
Technologies ▼

References & Guides ▼

Feedback ▼

Sign in 

 Search

Box model recap

Overview: Styling boxes

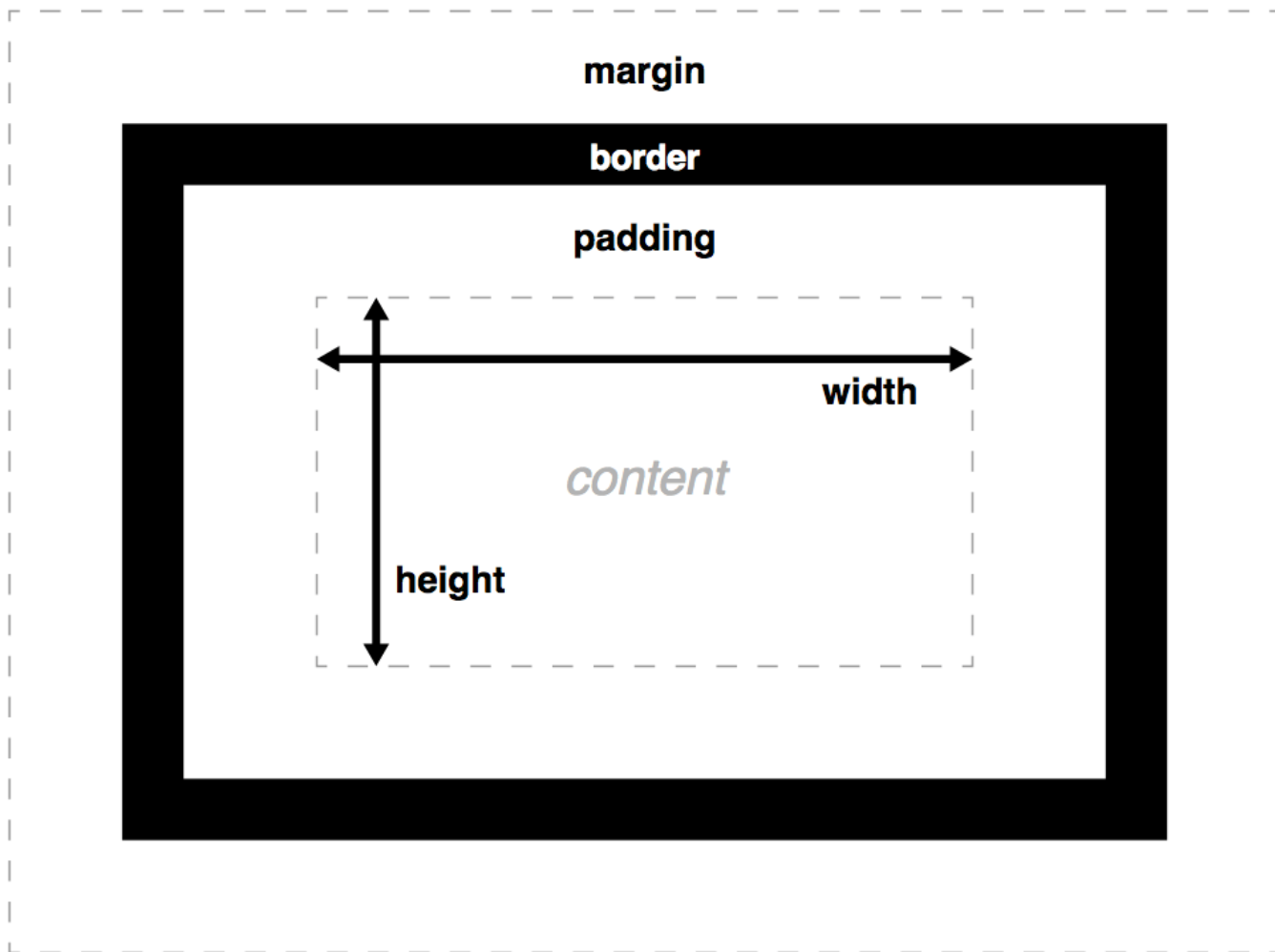
Next

We looked at the basics of the CSS box model in our Introduction to CSS module. This article will provide a recap, and dive into some further details on the subject.

