font Substitution

Figure 4.4 Font substitution

In CSS terms, the font variants are `normal` and `small-caps`: some font families include a special font that has small versions of upper case letters in place of lower-case, and this will be selected if `font-variant` is set to `small-caps`.

The `font-weight` property is more complicated. It selects the weight, or darkness, of a font. As the samples in Figure 4.2 illustrate, many font families include a boldface version, which is darker than the normal version, by virtue of using thicker strokes for the letter forms. Some font families include several versions with different weights; these are identified by terms such as ultra-light, light, bold, ultra-bold and black, but there is no standard system of terminology. Furthermore, the normal font of one family may actually be darker than the bold version of another, so boldness cannot be an absolute property. (Consider the font Impact in Figure 4.1.)

To accommodate the diversity of font weights, the `font-weight` property may take numerical values between 100 and 900. The value must be divisible by 100 (so it is something of a mystery why the range is not 1 to 9). The only meaning given to the weight values is that if $b > a$, a font with weight b is at least as dark as (i.e. is not lighter than) a font with weight a . The two fonts may, however, be of the same weight. This will often be the case, in fact, since few font families have as many as nine different weights. For convenience, some keywords may be used as values for this property: `normal` is the same as 400, `bold` is the same as 700. The keywords `bolder` and `lighter` can also be used, to specify the weight of a font relative to the weight inherited from an element's parent. The intention is to increase or decrease the weight, but, in view of our earlier remarks, this cannot be guaranteed. The effect of `bolder` is defined as specifying a font that is darker than the inherited one, if it exists; otherwise it

adds 100 to the inherited `font-weight` value. If the inherited value is 900, it has no effect. The meaning of `lighter` is similar, but changes the weight in the opposite direction.

A similar approach is taken to the `font-stretch` property. Some fonts come in versions which are spaced out horizontally to a different degree. Fonts which have letters closed up tightly together are called condensed fonts; they are used for setting type in narrow columns (in newspapers, for example). Fonts which have letters stretched wider apart are called expanded fonts. As with weight, the stretch of a font is not an absolute property. Times New Roman Regular, for example, is a condensed font and has no extended version; Univers has extended, condensed and ultra-condensed versions, in different weights. The CSS keywords `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `normal`, `semi-expanded`, `expanded`, `extra-expanded` and `ultra-expanded` form a partial ordering from narrowest to widest, like the numerical weight values. The keywords `wider` and `narrower` are used to select a font with a different width from the inherited value, in the same way as `bolder` and `lighter` are used to select relative weights.

Type Size

Specifying font sizes introduces additional complications. The keywords for the `font-size` property are `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large` and `xx-large`. It is recommended that the size corresponding to each of these values should be 1.2 times that corresponding to the previous value, so if `medium` is 14pt, say, `large` should be 16.8pt. The keywords `larger` and `smaller` can be used to specify size changes relative to an inherited value. Generally, user agents are able to scale fonts to arbitrary sizes, so there is no question of missing sizes, as there is of missing weights. However, the question that does arise is: how big is `medium`? The answer is that it might be anything. It is the default size set by the user in their browser. This means that, using the size keywords, a designer cannot know exactly how big the text on their Web pages will appear when displayed in a browser.

As an apparent way to control type size, CSS also allows the `font-size` property to specify the size as an absolute dimension. This may be in any units, but the size of a font is usually given in units of **points**, abbreviated pt; in CSS, 1pt is defined to be exactly equal to 1/72 inch. (And an inch is defined to be 2.54cm.) The font size is a measure of the maximum height occupied by text set in the font; it is not necessarily the height of any particular glyph, although brackets are often exactly equal in height to the font size. Nor is it the case that two fonts that are the same nominal size will have characters that are the same height. The complexities of letter forms mean that one might appear much smaller than the other,

hence font substitution can have a marked effect on the apparent size of type, even though the font size is unchanged (as Figure 4.4 demonstrated).

The CSS rule

```
body { font-size:12pt; }
```

should set the default font size in a document to 12pt. However, as with so many aspects of Web page layout, it is the user who has the final say. Browsers provide commands for increasing and decreasing the size of text, and this can override the effect of any `font-size` declarations in a stylesheet. Sizes will all be increased or decreased by the same factor, though, so the relative sizes of different elements will be maintained.

An alternative to specifying font sizes in points is to specify them in pixels, abbreviated `px` in CSS. Since, as we will explain in Chapter 5, the resolution of monitors is always taken to be 72 pixels per inch, `px` would appear to be identical to `pt`. Matters are more complicated, however, because browsers may use different resolutions for text in order to make type appear to be the same size on different platforms. Specifying sizes in pixels has the advantage that the type will scale appropriately when a page is displayed or printed on a device with a different resolution, so if this is an important consideration, the value for the `font-size` property should be given in pixels.

Font sizes can also be specified as a percentage of their inherited value. Thus

```
h1 { font-size: 135%; }
```

would cause level 1 headers to be set 1.35 times the default font size, which would be inherited from the enclosing body element. Another way of achieving the same effect, which is based on traditional typesetting practice, is to use units of `ems` instead of points. An `em` is a unit that is equal to the font size, so in a 12pt font, 1`em` equals 12pt, in a 10pt font, 1`em` equals 10pt, and so on. (The unit's name comes from the fact that in many fonts an upper case `M` is 1`em` wide.) When the value of `font-size` is given in `ems`, the `em` unit is inherited, that is, it is equal to the font size in the parent element. Thus, the following rule is equivalent to the previous one:

```
h1 { font-size: 1.35em; }
```

As a general rule, you should use relative units for font sizes, thereby respecting users' preferences. If you find it necessary to fix the size of type in your pages for any reason, it is advisable only to use an absolute size in a rule for the body element, and to set the size of type in any other elements relative to this. In this way, changing a single rule will be sufficient to change the sizes throughout the document, while maintaining their relative proportions.

The font size does not dictate the height of lines of text: extra space (known as leading) can be inserted between lines to produce a more open appearance. Therefore, the line height must be specified separately, and this is done with the `line-height` property. Like font-size, this can be set in absolute units (usually pt or px), relative units (em), or as a percentage. In the case of `line-height`, the percentage is relative to the current font size (not any inherited line height). The line height can also be expressed as a number, which is multiplied by the font size. Thus, if the font size has been set to 12pt, the following four rules all have the same effect:

```
body { line-height: 15pt; }  
body { line-height: 1.25em; }  
body { line-height: 1.25; }  
body { line-height: 125%; }
```

Notice, though, that the line height specified by the first of these rules will be the same irrespective of the font size, whereas the others will scale with the font.

Because it is often convenient to set several of the font properties and the line height at once, CSS has a `font` property to do exactly that. Its value consists of values for any or all of the font style, variant and weight (in any order) separated by spaces, followed by a font size, then optionally a / and a line height, and finally a font set. Defaults are used for any of the omitted values. The notation for the size and height is taken from conventional usage in typesetting, where 12pt text on a 14pt baseline is written 12pt/14pt. A CSS rule to set default font characteristics would be

```
body { font: 12pt/15pt Trebuchet,Verdana,sans-serif; }
```

Here, the style, variant and weight will take their default value. Although the notation for setting size and height is conventionally used for absolutely sized type, it can also be used with relative values. For instance, to set type in a small size with double the default line spacing, the following rule could be used:



```
body { font: small/200% Trebuchet,Verdana,sans-serif; }
```

Note that you cannot omit the size and font family components and have them inherited. If you wanted to set text belonging to the class `emphatic` in bold italics, for instance, you would need to use a rule like the following one:

```
.emphatic { font: italic bold small/200% Trebuchet,Verdana,sans-  
serif; }
```

(In fact, it would be easier to use explicit `font-style` and `font-weight` declarations in this case.)

Text Decorations

The font-related properties determine the font from which character glyphs are taken. Other properties can be used to modify the appearance of text. We demonstrated at length in the early part of this chapter how to change the colour of text, using the `color` property. In Chapter 5 we will explain how to access the full range of colours.

Other forms of decoration are possible. The `font-decoration` property may take any of the values `underline`, `overline`, `line-through`, `none` or `blink`. Underlining is used so commonly for links, that it can have few other uses without confusing people. Overlining is rarely used at all. A line through text is sometimes used to indicate changes that have been made to a page. The value `none` is not redundant, since `text-decoration` is inherited, and this value must therefore be used to turn it off. The final value `blink` is something of a curiosity: its intended effect is to make the text disappear and reappear at regular (short) intervals. This mimics the effect of a custom HTML element, `blink`, that was once supported by some browsers. The result is irritating to most people and has the potential to cause seizures in some, so the element fell into disuse. As if ashamed of the presence of the `blink` value for `text-decoration`, the CSS2 standard specifies that ‘... user agents ... are not required to support this value.’

It may seem that the case of letters is not a matter of presentation, but you may reflect that setting a heading in upper-case is often done for visual emphasis, so it can make sense to consider case as a style property, and CSS does so. The relevant property is `text-transform`, with the possible values `none`, `capitalize`, `uppercase` and `lowercase`. In this case, `capitalize` means setting the first letter of each word in upper case. (This style is sometimes called ‘title case’.)

Formatting Paragraphs

The contents of block elements – paragraphs in particular – are laid out in lines, in the familiar way. This offers further scope for styling. A line will rarely fit the available width exactly. Extra space can be used to fill up lines in one of four ways. It can be added at the right hand end (for text that is written from right to left), so that the left edge of the paragraph is straight, but the right is ragged. Alternatively, it can be added at the left end, causing the right edge to be straight. A third option is to add an equal amount of space at both ends, so that both edges are ragged and each line is centred in the available width. Finally, the space may be distributed between the words, so that both edges are straight, but the inter-word spacing may not be the same on consecutive lines. Text laid out in this way is said to be *justified*. Figure 4.5 illustrates these four possibilities. The `text-align` property is used to determine which option is used; it takes the values `left`, `right`, `center` or `justify`. In general, left alignment is the best option for extended passages of text. Centred and right-aligned text may be useful for headings, or for certain kinds of list. Justification is the norm for many sorts of printed matter, including books and academic papers, but Web browsers make a poor job of justification compared with the specialized software used for typesetting. Justified text must, therefore, be used judiciously on Web pages.

