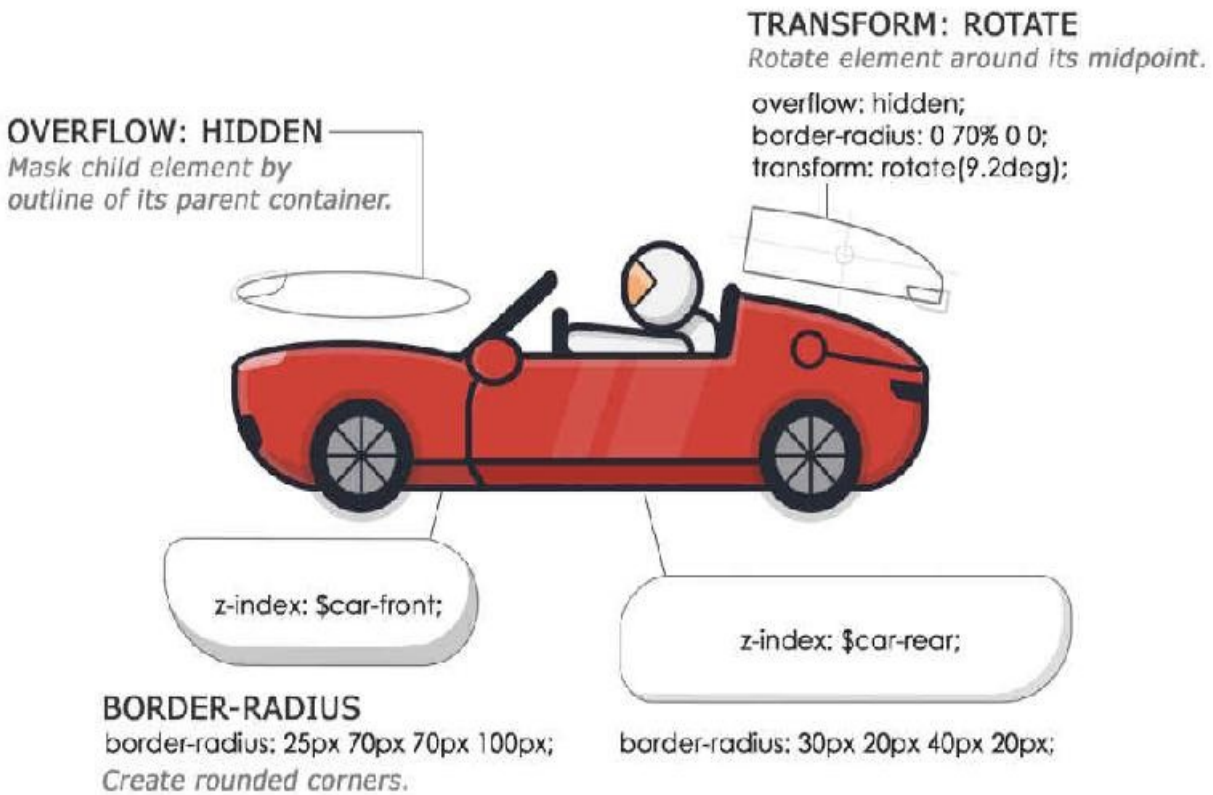
# CSS

## Cascading Style Sheets

# VISUAL DICTIONARY

**BOX MODEL FLEX CSS GRID TRANSFORMS**




LEARNING CURVE BOOKS

CSS Cascading Style Sheets VISUAL DICTIONARY

# Disclaimer for Kindle, IOS and Android-based Readers

You are reading **Kindle e-Book** conversion of the **original print manuscript.** To accurately transfer a technical manuscript containing **CSS Source Code** to the e-book format acceptable for Kindle readers often takes extraordinary editing measures. As a long time publisher Learning Curve strives to provide the best experience for our Kindle readers given limitations of the medium. For example, source code is traditionally difficult to implement correctly for Kindle format. Although Learning Curve books -- ***originally designed as a desk reference*** -- are best consumed in the paperback format. Nonetheless, we've gone to a great extent to make sure that the content of this technical book is readable on as many Amazon Kindle, Apple and Android-based devices as possible in its most acceptable format.




**Title:** CSS -- Visual Dictionary **Edition:** I **Release:** June 1, 2018
**Genre:** Web Design & Software Education **Publisher:** Learning Curve Books
**Imprint:** Independently published **ISBN:** 9781983065637 **Author:** Greg Sidelnikov **Contact: greg.sidelnikov@gmail.com** The primary purpose of Learning Curve Book publishing company is to provide *effective education* for web designers, software engineers and all readers who are interested in being edified in the area of web development. This edition of *CSS -- Visual Dictionary* was created to speed up the learning process of Cascading Style Sheets -- a language for decorating HTML elements. For questions and comments about the book you may contact the author or send an email directly to our office at the email address mentioned below.

**Special Offers & Discounts Available** Schools, libraries and educational organizations may qualify for special prices.

Get in touch with our distribution department at **hello@learningcurvebook.net**


**CSS Visual Dictionary** Several months have gone into creation of the book you are holding in your hands (or on your device) right now. Indeed, ***CSS -- Visual Dictionary*** is a work of love and hard labor. Thoughtfully created to help maximize your journey on your way to expanding your knowledge of CSS -- Cascading Style Sheets. A language for decorating HTML elements.
We hope that this volume will serve as a faithful guide on your desk in the years to come.
**Special Thanks To: Sasha Tran** *Front End Developer* for contributing the CSS rendition of the Tesla and complete CSS source code. If you like her CSS art work, you can get a hold of her via her website ***sashatran.com***, her *Codepen.io* account at ***https://codepen.io/sashatran/*** or on Twitter ***@sa_sha26***.
**Fabio Di Corleto** *Graphic Designer* for contributing the original concept work for the Tesla in space image. If you're looking for a talented Graphic Designer you can get in touch with him at ***fabiodicorleto@gmail.com*** or via his *Instagram* and Dribbble pages. His username ***fabiodicorleto*** is the same across his social media accounts.
...for their contributions and licensing permission to use their work in this edition of *CSS Visual Dictionary* published by ***Learning Curve*** book publishing company.

# CSS Properties and Values

CSS has **415** unique properties.

You can verify this with a simple *JavaScript* code snipplet as follows:

Source Code 0

```
    var element = document.createElement("div");
var count = 0;
for (index in element.style) p++;
console.log(p); // outputs 415 as of June 1st, 2018.
```

There may be more or less in the future as new features are being added to the specification and old ones deprecate.

A large number of CSS properties that are rarely in use (*or still don't have full browser support across all major browsers*) were skipped from the contents of this book. They would only create unneeded clutter.
Instead, in this book we focused only on CSS properties that are in common use by web designers and developers today. A great deal of effort went into creation of **CSS Grid** and **Flex** diagrams in particular.

# Placement

CSS code can be saved in a separate, external file and included as follows:

Source Code 1

```
<html>
<head>
<title>Welcome to my website.</title>
<link rel = "stylesheet" type = "text/css" href = "style.css" />
</head>
<body>CSS style instructions stored in "style.css" will be applied to this page.
</body>
</html>
```

Or you can type it directly into your HTML document between:
**<script type = "text/css">*here*</*script*>** tags.

# Simple Assignments

To assign a value to a property of an HTML element whose ***id*** is "box", you would write something like this:

Source Code 2

```
#box { property: value; }
```

Depending on the property, the value can be a measure of space specified in *pixels*, *pt*, *em* or *fr* units, a *color*... in named *red*, *blue*, *black*, etc..., hexadecimal ***#0F0*** or ***#00FF00***... or **rgb**(***r***, ***g***, ***b***) formats.

Other times the value is unique to a specific property name that cannot be used with any other property. For example, the CSS **transform** property can take a value called **rotate** that takes an *angle* in degrees -- here, CSS requires that you append "deg" to the numeric degree value:

Source Code 3

```
#box { transform: rotate(45deg); } /* rotate this element by 45 degrees in clock-wise direction */
```

# CSS Comments

CSS only supports "block comment" syntax for creating in-code comments. By surrounding a block of text or CSS code with ***/* comment */*** symbols.

Source Code 4

color: #FFFFFF; ***/* Set font color to white using Hexadecimal value */***

Source Code 5

color: #FFF; ***/* Set font color to white using short Hexadecimal value */***

Source Code 6

color: white; ***/* Set font color to white using named value */***

Source Code 7

color: rgb(255,255,255); ***/* Set font color to white using an RGB value */***

Source Code 8

color: var(--white-color); ***/* Set font color to white using a CSS variable */***

You can also comment out entire sections of CSS code to temporarily disable them for future use:

Source Code 9

```
/*
    content: "hello";
border: 1px solid gray;
color: #FFFFFF; */
```

CSS does not support inline syntax ***// inline comments are not allowed*** or rather... have no effect on the browser's CSS interpreter. Other than they might confuse it a bit!

# Assignment Patterns

You can use ***property: value*** pair combination to set background images, colors and other basic properties of HTML elements.
You could alternatively use ***property: value value value*** to assign multiple values to a single property, to avoid redundant declarations. These are called ***shorthands***. They usually separate multiple property values by space.
But CSS has undergone considerable upgrades over the years. Before we begin exploring the visual diagrams describing each CSS property it is imperative to understand how CSS interprets property and value patterns.
The majority of properties use these patterns:

Source Code 10
property: value; ***/* The most common pattern */***

Source Code 11
property: value, value, value;***/* separated by comma */***

Source Code 12
property: value value value; ***/* separated by space*/***

Properties that refer to a size of something can also be calculated using **calc** keyword:

Source Code 13
property: calc(value[px]); ***/* calculated */***

Source Code 14
property: calc(value[%] - value[px]); ***/* calculated between % and px -- ok. */***

Source Code 15
property: calc(value[%] - value[%]}; ***/* calculated between % and % -- ok. */***

Source Code 16
property: calc(value[px] + value[px]); ***/* add px to px -- ok. */***

Source Code 17
property: calc(value[px] - value[px]); ***/* subtract px from px -- ok. */***

Source Code 18

property: calc(value[px] * number}); ***/* multiply px by number -- ok. */***

Source Code 19

property: calc(value[px] / number});***/* divide px by number -- ok. */***

Source Code 20

property: calc(number / value[px]}); ***/* divide number by px -- error. */***

The last example will produce an error. When using **calc** you cannot divide a *number* by a value specified in pixels (*px*).

# CSS Variables

You can also use CSS variables to avoid redundancy when reusing the same values.

Source Code 21

element { --default-color: yellow } ***/* define variable --default-color */***

Source Code 22

element { --variable-name: 100px; } ***/* define variable --variable-name */***

Source Code 23

element { background-color: var(--default-color); } ***/* set background color to --default-color variable */***

Source Code 24

element { width: var(--variable-name); } ***/* set width to 100px */***

# Sass/SCSS

Although SASS and SCSS are outside of the scope of this book, they are recommended for advanced CSS specialists. Note, that Sass/SCSS will not work out of the box in any browser. You need to install SASS compiler from the command line in order to enable it on your web server.

Source Code 25

**$a:** #E50C5E;
**$b:** #E16A2E;
.mixing-colors {
background-color: mix(**$a**, **$b**, 30%);
}

I encourage you to further study Sass/SCSS on your own, but only once you feel comfortable with standard CSS described in this book!

# The Idea Behind Cascading Style Sheets

*Cascading Style Sheets* are named this way for a reason. Imagine a waterfall with water running down, breaking against the stones beneath. Every one of those stones on which the water fell becomes wet. Similarly, every CSS style inherits the styles already applied to its parent HTML element.

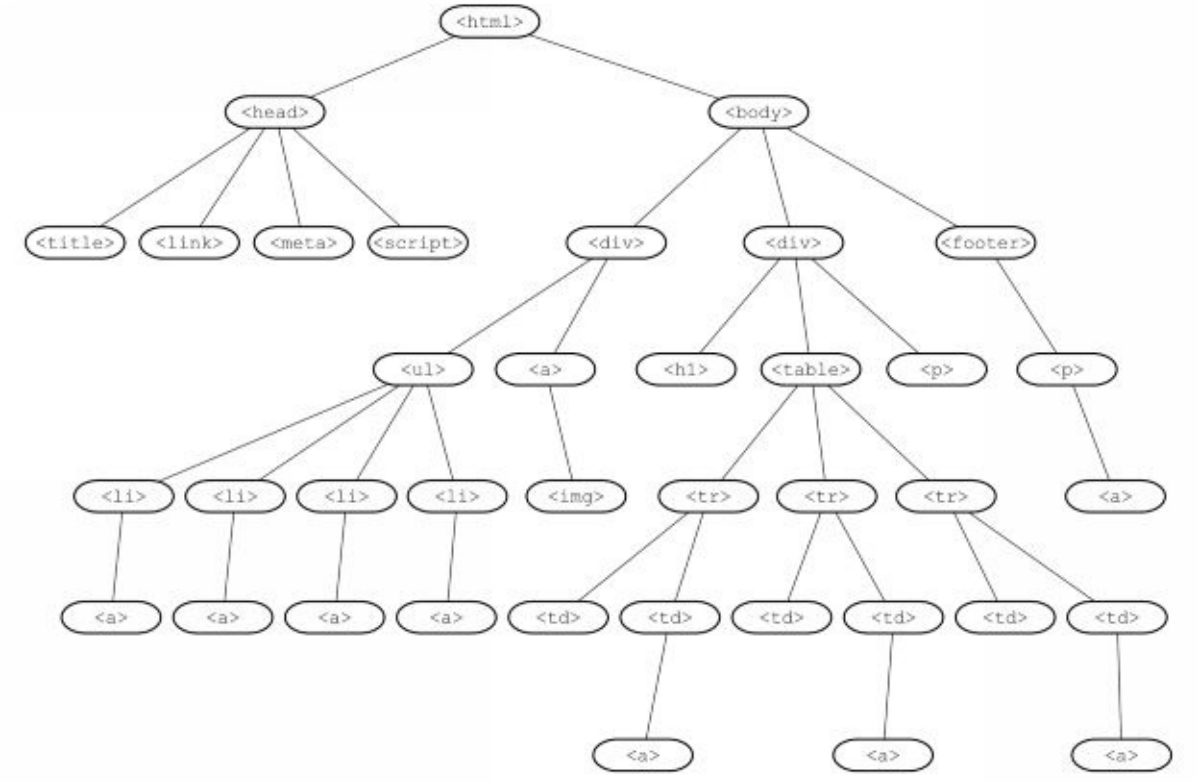



**Figure 1:** CSS selectors help traverse the Document Object Model.

CSS styles literally "trickle down" the DOM hierarchy, consisting of a tree-like structure of your website. The CSS language (specifically, by providing a number of CSS *selectors*) gives you ability to control this often quirky process.

Let's take a look at this simple website structure to demonstrate the basic concept behind CSS:


**Figure 2:** A few elements nested within the main website container. CSS is a lot like a pair of tweezers, that helps us pick elements we want to apply a certain style to.

If you apply black background to **<body>** tag then all of the nested elements within it will automatically inherit a black background: Source Code 26

**body** { **background**: black **color**: white; }; This style will "cascade" down the parent hierarchy, making all of the following HTML elements inherit white text on black background: Source Code 27

<**body**>
<**header**>Website header</**header**>
<**article**>Amazing article.</**article**>
<**footer**>Privacy Policy. <**span**>© 2018 Copyright.</**span**></**footer**>
</**body**> If you want to single out the footer and highlight the word *Privacy Policy* in **red** color and *2018 Copyright* in **green** color, you can expand on the cascading principle further by applying these CSS commands: Source Code 28

**body** { **background**: black; **color**: white; };
**footer** { color: red; };
**footer** span { color: green; }; Note that there is a space between footer and span. In CSS, a space is an actual CSS selector character. It means: "find within of the previously specified tag" (which is "footer" in this example.)

# CSS Selectors

Source Code 29

#id { } **/* Select a single element whose id attribute is "id" */**

Source Code 30

.class1 { } **/* Select all elements whose class name is "class1" */**

Source Code 31

#parent .class1 { } **/* Select all elements whose class name is "class1" cascading under another parent element whose id is "parent" */**

# Forgiving Nature

Because it was designed for environments where downloading the full copy of a website is not always guaranteed, CSS is one of the most forgiving languages, similar to HTML. If you make mistakes, or for some reason the page didn't finish loading completely, CSS code will degrade gracefully to as much as it *can* interpret. Ironically, this means you can still use the ***// inline comments*** but you probably shouldn't.

# Common

Some of the most common CSS property and value combinations:

Source Code 32

```
    color: #FFFFFF; /* Set font color to white */
```

Source Code 33

```
    background-color: #000000; /* Set background color to black */
```

Source Code 34

```
    border: 1px solid blue; /* Create 1px-thick blue border around the element */
```

Source Code 35

```
    font-family: Arial, sans-serif; /* Set font to Arial */
```

Source Code 36

```
    font-size: 16px; /* Set font size to 16px */
```

Source Code 37

```
    padding: 32px; /* Add padding 32px thickness in size */
```

Source Code 38

```
    margin: 16px; /* Add 16 pixels of margin around the content area */
```

# Shorthand Properties

Let's assign 3 different properties that contribute to the appearance of the background image of an HTML element:

Source Code 39

```
    background-color: #000000;
background-image: url("image.jpg");
background-repeat: no-repeat;
```

The same can be rewritten by using a single *shorthand* property **background**, separated by space: *background: background-color background-image background-repeat;*

Source Code 40

```
    background: #000000 url("image.jpg") no-repeat;
```

Shorthands also exist on various **CSS Grid** and **Flex** properties.

# Pseudo Selectors

In CSS a *pseudo-selector* is any selector that starts with a colon character (**:**) and usually appended to the end of the element name.
*Pseudo-selectors* **:first** and **:last** are used for selecting the very *first* or very *last* element from a list of children in a parent.
Another example is **:nth-child** for selecting a series of elements belonging to a row or column in a list of elements or even an HTML table.
Let's take at a few cases that demonstrate the use of *pseudo-selectors*:

**Figure 3:** table tr td:nth-child(2)

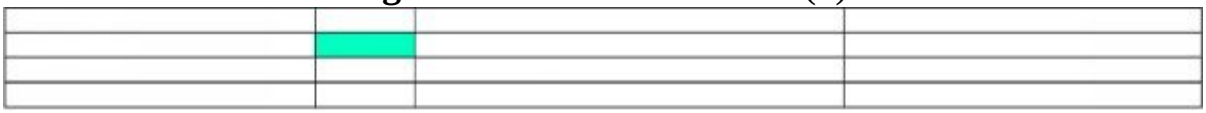


**Figure 4:** table tr:nth-child(2) td:nth-child(2)

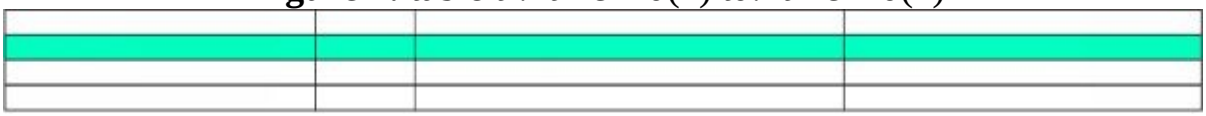


**Figure 5:** table tr:nth-child(2)

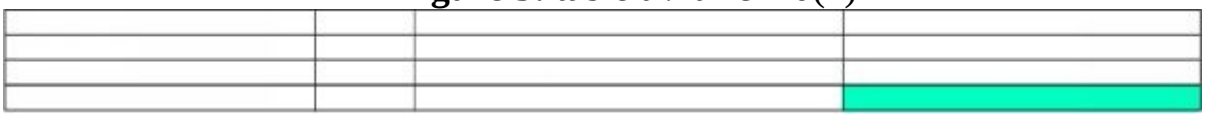


**Figure 6:** table tr:last td:last The same ***nth-child*** rules apply to all other *nested groups of elements*, like **ul** and **li** for example, and any other arbitrary parent/child combination.

What if you need to select absolutely all elements on the page or within some parent element? No problem!

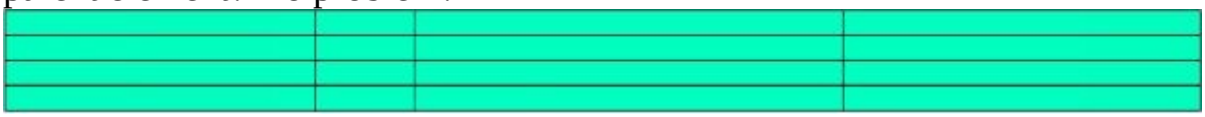


**Figure 7:** The star (*****) selector selects all elements within a parent. In this case **table *** selector was used.

Note that the ***space character*** itself is part of the selector. It helps you to drill down the hierarchy of elements via some parent element.
There are also **:before** and **:after** *pseudo-selectors* and later down the road in this book we will take a look at the visual diagrams that explain their relationship the HTML elements.

# CSS Box Model

The *box model* is the fundamental structure behind every HTML element. Traditionally, it consists of the content area, with padding, border and margin areas surrounding it.

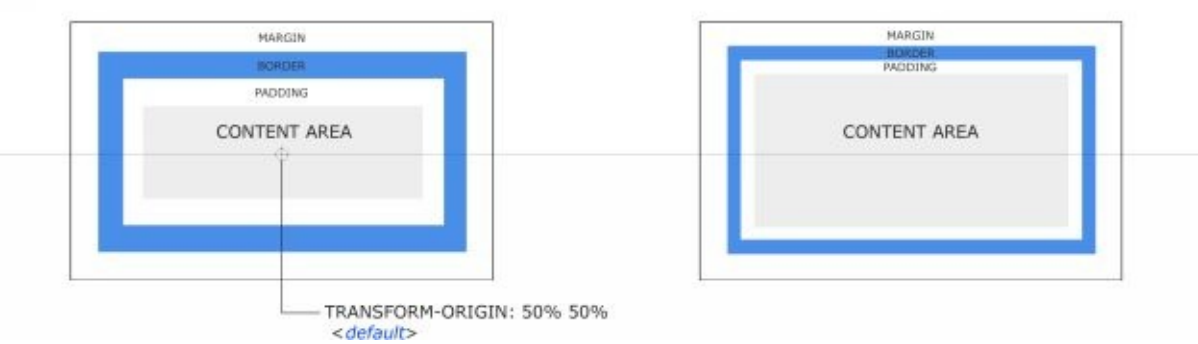



**Figure 8:** Just you regular HTML rectangle at first sight.

The most important thing about the box model is that by default its **box-sizing** property is set to **content-box**. I think it's a bit unfortunate because this means adding padding, border or margin will change the *physical* dimensions of its blocking area:


**Figure 9:** Note that the *value* **200px** of the **height** property of the element does not change, but its physical dimensions do, based on **box-sizing: [content-box|padding-box|border-box]**

.

There is no **margin-box** because margins by definition surround a given content area.

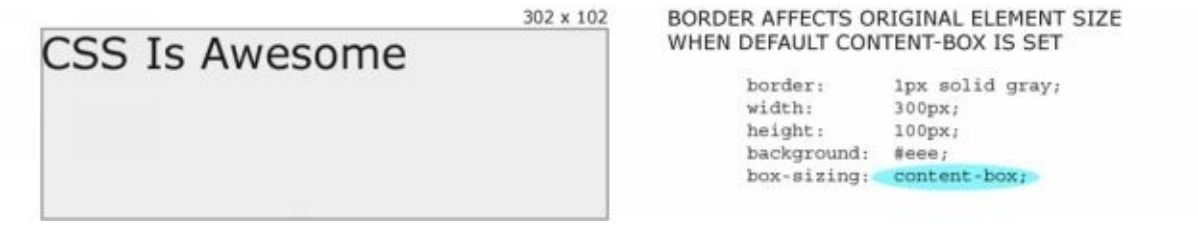



**Figure 10:** The **width** and **height** have increased by **2** pixels on each side because **1px** border was added to each of the 4 sides, when using default **content-box** model.


**Figure 11:** When both border and padding are present, the actual physical width becomes **334px x 134px**. This is **34** pixels greater than the original dimensions (**1px x 2 + 16px x 2 = 34px**).

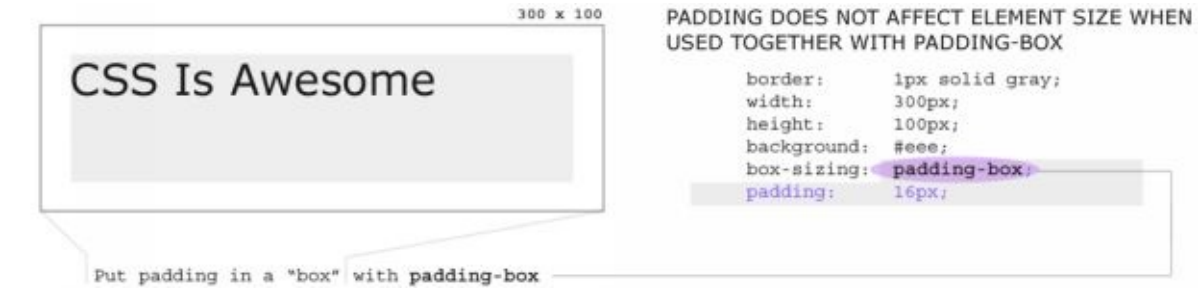



**Figure 12:** The **padding-box** value puts padding on the inside of content box. Now, the original dimensions are retained but the content is still padded.


**Figure 13:** Here we overwrite the original value of **border: 1px solid gray** from previous example to **border: 16px** and together with **padding: 16px** the original

width and height of the element are now padded by an extra **32px** pixels on each side, adding a total of **64px** to each dimension of the element.

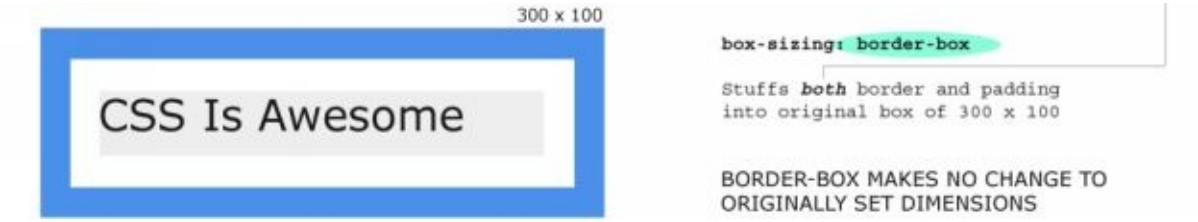



**Figure 14:** Using **border-box** will invert both the **border** and **padding** retaining original **width** and **height** of the element. This option is useful when you need to ensure your element will retain pixel-perfect dimensions, regardless of the size of its border or amount of padding.


**Figure 15:** There is no **margin-box** in CSS, because margins by definition always refer to the space surrounding the content.

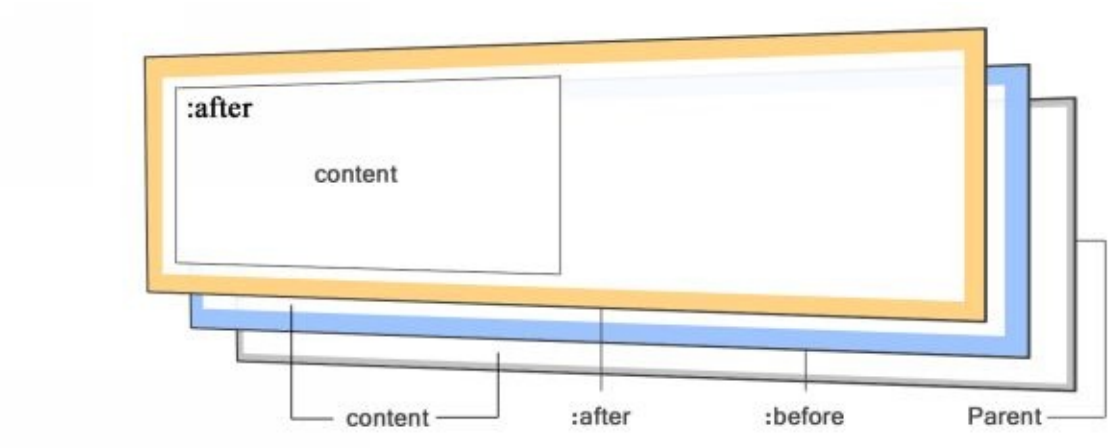



**Figure 16:** There is much more to a single HTML element than meets the eye.

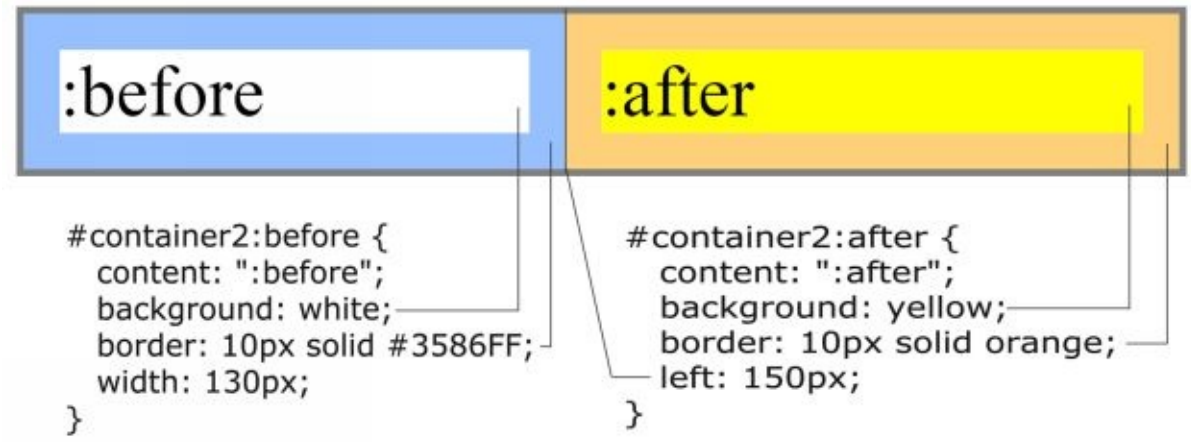



**Figure 17:** Both **:before** and **:after** elements are part of one single HTML element. You can even apply **position:absolute** to them and arrange them around without having to create any new elements!

# Position

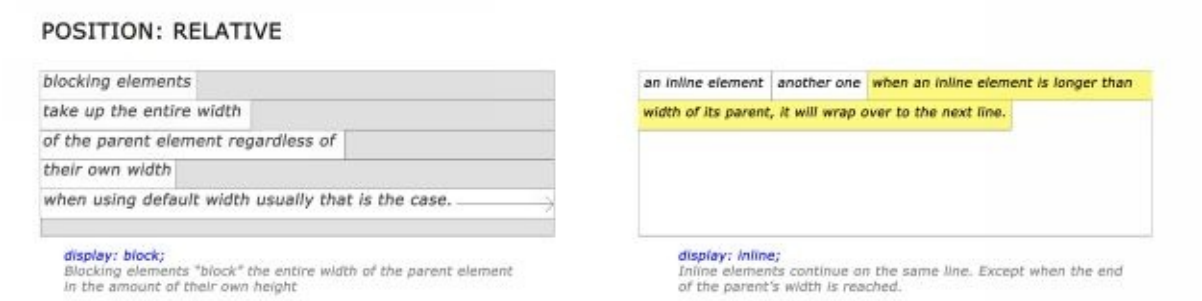



**Figure 18:** **position:relative** is the default value for both *blocking* and *inline* elements.

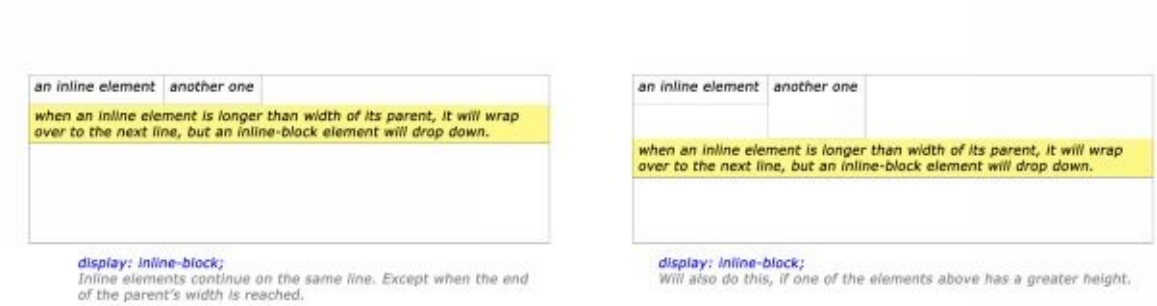



**Figure 19:** Using **display:inline-block** gives you the best of both worlds.

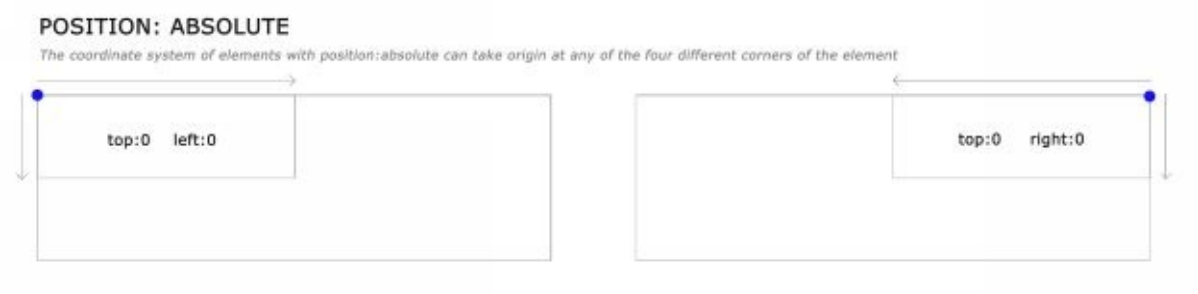



**Figure 20:** Using **position:absolute** together with **top & left** and **top & right** origin.


**Figure 21:** Using **position:absolute** together with **bottom & left** and **bottom & right** origin.

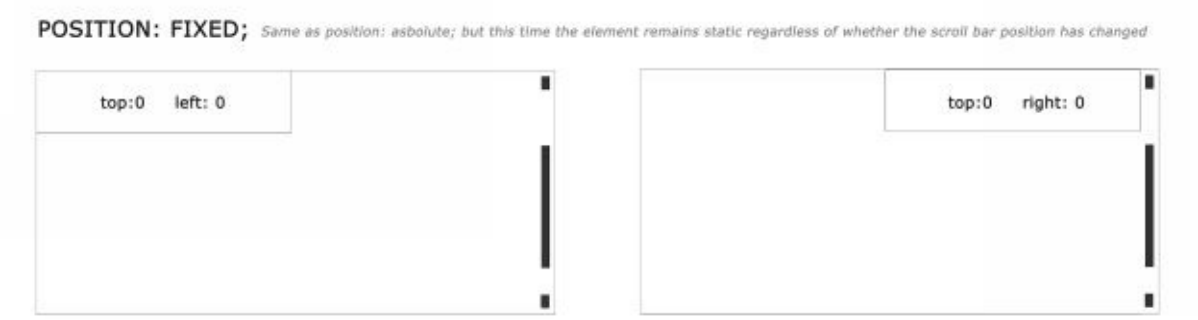



**Figure 22:** Using **position:fixed** works the same as **position:absolute** except that the scrollbar will not affect its position.


**Figure 23:** The origin point can be any corner of the element, depending on which property pair that was used (**top** & **left**, **top** & **right**, **bottom** & **left** or

**bottom** & **right**).

# Working With Text

We will not spend much space on diagrams for text because you have virtually seen that everywhere by just browsing websites or using social media websites. The primary properties for changing text in CSS are **font-family**, **font-size**. **color**, **font-weight** (*normal* or *bold*), **font-style** (*italic*, for example) and **text-decoration** (*underline* or *none*).


**Figure 24:** font-family:"CMU Classical Serif"; is the font used in the creation of this book. I suggest you check it out because it's one of the very best fonts around.


**Figure 25:** font-family: "CMU Bright"; is a variation of the CMU family fonts. Another nice-looking font!


**Figure 26:** font-family: Arial, sans-serif; is Google's favorite.


**Figure 27:** font-family: Verdana, sans-serif.

Note the sans-serif font is used here as a fall back font. You can specify even

more fonts, separating them by comma. If the first font on the list is not available or cannot be rendered by current browser, CSS will fall back to the next available font on the list. Times New Roman, shown in the last example here will be used if no other font was found.


**Figure 28:** Times New Roman. The default browser font.

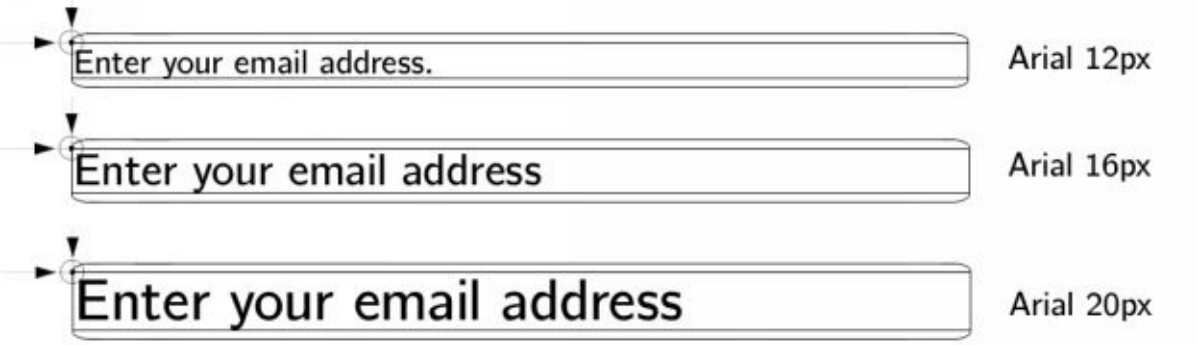



**Figure 29:** You can change the size of your font with **font-size** property. **16px** is the default "medium" size.

| pt | px | em | % | size | default sans-serif |
|---|---|---|---|---|---|
| 6pt | 8px | 0.5em | 50% | | Sample text |
| 7pt | 9px | 0.55em | 55% | | Sample text |
| 7.5pt | 10px | 0.625em | 62.5% | x-small | Sample text |
| 8pt | 11px | 0.7em | 70% | | Sample text |
| 9pt | 12px | 0.75em | 75% | | Sample text |
| 10pt | 13px | 0.8em | 80% | small | Sample text |
| 10.5pt | 14px | 0.875em | 87.5% | | Sample text |
| 11pt | 15px | 0.95em | 95% | | Sample text |
| 12pt | 16px | 1em | 100% | medium | Sample text |
| 13pt | 17px | 1.05em | 105% | | Sample text |
| 13.5pt | 18px | 1.125em | 112.5% | large | Sample text |
| 14pt | 19px | 1.2em | 120% | | Sample text |
| 14.5pt | 20px | 1.25em | 125% | | Sample text |
| 15pt | 21px | 1.3em | 130% | | Sample text |
| 16pt | 22px | 1.4em | 140% | | Sample text |
| 17pt | 23px | 1.45em | 145% | | Sample text |
| 18pt | 24px | 1.5em | 150% | x-large | Sample text |
| 20pt | 26px | 1.6em | 160% | | Sample text |
| 22pt | 29px | 1.8em | 180% | | Sample text |
| 24pt | 32px | 2em | 200% | xx-large | Sample text |
| 26pt | 35px | 2.2em | 220% | | Sample text |
| 27pt | 36px | 2.25em | 225% | | Sample text |
| 28pt | 37px | 2.3em | 230% | | Sample text |
| 29pt | 38px | 2.35em | 235% | | Sample text |
| 30pt | 40px | 2.45em | 245% | | Sample text |
| 32pt | 42px | 2.55em | 255% | | Sample text |
| 34pt | 45px | 2.75em | 275% | | Sample text |
| 36pt | 48px | 3em | 300% | | Sample text |

100% = 16px = medium

**Figure 30:** Font size can be specified using **pt**, **px**, **em** or **%** units. By default **100 %** is the same as **12pt**, **16px** or **1em**. Knowing this you can extrapolate values to arrive at either a bigger or smaller font relative to the default size.


**Figure 31:** **font-weight** is demonstrated here on custom *Raleway* font available via Google Fonts.

# Text Align

Aligning text within an HTML element is one of the most basic things you can do in CSS.


**Figure 32: text-align: left; is the default.**

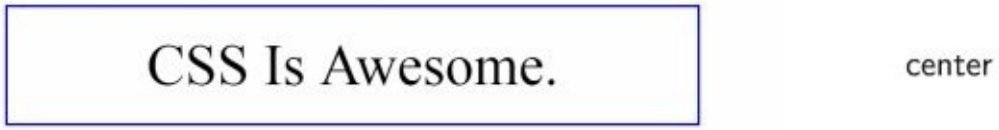



**Figure 33: text-align: center;**


**Figure 34: text-align: right;**

# Text Align Last

The **text-align-last** is the same as **text-align** except it refers only to the very last line of text in a paragraph:

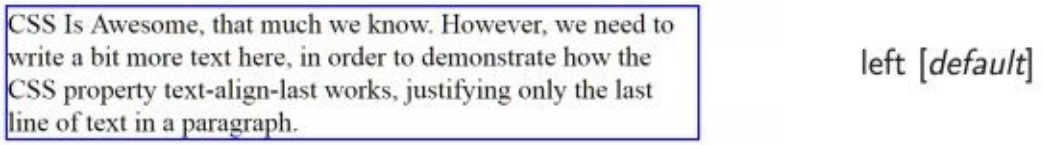



**Figure 35: text-align-last: left;**

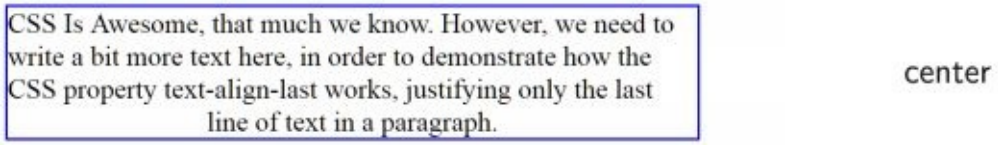



**Figure 36: text-align-last:center;**

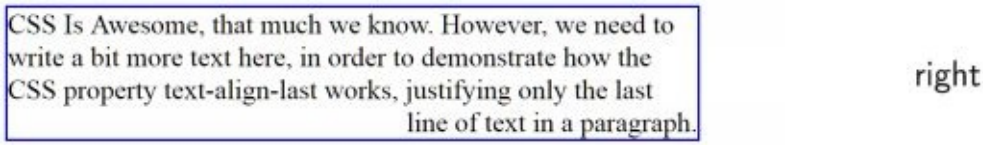



**Figure 37: text-align-last:right;**

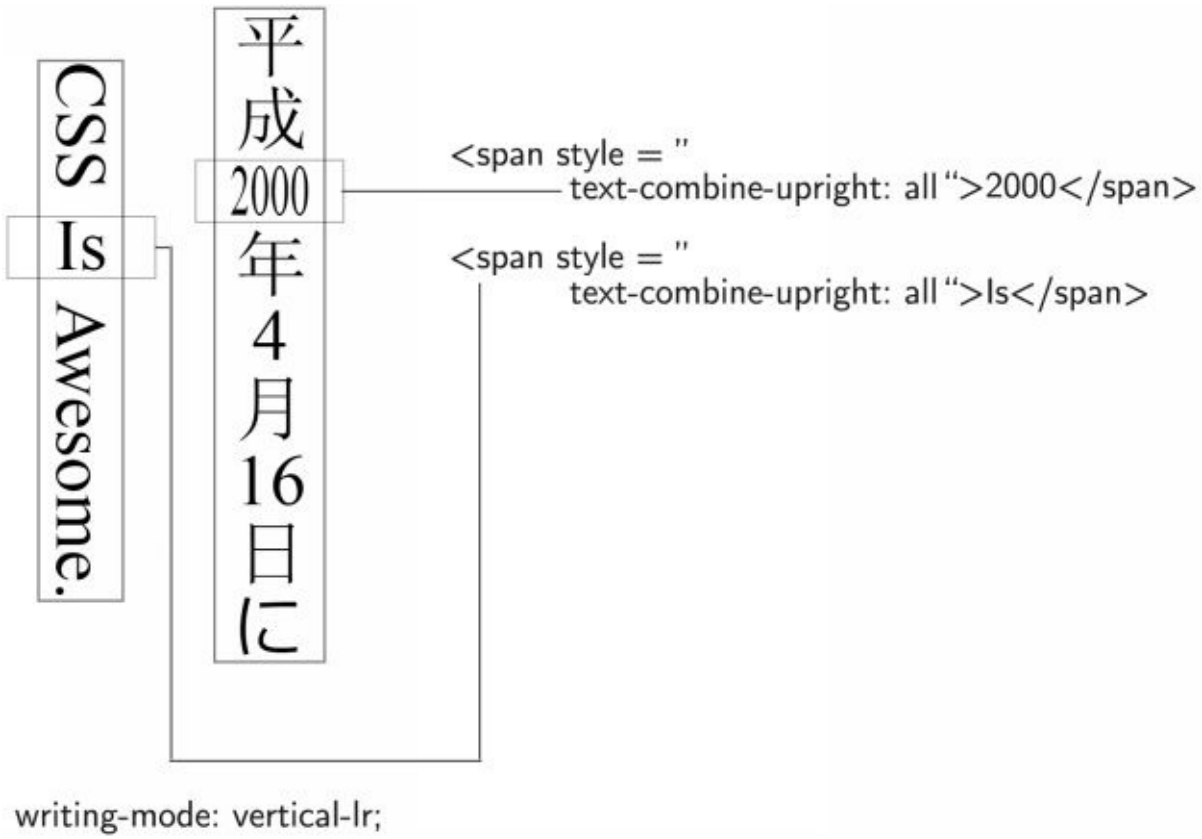



**Figure 38:** When **writing-mode** is set to **vertical**, you can also use **text-combine-upright: all** to produce the scenario shown on this diagram.

# Overflow


**Figure 39:** When text is nested within a parent element you can make it scrollable by applying **overflow:scroll** to the parent.

