Understanding dynamic positioning

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is designed to assist Web developers who need tight control over the placement of content on their pages, but who understand the need to separate content from presentation in order to improve compliance with standards and ease of maintenance.

The tutorial assumes that you are already comfortable using HTML and Cascading Style Sheets (CSS). Basic knowledge of JavaScript is helpful, but not required, for understanding the scripting examples. You can gain a thorough understanding of the topic without trying out the scripting examples. (See Resources on page 36for tutorials that can get you up to speed on JavaScript and CSS.)

What is this tutorial about?

Dynamic positioning using DHTML provides complete flexibility in the placement of content on a page. However, using it can be like using a professional-grade camera -- to get the most out of it, you need to understand its nuances. Building pages using dynamic positioning requires an understanding of the normal page flow and the properties that can alter that flow or even remove items from it altogether. It also requires an understanding of the internal workings of individual items and the interdependencies of various properties.

This tutorial takes you through an introduction to the normal flow of a page, and explains how it can be altered to suit your purposes. It also shows you the details behind the process of laying out a DHTML page using dynamic positioning, including the different types of elements and how to use them to create a page that behaves just as you planned. Finally, this tutorial takes a brief look at scripting CSS properties related to positioning to give you the foundation you need to build your own scripts.

Tools

This tutorial will help you understand the topic even if you only read through the examples without trying them out. If you want to try the examples as you go through the tutorial, make sure you have the following tools installed and working properly:

* A text editor: DHTML files are simply text. To create and read them, a text editor is all you need.
* Microsoft Internet Explorer 5.5, or other CSS2-capable browser: Not all browsers are able to exploit all of the capabilities of dynamic positioning. The examples in this tutorial use IE 5.5, but other alternatives are listed in Resources on page 36.

About the author

Nicholas Chase has been involved in Web site development for companies including Lucent

Technologies, Sun Microsystems, Oracle Corporation, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, and an Oracle instructor. More recently, he was the Chief Technology Officer of Site Dynamics Interactive Communications in Clearwater, Florida. He is the author of three books on Web development, including *Java and XML From Scratch* (Que). He loves to hear from readers and can be reached at *nicholas@nicholaschase.com* .

Section 2. Principles of page layout

The normal flow

Under normal circumstances, a Web author doesn't need to worry about how the browser goes about laying out the page: The browser simply takes each item and lays it out in the next available space on the page.

That "next available space" depends on a number of factors, including the size of the item, the size of the containing block, and the type of item. For example, block-level elements (such as `h1` and `p`) appear on a line by themselves, so the browser starts a new line before placing them.

For years, Web authors used their knowledge of how the browser constructs the flow to build attractive and functional pages. Unfortunately, this flow is ultimately dependent on the size of the browser window, which can be unpredictable. For several years authors have used constructs such as tables to attempt to control the size and location of their content, with some measure of success.

Unfortunately, these techniques can result in pages that are complex and inaccessible to those attempting to access the material from nontraditional browsers, and they completely ignore the goal of separating content from presentation. These techniques can also be a maintenance nightmare. In any case, there were still many effects that could not be accomplished without additional functionality.

Dynamic positioning

Dynamic positioning using Cascading Style Sheets allows Web authors to precisely control their content and where it appears on the page. In addition to directly specifying the size of items, authors can offset them from their original locations in the normal flow, or remove them from the flow altogether and place them in a specific location.

Dynamic positioning provides several advantages over simply manipulating the normal flow of the page:

* The actual position of content can be determined with precision. Browser window size is no longer a constraining factor, though good design mandates that it be taken into account.
* The content does not have to be distorted to fit into complex table structures.
* Content can be rendered visible or invisible, allowing for dynamic effects that lend themselves to scripting.
* Content can be layered, so that more than one item appears in the same location on a page.

These are just a few of the advantages of dynamic positioning. Dynamic positioning allows Web authors to build content that is attractive and predictable, and also has the potential to increase usability (as menu-like structures can be built within pages), decreasing the number of times users must click to reach their goal.

Understanding dynamic positioning requires an understanding of the different types of

content that may appear on a page, such as blocks, inline content, and floats.

Block elements

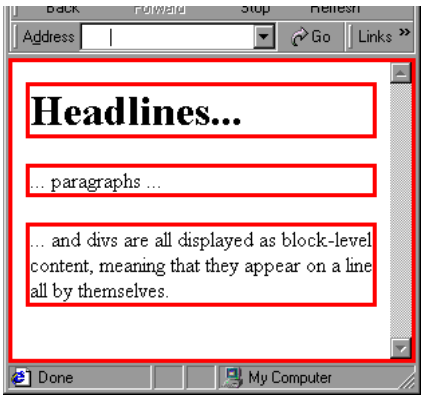
Every page is, at its heart, a block of content. This block also contains other content that may itself be made up of blocks or other content. Block elements are distinguished from other elements in two ways.

One major characteristic of blocks is that they are stacked on the page vertically, with each block appearing below the block that precedes it, even if it appears that there is sufficient room on the line for the new content.

HTML elements that are displayed as block elements include headers, paragraphs, and divs:

```
<html>
 <head><title>Block-level content</title>
    <style type="text/css">
       * { border: 1px solid red; }
    </style>
 </head>
 <body>
    <h1>Headlines...</h1>
    <p>... paragraphs ...</p>
    <div>
       ... and divs are all displayed as block-level
       content, meaning that they appear on a line
       all by themselves.
    </div>
 </body>
 </html>
```

As shown below, each of these blocks of content appears on a line by itself:



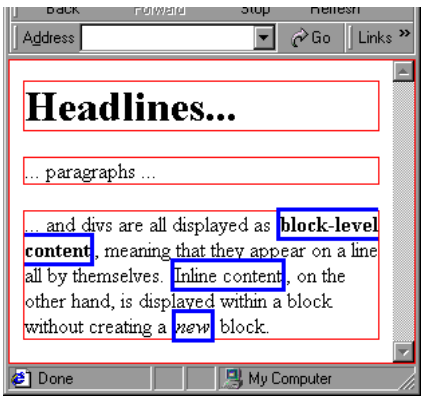
Notice that even the page itself is considered a block.

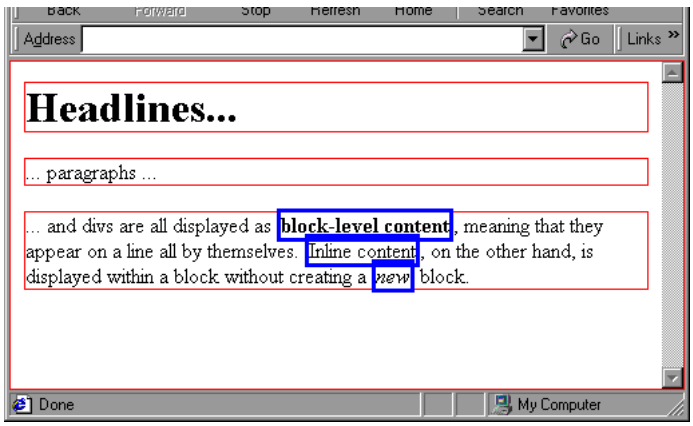
Inline elements

A second type of content, which always appears within a block, is known as *inline content*. Inline content items are rendered next to each other as long as there is room on the line. When there is no more room on the line, an inline item may be converted into two inline items, with the second appearing on the next line.