One of the standard conventions for marking the start of a new paragraph is to indent the first line. To format paragraphs in this way using CSS, use the `text-indent` property, whose value is the length by which you want the first line indented. Relative units, such as `ems` can be used here, to produce an indent that scales with the font size. Note that it is only the first line's indentation that is controlled by this property. To indent an entire paragraph you must use other properties, which we will describe later. The `text-indent` value can be negative, to 'exdent' the first line, like this:

```
p.exdent { text-indent: -2em;}
```

However, unless there is space to the left of the paragraph, the exdent material will disappear. To produce hanging indentation, the entire paragraph must be indented, in the way we will describe later, and the first line indent set to a negative value.

There is a tradition in certain sorts of printing of giving more elaborate special treatment to the first line of a paragraph, or its first letter. CSS supports these special cases with the pseudo-classes `:first-line` and `:first-letter`, which may be used in selectors. You may sometimes see books in which the first line of certain paragraphs is set in boldface. This effect could be applied to every paragraph on a Web page with the following rule:

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

left

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

right

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

center

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

justify

Figure 4.5 Alignment of paragraphs

```
p:first-line { font-weight:bold; }
```

The first letter of a paragraph (usually the first paragraph of a chapter) is often set larger than the rest. A simple rule for doing this to every paragraph is

```
p:first-letter { font-size:200%; }
```

Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat. Duis aute irure
dolor in reprehenderit in voluptate velit esse cillum
dolore eu fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.

Figure 4.6 Applying special styling to the first line and letter of a paragraph

Figure 4.6 shows the combined effect of these rules. The large initial letter sticks up above the first line, which is a nice and fairly modern-looking style. More traditionally, large initials are set as ‘drop caps’, which fall below the line, with later lines wrapping round them. We will explain how to achieve that effect later in the chapter.

Layout

The properties described so far allow you to control the appearance of text to the extent that a basic word processor would support. Achieving more sophisticated results requires an understanding of the ‘box model’ that is used for controlling the layout of elements within a page.

Every element (including the body) is placed inside a box. As the name suggests, a box is a rectangular area which contains the styled content of the element. Each box is divided into regions, as shown in Figure 4.7. The bounding box of the content forms the innermost region. This is surrounded by some **padding**, which separates it from a **border**. Often, the border has a thickness of zero, because you rarely want to surround elements with visible borders. Outside the border is a transparent **margin** region, which will separate the box from its neighbours when the boxes for the elements on a page are laid out on the screen. As the figure indicates, the padding, border and margin regions do not have to be symmetrical with respect to the content. For instance, the padding below an element may be bigger than the padding above it. Any or all of these regions may have a thickness of zero. The margins’ thickness may be negative, in which case the content may extend beyond the box.



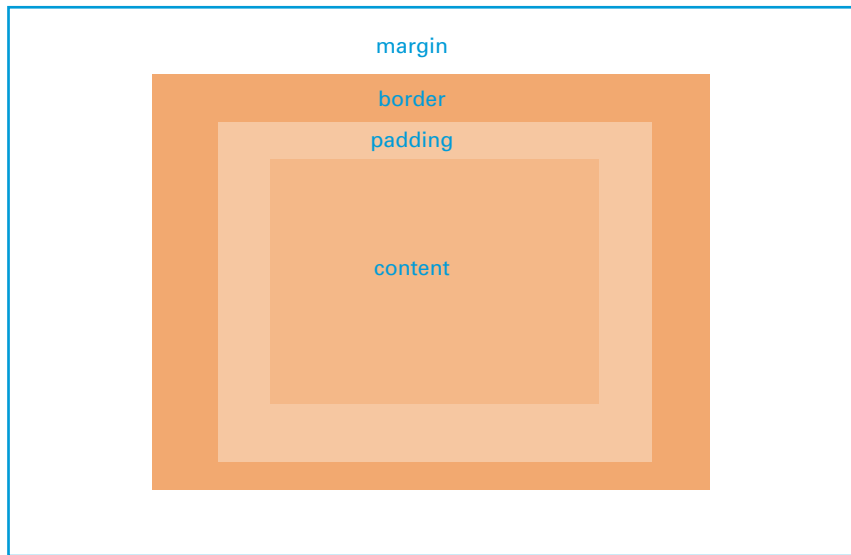


Figure 4.7 The CSS box model

Layout Algorithm for Normal Flow

The boxes containing the contents of the elements making up a page must be placed together in a browser window to display the page to a user. CSS2 has facilities that allow page authors to position the boxes just about wherever they want, relative to the page or relative to each other. If these facilities are not used, though, the browser puts the boxes together so that their content can be read from top to bottom, and in the normal reading direction for the page's language. (We will consistently assume that this is left to right; the necessary adjustments for right to left text should be obvious.) This is said to be the **normal flow** layout.

The essence of the layout algorithm for normal flow is easy to grasp: boxes for inline elements and the textual content of block elements are laid out next to each other, and then 'folded up' to fit inside the content region of the box of the block element that contains them. The boxes for block elements are then placed one above the other in the window. (See Figure 4.8.) The details are slightly complicated, though, especially in respect of the width of block elements' boxes.

Box Widths

Just as every element, except the document root, has an ancestor, every box has a **containing box**, which is the box corresponding to its closest block-level ancestor element.

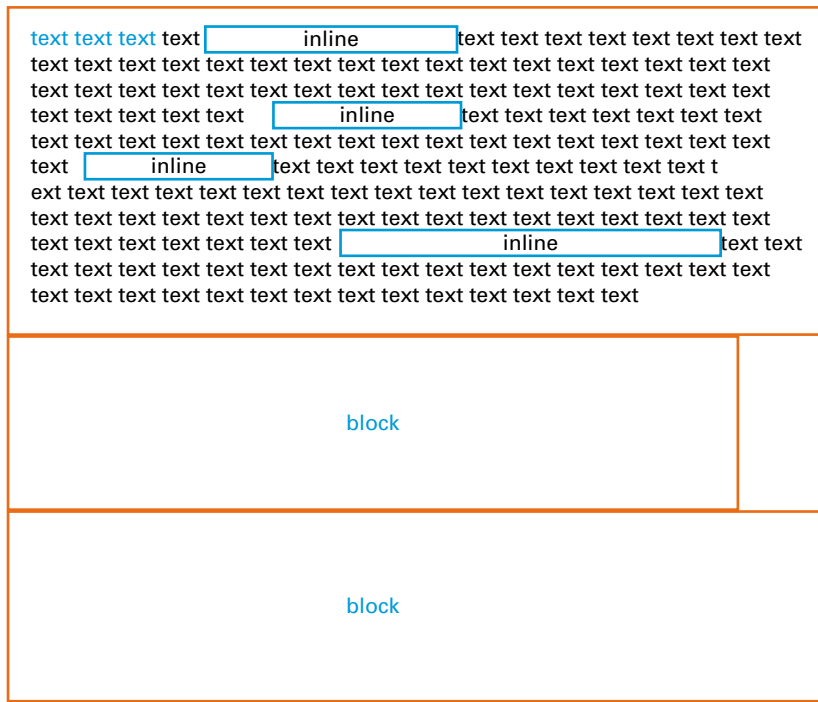


Figure 4.8 Normal flow

The root element's containing box is determined by the browser – in effect, it is the browser window. Thus, given the document body

```
<body>
  <div>
    This <span class="code">div</span> element contains some bare text
    as well as an embedded paragraph.
    <p>
      This paragraph lies <em>within</em> the <span class="xref">
      <span class="code">div</span> element</span>, so it is an example
      of a block element within a block element.
    </p>
  </div>
</body>
```

the containing box of the `p` element's box is that of the `div`, as is that of the first occurrence of a `span` with `class="code"`. The boxes for both the `span` elements nested inside

the paragraph are contained in the `p` element's box. The second occurrence of a `span` with `class="code"` does not have the `span` with `class="xref"` as its containing box, because containing boxes must correspond to block elements.

As this little example reminds you, `div` elements may contain bare text as well as block elements. For the purposes of layout, such text is considered to be in an anonymous box, as if it was the content of some block element. The containing box of the `em` element is therefore the anonymous box that holds the first sentence inside the `div`.

Text and inline elements are folded into lines so that they fit within the content area of their containing box. Extra space is added to fill up the width, according to the `text-align` property. This may entail breaking apart the box for an inline element, and assigning the parts to separate boxes on consecutive lines. Words are never broken using hyphens, though, and the line-breaking algorithms employed by Web browsers are not very sophisticated, so the resulting text is not laid out to the highest standards. After this, the block elements are laid out vertically inside their containing block. If there are nested block elements (`div`s within `div`s, for example), this process is repeated.

The main question that remains is: how is the width of each block element's box determined? This is what determines the length of the lines in which the inline elements are arranged.

Conceptually, the simplest case arises if the width is set explicitly to some absolute value. This is done by assigning a value in absolute units, such as pixels, to the `width` property. For instance

```
p { width: 600px; }
```

would set the content region of all paragraphs to be 600 pixels wide (unless a more specific rule applied). The text of each paragraph would be wrapped to fit into lines of that length. (The width of the padding, border and margin areas can be set independently using other properties, as we will describe later. Since the text only flows inside the content region, these properties have no influence on its layout.) Since `px` units have been specified, every paragraph to which the rule applied would be the same width. In particular, the paragraphs' width would be independent of the font size.

If, however, the width is specified using `ems`, like this:

```
p { width: 30em; }
```

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 4.9 Paragraph widths specified in em units

paragraphs set in different-sized type would have different widths, because the em unit relates to the font size within the paragraph (see Figure 4.9).

Furthermore, when paragraph widths are specified in em units, a user who increased the size of the type in their browser would see a ‘zooming’ effect, as the text and its enclosing boxes became larger. On the other hand, if the paragraph width is specified in pixels or some other absolute unit, the enlarged text would reflow to fit within the specified width when the font size is increased (see Figure 4.10).

A slightly more complex situation arises if the value of `width` is given as a percentage, for what is it a percentage of? The only sensible answer is that the percentage must refer to the containing box. In fact, it is a percentage of that box’s content region. (Usually... we will explain later on how it sometimes includes its padding.) This makes intuitive sense, because that is the region into which enclosed boxes must be fitted. A consequence of this arrangement is that, if the containing box’s width is specified in relative units, any enclosed boxes whose widths are percentages will expand along with it.



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 4.10 Paragraph widths specified in px units

An interesting degenerate case arises when the containing box is the browser window. This will happen to block elements that are not contained in any other element with an explicitly specified width. In that case, as you would expect, the following rule causes all paragraphs to extend half-way across the window:

```
p { width: 50%; }
```

This relationship is maintained no matter what the width of the window, and continues to be maintained if the user resizes the browser window. In that case, the text reflows in order to keep the lines restricted to half the width of the window.