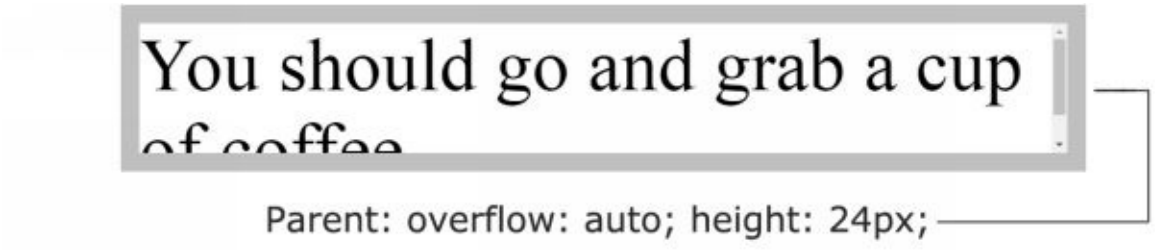



**Figure 40: overflow auto; height:24px;**

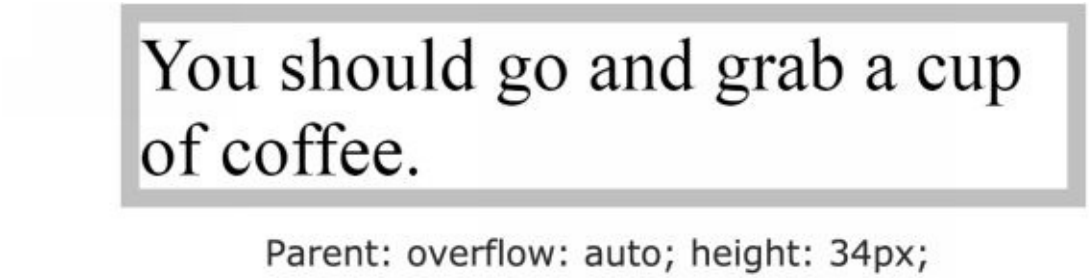



**Figure 41: overflow:auto; height:34px;**


**Figure 42: overflow:hidden;** and **position:absolute;**

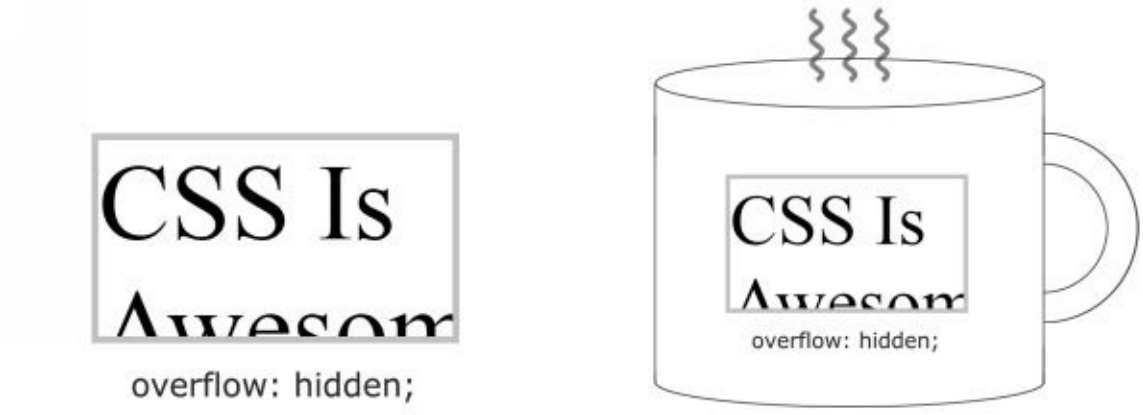



**Figure 43:** The classic case of **overflow:hidden;** You should go and grab a cup of coffee.

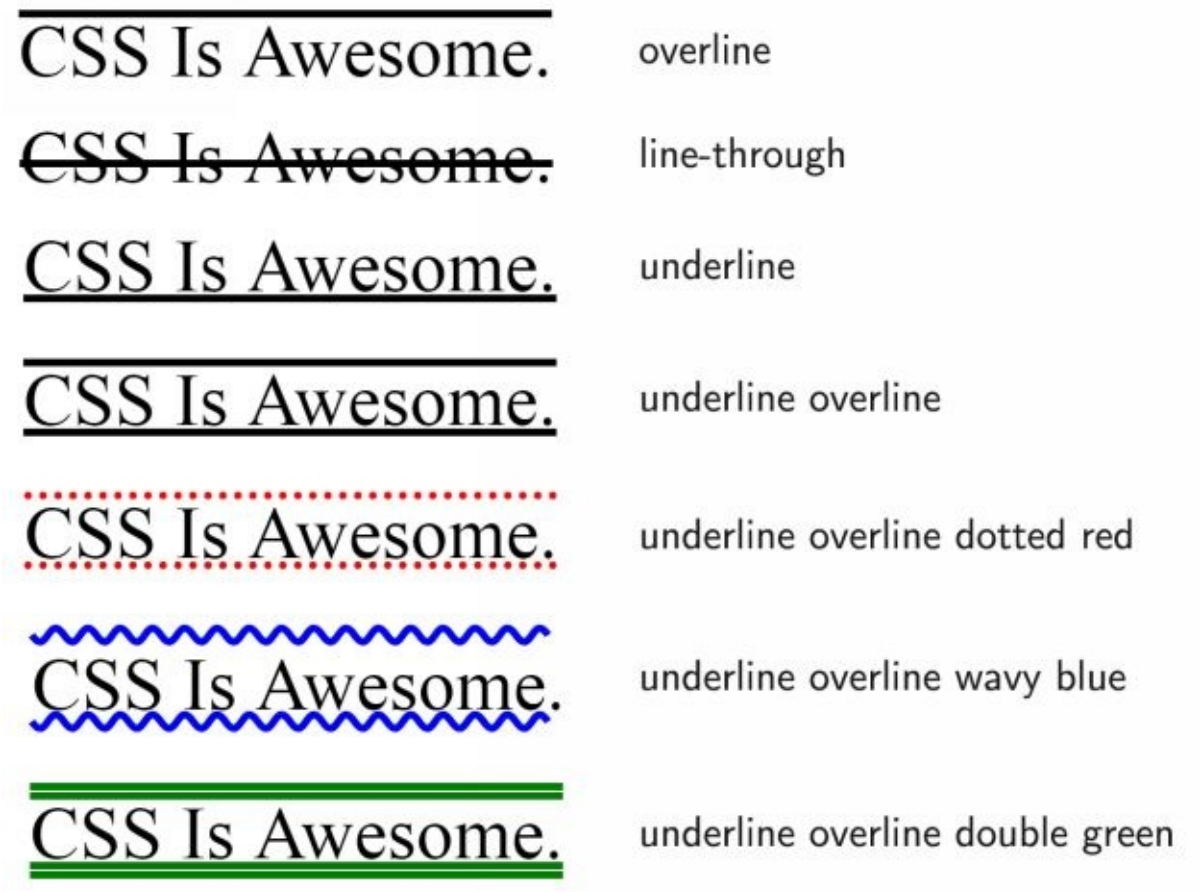
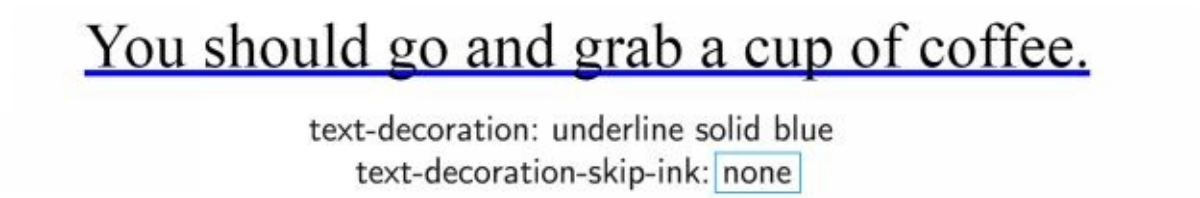


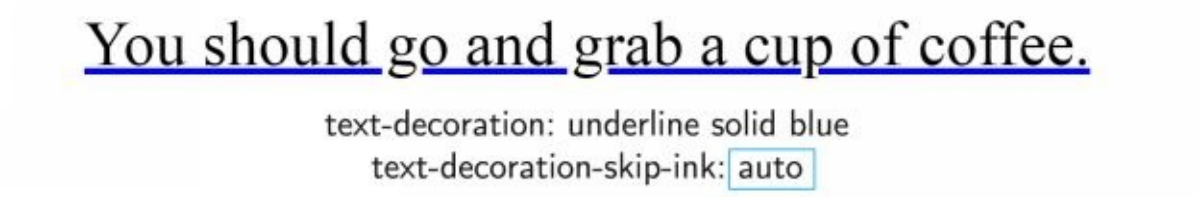

**Figure 44:** Note separate values are separated by space. You'll see a lot of this across the whole spectrum of CSS value combinations usually used as "shorthands" for individual properties. You can add underline to text using **text-decoration** property on both top and bottom of the text. Though this property is

uncommon in layout design, it's nice to know it exists and is supported by all browsers.

# Skip Ink

The **text-decoration-skip-ink** property can be used to superimpose text over the underline. This is actually useful for improving visual integrity of page titles or any underlined text that must use large letters.



# Text Rendering

The **text-rendering** property will probably not produce a noticeable difference in the four of its manifestations (**auto**, **optimizeSpeed**, **optimizeLegibility** and **geometricPrecision**). But it is believed that in some browsers using **optimizeSpeed** value is known to improve rendering speed of large blocks of text. The **optimizeLegibility** is the only value that actually produced a physical difference on the text in our experiments with Chrome browser, by shifting words closer together in some character combinations.


text-rendering: auto;


text-rendering: optimizeSpeed;


text-rendering: optimizeLegibility;


text-rendering: geometricPrecision;

The names of the four possible values used here are self-explanatory to their intended function.

# Text Indent

The text-indent property will take care of aligning your text. It is rarely used but in some cases, specifically, news sites for example, or book editing software, they might prove to be useful.

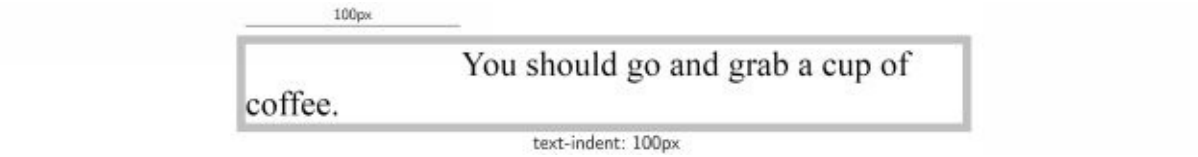



**Figure 45: text-indent:100px;**

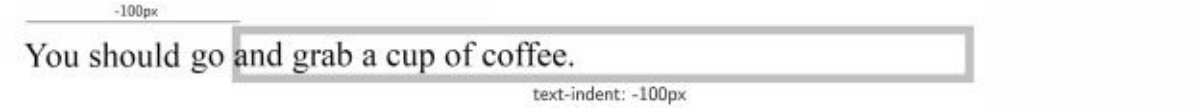



**Figure 46: text-indent:-100px;**

# Text Orientation

Text orientation is controlled by **text-orientation** property. Might be useful for rendering text in different languages where the flow of text can either go from right to left or from top down. Often used together with **writing-mode** property.


**Figure 47: text-orientation:upright;**


**Figure 48: text-orientation:upright;**

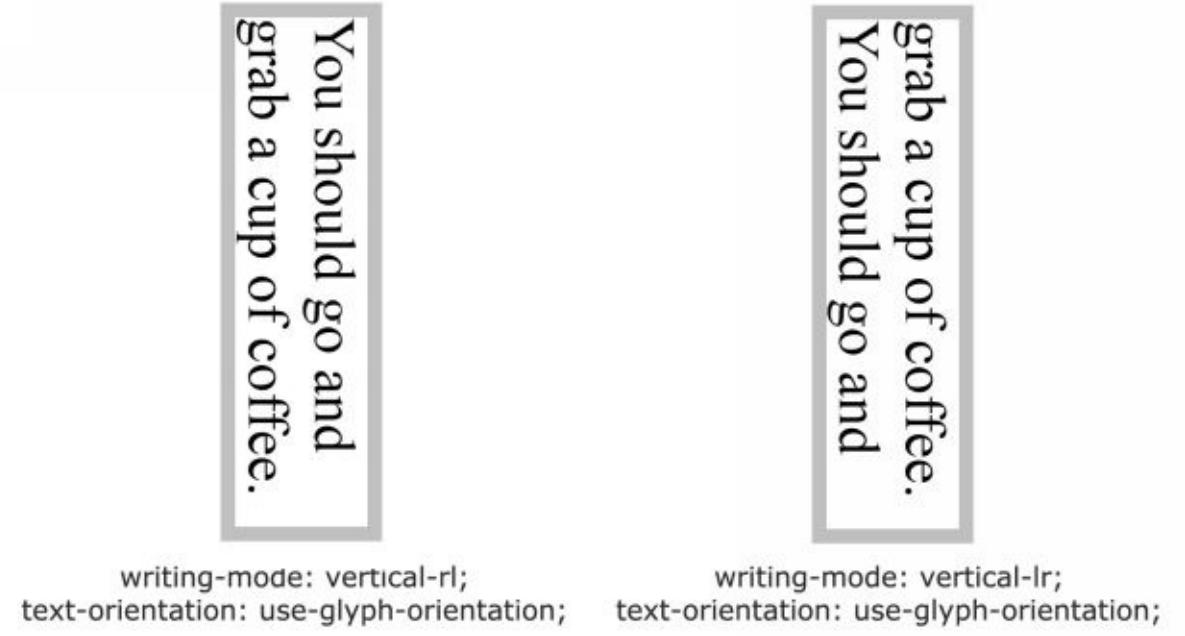



**Figure 49:** Together with **writing-mode:vertical-rl** (right to left) or **writing-mode:vertical-lr** (left to right) the **text-orientation** property can be used to produce text align in pretty much any direction.

The same as before only this time with **text-orientation** set to **upright**:

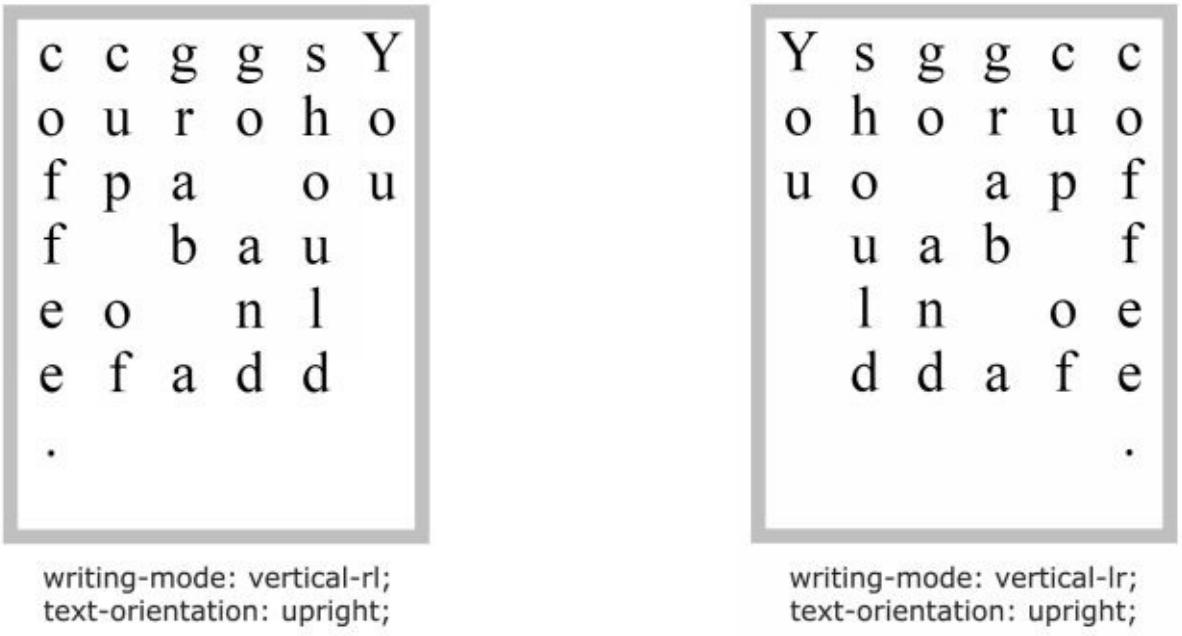



**Figure 50: text-orientation:upright; writing-mode:vertical-rl;**


**Figure 51:** To center text vertically in any element set its line height with **line-height:60px;** to the height of the element. Text size (the height of actual letters) and its **line-height** are not always the same.


**Figure 52:** Ligatures with **font-feature-settings: "liga" 1**, or alternatively **font-feature-settings: "liga" on**

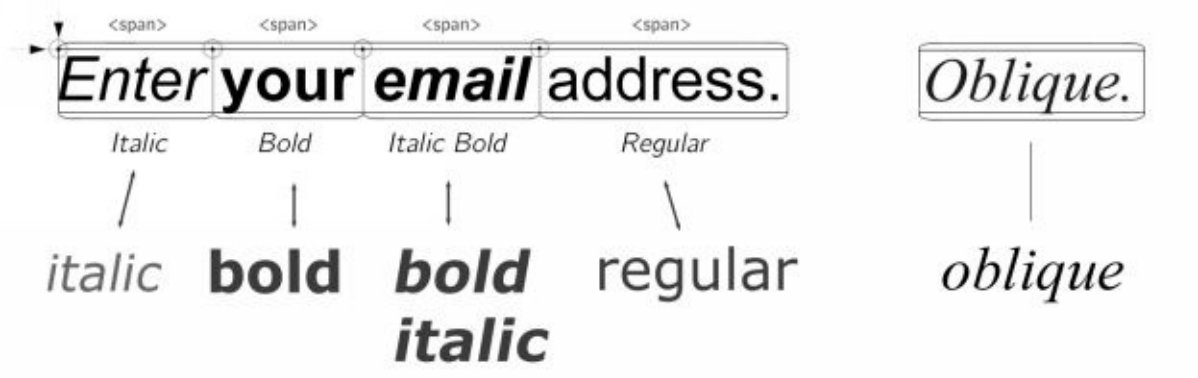



**Figure 53:** Common text effects (*italic*, *bold*, and *oblique*) are achieved by using the properties **font-style** and **font-weight**.

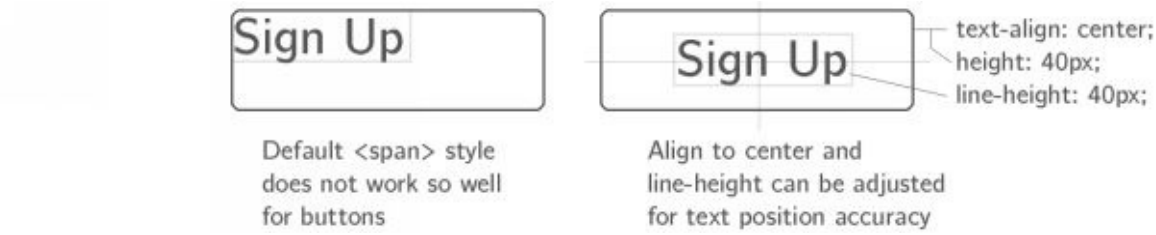



**Figure 54:** The **text-align** and **line-height** properties are often used to center text inside buttons.

# Text Shadow


text-shadow: 0px 0px 0px #0000FF


text-shadow: 0px 0px 1px #0000FF


text-shadow: 0px 0px 2px #0000FF


text-shadow: 0px 0px 3px #0000FF


text-shadow: 0px 0px 4px #0000FF


text-shadow: 2px 2px 4px #0000FF


text-shadow: 3px 3px 4px #0000FF


text-shadow: 5px 5px 4px #0000FF

**Figure 55:** You can add a shadow to your text using **text-shadow** property. See the next diagram to understand its parameters.

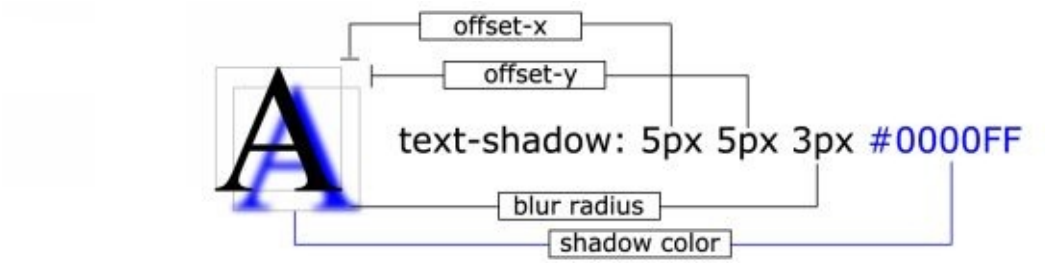



**Figure 56:** The **text-shadow** property takes the offset on both *x* and *y* axis, *blur radius* and *shadow color*.

We won't be going much into SVG, which can also be controlled by CSS properties. An entire book can be written on the subject alone. But as a brief inset here, you can create rotated SVG text as follows:

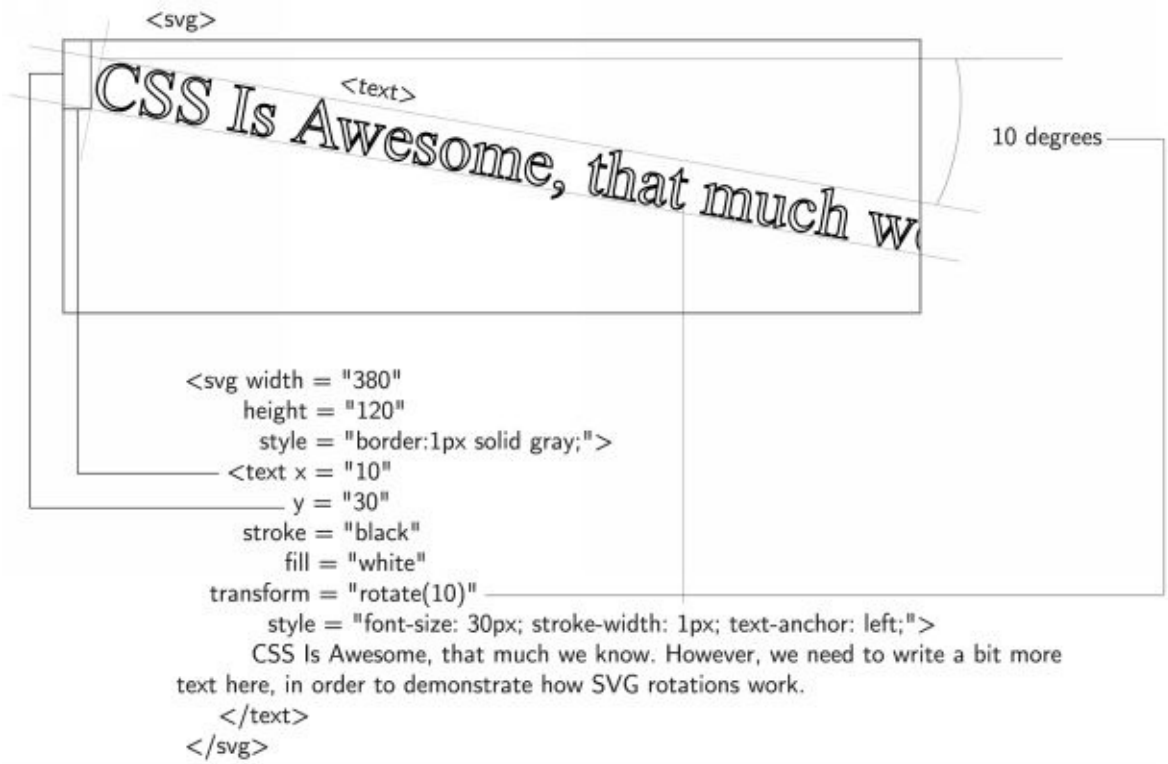



**Figure 57:** Using CSS to manipulate SVG text rotation.

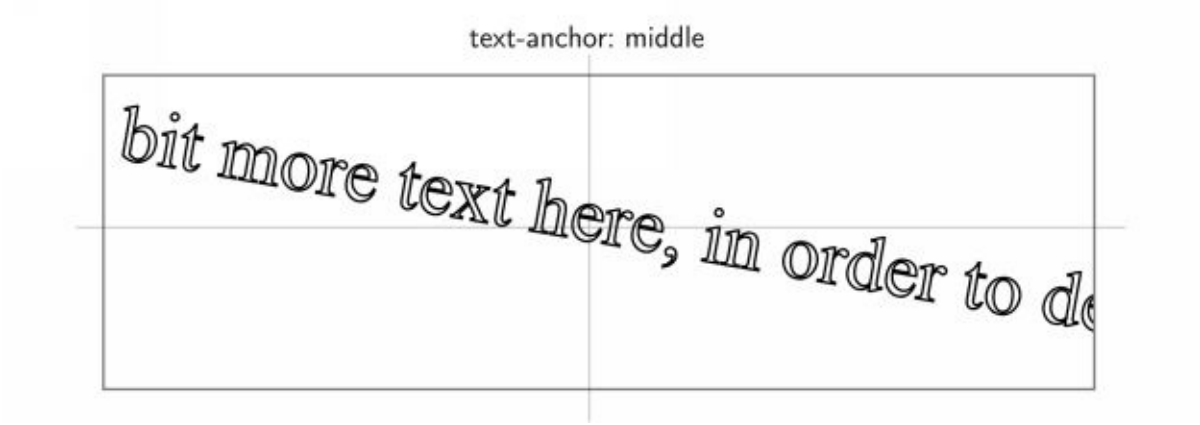



**Figure 58:** Using **text-anchor** it's possible to set the center point of the text, around which it will be rotated.

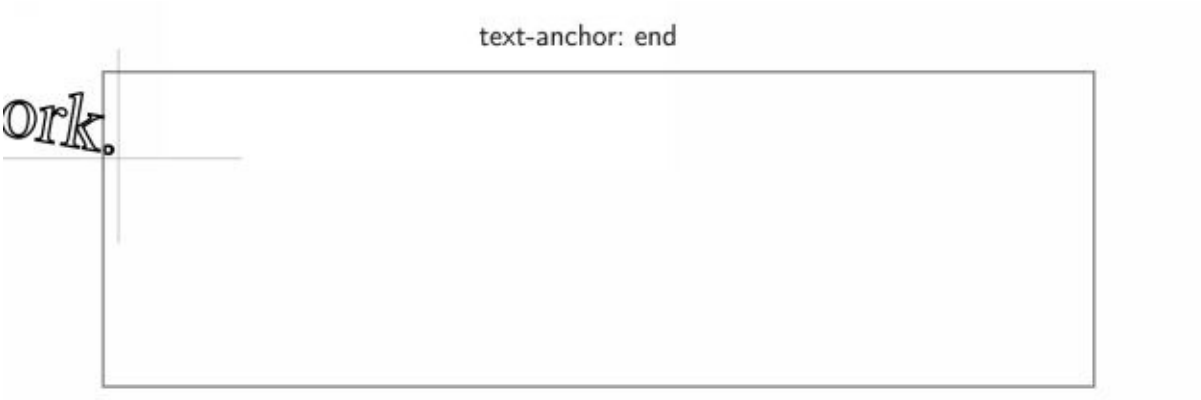



**Figure 59:** Setting **text-anchor** to **end** to offset center of rotation to the very end of the text block. We'll see similar behavior on CSS transform property that can be used to rotate entire HTML elements and text within them.

# Margin, Rounded Corners, Box Shadow and Z-Index

These few subjects, in no particular order, were chosen to briefly demonstrate commonly used CSS properties.

# Border Radius

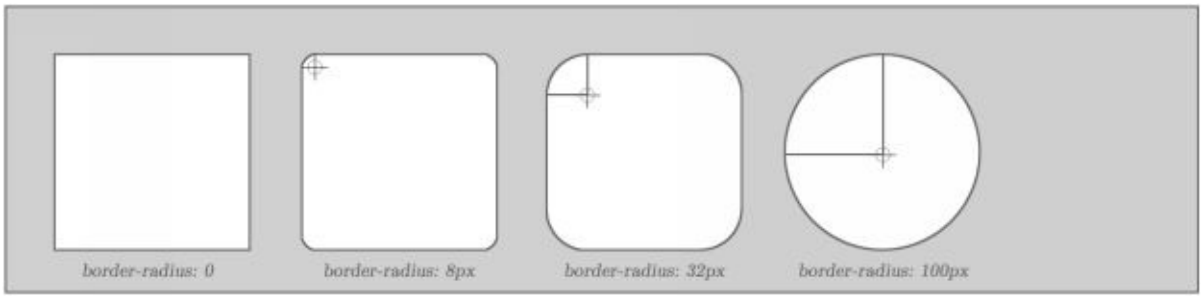



**Figure 60:** The **border-radius** is the property used to add rounded corners to square or rectangular HTML elements.

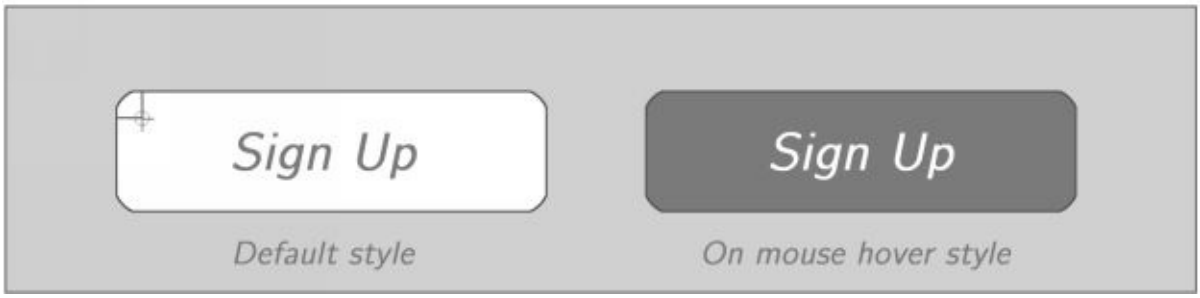



**Figure 61:** Using the **:hover** *pseudo-selector* you can choose what happens when the mouse hovers (enters the area of) over an element.

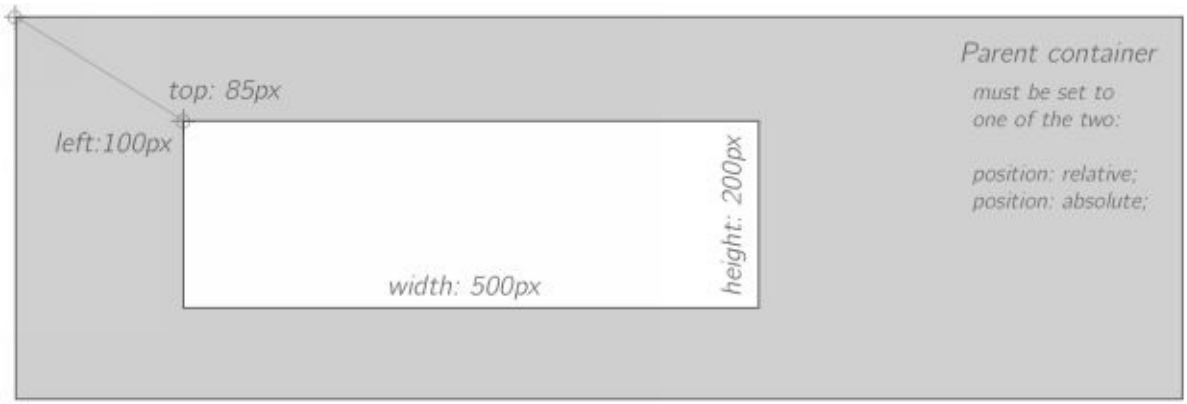



**Figure 62:** The parent container ***must*** be explicitly set to either **position:relative** or **position:absolute** in order to use a child element within it that also uses **position:absolute** align.

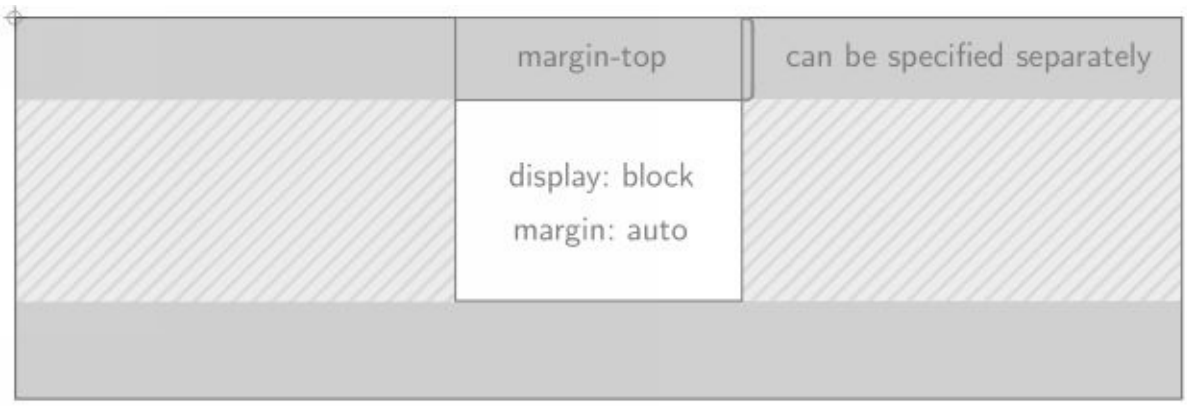



**Figure 63:** You can use **margin:auto** to align an element horizontally. Just make sure its **display** property is set to **block;** The property **margin-top** can be used to offset an element by a space on its upper side. You can also use **margin-left**, **margin-right**, **margin-bottom**.

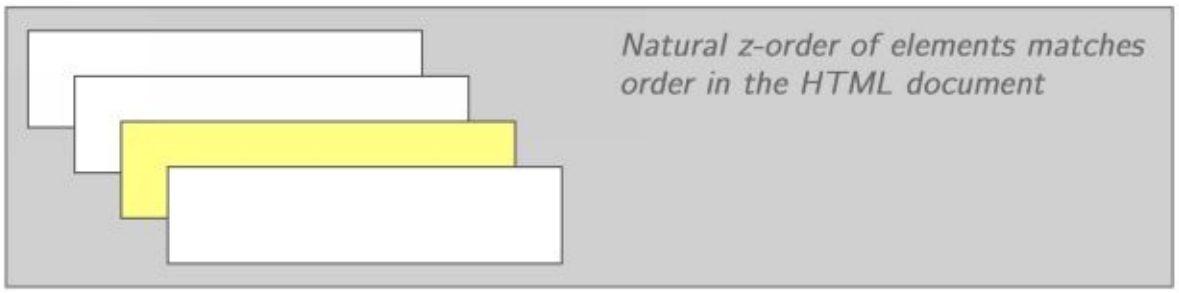



**Figure 64:** The **z-index** property takes a *numeric value* between ***0*** -- ***2147483647*** to determine element's drawing order on most common browsers. In Safari 3 the maximum **z-index** value is ***16777271***.

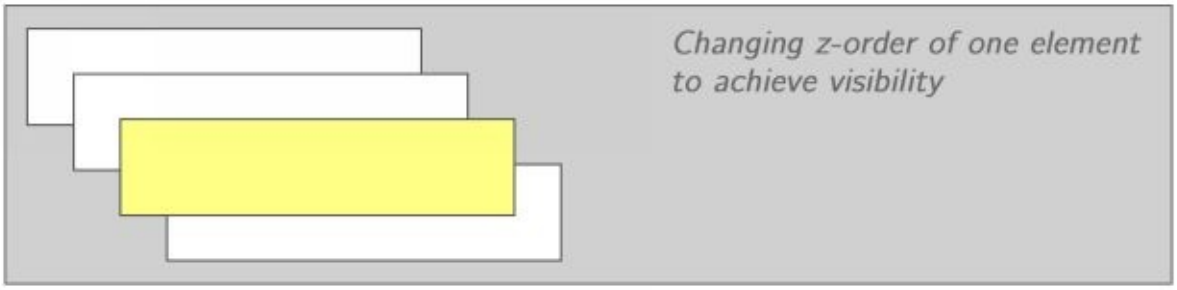



**Figure 65:** Changing **z-order** of one element to change visibility order and make it stand out.

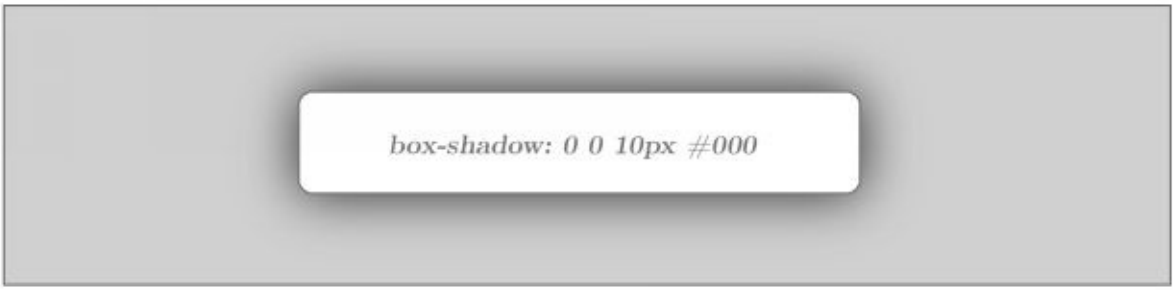



**Figure 66:** Here **box-shadow** is used to add a shadow around a wide element. It takes the same parameters **text-shadow** does, for example: **box-shadow: 5px 5px 10px #000** (*x* and *y* offset, *radius* of the shadow, and shadow *color*.)


**Figure 67:** The property **box-radius** controls the radius of the corner's curve on both **X** and **Y** axis.

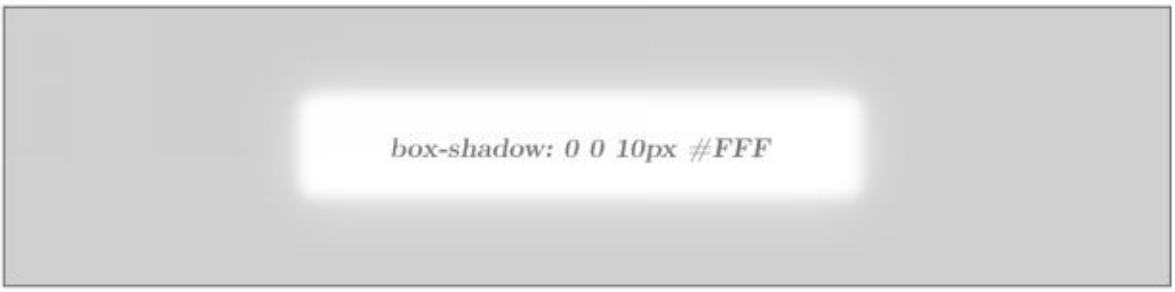



**Figure 68:** Using bright colors with **box-shadow** property it is possible to create glowing effect around HTML elements.

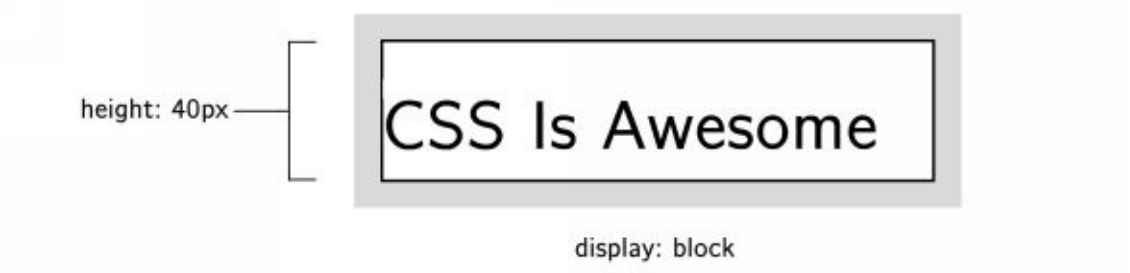


**Figure 69:** Just what you would expect from a simple blocking element.

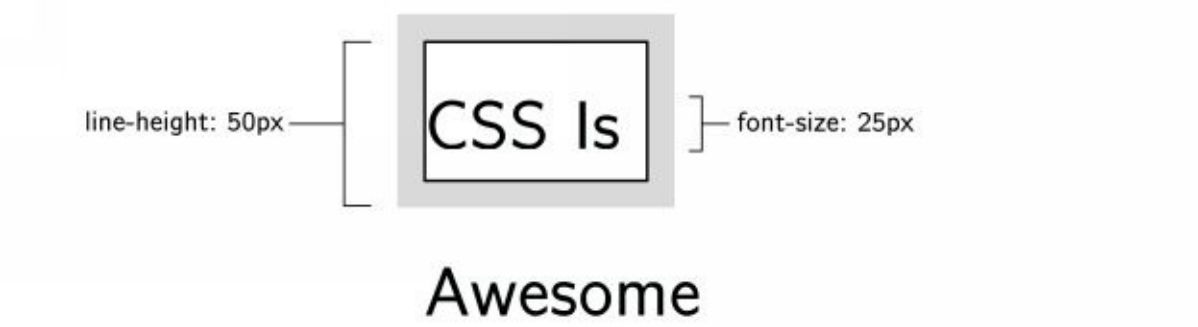



**Figure 70:** When the width of an element becomes smaller than the width of its text content, text automatically moves to the next available line, even if it exists outside of the element's boundary.

Let's take an even closer look at the previous scenarios.

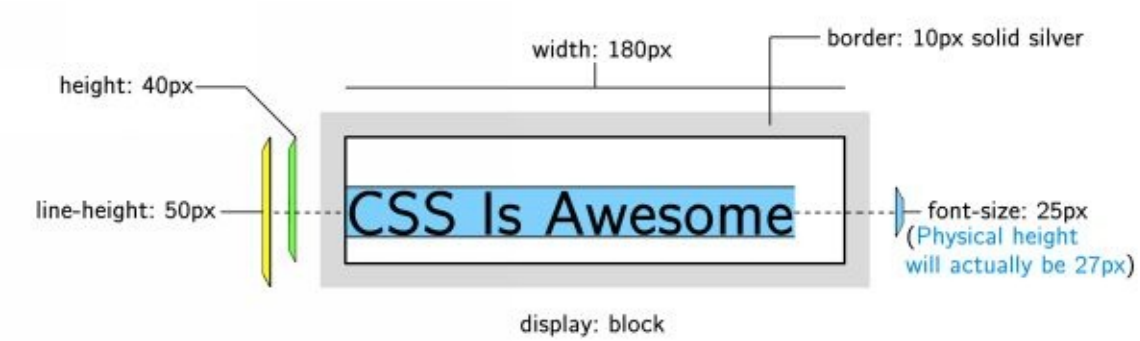



**Figure 71:** The physical height of the text will actually be **27px**, 2 pixels more than **25px** -- the original value set. The value provided by **line-height** can stretch outside of content area.

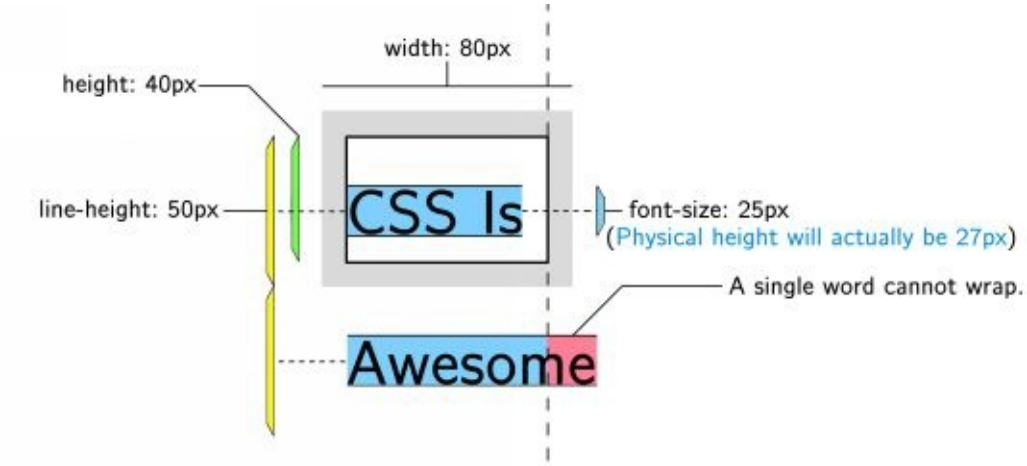



**Figure 72:** Here we can clearly see that the word "Awesome" jumped over to the next line. In addition to this, note that a single word cannot wrap around the container element even if its width is smaller. In other words, **overflow** property is **visible** by *default.*

You can effectively cut off the content outside of the content container by setting **overflow:hidden** . This will work even on elements with rounded corners:


**Figure 73: overflow:hidden** works on rounded corners.

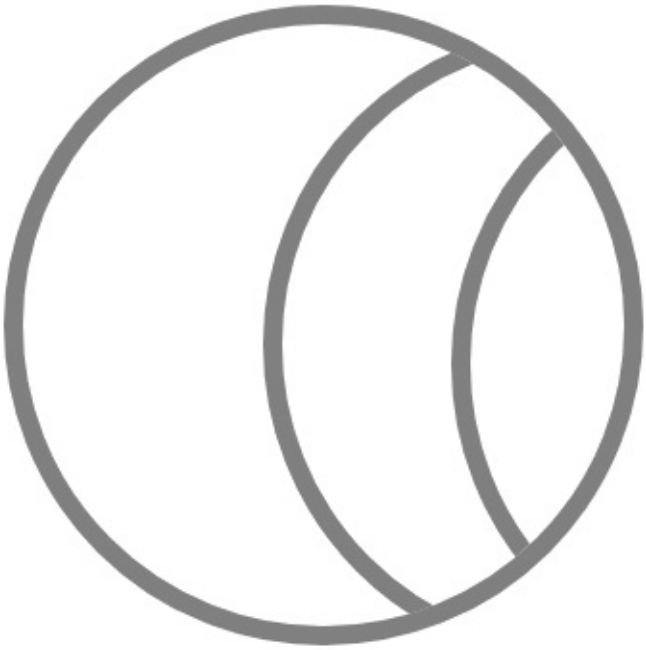


**Figure 74:** Hiding other round elements within a circle can create some interesting, irregular shapes.

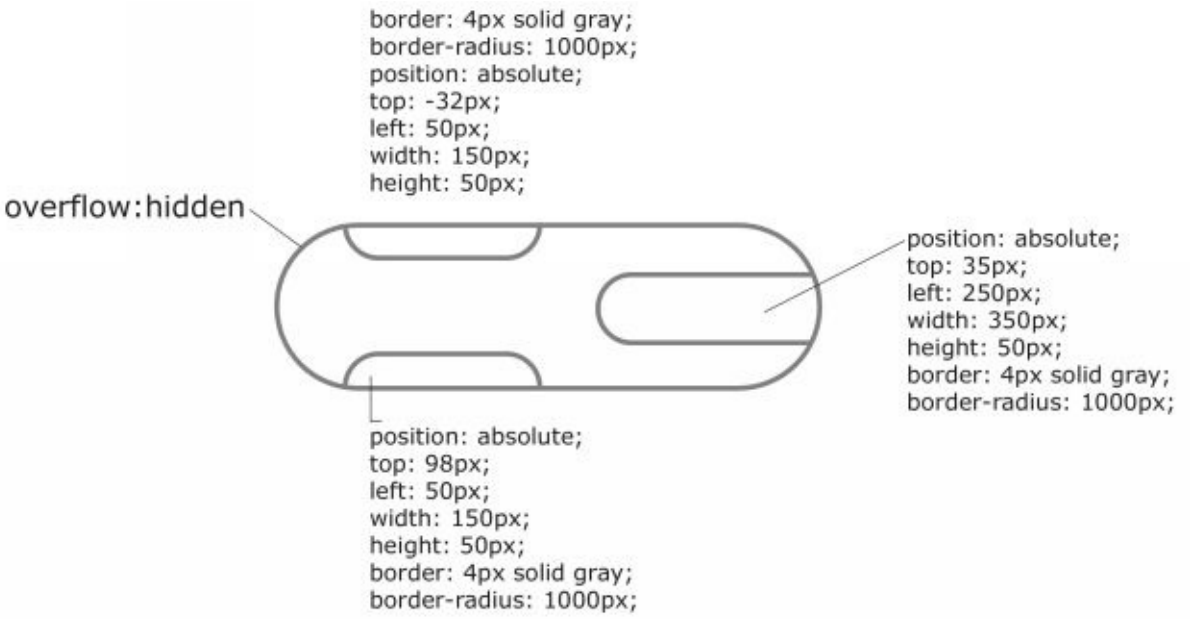



**Figure 75:** By using multiple elements with **overflow:hidden** it is possible to create irregular shapes.

**Figure 76:** The same as previous example, except with parent container **background** set to **gray**, and the **background** of elements within it set to **white**.