In other words, an inline item doesn't force the start of a new line the way a block-level element does. Inline elements are typically used for formatting in traditional HTML (using tags such as b and i), though they can also be used to provide information or styling properties (using span):

```
<html>
 <head><title>Inline content</title>
    <style type="text/css">
       * { border: 1px solid red }
      b, span, i { border: 3px solid blue }
    </style>
 </head>
 <body>
    <h1>Headlines...</h1>
    <p>... paragraphs ...</p>
    <div>
        ... and divs are all displayed as <b>block-level content</b>, meaning that they appear
        on a line all by themselves. <span class="definedTerm">Inline content</span>, on the
        other hand, is displayed within a block without creating a <i>new</i> block.
    </div>
 </body>
 </html>
```





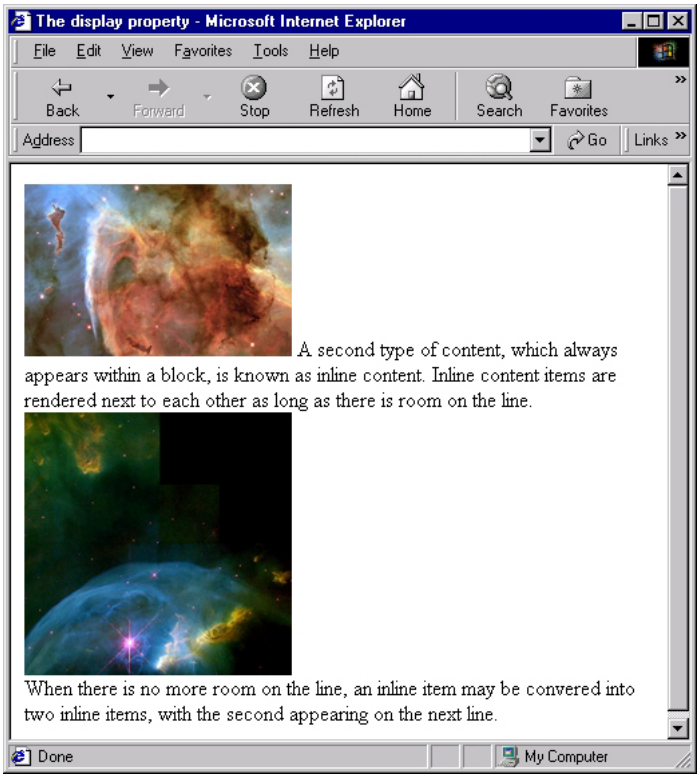
The exact layout of inline content can vary wildly depending on the width of the block in which it lives.

Controlling block vs. inline display

Whether an element appears inline or becomes a block can be determined by the display property. Every HTML element has an intrinsic value for display, but it can also be altered using style sheets. The img tag is normally displayed inline, but can be made into a block

element, as seen below:

```
<html>
 <head><title>The display property</title>
 <style type="text/css">
    #nebula { display: inline }
    #bubble { display: block }
 </style>
 </head>
 <body>
 <img id="nebula" src="nebula.jpg" />
 A second type of content, which always appears within a block, is known
 as inline content. Inline content items are rendered next to each other
 as long as there is room on the line. <img id="bubble" src="bubble.jpg" />
 When there is no more room on the line, an inline item may be converted
 into two inline items, with the second appearing on the next line.
 </body>
 </html>
```



Notice that the nebula image shares the first line with text because it is designated as `inline`, but the bubble image forces a new line before and after because it is designated as a `block`.

Other display values

In addition to `block` and `inline`, the `display` property can take more than a dozen values, each serving a different purpose.

Removing content

Setting the `display` property to `none` ensures that the element creates no box within the flow. The layout is completely unaffected by the element. Children of the element inherit this value and cannot override it.

Table-related values

These are the default values for their corresponding HTML elements, such as `table`, `tr`, and so on, and can be used to mimic the behavior of those elements. They include:

* `inline-table`
* `table-row-group`
* `table-header-group`
* `table-footer-group`
* `table-row`
* `table-column-group`
* `table-column`
* `table-cell`
* `table-caption`
* `table`

Other values

Other values serve their own purposes:

* `list-item:` This value causes the element to mimic the `li` element.
* `marker:` This value designates content generated with the `:before` and `:after` psuedo-elements as a marker.
* `compact:` This value allows an author to indicate that content should appear in the margin of a block. If it doesn't fit in the margin, the browser displays the block on the next line.
* `run-in:` This value generally renders as inline if the element is followed by a block element. If not, it renders as a block.
* `inherit:` This value instructs the browser to use the `display` value of the element's parent.

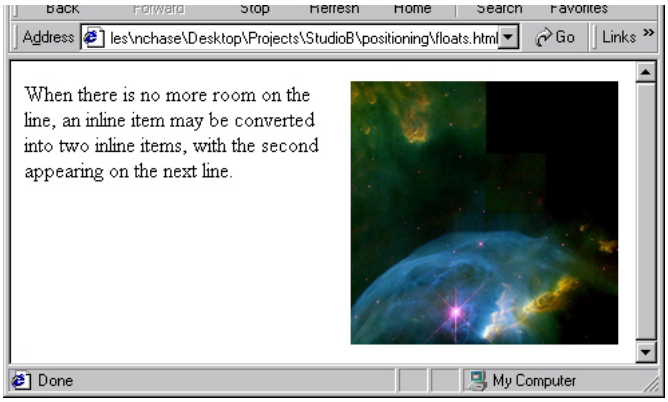
Floats

Floats, or elements that have been floated, combine some of the characteristics of both block and inline elements. A floated element is initially laid out according to the normal flow of the page, but it is then floated to either the right or the left until its outer edge touches the edge of its containing block. (Determining the containing block involves several factors. See The containing block on page 22for more information.)

In addition to their positioning, floats differ from normal block elements in that content can flow along their side. For example, even though an unfloated paragraph and an image that's floated to one side are both block-level elements, they can exist next to each other:

```
<html>
 <head><title>Floated Items</title>
 <style type="text/css">
    #bubble { float: right; }
 </style>
 </head>
 <body>

 <img id="bubble" src="bubble.jpg" />
 <p>When there is no more room on the line, an
 inline item may be converted into two inline items, with the second appearing on the next
```

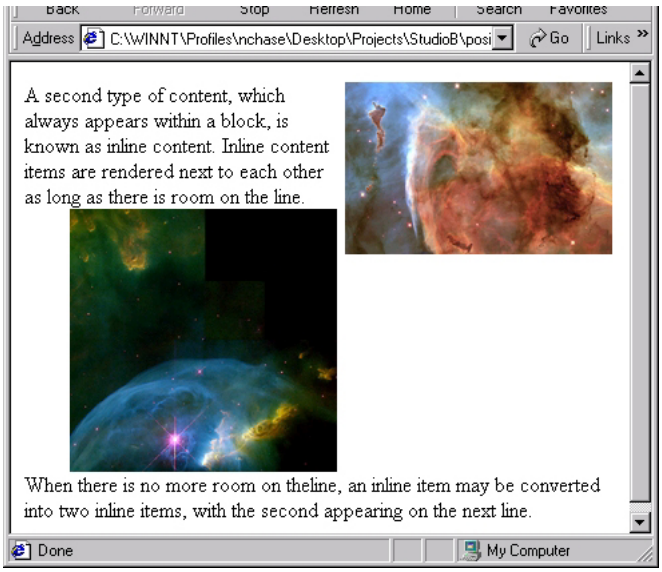
```
</p>

 </body>
 </html>
```



Note that regardless of other settings, a float always becomes a block element.

The clear property

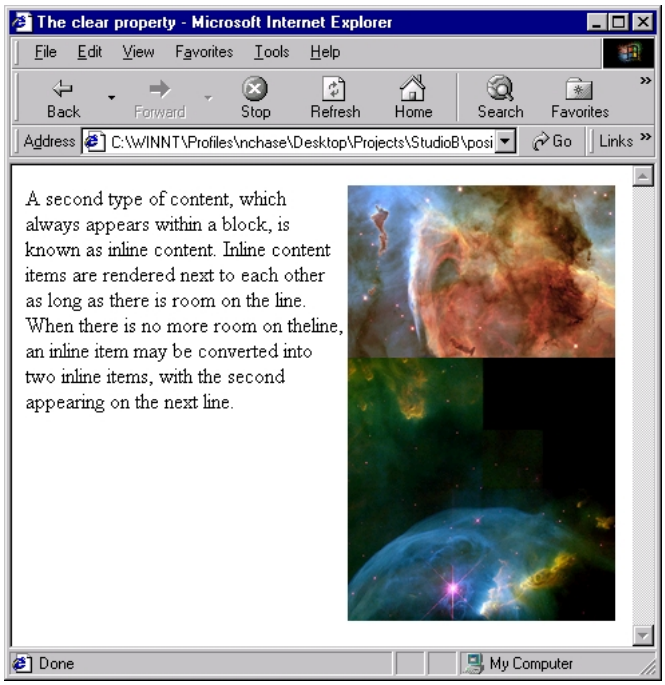
Floats provide a great deal of flexibility in that the Web author doesn't need to know precisely where the edge of the containing block is. That flexibility, however, can be a double-edged sword. A floated element floats to the edge of the containing block, unless there is an additional float in the way. For example, if both images in an earlier example were floated to the right, they could stack up, preventing the text from flowing properly:



The `clear` property prevents this from happening by indicating that one side of the floated object should be free of other floating objects. In other words, the browser must move the element down until it can be laid out on that side free of other floated elements.

```
<html>
 <head><title>The clear property</title>
 <style type="text/css">
    #nebula { float: right; }
    #bubble { float: right;
              clear:right; }
```

```
</style>
</head>
<body>

...

</body>
</html>
```



Replaced elements

In the actual rendering of the page, elements fall into two categories: *replaced* and *non-replaced* .