This brings us to the final possibility. What if the width is not specified for a block element? In that case, as you might guess, the width is computed so that the element's box fills the

available width. That is, the sum of its left and right margins, borders and padding, together with the width of the content region, must add up to the width of the containing block. We will describe in the next section how to set the widths of the margins and so on, but it will hold few surprises. As long as all the relevant properties have explicit values, the content of the inner box will flow into the available left-over space. The same result is achieved by specifying the `width` explicitly as `auto`. In the degenerate case, where the containing block is the window, the text of the page will be broken into lines that fill the available width, reflowing as the window is resized, in the familiar fashion.

Where widths are specified in relative units, it is possible that some combination of font size and window width may lead to impossibly narrow or inconveniently wide text boxes. To avoid this possibility, the properties `min-width` and `max-width` can be used to specify a minimum and maximum width. If the computed width is less than the minimum, the minimum is used, and similarly if the computed width is greater than the maximum, the maximum is used. Thus, if you wanted a paragraph to be half the width of its containing box, unless that exceeded 30em in the current font, you could use the following rule:

```
p {  
  width: 50%;  
  max-width: 30em;  
}
```

Box Height and Overflow

Since boxes are rectangles, they have height as well as width; you can specify the height of their content region using the `height` property. Like `width`, its value can be a length in absolute units, which does not therefore scale with font size, a length in relative units, such as ems, which does, or a percentage. In this case, this is a percentage of the height of the content region of any enclosing box. In the case that the enclosing box is the browser window, though, using a percentage height is meaningless, since windows are effectively unbounded in height. You can also leave the height unspecified, or set it to `auto`. If you do so, the box will be made tall enough to accommodate its contents.

There are `min-height` and `max-height` properties to set bounds on the height in the same way as `min-width` and `max-width` limit the width.

If a box's height is set explicitly, it is possible that the contents will be too tall to fit into it. For instance, given the rule

```
p { height: 6em; line-height: 1.2em; }
```

any paragraph longer than five lines will be taller than its box. What happens to the over-matter (the lines that won't fit) depends on the value of the `overflow` property.

If `overflow` is set to `hidden`, the paragraph is clipped to the box; over-matter disappears. (This can actually be very convenient, as we will demonstrate later.) In contrast, if `overflow` is set to `visible`, the over-matter spills out below the box. This may cause it to run into any following text, usually with illegible results. An alternative is to set `overflow` to `scroll`. This causes scroll bars to be added to the box, so that over-matter can be brought into view by scrolling. It is difficult to make this look good on a page, and, since scroll bars require a certain amount of vertical space to display their arrows, using this setting with very short boxes may produce peculiar results which can be very frustrating for users. A final possible value for `overflow` is `auto`. The effect is browser-dependent, but should cause scroll bars to be provided if they are necessary; the standard does not stipulate whether they should be displayed if there is no over-matter, but a common interpretation of `auto` is to display scroll bars only when there is over-matter.

The `overflow` value is inherited. If there is no explicit value available to be inherited, it defaults to `visible`.

Padding, Borders and Margins

There are CSS properties for controlling the size of each of the padding, border and margin regions. For each of these regions there are five separate properties. We will illustrate the pattern these follow by describing the properties controlling the padding.

To set the thickness of the padding on each of the four sides of the box, you can use the properties `padding-top`, `padding-right`, `padding-bottom` and `padding-left`, each of which affects the side indicated by the last part of its name. The value may be a length, specified in any unit. You can use `px`, for instance, to set a padding width that is independent of the font size, or you can use `em` to get a padding whose width scales up with the font size. You can also specify a percentage. This will be interpreted as a percentage of the width of the containing box (not the box to which the property is being applied). Percentages in this context (unlike the `height` property) relate to the enclosing box's *width*, even for vertical measurements, such as the thickness of the top and bottom borders.

Values	Effect
v	top = right = bottom = left = v
v1 v2	top = bottom = v1, right = left = v2
v1 v2 v3	top = v1, right = left = v2, bottom = v3
v1 v2 v3 v4	top = v1, right = v2, bottom = v3, left = v4

Table 4.1 Shorthand notation for properties

The padding property is provided as a shorthand for setting the thickness of the padding on all four sides at once. It takes between one and four values (separated by spaces), which are interpreted as shown in Table 4.1.

For example,

```
h1 { padding: 1em 0em 0.5em; }
```

is equivalent to

```
h1 {  
  padding-top: 1em;  
  padding-right: 0em;  
  padding-bottom: 0.5em;  
  padding-left: 0em;  
}
```

Region	Properties for Individual Sides	Shorthand
padding	padding-top, padding-right, padding-left, padding-bottom	padding
border	border-top-width, border-right-width, border-left-width, border-bottom-width	border-width
	border-top-color, border-right-color, border-left-color, border-bottom-color	border-color
	border-top-style, border-right-style, border-left-style, border-bottom-style	border-style
margin	margin-top, margin-right, margin-left, margin-bottom	margin

Table 4.2 Properties controlling the box regions

You may find it easier to remember how these values work if you bear in mind that the specified values are applied to the sides in clockwise order: the first value is always applied to the top, the second, if present, is always applied to the right, the third to the bottom, the fourth to the left. Missing values are filled in by symmetry.

This pattern of four properties to set the thicknesses of individual sides and a fifth shorthand to set all four at once is used for all the regions of the box whose thickness can be controlled by the stylesheet. The names of the relevant properties are given in Table 4.2. As a special case, the properties for the border widths can also take one of the values *thick*, *medium* and *thin*, which give standard borders; the actual thickness corresponding to these values is browser-dependent. The only guarantee about them is that a thick border will be at least as thick as a medium border, which will be at least as thick as a thin one.

Borders

The table includes some extra properties which control borders. Since these are potentially visible, their appearance may vary as well as their thickness. These properties, too, come in sets of five, which follow the same pattern as *border-width* and its relatives. The colour is controlled by *border-color* and its variants, whose value can be any colour name or a more precise colour specification of the kind we will describe in Chapter 5. The *border-style* properties control the kind of line used for the border. The available set of values is summarized in Table 4.3.

Value	Appearance
<i>none</i>	no border
<i>hidden</i>	no border, and suppress any coincident box's border
<i>dotted</i>	a dotted line
<i>dashed</i>	a dashed line
<i>solid</i>	a solid line
<i>double</i>	two parallel solid lines with a gap between them
<i>groove</i>	a groove carved into the page
<i>ridge</i>	a ridge extruded from the page
<i>inset</i>	a shaded border that makes the box appear to be sunk into the page
<i>outset</i>	a shaded border that makes the box appear to be coming out of the page

Table 4.3 *Border styles*

Emerging Technology

CSS3 Borders

Some Web designers have expended a great deal of energy and ingenuity creating boxes with rounded corners, using a mixture of tricky XHTML markup, CSS backgrounds and even JavaScript. The CSS3 Backgrounds and Borders Module defines a `border-radius` property, which will make all this effort redundant. It takes two lengths, which define the radii of a quarter-ellipse to be used as the corner of a box border. The radii of the four corners can be defined separately, so for example, tabs with round corners can be produced by setting non-zero radii for the two top corners of a box.

The same module defines a `border-image` property, which allows you to define an image to be used along the border, so that you can have fancy patterned borders, as favoured for intertitles in silent melodramas.

The CSS3 Backgrounds and Borders Module is only a working draft at the time of writing, so the details of this feature may change before it is generally implemented in browsers.

There is still another level of shorthand applicable to borders. The style, width and colour of the border on any side can all be set at the same time, using `border-top`, `border-right`, `border-bottom`, `border-left` and `border`. These take up to three values, specifying the style (as in `border-style`), the width (as in `border-width`) and colour (as in `border-color`), in any order. Thus, the following rule puts elements belonging to the class `boxed` into a thin red box:

```
.boxed { border: thin red solid; }
```

Omitted values are set to defaults (which usually means that they are inherited). The values specified for the `border` property are applied to the borders on all four sides of the box – you can't specify different values for different sides, the way you can with `padding` and so on.

Despite the wealth of properties for controlling the appearance of borders, you probably won't use them very often.

Collapsing Margins

The values for `margin-top` and `margin-bottom` for vertically adjacent boxes are combined when boxes are placed above each other in the normal flow. The way this is done sounds odd, but it usually produces the expected effect. If the adjacent margins' values are either both positive or both negative, the margins are collapsed and the space between the boxes is set to the value with the larger magnitude. For instance, suppose the following rules apply:

```
p {
  font-size: 14px;
  margin-top: 1em;
  margin-bottom: 6em;
}
h1 {
  font-size: 18px;
  margin-top: 0em;
  margin-bottom: 3em;
}
```

and a document contained a level 1 heading followed by two paragraphs. The distance below the heading would be 54px (3em in the specified font size): the top margin of the paragraph would have been removed. Similarly, the space between paragraphs would be 84px, not 98px, because of the collapsing margins. However, if one of the margins is negative and the other positive, their values are added (i.e. the negative value is taken off the positive one), so if the paragraph rule was changed to

```
p {
  font-size: 14px;
  margin-top: -1em;
  margin-bottom: 6em;
}
```

the negative margin above the paragraphs would be subtracted, giving a distance of just 40px between the heading and the first paragraph.

These rules perhaps appear somewhat arbitrary, but in practice they usually behave so as to honour the designer's likely intent, maintaining important vertical spacing rules and discarding lesser ones where there is potentially a conflict. For straightforward layouts, it may be advisable to use either bottom or top margins consistently for spacing most elements, so that the question of collapsing margins does not arise.

Backgrounds

The background of every element's box can be set independently, either to a colour or to an image. To be precise, the background of the content, padding and border regions can be set to a colour or an image: the margin region is always transparent. Solid borders obscure the background, but if a dashed border is used, the background will show through in the gaps between the dashes.

The case of a plain coloured background is extremely simple: the `background-color` property has a colour name or specification (see Chapter 5) as its value, so the rule

```
.highlighted { background-color: yellow; }
```

would cause all elements belonging to the class `highlighted` to appear on a yellow background. To put the entire page on a coloured background, you simply use a rule that sets the `background-color` for the `body` element, like this:

```
body { background-color: gray; }
```

The value `transparent` may also be used, which has the effect of letting whatever is below the element show through. It is the default value for this property so – although backgrounds are not, technically speaking, inherited – the background of an element's parent will, by default, show through behind the element, giving the same effect as if it was inherited.