You can get really creative with this and make some interesting objects. We'll take a look at an entire car example toward the end of the book.

# Nike Logo

By combining techniques from previous section with **transform:rotate** (*it will be discussed in greater detail later in the book*) and our current knowledge of **:before** and **:after** pseudo selectors, it is possible to create the NIKE logo from a single HTML element:

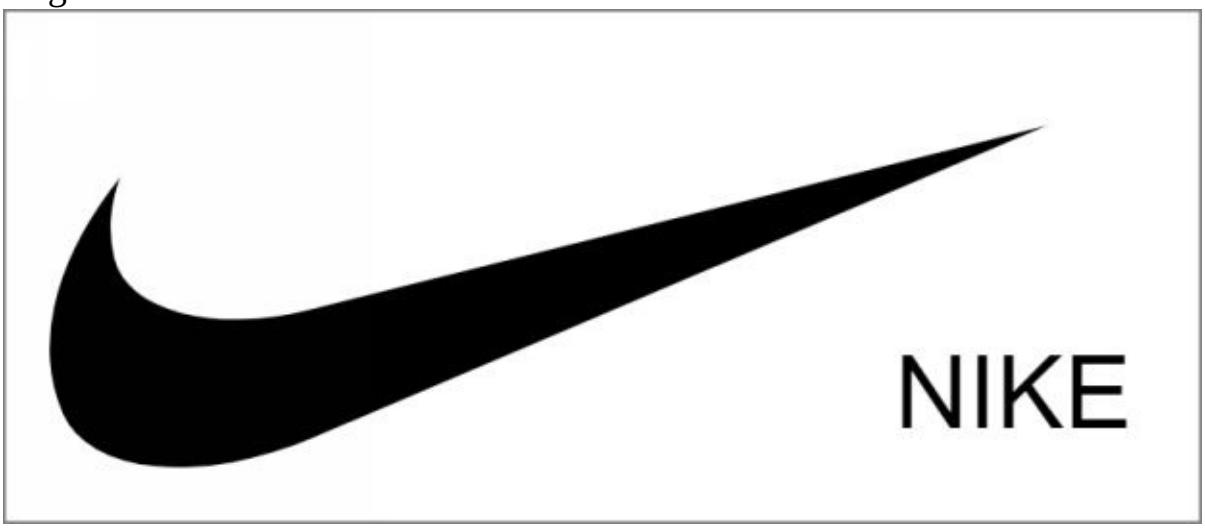



**Figure 77:** Nike logo created 1 HTML element and 3 CSS commands.

Let's define our main container: Source Code 41

```
#nike {
position: absolute;
top: 300px; left: 300px;
width: 470px; height: 200px;
border: 1px solid gray;
overflow: hidden;
font-family: Arial, sans-serif;
font-size: 40px;
line-height: 300px;
text-indent: 350px;
z-index: 3;
}
```

Note **overflow:hidden** here is used to ensure everything outside of the container is clipped away.

Using **#nike:before** and **#nike:after** pseudo elements we'll create the base of the logo which is a long black bar. Rounded corners used here to create the famous Nike curve: Source Code 42

```
#nike:before {
content: "";
position: absolute;
top: -250px;
left:190px;
width: 150px;
height: 550px;
background: black;
border-top-left-radius: 60px 110px;
border-top-right-radius: 130px 220px;
transform: rotate(-113deg);
z-index: 1;
}
```

Similarly, we'll create another curved box. Its white background will serve as a mask to block out the rest of the logo. Here, the rotation angle is everything. It's what forms the recognizable curve of the logo. We've also used **z-index** of **1**, **2** and **3** respectively to ensure proper layering of the elements.

Source Code 43

```
#nike:after {
content: "";
position: absolute;
top: -235px;
left: 220px;
width: 120px;
height: 500px;
background: black;
border-top-left-radius: 60px 110px;
border-top-right-radius: 130px 220px;
background: white;
transform: rotate(-104deg);
z-index: 2;
}
```

Here is another view of the logo. This time with transparent background, so we can actually its geometric composition:

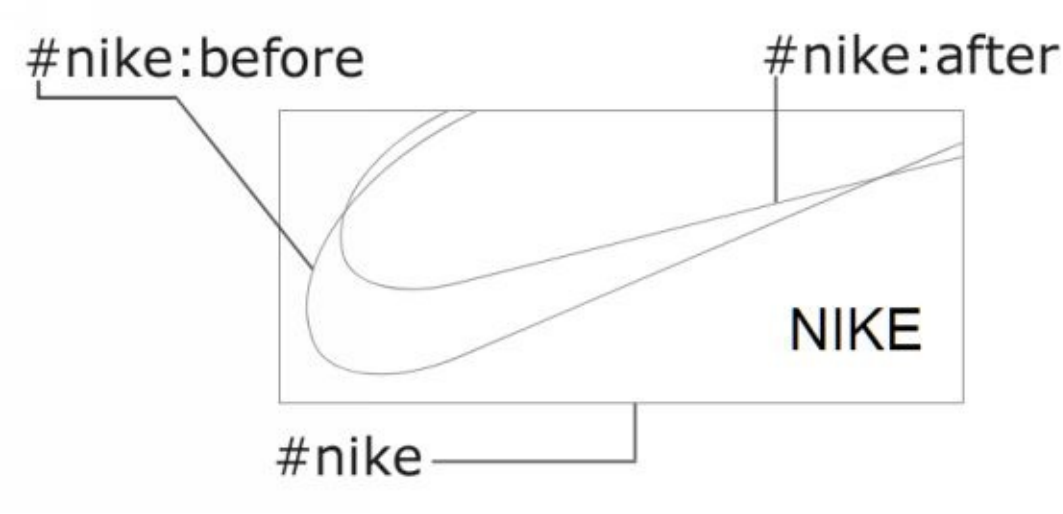



**Figure 78:** Composition of the Nike logo, consisting of 3 elements (1 HTML element and its 2 pseudo-element counterparts.) The actual HTML is just one **div** element with ***id*** **nike**.

Source Code 44

```
<div id = "nike">NIKE</div>
```

# Display

CSS properties are used to assign behavior to HTML elements that determine their placement on the screen. Diagrams in this section demonstrate the effect on each element in a set of common cases.

The CSS property **display** can take any one of the several values: **inline**, **block**, **inline-block** or **float** to define placement of individual elements.

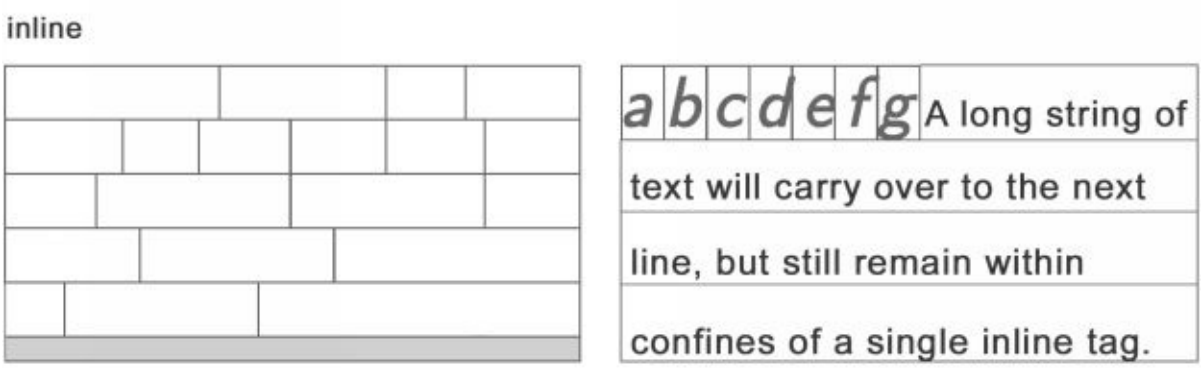



**Figure 79: display:inline**. This is the **default** value used for ****, **<b>**, **<i>** and a few other HTML tags created for dealing with displaying text inside parent containers with unknown width.

Here, each element is placed directly on the right hand side after the length of its content (or its width) in the previous element has been exceeded, making it the natural option for displaying text.

**Note:** *long* inline elements are automatically carried over to the next row.

Later, when we arrive at the Flex and CSS grid chapters, you will see how applying the values **flex** or **grid** to the **display** property can modify the behavior of its *items* -- elements residing inside a container element, often referred to as their *parent* element.

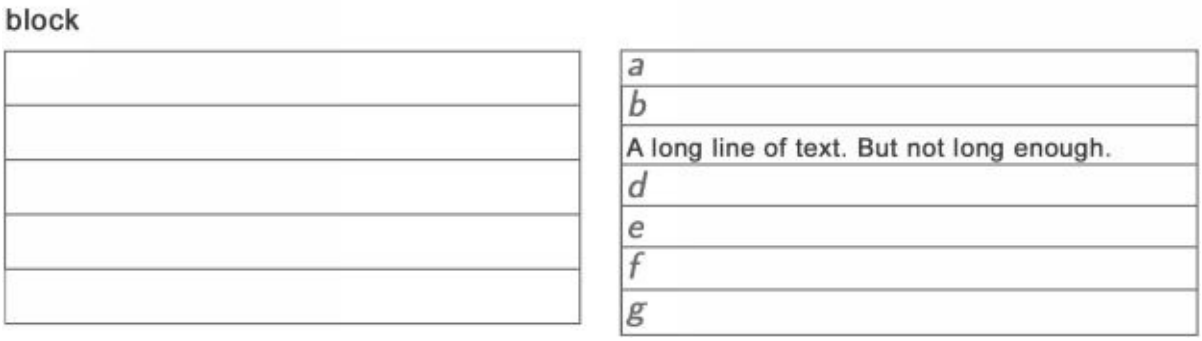


**Figure 80:** **display:block** -- in contrast to **inline** elements -- will automatically block an entire row of space, regardless of the width of its content. The HTML tags ** ** is a blocking element by default.

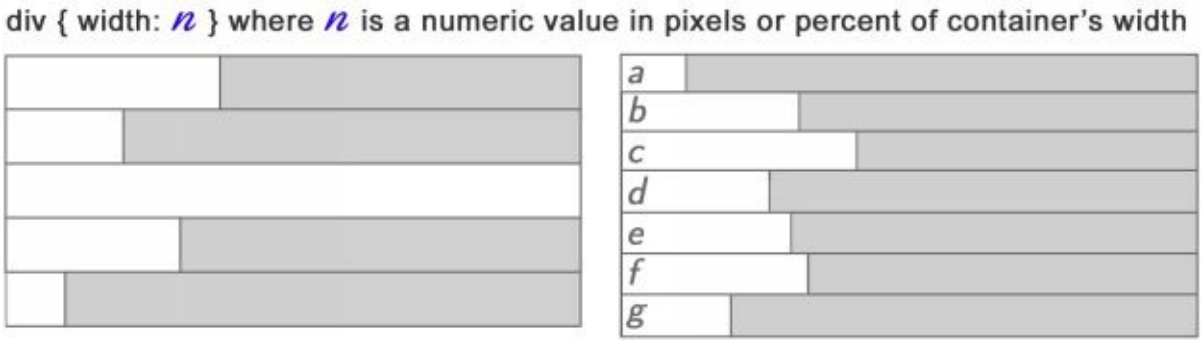



**Figure 81:** **display:block**, with explicitly defined element widths introduces the idea of discerning between element's container width and its content width.

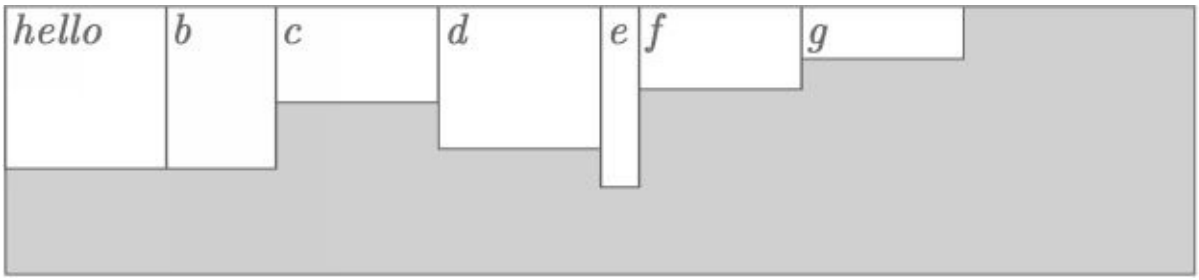



**Figure 82:** **display:inline-block** *combines* **inline** and **blocking** behavior to enable custom size for inline elements


**Figure 83:** Centered text (**text-align:center**) inside two blocking elements with width set to 50
% of the container. Note, that while blocking the entire row of the parent element, the content area only stretches up to 50
% of its width. A blocking element is not defined by the width of its content.

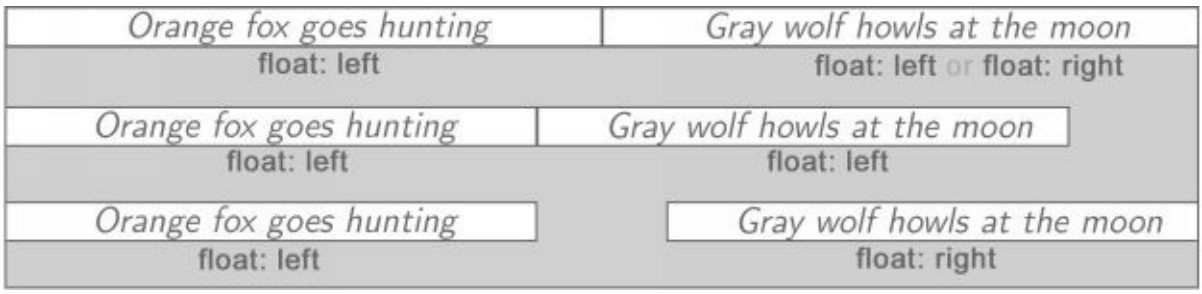



**Figure 84:** Two blocking elements with explicit width of about 50 % and **text-align:center** can somewhat imitate inline elements by also applying **float:left** and/or **float:right**. However, unlike inline elements, a single blocking element can never cross over to the next row.

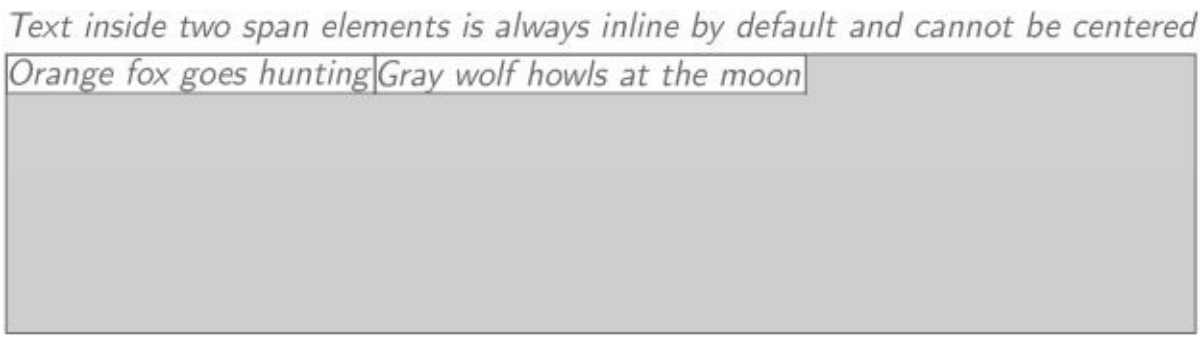



**Figure 85:** Inline elements are always limited to the width of their content and therefore text within them cannot be centered.

# Element Visibility

Element's visibility can hide the element's box without completely removing it from the drawing hierarchy.

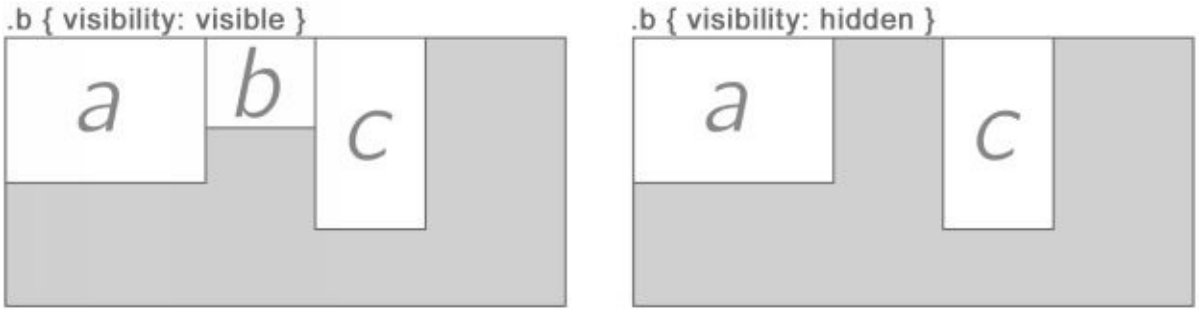



**Figure 86:** Setting **visibility** property to **hidden** on "b" element. The default is **visible** (Same as **unset**, or **auto**, or **none**).

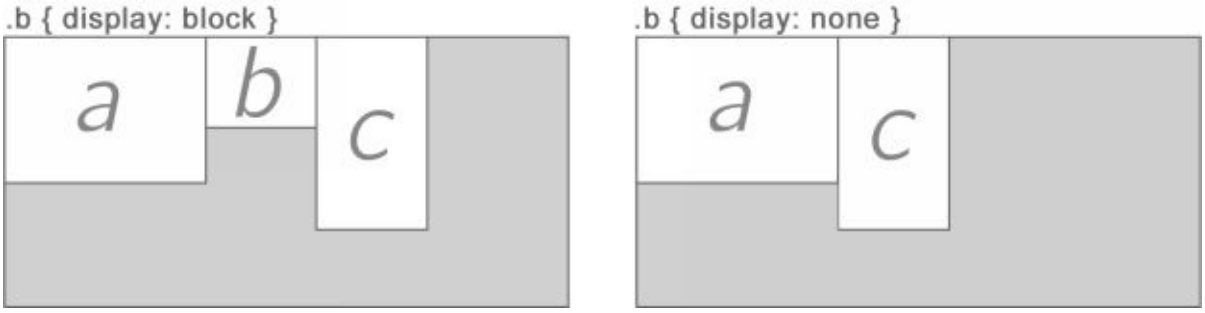



**Figure 87:** **display:none** completely removes the element.

# Floating Elements


**Figure 88:** Blocking elements with **float:left** and **float:right** appear on the same row, as long as the sum of their widths is less than the width of the parent element.

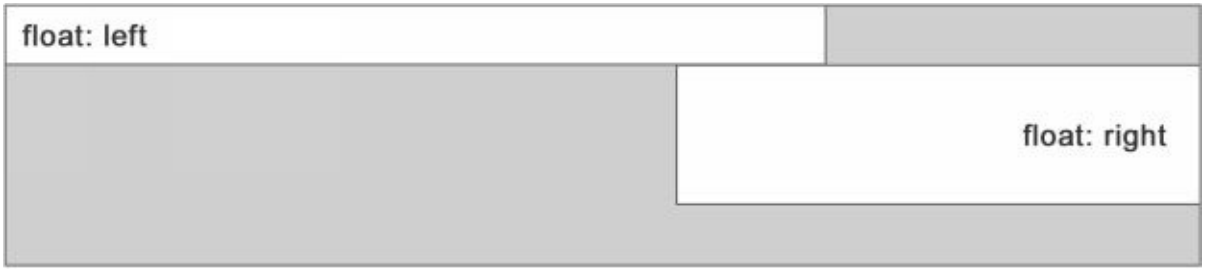



**Figure 89:** If the total sum of two floating element's is greater than parent's width, one of them will be blocked by the other and skip over to the next row.

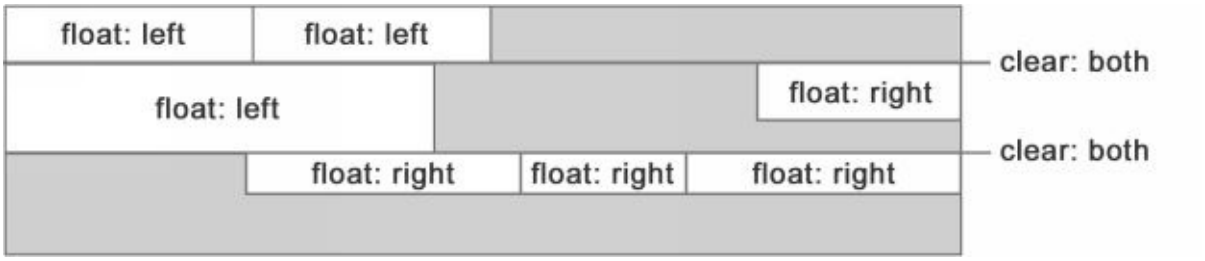



**Figure 90:** You can use **clear:both** to clear floating elements and start a new floating row.

# Color Gradients

Gradients can be used for a variety of reasons. But the most common thing they're usually used for is to provide a smooth shading effect across the area of some User Interface element.

Here are a couple more reasons for using them: **Smooth Background Color Shading** provides an elegant solution for making your HTML elements more appealing to the eye.

**Saving Bandwidth** is another benefit of using gradients, because they are automatically generated in browser by an efficient color shading algorithm. This means that they can be used instead of images, which would otherwise take a lot longer to download from the web server.

**Simple Definitions** can be used in **background** property to create some quite interesting and sometimes surprising effects. You will be supplying the minimum required parameters to either **linear-gradient** or **radial-gradient** properties to create any of the effects demonstrated in the next section.

# Overview

In this chapter we will learn how to create these gradients in HTML:


**Figure 91:** If this is a black and white print, you will not see the difference between gradients that actually use color. However, to master gradients you really only need a good grasp on their direction and type of which there are four -- **linear-gradient**, **radial-gradient**, **repeating-linear-gradient** and **repeating-radial-gradient**. This diagram gives you a good idea of the variety of gradients it is possible to create for your HTML elements with CSS.

I cheated a little here... the images above are files from my gradients folder that I created while working on this book. But how do we actually create them using CSS commands? The rest of this chapter will provide a solution!

***Specimen Element for Displaying Gradients*** We will perform our experiments with the background gradients using this simple DIV element. Let's set some basic properties to it first, including width=500px and height=500px.

For now, we just need a simple square element. Paste this code anywhere in between **<head>** tags in your HTML document.

Source Code 45

```
<style type = "text/css">
div {
position: relative;
display: block;
width: 500px;
height: 500px;
}
```

<style> This CSS code will turn every ** ** element on the screen to a square with dimensions of 500 by 500 pixels. The **position** and **display** properties will be explained further in the book.
Alternatively, we probably want to assign gradients only to one HTML element. In which case you can either specify the CSS to an individual **div** element using a unique ID such as **#my-gradient-box** *or any other that makes sense to you.*
Source Code 46
<style type = "text/css">
div#my-gradient-box { position: relative; display: block; width: 500px; height: 500px; }
<style> And then add it somewhere within your **<body>** tag as: Source Code 47
<!-- Experimenting with Color Gradient Backgrounds in HTML //-->
 Or type the same CSS commands directly into **style** attribute of an HTML element you wish to apply a color gradient to:
Source Code 48

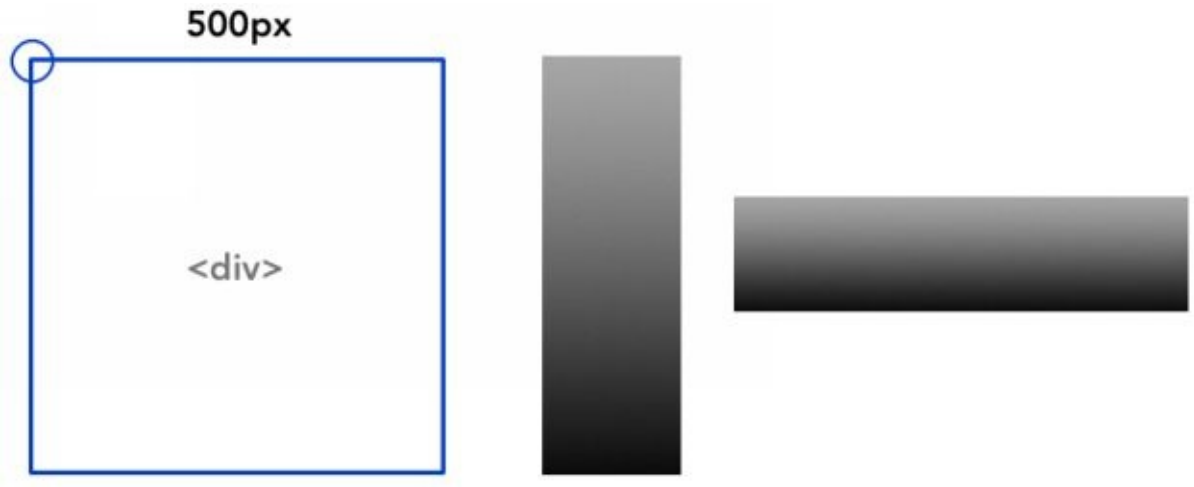



**Figure 92:** A div element with dimension of 500 x 500 pixels. The row and column on the right hand side demonstrate how gradients automatically adapt to the element's size. The gradient property was not changed here. Only the element's dimensions, yet the gradient looks quite different. Keep this in mind when making your own gradients!

CSS gradients will automatically adapt to the element's width and height. *Which might produce a slightly different effect.*

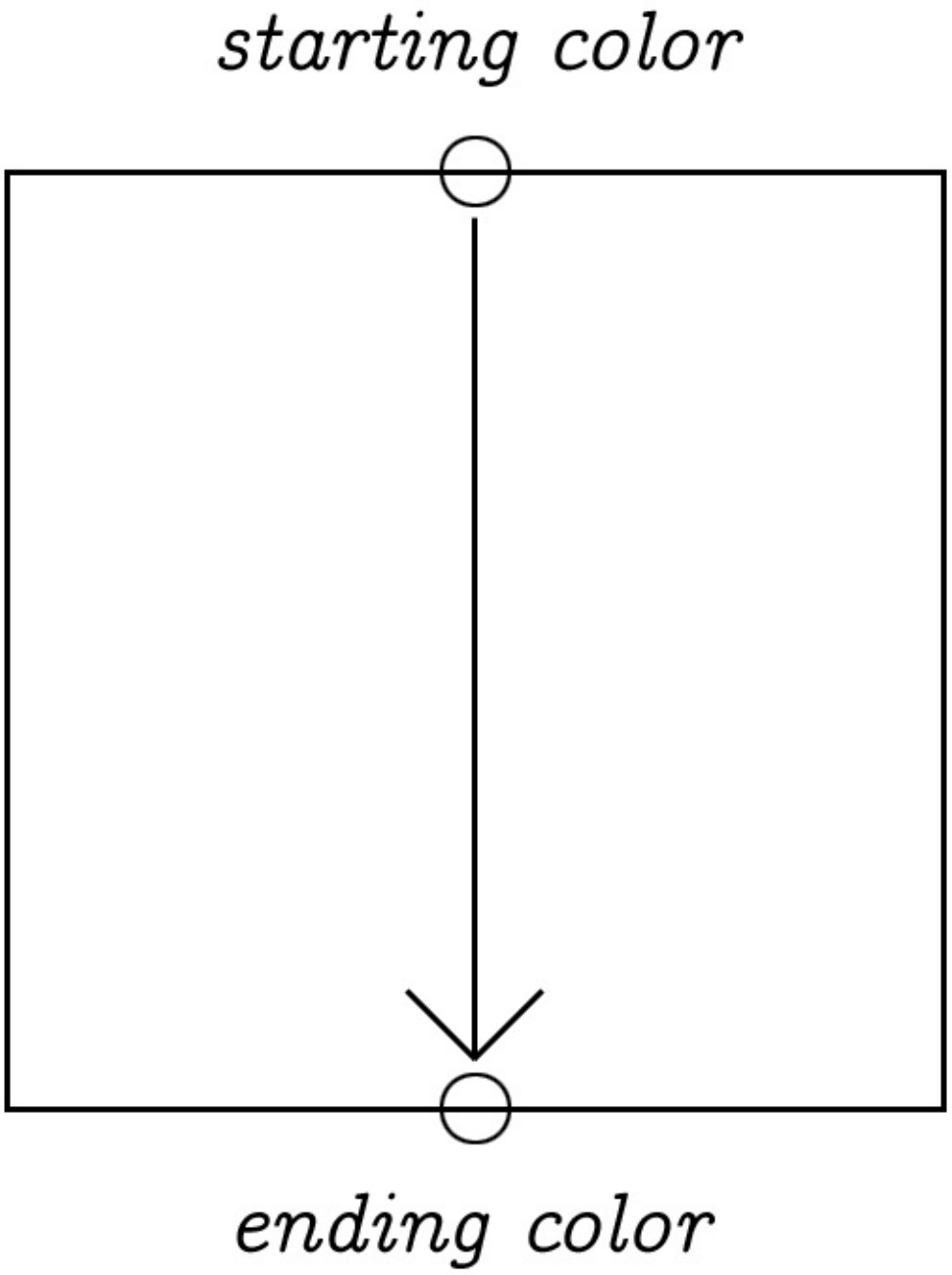



**Figure 93:** The basic idea behind gradients is to interpolate between at least two colors. By default, without providing any extra values, vertical direction is assumed. The starting color will begin at the top of the element, gradually blending in with 100
% of the second color at the bottom. It's possible to create gradients by combining more than two colors. We'll take a look at that in a moment!

All CSS gradient values are supplied to CSS **background** property!

Having said that, here's an example of creating a simple linear gradient: Source Code 49

background: linear-gradient(black, white); These values will be demonstrated in action below, shown just underneath the gradient effect they produce.

# Gradient Types

Let's walk through different gradient styles one by one and visualize the type of gradient effects you would expect to be rendered within the HTML element, when these styles are applied to it.

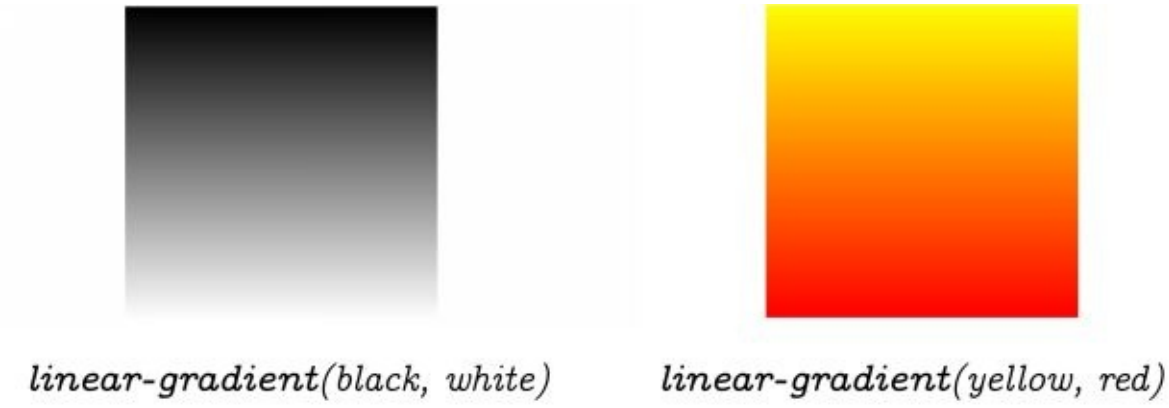



**Figure 94:** A simple linear gradient. Left: black to white. Right: yellow to red.

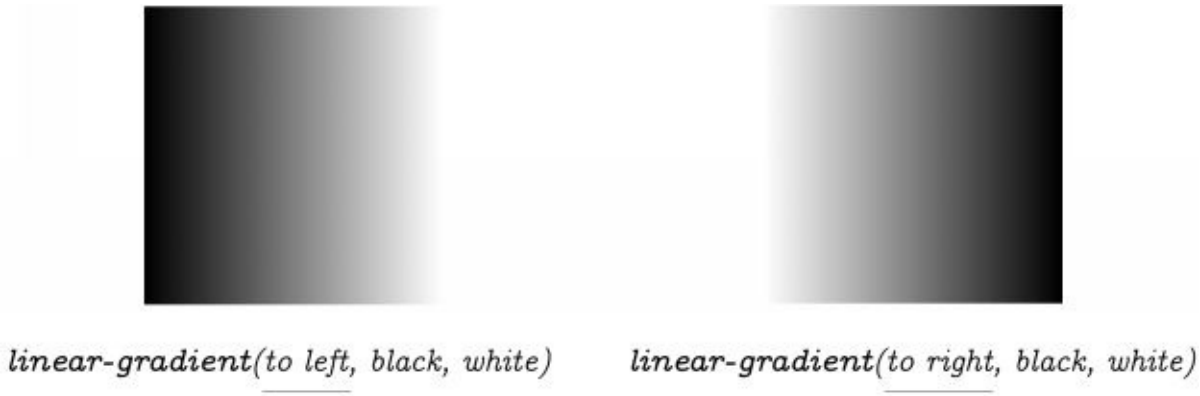



**Figure 95:** Horizontal gradients can be created by specifying a leading value of either "to left" or "to right", depending on which direction you wish your gradient to flow across the element.

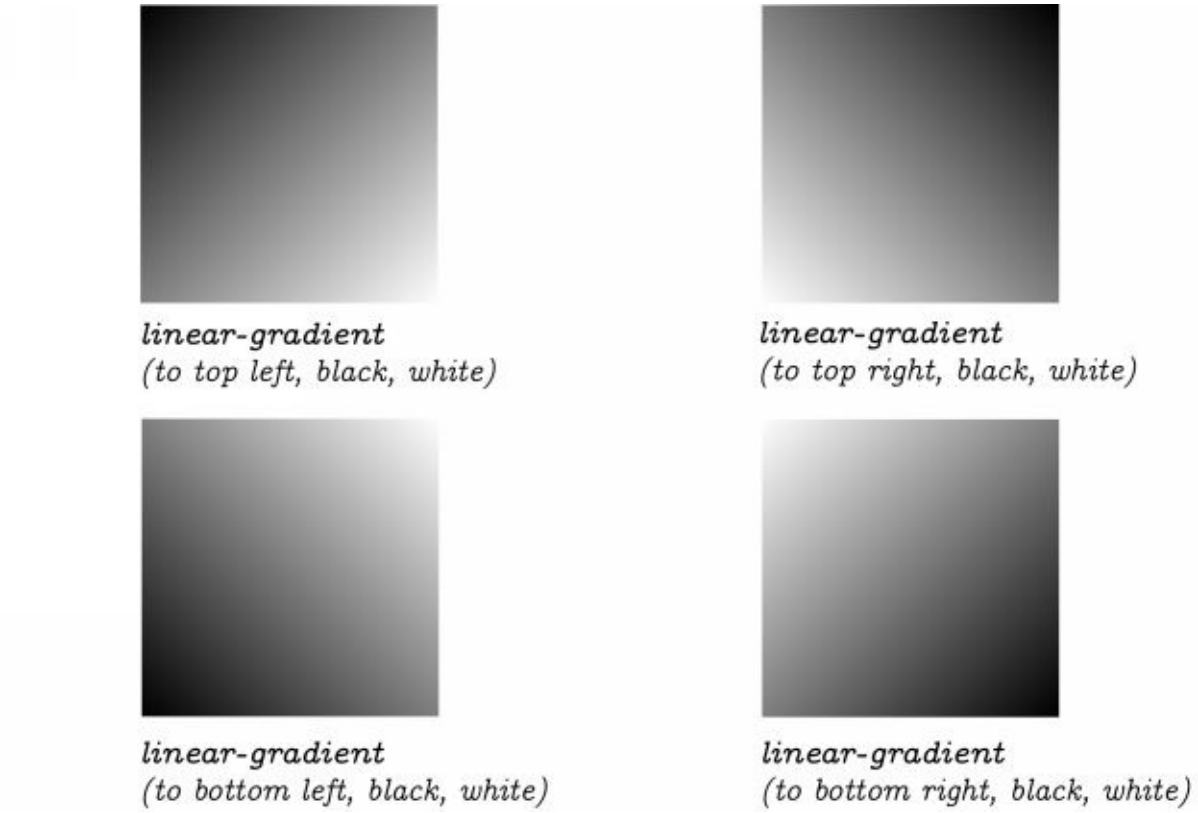



**Figure 96:** You can start gradients at corners too to create diagonal color transitions. Values "to top left", "to top right", "to bottom left" and "to bottom right" can be used to achieve that effect.

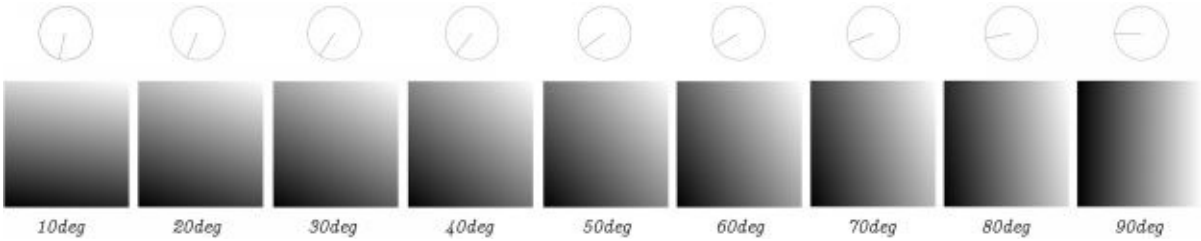



**Figure 97:** When 45 degree corners are not enough, you can supply a custom degree between 0 -- 360 directly to the **linear-gradient** property as in **linear-gradient(30deg, black, white);** Notice how in this example the gradient gradually changes direction from flowing toward the bottom, toward the left hand side when angle is changed in progression from 10 to 90 degrees.

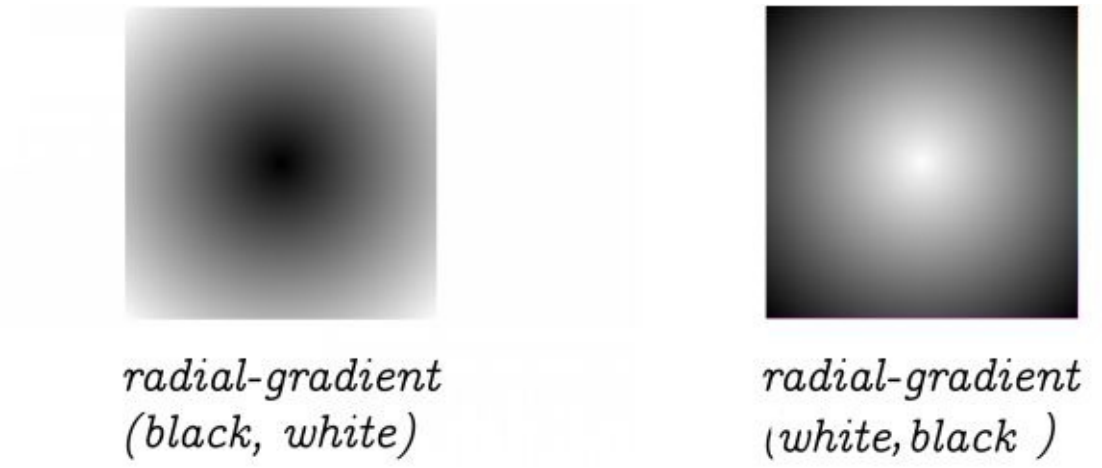



**Figure 98:** Radial gradients can be created by using **radial-gradient** property. Swapping colors around will produce an inverse effect.

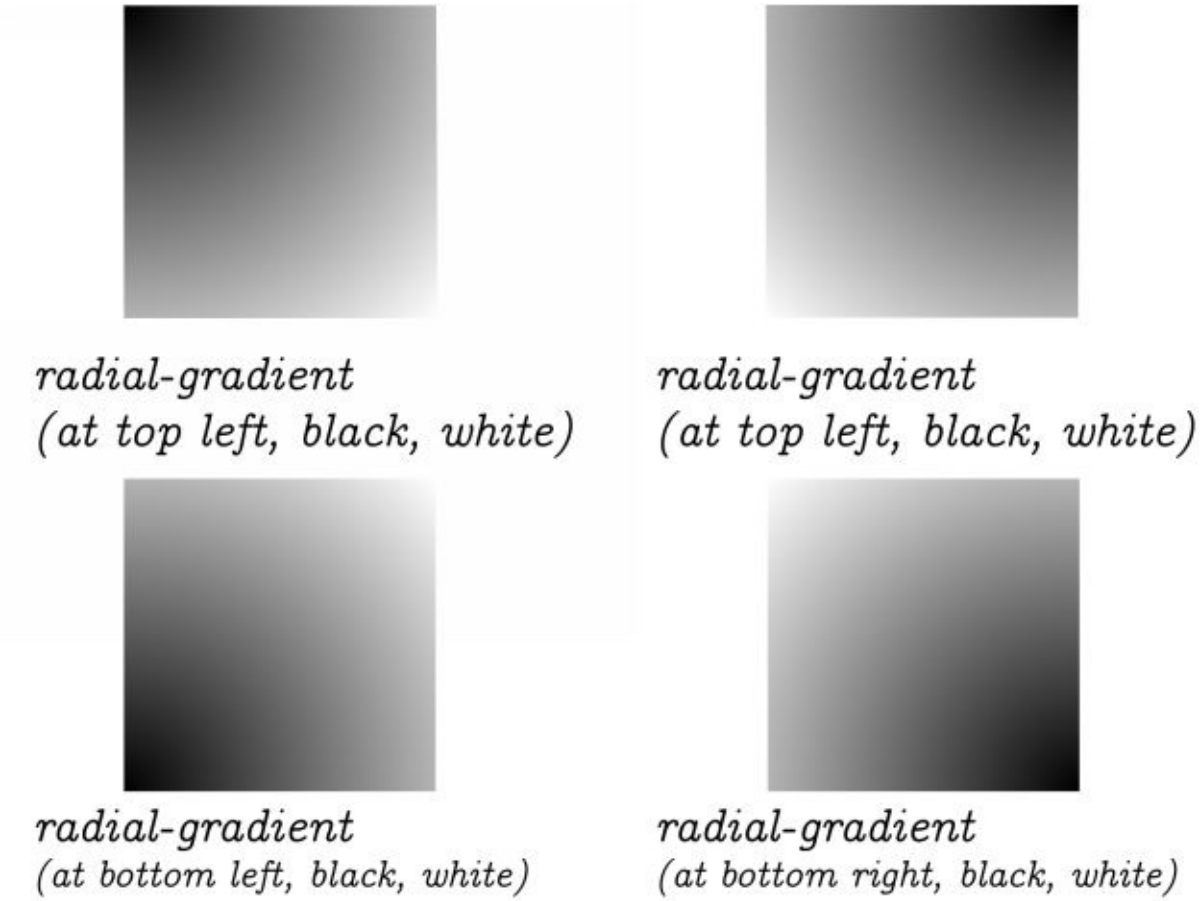
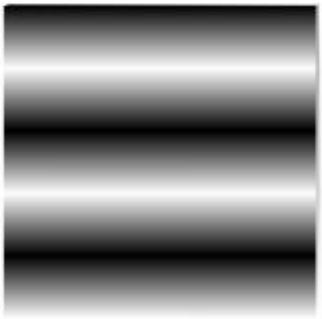
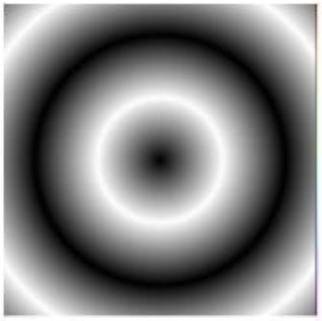



**Figure 99:** In the same way as linear gradients, radial gradients can also take origin at any of the four corners of an HTML element.

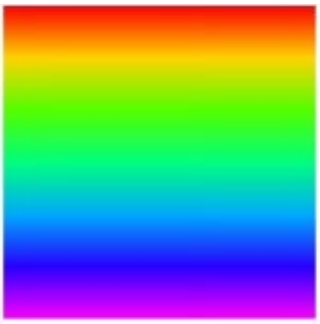


```
repeating-linear-gradient
    (white 100px,
     black 200px,
     white 300px);
```



```
repeating-radial-gradient
    (white 100px,
     black 200px,
     white 300px);
```

**Figure 100:** Repetitive patterns for linear and radial gradients can be created using **repeating-linear-gradient** and **repeating-radial-gradient** respectively. You can provide as many repetitive color values in a row as needed. Just don't forget to separate them by a comma!



```
linear-gradient
    hsl(0,100%,50%),
    hsl(50,100%,50%),
    hsl(100,100%,50%),
    hsl(150,100%,50%),
    hsl(200,100%,50%),
    hsl(250,100%,50%),
    hsl(300,100%,50%)
```

```
linear-gradient
    hsl(0,100%,50%),
    hsl(50,100%,50%),
    hsl(300,100%,50%)
```

**Figure 101:** Finally -- the most advanced type of a gradient can be created using a series of HSL values. HSL values don't have named or RGB equivalents, they

are counted on a scale from 0 -- 300. See the explanation below.


**Figure 102:** You can cherry-pick any color by using values between 0 -- 300.

We've already provided examples of property values associated with each gradient. But here they are again in one place. Play around with the values and see what type of effects they produce on your custom UI elements: Source Code 50

```
background: linear-gradient(yellow, red);
background: linear-gradient(black, white);
background: linear-gradient(to right, black, white);
background: linear-gradient(to left, black, white);
background: linear-gradient(to bottom right, black, white);
background: linear-gradient(90deg, black, white);
background: linear-gradient(
hsl(0,100%,50%),
hsl(50,100%,50%),
hsl(100,100%,50%),
hsl(150,100%,50%),
hsl(200,100%,50%),
hsl(250,100%,50%),
hsl(300,100%,50%));
background: radial-gradient(black, white);
background: radial-gradient(at bottom right, black, white);
background:
repeating-linear-gradient
(white 100px, black 200px, white 300px);
background:
repeating-radial-gradient
(white 100px, black 200px, white 300px);
```

# Background Images

So you think you know HTML backgrounds? Well maybe you do and maybe you don't. This section was created as a brief backgrounds tutorial that hopefully introduces the reader to the big picture. We'll explore several CSS properties that help us change background image settings on any HTML element.

**Figure 103:** **background: url("image.jpg")** or **background-image: url("image.jpg")** The specimen image used in this section is this adorable kitten on a stripey background.

If the element's dimensions are bigger than those of the source image, the image will be repeated within the body of that element - repetitively filling the remainder of the element's sides with the contents of the image. It's almost like stretching infinite wall paper over an element.

