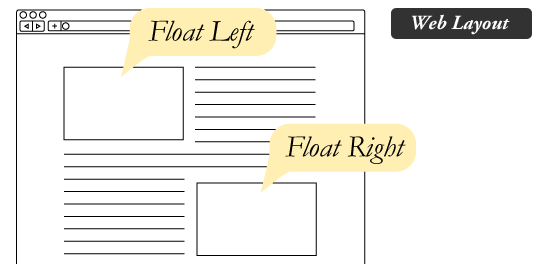
## What is the difference between classes and IDs in CSS?

* **IDs** — meant to be unique within the document. Can be used to identify an element when linking using a fragment identifier. Elements can only have one id attribute.
* **Classes** — can be reused on multiple elements within the document. Mainly for styling and targeting elements.

**Describe floats and how they work.**

Float is a CSS positioning property. Floated elements remain a part of the flow of the page, and will affect the positioning of other elements (e.g. text will flow around floated elements), unlike position: absolute elements, which are removed from the flow of the page.



**4 valid values for the float property**.

1. **Left** and **Right** float elements those directions respectively.
2. **None** (the default) ensures the element will not float.
3. **Inherit** which will assume the float value from that elements parent element.

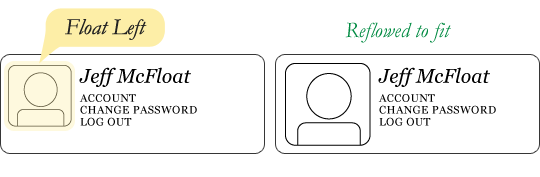
**Difference between float and position: absolute**

1. Absolutely positioned page elements are **removed** from the flow of the webpage, like when the text box in the print layout was told to ignore the page wrap.
2. Absolutely positioned page elements will not affect the position of other elements and other elements will not affect them, whether they touch each other or not.

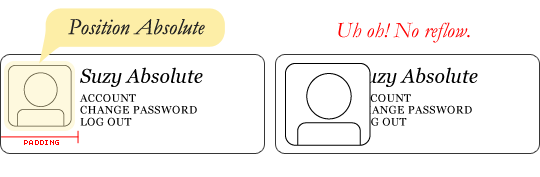
**Want to understand better? See below example…**

*“Floats are also helpful for layout in smaller instances.”*

If we use float for our little avatar image, when that image changes size the text in the box will reflow to accommodate:

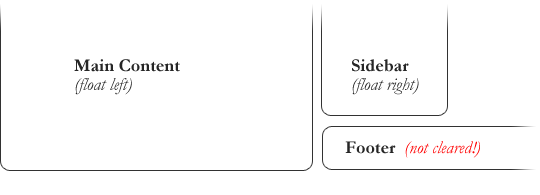


This same layout could be accomplished using relative positioning on container and absolute positioning on the avatar as well. In doing it this way, the text would be unaffected by the avatar and not be able to reflow on a size change.



### **Clearing the Float**

Float's sister property is clear. An element that has the clear property set on it will not move up adjacent to the float like the float desires, but will move itself down past the float. Again an illustration probably does more good than words do.

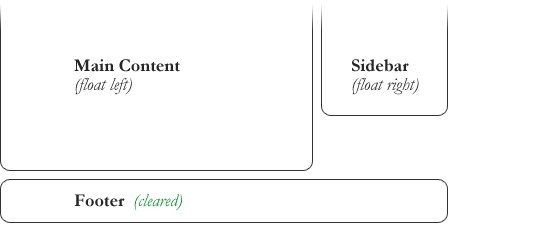


To fix this problem, the footer can be cleared to ensure it stays beneath both floated columns.