Non-replaced elements generally make up the majority of HTML. Replaced elements are those that are typically "linked in" to a page, such as images (`img`) and objects (`object`). Select boxes (`select`) are also replaced elements.

The distinguishing factor of a replaced element is that the browser knows only the intrinsic dimensions. All other information is determined by the content of the element.

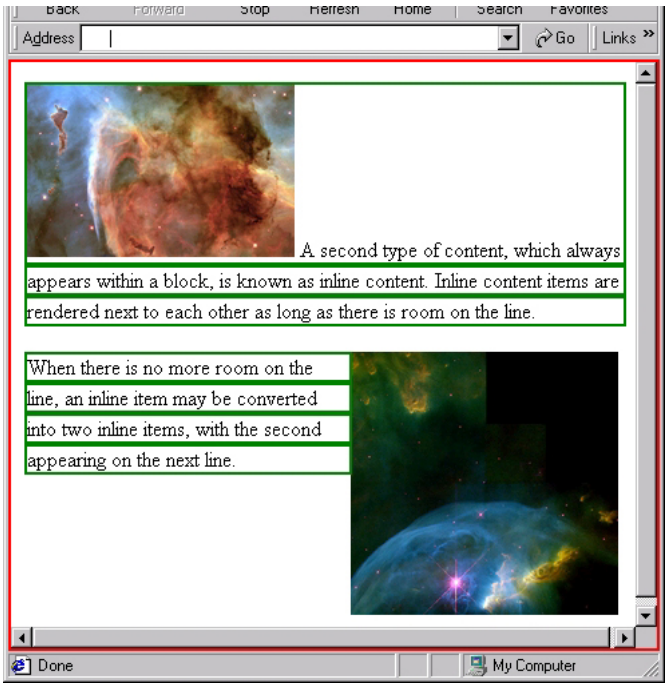
Whether an element is replaced or non-replaced is important because in certain situations involving determinations of size and location, the browser treats replaced and non-replaced elements differently.

Line boxes

Ultimately, a page is made up of blocks, and a block is made up of line boxes, or rows, of content. The browser creates a line box by adding, in order, all inline elements until one of two events occurs.

If the browser encounters a block-level element, it ends the current line box and creates a new one for the block element, then a third line box for subsequent content.

Otherwise, the browser continues to add elements until the length of the row is filled. The length of the row is typically the width of the containing block, but it may be reduced by the width of elements that have been floated to one side of the block or the other.



The height of a line box is the distance from the top of the highest element to the bottom of the lowest element. Note that in a situation where multiple elements (such as images) are aligned to a common baseline, the line might be taller than the tallest element within it.

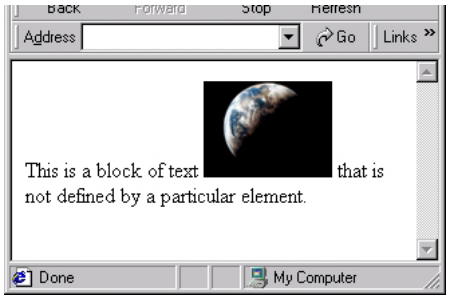
Section 3. A single box

Anonymous boxes

The basic building block of a page is the box. Every single item, whether it is a block-level or inline-level element, is considered a box.

Sometimes the layout of a page creates an *anonymous box*. This content is typically one of a series of inline items within a block not defined by any particular element. For example:

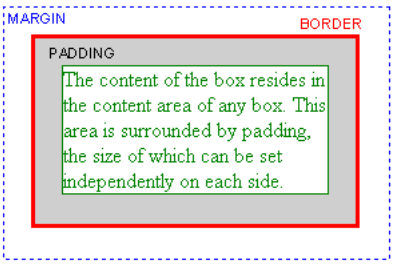
```
<div>
   This is a block of text
   <img src="earth.jpg" /> that is not
   defined by a particular element.
<div>
```



The text sections "This is a block of text," "that is," and "not defined by a particular element" are all anonymous boxes.

The box model

Even if a box is placed precisely, its content could still be out of position due to the internal construction of the box, so to precisely place any item, it is crucial to understand how the individual box works.



Four areas make up every box. They are:

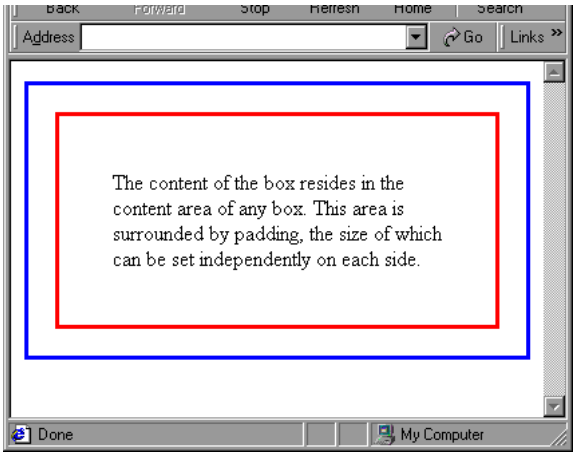
* margin: This area surrounds the box itself. No boxes placed within the normal flow encroach on this area, which is always transparent. The line around the outside of this area is known as the box's margin edge, or outer edge.
* border: This line surrounds the content and padding of the box. If the border has a width

greater than 0, the outside of the border is considered to be the border edge.

* padding: This area is the blank space between the content and the border of the box. The outside edge of the padding is known as the *padding edge*, and defines the containing block created by the box.
* content: This area contains the actual content of the box. The edge of this area is known as the *content edge*, or the inner edge.

For example:

```
<html>
 <head><title>Inline content</title>
 </head>
 <style type="text/css">
     #box { position: relative;
            margin: 20px;
            padding: 40px;
            border: 3px solid red; }
 </style>
 <body>
 <div style="border: 3px solid blue">
     <div id="box">
         The content of the box resides in the content area of any
         box. This area is surrounded by padding, the size of which
         can be set independently on each side.
     </div>
 </div>

 </body>
 </html>
```

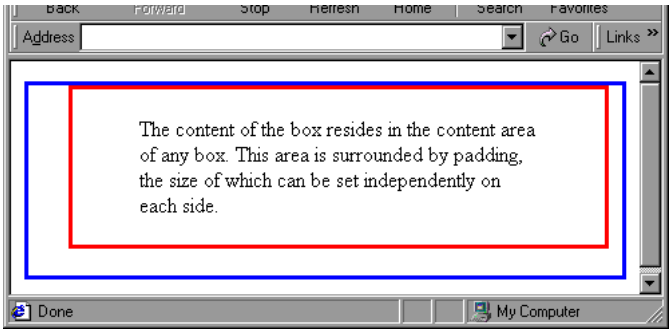


Controlling box properties

Each side of a box's margin, padding, and border can be controlled individually. For example:

```
<html>
 <head><title>Box Properties</title>
 </head>
 <style type="text/css">
     #box { position: relative;
            margin-top: 0px;
            margin-right: 10px;
            margin-bottom: 20px;
            margin-left: 30px;
            padding: 20px 50px;
            border: 3px solid red; }
 </style>
 <body>
 <div style="border: 3px solid blue">
```

```
    <div id="box">
        The content of the box resides in the content area of any
        box. This area is surrounded by padding, the size of which
        can be set independently on each side.
    </div>
</div>
</body>
</html>
```



To control individual sides, set each side explicitly (as seen above in the margin values) or use shorthand properties (as seen in the padding property). If the browser sees two values for any of these properties, it assigns the first to the top and bottom, and the second to the left and right.