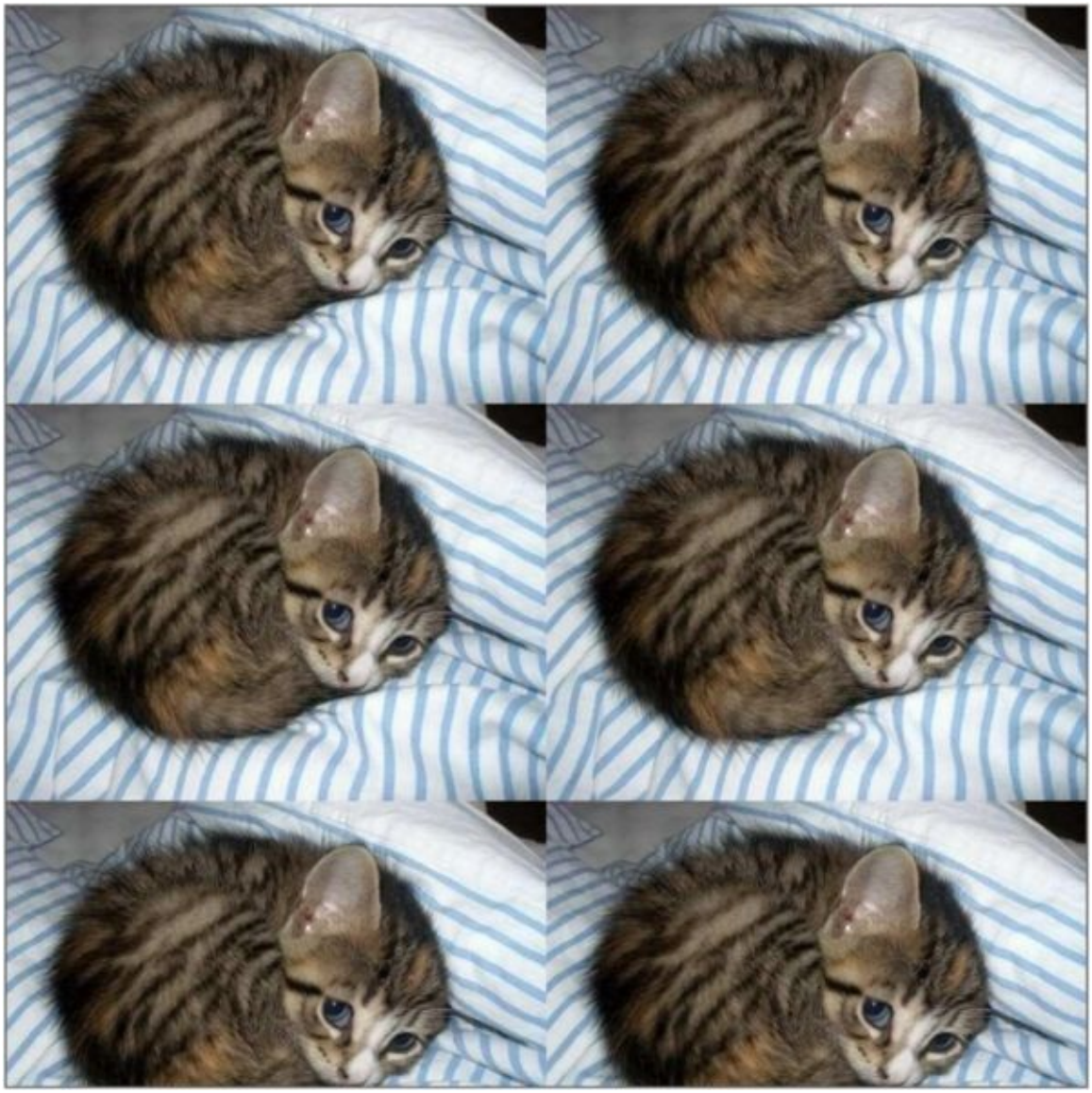


**Figure 104:** If the image is smaller than the element's dimensions, it will continue to repeat to fill up the remaining space.

To set the background image to any element you can use the following CSS commands.

Source Code 51

background: url(kitten.jpg'); Or alternatively...

Source Code 52

background-image: url(kitten.jpg'); You can also use internal CSS by placing this CSS code between **<style></style>** tags.

Let's take a look at the same kitten background... except this time around with **no-repeat** value set with the additional **background-repeat** property:

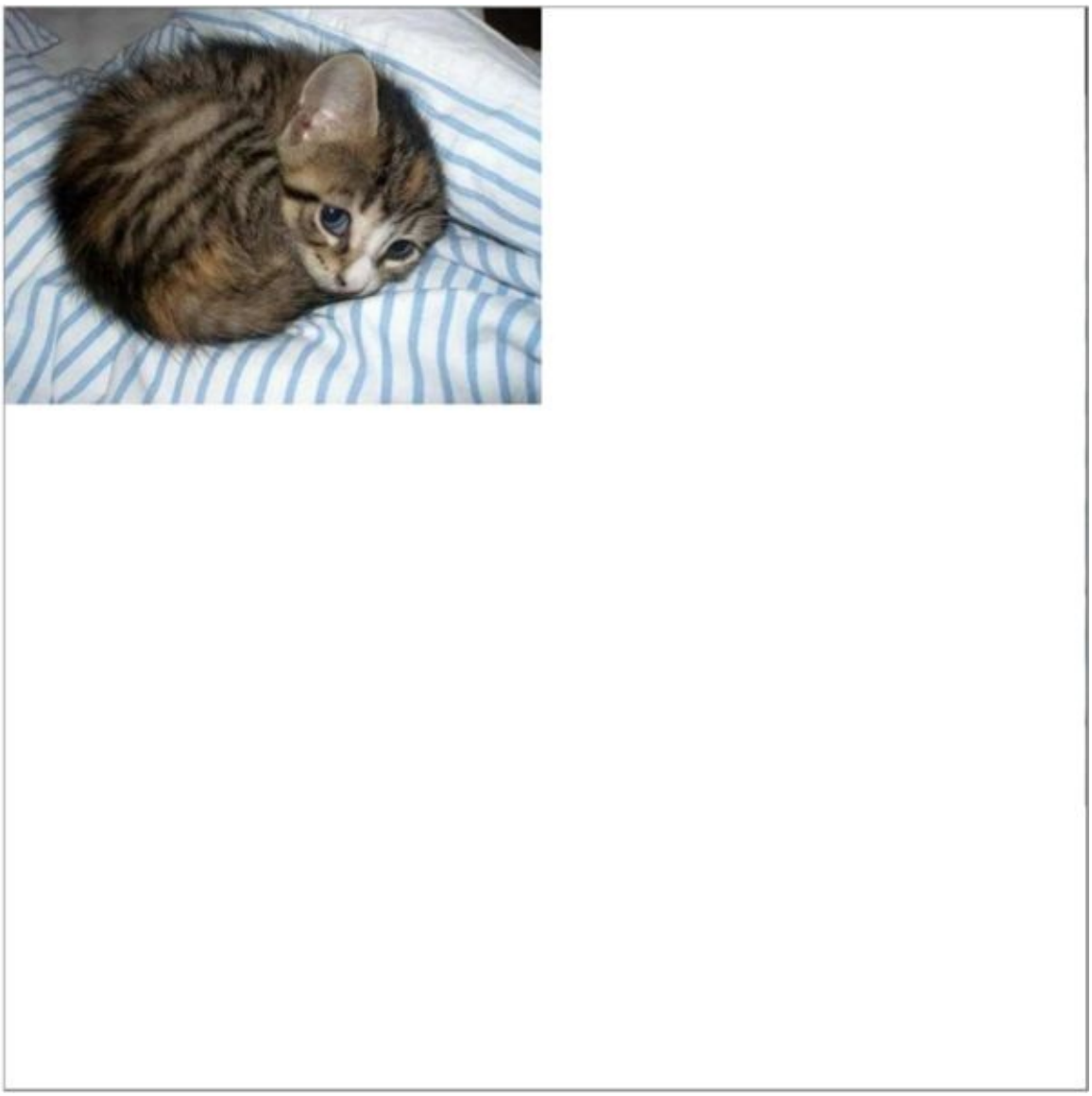


**Figure 105: background-repeat: no-repeat;**


**Figure 106:** A closer look at the results created with **background-size**. From left to right examples are listed as follows: (*unset*|*none*|*initial*|*auto*) which all produce default behavior. The value of **100 %** will stretch the images in horizontal direction, but not vertically. The value of **100 % 100 %** will stretch the image across all available space. The value **cover** will stretch the image across entire vertical space of the element, it will cut off everything in the horizontal direction, similar to overflow. The value **contain** will make sure that the image is stretched horizontally across the width of the element, and while remaining original proportion, stretch it vertically for however long it needs to, repeating the image until it overflows at the bottom of the element.

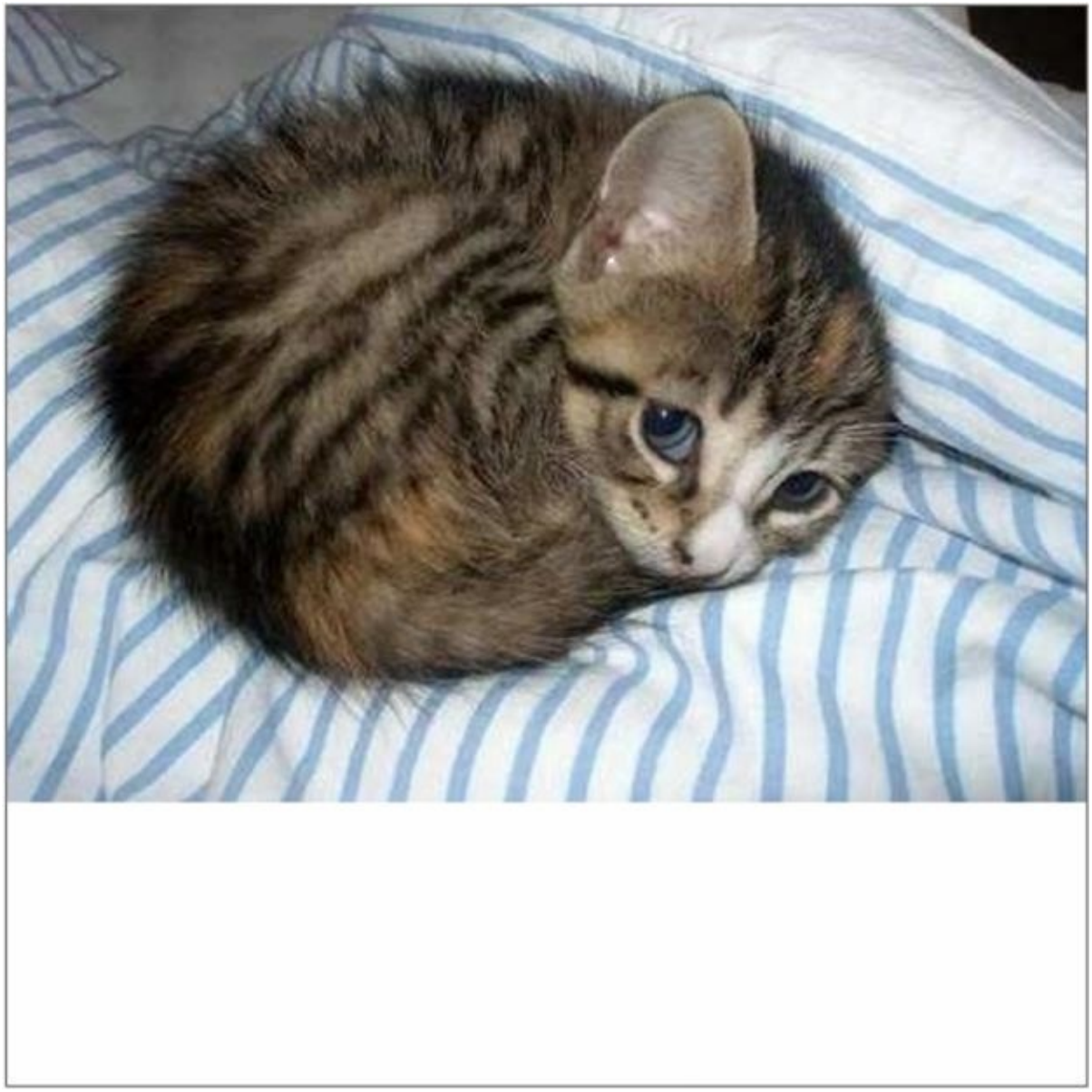


**Figure 107:** By combining **background-repeat: no-repeat; with background-size: 100 % it is possible to stretch the image only horizontally, across the entire width of the element.**

What if you want to repeat background vertically but keep it stretched across the width? No problem, simply remove no-repeat from previous example.

This is what you will end up with:

**Figure 108:** Repeat vertically.

***Above:*** This HTML / CSS background technique is used for sites whose content stretches vertically over a long area of space. I think one of the iterations of the Blizzard site used it in the past. Sometimes you want to cut it off, and make it static. Other times you want it to go on forever vertically. This will depend on your vision of the layout.

Sometimes it is needed to stretch the image across to fit the bounding box of an element. This often comes at a price of some distortion, however. CSS will automatically stretch the image according to some automatically-calculated percentage value:

**Figure 109:** Needless to say, this effect will only be observed when the HTML element and the size of the image do not match.

***Above:*** Set **background-size:100**
**% 100**
**%** to stretch the image.
Note here, **100**
**% 100**
**%** is repeated twice. The first value tells CSS to stretch the image vertically and the second **100**
**%** does the same horizontally. You can use values between **0 100**
**%** here although I do not see many cases where this would be necessary.

# Specifying Multiple Values

In HTML, whenever you need to specify multiple values they are often separated by a space. Vertical coordinates (***Y-axis*** or ***height***) always come first. Sometimes values are separated by comma. Example? When we need to specify multiple backgrounds they are usually separated by comma and not the space character. (*As we will see from the last section in this tutorial.*)

# background-position

This is **background-position: center center** at work here.
You can force the image to be always in the center but lose repetitiveness of the pattern by specifying **no-repeat** value to **background-repeat** property:

Center the image: Source Code 53
background-position: center center; Turn repeat off: Source Code 54
background-repeat: no-repeat; You can repeat the image across the *x-axis* only (*horizontally*) using **repeat-x**:

**Figure 110:** This is **repeat-x** in action.

You can easily center and repeat the image only horizontally by supplying **repeat-x** as the value for the **background-repeat** property.

To the same effect but on the *y-axis* **repeat-y** property can be used:

**Figure 111:** Vertical wallpaper with **repeat-y**.

Like any other CSS property you have to juggle around the values to achieve the results you want. I think we covered pretty much everything there is about backgrounds. Except one last thing...

# Multiple Backgrounds

It is possible to add more than one background to the same HTML element. The process is rather simple.
Consider these images stored in two separate files:

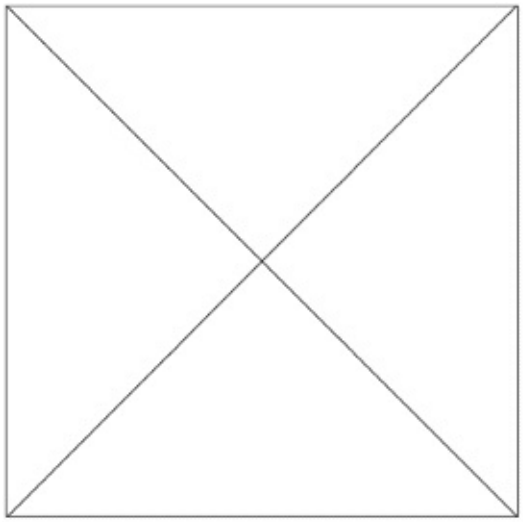


*image1.png*

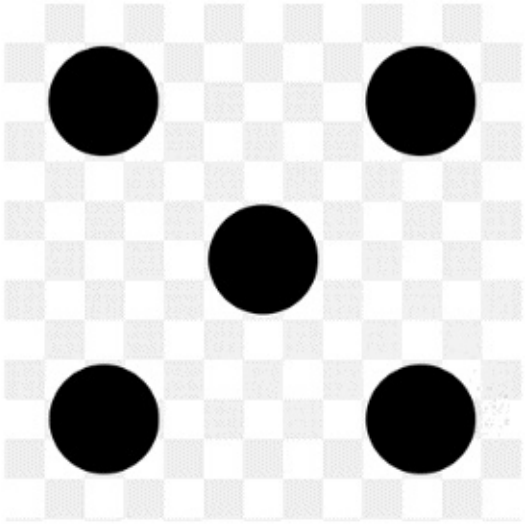


*image2.png*

The chessboard pattern in the image on the right is only used to indicate transparency here. The white and grayish squares are not an actual part of the image itself. This is the "see-through" area which you would usually see in digital manipulation software.
When the image on the right is placed on top of other HTML elements or images, the checkered area will not block that content underneath. And this is the whole idea behind multiple backgrounds in HTML.

# Image Transparency

To fully take advantage of multiple backgrounds one of the background images should have a transparent area. But how do we create one?
In this example, the second image **image2.png** contains 5 black dots on a transparent background indicated by a checkered pattern.

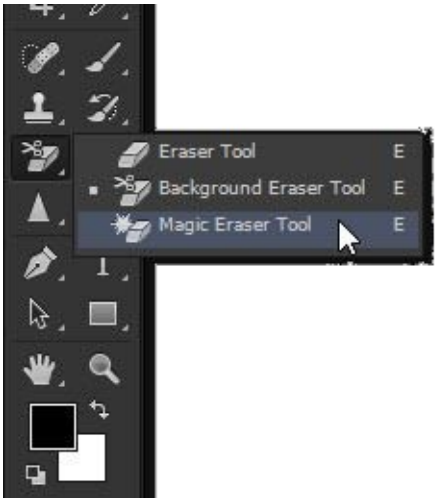



**Figure 112:** Finding the Magic Eraser Tool in Photoshop.

Like many other CSS properties that accept multiple values - all you have to do - to set up multiple backgrounds is to provide a set of values to the background property separated by comma.

# Multiple Backgrounds

To assign multiple (*layered*) background images to the same HTML element, the following CSS can be used: Source Code 55

```
body { background: url("image2.png"), url("image1.png"); }
```

The order in which you supply images to the background's **url** property is important. Note that the top-most image is always listed first. This is why we start with **image2.png**.

This code produces the following result:

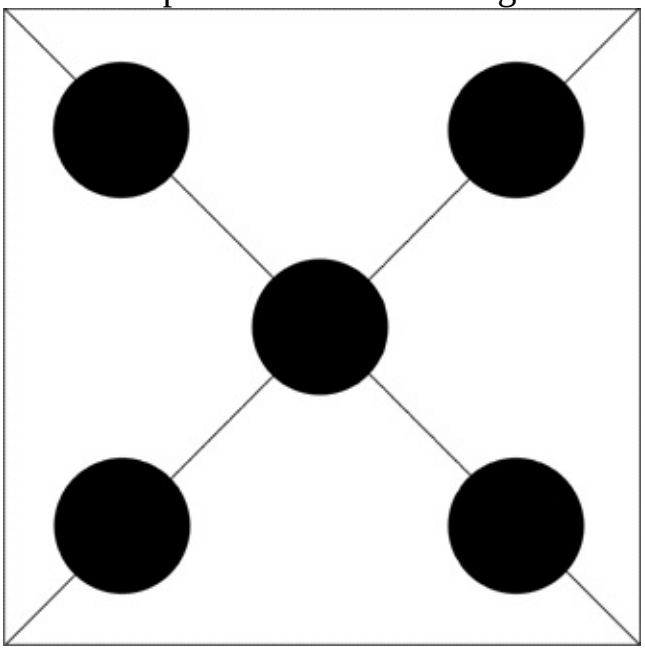
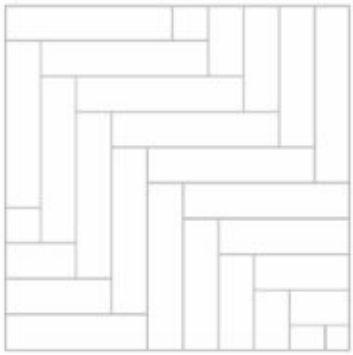


**Figure 113:** Superimposing a transparent image over another one using multiple backgrounds in CSS.

In this example we demonstrated multiple backgrounds in theory on a subjective ** ** (*or similar*) element with square dimensions.

Let's take a look at another example.

puppy.png



pattern.png

Note here that the **puppy.png** image will be the first item on the comma-separated list. This is the image we want to superimpose on top of all of the other images on the list.
Combining the two: Source Code 56
body { background: url(�puppy.png'), url(�pattern.png') We get the following result:

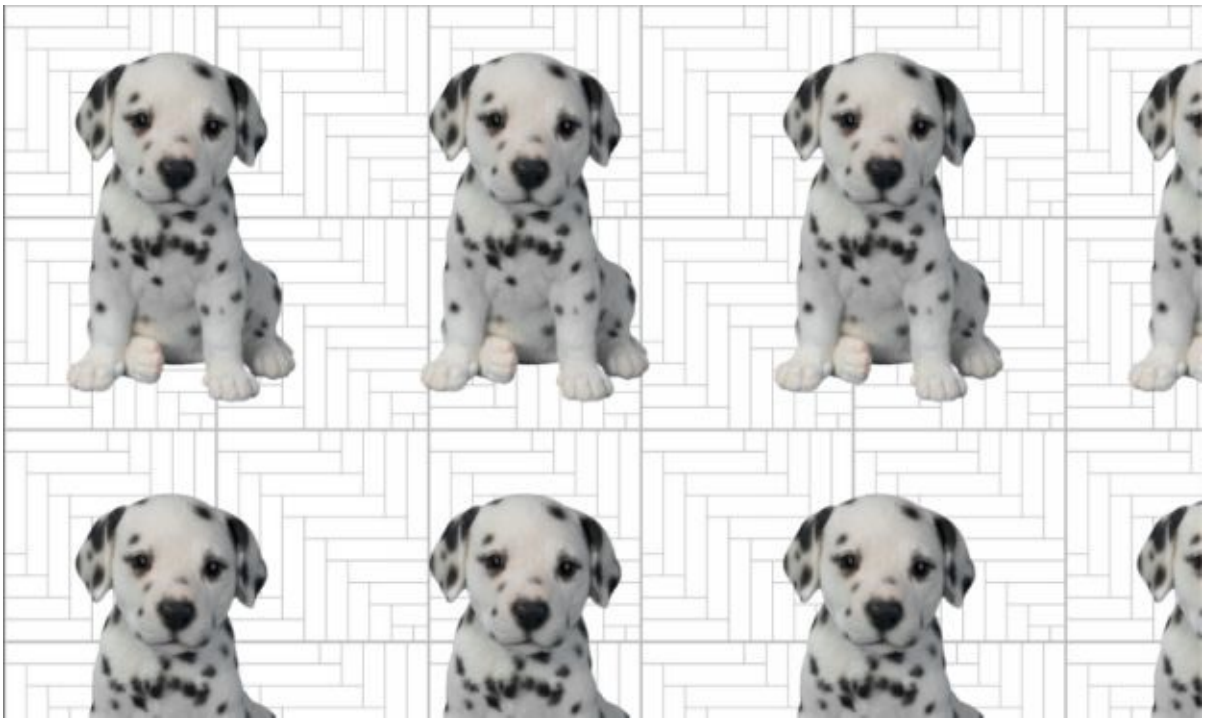


**Figure 114:** Background 15

Other background properties that also take comma-separated lists exist. Pretty much every other background related property other than background-color. In the same way, you can supply other parameters to each individual background, using the other background properties demonstrated below: Source Code 57

background
background-attachment
background-clip
background-image
background-origin
background-position
background-repeat
background-size The following property cannot be used with a list for obvious reasons: Source Code 58

background-color What would it mean to provide multiple color values to a background? Whenever color background property is set, it usually fills the entire area with a solid color. But multiple backgrounds require that at least one of the backgrounds contains transparency of some sorts. Therefore, it cannot be used in the case of multiple backgrounds for any meaningful purpose.

But that's not all you can say about background images. Let's finish our discussion by taking a look at these other few cases.

# background-attachment

You can determine behavior of the background image relative to scroll bar.

**Figure 115: background-attachment:scroll**

Before (*left*) and after (*right*) images are shown here.

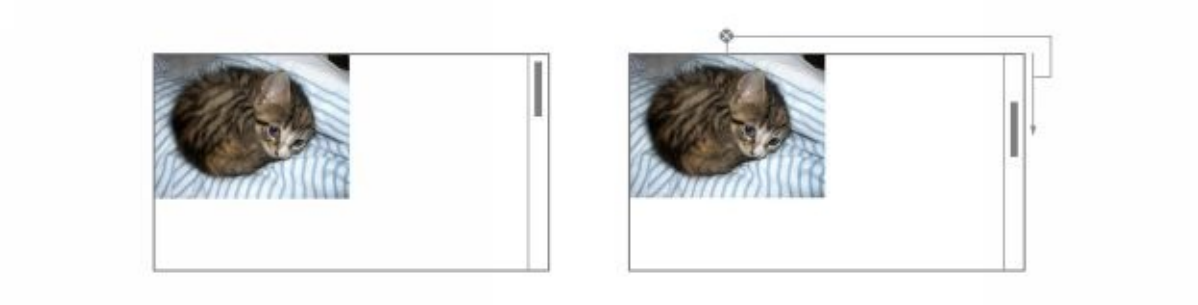


**Figure 116: background-attachment:fixed**

Fixed backgrounds don't respond to the scroll bar.

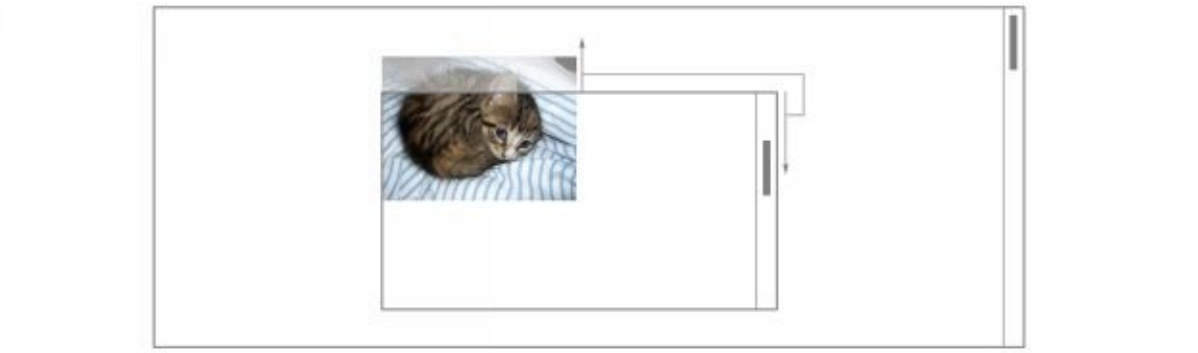


**Figure 117: background-attachment:scroll**

# background-origin

Property **background-origin** determines the extent of the area that will be used by the background image, based on the *CSS Box Model*.


**Figure 118: content-box | padding-box | border-box**


**Figure 119: content-box | padding-box | border-box**

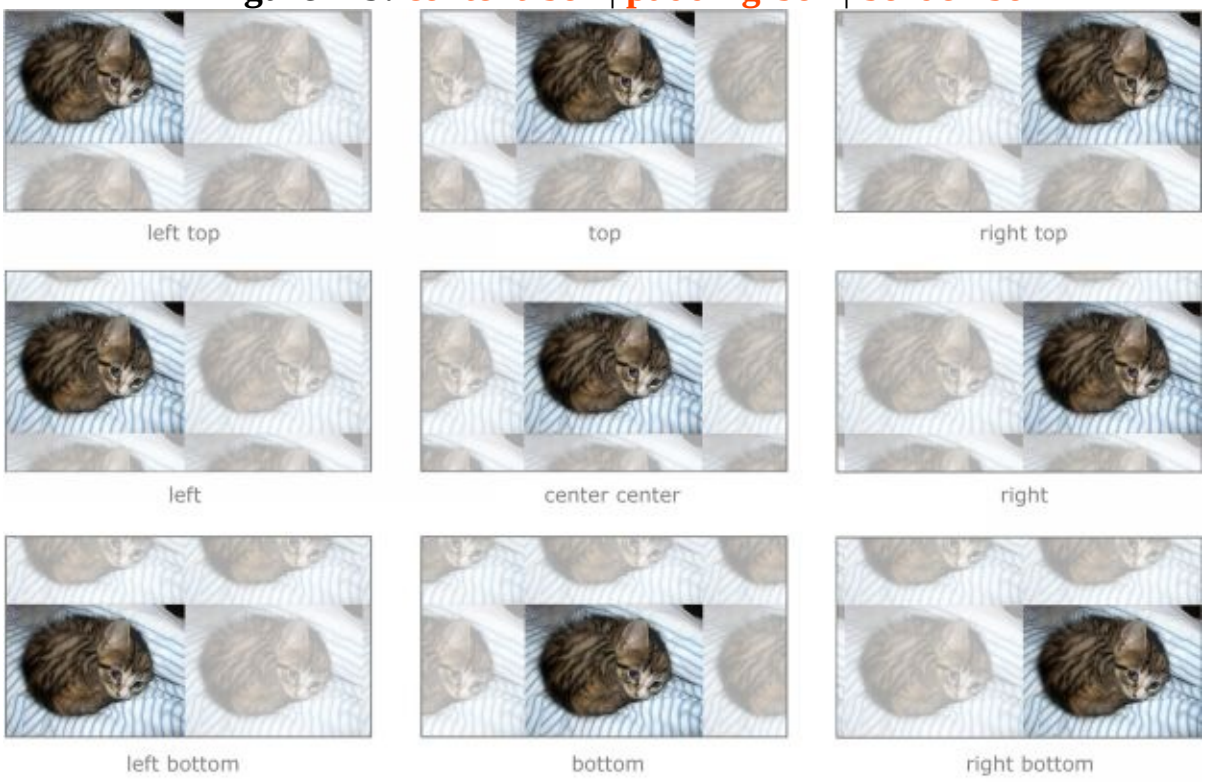


**Figure 120:** background-position-x and background-position-y are supplied the following values to create any of the background positioning patterns: **left top** | **top** | **right top** | **left** | **center center** | **right** | **left bottom** | **bottom** | **right bottom**.

And finally... in addition to images, the **background** property can also specify either a *solid color*, a *linear gradient* or a *radial gradient*.

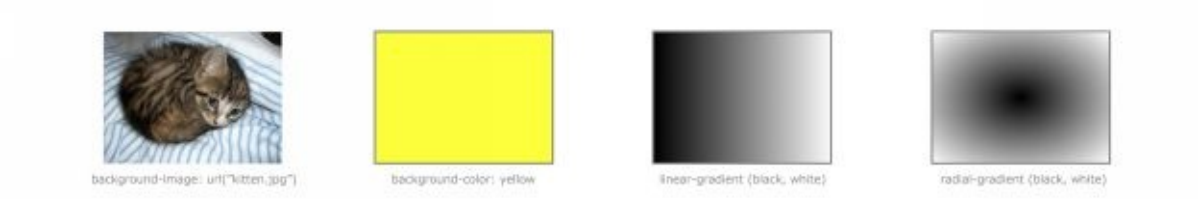



**Figure 121:** Examples of other possible values supplied to the **background** property. Note that an entire chapter is dedicated to describing *linear* and *radial* gradients in this book.

# object-fit

Some of the backgrounds functionality has been superseded by a slightly different image-fitting solution based on **object-fit** property. By providing various values you can achieve any of the following results:

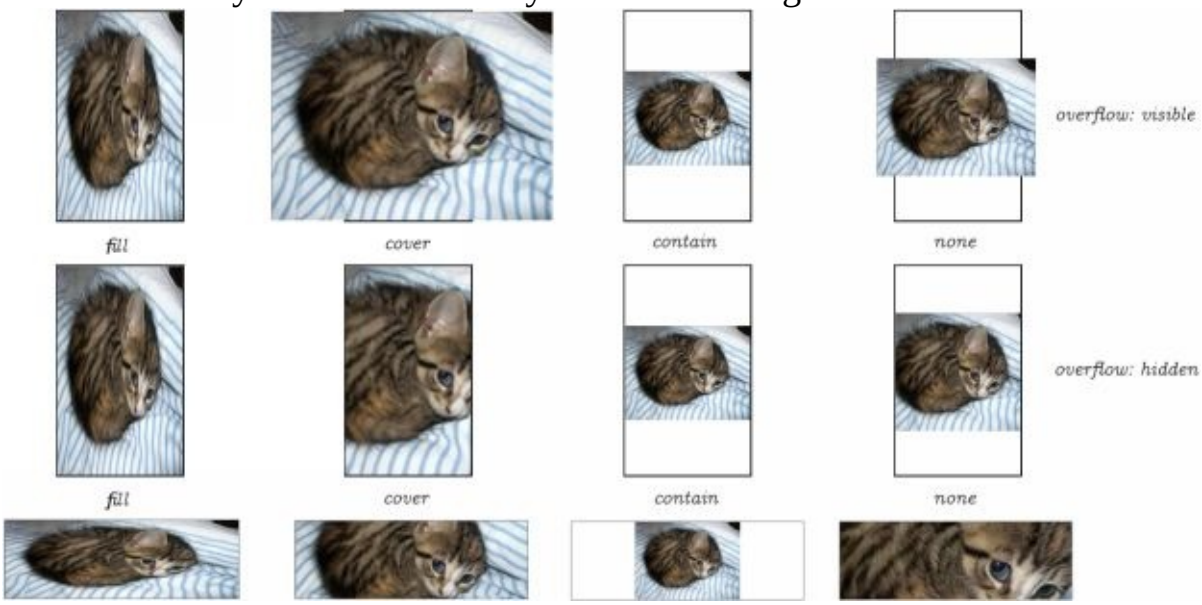



**Figure 122:** Here **object-fit** property presents us with pretty much every single possible case of how we wish to fit our object into the parent container. Note that although similar to background property, object-fit works with non-background images (created using the **<img>** tag), videos and other "objects", rather than background images.

The available values demonstrated in the above image examples reading from left to right are: **fill**, **cover**, **contain** and **none**.

The first row has **overflow:visible**. The second row has **overflow:hidden**. And the third row is the same as the second, but in this example the dimensions of the actual HTML element were flipped to demonstrate that it does produce a slightly different results when either vertical or horizontal dimension is prevalent over the other.

# Borders

There is much more to CSS borders than meets the eye. In particular, you want to learn how border radius (*only when values for both X and Y axis are provided*) affects other corners of the same element. But before moving forward, let's take a look at borders.

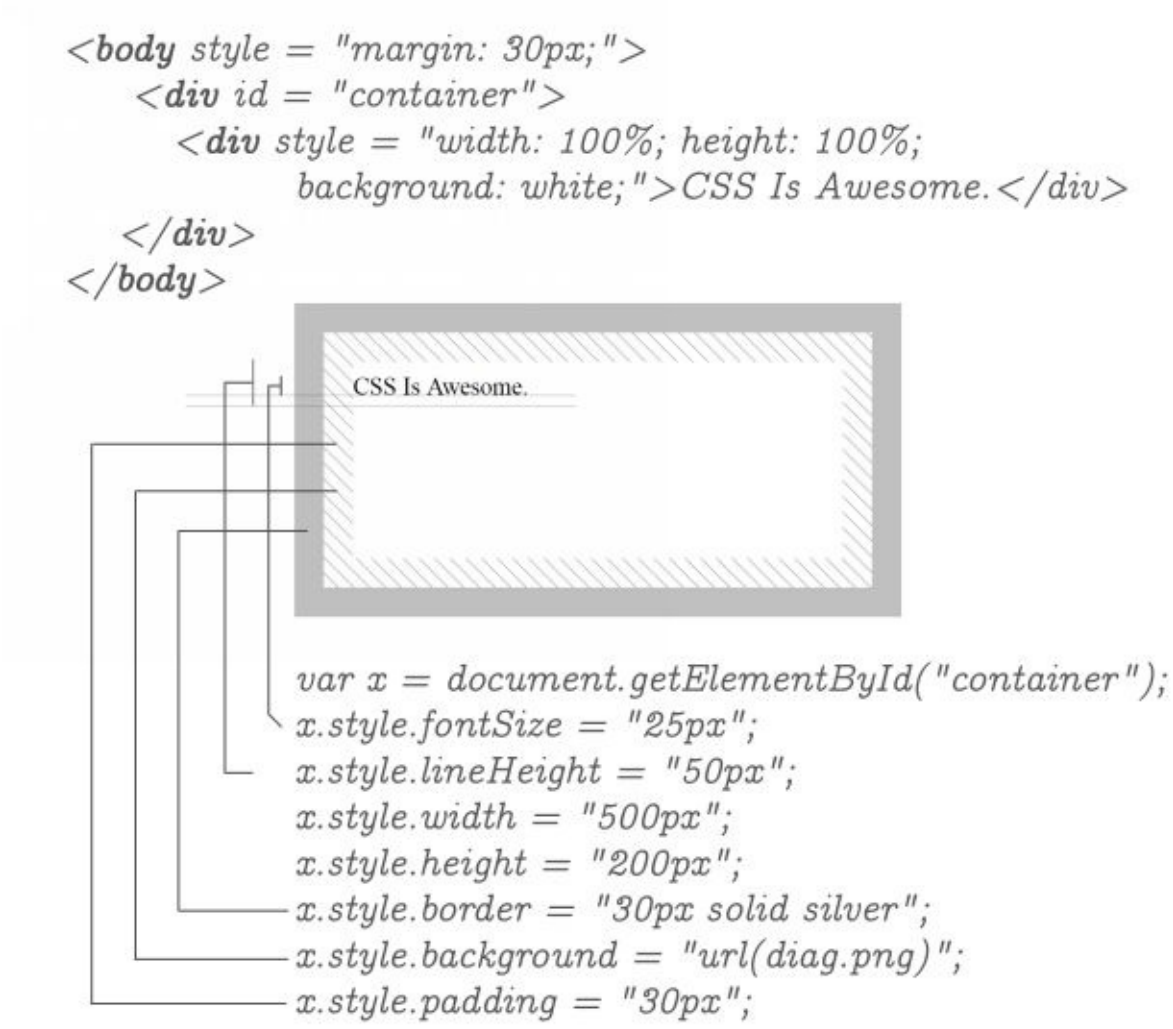



**Figure 123:** You can easily access all of the same CSS properties via JavaScript. Just grab an object with **document.getElementById("container")** -- for example -- to gain access to all CSS properties. They are attached to element***.style*** property on the object.

Borders can be set on all sides at the same time with the **border** property.

Source Code 59

border: 5px solid gray; You can also set the curvature on each of the four

corners of the element with **border-radius**, by specifying the radius of the circle:

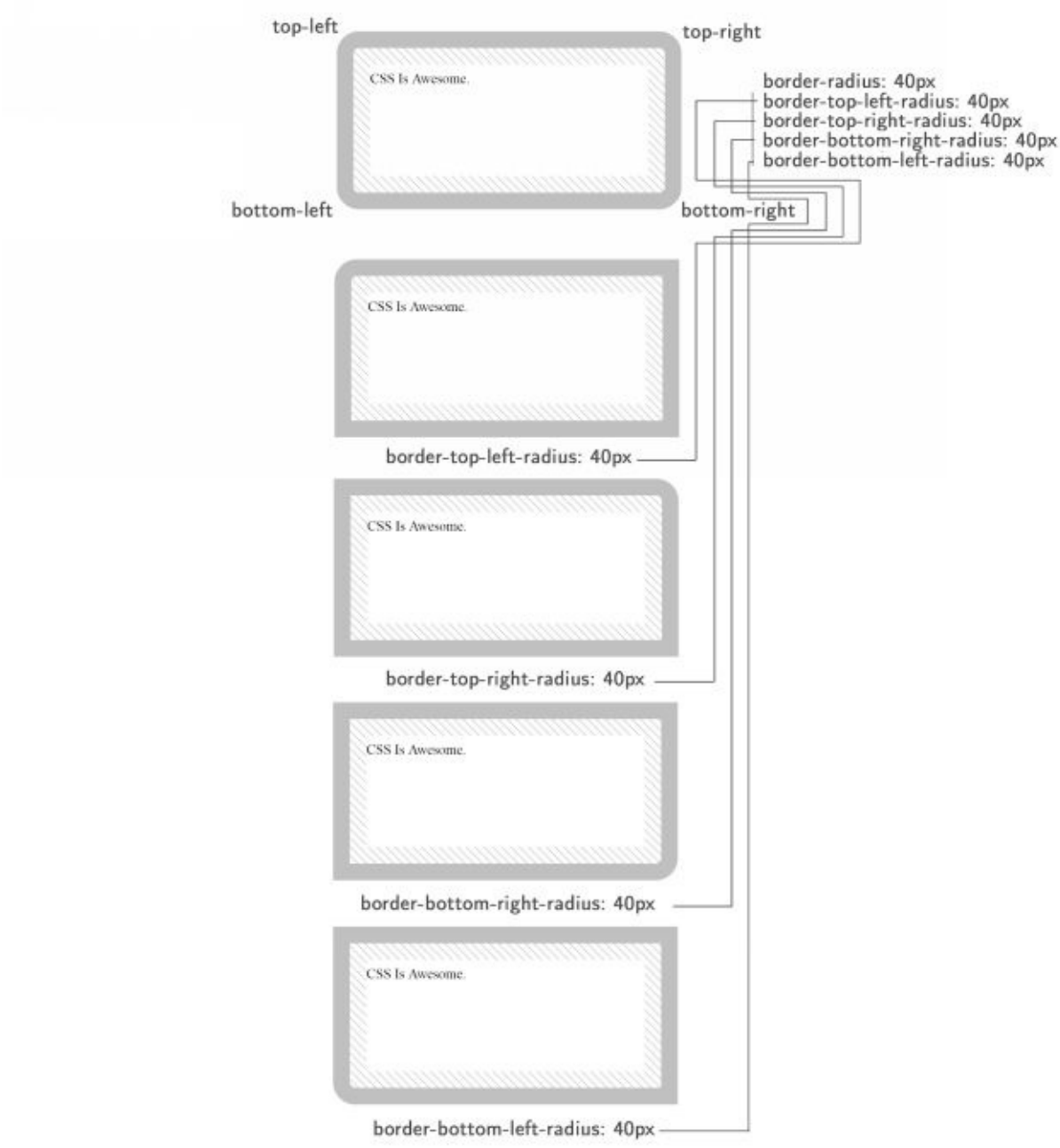



**Figure 124: border-radius**

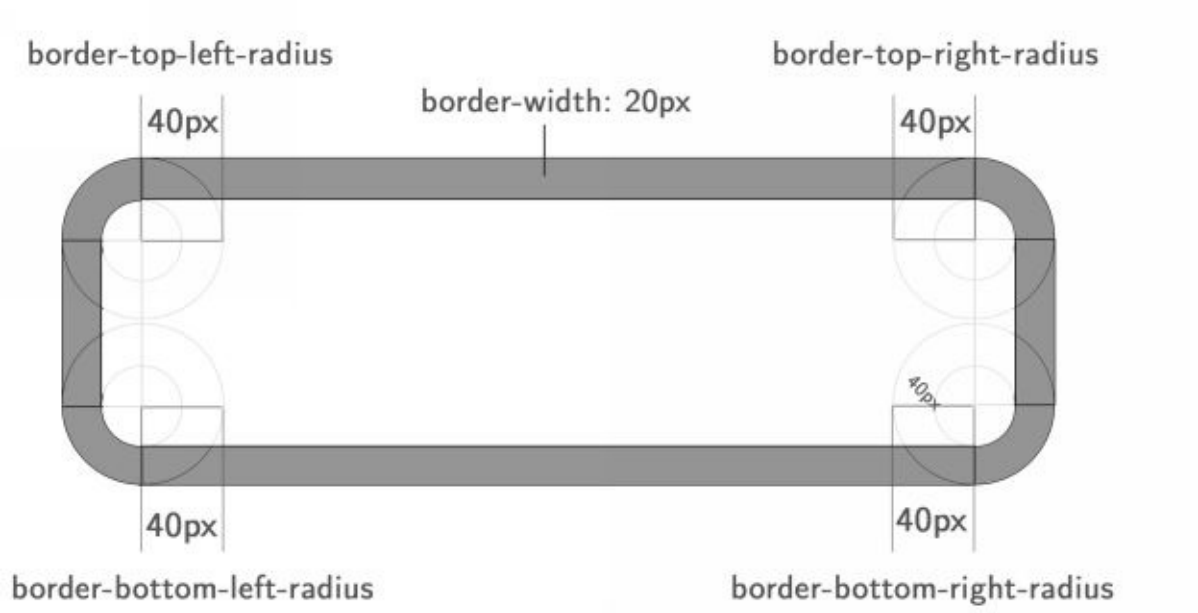



**Figure 125:** **border-top-left-radius** | **border-top-right-radius** | **border-bottom-left-radius** | **border-bottom-right-radius** .

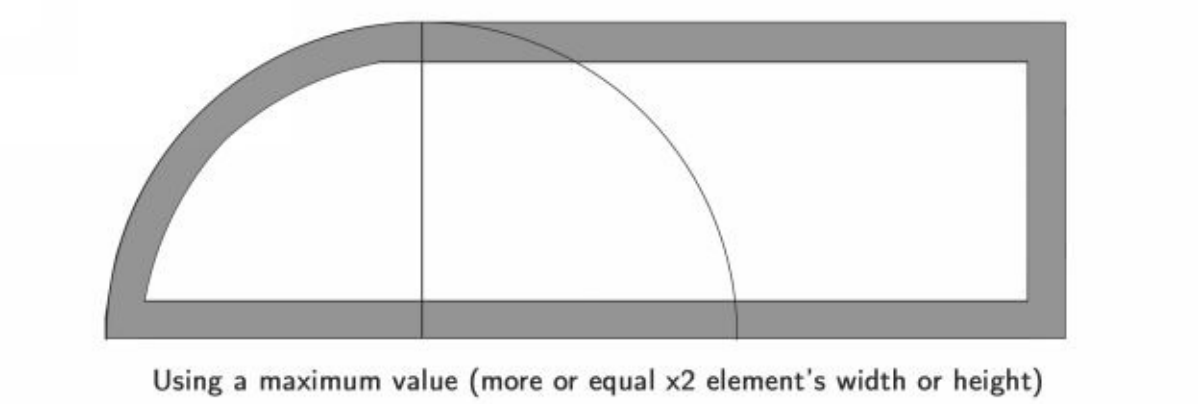



**Figure 126:** Using a value equal to or greater than the size of an element's side -- to which border radius is applied -- will be clamped to the greatest radius that fits in that area.

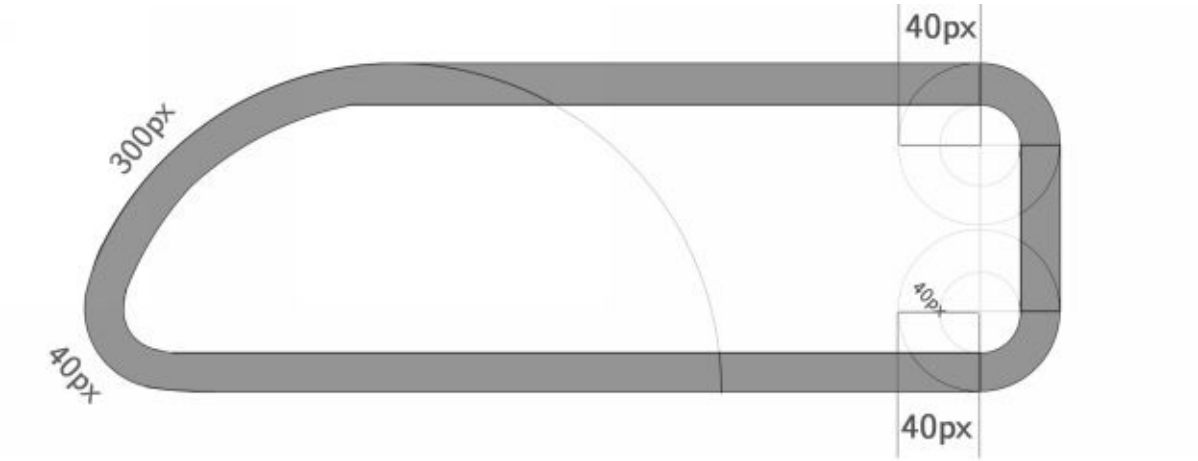



**Figure 127: border-top-left-radius:300px | border-top-left-radius:40px | border-bottom-left-radius:40px | border-bottom-right-radius:40px**

# Elliptical Border Radius

Even after a long time working with CSS I still failed to notice that **border-radius** property can be used to create elliptical borders. But indeed, this is true. The results of elliptical curves are not always as easily predictable as is the case with axis-uniform radius values.