#footer {

clear: both;

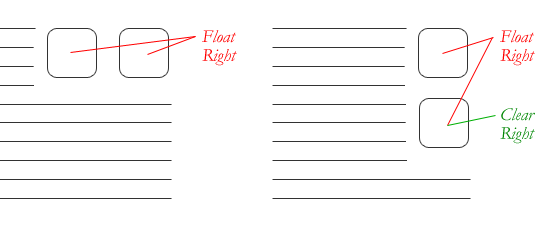
}



**4 valid values for Clear:**

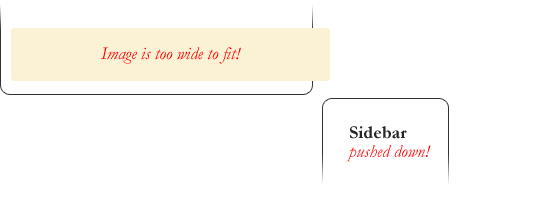
1. **Both** is most commonly used, which clears floats coming from either direction.
2. **Left** and **Right** can be used to only clear the float from one direction respectively.
3. **None** is the default, which is typically unnecessary unless removing a clear value from a cascade.
4. **Inherit** would be the fifth, but is strangely not supported in Internet Explorer.

* Clearing only the left or right float, while less commonly seen in the wild, definitely has its uses.



### **Problems with Floats**

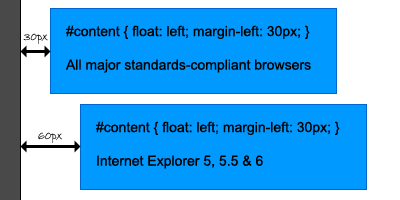
1. **Pushdown** is a symptom of an element inside a floated item being wider than the float itself (typically an image). Most browsers will render the image outside the float. IE will expand the float to contain the image, often drastically affecting layout. A common example is an image sticking out of the main content push the sidebar down below.



Quick fix: Make sure you don't have any images that do this, use overflow: hidden to cut off excess.

1. **Double Margin Bug** - Another thing to remember when dealing with IE 6 is that if you apply a margin in the same direction as the float, it will [double the margin](http://www.cssnewbie.com/double-margin-float-bug/).

So what is the double-margin float bug? If you were to float an element to the left and give it a 20-pixel left margin, in IE the margin would actually be 40 pixels wide. It only happens when the margin is in the same direction as the float, but it happens to both left and right floats. At least IE is consistent in its inconsistency.



Quick fix: set display: inline on the float, and don't worry it will remain a block-level element.

#content {

float: left;

width: 500px;

padding: 10px 15px;

margin-left: 20px; }

#content {

float: left;

width: 500px;

padding: 10px 15px;

margin-left: 20px;

display: inline; }

**And why does a display property fix our margin problem?**

In all truthfulness, applying a display property to a float should do exactly nothing (unless it’s display: none, that is). Floats are by definition block-level elements, and cannot become inline elements. And even IE knows this – after you apply this rule all browsers including Internet Explorer will continue to treat your floats like block-level elements. But now IE will also start behaving itself when it comes to your margins.

1. The **3px Jog** is when text that is up next to a floated element is mysteriously kicked away by 3px like a weird forcefield around the float.

Quick fix: set a width or height on the affected text.

1. In IE 7, the **Bottom Margin Bug** is when if a floated parent has floated children inside it, bottom margin on those children is ignored by the parent.

Quick fix: using bottom padding on the parent instead.

## Describe z-index and how stacking context is formed.

The z-index property in CSS controls the vertical stacking order of elements that overlap. As in, which one appears as if it is physically closer to you. z-index only affects elements that have a [position](https://css-tricks.com/almanac/properties/p/position/) value other than static (the default).

The bottom of the stack is the furthest from the user, the top of the stack is the nearest to the user:

| | | |

| | | | ⇦ ☻

| | | user

z-index: canvas -1 0 1 2

**'z-index'**

|  |  |
| --- | --- |
| *Value:* | auto | [<integer>](https://www.w3.org/TR/CSS2/syndata.html#value-def-integer) | [inherit](https://www.w3.org/TR/CSS2/cascade.html#value-def-inherit) |
| *Initial:* | auto |
| *Applies to:* | positioned elements |
| *Inherited:* | no |
| *Percentages:* | N/A |
| *Media:* | [visual](https://www.w3.org/TR/CSS2/media.html#visual-media-group) |
| *Computed value:* | as specified |

**For a positioned box, the**[**'z-index'**](https://www.w3.org/TR/CSS2/visuren.html#propdef-z-index)**property specifies:**

1. The stack level of the box in the current stacking context.
2. Whether the box establishes a stacking context.

**Values have the following meanings:**

**[<integer>](https://www.w3.org/TR/CSS2/syndata.html" \l "value-def-integer)**

This integer is the stack level of the generated box in the current stacking context. The box also establishes a new stacking context. This can be positive, **negative**, or 0.