Setting width

It would seem obvious that the width of a box is determined by the `width` property, and in most cases this is correct. As long as the setting takes padding into account, the content width appears as expected. For example, a block with a `width` of 300 pixels, a border 3 pixels wide, and 10 pixels of padding is going to display the content in an area 274 pixels wide (300 - 3 - 10 - 10 - 3 = 274).

The `width` property does not apply to non-replaced inline elements, but the width of a block can be set using an absolute length (using `px`, `em`, or `ex` units), a percentage of the containing block, or a value of `auto`.

In the first two situations, the results are fairly predictable. However, if the width is set to `auto`, the browser has a great deal of latitude, and how the value is ultimately set depends on the type of content being sized.

Determining width

If the width of a block is set to `auto`, a number of different factors come into play. The type of element, the width of the containing block and other values (such as `margin-right` and `margin-left`), and the positioning scheme of the element all combine to determine the actual value.

For replaced elements, a width of `auto` is always replaced with the intrinsic width for the element.

For non-replaced elements in the normal flow, the browser obeys the following constraint:

```
margin-left + border-left-width + padding-left + width +
padding-right + border-right-width + margin-right = width of the containing block
```

Sizing non-replaced elements that are absolutely positioned also involves taking the `left` and `right` values into account for a constraint of:

```
left + margin-left + border-left-width + padding-left + width
+ padding-right + border-right-width + margin-right + right = width of containing block
```

Specifying a range of sizes

Instead of (or in addition to) specifying a value for the `width`, the Web author may specify a range into which the width must fit by using the `min-width` and `max-width` properties.

The width is not the only property that may be affected by this range. The previous panel shows that properties such as width, margin, and padding can be interdependent. If the calculated width is outside the range specified by `min-width` and `max-width`, the appropriate value is substituted for `width` and values are recalculated.

The height of a box can be similarly constrained using `min-height` and `max-height`, or it can be specifically set.

Determining height

Like the `width` property, the `height` property can be explicitly set for a box, or can be set to `auto`.

Also like the `width`, a value of `auto` uses the intrinsic value for replaced elements and a calculated value for non-replaced elements.

The `height` property does not apply to non-replaced inline elements. Instead, these elements take the height of the line box in which they're contained.

The height of a non-replaced block element depends on the type of children it contains. If it contains only inline children, the height of the box runs from the top of the topmost line box to the bottom of the bottommost line box. If it contains block-level children, the height runs from the top border edge of the highest block-level child to the bottom border edge of the lowest block-level child.

Absolutely positioned elements also have to take into account settings for the `top` and `bottom`, for a constraint of:

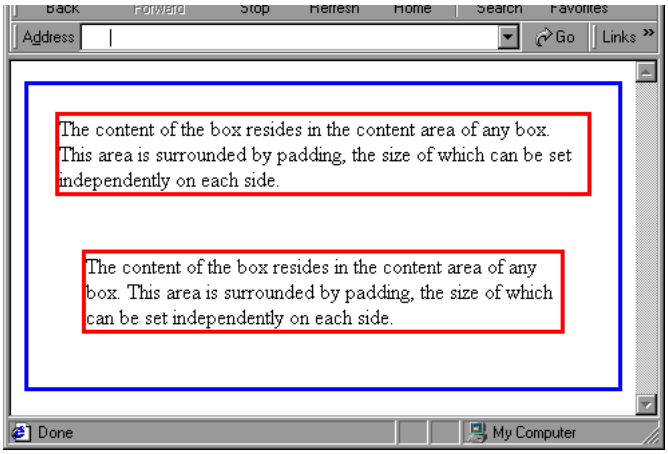
```
top + margin-top + border-top-width + padding-top + height +
 padding-bottom + border-bottom-width + margin-bottom + bottom = height of containing block
```

Collapsing margins

One of the properties that affects the height of a box is the margin. The margin is the transparent area around the box that sets it off from the content around it. In the case of

block boxes positioned above and below one another by the normal flow of the page, the vertical margins between them *collapse*. When margins are collapsed the browser generally chooses the larger of the two instead of showing both. For example:

```
<html>
 <head><title>Collapsing margins</title>
 </head>
 <style type="text/css">
   #box20 { margin: 20px;
            border: 3px solid red; }
   #box40 { margin: 40px;
            border: 3px solid red; }
 </style>
 <body>
 <div style="border: 3px solid blue">
     <div id="box20">The content of the box...</div>
     <div id="box40">The content of the box...</div>
 </div>
 </body>
 </html>
```



The larger margin, 40 pixels, is used between the two boxes, rather than the sum of the two margins (60 pixels).

If one of the margins is negative, the negative value is added to the positive value, so a 10px margin combines with a -5px margin to make a 5px margin. If both of the margins are negative, the margin with the greatest absolute value is used, so a -10px margin and a -5px margin combine to make a -10px margin.

The overflow property

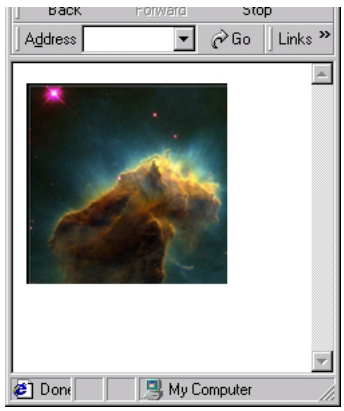
In some situations, such as the placing of an image, the content of a box may be larger than the box itself. The `overflow` property determines whether all of the content will be shown.

Take the following page, for example:

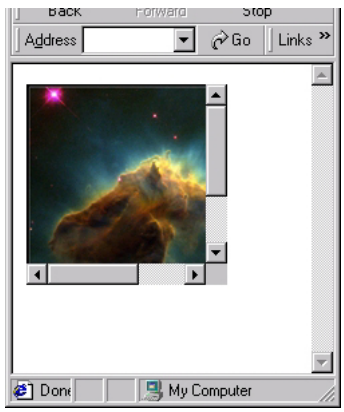
```
 <html>
 <head><title>The overflow
/title>
 <style type="text/css">
    #overflowBlock { width: 150px;
                     height: 150px;
                     overflow: visible }
 </style>
 </head>
 <body>
 <div id="overflowBlock">
    <img src="pillars.jpg"
width="200" height="196"  />
 </div>
 </body>
 </html>
```

Even though the image is larger than the block containing it, the entire image appears because the `overflow` property is set to `visible`.

If, on the other hand, the `overflow` property were set to `hidden`, the image would be clipped at the border of the block.



A third option is to set the `overflow` property to `scroll`, causing the browser to add scrollbars if the content is too large for the box.



The `overflow` property can also be set to `auto`. For browsers, this value produces the same behavior as `scroll`.

It's important to note that any block that exceeds the size of its containing block, such as a div with a fixed size or even a section of preformatted (`<pre></pre>`) text, can make use of

this property. It's not limited to images.

CSS2 also defines the `clip` property, which allows the definition of a clipping path around the content. Unfortunately, this is not yet supported by common browsers.

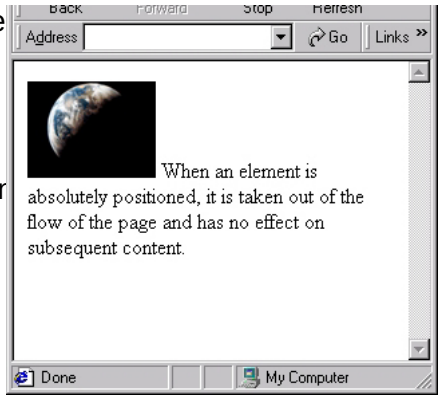
Section 4. Positioning content

Setting a position

Now that you understand the basics of an individual box, it's time to look at actually positioning it on the page.

Generally, the position of a box is set using the `top` and `left` properties that determine the amount of vertical and horizontal space between the margin edge of the box and the *reference point*. The reference point may be a fixed position on the page, such as the upper left-hand corner of the window, or it may be a point that is moved relative to the overall layout of the page. How the reference point is set depends on the *positioning scheme* of the box.