Prerequisites:	HTML basics (study Introduction to HTML), and an idea of How CSS works (study Introduction to CSS.)
Objective:	To recap the basics of the CSS box model, and add a few more details.

Box properties

As you'll no doubt know by now, every element within a document is structured as a rectangular box inside the document layout, with properties that can be altered like so:



- **width** and **height** set the width/height of the content box.
- The **padding** family of properties sets the width of the padding (don't forget longhand properties like **padding-bottom** that allow you to set one side at once).
- The **border** family of properties sets the width, style and color of the border (there are many longhand border properties available).
- The **margin** family of properties sets the width of the outer area surrounding the CSS box, which pushes up against other CSS boxes in the layout (again, longhand properties like **margin-bottom** are available).

Some other points worth remembering:

- Box heights don't observe percentage lengths; box height always adopts the height of the box content, unless a specific absolute height is set (e.g. pixels or ems.) This is more convenient than the height of every box on your page defaulting to 100% of the viewport height.
- Borders ignore percentage width settings too.

- Margins have a specific behavior called **margin collapsing**: When two boxes touch against one another, the distance between them is the value of the largest of the two touching margins, and not their sum.

Note: See The box model article's *Box properties* section for a more complete overview and an example.

Overflow [↗](#)

When you set the size of a box with absolute values (e.g. a fixed pixel width/height), the content may not fit within the allowed size, in which case the content overflows the box. To control what happens in such cases, we can use the `overflow` property. It takes several possible values, but the most common are:

- `auto`: If there is too much content, the overflowing content is hidden and scroll bars are shown to let the user scroll to see all the content.
- `hidden`: If there is too much content, the overflowing content is hidden.
- `visible`: If there is too much content, the overflowing content is shown outside of the box (this is usually the default behavior.)

Note: See The box model article's *Overflow* section for a more complete overview and an example.

Background clip [↗](#)

Box backgrounds are made up of colors and images, stacked on top of each other (`background-color`, `background-image`.) They are applied to a box and drawn under that box. By default, backgrounds extend to the outer edge of the border. This is often fine, but in some cases it can be annoying (what if you have a tiled background image that you want to only extend to the edge of the content?) This behaviour can be adjusted by setting the `background-clip` property on the box.

Note: See The box model article *background-clip* section for an example.

Outline [↗](#)

The `outline` of a box is something that looks like a border but which is not part of the box model. It behaves like the border but is drawn on top of the box without changing the size of the box (to be specific, the outline is drawn outside the border box, inside the margin area.)

Note: Beware when using the `outline` property. It is used in some cases for accessibility reasons to highlight active parts of a web page such as links when a user clicks on them. If you do find a use for outlines, make sure you don't make them look just like link highlights as this could confuse users.

Advanced box properties [↗](#)

Now we've had a brief recap, let's look at some more advanced box properties that you'll find useful.

Setting width and height constraints [↗](#)

Other properties exist that allow more flexible ways of handling content box size — setting size constraints rather than an absolute size. This can be done with the properties `min-width`, `max-width`, `min-height`, and `max-height`.

An obvious use is if you want to allow a layout's width to be flexible, by setting its outer container's width as a percentage, but you also don't want it to become too wide or too narrow because the layout would start to look bad. One option here would be to use responsive web design techniques (which we'll cover later), but a simpler method might be to just give the layout a maximum and minimum width constraint:

```
1 | width: 70%;  
2 | max-width: 1280px;  
3 | min-width: 480px;
```

You could also couple this with the following line, which centers the container it is applied to inside its parent:

```
1 | margin: 0 auto;
```

0 causes the top and bottom margins to be 0, while `auto` (in this case) shares the available space between the left and right margins of the container, centering it. The end result is that when the container is within the min and max width constraints, it will fill the entire viewport width. When 1280px width is exceeded, the layout will stay at 1280px wide, and start to be centered inside the available space. When the width goes below 480px, the viewport will become smaller than the container, and you'll have to scroll to see it all.