**auto**

The stack level of the generated box in the current stacking context is 0. The box **does not establish a new stacking context unless it is the root element.**

**Rules:**

1. When the z-index and position properties aren’t involved, the rules are pretty simple: basically, the stacking order is the same as the order of appearance in the HTML.
2. Elements with non-static positioning will always appear on top of elements with default static positioning.
3. If you try to set a z-index on an element with no position specified, it will do nothing.

**References**

* <https://css-tricks.com/almanac/properties/z/z-index/>
* <https://philipwalton.com/articles/what-no-one-told-you-about-z-index/>

## Stacking Contexts

Groups of elements with a common parent that move forward or backward together in the stacking order make up what is known as a stacking context. A full understanding of stacking contexts is key to really grasping how z-index and the stacking order work.

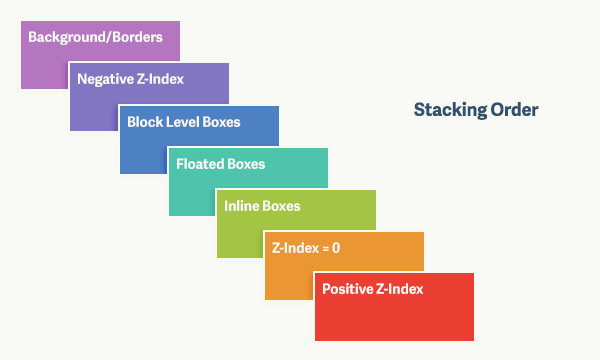
Every stacking context has a single HTML element as its root element. When a new stacking context is formed on an element, that stacking context confines all of its child elements to a particular place in the stacking order. That means that if an element is contained in a stacking context at the bottom of the stacking order, there is no way to get it to appear in front of another element in a different stacking context that is higher in the stacking order, even with a z-index of a billion!

New stacking contexts can be formed on an element in one of three ways:

1. When an element is the root element of a document (the <html> element)
2. When an element has a position value **other than static** and a z-index value **other than auto**
3. When an element has an **opacity value less than 1**

**Update:** In addition to opacity, several newer CSS properties also create stacking contexts. These include: [transforms](http://www.w3.org/TR/css3-transforms/), [filters](http://www.w3.org/TR/filter-effects/), [css-regions](http://www.w3.org/TR/css3-regions/), [paged media](http://www.w3.org/TR/css3-page/), and possibly others. As a general rule, it seems that if a CSS property requires rendering in an offscreen context, it must create a new stacking context.

There are **seven possible levels** on each stacking context, which are listed below.



## What are the various clearing techniques and which is appropriate for what context?

* Empty div method - <div style="clear:both;"></div>.
* Clearfix method — Refer to the .clearfix class above.
* overflow: auto or overflow: hidden method - Parent will establish a new block formatting context and expand to contains its floated children.

The .clearfix hack uses a clever CSS pseudo selector (:after) to clear floats.

.clearfix:after {  
content: '.';  
visibility: hidden;  
display: block;  
height: 0;  
clear: both;  
}

In large projects, I would write a utility .clearfix class and use them in places where I need it. overflow: hiddenmight clip children if the children is taller than the parent and is not very ideal..

### **What are CSS Sprites?**

In short: CSS Sprites are a means of combining multiple images into a single image file for use on a website, to help with performance.

The [history of sprites](http://en.wikipedia.org/wiki/Sprite_(computer_graphics)), dating back to 1975, should help clear things up.

To summarize: the term "sprites" comes from a technique in computer graphics, most often used in video games. The idea was that the computer could fetch a graphic into memory, and then only display parts of that image at a time, which was faster than having to continually fetch new images. The sprite was the big combined graphic.

CSS Sprites is pretty much the exact same theory: get the image once, and shift it around and only display parts of it. This reduces the overhead of having to fetch multiple images.