The positioning scheme of the box is set using the `position` property. The default value for `position` is `static`. Content positioned as part of the normal flow of the page is said to be statically positioned. For example, both the image and text on the page to the right have a `positior` value of `static`.



A second possible value, `fixed`, specifies that the element should, in the case of the browser, remain in a fixed position with respect to the viewport, or visible section of the window. In other words, if `top` and `left` were set to `50px`, the box would remain 50 pixels from the top and left edges, even if the page were scrolled. While this capability would be enormously helpful for menus and other purposes, it is unfortunately not well supported as of this writing, and must be approximated using scripting.

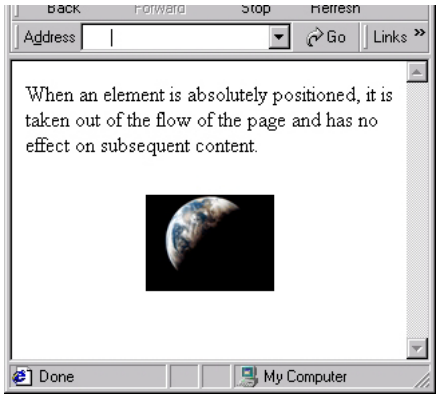
In terms of dynamic positioning, however, the most useful values for the `position` property are `absolute` and `relative`.

Absolute positioning

When an element uses a `position` value of `absolute`, it is removed from the normal flow altogether and positioned relative to the containing block. For example:

```
<html>
<head><title>Absolute positioning</title>
<style type="text/css">
   #earth {position: absolute;
           top: 100px;
           left: 100px; )
</style>
</head>
<body>
<img id="earth" src="earth.jpg" />
When an element is absolutely positioned,
it is taken out of the flow of the page
```

```
and has no effect on subsequent content.
</body>
</html>
```



The subsequent text is positioned as though the image didn't exist.

Positioning values may be positive or negative numbers. A negative value simply provides the opposite effect. For example, assigning the `top` property a value of `-10px` moves the block up 10 pixels, where assigning the `left` property a value of `-10px` moves the block 10 pixels to the right.

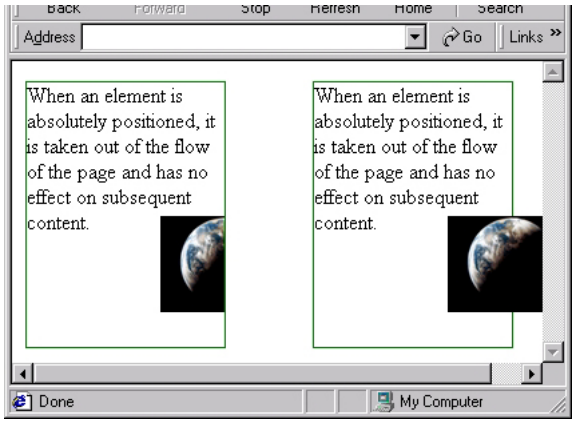
Nesting and absolute positioning

Because absolute positioning uses the containing block as its reference point, nested content that is absolutely positioned knows only of its parent. What's more, it stays within the containing block, so if `overflow` is set to `hidden`, it is possible for the content to disappear altogether.

For example:

```
<html>
 <head><title>Absolute positioning</title>
 <style type="text/css">
    #container1 { position: absolute;
                  height: 200px; width: 150px;
                  overflow: hidden;
                  border: 1px solid green; }
    #container2 { position: absolute;
                  top: 15px; left: 225px;
                  height: 200px; width: 150px;
                  border: 1px solid green; }
    .earth {position: absolute;
            top: 100px;
            left: 100px; )
 </style>
 </head>
 <body>

 <div id="container1">
    <img class="earth" src="earth.jpg" />
    When an element is absolutely positioned, it is taken out of the
    flow of the page and has no effect on subsequent content.
 </div>

 <div id="container2">
    <img class="earth" src="earth.jpg" />
```

```
   When an element is absolutely positioned, it is taken out of the
   flow of the page and has no effect on subsequent content.
</div>
</body>
</html>
```



Although both graphics are absolutely positioned and use the same positioning information, they appear in different locations because they use different reference points. Each graphic uses its own containing block as a reference point, even when, as is the case with `container2`, that containing block has itself been absolutely positioned.

Notice also that because the second graphic is absolutely positioned, it can extend past the edge of the browser window.

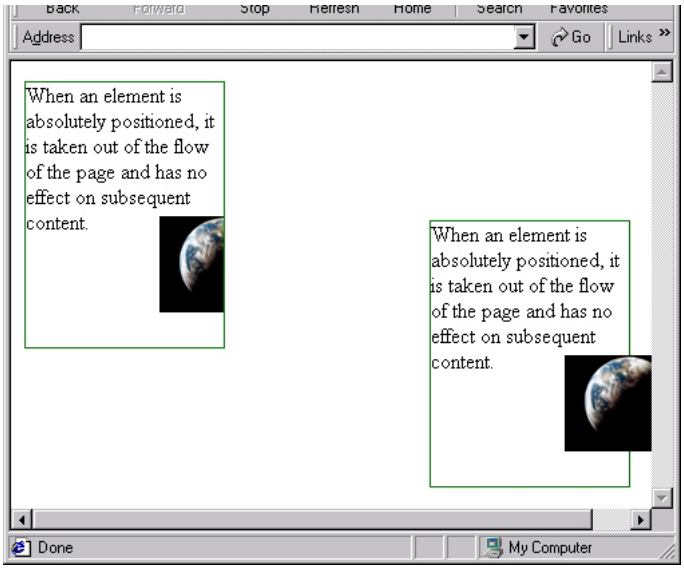
Using the bottom right edge

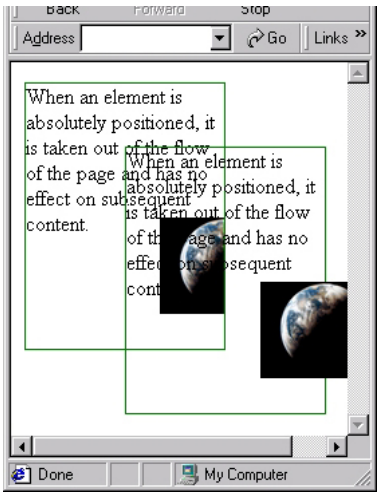
Though it is customary to use the `top` and `left` properties to position a block, it is not unusual to use the `right` and `bottom` properties for positioning.

For example, designs might call for a block that is always positioned in the lower right-hand corner of the page, regardless of the size of the browser window:

```
<html>
 <head><title>Absolute positioning</title>
 <style type="text/css">
    #container1 { position: absolute;
                  height: 200px; width: 150px;
                  overflow: hidden;
                  border: 1px solid green; }
    #container2 { position: absolute;
                  bottom: 15px; right: 15px;
                  height: 200px; width: 150px;
                  border: 1px solid green; }
    .earth {position: absolute;
            top: 100px;
            left: 100px; )
 </style>
 </head>
 <body>

 <div id="container1">
    <img class="earth" src="earth.jpg" />
    When an element is absolutely positioned, it is taken out of the
    flow of the page and has no effect on subsequent content.
 </div>

 <div id="container2">
    <img class="earth" src="earth.jpg" />
```

```
   When an element is absolutely positioned, it is taken out of the
   flow of the page and has no effect on subsequent content.
</div>
</body>
</html>
```





Notice that the edge of the block is not affected by the fact that one of its children extends past its border.

The containing block

Because absolute positioning uses the containing block as its reference point, it's crucial to be able to determine which element generates a box's containing block.

The top-level ancestor for all boxes is the initial containing block. For browsers, the initial containing block is the same as the content edge of the page itself. To determine the containing block for page content elements, certain rules apply.

In general, an element's containing block is found by determining the nearest block-level ancestor and using its content edge. Consider this example:

```
<div id="container1">
    <span class="images">
       <img class="earth" src="earth.jpg" />
    </span>
    <span class="bodyText">
       When an element is absolutely positioned, it is taken
       out of the flow of the page and has no effect on subsequent
       content.
    </span>
 </div>
```

Both the image and the text have as their containing block `container1`.