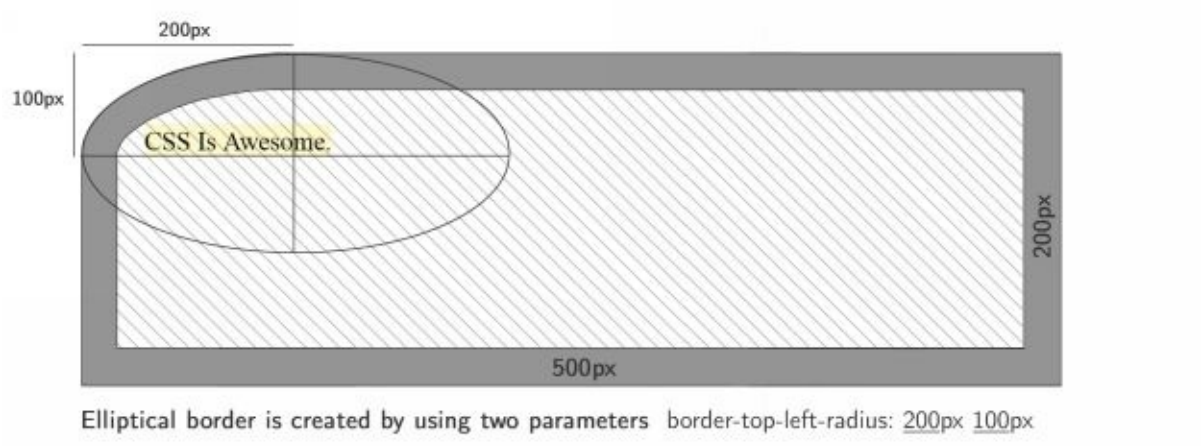



**Figure 128: border-top-left-radius:200px 100px**

Elliptical radius is set by specifying two values for each axis on the same corner, separated by space.

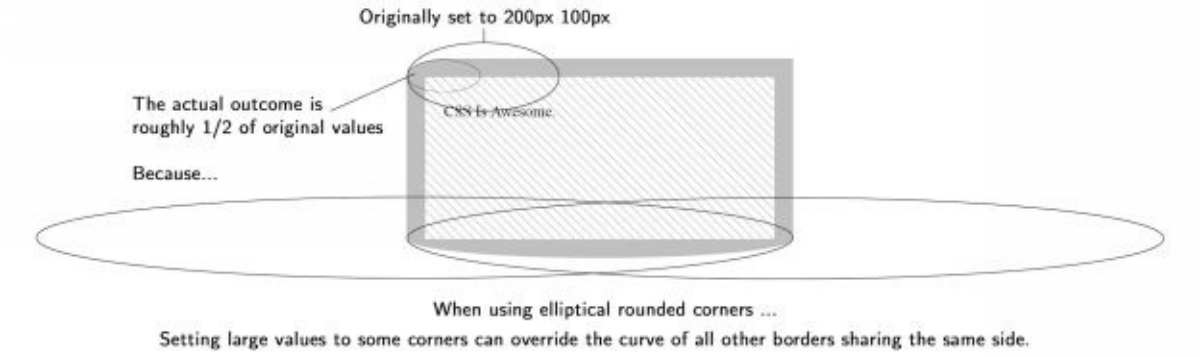



**Figure 129:** When using elliptical radius with extremely large values the curve of one corner can affect the curve of adjacent corners, especially ones with smaller radius values. This is where things get a bit unpredictable. But the good thing is that this level of looseness opens room for more creative experimentation. You just have to play around with different values to achieve a certain effect or a curve you're looking for.

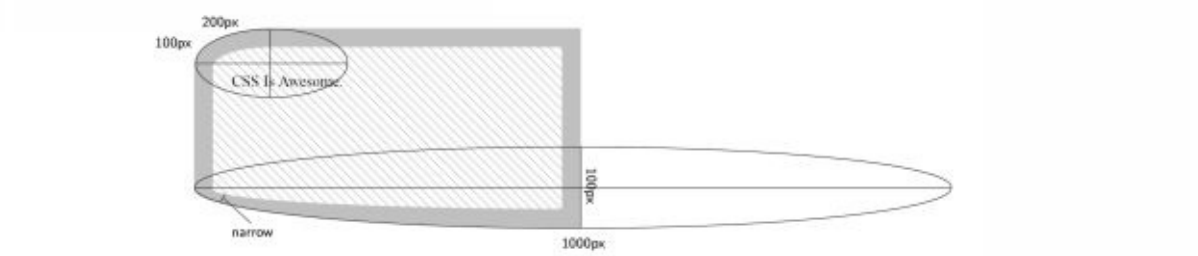



**Figure 130:** The principle behind applying large values to the elliptical corners.

**Figure 131:** In this example, we are changing only the value of the upper right corner's curve. Notice however, that all rounded corners of the element are codependent to one another -- even the ones whose values we are not changing explicitly.

# 2D Transforms

2D transforms can **translate**, **scale** or **rotate** an HTML element.

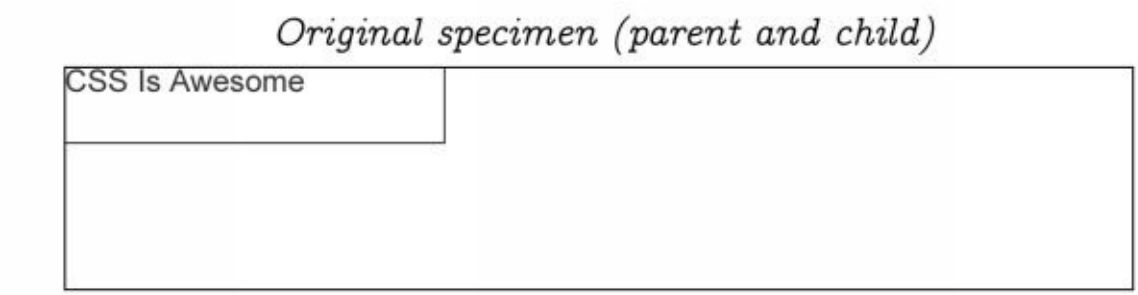



**Figure 132:** We'll use this simple HTML element specimen to demonstrate 2D CSS transforms.

# translate

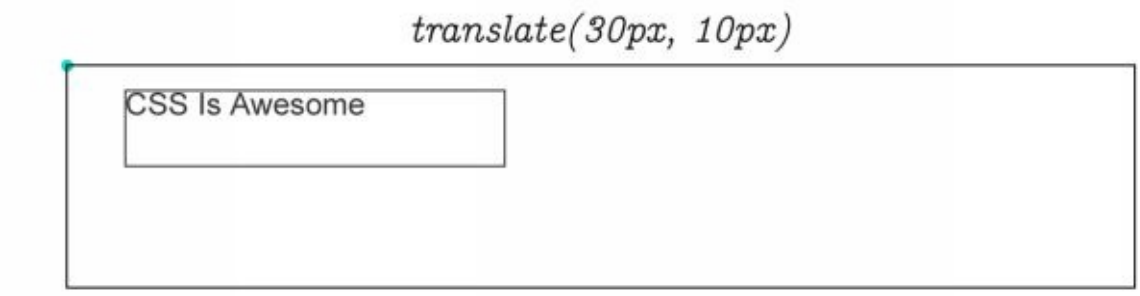



**Figure 133:** Instead of using top and left properties, we can use **transform:translate(30px,10px)** to move the element on its X and Y axis.

# rotate

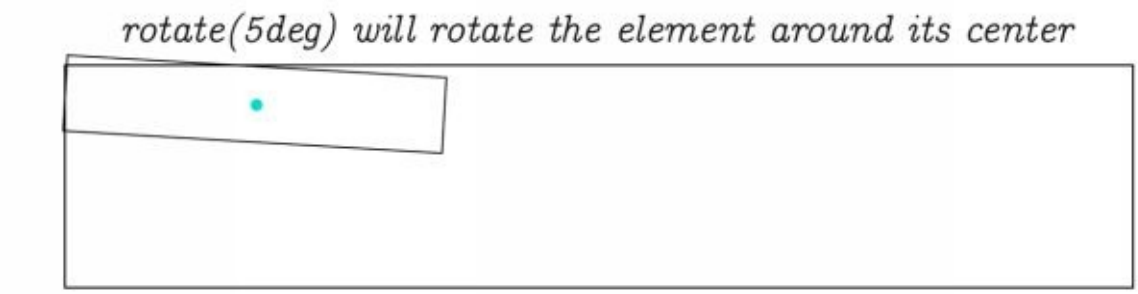



**Figure 134:** Rotating an element around its center using **rotate(*angle*)**, where *angle* is an angle between 0 and 360 degrees, with "deg" appended.

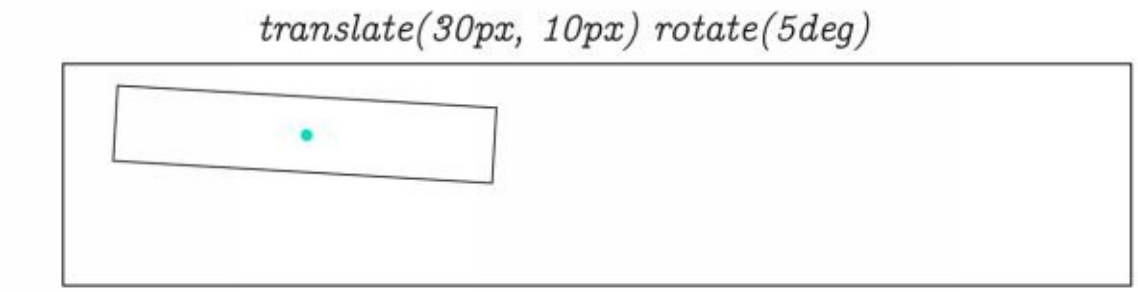



**Figure 135:** It's possible to translate and rotate an element.

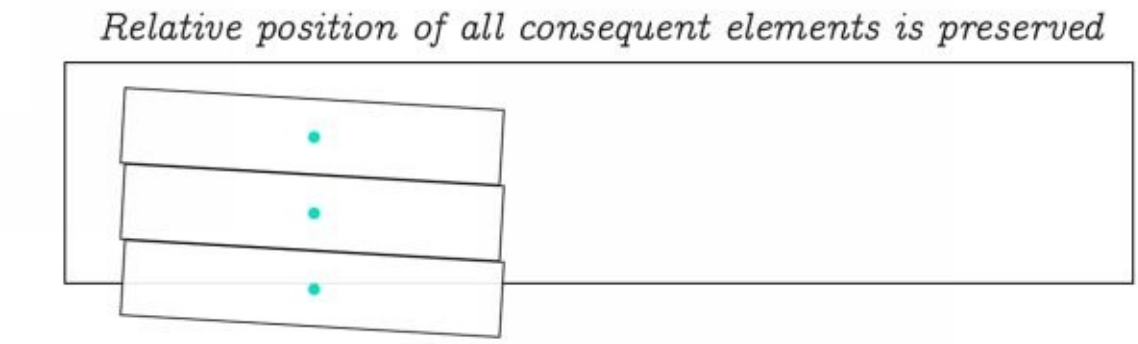



**Figure 136:** Three elements with **display:block;position:relative;** set to the same angle.

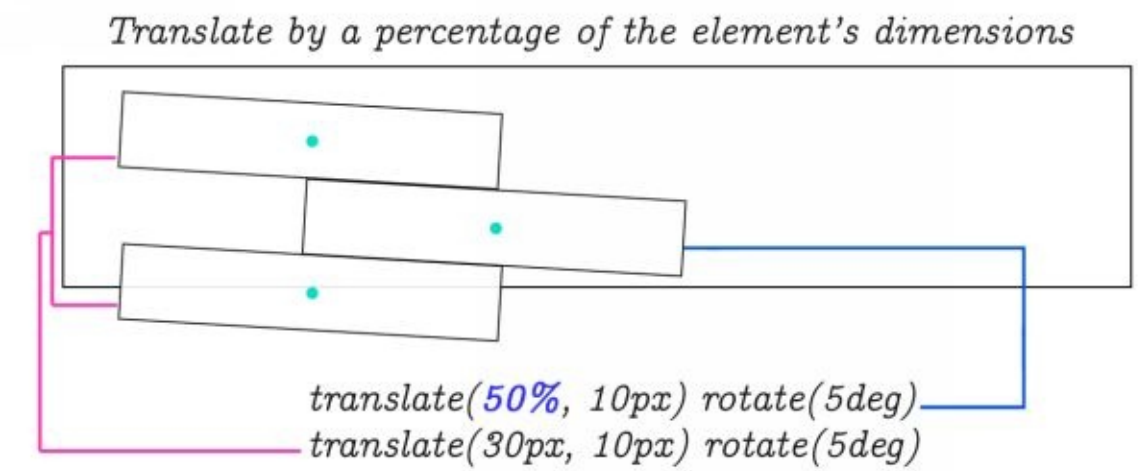



**Figure 137:** Translate transform can take a percentage of the element's size.

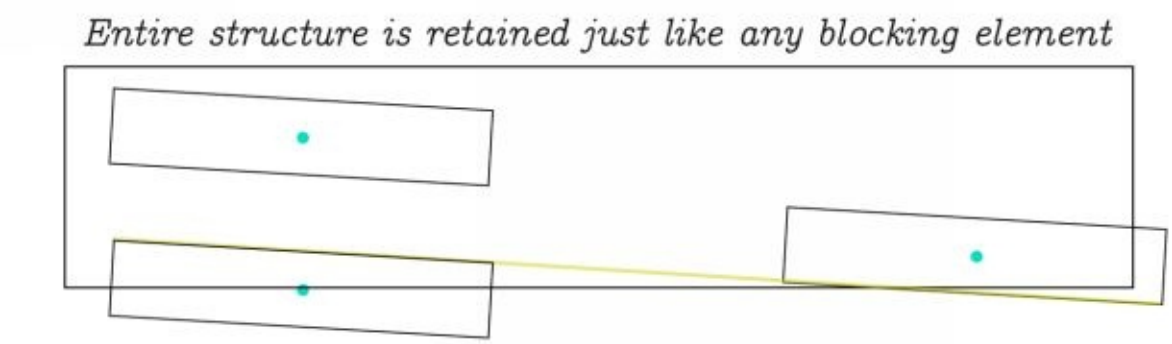



**Figure 138:** Relative elements retain their position within the document even after rotation.

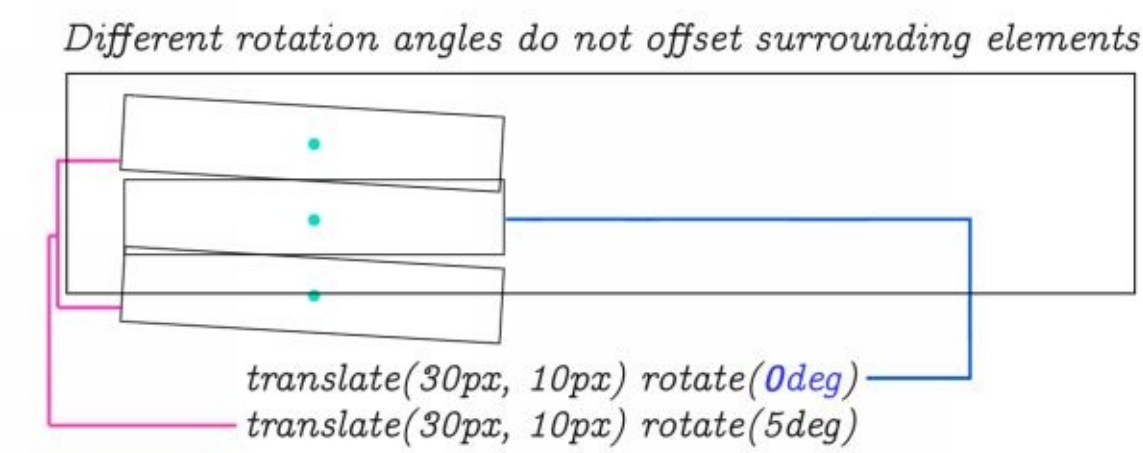



**Figure 139:** Rotating an element between others does not affect their position. The edges will overlap.

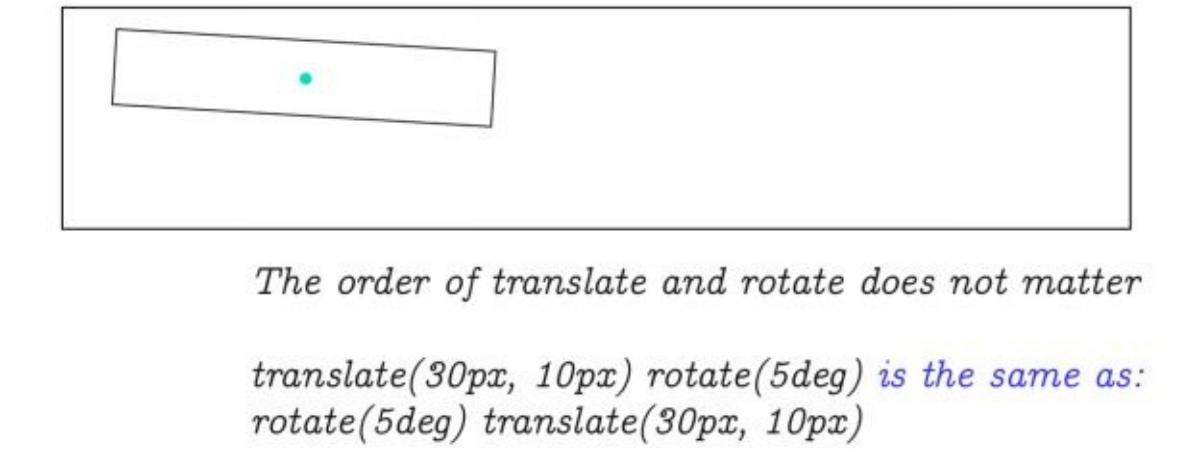



**Figure 140:** The order is irrelevant.

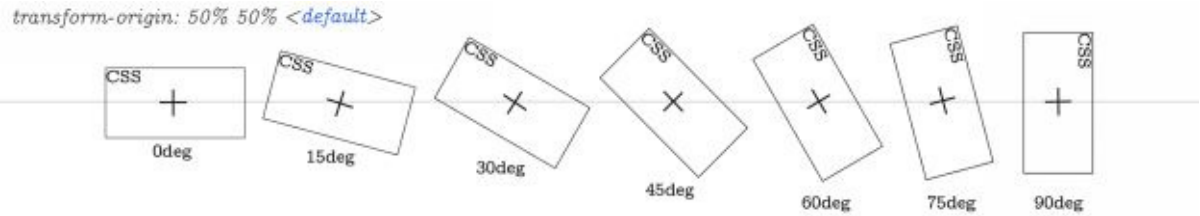



**Figure 141:** Rotate transform will rotate the element around its midpoint by default.

# transform-origin

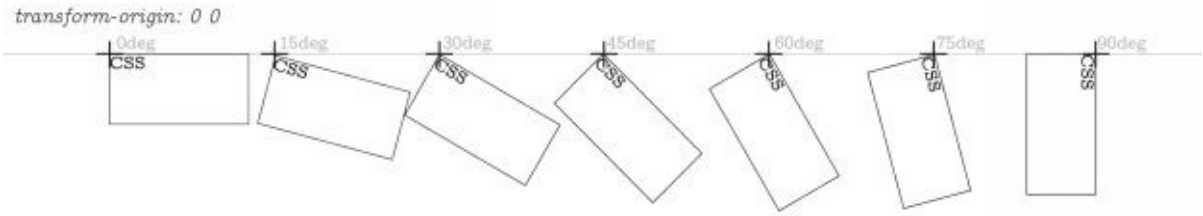



**Figure 142:** Moving element rotation origin using **transform-origin:0 0;**

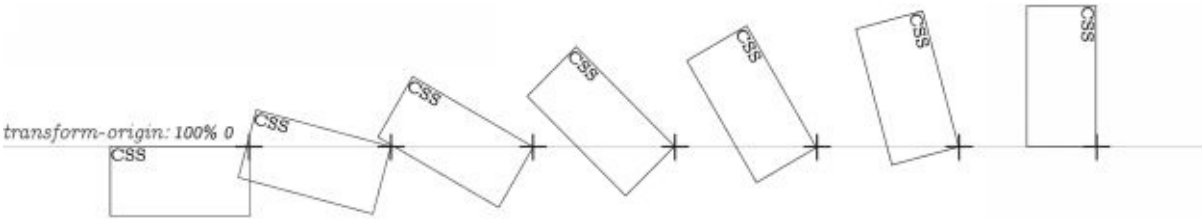



**Figure 143:** **transform-origin:100 % 0**

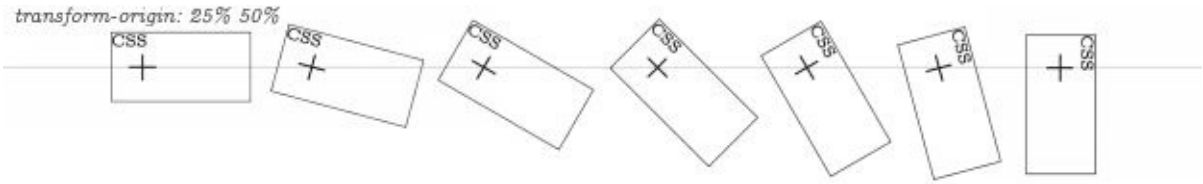



**Figure 144:** The rotation origin doesn't have to be in the middle or at the corners of the element. It can be anywhere.

# 3D Transforms

3D transforms can transform your regular HTML elements into 3D by adding perspective.

# rotateX

Let's rotate the element on X-axis using **transform:rotateX**.

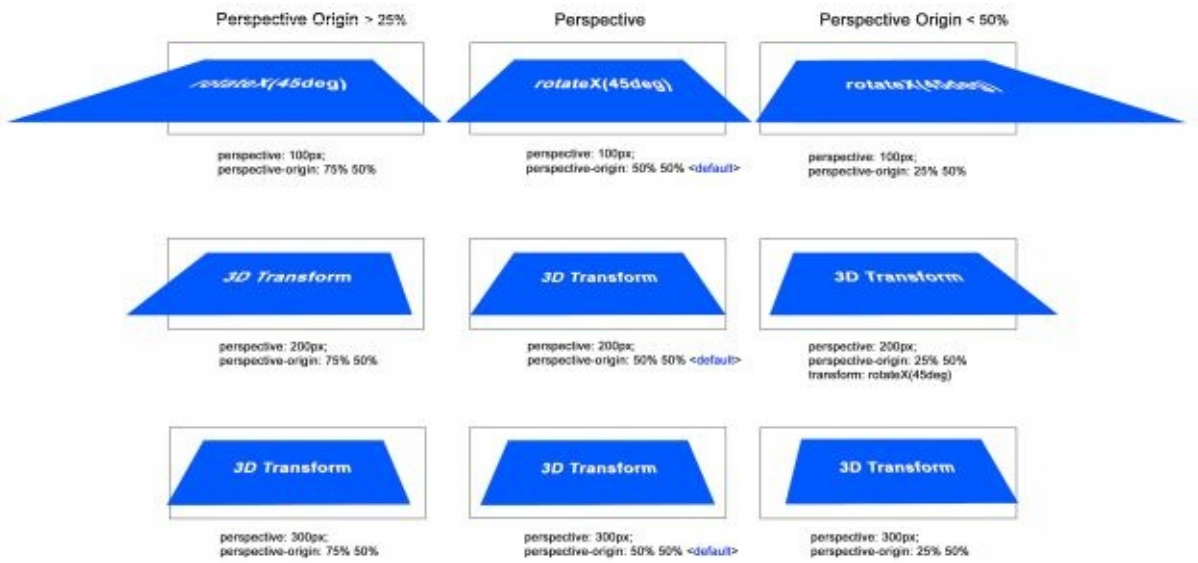



**Figure 145:** Each row in this example portrays what happens to an HTML element when its perspective is changed from 100px, to 200px and then to 300px from top down, using **perspective** property. The **perspective-origin** property is also used to demonstrate the slant created when the origin is displaced.

# rotateY and rotateZ

Rotating the element on Y and Z axis produces these results:

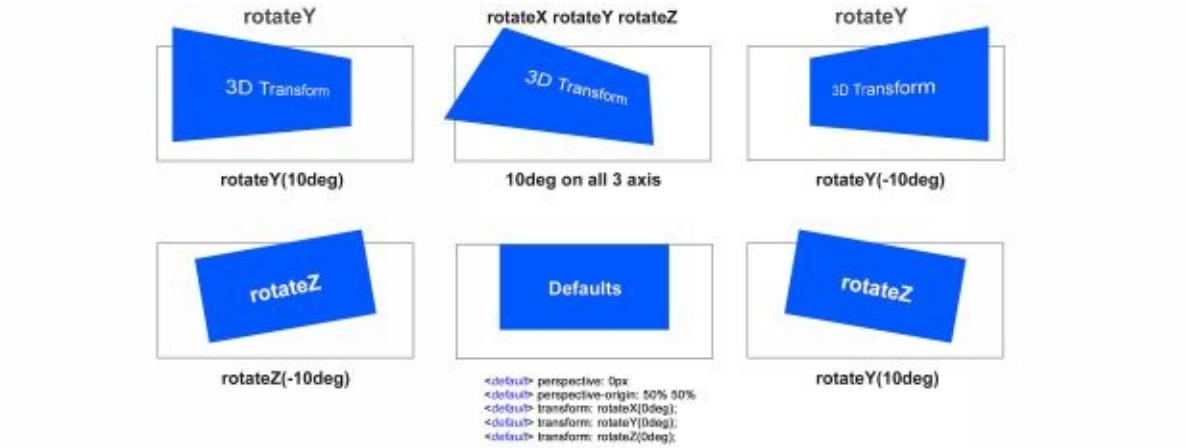



**Figure 146:** Rotating on the Y and Z axis.

# scale

Scaling an element either reduces or increases its relative size on any of the 3 axis.

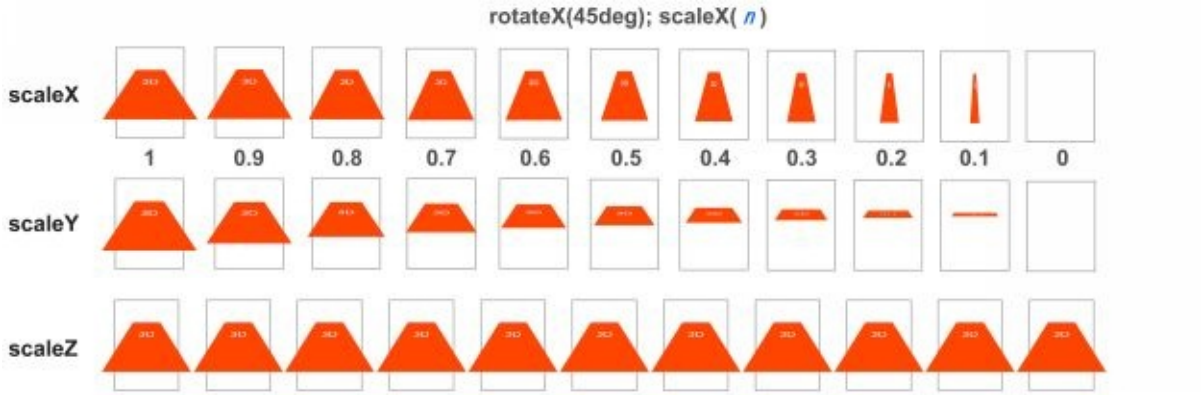



**Figure 147:** Likewise, you can "scale" an element on any of the 3 axis. Scaling on Z axis does not change the element's appearance when no perspective is set.

# translate

You can translate an element in 3 dimensions. This diagram explains what happens when an element is translated on either X, Y or Z axis. Note that the camera is facing down the negative Z axis. So, scaling an element on Z axis up will make it appear "closer" to the view. In other words, its size will increase as it moves closer toward the camera.

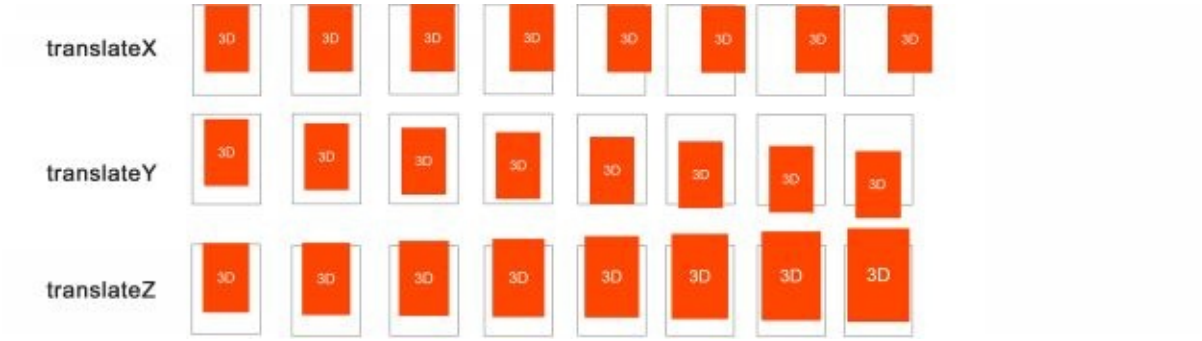



**Figure 148:** Translating an element across 3 axis -- X, Y and Z.

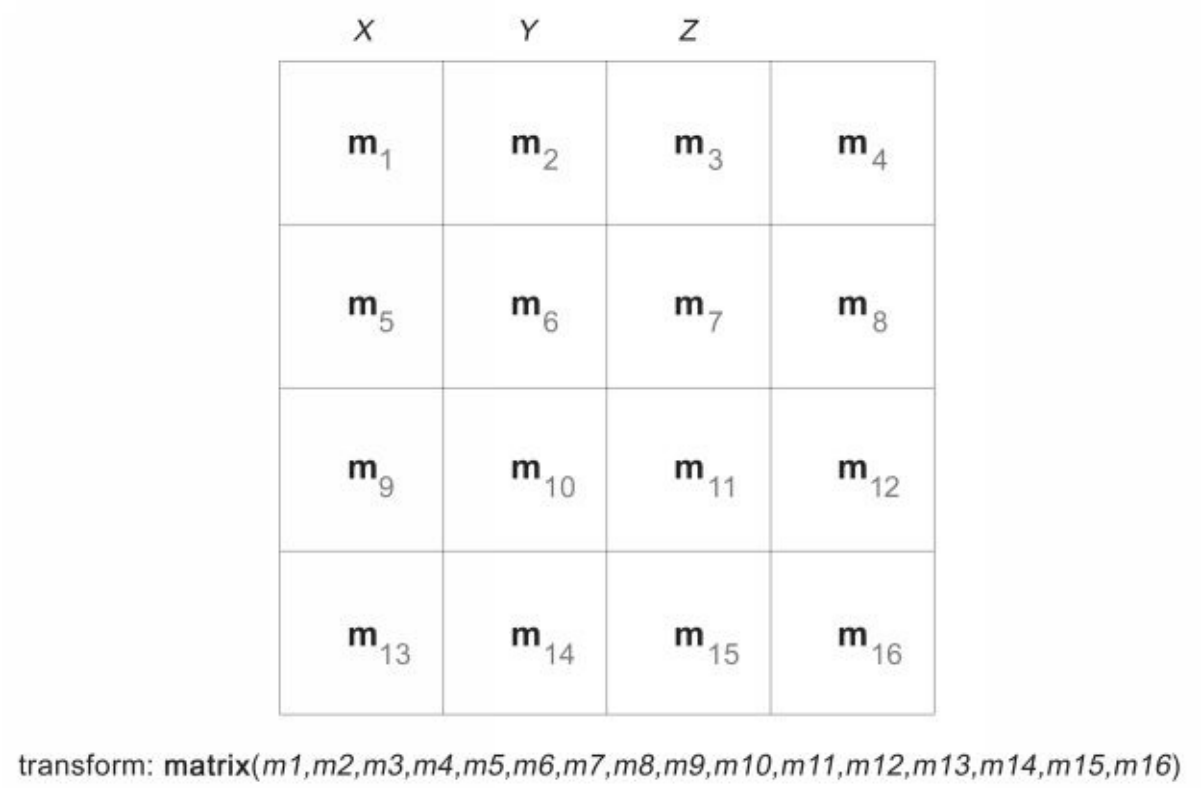



**Figure 149:** CSS provides a "matrix" consisting of a 4 x 4 grid. How 3D matrices work is outside the scope of this book. But basically, they modify the

perspective. They are often used in 3D video games to set up the camera view to look at the main character or "lock in" on a moving object.

# Creating A 3D Cube

Let's take our knowledge of 3D transforms in CSS and construct a 3-dimensional cube from 6 HTML elements.

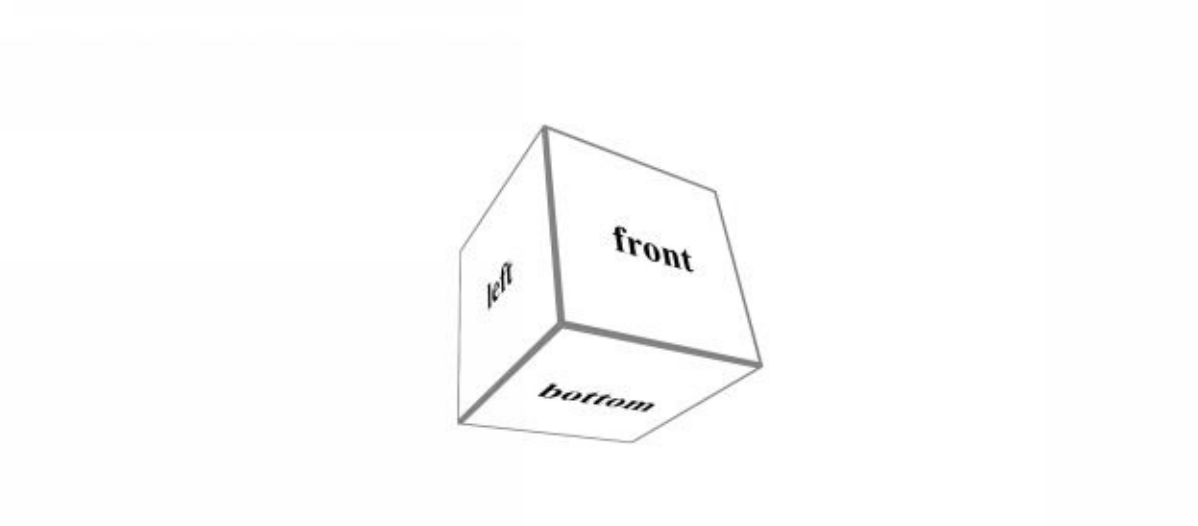



**Figure 150:** A 3D cube made up from 6 HTML elements, each translated by half of its width and rotated 90 degrees in all directions.

```
<div class="view">
      <div class="cube">
      <div class="face front">front</div>
      <div class="face back">back</div>
      <div class="face right">right</div>
      <div class="face left">left</div>
      <div class="face top">top</div>
      <div class="face bottom">bottom</div>
   </div>
</div>
```

```
.view {
  width: 200px;
  height: 200px;
  perspective: 300px;
}
```

**Figure 151:** This is our setup. It's simply 6 HTML elements, each with a unique class and 3D transforms.

Let's build the cube!

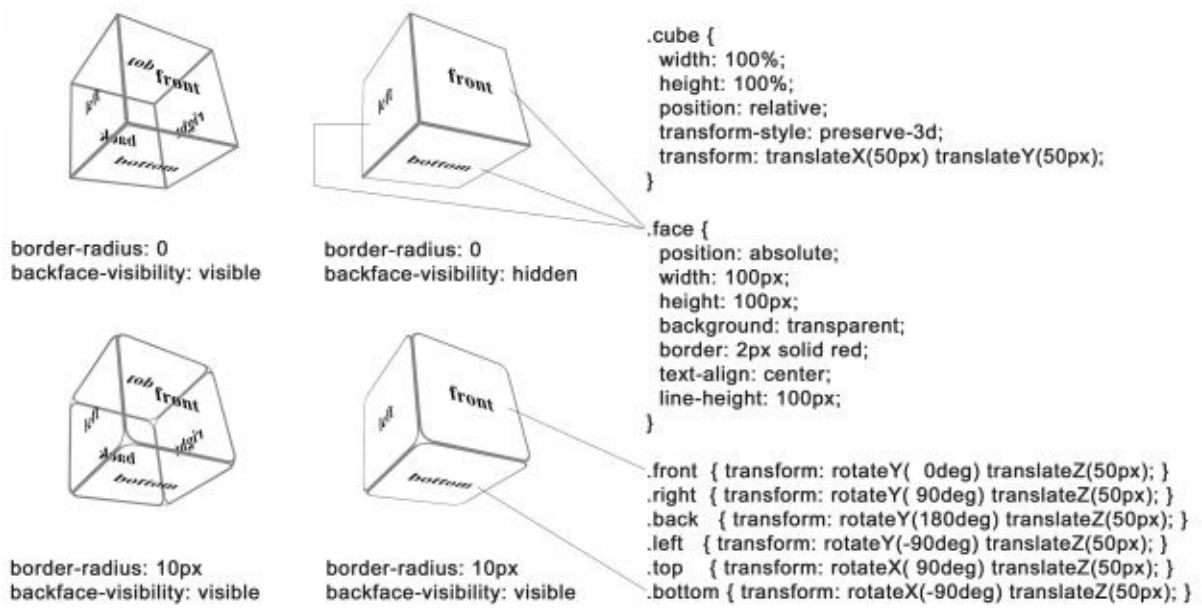



**Figure 152:** By rotating each face around the hypothetical center of the cube, we can construct this 3D object.

Note here **backface-visibility** property was set to hidden, to hide elements that are facing away from the camera. This makes our cube appear solid.

# Flex

Flex is a set of rules for automatically stretching multiple columns and rows of content across parent container.

# display:flex

Unlike many other CSS properties, in Flex you have a main container and items nested within it. Some CSS flex properties are used only on the parent. Others only on the items.

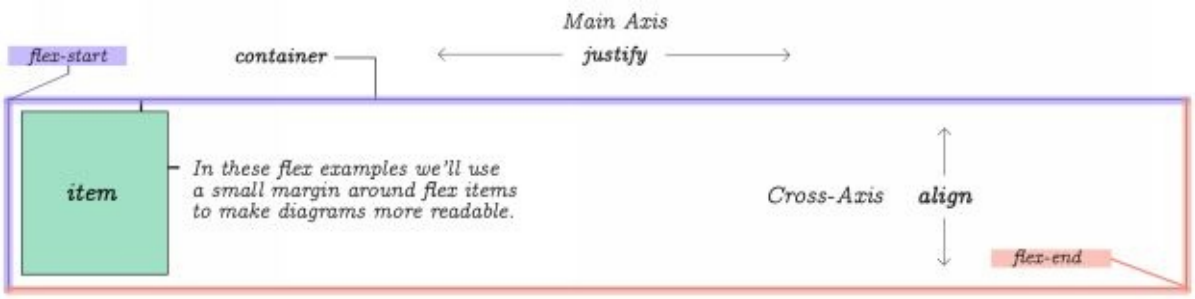



**Figure 153:** You can think of a flex element as a parent container with **display:flex**. Elements placed inside this container are called items. Each container has a **flex-start** and **flex-end** points as shown on this diagram.

# Main-axis and Cross-axis

While the list of items is provided in a linear way, Flex requires you to be mindful of rows and columns. For this reason, it has two coordinate axis. The horizontal axis is referred to as *Main-Axis* and the vertical is the *Cross-Axis*. To control the behavior of content's width and gaps between that stretch horizontally across the Main-Axis you will use *justify* properties. To control vertical behavior of items you will use *align* properties.

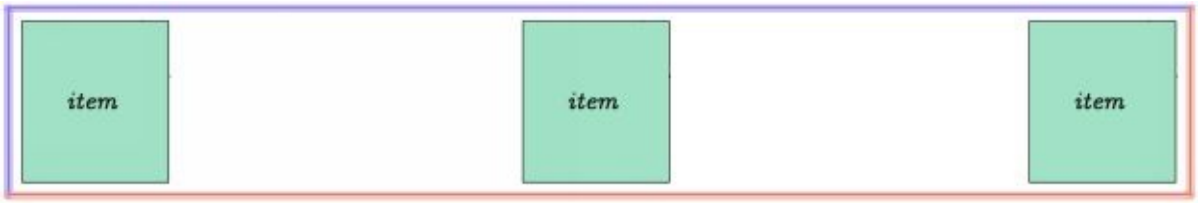



**Figure 154:** Flex items equally distributed on the Main-Axis. We'll take a look at the properties and values to accomplish this in just a moment.

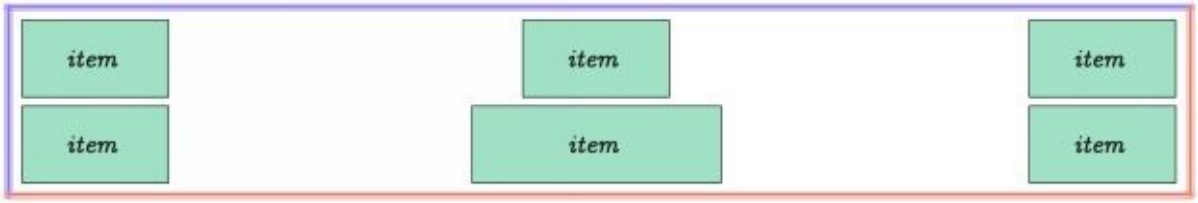



**Figure 155:** You can determine the number of columns.

If you have 3 columns and 6 items, a second row will be automatically created by Flex to accommodate for the remaining items.
If you have more than 6 items listed, even more rows will be created.
How the rows and columns are distributed inside the parent element is determined by CSS Flex properties **flex-direction**, **flex-wrap** and a few others that will be demonstrated throughout the rest of this chapter.

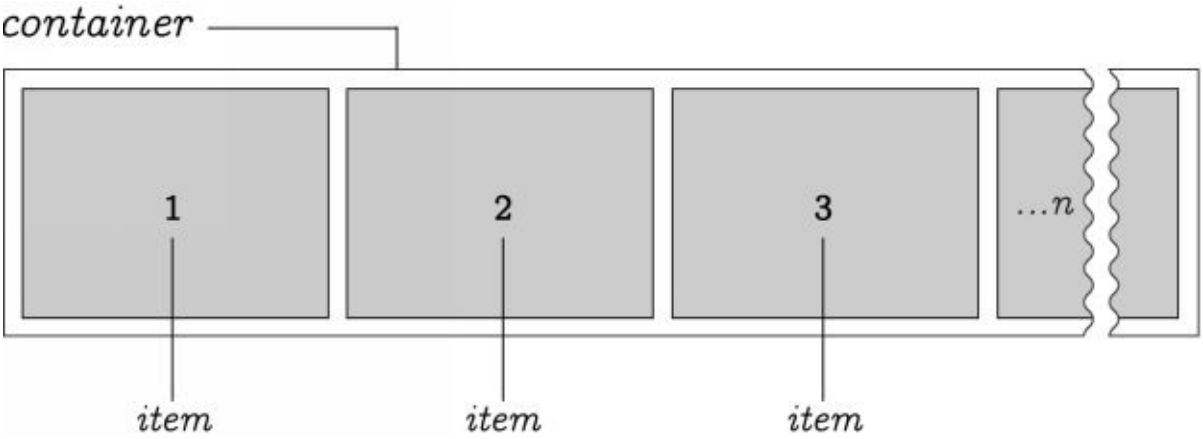



**Figure 156:** Here we have an arbitrary *n*-number of items positioned within a container. By default, items stretch from left to right. However, the origin point can be reversed.

# Direction

It's possible to set direction of the item's flow by reversing it.

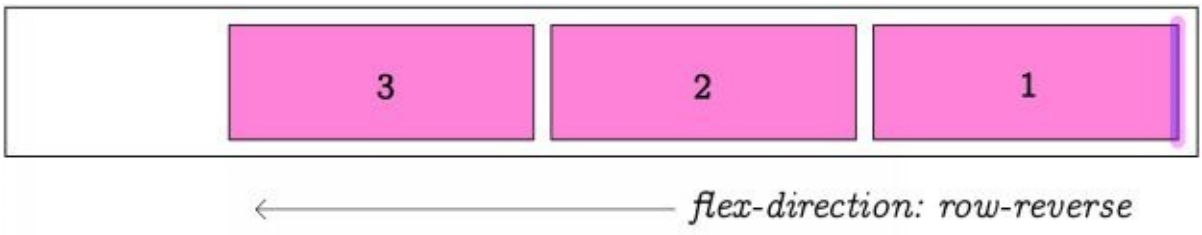



**Figure 157: flex-direction:row-reverse** changes direction of the item list flow. The default is **row**, which means flowing from left to right, as you would expect!

# Wrap

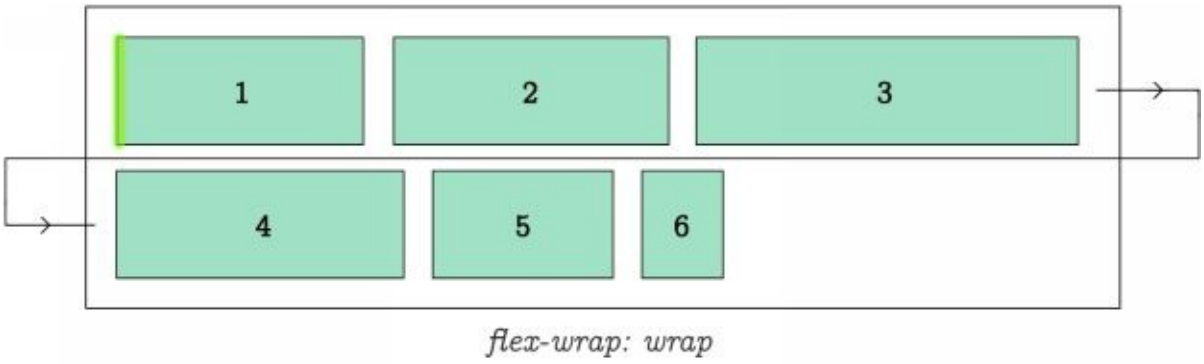



**Figure 158: flex-wrap:wrap** determines how items are wrapped when parent container runs out of space.

# Flow

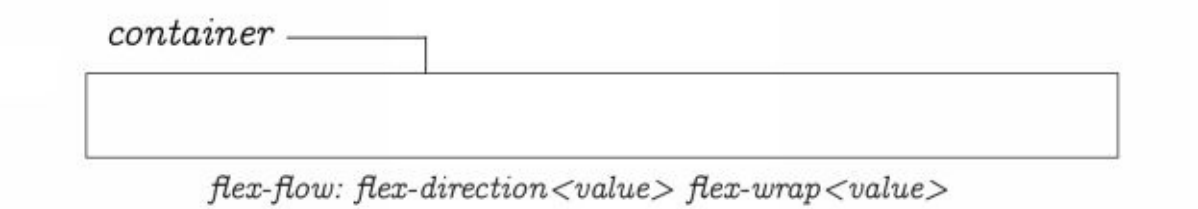



**Figure 159: flex-flow** is a short hand for **flex-direction** and **flex-wrap** allowing you to specify both of them using just one property name.