More elaborate pages can be created by using images as backgrounds. The `background-image` property is used for this purpose. It is the first example we have seen of a property which has a URL as its value. In such cases, the value is written as `url(the url)`. For instance,

```
body { background-image: url(bg.gif); }
```

This rule would place the image found at the URL `bg.gif` behind the content of the document body. While using an image as the background for the entire page is perhaps the most common use of the `background-image` property, it can be used in any rule, to apply a background to any element in the document. Applying different backgrounds to `div` elements can be an effective way of breaking up a page into visually distinct regions.

It is unlikely that any particular image will fit exactly the area occupied by a page or element – especially since pages can be resized by the user. If the image is larger than the box it will be clipped, but what happens if the background image is too small to fill the box it has been

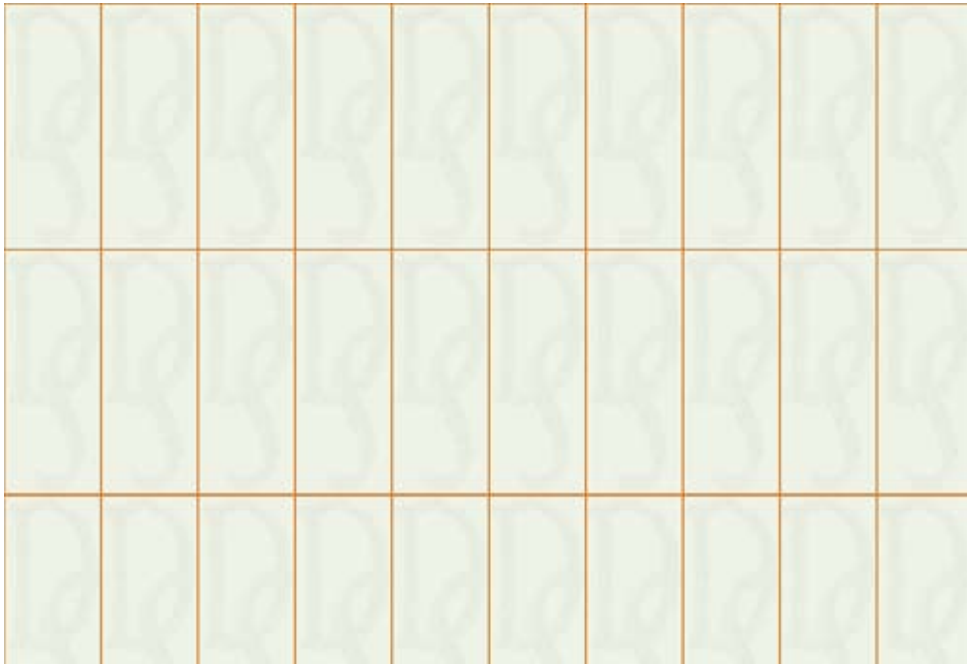


Figure 4.11 A tiled background

applied to? This depends on the value of the `background-repeat` property. If this is `repeat` (which it is by default), the image will be tiled, that is copies of it will be placed across and down the box to fill up the available space (see Figure 4.11 – we have added orange lines to show the edges of the tiles). This behaviour is most appropriately used to fill the background with a texture. For this to work, the image must be designed so that copies of it fit together smoothly, or at least in such a way as to create an obviously deliberate pattern.

If you intend to use a single image as a background for all the other content of a page, make sure to set `background-repeat` to `no-repeat`. This prevents tiling occurring: a single copy of the background image is used, and if it is too small, the background colour is used for the rest of the box. Failure to prevent a background image repeating in this way leads to pages that look amateur at best.

Between the two extreme options for `background-repeat` there are two other possibilities: `repeat-x` and `repeat-y` cause the image to be repeated horizontally and vertically, respectively. Figure 4.12 shows a possible application of horizontal tiling. The colour at the bottom of the gradient matches the background colour of the page, so no matter how tall



Figure 4.12 A background gradient using horizontal tiling

the window is, the colour at the top will always fade into the body of the page. In all cases where a background texture is produced by tiling, the tiled image itself can be much smaller than the box it fills: the gradient in Figure 4.12 is only two pixels wide, for instance. Thus, the background image will only occupy a small number of bytes. In contrast, an image that is used to provide a background for the whole page must be big enough to fill the page in a typical browser. Using such a background may noticeably increase the time a page takes to download.

One of the more interesting things you can do with a background image on a page that is too long to fit in a browser window is to hold it fixed while the page content scrolls on top of it – this is an effect that has no analogy in printed matter. The relevant CSS property is `background-attachment`. Setting this to `scroll` (its default value) allows the background image to be scrolled along with the rest of the page, but setting it to `fixed` holds the background, allowing the page content to slide over it as the user scrolls down the page. (Try it.) There isn't much point in fixing a repeated background, but if a single image is used, fixing it in this way should ensure that it is always visible behind the page.

There is a shorthand background property, which has as its value a list of values for the background properties. It is not necessary to include values for all of the properties `background-color`, `background-image`, `background-repeat`, `background-attachment` and `background-position` because default values will be used for any that are omitted. For example,

```
body { background: gray url(bg.gif); }
```

would be sufficient to set the background colour to grey, and use the image in `bg.gif` as a tiled scrolling background.

As in this example, it is a good idea to specify a background colour even if a background image is being used, to provide a fallback in case the image is not available owing to server or network errors. If the image has any transparent areas the background colour will show through them, so it is also necessary to provide a colour for such a contingency. (You cannot rely on the browser default for a background colour being set to white; several browsers allow the user to specify a default colour.)

Positioning

The normal flow algorithm produces a single text flow, intended to be read from top to bottom. This is not always what is required. Most monitors are wider than they are tall, which means that they can accommodate relatively short fat windows. There is ample evidence, though, that using long lines of text substantially decreases readability for most people, so if a designer wishes to take advantage of the available width it will be necessary to break the page into regions, none of which is excessively wide. (As a rule of thumb, a line of text is definitely too long if it exceeds seventy characters.) One possibility is to emulate the multi-column layout of newspapers; a more popular alternative is to use a fairly narrow region for the main textual content, and separate regions for auxiliary matter, such as navbars or external references. We will look at layout design in Chapter 10. Here we will describe the various mechanisms provided by CSS for breaking out of the normal flow.

Floats

The longest established way of disrupting the normal flow is by *floating* elements to one side. A floated element is taken from its position in the normal flow and moved as far left or right as possible, to the edge of its containing block. That is, if a box is floated to the left, it is positioned with its left edge coincident with the left edge of its containing box. Meanwhile, the normal flow continues down the right of the floated box (the lines being made shorter to make room for the float), as shown in Figure 4.13, where the coloured background shows the extent of the floated box. The top of the floated box is aligned with the top of the line in which the floated element appears. Once floated, a box becomes a block box, irrespective of the element whose content it contains. The `float` property controls floated elements. Its possible values are `left`, `right` and `none`, with the obvious meanings.

To force an element past a float and restore the normal flow, use the `clear` property, which has the possible values `left`, `right` and `both`, specifying which side of the element's box may *not* have a float next to it. For instance, if `clear` is set to `left` for a paragraph element, the paragraph will be placed on the first available line that does not have any preceding left-floated element beside it. In other words, it is pushed past any left floats that would have interfered with its normal flow.

**Lorem ipsum
dolor sit amet,
consectetur
adipiscing elit.**

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum

dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 4.13 A left-floated element

Traditionally, floats have been used for wrapping text around images that are too narrow to be sensibly placed on their own, but in conjunction with other CSS features they can be used to produce some attractive and useful text layouts.

Floating provides a way of implementing dropped initials, as illustrated in Figure 4.14. You will recall from earlier in this chapter that the `:first-letter` pseudo-class lets us apply special formatting to the first letter of a paragraph or other element, but that simply increasing the size of the first letter makes it stick up above the first line, rather than being embedded in the paragraph in the more traditional manner. The required behaviour is precisely that of a float. It is necessary to use a little care and some experimentation in setting the padding, margins and line height to make an initial letter of any given size fit into the paragraph. Figure 4.14 was produced using the following rule:

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 4.14 A dropped initial letter

Lorem Ipsum
Dolor Sit Amet

Lorem ipsum dolor sit amet, consectetur
 adipiscing elit, sed do eiusmod tempor
 incididunt ut labore et dolore magna
 aliqua. Ut enim ad minim veniam, quis
 nostrud exercitation ullamco laboris nisi
 ut aliquip ex ea commodo consequat. Duis aute irure
 dolor in reprehenderit in voluptate velit esse cillum
 dolore eu fugiat nulla pariatur. Excepteur sint occaecat
 cupidatat non proident, sunt in culpa qui officia
 deserunt mollit anim id est laborum. Lorem ipsum
 dolor sit amet, consectetur adipiscing elit, sed do
 eiusmod tempor incididunt ut labore et dolore magna
 aliqua. Ut enim ad minim veniam, quis nostrud
 exercitation ullamco laboris nisi ut aliquip ex ea
 commodo consequat. Duis aute irure dolor in
 reprehenderit in voluptate velit esse cillum dolore eu
 fugiat nulla pariatur. Excepteur sint occaecat cupidatat
 non proident, sunt in culpa qui officia deserunt mollit
 anim id est laborum.

Figure 4.15 A called-out heading

```

p:first-letter {
    width: auto;
    margin: 0em;
    padding: 0 0.1em 0 0;
    font-size: 300%;
    line-height: 0.733em;
    float: left;
}
    
```

A more radical use of floated boxes is to provide ‘called out’ headings, such as the one in Figure 4.15. Here, instead of putting the section heading on a line above the section, in the conventional manner, it has been placed in a box which has been floated to the left. You will notice a significant difference between the placement of the call-out in this example and the dropped initial in the previous one: the call-out box sticks out so that it only partly overlaps the following paragraph. This creates a more effective visual distinction. It is done by adding padding to the left of the paragraph, while placing the h1 element for the call-out before the p element in the XHTML document. Thus, the containing box for the heading is the body’s box, and the call-out moves all the way to the left of it. The paragraph shares this containing box, but – because of its left padding – the actual text is indented from the left, allowing room for the call-out to overlap. The relevant CSS rules are as follows:

```

h1 {
    width: 8em;
    float: left;
    color: black;
    font-size: 125%;
}
    
```

```

    line-height: 150%;
    padding: 0.25em;
    margin: 0 0.5em 0 0;
}
p { padding: 0 0 0 4em; }

```

The XHTML document's body has the following outline:

```

<body>
<h1>Lorem...</h1>
<p>
Lorem ipsum...
</p>
</body>

```

Once again, some care and experimentation is required in setting the exact padding and margin values. Note also that the width of the call-out box has been set explicitly.