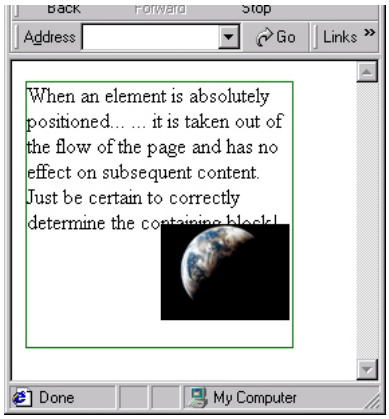
The exceptions to this rule are elements that use `position:fixed` (which use the browser window as their containing block) and absolutely positioned elements. To find the containing block for an absolutely positioned element, first determine the element's nearest absolutely, relatively, or fixed-positioned ancestor. If that ancestor is a block-level element, its padding edge forms the containing block.

The containing block (continued)

If the ancestor is an inline element, the extent of the containing block is found by determining the extent of the outside edges of the first and last boxes within the ancestor.

Consider this example:

```
<html>
<head><title>Absolute positioning</title>
<style type="text/css">
   #container1 { position: absolute;
                 height: 200px; width: 200px;
                 overflow: hidden;
                 border: 1px solid green; }
   .earth {position: absolute;
           bottom: 20px;
           left: 100px; }

</style>
</head>
<body>
<div id="container1">
   <span id="innerContainer">
      When an element is absolutely positioned...
      <img class="earth" src="earth.jpg" />
      ... it is taken out of the flow of the page
      and has no effect on subsequent content.
      Just be certain to correctly determine the
      containing block!
   </span>
</div>
</body>
</html>
```

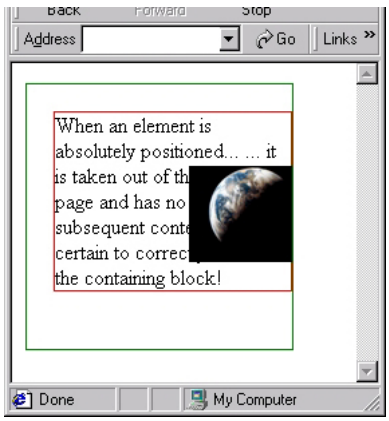


Because the positioning scheme of `innerContainer` is `static`, the containing block for the image is `container1`.

If, on the other hand, positioning information is added:

```
#innerContainer { position: absolute;
                  top: 20px;
                  left: 20px;
                  border: 1px solid red; }
```

the containing block becomes `innerContainer`.



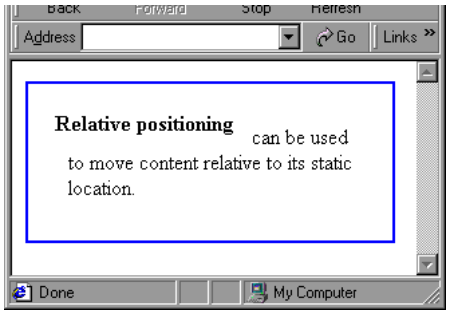
In the event that there is no appropriate ancestor for the element, the initial containing block becomes the containing block for the element.

Relative positioning

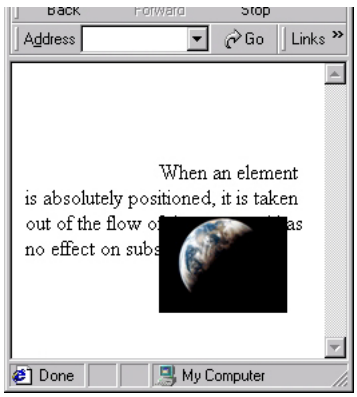
Sometimes, the goal isn't so much to place an element precisely as it is to offset it from its normal position. For example:

```
<html>
<head><title>Relative positioning</title>
<style type="text/css">
   #container1 { position: absolute;
                 padding: 30px;
                 border: 2px solid blue; }
```

```
   .emphasis {position: relative;
              top: -10px; left: -10px; )
</style>
</head>
<body>
<div id="container1">
   <span class="emphasis"><b>Relative
   positioning</b></span> can be used to
   move content relative to its static
   location.
</div>
</body>
</html>
```



The major difference is that a box that is relatively positioned is laid out according to the normal flow, then offset by the values specified in `top` and `left`. The browser places subsequent content as though the box exists in its original static position.



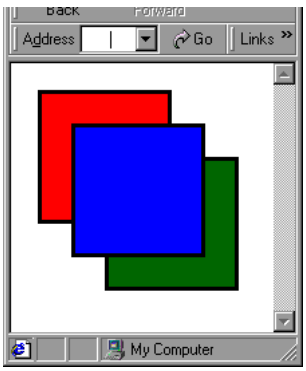
Section 5. Layering content

The z-index property

With all of these elements being moved around on the page, it's virtually inevitable that at some point, two or more are going to overlap. The order in which they do determines which content is visible, and which falls "behind" the other content.

In normal circumstances, elements simply pile up on the page, with each new element rendered "in front of" the previous elements. The last element specified is the one that's visible. To change that, use the `z-index` property. For example:

```
<html>
 <head><title>The z-index</title>
 <style type="text/css">
    div { position: absolute;
          height:100; width:100;
          border: 3px solid black; }
    #redBox { z-index: 5;
              top:20px; left: 20px;
              background-color: red; }
    #blueBox { z-index: 20;
               top:45px; left: 45px;
               background-color: blue; }
    #greenBox { z-index: 10;
                top:70px; left: 70px;
                background-color: green; }
 </style>
 <body>
 <div id="redBox"></div>
 <div id="blueBox"></div>
 <div id="greenBox"></div>
 </body>
 </html>
```



The higher the value of the `z-index` property, the "closer" the block is rendered, so the middle blue box is rendered in front, even though the green box was rendered after it.

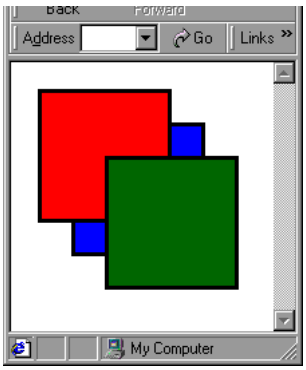
The `z-index` property takes any integer value. In the event that two boxes in the same stacking context have the same value, the last one rendered takes precedence.

Each element may actually have `z-index` values for two contexts: the *root stacking context* and the *local stacking context*.

Root stacking context

The root stacking context determines the overall stack of the document. In building a document, all elements are assumed to have a stacking order of zero, so any element that has a specific value for the `z-index` property is going to be rendered in front of any that doesn't. For example:

```
   #redBox { z-index: 5;
             top:20px; left: 20px;
             background-color: red; }
   #blueBox { top:45px; left: 45px;
              background-color: blue; }
   #greenBox { z-index: 10;
               top:70px; left: 70px;
               background-color: green; }
```

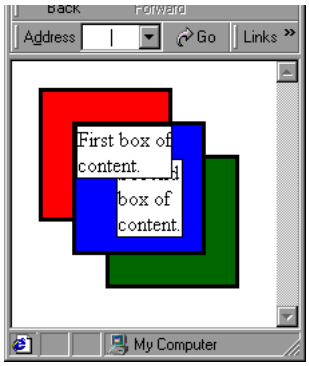


Because the blue box no longer has a `z-index` specified, it is rendered behind the other two boxes.

Local stacking context

If an element has a specific value for `z-index`, that element establishes a local stacking context. This determines the order of rendering for each element within it, and is independent of any other local stacking context. For example:

```
<html>
<head><title>The z-index</title>
<style type="text/css">
   div { position: absolute; height:100; width:100;
         border: 3px solid black; }
   #redBox { z-index: 5;
             top:20px; left: 20px; background-color: red; }
   #blueBox { z-index: 20;
              top:45px; left: 45px; background-color: blue; }
   #greenBox { z-index: 10;
               top:70px; left: 70px; background-color: green; }
   #blueText1 { border: 1px solid black; height: auto; width: auto;
                background-color: white;
                z-index: 1; }
   #blueText2 { border: 1px solid black; height: auto; width: auto;
                background-color: white;
                top: 25px; left: 30px; }
</style>
<body>
<div id="redBox"></div>
<div id="blueBox">
   <div id="blueText1">
      First box of content.
   </div>
   <div id="blueText2">
      Second box of content.
   </div>
</div>
<div id="greenBox"></div>
</body>
</html>
```



Notice that because the root stacking context of the second blue div places it in front of the other two divs, all of its content is rendered in front of the other two divs even though the first section of content has a `z-index` smaller than the other two, and the second section of content doesn't have a `z-index` at all. Within the block, however, they render as expected, with the first section taking precedence because it has a value specified.