## Simple Example

Instead of using three separate images, we use this single image ("img\_navsprites.gif"):

navigation images

With CSS, we can show just the part of the image we need.

In the following example the CSS specifies which part of the "img\_navsprites.gif" image to show:

<!DOCTYPE html>

<html>

<head>

<style>

#sprite {

width: 133px;

height: 44px;

background: url(img\_navsprites.gif);

}

#home {

width: 46px;

height: 44px;

background: url(img\_navsprites.gif) 0 0;

}

#next {

width: 43px;

height: 44px;

background: url(img\_navsprites.gif) -91px 0;

}

#prev {

width: 43px;

height: 44px;

background: url(img\_navsprites.gif) -47px 0;

}

</style>

</head>

<body>

<div > Image with all three icons<img id="sprite" src="img\_trans.gif" width="1" height="1"></div><br><br>

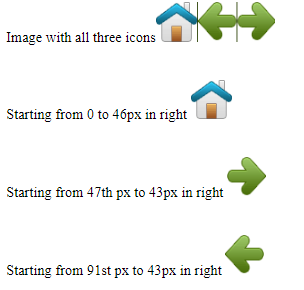
<div> Starting from 0 to 46px in right <img id="home" src="img\_trans.gif" width="1" height="1"></div><br><br>

<div>Starting from 47th px to 43px in right <img id="next" src="img\_trans.gif" width="1" height="1"></div><br><br>

<div>Starting from 91st px to 43px in right <img id="prev" src="img\_trans.gif" width="1" height="1"></div>

</body>

</html>



**Advantages:**

* Reduce the number of HTTP requests for multiple images (only one single request is required per spritesheet). But with HTTP2, loading multiple images is no longer much of an issue.
* Advance downloading of assets that won’t be downloaded until needed, such as images that only appear upon:hover pseudo-states. Blinking wouldn't be seen.

## How would you approach fixing browser-specific styling issues?

* After identifying the issue and the offending browser, use a separate style sheet that only loads when that specific browser is being used. This technique requires server-side rendering though.
* Use libraries like Bootstrap that already handles these styling issues for you.
* Use autoprefixer to automatically add vendor prefixes to your code.
* Use Reset CSS or Normalize.css.

Another answer from quora (<https://www.quora.com/How-do-you-approach-fixing-browser-specific-styling-issues>)

While I am developing the front-end of a website, I follow these guidelines:

1. Always build responsive layouts.
2. Check features of HTML5 and CSS3 on [caniuse.com](http://caniuse.com/) to see what browser support there is for something new I want to use and avoid things that have sketchy support at this time.
3. Develop all of the views at the same time and test across browsers and operating systems as I go to avoid getting something that looks nice in my favorite browser, but breaks in other places that I then have to fix later with the potential of having to start from scratch.
4. Realize that we don’t have to make the display on every browser exactly identical as long as what we make looks good in each browser. Graceful degradation means that your design is able to fall back to a decent, but simplified, version of the layout rather than just looking broken in browsers that don’t support certain features of HTML or CSS.
5. Keep up-to-date on what browsers and systems my target-audience is using, either by looking at Google Analytics on the existing site I’m replacing for the client, or periodically checking [Stat Counter](http://gs.statcounter.com/).
6. Use [BrowserStack](https://www.browserstack.com/) for more intense cross-browser testing.
7. Use the knowledge and experience I’ve gained from my years in web development to know in advance what to avoid or what to do to make something work in all the browsers.

## Describe pseudo-elements and discuss what they are used for.

A CSS pseudo-element is a keyword added to a selector that lets you style a specific part of the selected element(s). They can be used for decoration (:first-line, :first-letter) or adding elements to the markup (combined with content: ...) without having to modify the markup (:before, :after).

* :first-line and :first-letter can be used to decorate text.
* Used in the .clearfix hack as shown above to add a zero-space element with clear: both.
* Triangular arrows in tooltips use :before and :after.