The idea of floating a box so that it ends up in the padding area of following paragraphs can be taken further. If the padding is made wider than the floating box, then the latter will end up entirely to the left of any paragraph. It can then be used as a place to put content that is complementary to the main page – a navbar or a collection of links to other sites, for example. This is a commonly employed layout that makes good use of the available space in the window.

Absolute Positioning

Normal flow and floating provide a surprising amount of flexibility for designing page layouts that adapt themselves to different screen and font sizes, but sometimes it will be necessary to exert more explicit control over the positioning of elements on the screen. This can be done by setting the `position` property to `absolute`. If this is done, then some or all of the properties `left`, `right`, `top` and `bottom` can be set in order to determine the position of the four sides of the box. The box will be removed from the normal flow, and placed according to the values of these properties, each of which is a length, measured from the corresponding side of the box's containing box to the edge of the box's margin on that side (see Figure 4.16). As the diagram shows, for absolutely positioned elements, the position is measured relative to the outside of any padding in the containing box, but to the edge of the margin of the absolutely positioned box.



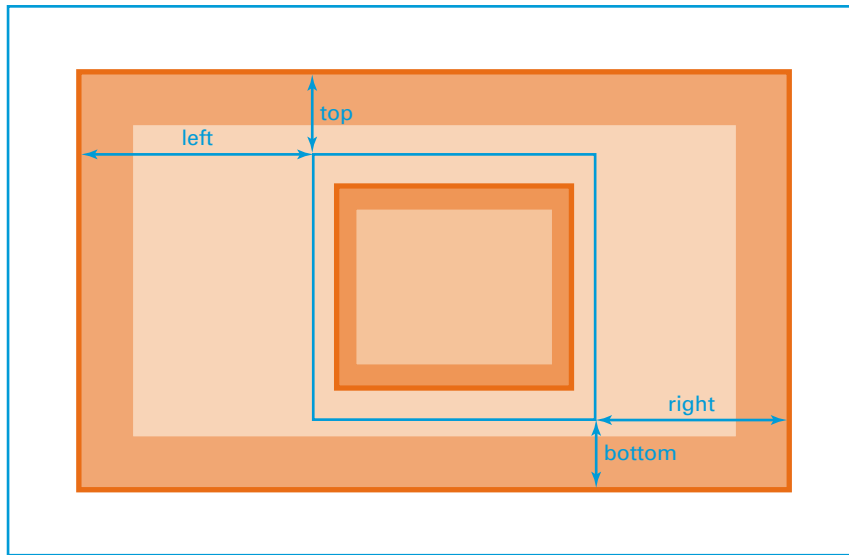


Figure 4.16 Absolute positioning

Effectively, absolute positioning allows you to place an arbitrarily sized box anywhere you want it on a Web page. It would seem, therefore, to be the perfect approach to layout, allowing designers to compose a page from boxes, much as they would when creating page layouts for print in a desktop publishing application such as InDesign. However, the Web is not a print medium, and some consequences of this make absolute positioning slightly less attractive than it might seem at first sight.

There is one technical point that can be easily dismissed. Astute readers will have understood that if `left`, `right` and `width` are all specified, there is a potential for disagreement about how wide the box should be. In such a case, the value of `width` takes precedence: if the language of the page is read from left to right, the value of `right` is ignored; if it is read from right to left, the value of `left` is ignored. Similarly, if the specified values lead to a conflict in determining the box's height, the `height` property is used. Generally, the best way to position a box is using `left` and `top`, essentially providing the coordinates of its top left corner, and then specifying its size, if that is desired, with `width` and `height`.

Thornier issues are raised by the different units in which values for the positions of the box's sides may be specified. These are lengths, so they can be specified in absolute units (such as `px`) or relative units (such as `em`). If absolute units are used, then the position is also absolute,

that is, it is independent of any font size. This might seem to be what is required, but if there are several boxes on a page, their interactions need to be taken into account. Consider, for example, the following stylesheet for a two-column page:

```
div {
    position: absolute;
    width: 350px;
    height: 600px;
}
#leftcolumn {
    top: 0px;
    left: 0px;
    background-color: gray;
}
#rightcolumn {
    top: 0px;
    left: 350px;
    background-color: silver;
}
```

The stylesheet could be used in conjunction with an XHTML document with the following structure:

```
<body>
  <div id="leftcolumn">
    contents of the left column
  </div>
  <div id="rightcolumn">
    contents of the right column
  </div>
</body>
```

The result would be a page with two text boxes side by side, each 350px wide. Because of the fixed width, this layout would be vulnerable to changes in font size. If the font size was changed, either by the designer or the user, any text that had been fitted to the two boxes would reflow. Since the height of the boxes is fixed, any overmatter would spill out by default. This can be avoided by changing the height to auto, but this could cause the boxes themselves to grow taller if the font size was increased, which might result in their overlapping any absolutely positioned elements lower down the page.

These problems can be partially avoided by specifying the boxes' dimensions in em units:

```
div {  
  position: absolute;  
  width: 25em;  
  height: 50em;  
}
```

Now the boxes will grow to accommodate any increase in the font size, but, because the left side of the right column is positioned absolutely at 350px, the left column will overlap the right if it expands, and if it shrinks a gap will appear between the two. Therefore, the position of the left side of the right column must now also be specified in em units, so that it moves to accommodate any change in the width of the left column. Similar considerations apply to the top of any boxes positioned below these two.

As with box dimensions, positions can also be specified as a percentage. In this context, percentages refer to the corresponding dimension of the containing box, so one way of splitting our page into two columns would be to use the rule

```
div {  
  position: absolute;  
  width: 50%;  
  height: auto;  
}
```

with a corresponding change to the left value for the right column. This way, the two columns will expand or contract if the browser window is resized, while maintaining their relationship.

Hopefully, these considerations will seem obvious. However, a common way of using absolutely positioned boxes for layouts consists of drawing the boxes on screen in a visual Web authoring program, so that the designer can position them by eye. Almost without exception, such programs will generate layout rules which combine absolute positioning with absolute units, which will lead to the problems outlined above.

When boxes can be placed anywhere on the page, the possibility of their overlapping arises. (Text does not flow around an absolutely positioned element the way it does around a floated one.) If two boxes overlap, partially or totally, the question arises of which is in front, since whichever it is will obscure the box behind it. The stacking of elements is controlled by the *z-index* property, which may have any integer as its value. This property only applies to absolutely positioned elements. Although the full details of the stacking algorithm are

Technical Detail

Fixed and Relative Positioning

These two variations on the absolute positioning scheme deserve a brief mention.

If the `position` property is set to `fixed`, the box is positioned according to `left`, `top` and so on, with respect to the browser window, irrespective of which part of the page is being displayed. In other words, the box for an element whose position is fixed stays still if the content of the window is scrolled, in the same way as a fixed background image does. Fixed positioning can be used to keep an element such as a navbar in a fixed and easily accessible position on the screen as the user scrolls through a long page.

If `position` is `relative`, `left`, `top`, `right` and `bottom` define offsets from the edges of the box that the element would have occupied if it had been set in the normal flow. One possible use of relative positioning is to move elements into sub- and superscript positions below or above the normal baseline. More interestingly, if an absolutely positioned element appears within a relatively positioned element, it will be positioned relative to the containing block of the latter. Since a relatively positioned element moves as text reflows (because it is offset with respect to its position in the normal flow), the enclosed absolutely positioned box will move too. This can be used, for instance, to place images in the margin that stay aligned with some textual element within the text. (If an element is relatively positioned, but all the offsets have value zero, it remains where it would have been in the normal flow, but can be used as the basis for absolute positioning.)

surprisingly complex, in most cases the behaviour is simple: elements are stacked from front to back in increasing order of `z-index`. The content of the root element effectively has a `z-index` of zero, so to put a box behind that, its `z-index` must be negative. The content of boxes behind the frontmost box will show through any transparent areas; setting a background colour or image will conceal any underlying content.

A little thought will show you that, strictly speaking, absolutely positioned elements are actually positioned relatively, since the `left` and `top` properties specify distances from the

edges of the containing block, which might itself be an absolutely positioned element. Only elements at the outermost level are positioned absolutely with respect to the page.

In case you ever need to specify it explicitly, the normal flow algorithm is used if the `position` property is set to `static`.

Special Cases

Links

The source anchors of links demand special typographic treatment. It is vital that users be able to identify links. The customary way of distinguishing them is by underlining; traditionalists also colour them blue. In XHTML (including XHTML 2), source anchors are always elements with an `href` attribute, so an attribute selector would suffice in a rule that was intended to apply to links. This is not the case in XML, though, so CSS provides a pseudo-class `:link`, which matches any elements that are links. Thus, to enforce traditional link designations in XHTML documents, either of the following rules could be used:

```
:link { color:blue; text-decoration:underlined;}
a:link { color:blue; text-decoration:underlined;}
```

More varied and subtle underlines can be produced by applying a bottom border to linking elements, and using the various possibilities offered by the `border-bottom-style` property. For instance

```
a:link {
  border-bottom: dotted blue 0.15em;
  text-decoration: none;
  color: inherit;
}
```

will cause a subtle blue dotted underline to appear beneath links.

Three other link-related pseudo-classes are also provided; these make possible some effects which cannot be achieved by any other form of selector. The first, `:visited`, matches the source anchors of links that the user has already visited. (The question of how recently a link must have been visited to count as a visited link is an open one. The CSS standard merely notes that after a certain amount of time a user agent may return a visited link to the unvisited state, when it will belong to the pseudo-class `:link` again, not `:visited`.) The following CSS rule causes visited links to be shown in purple, as early browsers always did.

```
a:visited { color:purple; text-decoration:underlined;}
```

(As with `:link`, the element `a` in the selector is redundant in XHTML.)