You can see the above example in action in [min-max-container.html](#) ([see the source code](#)).

Another good use of `max-width` is to keep media (e.g. images and video) constrained inside a container. Returning to the above example, the image causes a problem — it looks ok until the container becomes narrower than the image, but at this point the image starts to overflow the container (as it is a fixed width). To sort the image out, we can set the following declarations on it:

```
1  display: block;
2  margin: 0 auto;
3  max-width: 100%;
```

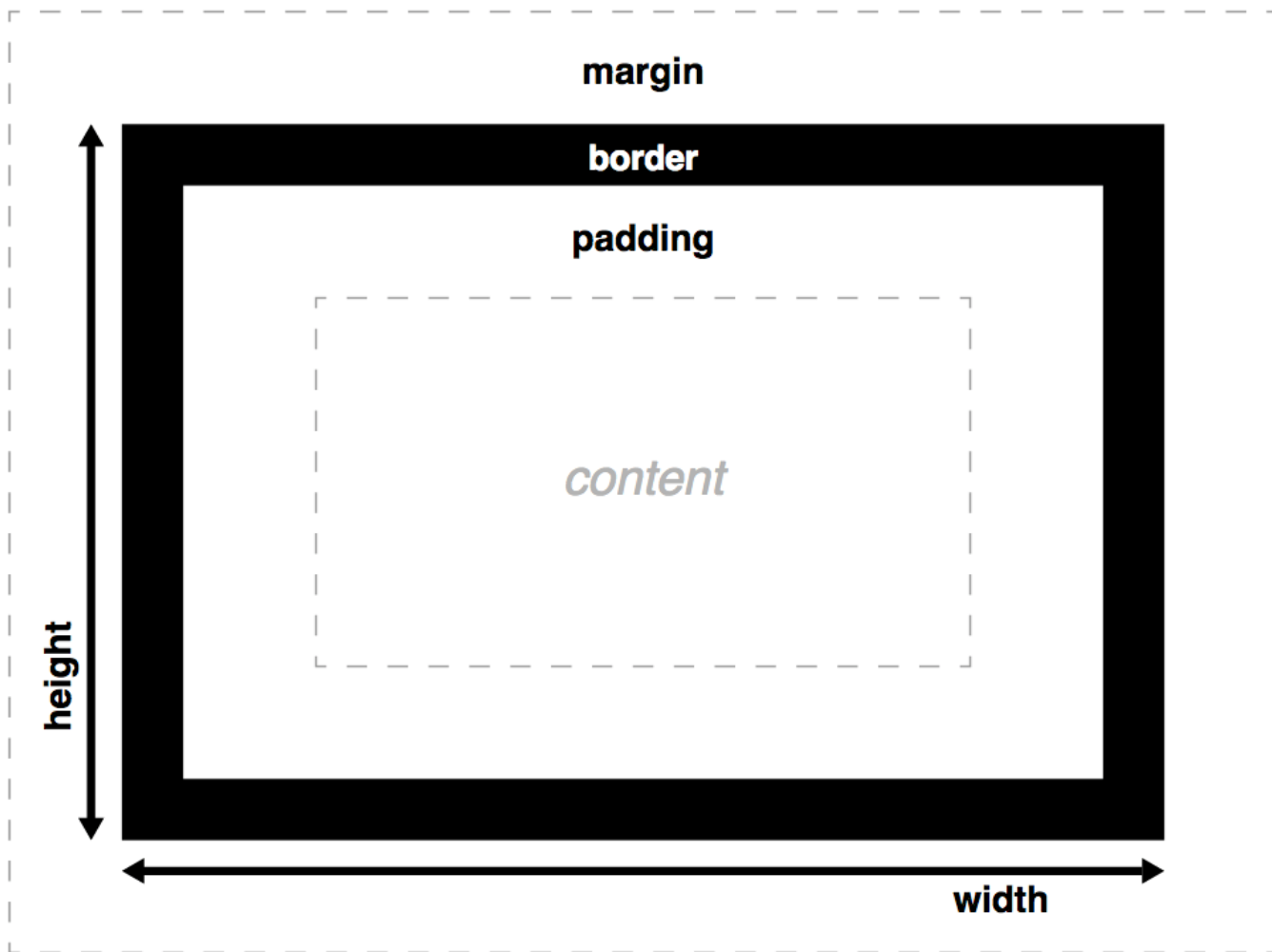
The first two make it behave like a block element and center it inside the parent container. But the real magic is in the third one. Setting the image's `max-width` to 100% constrains the image's width to be smaller than the width of the container. Therefore, if the container shrinks to be smaller than the image width, the image will shrink along with it.