Section 6. Visibility

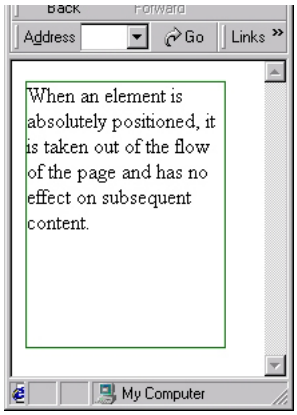
The visibility property

Many of the applications for dynamic positioning, such as popup menus and informational elements, require content to be invisible until it's needed. In most such cases, the `visibility` property is the solution.

The default value for `visibility` is `visible`, but other values are possible: `hidden` makes the element invisible; `collapse` applies only to table-related elements and not only renders the content invisible, but also collapses the affected element into those around it. (For non-table-related elements, a value of `collapse` acts like a value of `hidden`.)

For example:

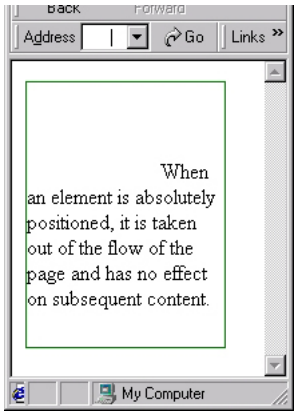
```
<html>
<head><title>Absolute positioning</title>
<style type="text/css">
   #container1 { height: 200px; width: 150px;
                 border: 1px solid green; }
   #earth { visibility: hidden; }
</style>
</head>
<body>

<div id="container1">
   When an element is absolutely positioned,
   it is taken out of the flow of the page
   and has no effect on subsequent
   content.<br />
   <img id="earth" src="earth.jpg" />
</div>
</body>
</html>
```



The image doesn't appear on the page because the `visibility` property for the image is set to `hidden`. However, it still affects the flow of the page.

The effect on flow

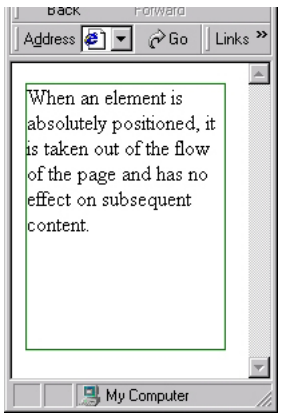
In the previous example, it seemed that the image was removed from the page altogether, but that's not quite the case. Moving the image ahead of the text shows a different result, as seen to the right.



Although the image doesn't appear, the hole where it belongs is obvious. Setting the `visibility` property to `hidden` doesn't remove an element from the flow, it just prevents it from appearing. The rest of the page is rendered just as if the element were right where it belongs.

To completely remove an element from the flow of the page, set the `display` property to `none`:

```
<html>
<head><title>Absolute positioning</title>
<style type="text/css">
   #container1 { height: 200px; width: 150px;
                 border: 1px solid green; }
   #earth { display:none }
</style>
</head>
<body>

<div id="container1">
   <img id="earth" src="earth.jpg" />
   When an element is absolutely positioned,
   it is taken out of the flow of the page
   and has no effect on subsequent
   content.<br />
</div>
</body>
</html>
```



Section 7. Scripting position

Scripting and CSS properties

Dynamic positioning is not limited to laying out the page. You can use client-side scripting to make your pages even more dynamic. In fact, the combination of CSS and JavaScript is known by the (arguably inaccurate) name of Dynamic HTML, or DHTML.

All of the effects demonstrated so far in this tutorial have been accomplished by setting CSS properties. To control them from a script, simply use the script to set or alter the properties. This control requires an understanding of how the page is structured.

Like an XML document, a browser page follows a form of Document Object Model, or DOM. Each element has children, and each of these children has properties, all accessible via dot notation. For example, the `location` property of the `document` object can be accessed as:

```
document.location
```

The CSS properties are stored as part of the style property of the object. If an element, (a div, for example) were represented by an object named `box`, the `visibility` property could be accessed through:

```
box.style.visibility
```

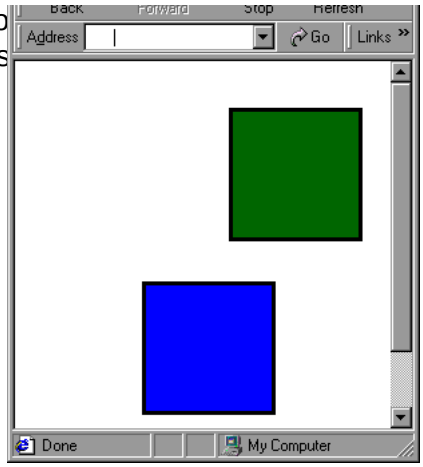
The scripts that follow demonstrate this concept more fully.

Invisible items

The first example shows a simple script that allows the user to make boxes disappear by clicking on them:

```
<html>
 <head><title>Whack-a-box</title>
 <style type="text/css">
    div { position: relative; height:100; width:100; border: 3px solid black; }
    #redBox { top:20px; left: 20px; background-color: red; }
    #blueBox { top:50px; left: 85px; background-color: blue; }
    #greenBox { top:-180px; left: 150px; background-color: green; }
 </style>
 <script type="text/javascript">
    function hide(box) {
       box.style.visibility = 'hidden';
    }
 </script>
 </head>
 <body>
 <div onclick="hide(this)" id="redBox"></div>
 <div onclick="hide(this)" id="blueBox"></div>
 <div onclick="hide(this)" id="greenBox"></div>
 </body>
 </html>
```

To activate the script, use the `onclick()` event to execute the `hide()` function. When the user clicks the div, the browser replaces the keyword `this` with a reference to the object that fired the event, so in the `hide()` function, `box` always refers to the div that the user clicked.

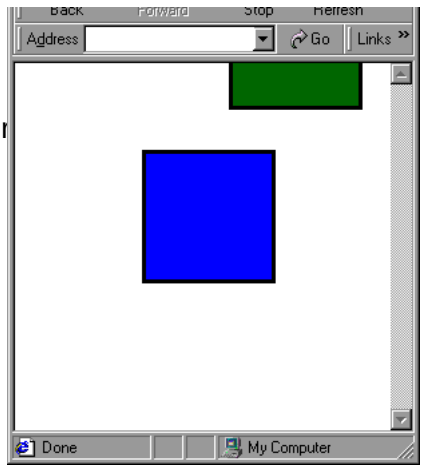


Once inside the function, the script changes the value of the `visibility` property to `hidden`, making the box disappear. Note that even though the boxes are relatively positioned, hiding them doesn't affect the layout of the page because they are still within the flow.

Disappearing items

If the browser supports it, a small change to the script can create an even more dynamic page. Setting the `display` property within the script causes the page to re-flow, taking the missing elements into account:

When the user clicks the box, it is removed from the flow, causing subsequent elements to shift. If this effect is not desirable, be sure to either use absolute positioning on elements that disappear, or simply don't change the `display` property.