It has become customary to attach rollovers to links. This is often done using scripting (see Chapter 7), but simple rollover effects can be implemented using CSS. (Actually, with a certain amount of ingenuity, quite elaborate rollovers can be implemented purely in CSS.) The pseudo-class `:hover` matches an element that the cursor is presently over. The standard actually specifies a less specific condition: it matches ‘while the user designates an element (with some pointing device), but does not activate it.’ To take the simplest possibility, a link could turn silver when the cursor was over it (that is, when a click would activate it), if the following rule was used:

```
a:hover { color:silver; text-decoration:underlined;}
```

A rollover effect that has been widely adopted consists of changing the background colour of the link element. Thus,

```
a:hover { background-color: yellow; }
```

will cause a yellow highlight to appear behind the link when the cursor is over it – an effective but relatively subtle way of signifying that something will happen if the mouse button is clicked.

Finally, the `:active` pseudo-class matches elements that are in the process of being activated. That is, for graphical browsers and a conventional input device, the element below the cursor is in this state during the time between pressing and releasing the mouse button. Carrying on with the simple expedient of altering the colour, we could make the link turn black while it was being clicked, like this:

```
a:active { color:black; text-decoration:underlined;}
```

If you provide an explicit rule for `a:link`, it is as well to provide rules for `a:visited` and `a:active` as well, otherwise the default appearance will be used for visited and active links. This may look incongruous in conjunction with your custom unvisited links.

Don’t be tempted to change the font weight or style in rules for `:hover` elements. In almost all fonts, this will change the space occupied by the link on the screen, which will usually

cause other elements to move in a disturbing fashion when the cursor hovers over the link. This will invariably occur if the link is embedded in some text within a paragraph. Even with isolated links (on navbars and so on) the change in size can be disorienting.

Links that occur within the text will usually need different treatment from links in navbars, since what works in isolation will not necessarily work in context. You may not consider it necessary to distinguish visited links in a navbar, for example. To discriminate between links in different contexts, contextual selectors can be used. A collection of links, such as a navbar, is usually marked up as an unordered list; `ul` elements with a distinguishing `class` attribute can be used for any list of links that needs special treatment. Thus, if a navbar is enclosed between `<ul class="navbar">` and `</ul>`, the selector `.navbar a:link` would match navbar links, while `a:link` would match any others, and so on, for visited, hovered-over and active links.

Lists

Because of the way they are laid out, with bullets or numbers and so on, lists provide additional opportunities for styling, and several CSS properties apply only to them. Each item in an ordered or unordered list generates two boxes: a principal box, which holds the content of the item, and a marker box, which holds the bullet or item number. (Definition lists are different, and the remarks in this section do not apply to them. The `dt` and `dd` elements can be styled separately, like any other elements.)

Type	Value	Marker
Glyph	disc	•
	circle	○
	square	■
Numerical	decimal	1, 2, 3,...
	decimal-leading-zero	01, 02, 03,...
Alphabetical	lower-roman	i, ii, iii,...
	upper-roman	I, II, III,...
	lower-latin	a, b, c,...
	lower-alpha	
	upper-latin	A, B, C,...
	upper-alpha	

Table 4.4 Common list style types

The value of the `list-style-type` property determines what goes into the marker box. The most widely supported possibilities are summarized in Table 4.4. (We have omitted some more esoteric numerical schemes, since these are not implemented in most browsers.) The disc, circle and square options provide for bullet lists; the precise appearance of these glyphs is determined by the browsers. The disc is the familiar circular solid bullet, and this is the default. All the other values for this property produce numbered lists using various different numbering schemes. To create lists with square bullets, the following rule could be used:

```
ul { list-style-type: square; }
```

Any of these values can be used with either `ul` or `ol` elements: as far as CSS is concerned there is no distinction between ordered and unordered lists: either can be displayed with bullets or with numbers. However, there remains a structural distinction, which should be expressed in the XHTML code for the benefit of non-visual user agents, and unless there is a good reason to do otherwise, `ul` elements should be styled with bullets of some sort, and `ol` elements with numbers.

If none of the bullet styles is satisfactory, you can use a small image as the list item marker, by specifying its URL as the value of the `list-style-image` property, like this:

```
ul { list-style-image: url(Tiny-Logo.gif); }
```

This property allows you to use arbitrary bullets, but there are inevitably pitfalls. The image is never scaled, so it must be prepared at the correct size to match the text of the list items before it can be used in this way. If the user changes the font size in the browser, the list item markers will stay the same size, no longer matching the text they accompany.

Tables

Tables offer special formatting opportunities. As we saw in Chapter 3, the default layout of tables is ugly and hard to read. A great deal can be done to improve on it. We will sketch the basics, which should be enough to enable you to produce decent-looking tables for your Web pages.

Each type of element involved in building tables – `table`, `caption`, `tr`, `td` and `th` – generates a box, which can therefore be styled like any other: it can have margins, padding, borders and a background, and the text within it can be aligned and styled. These properties alone can make a big difference to the appearance of tables.

Consider the example in Figure 3.5 from Chapter 3. We can approximate the layout of tables in this book using a few simple rules. To begin with, the text in headers and data cells should be left-aligned:

```
th,td { text-align: left; }
```

Setting the fonts and colours does not introduce any new CSS:

```
th {
  color: navy;
  font: bold 1.1em Univers,Verdana,sans-serif;
}

td {
  color: black;
  font: 1.1 em Goudy,Georgia,serif;
}
```

(Navy isn't the right shade of blue, but it will do until we have described how to choose colours more accurately in Chapter 5.)

Besides controlling the typography of the contents of table cells, CSS rules can be used to provide them with backgrounds, just as they can for other elements. The lines that are often used to separate rows and columns of tables can be created using border properties on the elements making up the table, but their behaviour must be modified slightly to create the layouts usually preferred for tables.

Suppose that, instead of the sparser layout used in this book, you wished to create a table with a complete grid of lines separating each of the rows and columns from its neighbours. A little thought would suggest that you could use the following rule:

```
th,td { border: navy medium solid; }
```

	kpbs (theoretical)	6kB text page	100 kB text page	4 MB movie
Modem	56	1 sec	15 secs	10 mins
ADSL (typical)	512	0.1 sec	2 secs	1 min

Figure 4.17 Separated borders

	kbps (theoretical)	6kB text page	100 kB text page	4 MB movie
Modem	56	1 sec	15 secs	10 mins
ADSL (typical)	512	0.1 sec	2 secs	1 min

Figure 4.18 Collapsed borders

The result, shown in Figure 4.17, is probably not what you would expect. The browser has applied CSS's *separated border* model to the table, adding a little space between each cell's box, so that each has its own border. If that is actually what you want to do, the amount of inter-border space can be controlled using the `border-spacing` property. More often, though, the table should use the collapsing border model, where the borders for adjoining cells are combined into a single separator. The border model is controlled by the `border-collapse` property, whose value can be `separate` (the default) or `collapse`. This property only sensibly applies to an entire table. Adding the rule

```
table { border-collapse: collapse; }
```

would cause the sample table to display as shown in Figure 4.18.

This is not actually what we set out to do: the original intention was to simulate the table layout of the book. This uses no vertical separators, but has thick lines above and below the header and at the very bottom of the table, and thin separators between the rows. Since we are only concerned with separating entire rows, we can use the borders of `tr` elements as the separators. To distinguish the top and bottom rows, it is necessary to add some `id` attributes to the XHTML markup.

```
<table>
  <tr id="heading">
    heading cells
  </tr>
  other data rows
  <tr id="bottomrow">
    data cells for last row
  </tr>
</table>
```

In the stylesheet, we must remove the rule we added earlier for the `border` property with selector `th,td`. Now we can add rules to create the required separator lines.

MACAVON MEDIA LECTURER'S COPY

	kbps (theoretical)	6kB text page	100 kB text page	4 MB movie
Modem	56	1 sec	15 secs	10 mins
ADSL (typical)	512	0.1 sec	2 secs	1 min

Figure 4.19 Applying borders to rows

	kbps (theoretical)	6kB text page	100 kB text page	4 MB movie
Modem	56	1 sec	15 secs	10 mins
ADSL (typical)	512	0.1 sec	2 secs	1 min

Figure 4.20 Adding extra padding to the cells

```
tr {
    border-color: navy;
    border-width: thin;
    border-style: none none solid none;
}

#bottomrow {
    border-color: navy;
    border-width: medium;
    border-style: none none solid none;
}

#heading {
    border-color: navy;
    border-width: medium;
    border-style: solid none solid none;
}
```

The result is shown in Figure 4.19. The lines and fonts are correct, but as you can see the spacing remains unacceptable. The table cells are just boxes, though, so we can add some padding. The rule

```
th,td { padding: 1em; }
```

is sufficient to produce a readable and fairly attractive table, as shown in Figure 4.20.

Technical Detail

Conflicting Borders

If the collapsing border model is used, most of the horizontal separators can be considered to be both the bottom border of one row and the top border of the one below it. What happens if two different CSS rules apply to these borders? The standard specifies the exact rule for resolving such conflicts. The general idea is that the most eye-catching border is used, so thick borders are used in preference to thinner ones, solid to dashed, and so on. If the only disagreement is between colours, then the colour specified for a cell takes precedence over that for a row. (If two adjoining cells disagree as to the colour of their shared border, the result is undefined.)

Padding is usually a good way to space out tables for readability, but precise control over cells' widths is possible using the `width` property. The CSS standard allows some latitude in how table widths are calculated. It is safest to assume that the widths of the cells in the first row may be used to set the width of each column and to work out how wide the whole table is, so it is unwise to leave out any cells in that row. If an explicit width is set for all the cells, the table width will be equal to their sum; if a width has also been specified for the whole table, and this is larger than the sum of the cell widths, the extra space will be evenly divided between columns.

The height of each row can also be specified: it will be the maximum of the height of the cells in that row and any explicit height set for the row as a whole.

A table element can include a caption. This can of course be styled like any other element, and it is usual to apply some distinctive typography to captions. It is also possible to control the positioning of the caption with respect to the table, using the `caption-side` property. This takes the value `top` or `bottom`, to place the caption above or below the body of the table, respectively. All the captions in a document could be set in a suitable font and positioned a short way below their tables with the following rule:

```
caption {
  color: black;
  font: italic bold 1.1em Goudy,Georgia,serif;
  padding: 1em 0em;
  text-align: left;
```

```
caption-side: bottom;
}
```

Forms

XHTML forms are among the most difficult of elements to format satisfactorily, because control elements are often implemented by browsers using the corresponding objects from the host operating system's user interface. These are not amenable to styling. Even browsers which use their own platform-independent controls rarely allow their appearance to be controlled by stylesheets. The 2.1 revision to the CSS specification is explicit about this:

'CSS2.1 does not define which properties apply to form controls [...] or how CSS can be used to style them. User agents may apply CSS properties to these elements. Authors are recommended to treat such support as experimental.'