You can try this modified example at [min-max-image-container.html](#) ([see the source code](#)).

Note: This technique works as far back as Internet Explorer 7, so it is nicely cross browser.

Changing the box model completely [↗](#)

The total width of a box is the sum of its `width`, `padding-right`, `padding-left`, `border-right`, and `border-left` properties. In some cases it is annoying (for example, what if you want to have a box with a total width of 50% with border and padding expressed in pixels?) To avoid such problems, it's possible to tweak the box model with the property `box-sizing`. With the value `border-box`, it changes the box model to this new one:



Let's look at a live example. We will set up two identical `<div>` elements, giving each one the same width, border and padding. We will also use some JavaScript to print out the computed value (the final on-screen value in pixels) of the width for each box. The only difference is that we've given the bottom box `box-sizing: border-box`, but we've left the top box with its default behaviour.

First, the HTML:

```
1 <div class="one"></div>
2 <div class="two"></div>
```

Now the CSS:

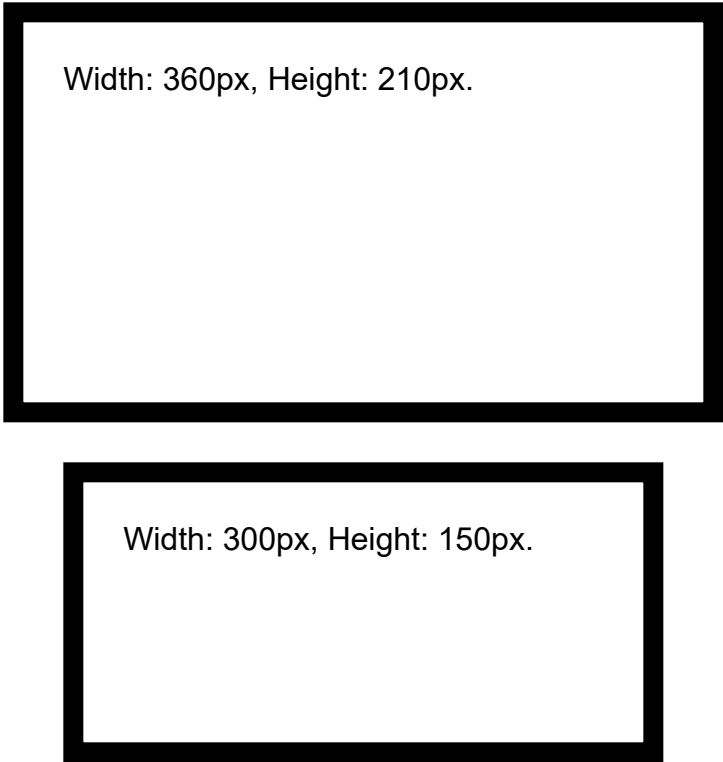
```
1 html {
2   font-family: sans-serif;
3   background: #ccc;
}
```

```
4
5  .one, .two {
6    background: red;
7    width: 300px;
8    height: 150px;
9    padding: 20px;
10   border: 10px solid black;
11   margin: 20px auto;
12 }
13
14 .two {
15   box-sizing: border-box;
16 }
17
```