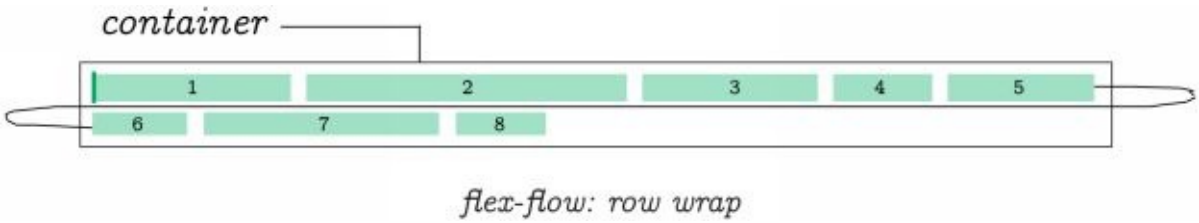



**Figure 160: flex-flow:row wrap** determines **flex-direction** to be **row** and **flex-wrap** to be **wrap**.

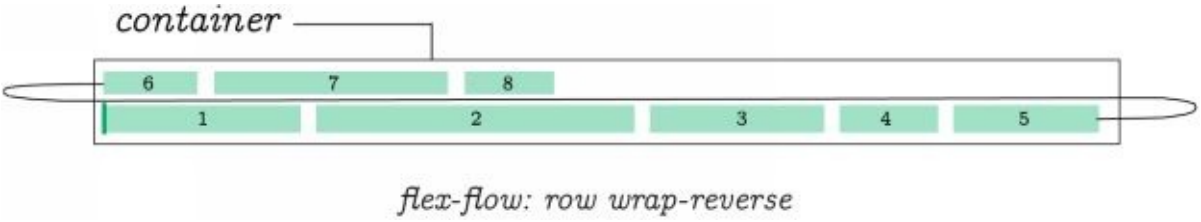



**Figure 161: flex-flow:row wrap-reverse;**

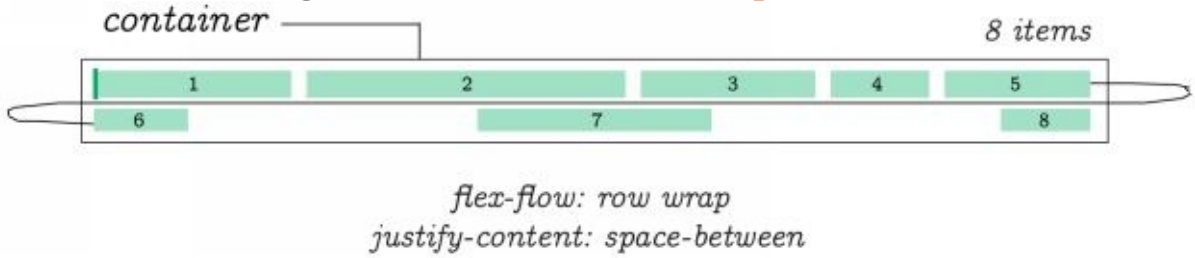



**Figure 162: flex-flow:row wrap; justify-content: space-between;**

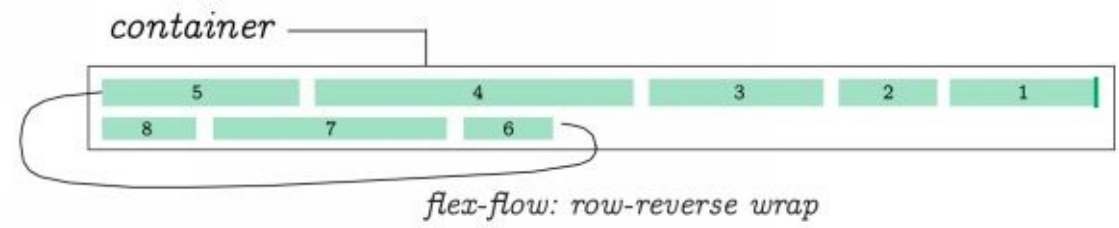



**Figure 163: flex-flow:row-reverse wrap;**

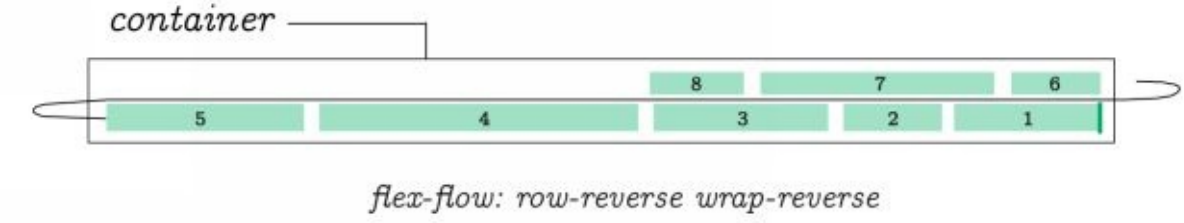



**Figure 164: flex-flow:row-reverse wrap-reverse;**

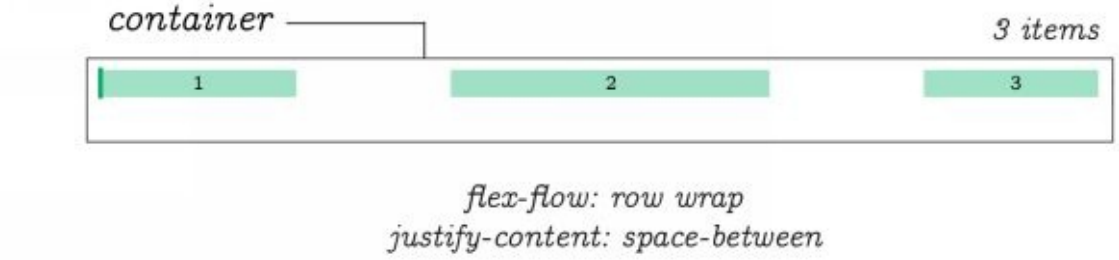



**Figure 165: flex-flow:row wrap; justify-content: space-between;**

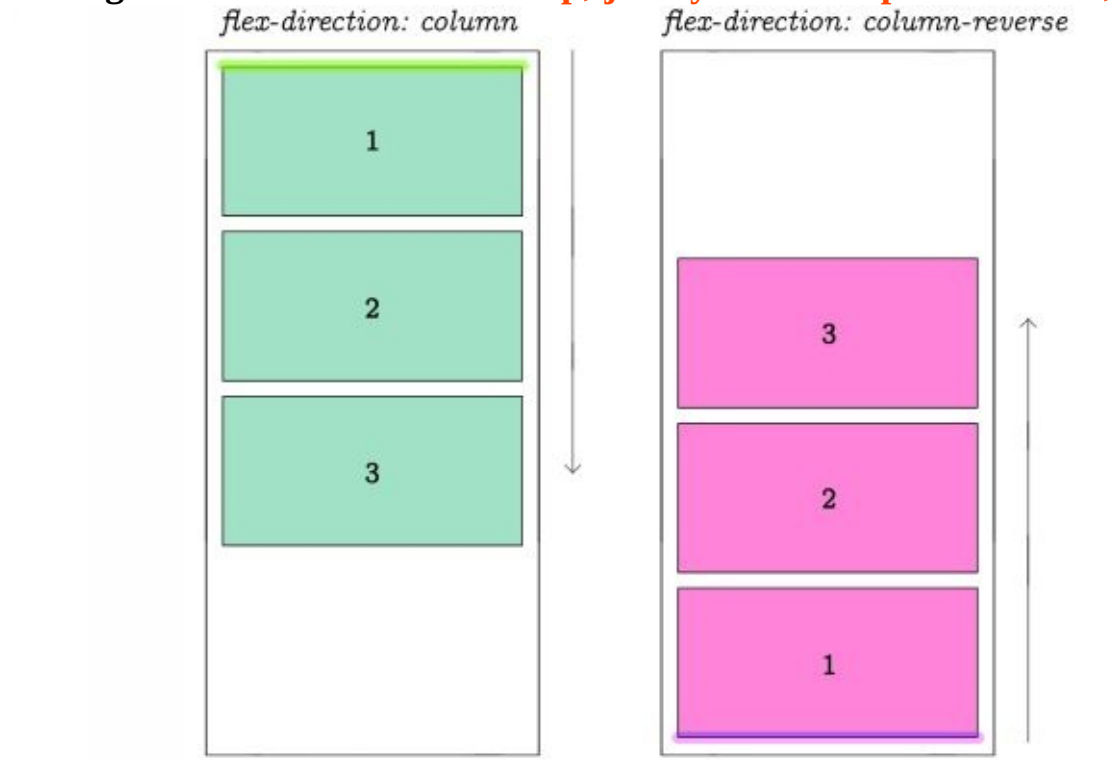



**Figure 166:** The direction can be changed to make the Cross-Axis primary. When we change flex direction to **column**, the flex-flow property behaves in

exactly the same way as in previous examples. Except this time, they follow the vertical direction of a column.

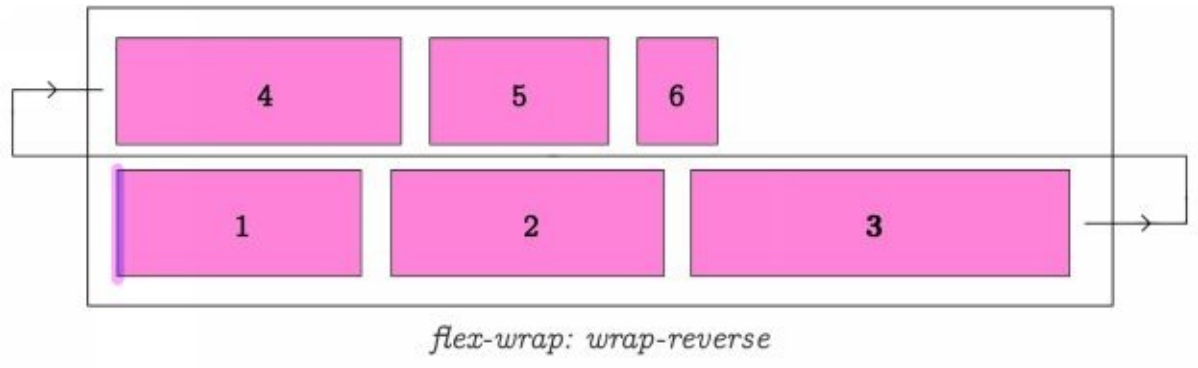



**Figure 167: flex-wrap:wrap-reverse**

# justify-content

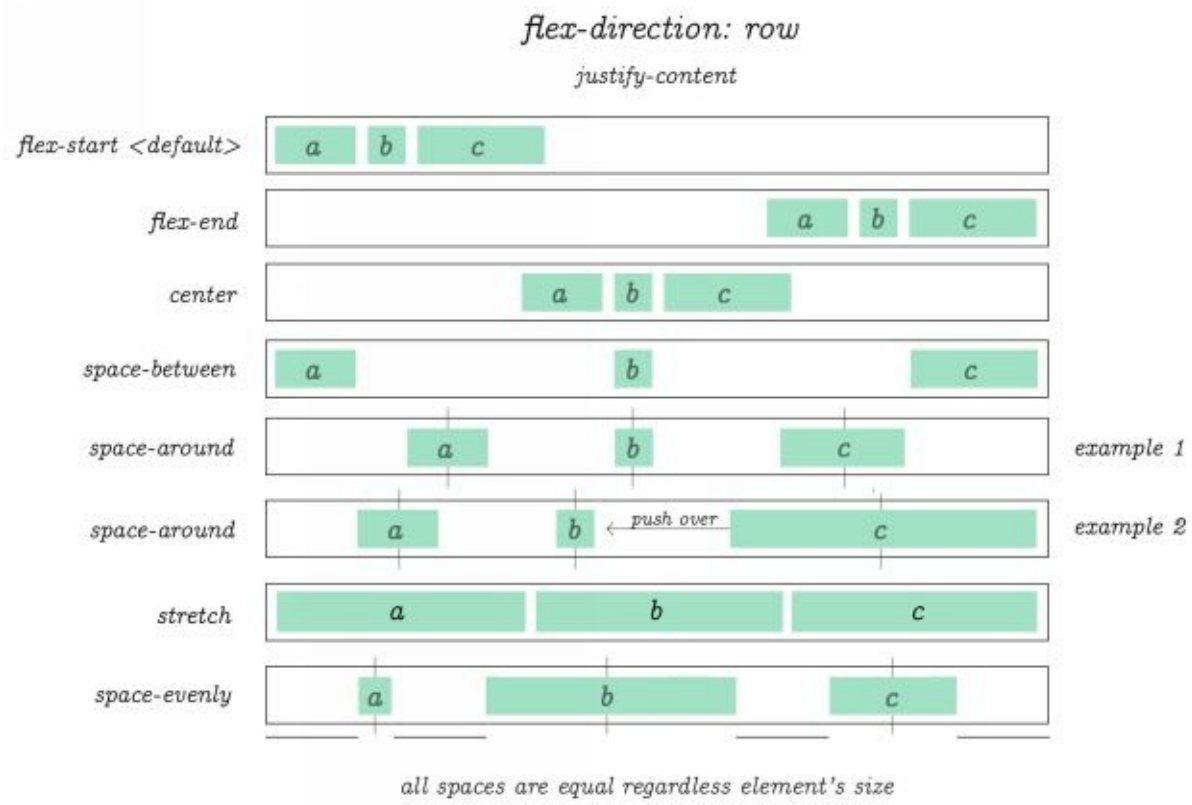



**Figure 168: flex-direction:row; justify-content:** *flex-start* | *flex-end* | *center* | *space-between* | *space-around* | *stretch* | *space-evenly*. In this example we're using only 3 items per row. There is no limit on the number of items you wish to use in flex. These diagrams only demonstrate the behavior of items when one of the listed values is applied to **justify-content** property.

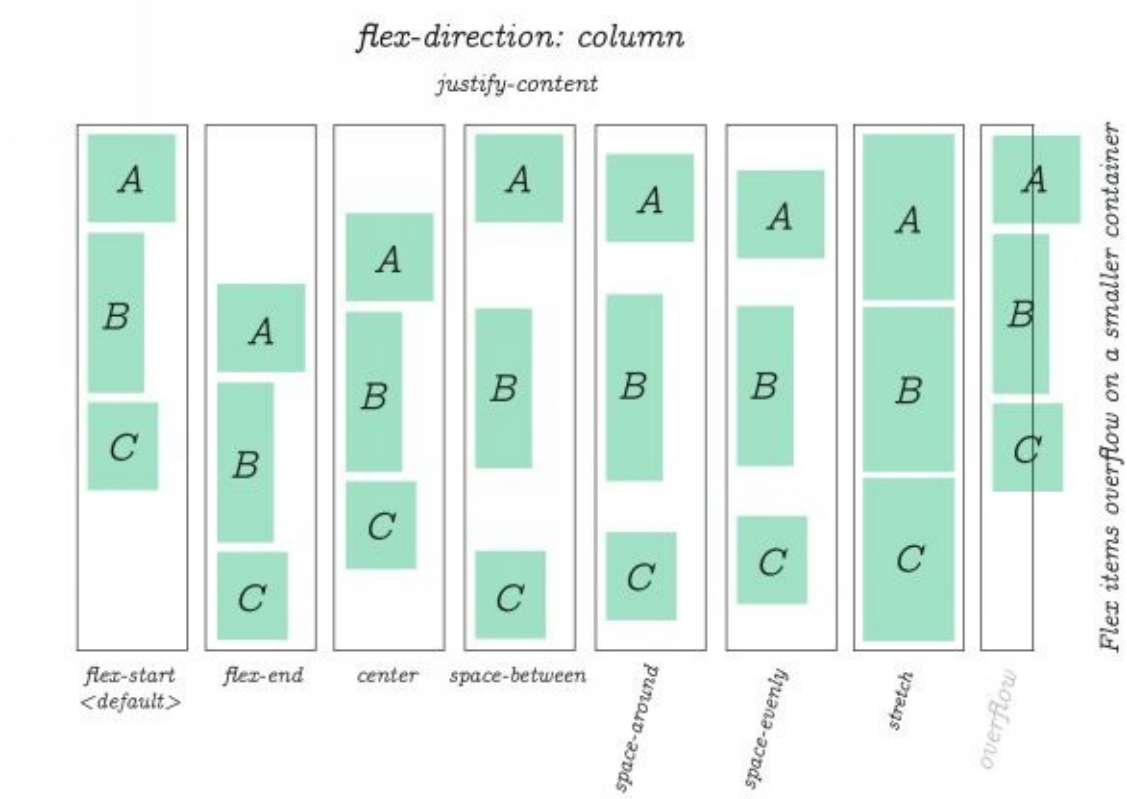



**Figure 169:** The same **justify-content** property is used to align items when **flex-direction** is column.

# Packing flex lines

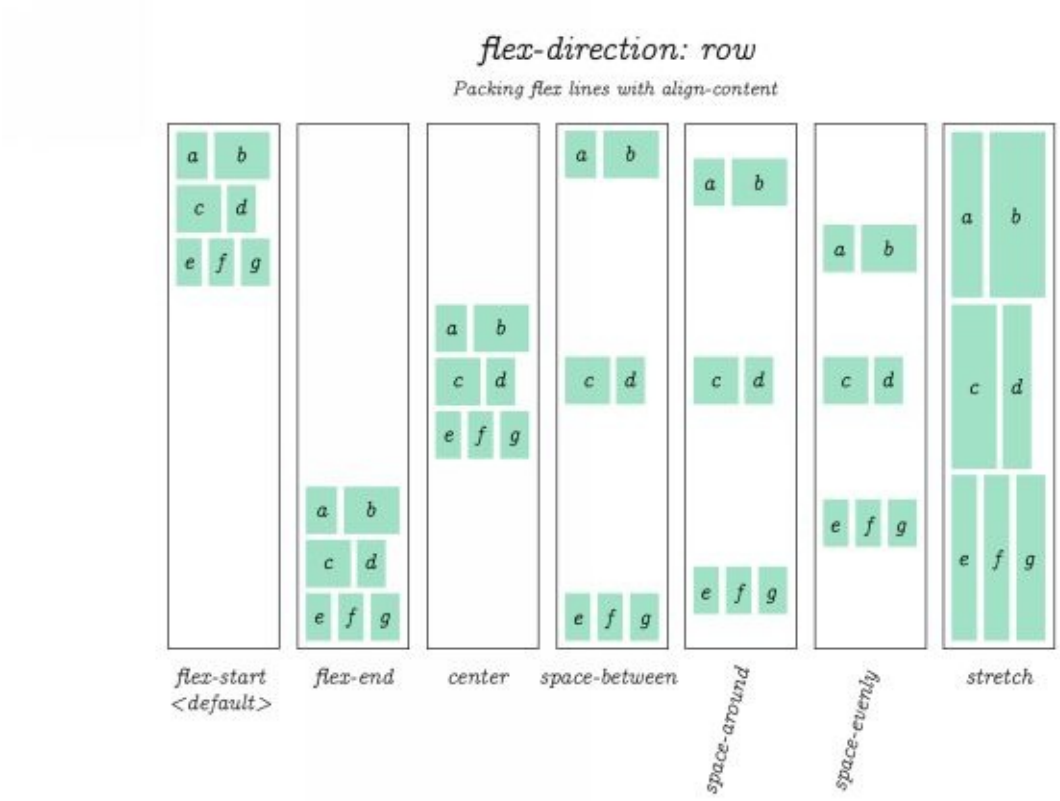



**Figure 170:** Flex specification refers to this as "packing flex lines." Basically, it works just like the examples we've seen on the previous few pages. Except this time, note that the spacing is between whole sets of items. This is useful when you want to crate gaps around a batch of several items.

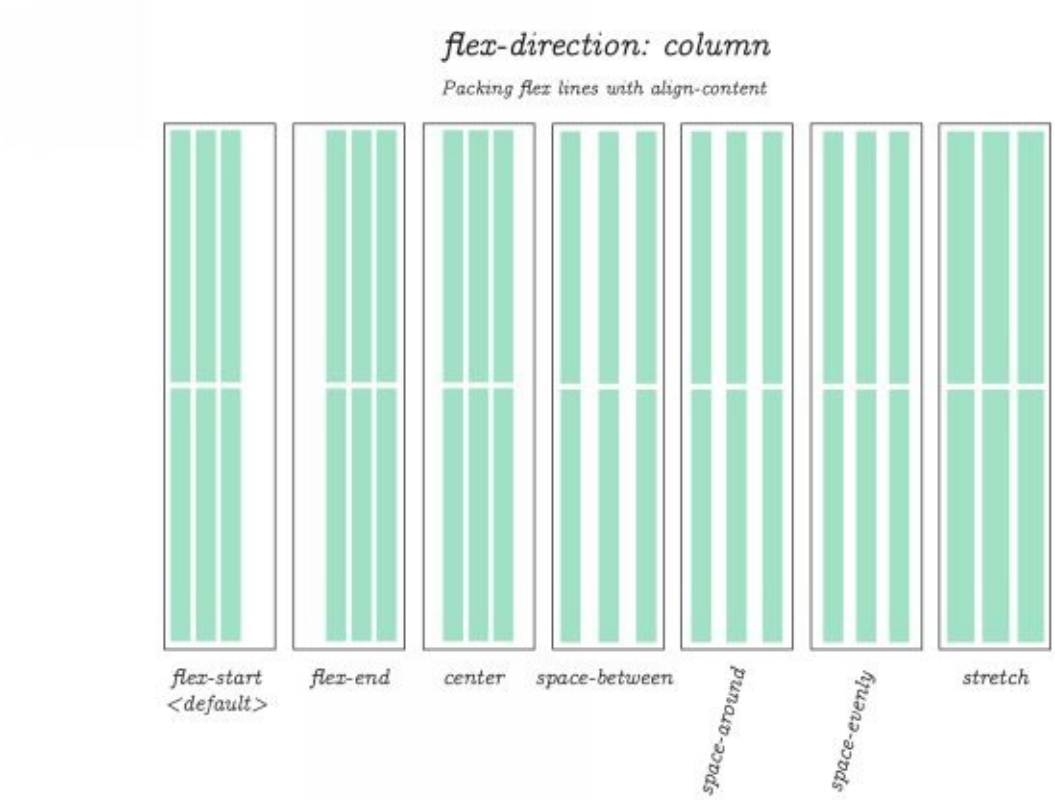



**Figure 171:** Packing flex lines (continued.) But now with **flex-direction** set to **column**.

# align-items

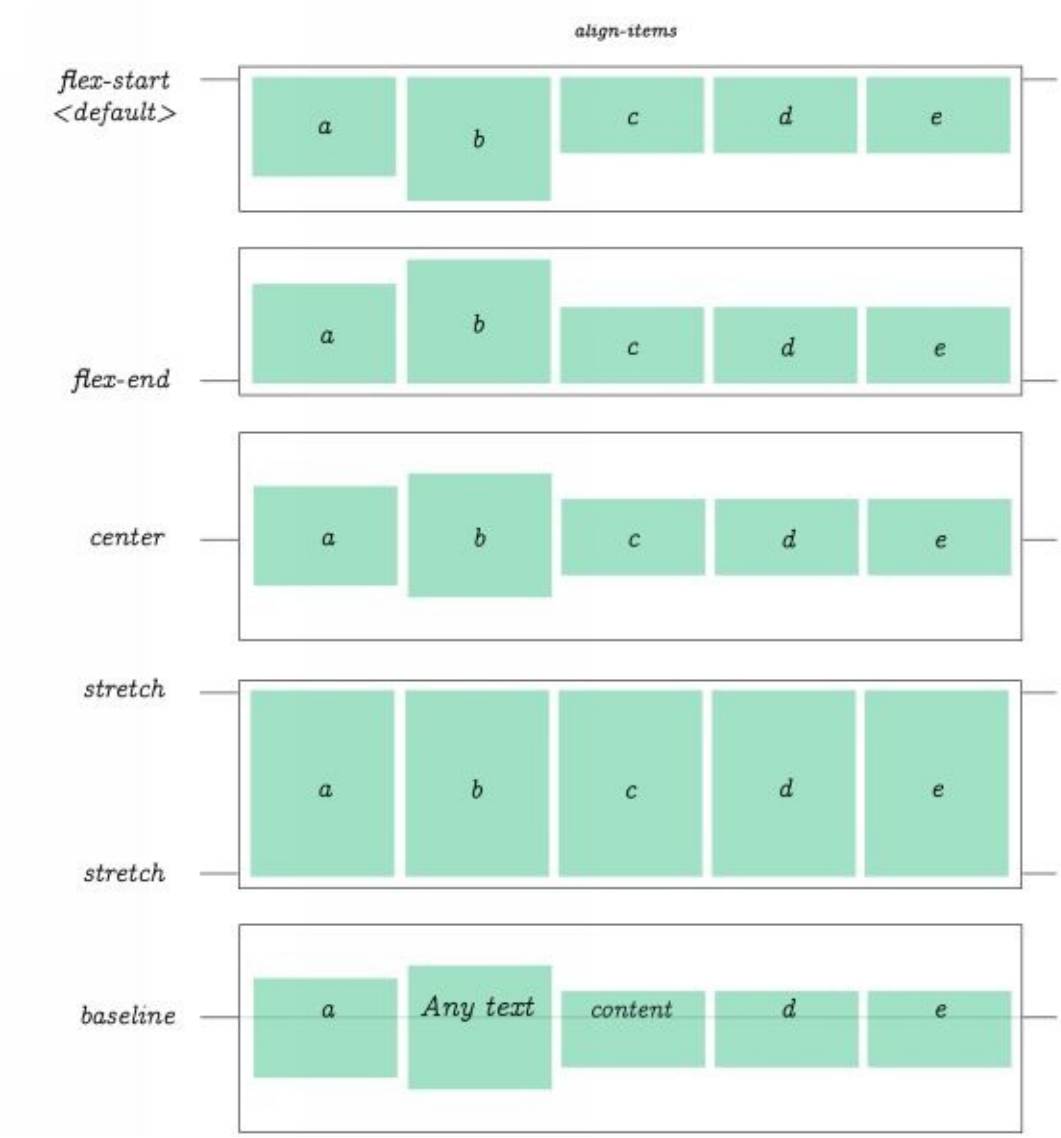



**Figure 172:** **align-items** controls the align of items horizontally, relative to the parent container.

# flex-basis

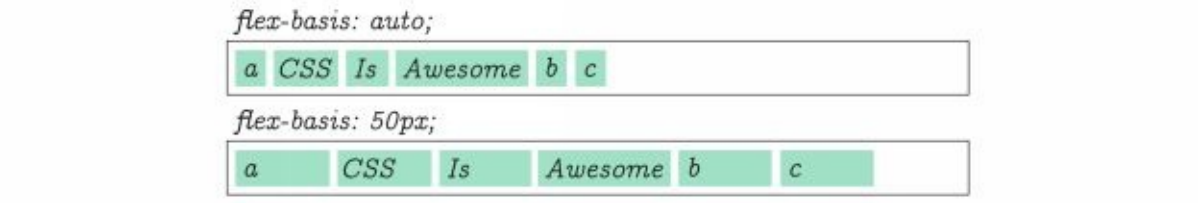



**Figure 173:** **flex-basis** works similar to another CSS property: **min-width** outside of flex. It will expand item's size based on inner content. If not, the default *basis* value will be used.

# flex-grow

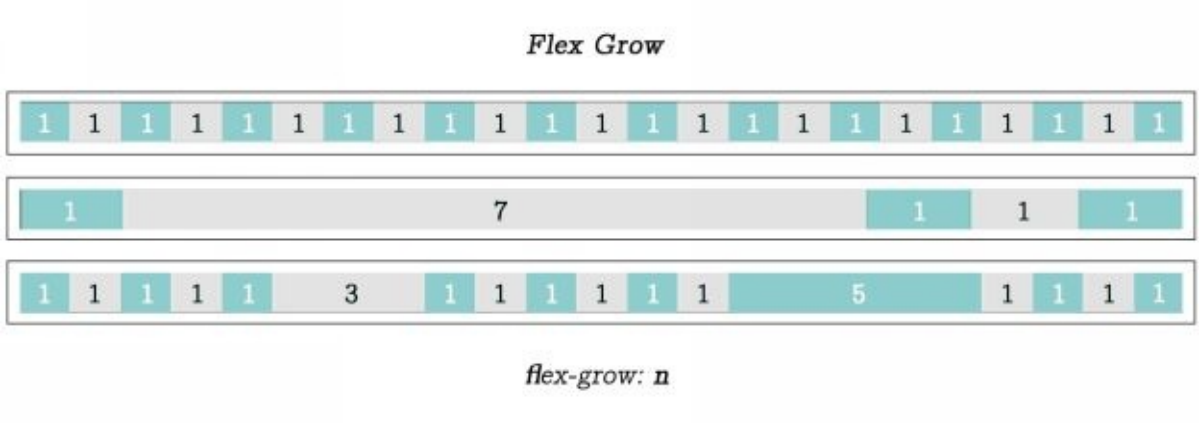



**Figure 174:** **flex-grow**, when applied to an item will scale it relative to the sum of the size of all other items on the same row, which are automatically adjusted according the the value that was specified. In each example here the item's flex-grow value was set to 1, 7 and (3 and 5) in the last example.

# flex-shrink

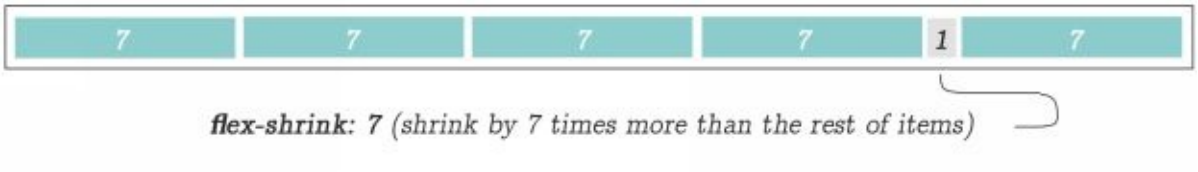



**Figure 175:** **flex-shrink** is the opposite of **flex-grow**. In this example value of 7 was used to "shrink" the selected item in the amount of time equal to 1/7th times the size of its surrounding items -- which it will be also automatically adjusted.

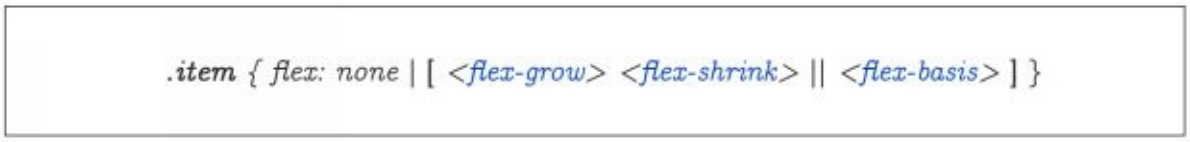



**Figure 176:** When dealing with individual items, you can use the property **flex** as a shortcut for **flex-grow**, **flex-shrink** and **flex-basis** using only one property name.

# order

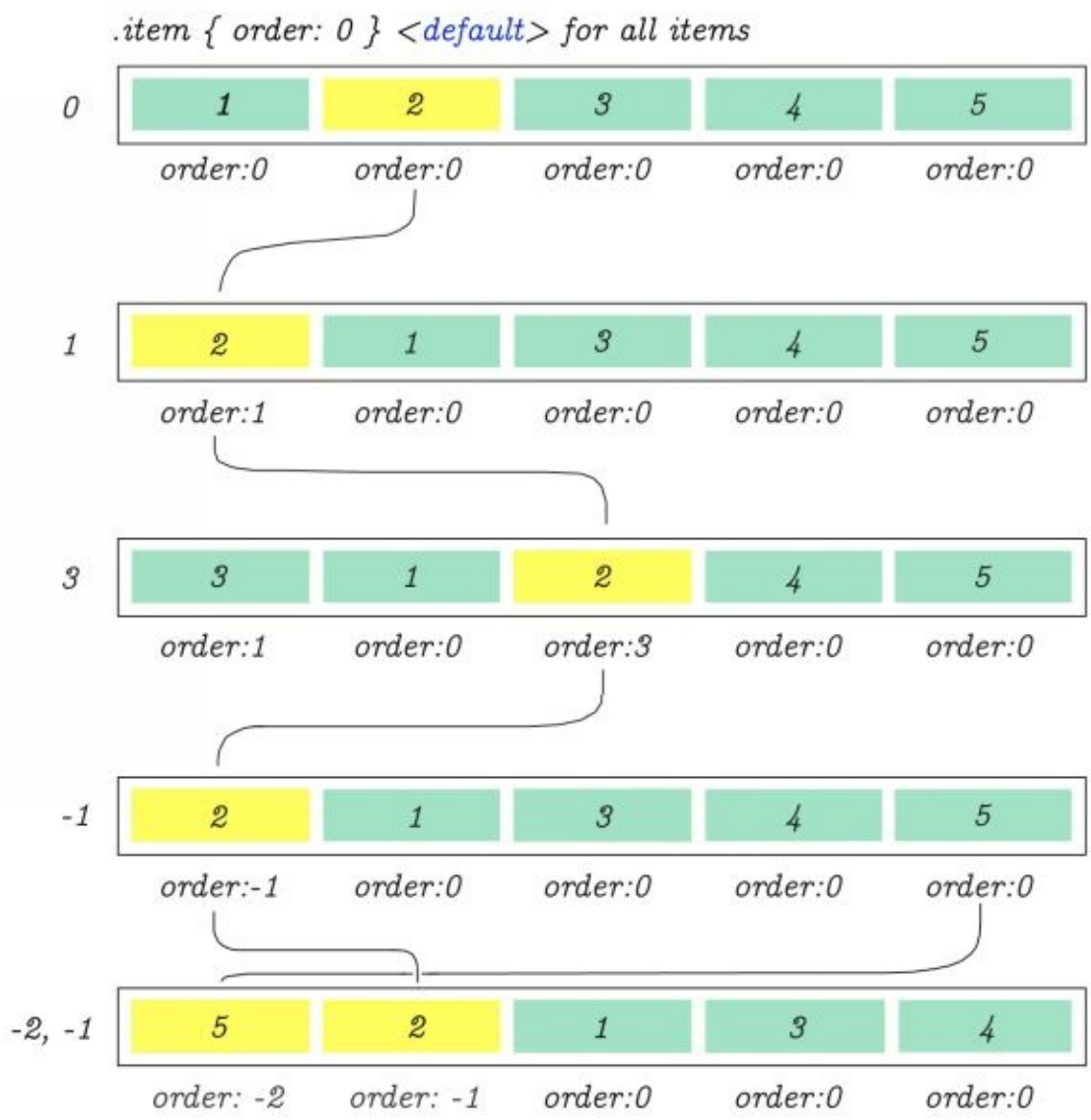



**Figure 177:** Using **order** property it's possible to re-arrange the natural order of items.

# justify-items

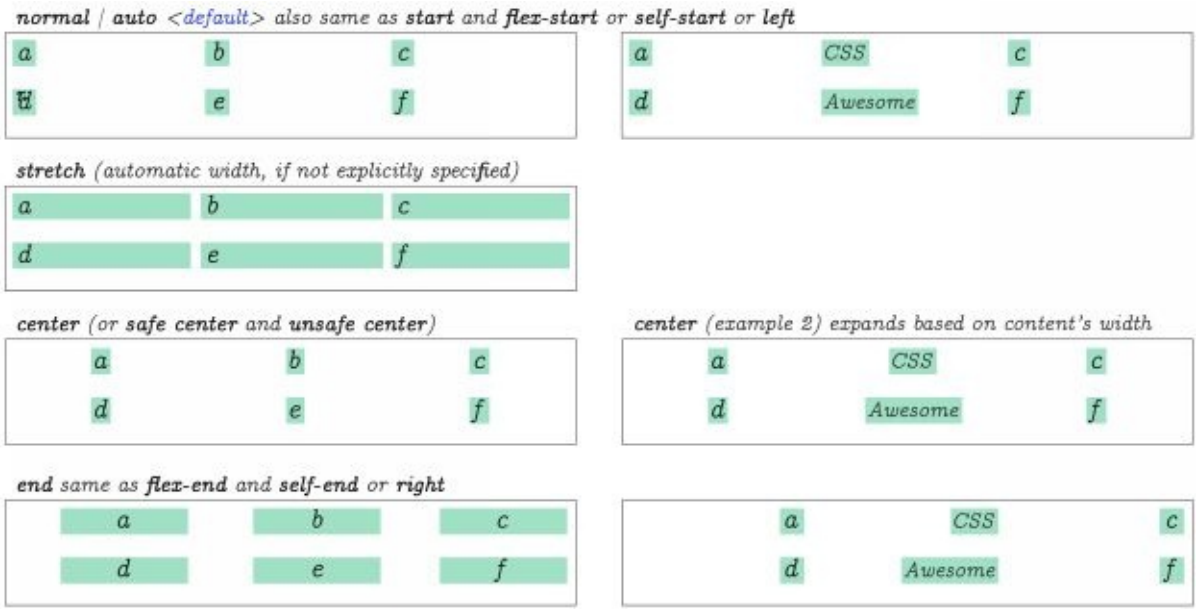



**Figure 178: justify-items** is similar to Flex's **justify-content** but for CSS grid, which is our next subject.

# CSS Grid

During my job interview at a Texas software company back in 2017 the team lead has introduced me to the idea that computer scientists (and probably scientists in general) make progress by "filling in the gaps."

***That idea stuck with me.***

I don't know but maybe like me, you found yourself trying to fill in the gaps learning CSS grid. Just as it ought to happen, every time some new tech emerges on the unsuspecting JavaScript crowd.

This tutorial was created by applying this ideology in preparation for working on the Visual Dictionary - the book you are reading right now - my postmortem to CSS. This is when I decided to take all of the existing 413 CSS properties and visualize them by making diagrams.

While primarily I consider myself a JavaScript programmer I'd like to think that I also lean a bit toward the graphic design mindset as well. Perhaps I took the idea of gaps out of context when I applied it to CSS grid in this tutorial.

***But bear with me.***

As professional book designers may already know, the key to *CSS grid* lies in grasping not the visible parts of layout design but ones that are not.

***Let me explain.***

Book designers care about margins -- essentially an invisible element of book design. It may not seem like much but remove margins -- the element that the reader is least aware of -- and the whole reading experience turns awkward. Readers notice lack of margins only when they are absent. Therefore?--?as a designer (of anything) it is imperative to pay equal attention to the invisible elements of design.

We can always just plug in the content into the layout. Could it be then... that when we design with *CSS grid*, all we're doing here to create beautiful layouts is working with an advanced version of book margins? Could be.

***The gaps.***

Of course CSS grid goes way beyond designing margins in books but the principle of the so-called invisible design remains the same. Things we cannot see are important. In CSS grid this concept is thought of as gaps.

A CSS grid after all is just like taking book margins to the next level. Sort of.

# Creating Your First CSS Grid

Much like Flex, CSS grid properties are never applied to just one element. The grid works as a single unit, consisting of a parent element and items contained within it.

First... we will need a container and some items.

A CSS grid flow can go in either direction. But by default it's set to row.

This means that if all other defaults are untouched your items will automatically form a single row where each item inherits its width from the grid's container element:

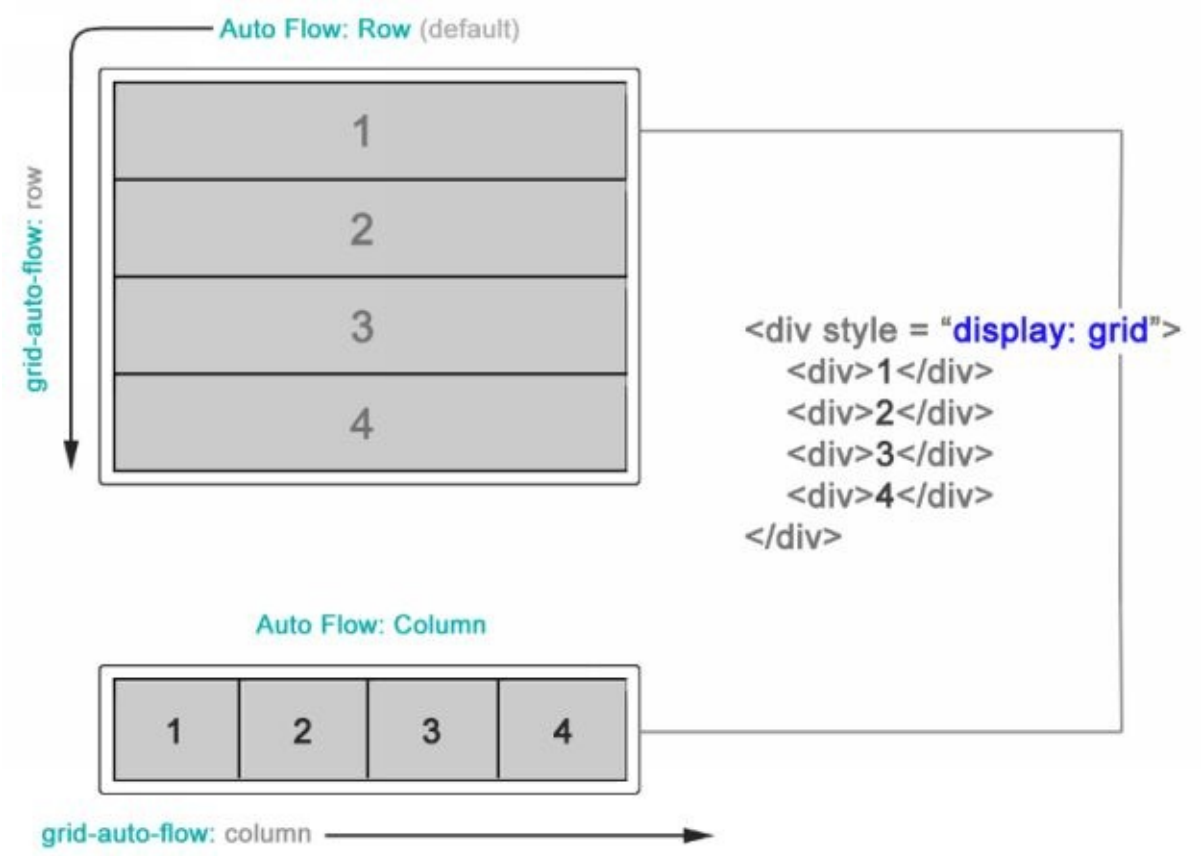



**Figure 179:** Just like Flex, CSS Grid can align items in one of the either **direction:rows** or columns specified by **grid-auto-flow** property.

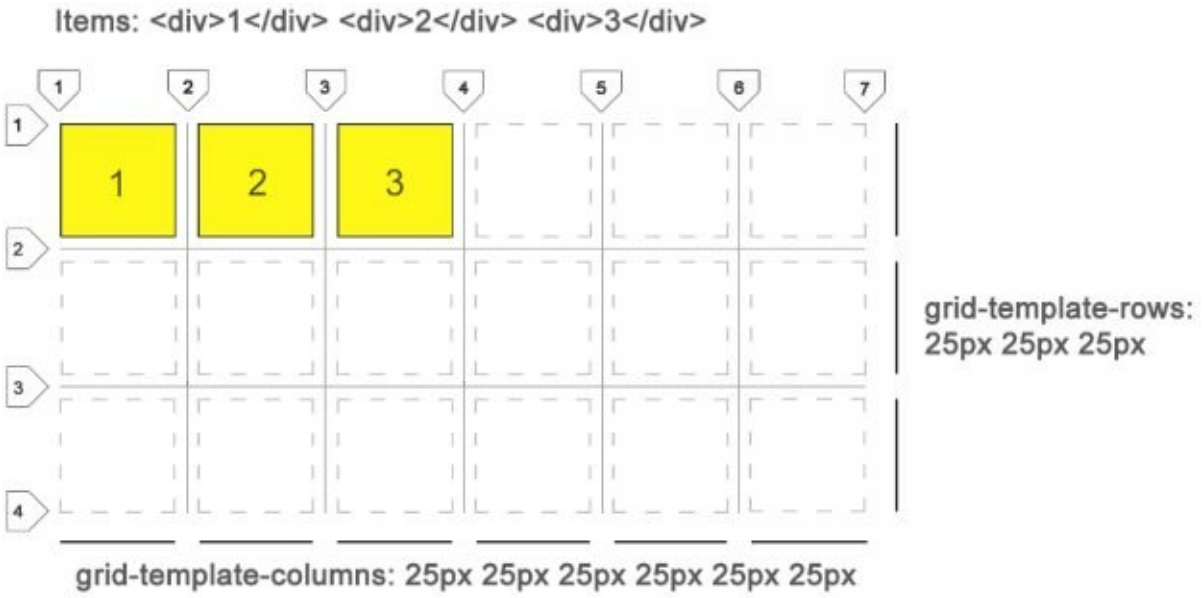



**Figure 180:** The CSS grid creates a virtual grid environment, where the items don't have to fill up the entire area of the grid. But the more items you add, the more placeholders will become available to populate the grid. CSS grid just makes this automatic process a bit more graceful.

The CSS grid uses column and row templates to choose how many items will be used in your grid down and across. You can specify their number by using CSS properties **grid-template-rows** and **grid-template-columns** respectively as shown on the diagram above. This is the basic construct of CSS grid.

One thing you will notice about CSS grid right away is the definition of gaps. This is different from what we've seen in any other CSS property before. Gaps are defined numerically starting from the upper left corner of the element. There are **columns + 1** gaps between columns and similarly, **rows + 1** gaps between rows. As you would expect.

CSS grid does not have a default *padding*, *border*, or *margin* and all of its items are assumed to be **content-box** by default. Meaning, content is padded on the inside of the item, not outside like in all other common blocking elements. That's one of the best things about CSS grid in general. Finally we have a new layout tool that treats its box model as **content-box** by default.

CSS grid's gap size can be set individually per row or column when you use properties **grid-row-gap** and **grid-column-gap**. Or, for convenience... together by just one property **grid-gap** as a shorthand.

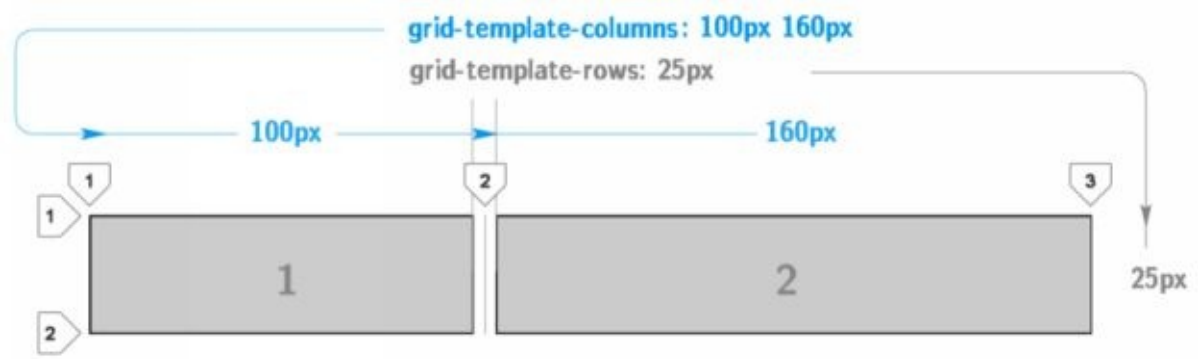



**Figure 181:** Here I created a miniature grid consisting of one row and 2 columns. Note the wedges here specify horizontal and vertical gaps between the items. In all future diagrams from now on, these wedges will be used to illustrate gaps. Gaps are a bit different than borders or margins, in that the outside of the grid area is not padded by them.