Thus, the Web designer is faced with the problem of incorporating these given elements into a page design. Their positioning offers no special difficulties: controls may be placed in `div` and `span` elements, floated or absolutely positioned. Altering the fonts that they use, or fixing their size or colour, are jobs that cannot be done reliably by CSS. Labels, however, can be treated like any other textual element, and styled and positioned however you like.

A special pseudo-class selector, `:focus`, is available for those browsers that do support styling of form controls. An element belongs to this class when it is being used for input. For example, when the cursor is placed in a text input field, either by moving the mouse or tabbing, that field is said to have the focus. Browsers usually indicate this fact by highlighting the field in some way. If you wish to customize the highlighting, you can do so by using `:focus` as a selector in a rule (on the understanding that it might have no effect on some browsers).

A set of properties that can be profitably used in such rules control the presence and appearance of an outline around an element. An outline is much like a border, except that it is placed over the element, without occupying any space itself. This means that turning an outline on and off does not cause any page content to reflow. Because of the similarity between outlines and borders, the set of properties affecting outlines has the same structure as the border properties we met earlier. The following rule puts a thick red line around the element that has the focus (providing the browser supports it):

```
:focus { outline: thick solid red; }
```

Advanced Features of CSS

Despite the excessive length of this chapter, we have by no means described every feature of CSS. Enough has been said to allow you to control the appearance of Web pages to a considerable extent, and in Chapter 10 we will explore some of the design possibilities that this opens up. A few more CSS features that will be needed therefore require a brief description.

Generated Content

The content property creates new text to be displayed on a page. Its value may take several forms, some of which we will describe. Most frequently, it is simply a string (written between double quote marks, with `\` being used to represent such marks within the string). This property can only be used with any effect in conjunction with the pseudo-elements `:before` and `:after`, whose meaning is best conveyed by a simple example.

The effect of the following rule is to insert the word **NEW** in red (followed by a space) before every paragraph belonging to the class `new`.

```
p.new:before { content: "NEW "; color: red; }
```

The selector applies the rule to any inserted content that appears before a `p` element with `class="new"`. The declaration of the content property specifies what this generated content should be. Any further declarations in the rule will be applied to this content, so the effect is as originally described. In a similar way, selectors using `:after` apply to generated content placed after any element matched by the rest of the selector; the content property must be declared in the rule if there is to be anything to match.

The generation of content by rules such as this takes place after the XHTML has been parsed, so it is not possible to include any markup in the string that is the value of content. For instance,

```
p.new:before { content: "<span class=\"prefix\">NEW </span>"; }
```

doesn't assign the generated content to the class `prefix`, so that you can apply further styling to it; it just inserts the string `<span class="prefix">NEW </span>` verbatim in front of every new paragraph. Any styling that is to be applied to generated content must be declared in the rule that generates the content. Furthermore, the generated content becomes part of the element, so that any rules that apply to that element will also be applied to it.

As a consequence, the `:first-letter` pseudo-element will match the first letter of any inserted string, not the first letter of the original element.

The `content` property may also have a URL as its value, which typically will identify an image to be inserted before or after some element. For instance, if the URL `icon/new.gif` pointed to a small image of an icon used to indicate new items, the following rule could be used to insert it before all new paragraphs:

```
p.new:before { content: url(new.gif); }
```

This property can also be used to insert the values of attributes of the element it applies to, using the form `attr(attribute name)` as its value. Sometimes, Web pages will have links which are simply the text of the URL to which each link points. (Sometimes there really is nothing better.) To avoid the necessity of retyping the URL, and the consequent risk of errors and inconsistency, the following rule could be used to pull the value out of the `href` attribute of links belonging to the class `selfref`:

```
a.selfref:before { content: attr(href); }
```

One of the most useful applications of generated content is in the insertion of opening and closing quotation marks around a quotation, marked up as a `q` or `blockquote` element. If the quotation marks appear in the marked-up document, the possibility of adopting a convention for displaying quotations that does not use them (for instance, italicizing or setting as an indented block) is no longer available, since CSS provides no way of deleting text from a document. For maximum flexibility, quotation marks should be inserted by a stylesheet. CSS provides some special values for this case: `content` may be `open-quote` or `close-quote`, so a rule for setting inline quotes using conventional punctuation would be:

```
q:before { content: open-quote; }
q:after { content: close-quote; }
```

Actually, this is what browsers do by default, but using the explicit rule provides greater freedom, since additional styling can be applied, as with any other generated content, and the glyphs to be used for opening and closing quotes can be specified using the `quotes` property. For example, to use single curly quotes instead of the default double straight quotes throughout a document, the following rule could be used:

```
body { quotes: "\002018" "\002019"; }
```

(In a CSS string, a `\` followed by a hexadecimal number stands for the character which has that number as its Unicode character code.)

Finally, generated content can include automatically incremented counters, for generating sequential numbers for elements other than list items. A stylesheet may use any number of named counters. For instance, if we wished to add numbered captions to all our tables, we might use a counter called `tablenum`. The expression `counter(tablenum)`, when used as the value of the `content` property, produces the value of this counter. A second argument may be supplied, in the form of any of the possible values for the `list-style-type` property. If such an argument is present, it is used to format the counter. For example, in principle `counter(tablenum, upper-latin)` should produce a Roman numeral corresponding to the counter's value (though no browser available to us at the time of writing managed to do so).

None of this is much use unless the value can be changed. The `counter-increment` property has a name as its value, and increments the corresponding counter. By default, it adds one; if an integer follows the counter name, it is added to the counter. The `counter-reset` property is used similarly to set the counter to an initial value. If no value is given, the counter is set to zero. If a rule uses a counter's value in a declaration for `content`, and also increments it, the increment is performed first, irrespective of the order the declarations appear. Hence, to add numbered captions to the tables on a page, starting at 7 (perhaps the first six were on a previous page), the following rules will suffice:

```
body { counter-reset: tablenum 6; }
caption:before {
  content: "Table " counter(tablenum) ". ";
  counter-increment: tablenum;
}
```

(This example also illustrates how string values may be concatenated in declarations.)

Specifying Box Types

It is usually clear which XHTML element types should be blocks and which should be inline: `p` and `blockquote` should obviously generate blocks, `em` and `strong` elements should be displayed inline. Sometimes, though, it may be desirable to change the default display for certain elements. You might ask, why not use a different element type that displays the way you want it? The answer is that element types should be chosen to reflect the semantic structure of the document, not to specify its appearance, so the choice of element type

Figure 4.21 Breadcrumbs

comes first, and is dictated by the meaning of its content; the styling comes afterwards. Of course, since XHTML has only a limited repertoire of element types, it is not always clear which one should be used for a specific purpose that is not directly catered for, so although choosing element types to reflect semantics is a good rule, it cannot be an exhaustive guide to marking up a document.

To illustrate both these points, consider the feature known as ***breadcrumbs***. Breadcrumbs provide a display of the route through a hierarchically organized site that leads to the page in the browser window. Figure 4.21 shows a typical style of display for breadcrumbs. What markup should be used? There is no `breadcrumb` element in XHTML. However, a breadcrumb trail is like a list, itemizing the steps to the current page, so the most apposite markup is an ordered list:

```
<ol>
  <li>Web Design</li>
  <li>Accessibility</li>
  <li>Graphics</li>
  <li>Colour</li>
  <li>Colour Blindness</li>
</ol>
```

(The list items would normally contain links, but we have omitted the anchors in the interests of clarity.) This expresses the structure and will work for non-visual user agents, but displaying the trail as a numbered list is not very perspicuous, and, with each item on a separate line, occupies more screen space than the feature deserves. To obtain the preferred display of Figure 4.21, the list items must be displayed inline. This can be done using a declaration for the property `display`. This may have the value `inline` or `block`, with the expected effect. The full collection of rules for producing the breadcrumb display as illustrated is as follows:

```
li { display: inline; }
li:before { content: ">"; }
li:first-child:before { content: ""; }
```

The first rule turns the items into inline elements, the second inserts > signs in front of each element, while the last rule suppresses the > before the first item.

As well as `inline` and `block`, `display` can take a range of other values that cause arbitrary elements to be displayed as list items or the various components of tables. These are mostly of little use in the context of XHTML. However, for XML documents, which give no *a priori* meaning to any element type, the use of `display` is vital.

One other value that may be of occasional use is `none`. This causes any elements to which it applies to generate no box at all. The element cannot be seen and it does not affect the layout of the rest of the document. Visually, it is as if it was not there at all, although the markup is still in the document and available to non-visual user agents.

Media-Specific Stylesheets

Most of the time, Web pages are displayed in a graphical Web browser, but not always. Sometimes, visitors to a site will want a hard copy of some page, so they will print it. Formatting that is suitable for display on a screen will not necessarily be appropriate on paper. For instance, type is usually set in a larger size on screen, and sans serif fonts are generally preferred for readability at low resolution, but at higher resolution on paper, a smaller, serifed font may be more readable. A layout that makes good use of screen space may not work well on a sheet of paper, with its different aspect ratio. Page breaks can be profitably used to break up the printed version of an XHTML document at suitable places, whereas the same document will be displayed continuously in a browser window.

Conventional computer screens and print are not the only potential destinations for a document. Other possibilities include Braille devices, speech synthesizers, TV screens, handheld devices with small screens and data projectors. Each has its own distinctive capabilities and limitations. It is unrealistic to expect a single stylesheet to produce a layout that will work for all output devices and media. Instead, if it is important that a document be displayed optimally on several types of device, it is necessary to provide alternative stylesheets. For instance, the following rule will produce nice clear text on a neutral background on a monitor:

```
body {  
    background-color: silver;  
    color: black;  
    font-family: Arial, Verdana, sans-serif;  
    font-size: 18px;
```

```
    line-height: 1.6;
}
```

Text styled according to this rule would look too big on paper, and unless the browser was intelligent enough to remove the background when printing (most are) it would waste a tremendous amount of ink.