Affecting invisible items

Once the boxes have disappeared, there appears to be no way to retrieve them. Because they are invisible, they do not receive events such as mouse clicks. One way to get around this problem is not to make the element disappear; instead of making the actual element disappear, you can create a child element and make that disappear instead. To do this, a script needs a way to refer specifically to an element:

```
<html>
 <head><title>Whack-a-box</title>
 <style type="text/css">
    div { position: relative; height:100; width:100; border: 3px solid black; }
    #redBox { top:20px; left: 20px; border: none; }
    #redSubBox { height:100; width:100; background-color: red; }
    #blueBox { top:50px; left: 85px; border: none; }
    #blueSubBox { height:100; width:100; background-color: blue; }
    #greenBox { top:-180px; left: 150px; border: none; }
    #greenSubBox { height:100; width:100; background-color: green; }
 </style>
 <script type="text/javascript">
    function toggle(boxId) {
       var currentVisibility
       currentVisibility = document.getElementById(boxId).style.visibility;

       if (currentVisibility == 'hidden') {
          document.getElementById(boxId).style.visibility = 'visible';
       } else {
          document.getElementById(boxId).style.visibility = 'hidden';
       }

    }
 </script>
 </head>
 <body>
 <div onclick="toggle('redSubBox')" id="redBox">
    <div id="redSubBox"></div>
 </div>
 <div onclick="toggle('blueSubBox')" id="blueBox">
    <div id="blueSubBox"></div>
 </div>
 <div onclick="toggle('greenSubBox')" id="greenBox">
    <div id="greenSubBox"></div>
 </div>
 </body>
 </html>
```

Notice that all appearance properties now apply to the child elements, but the `onclick` event is still referenced from the parent element. Fortunately, when a child item of an element is clicked, the parent still receives the event. Unfortunately, that also means that the `this` keyword is no longer useful, because the object clicked is no longer the object to be affected.

To solve the problem of identifying which object to alter, use the `getElementById()` method, part of the `document` object. It returns an object based on the `id` attribute, and from there the script can set properties as before.

Now the user can click a box to make it disappear, and click the (seemingly) empty space to make it reappear.

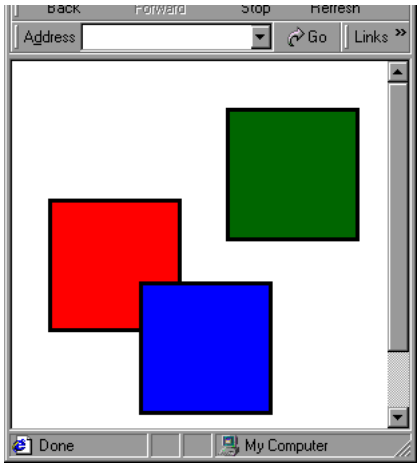
Following the mouse

Sometimes, instead of knowing where an object is, you want to know where the user's mouse is and act accordingly. This example shows how to access the current coordinates of the mouse and use them to drag content around the page.

```
<html>
 <head><title>Whack-a-box</title>
 <style type="text/css">
    div { position: relative; height:100; width:100; border: 3px solid black; }
    #redBox { top:20px; left: 20px; background-color: red; }
    #blueBox { top:50px; left: 85px; background-color: blue; }
    #greenBox { top:-180px; left: 150px; background-color: green; }
 </style>
```

```
<script type="text/javascript">
   function follow(box) {
       box.style.left=(event.clientX - 50);
       box.style.top=(event.clientY - 50);
   }
</script>
</head>
<body>
<div onmousemove="follow(this)" id="redBox"></div>
<div id="blueBox"></div>
<div id="greenBox"></div>
</body>
</html>
```

The `onmousemove` event fires every time the user moves the mouse over the affected area (in this case, `redBox`). The function then retrieves the position of the mouse at that instant from `event.clientX` and `event.clientY`), and uses it to create a new position for the box. The process is repeated every time the mouse moves, causing the box to "follow" the user's mouse movements.



The browser then uses that position information to create a new position for the box.

Controlling layering

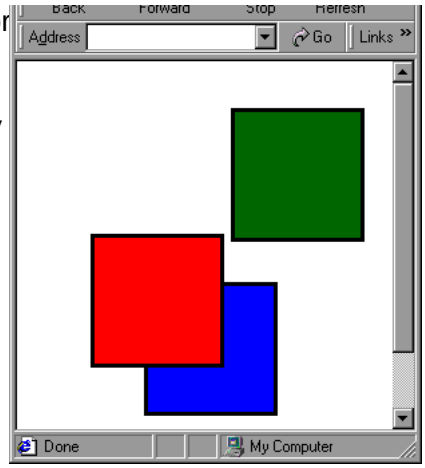
In dragging the box around the page, you may notice that you can't drag it under any of the other boxes, because they are in front of it. When the mouse reaches the boundary of a box that is in front of `redBox`, the `onmousemove` event no longer affects `redBox`, so it doesn't move.

This example shows how to remedy this by controlling the `z-index` property from within a script:

```
<html>
 <head><title>Whack-a-box</title>
 <style type="text/css">
    div { position: relative; height:100; width:100; border: 3px solid black; }
    #redBox { z-index: 5; top:20px; left: 20px; background-color: red; }
    #blueBox { z-index: 10; top:50px; left: 85px; background-color: blue; }
    #greenBox { z-index: 15; top:-180px; left: 150px; background-color: green; }
 </style>
 <script type="text/javascript">
    function follow(box) {
        box.style.left=(event.clientX - 50);
        box.style.top=(event.clientY - 50);
    }
    function setBehind(box) {
       box.style.zIndex=1;
    }
 </script>
 </head>
 <body>
 <div onmousemove="follow(this)" id="redBox"></div>
```

```
<div onclick="setBehind(this)" id="blueBox"></div>
<div onclick="setBehind(this)" id="greenBox"></div>
</body>
</html>
```

When the user reaches the boundary of the blue or green box, clicking that box causes the `z-index` property to be set lower than the `z-index` property for `redBox`, causing `redBox` to suddenly appear in front.



Note that due to conflicting naming restrictions, the `z-index` property is actually referenced as `zIndex`. This conversion applies to all of the hyphenated properties, such as `border-width` (`borderWidth`) and `padding-right` (`paddingRight`).

Section 8. Dynamic positioning summary

Summary

Dynamic positioning of content brings the browser much closer to the goal of providing the same flexibility and aesthetics as a page layout program, but also provides the advantage of separating positioning information from content.

Items on a page can be absolutely or relatively positioned, and their sizes, padding, margins, and borders can be controlled in order to place them precisely. The use of CSS properties can also provide scripting capabilities, in which positioning information can be controlled programmatically in response to user actions.

Resources

Some good places to find additional information on dynamic positioning and its related technologies are listed below:

* Read *CSS Layout Techniques: for Fun and Profit* for an excellent look at using dynamic positioning as an alternative to HTML tables.
* Read the complete *Cascading Style Sheets level 2 Recommendation* .
* Follow the progress of work on *Cascading Style Sheets level 3* at the W3C.
* Read the *Intro to Cascading Style Sheets: Type* tutorial for a look at CSS in general and text effects in particular.
* Explore additional CSS resources at *The CSS Pointers Group* .
* Read *Writing Cross-Browser Dynamic HTML* , by David Boles and Rachael Ann Siciliano, for a look at building your DHTML for all browsers.
* Read *How to Build Pull-Down Menus with JavaScript* , an excerpt from *Javascript for the World Wide Web: Visual QuickStart Guide, 4th Edition* by Tom Negrino and Dori Smith.
* Explore additional DHTML resources at the *Web Developer's Virtual Library's Dynamic HTML pages* .
* Read a *JavaScript Tutorial for Programmers* by Aaron Weiss.
* Read *Creating Dynamic HTML in Internet Explorer 4+ using JavaScript* , an excerpt from Paul Wilton's *Beginning Javascript*, for a look at using JavaScript to change HTML elements.
* Read *A Cross-browser DHTML table* for a look at adapting JavaScript to different browsers.
* Explore *Danny Goodman's JavaScript Pages* for a look at what functions and tags are supported in which browsers.

Downloads

* Download a *zip archive of the sample code* presented in this tutorial.
* Download *IBM Web Browser for OS/2* .
* Download *Microsoft Internet Explorer 5.5* , *Internet Explorer 6* , or *Internet Explorer 5.0 for Macintosh* .
* Download *Netscape 6* , with improved compliance over earlier versions.
* Download *Opera* , a standards-compliant browser available for OS/2, BeOS, Linux/Solaris, Mac, QNX, Symbian OS, and more than 20 different localized versions for Windows.

Feedback

We welcome your feedback on this tutorial -- let us know what you think. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.