To start getting to know CSS grid, let's take a look at this first simple example (Figure 3 from above). Here we have **grid-template-columns** and **grid-template-rows** CSS properties defining basic CSS grid layout. These properties can take multiple values (which you should separate by space) that in turn become columns and rows. Here we used these properties to define a minimalist CSS grid composed of two columns (100px 160px) and one row (25px.) In addition, the gaps on the outside border of the grid container do not add extra padding even when gap size is defined. Therefore they should be thought of as defined right on the edge. The gaps in between columns and rows - on the other hand - are the only ones that are affected by gap size.

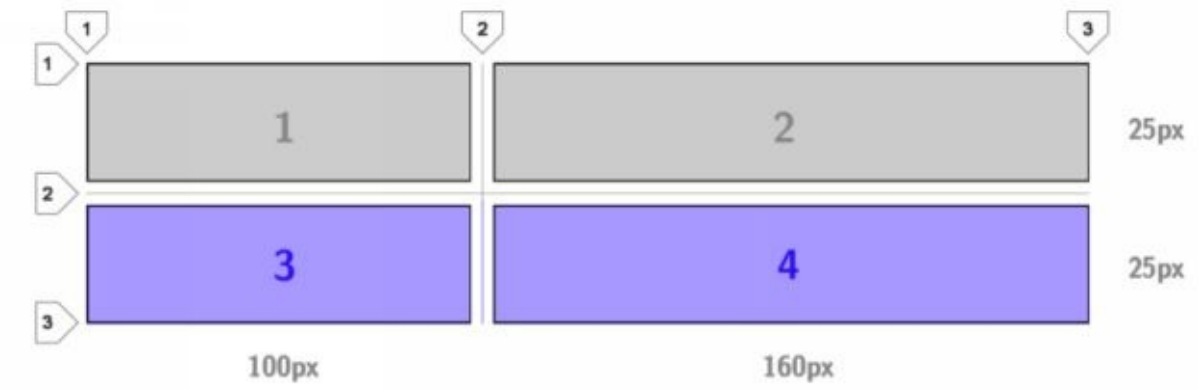



**Figure 182:** Adding more items into a CSS grid whose row template does not provide enough room to place them will *automatically* extend the CSS grid to open up more space. Here items 3 and 4 were added to previous example. But the **grid-template-columns** and **grid-template-rows properties** provide a

template only for 2 items maximum.

# Implicit Rows and Columns

CSS grid then adds them into *implicit* placeholders that it creates automatically, even if they were not specified as part of the grid template.
**Implicit** (I also like to call them *automatic*) placeholders inherit their width and height from the existing template.
They simply extend the grid area when necessary. Usually, when the number of items is unknown. For example, when a callback returns from talking to a database grabbing a number of images from a product profile.

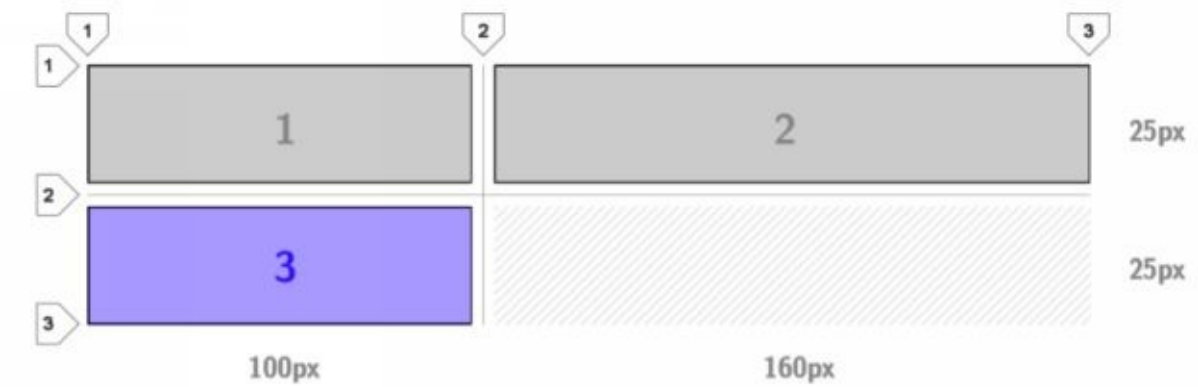



**Figure 183:** In this example we have an implicitly added placeholder for item 3. But because there is no item 4, the last placeholder is not occupied, leaving the grid unevenly balanced.

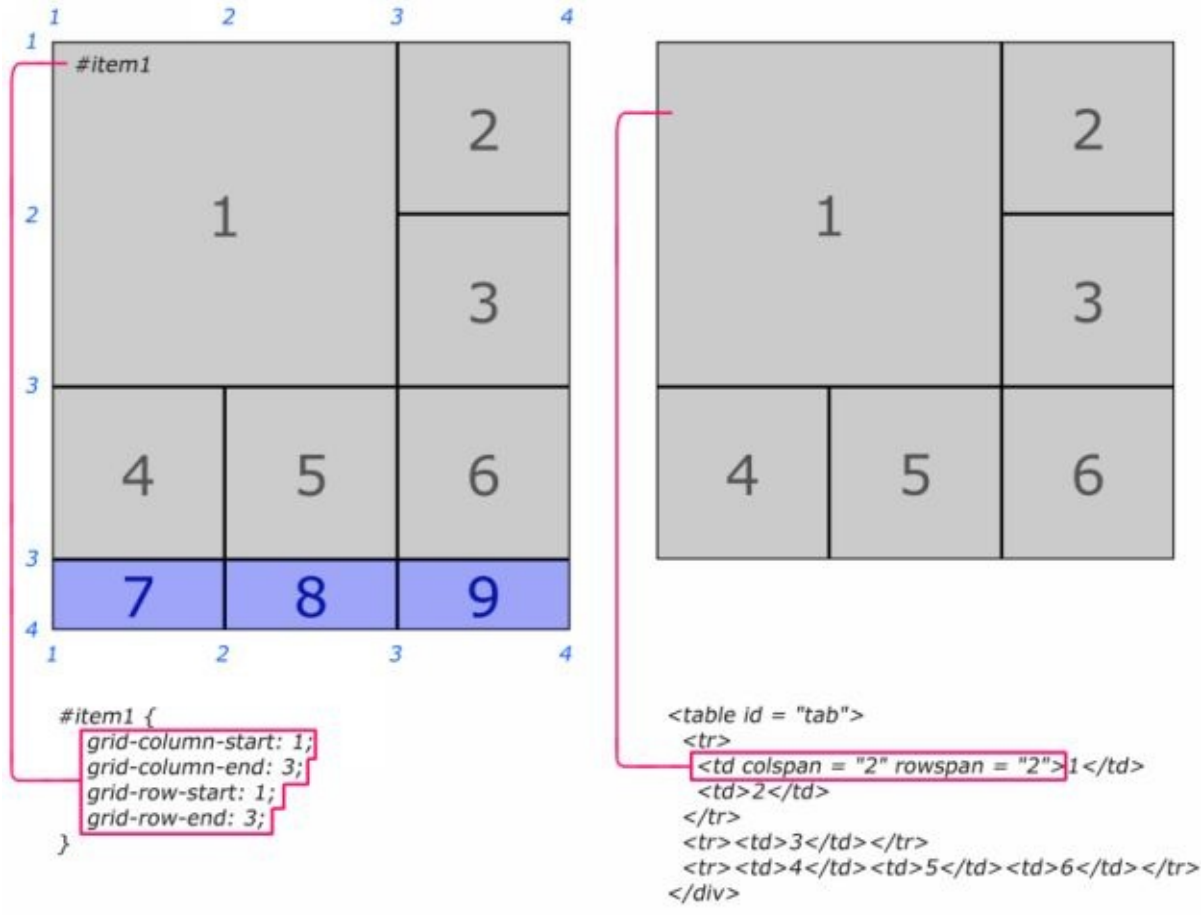



**Figure 184:** CSS grid should never be compared or used in the same way as a table. But it's interesting to note that a CSS grid inherits some design from the HTML table. In fact, the similarities are incredible upon a closer analysis. On the left hand side you are seeing a grid layout. Here **grid-column-start**, **grid-column-end**, **grid-row-start** and **grid-row-end** provide the same function of table's **colspan** and **rowspan**. The difference is that CSS grid uses the gap space in between to determine the span areas. Later you will also see that there is a shortcut for this. Note that here items 7, 8 and 9 were added implicitly, because the span occupied by the item 1 on the grid has pushed 3 items out of the original grid template layout. A table would never do this.

# grid-auto-rows

The **grid-auto-rows** property tells CSS grid to use a specific height for *automatic* (implicitly created) rows. Yes, they can be set to a different value! Instead of inheriting from **grid-template-rows** we can tell CSS grid to use a specific height for all implicit rows that fall outside of your default definitions.

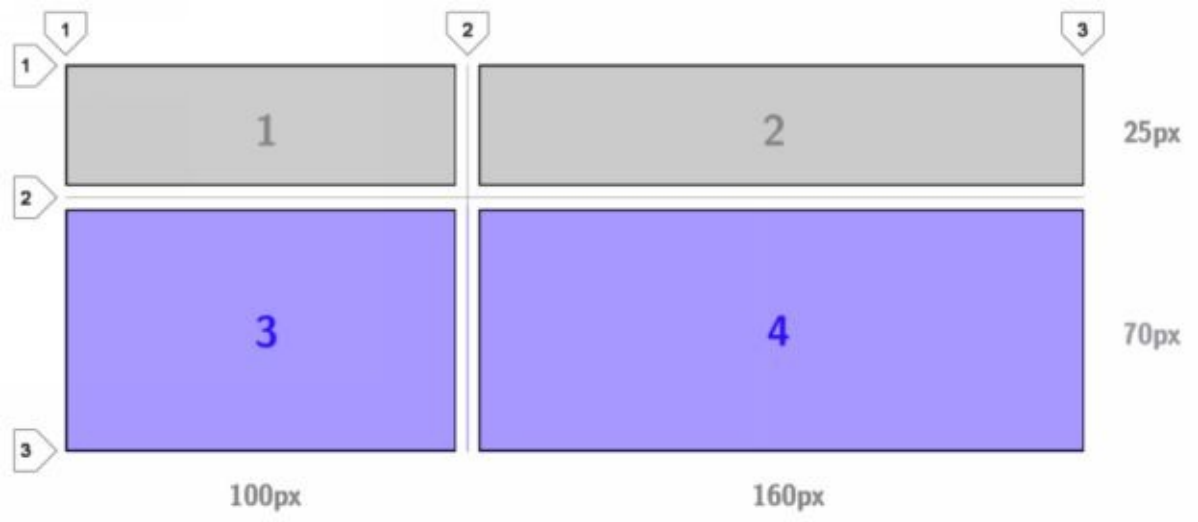



**Figure 185:** Implicit row height is determined by **grid-auto-rows**.

Bear in mind - of course - you can still set all of the values explicitly yourself, as shown in the following example:

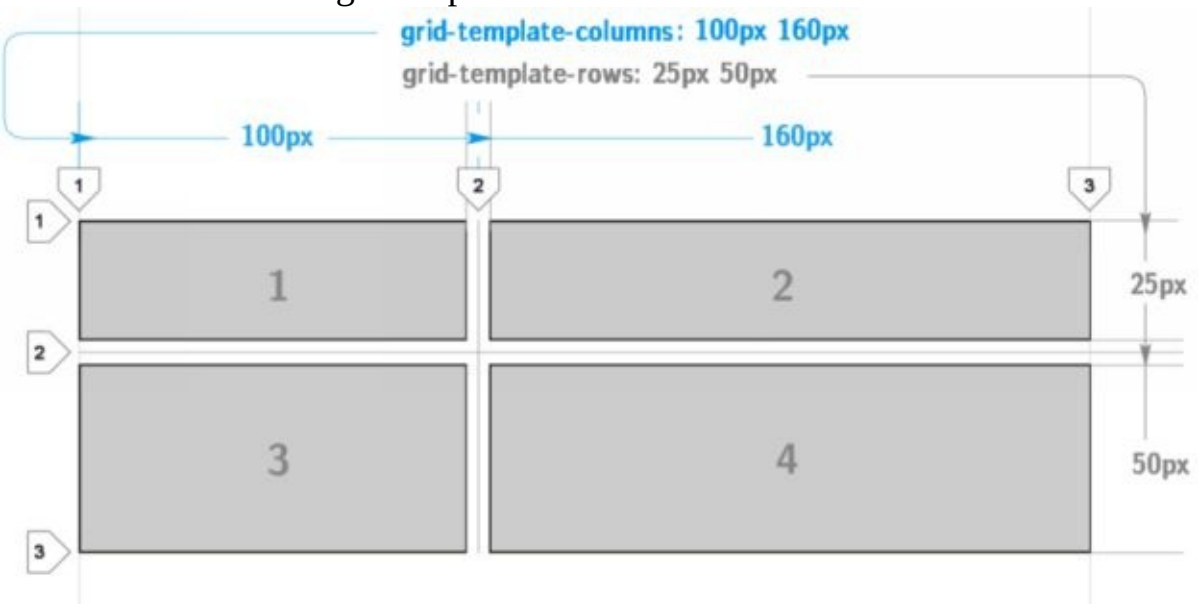



**Figure 186:** Explicitly specifying dimensions of all rows and columns.

In a way, CSS grid's **grid-auto-flow:column** invites *Flex-like* functionality:


**Figure 187:** You can make your CSS grid behave similar to Flex by overwriting its **grid-auto-flow** property's default value of row to column. Note that here in this example we also used **grid-auto-columns: 25px**to determine the width of consecutive columns. This works in the same way as **grid-auto-rows** in one of the previous examples except this time the items are stretched horizontally.

# Automatic Column Cell Width

CSS grid is excellent for creating traditional website layouts with two smaller columns on each side. There is an easy way to do this. Simply provide auto as a value to one of the widths in your **grid-template-column** property:

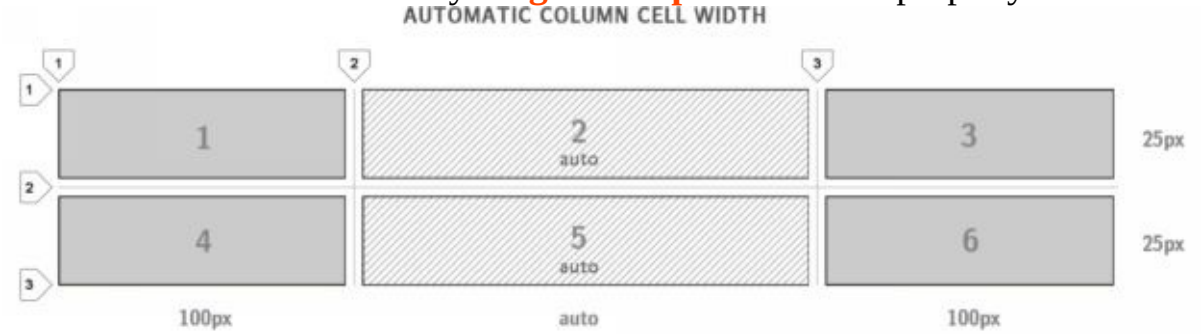



**Figure 188:** This is what happens with **grid-template-columns: 100px auto 100px** Your grid will span across the entire width of the container or the browser.

As you can see already CSS grid offers a wide variety of properties to help you get creative with your website or application layout! I really like where this is going so far.

# Gaps

Look, it's those gaps again.
We already talked about the gaps. Mostly just the fact that they *cover the space between columns and rows*. But we haven't talked about actually changing them. The set of diagrams that follows will provide visual clues as to how gaps modify the appearance of your CSS grid.

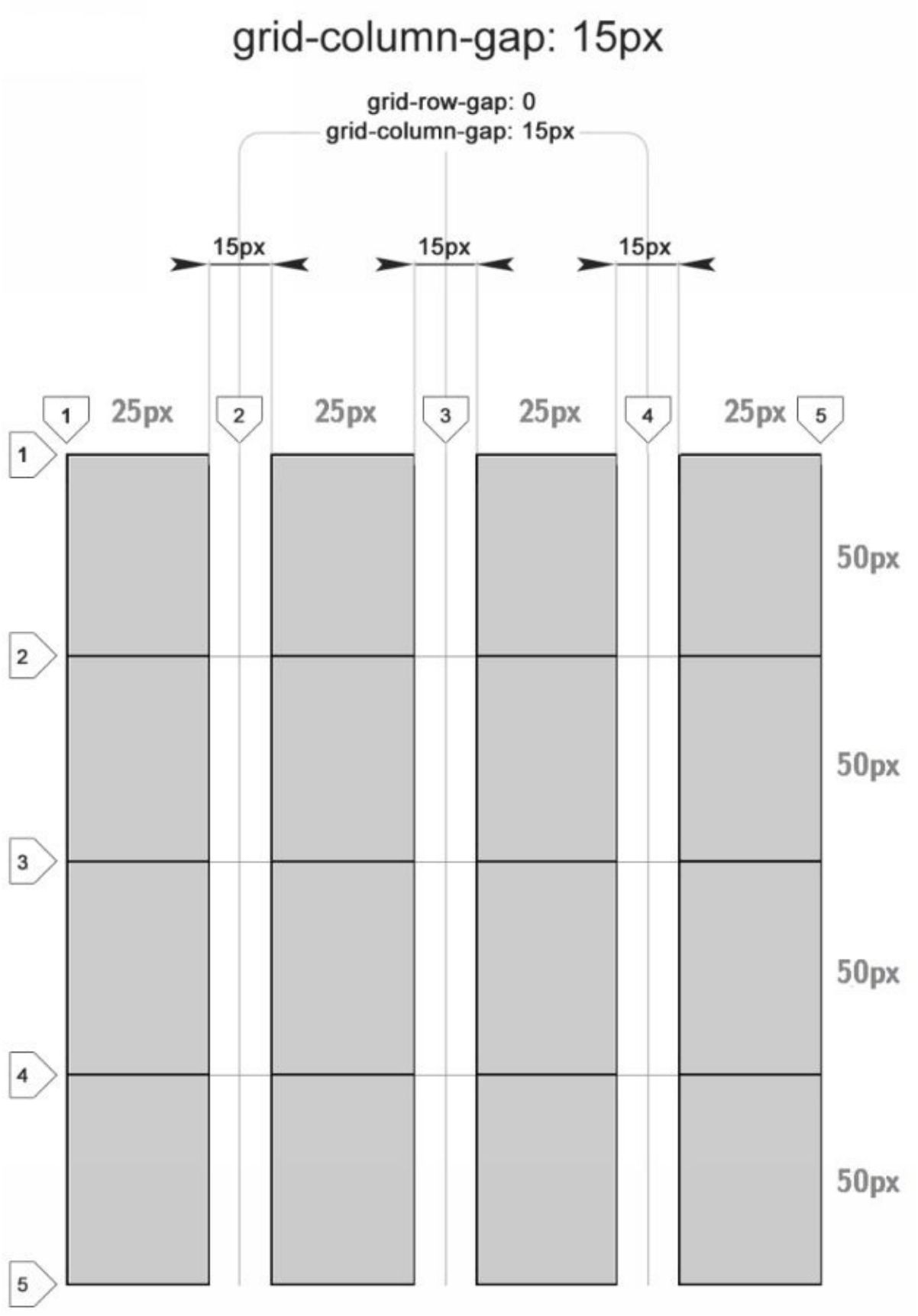

grid-column-gap: 15px
grid-row-gap: 0
grid-column-gap: 15px
15px
15px
15px
1
25px
2
25px
3
25px
4
25px
5
1
2
3
4
5
50px
50px
50px
50px

**Figure 189:** CSS grid has a property **grid-column-gap**, which is used to specify vertical gaps of equal size between all columns in your CSS grid.

I intentionally left the horizontal gaps clasped to their default value of 0, because they're not being discussed in this example.

I can already envision a Pinterest-like design with multiple columns using the setup above.

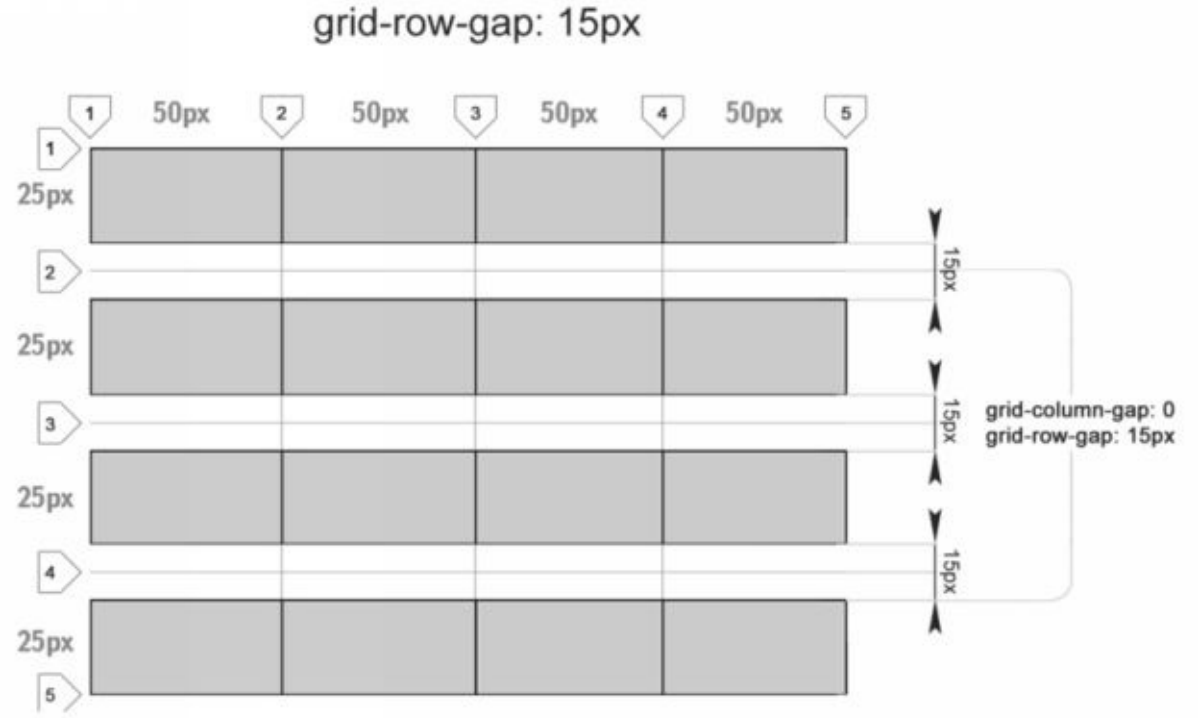



**Figure 190:** Likewise, using **grid-row-gap** property we can set horizontal gaps for the entire grid.

This is the same thing, except with horizontal gaps.

Using **grid-gap** property we can set gaps in both dimensions at the same time:

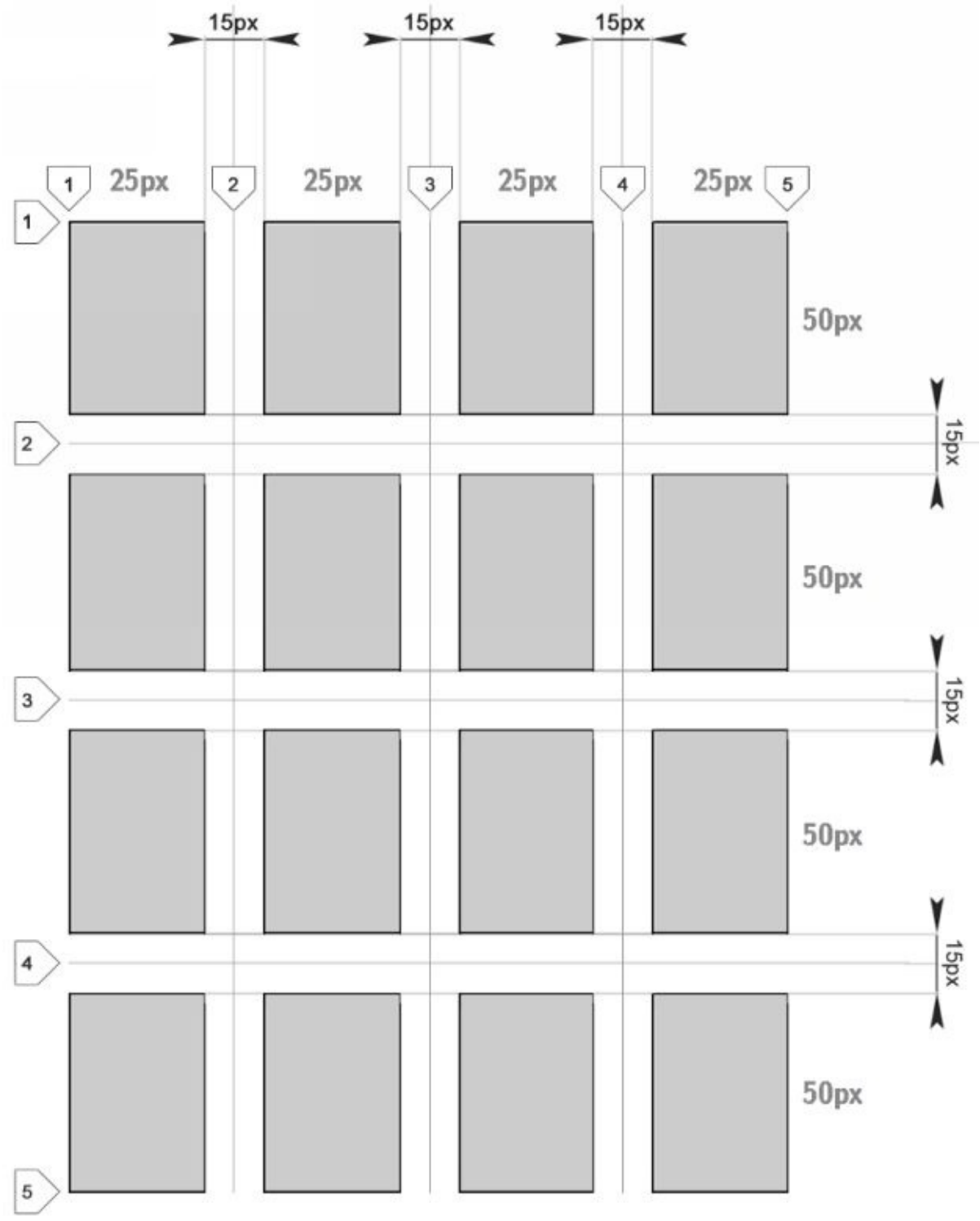



**Figure 191:** It is possible to set the gaps on entire CSS grid by using shorthand property **grid-gap**. But this means that gaps in both dimensions will be set to the same value. In this example it is **15px**.

And finally... you can set gaps individually for each of the two dimensions.

The next 3 diagrams were created to demonstrate the different possibilities made of using the CSS grid that can be useful in various cases.

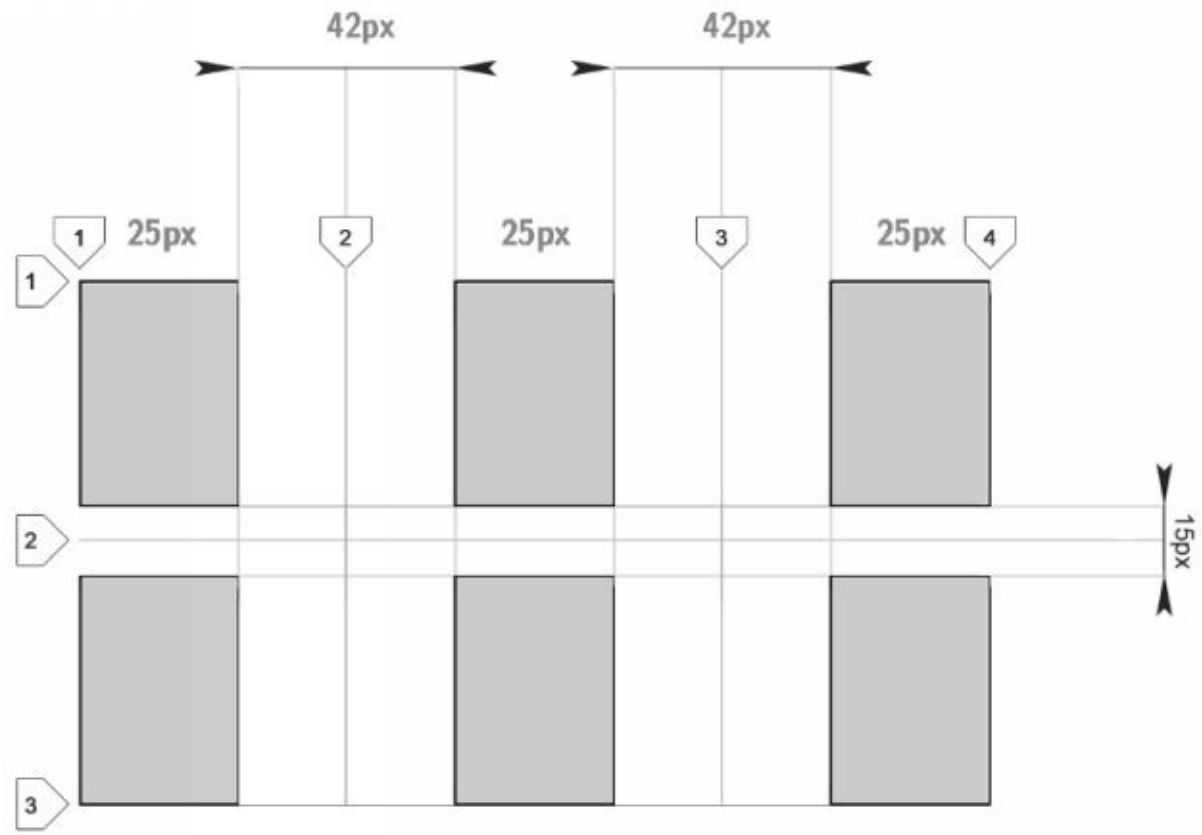



**Figure 192:** Here gaps are set individually per row and column which allows for varied column design. In this example wide column gaps are used. You can probably use this strategy for crafting *image galleries* for wide-screen layouts.

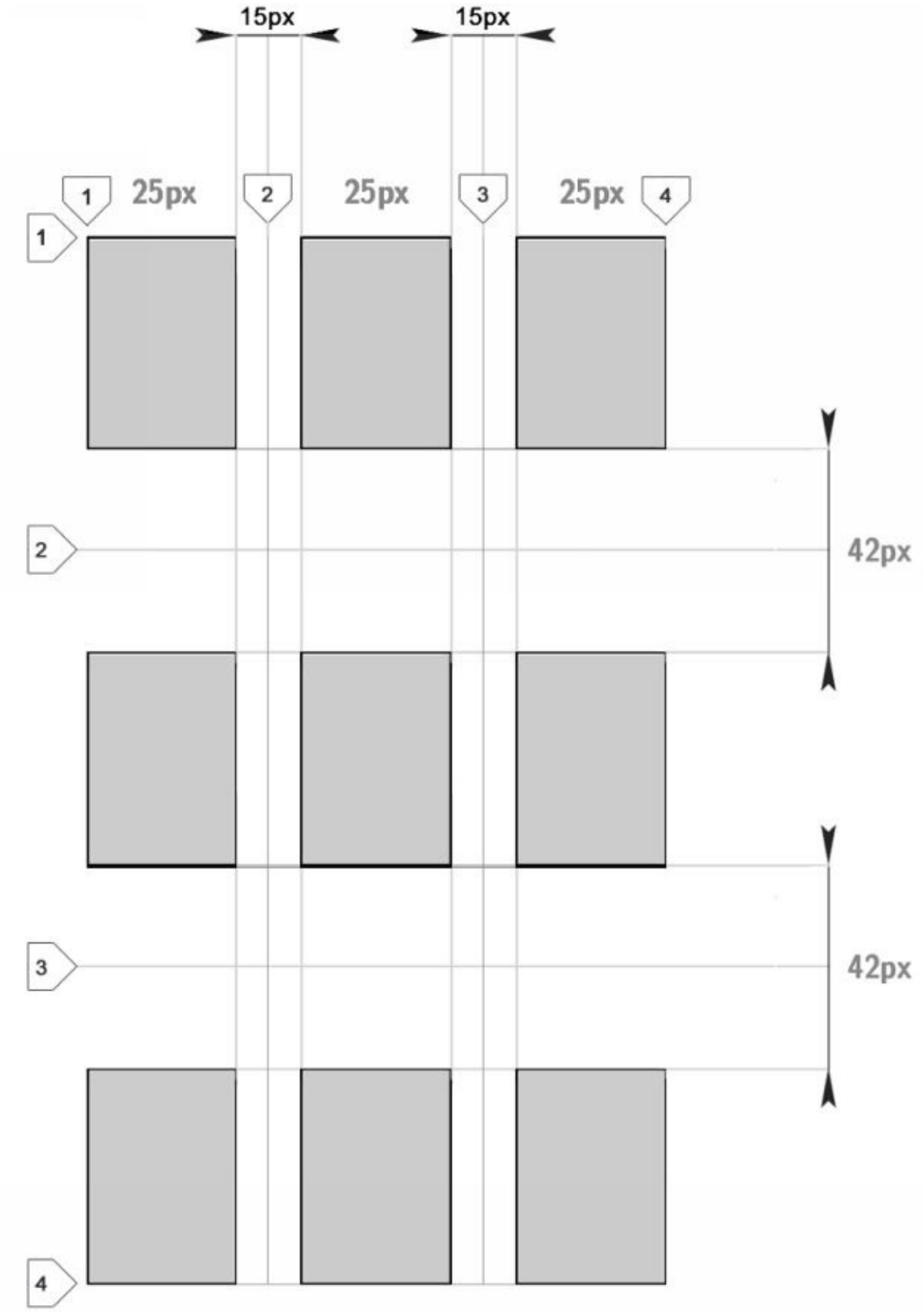

15px
15px
1
25px
2
25px
3
25px
4
1
2
42px
3
42px
4

**Figure 193:** Here is the same thing as the previous example. Except we're using wider row gaps.

One thing I was disappointed about was the lack of support for the ability to create varied gap sizing within the same dimension. I think this is the most daunting limitation of CSS grid. And I hope in the future it gets fixed.

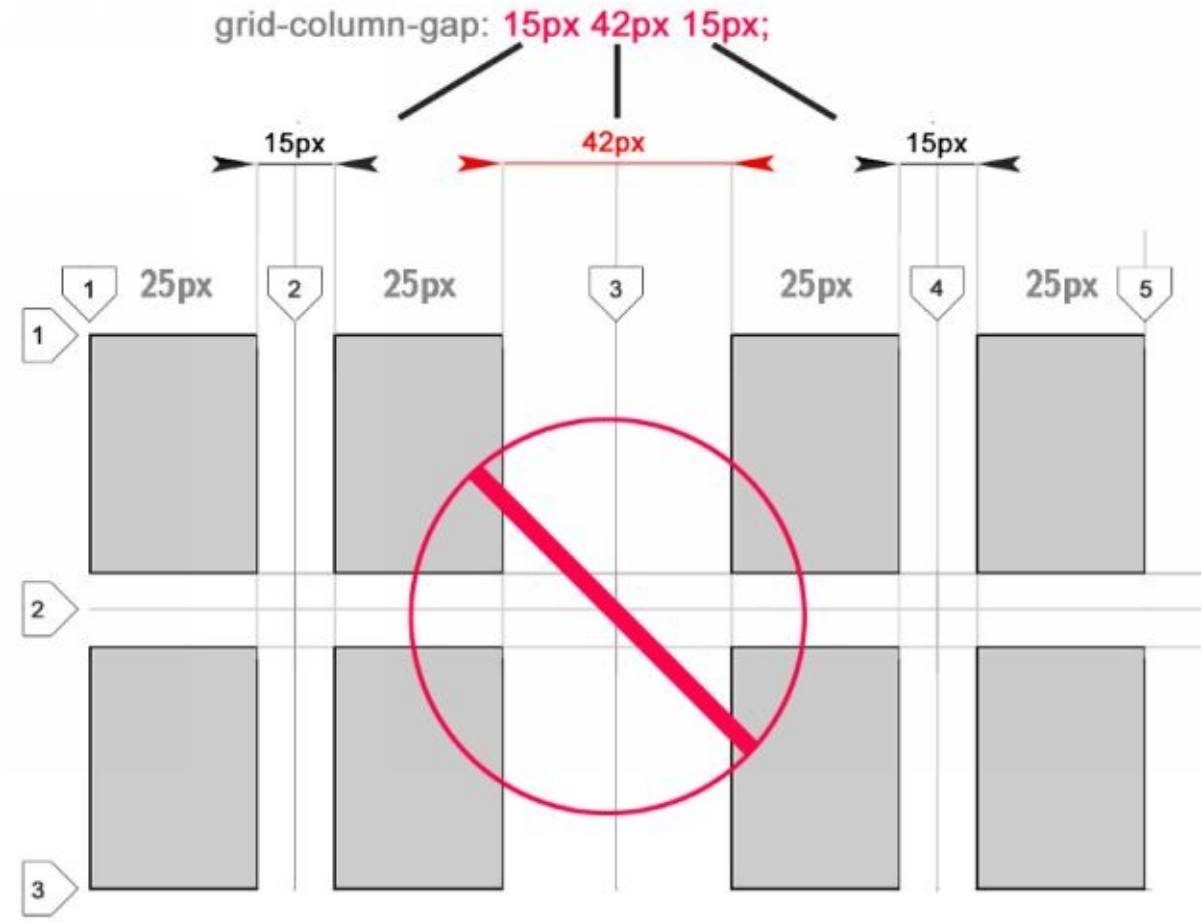



**Figure 194:** This layout cannot be created using CSS grid. Varied gap sizing is currently (June 2nd, 2018) not a possibility with CSS grid.

# fr -- Fractional Unit -- for efficiently sizing the remaining space.

One *somewhat* recent addition to CSS language is the ***fr*** unit.
The ***fr*** units can be used on things other than CSS grid. But in combination with one they are magical for creating layouts with unknown screen resolution... and still preserve proportion without thinking in percent.
The ***fr*** unit is similar to percentage values in CSS (25
%, 50
%, 100
%... etc) except represented by a fractional value (0.25, 0.5, 1.0...) But although it could be ***1fr*** is not always 100
%. The ***fr*** unit automatically dissects remaining space. The easiest way to demonstrate this is by following diagrams.
Here is a basic example of using ***fr*** units:

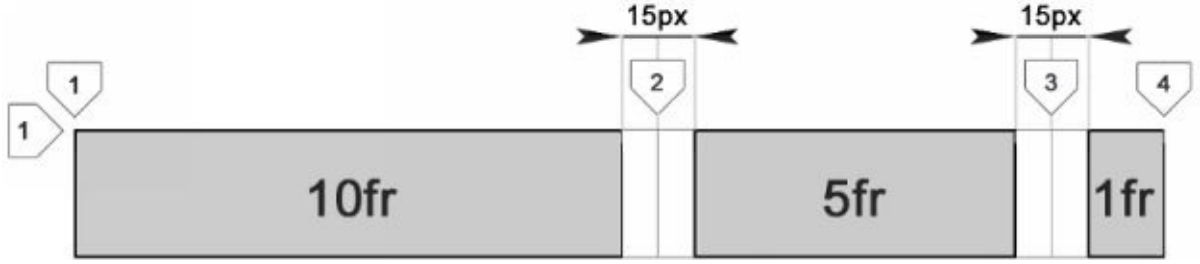



**Figure 195:** An example of using the ***fr*** CSS unit.

This is great news for intuitive designers.
***1fr*** will be 10/1 of ***10fr*** regardless of how much space ***10fr*** takes up.
It's all relative.

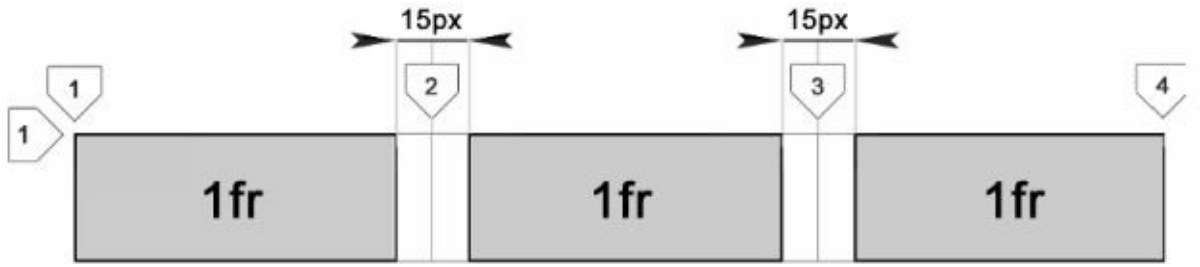



**Figure 196:** Using ***1fr*** to define 3 columns produces columns of equal width.

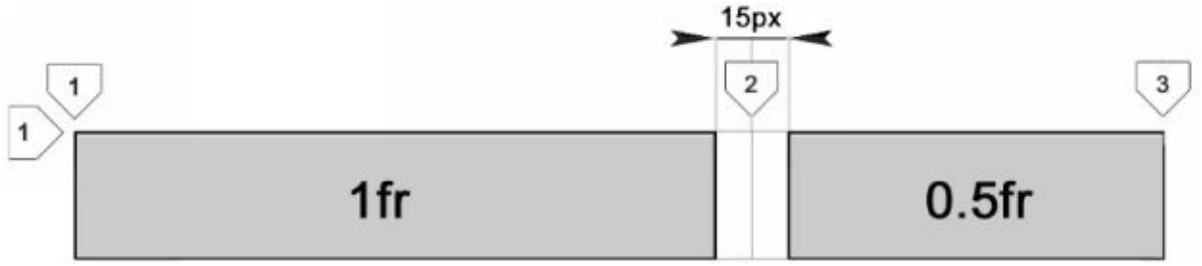



**Figure 197:** You can also use fractions.

Relative to ***1fr***, ***0.5fr*** is exactly half of ***1fr***.
These values are calculated relative to the parent container.
Can you mix percentage values with ***1fr***? Of course you can!

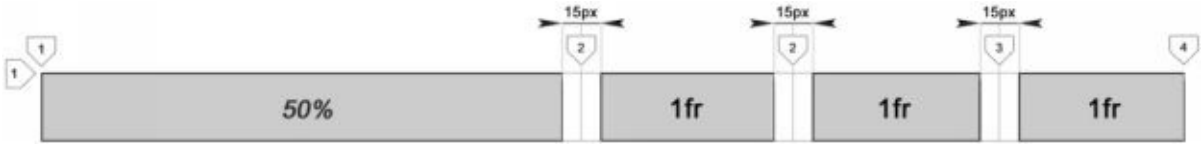



**Figure 198:** The example here demonstrates mixing % units with ***fr***. The results are always intuitive and produce the effect you would expect.

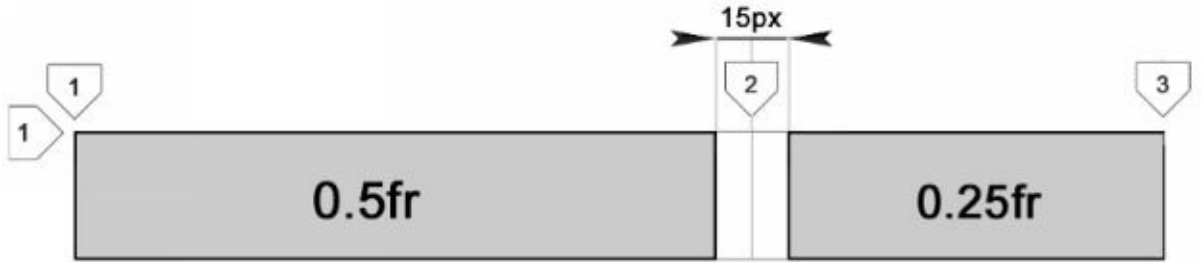



**Figure 199:** Fractional ***fr*** units are relative to themselves within some parent container.

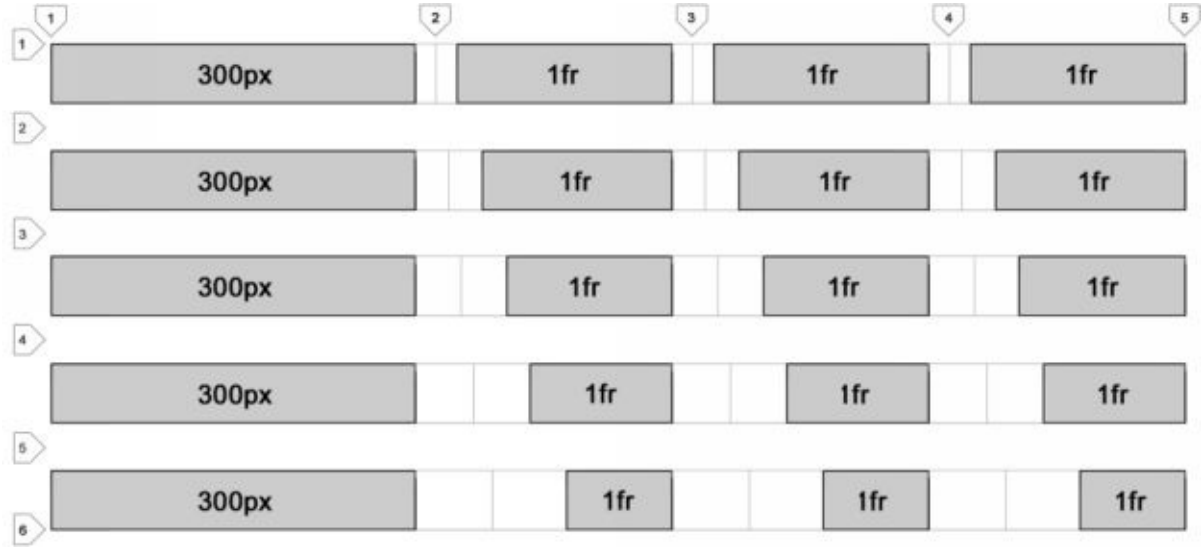


**Figure 200:** Using ***1fr*** units and increasing column gaps at the same time will produce this result. I just wanted to include this here to demonstrate that ***1fr*** units will be affected by gaps too. 5 different CSS grids used here to demonstrate how we should be also mindful of the gaps when designing with ***1fr*** units.