## Describe pseudo-class selector and discuss what they are used for.

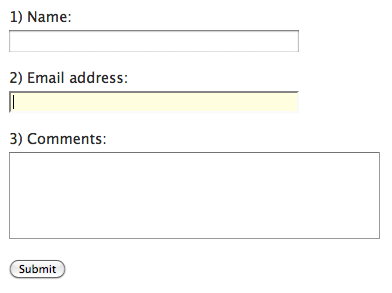
A [CSS](https://developer.mozilla.org/en-US/docs/Web/CSS) **pseudo-class** is a keyword added to a selector that specifies a special state of the selected element

### Link-related pseudo class selectors

1. [:link](https://css-tricks.com/almanac/selectors/l/link/) - This selects only those that do don't have an href attribute, thus is essentially the same as a[href].
2. [:visited](https://css-tricks.com/almanac/selectors/v/visited/) - Selects links that have already been visited by the current browser.
3. [:hover](https://css-tricks.com/almanac/selectors/h/hover/) - When the mouse cursor rolls over a link, that link is in it's hover state and this will select it.
4. [:active](https://css-tricks.com/almanac/selectors/a/active/) - Selects the link while it is being activated (being clicked on or otherwise activated).

### Input & link related pseudo class selectors

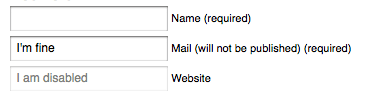
[:focus](https://css-tricks.com/almanac/selectors/f/focus/) -:focus will select links that are the current focus of the keyboard. This is not limited to links, but can be used (and really should be used) on inputs and textareas as well.



[:target](https://css-tricks.com/almanac/selectors/t/target/) - The target pseudo class is used in conjunction with IDs, and match when the hash tag in the current URL matches that ID. So if you are at URL www.yoursite.com/#home then the selector #home:target will match.

[:enabled](https://css-tricks.com/almanac/selectors/e/enabled/) - Selects inputs that are in the default state of enabled and ready to be used.

[:disabled](https://css-tricks.com/almanac/selectors/d/disabled/) - Selects inputs that have the disabled attribute. A lot of browsers will make the input a faded out gray, you can control that with this selector.

Form using the :disabled attribute.

[:checked](https://css-tricks.com/almanac/selectors/c/checked/) - Selects checkboxes that are, wait for it, checked.

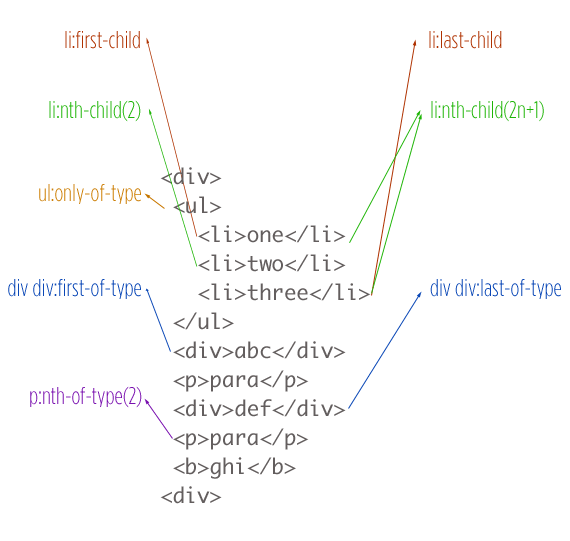
[:indeterminate](https://css-tricks.com/indeterminate-checkboxes/) - Selects radio buttons that are in the purgatory state of neither chosen or unchosen (like when a page loads with radio button choices but no default is set).

https://css-tricks.com/wp-content/csstricks-uploads/radiopurgatory.pngSet of radio buttons in purgatory. Or more accurately, in their :indeterminate status.

[:required](https://css-tricks.com/almanac/selectors/r/required/) - Selects inputs with the required attribute.  
[:optional](https://css-tricks.com/almanac/selectors/o/optional/) - Selects inputs that do not have the required attribute.

[:read-only / :read-write](https://css-tricks.com/almanac/selectors/r/read-write-read/) - Selects elements based on a combination of readonly and disabled attributes.

### Position/Number-based pseudo class selectors



## Explain your understanding of the box model and how you would tell the browser in CSS to render your layout in different box models.

The CSS box model is responsible for calculating:

* How much space a block-level element takes up.
* Whether or not borders and/or margins overlap, or collapse.
* A box’s dimensions.