This rule, on the other hand, would produce compact, but still clean and readable, text on paper:

```
body {
    background-color: white;
    color: black;
    font-family: Times, Georgia, serif;
    font-size: 12px;
    line-height: 1.2;
}
```

What is required is some way of specifying which rule should be applied to which medium. CSS provides one mechanism; XHTML another.

In a CSS stylesheet, rules can be grouped using a special construct, the `@media` rule. (This is an example of a more general class of CSS construct, the `@-rule`. We will not describe any others; there are only a few in the current version of CSS, and they are not generally useful.) The keyword `@media` is followed by a list of media types, separated by commas, and then

all	All devices
braille	Refreshable tactile Braille displays
embossed	Braille printers
handheld	Handheld devices (PDAs etc)
print	Printers
projection	Data projectors
screen	Computer displays
speech	Speech synthesizers
tty	Teletype-like devices, using fixed-width text
tv	Television screens

Table 4.5 Media types

some ordinary CSS rules, enclosed in curly brackets. The implication is that those rules only apply when the document is displayed on the named media. Table 4.5 lists the currently recognized media types. We could create a stylesheet that formatted text differently for screen and print along the following lines:

```
@media screen {
  body {
    background-color: silver;
    Other declarations for body on screen
  }
  Other rules for screen display
}
@media print {
  body {
    background-color: white;
    Other declarations for body on paper
  }
  Other rules for printed output
}
Rules for both media
```

This mechanism allows the rules for all media to be kept in a single stylesheet. If you are happy to maintain separate stylesheets for different media, it is possible to use the `media` attribute of the `link` element when referencing an external stylesheet, to indicate the media for which it is appropriate. If, for example, the file `print-styles.css` held a collection of rules for print formatting, and `screen-styles.css` a collection for on-screen display, they could be referenced in a document as follows:

```
<link rel="stylesheet" href="screen-styles.css" media="screen" />
<link rel="stylesheet" href="print-styles.css" media="print" />
```

The browser would then use the appropriate stylesheet for display and printing.

CSS includes special properties for use in stylesheets for media that break text into pages (print and projectors, for example), that allow the designer to specify page breaks and paper sizes. It also includes an extensive sub-language for aural stylesheets, with properties that control the volume, voice, stress and other properties of spoken text synthesized from the document. Pauses and cues can be inserted, and the position of voices in a 3D surround-sound space can be controlled.



Key Points

Stylesheets are used to specify appearance independently of structure and content.

Rules in CSS

A stylesheet is a set of rules, each consisting of a selector and some declarations. Each declaration sets the value of a property.

The declarations in a rule with an element name (e.g. `h1`) as the selector are applied to all occurrences of that element.

A selector consisting of an element type followed by a dot and a name (e.g. `p.first`) matches all elements whose `class` attribute includes that name.

The selector `*` matches any element. It can be omitted in selectors of the form `*.c`, where `c` is a class name.

A selector consisting of an element name followed by a `#` (e.g. `p#summary`) and a name matches the element whose `id` is equal to the name. The element name can be omitted.

A selector consisting of two element names separated by a space (e.g. `ul li`), matches elements with the second name in the content of an element with the first name (e.g. `li` elements inside a `ul` element).

A selector consisting of two element names separated by a `>` (e.g. `ul>li`), matches elements with the second name that are children of an element with the first name (e.g. `li` elements immediately inside a `ul` element).

A selector consisting of two element names separated by a `+` (e.g. `h1+h2`), matches elements with the second name that follow an element with the first name (e.g. `h2` elements after an `h1` element).

The pseudo-class `:first-child` matches the first child of an element.

A rule consisting of a sequence of selectors separated by commas followed by some declarations is equivalent to a sequence of rules, one for each of the selectors, with the same declarations.

A value that is assigned to some property for an element is inherited by that element's children, unless a rule causes a value to be explicitly assigned.

Typography

A font is a collection of character shapes, called glyphs.

The `font-family` property has a list of fonts as its value; the browser will use the first in the list which is installed on the user's system.

The generic font families `serif`, `sans-serif`, `cursive`, `fantasy` and `monospace` provide fallbacks in case no font named in the list is available.

The `font-style`, `font-variant`, `font-weight` and `font-stretch` properties control other aspects of type's appearance.

The `font-weight` property may take numerical values between 100 and 900, or the keyword values `normal`, `bold`, `bolder` and `lighter`.

The `font-size` property may take the values `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, `smaller` and `larger`.

`medium` is the default size set by the user in their browser.

Font size can also be specified in absolute units (including `pt` and `px`) or relative units (`em`, `ex` and `%`).

Relative units are calculated with respect to the inherited font size.

The `line-height` property is used to specify leading.

Relative units for `line-height` are calculated with respect to the current `font-size` value.

The `color`, `font-decoration` and `text-transform` properties modify the appearance of text.

The `text-align` property takes the values `left`, `right`, `center` or `justify` to control the alignment of text in paragraphs and other block elements.

The `text-indent` property controls indentation of the first line of a paragraph. Its value may be negative for extended first lines.

The pseudo-classes `:first-line` and `:first-letter` are used to apply special styling to the first line and letter of a paragraph.

Layout

Every element is placed inside a box.

Each box is divided into content, padding, border and margin regions.

Unless explicit positioning is specified, boxes are laid out using the normal flow algorithm.

Text and inline elements are folded into lines so that they fit within the content area of their containing box.

Box widths can be specified in absolute or relative units with the `width` property.

`em` units cause the width to change with font size.

`%` units refer to the width of the containing box.

Box heights can be set using the `height` property. The `overflow` property determines what happens to any over-matter.

The thickness of padding can be set to an absolute or relative length using the `padding-top`, `padding-right`, `padding-bottom` and `padding-left` properties, or the shorthand padding property.

Border and margin thicknesses are controlled in a similar way.

The border thicknesses may use the keyword values `thick`, `medium` and `thin`.

Border colour and style may also be specified for each side, using similar properties.

The style, width and colour of the border on any side can all be set at the same time, using `border-top` etc, or the shorthand border property.

The top and bottom margins of vertically adjacent boxes are collapsed.

The background of every element's box can be set to a colour or to an image, using the `background-color` and `background-image` properties.

The value for `background-image` is written as `url(the url)`.

Tiling of background images can be specified or prevented using the `background-repeat` property.

Positioning

Setting the `float` property to `left` or `right` causes an element's box to be taken from its position in the normal flow and moved as far left or right as possible.

If the `position` property is set to `absolute`, `left`, `right`, `top` and `bottom` can be set in order to determine the position of the four sides of the box.

The positions of the sides can be set in absolute or relative units.

The stacking of overlapping absolutely positioned elements is controlled by the `z-index` property.

Special Cases

The pseudo-class `:link` matches any elements that are links, in particular, a elements in XHTML documents.

The pseudo-class `:visited` can be used to style recently visited links.

The pseudo-class `:hover` matches an element when the cursor is over it. It can be used to implement rollovers in CSS.

The `:active` pseudo-class matches elements that are in the process of being activated.

The `list-style-type` property is used to specify markers (bullets or numbers) for list elements.

The `list-style-image` property allows you to choose a small image as a list item marker.

The padding, margins and borders of table cells can be specified using the appropriate CSS properties applied to `td`, `th` and `tr` elements.

The `border-collapse` property must be set to `collapse` if spaces between cells are not required.

CSS styling cannot be applied to form control elements reliably.

Advanced Features of CSS

The `content` property creates new text to be displayed.

The pseudo-elements `:before` and `:after` cause generated content to appear before or after a matching element.

The `display` property can be used to change the type of an element from block to inline or vice-versa.

Setting `display` to `none` causes the element to generate no box.

`@media` rules allow different rules to be applied to different media.

Exercises

Test Questions

1. Find the syntactic errors in the following stylesheet rule:

```
h1: { color = "red" }
```

2. Suppose a stylesheet includes the following rules:

```
.red {color: red;}
.green {color:green;}
p,.orange {color:orange;}
```

What colour will the word colour appear in when each of the following is displayed in a browser that correctly interprets the stylesheet?

```
<p>What colour is it?</p>
<p>What <span class="red">colour</span> is it?</p>
<p class="red">What colour is it?</p>
<p>What <span class="orange red">colour</span> is it?</p>
```

3. Show how you can use `class` attributes instead of contextual selectors to make items in an ordered list a different colour from those in an unordered list. Which is better?
4. If the `font-weight` property has the value `bold`, will the chosen font be a boldface?
5. Do the following two rules have the same effect?

```
* { font-size: 14px; }
body { font-size: 14px; }
```

6. What is the minimum set of font properties that must be specified in a font declaration?
7. Write a pair of CSS rules that set the first paragraph after each `h1` with its first line in bold and a red double-height initial letter.

8. Why would you not use a rule with a as its selector to style links?
9. If you specify the width of a box in ems, what units should you use for its height?
10. Write a rule or rules to automatically number all the paragraphs in a document, with a prefix of the form Para 23, and so on.

Discussion Topics

1. Adobe's Portable Document Format (PDF) allows any document to be displayed with all its formatting intact, no matter how it was originally created. PDF also supports hypertext links and embedded images and time-based media. Does PDF, therefore, provide a superior alternative to standard Web formats for hypermedia on the Internet?
2. Large CSS stylesheets tend to become disorganized and unwieldy, since the only structuring facility in CSS is the cascade. What conventions could be adopted for keeping stylesheets organized and maintainable? Are there any language features that could be added to CSS to help with organizing stylesheets?
3. Explain why the idea of a WYSIWYG (what you see is what you get) editor for Web pages is misconceived.

Practical Tasks

1. Add a CSS stylesheet to the résumé you created as an XHTML document in Chapter 3, to format it in a neat and attractive way that will impress a prospective employer.
2. Create two alternative stylesheets for the same résumé page, one that would be suitable for reading on screen by people with poor eyesight, the other to present yourself on a social networking site. (If you feel it necessary to suppress some of the information for one or other of these versions, do so using the stylesheet, not by editing the XHTML document.)

3. Rectangles with rounded corners are popular in some forms of graphic design. The top halves of such rectangles are often used as tabs, for example on navbars. Although CSS3 will include support for rounded corners, current Web standards do not. Devise methods of setting blocks of text in rounded rectangles. (You may need to add semantically superfluous markup to achieve the desired result.)
4. The `blockquote` element is used to markup passages that are quoted from some other source. Research different ways block quotes are typeset in books and magazines, and how they are presented on Web pages. Write CSS stylesheets to display `blockquote` elements in many different styles.