Finally, a little JavaScript to measure the overall computed widths:

```
1  var one = document.querySelector('.one');
2  var two = document.querySelector('.two');
3
4  function outputWH(box) {
5    var width = box.offsetWidth;
6    var height = box.offsetHeight;
7    box.textContent = 'Width: ' + width + 'px, Height: ' + height + 'px.';
8  }
9
10 outputWH(one);
11 outputWH(two);
```

This gives us the following result:



Width: 360px, Height: 210px.

Width: 300px, Height: 150px.

So what's happened here? The width and height of the first box are equal to content + padding + border, as you'd expect. The second box however has its width and height equal to the width and height set on the content via CSS. The padding and border haven't added onto the total width and height; instead, they've taken up some of the content's space, making the content smaller and keeping the total dimensions the same.

You can also see this example running live at [box-sizing-example.html](#) ([see the source code](#)).

Box display types [↗](#)

Everything we've said so far applies to boxes that represent block level elements. However, CSS has other types of boxes that behave differently. The type of box applied to an element is specified by the `display` property.

Common display types [↗](#)

There are many different values available for `display`, the three most common ones of which are `block`, `inline`, and `inline-block`.

- A `block` box is defined as a box that's stacked upon other boxes (i.e. content before and after the box appears on a separate line), and can have width and height set on it. The whole box model as described above applies to block boxes.
- An `inline` box is the opposite of a block box: it flows with the document's text (i.e. it will appear on the same line as surrounding text and other inline elements, and its content will break with the flow of the text, like lines of text in a paragraph.) Width and height settings have no effect on `inline` boxes; any padding, margin and border set on `inline` boxes will update the position of surrounding text, but will not affect the position of surrounding block boxes.
- An `inline-block` box is something in between the first two: It flows with surrounding text without creating line breaks before and after it like an `inline` box, but it can be sized using width and height and maintains its block integrity like a `block` box — it won't be broken across paragraph lines (an `inline-block` box that overflows a line of text will drop down onto a 2nd line, as there is not enough space for it on the first line, and it won't break across two lines.)

By default, block level elements have `display: block`; set on them, and inline elements have `display: inline`; set on them.

Note: See The box model article's Types of CSS boxes section for a more complete overview and an example.

Uncommon display types [↗](#)

There are also some less commonly-used values for the `display` property that you will come across in your travels. Some of these have been around for a while and are fairly well supported, while others are newer and less well supported. These values were generally created to make creating web page/application layouts easier. The most interesting ones are:

- `display: table` — allows you to emulate table layouts using non-table elements, without abusing table HTML to do so. To read more about this, See [CSS tables](#).
- `display: flex` — allows you to solve many classic layout problems that have plagued CSS for a long time, such as laying out a series of containers in flexible equal width columns, or vertically centering content. For more information, see [Flexbox](#).

- `display: grid` — gives CSS a native way of easily implementing grid systems, whereas it has traditionally relied on somewhat hack-ish CSS grid frameworks. Our CSS Grids article explains how to use traditional CSS grid frameworks, and gives a sneak peek at native CSS Grids.
-

What's next [↗](#)

After a quick little recap, we looked at some more advanced features of CSS for manipulating boxes, and finished up by touching on some advanced layout features. With this all behind us, we will now go on to look at backgrounds.

Overview: Styling boxes

Next

In this module [↗](#)

- Box model recap
 - Backgrounds
 - Borders
 - Styling tables
 - Advanced box effects
 - Creating fancy letterheaded paper
 - A cool looking box
-