The box model has the following rules:

* The dimensions of a block element are calculated by width, height, padding, borders, and margins.
* If no height is specified, a block element will be as high as the content it contains, plus padding (unless there are floats, for which see below).
* If no width is specified, a non-floated block element will expand to fit the width of its parent minus padding.
* The height of an element is calculated by the content's height.
* The width of an element is calculated by the content's width.
* By default, paddings and borders are not part of the width and height of an element.

## What does \* { box-sizing: border-box; } do? What are its advantages?

* By default, elements have box-sizing: content-box applied, and only the content size is being accounted for.
* box-sizing: border-box changes how the width and height of elements are being calculated, border and padding are also being included in the calculation.
* The height of an element is now calculated by the content's height + vertical padding + vertical border width.
* The width of an element is now calculated by the content's width + horizontal padding + horizontal borderwidth.

## List as many values for the display property that you can remember.

* none, block, inline, inline-block, table, table-row, table-cell, list-item.

## What’s the difference between inline and inline-block?

I shall throw in a comparison with block for good measure.

**Block**

* Size — Fills up the width of its parent container.
* Positioning — Start on a new line and tolerates no HTML elements next to it (except when you add float).
* Can specify width and height — Yes.
* Can be aligned with vertical-align — Yes.
* Margins and paddings — All sides respected.

**Inline-Block**

* Size — Depends on content.
* Positioning — Flows along with other content and allows other elements beside.
* Can specify width and height — Yes.
* Can be aligned with vertical-align — Yes.
* Margins and paddings — All sides respected.

**Inline**

* Size — Depends on content.
* Positioning — Flows along with other content and allows other elements beside.
* Can specify width and height — No. Will ignore if being set.
* Can be aligned with vertical-align — Only horizontal sides respected. Vertical sides, if specified, do not affect layout. Vertical space it takes up depends on line-height, even though the borderand padding appear visually around the content.
* Margins and paddings — Becomes like a block element where you can set vertical margins and paddings.

## What’s the difference between a relative, fixed, absolute and static-ally positioned element?

A positioned element is an element whose computed position property is either relative, absolute, fixed or sticky.

* **static** - The default position; the element will flow into the page as it normally would. The top, right, bottom, left and z-index properties do not apply.
* **relative** - The element's position is adjusted relative to itself, without changing layout (and thus leaving a gap for the element where it would have been had it not been positioned).
* **absolute** - The element is removed from the flow of the page and positioned at a specified position relative to its closest positioned ancestor if any, or otherwise relative to the initial containing block. Absolutely positioned boxes can have margins, and they do not collapse with any other margins. These elements do not affect the position of other elements.
* **fixed** - The element is removed from the flow of the page and positioned at a specified position relative to the viewport and doesn't move when scrolled.
* **sticky** - Sticky positioning is a hybrid of relative and fixed positioning. The element is treated as relative positioned until it crosses a specified threshold, at which point it is treated as fixed positioned.

## Have you played around with the new CSS Flexbox or Grid specs?

Yes. Flexbox is mainly meant for 1-dimensional layouts while Grid is meant for 2-dimensional layouts.

Flexbox solves many common problems in CSS, such as vertical centering of elements within a container, sticky footer, etc. Bootstrap and Bulma are based on Flexbox, and it is probably the recommended way to create layouts these days. Have tried Flexbox before but ran into some browser incompatibility issues (Safari) in using `flex-grow`, and I had to rewrite my code using `inline-blocks` and math to calculate the widths in percentages, it wasn’t a nice experience.

Grid is by far the most intuitive approach for creating grid-based layouts (it better be!) but browser support is not wide at the moment.

## Is there any reason you’d want to use translate() instead of absolute positioning, or vice-versa? And why?

translate() is a value of CSS transform. Changing transform or opacity does not trigger browser reflow or repaint, only compositions, whereas changing the absolute positioning triggers reflow. transform causes the browser to create a GPU layer for the element but changing absolute positioning properties uses the CPU. Hence translate() is more efficient and will result in shorter paint times for smoother animations.

When using translate(), the element still takes up its original space (sort of like position: relative), unlike in changing the absolute positioning.