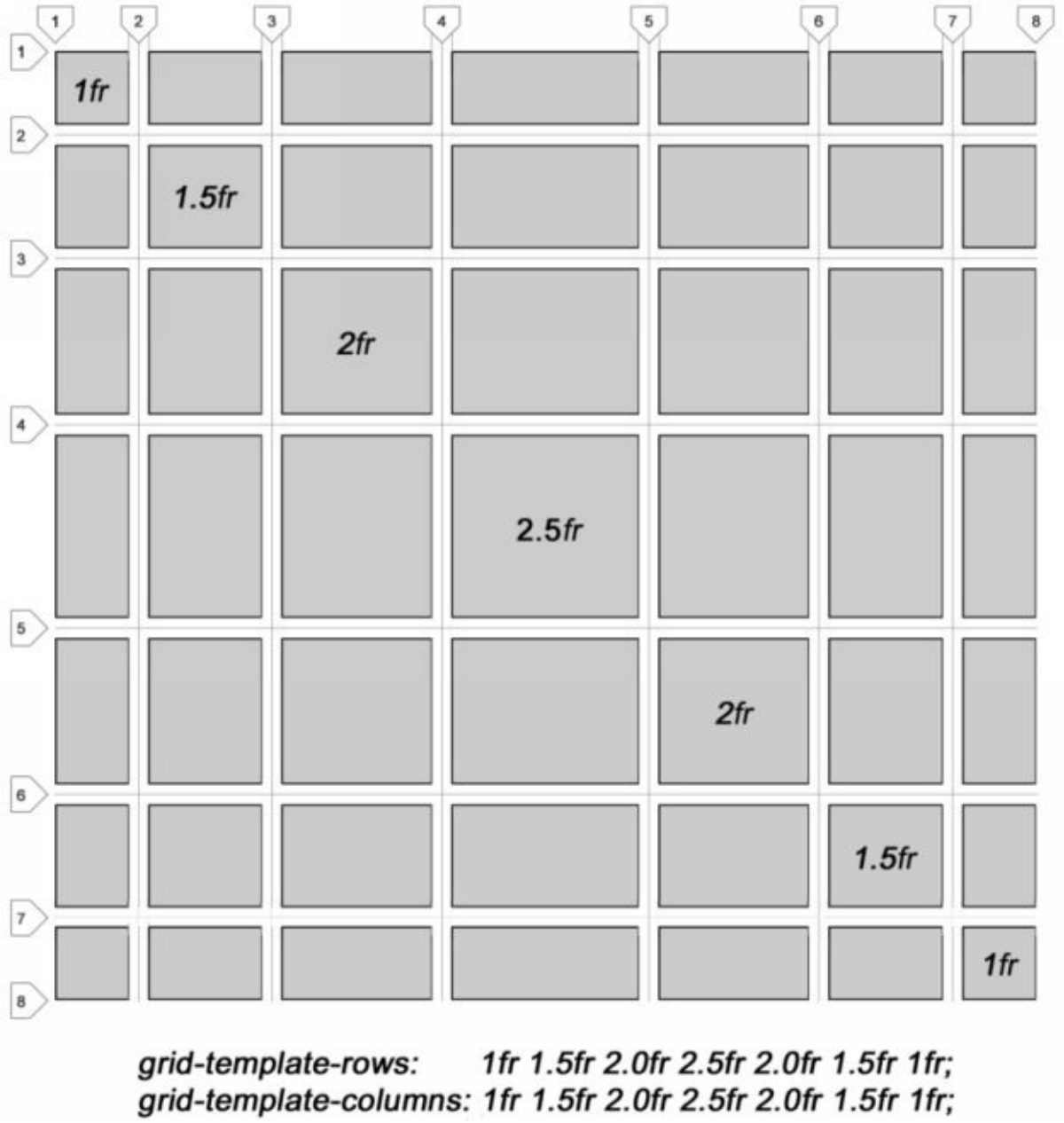



**Figure 201:** And just to be complete in our understanding of ***fr*** units, you can use them to create something like this. Although I don't know where you would require such a dinosauric layout it clearly demonstrates how ***fr*** units can affect both rows and columns.

# Repeating Values

CSS grid allows the use of repeat property value.
The repeat property takes two values: times to repeat and what to repeat.
**repeat(times, ...what);** At its basic this is how it works:

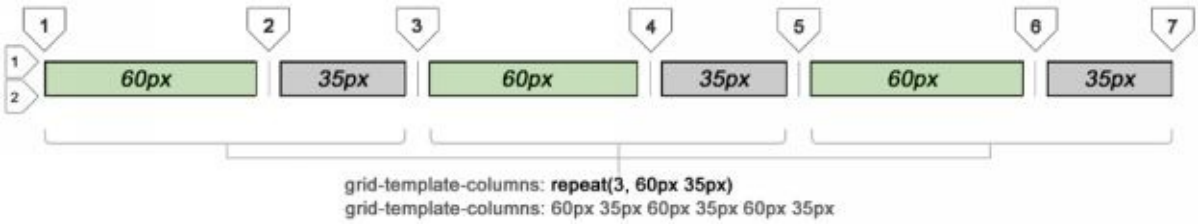



**Figure 202:** Here we're using **grid-template-columns** with repeat and without repeat to produce exactly the same effect. It is usually wise to choose the shortest path.

Here **grid-template-columns** are provided two different values to produce the same effect. Obviously repeat saves a lot of hassle here.
Final verdict: to save yourself from redundancy in cases where your grid must contain repetitive dimension values use repeat as a remedy.
The **repeat** property can be sandwiched between other values, too.


**Figure 203: grid-template-columns: 50px repeat(3, 15px 30px) 50px** In this example we repeat a section of two columns 15px 30px for 3 times in a row. I mean in a column. Ahh! You know what I mean.

# Spans

Using CSS grid spans you allow your items to stretch across multiple rows or columns. This is a lot like **rowspan** and **colspan** in a **<table>**.
We will create a grid using repeat to avoid redundant values. But it could have been created without it - anyway, let's make it our specimen for this section.
When we add **grid-column:span 3** to item
#4 a somewhat unexpected effect has occurred:

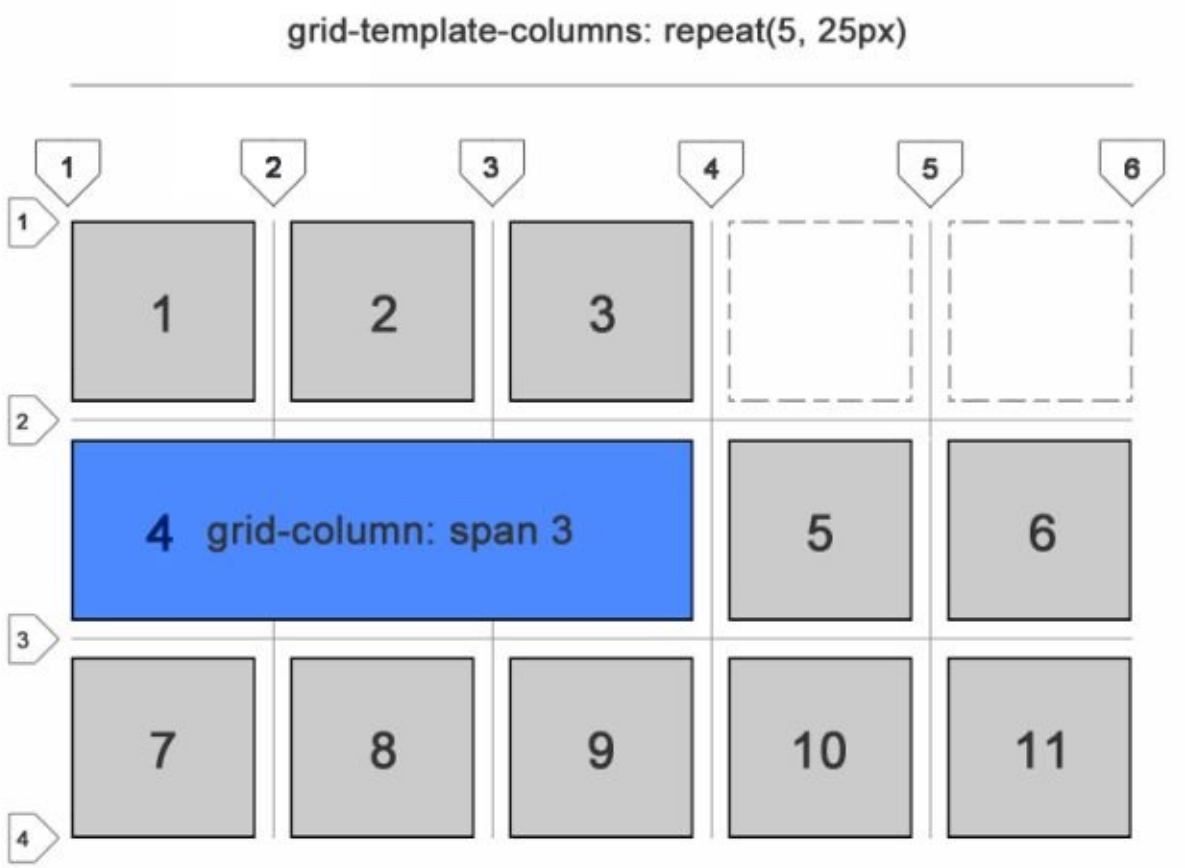



**Figure 204:** Using **grid-column:span 3** to take up 3 columns. However, CSS grid makes a decision to remove some of the items, because the "spanned" item cannot fit into suggested area. Notice the blank squares!

In CSS grids spans can also be used to cover multiple rows. And if it so happened that the column is now greater in height than the height of the grid itself the CSS grid adapt itself to this:

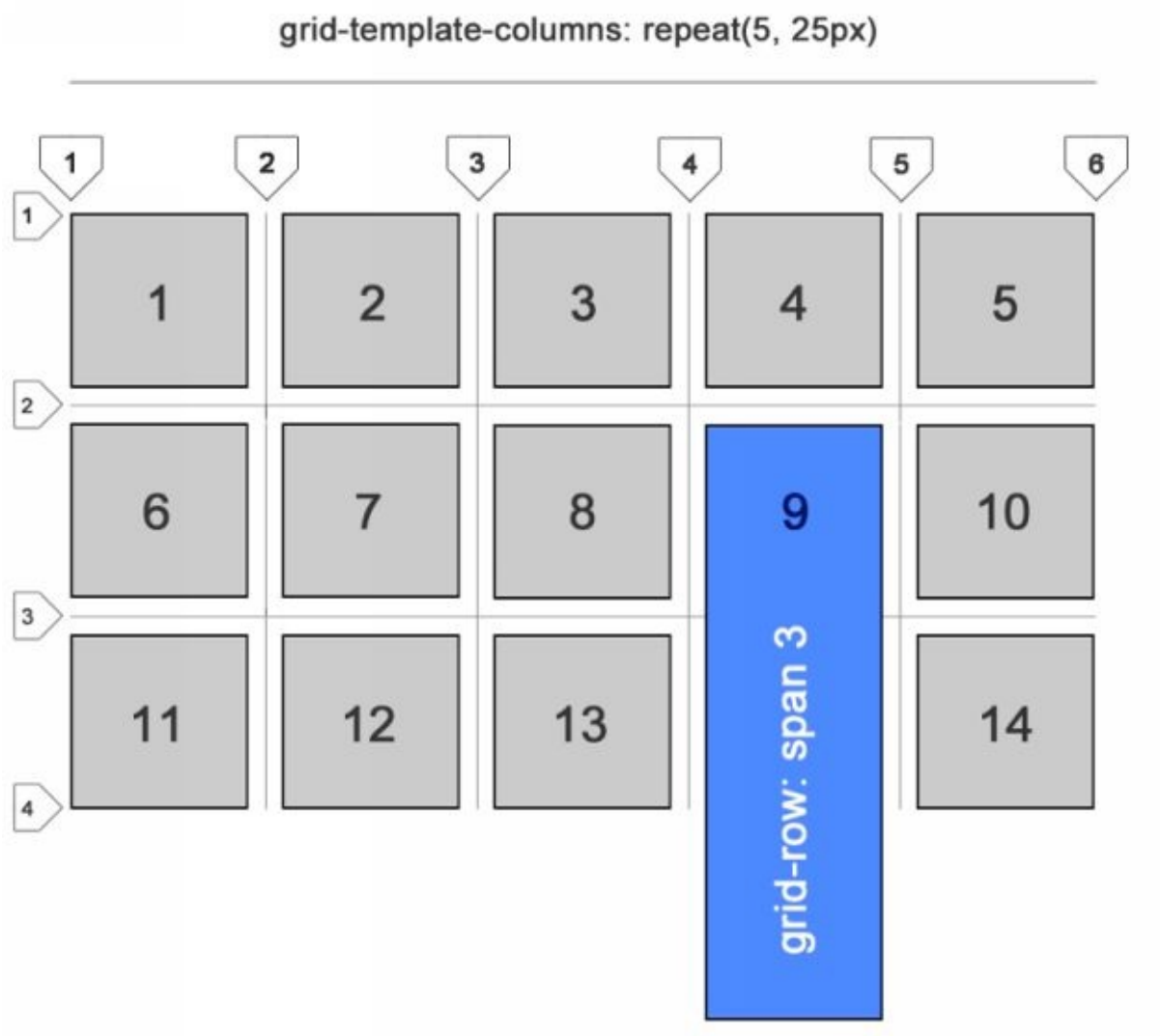



**Figure 205:** CSS grid is adapting itself in several cases where items go beyond grid's parent container.

You can also span across multiple rows and columns at the same time. I created this other minimalistic example just to quickly demonstrate limitations, even though in most cases this will probably not happen:

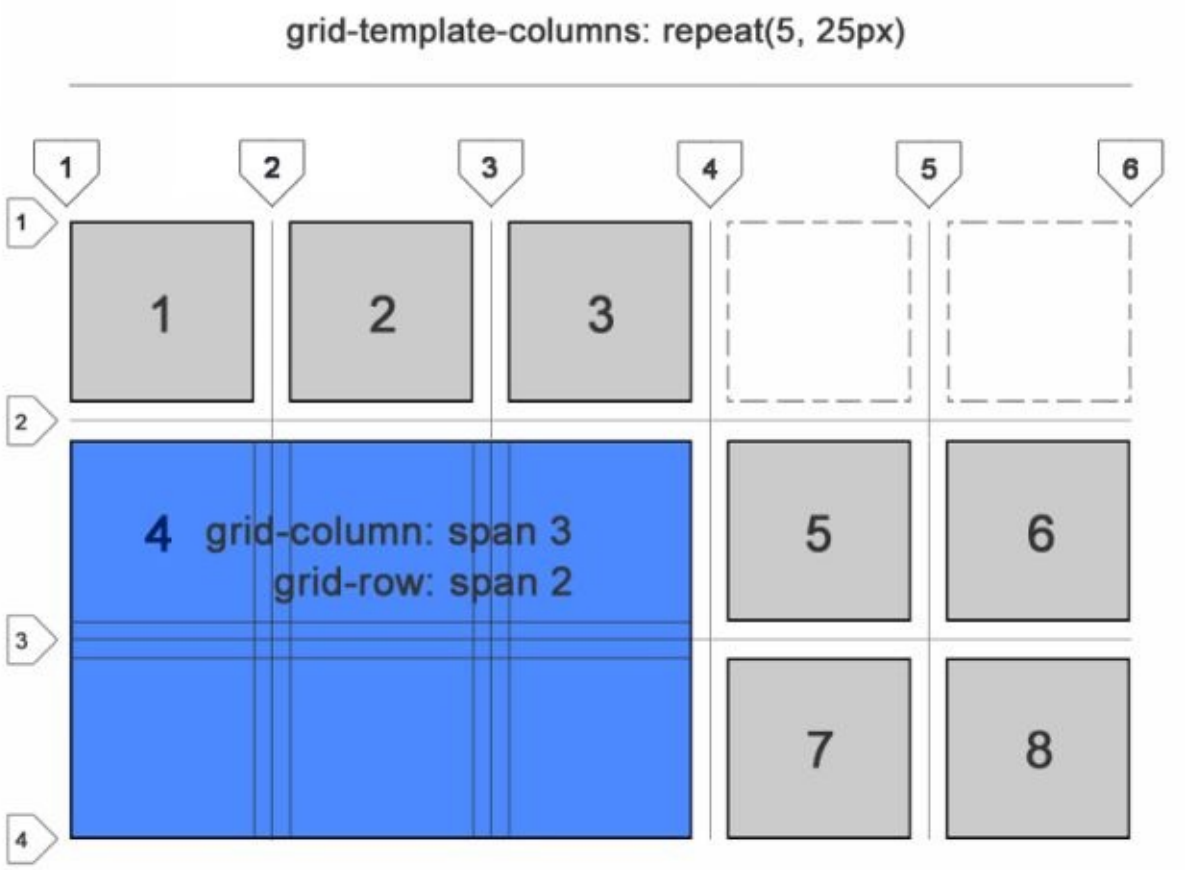
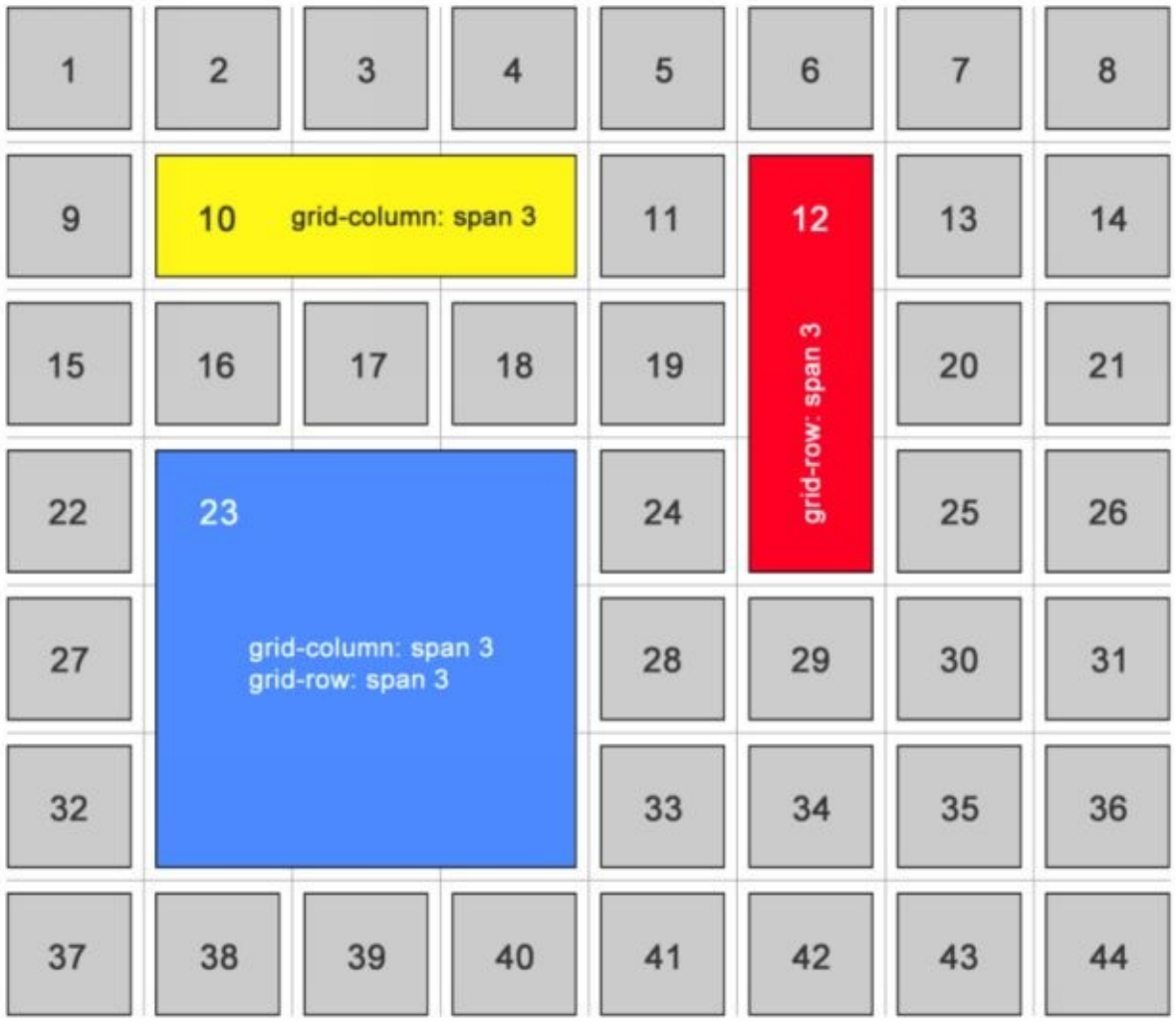



**Figure 206:** CSS grid fills in the blanks.

Pay attention to how CSS grid adapts to the items around spans that cover multiple rows and columns. All of the items still remain in the grid but intuitively wrap around other spanned items.




When I tried to break the layout with a large span I ended up with the following case that demonstrates the key limitations of CSS grid:

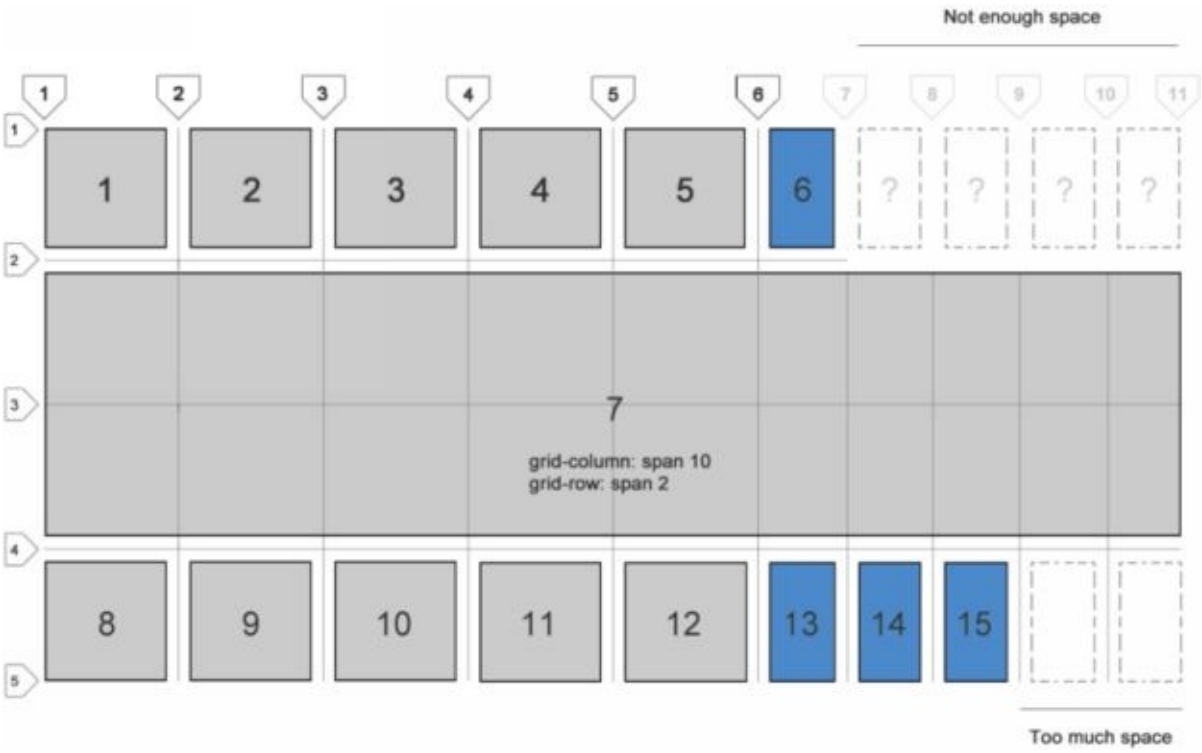



**Figure 207:** CSS grid replaces potential item cells with empty space in two distinct situations.

But it's still a lot like a . See my other CSS grid vs table tutorial where I show you the frightening similarities.

However... there is a solution.

# Start and End

So far we used CSS grid ***spans*** to create multi-column and multi-row items that occupy a ton of space. But... CSS grid has another much more elegant solution to solve the same problem.

The **grid-row-start** and **grid-row-end** properties can be used to define the starting and ending point of an item on the grid Likewise, their column equivalents are **grid-column-start** and **grid-column-end**. The are also two short-hand properties: **grid-row:1/2** and **grid-column:1/2**.

These work an a slightly different way than ***spans***.

With **-start** and **-end** properties, you can physically move your item to another location in the grid. Let's take a look at this minimalist example:

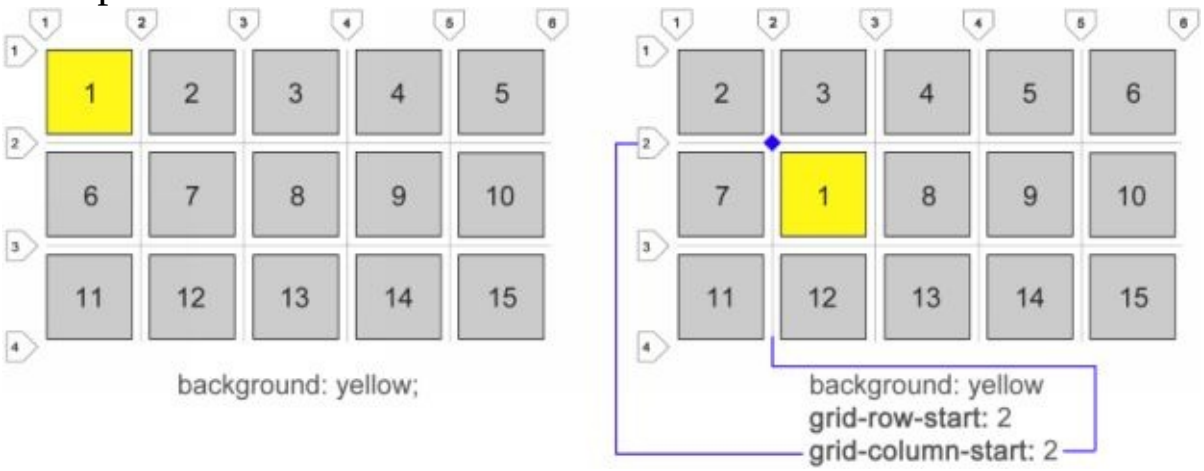



**Figure 208:** Using CSS grid's **grid-row-start** and **grid-column-start** on an item (first item in this example) you have the ability to physically move an item within your grid to another location!

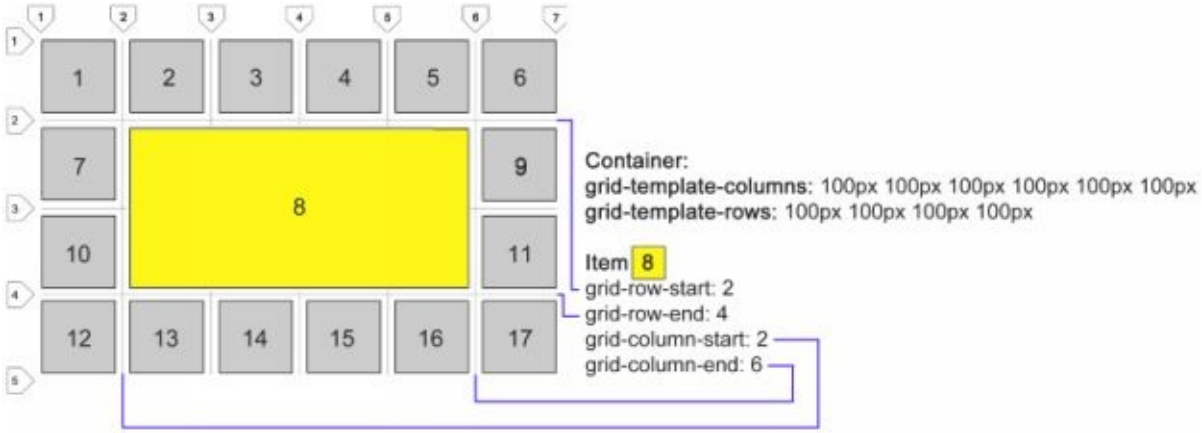



**Figure 209:** Here we've taken item 8 and (redundantly) specified its location using **grid-row-start** and **grid-column-start**. Notice however, this alone has no effect on item 8, because item 8 is already positioned at that location on the grid anyway. However, by doing this you can achieve span-like functionality if you also specify an ending location using **grid-column-end** and **grid-column-end**.

Interestingly, designers of CSS grid have decided for the direction of the span vector to be insignificant. The span is still created within the specified area regardless of whether starting or ending points are provided in reverse order:

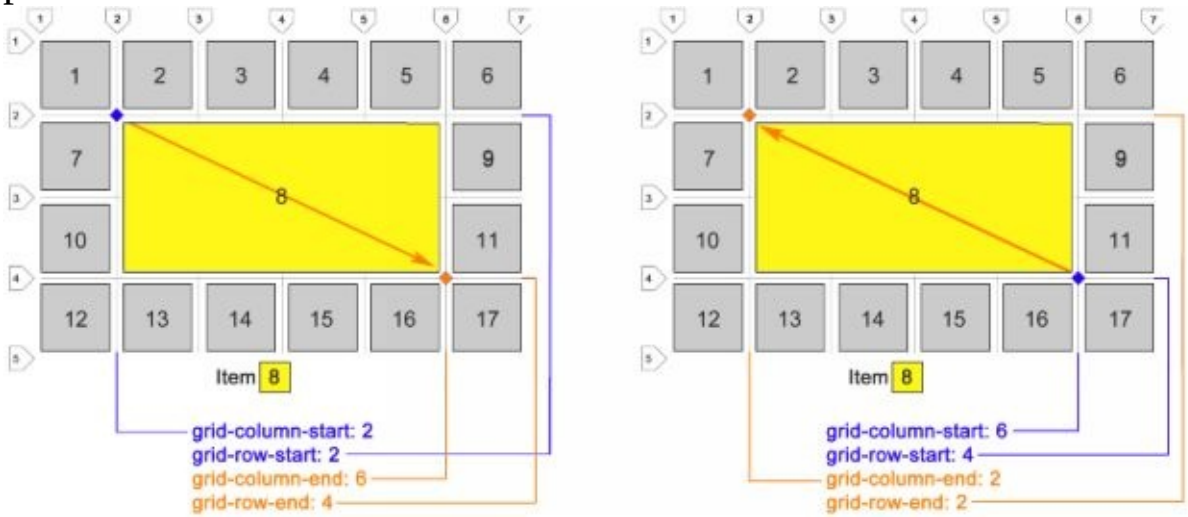



**Figure 210:** Specifying item span regardless of the direction of the start/end points produces the same results.

Let's consider this 6 x 4 CSS grid. If you explicitly specify an items's column end position that goes outside the number of specified columns (>=7) you will experience this wonky effect:

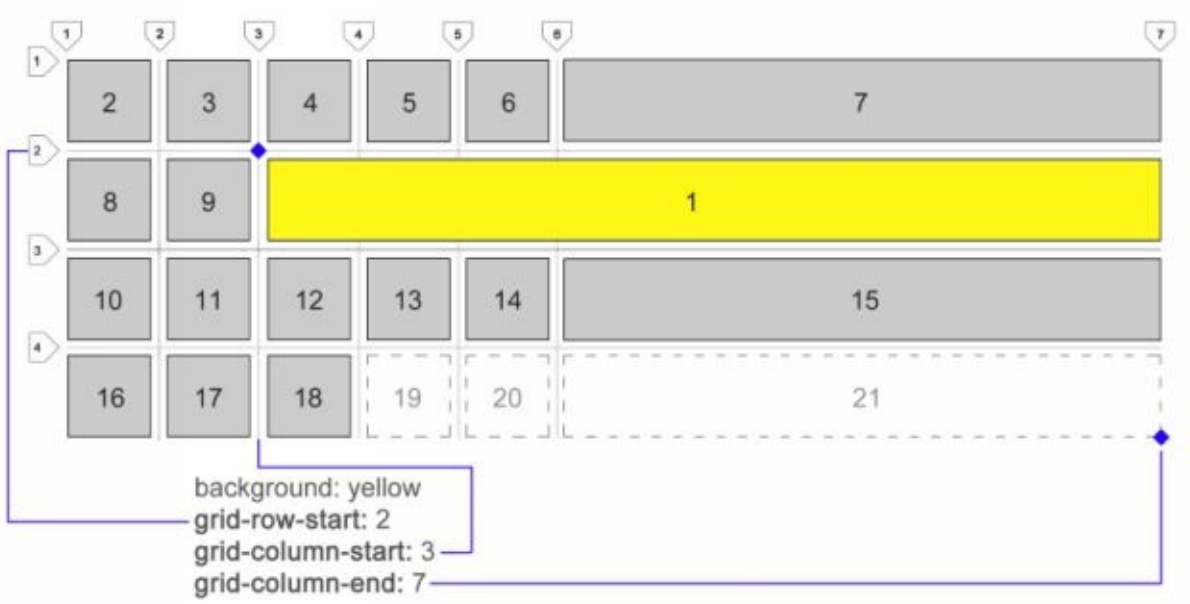



**Figure 211:** Making column width of an item greater than the original number of columns specified in the CSS grid.

In this case CSS grid will adapt the resulting layout of your grid to what you see in the example above. It's usually a good idea to design your layouts by being mindful of grid's boundaries to avoid these types of scenarios.

# Start and End's Shorthand

You can use the shorthand properties **grid-row** and **grid-column** to same effect as above using / to separate values. Except instead of providing an end value, it takes width or height of the span:

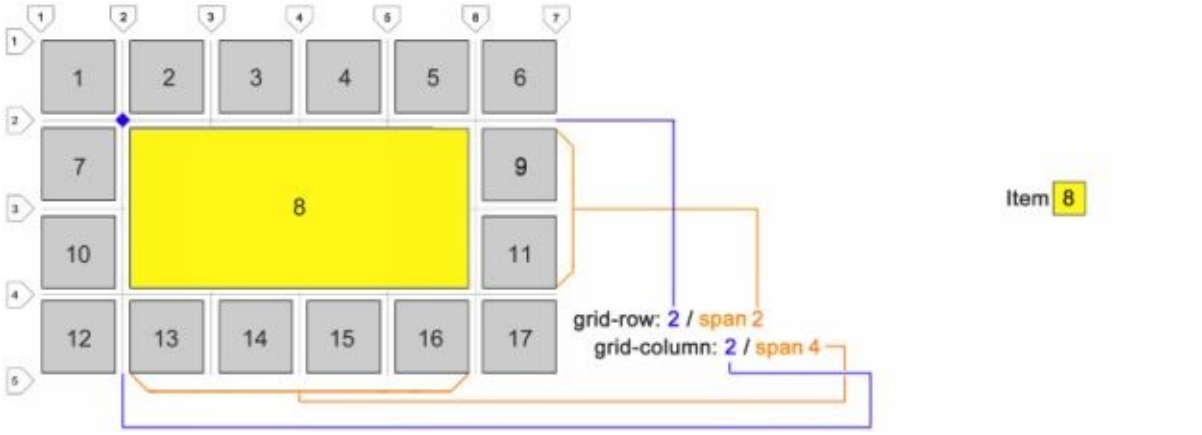



**Figure 212:** We can use / character for this short-hand syntax.

What if we need to reach the absolute maximum boundary of the grid? Use **-1** to extend a column (or row) all the way to the end of CSS grid's size when number of columns or rows is unknown. But be mindful of any implicit items (16, 17) slipping away from the bottom of the grid:

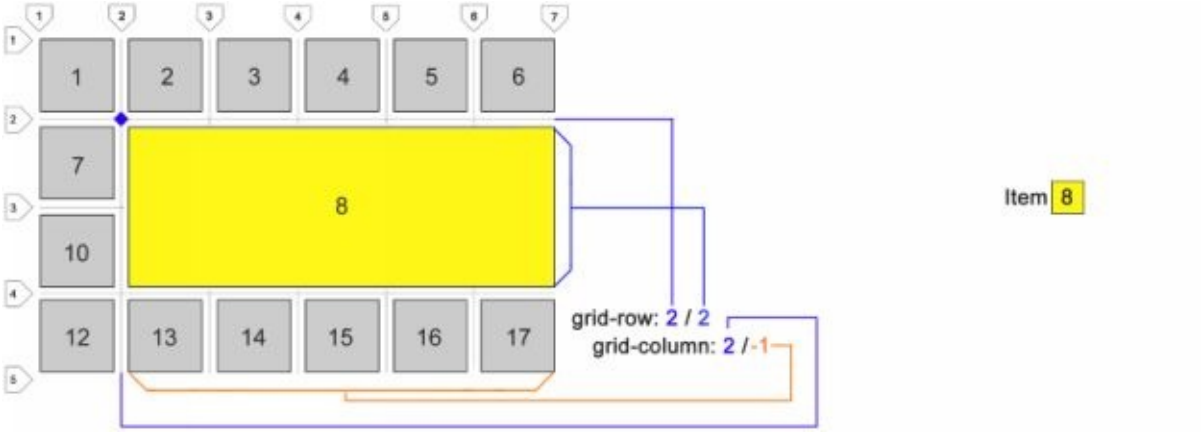



**Figure 213:** Using negative **-1** value to count from right-most gap to left.

Then I tried to do the same with rows, but the results were more chaotic, depending on which combinations of values I provided. I know there are other ways of using / but for the sake of clarity I want

to keep things simple.

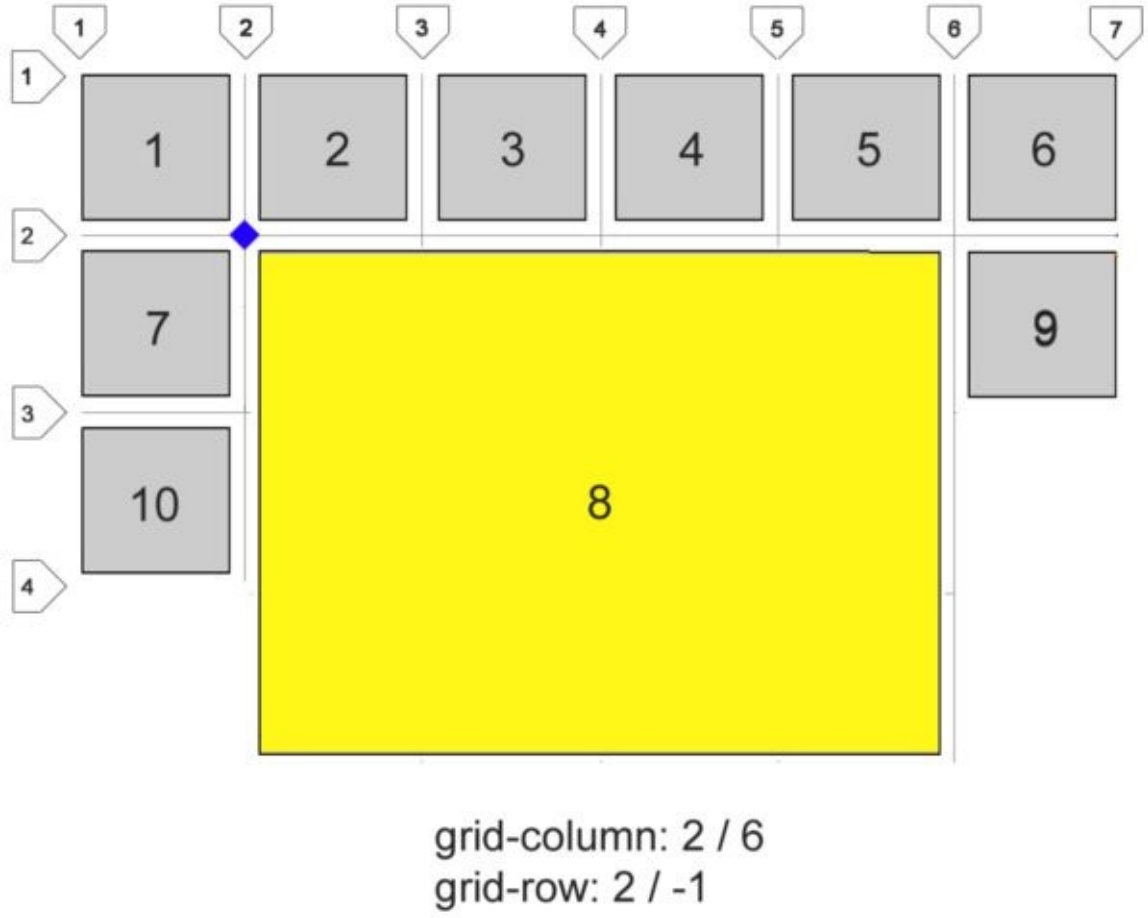



**Figure 214:** I only used 10 items in this example. CSS grid seems to gracefully resize itself.

When I was experimenting with rows to do the same thing, it seems like 4 in **grid-column:2/4** had to be changed to **2/6**... but only if **grid-row:2/-1** was specified.

That puzzled me a bit. But I guess I still have a lot of learning to do on how /-separated values work.

What I found out though is that juggling around values here produced results that cannot be easily documented using visual diagrams.

Well, at least we get the basic idea here. You can extend either column or row all the way to the maximum boundary using **-1**. How one affects the other takes a bit of practice to figure out in some specific cases.

We can expand on this a bit. CSS grid has a secondary coordinate system, so to speak. And because it doesn't matter which direction you use to make cross-column and cross-row spans you can use negative values:

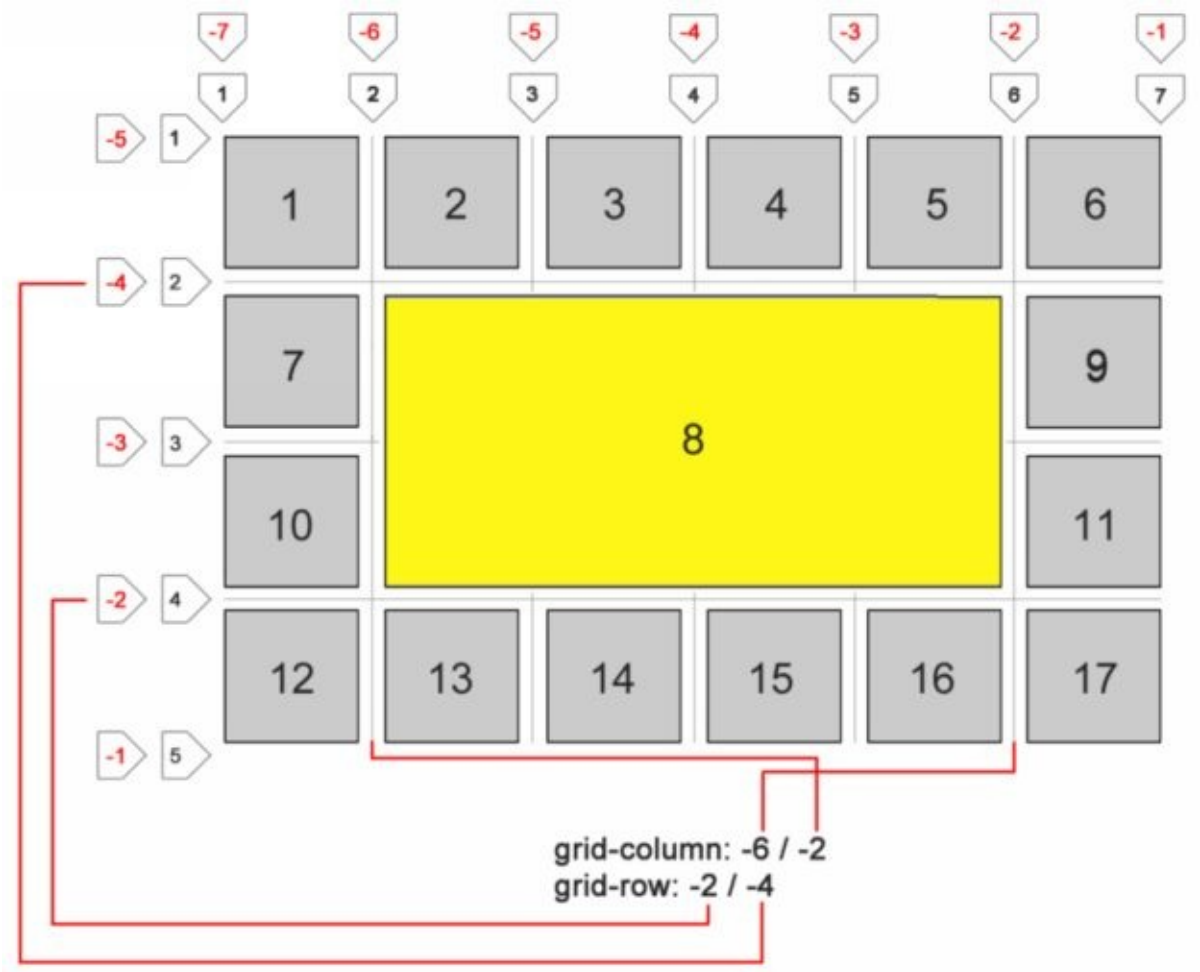



**Figure 215:** Using negative values to specify column and row's start and end, we can create the same span from previous examples, since CSS grid is coordinate system agnostic. You can use both positive and negative numbers!

As you can see CSS grid coordinate system is pretty flexible.

# Content Align Within CSS Grid Items

Let's say you've gone to the great lengths mastering CSS grid item spans. You crossed the seas of implicitly generated rows and columns. Now you're curious to see what else is in store for you.
Good news for you then.
As a web designer, I've for a long time craved multi-directional float. I wanted to be able to float in the middle and on any corner of the container.
This functionality is only limited to CSS grids' **align-self** and **justify-self** and does not appear to work on any other HTML element. If your entire site's layout is built using a CSS grid then it solves a lot of issues associated with corner and center element placement.

# align-self

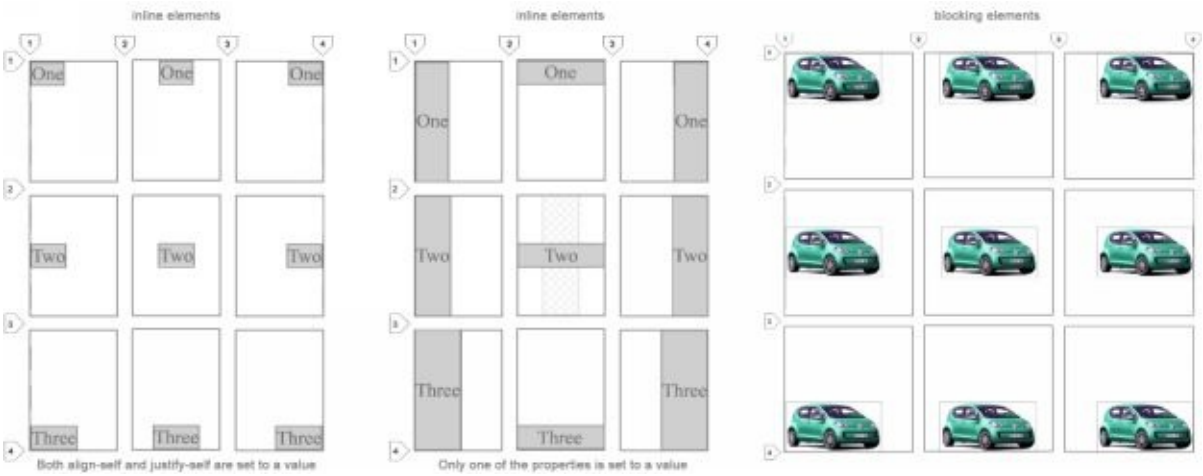



**Figure 216:** An example of using **align-self** and **justify-self** properties. The difference between the 9 squares is the combination of start and end values provided to the said properties to produce any of the results depicted above. I won't mention all of these combinations here, because it's quite intuitive.

**VERTICAL:** Use align-self: end to align the content to the bottom of the item. Likewise, align-self: start will make sure content sticks to the upper border.

**HORIZONTAL:** Use justify-self: start (or end) to justify your content left or right. In combination with align-self you can achieve placement depicted on any of the above examples.

Just to finalize this discussion here is how align-self affects a slightly more complex situation - one we've taken a look at before in this tutorial:

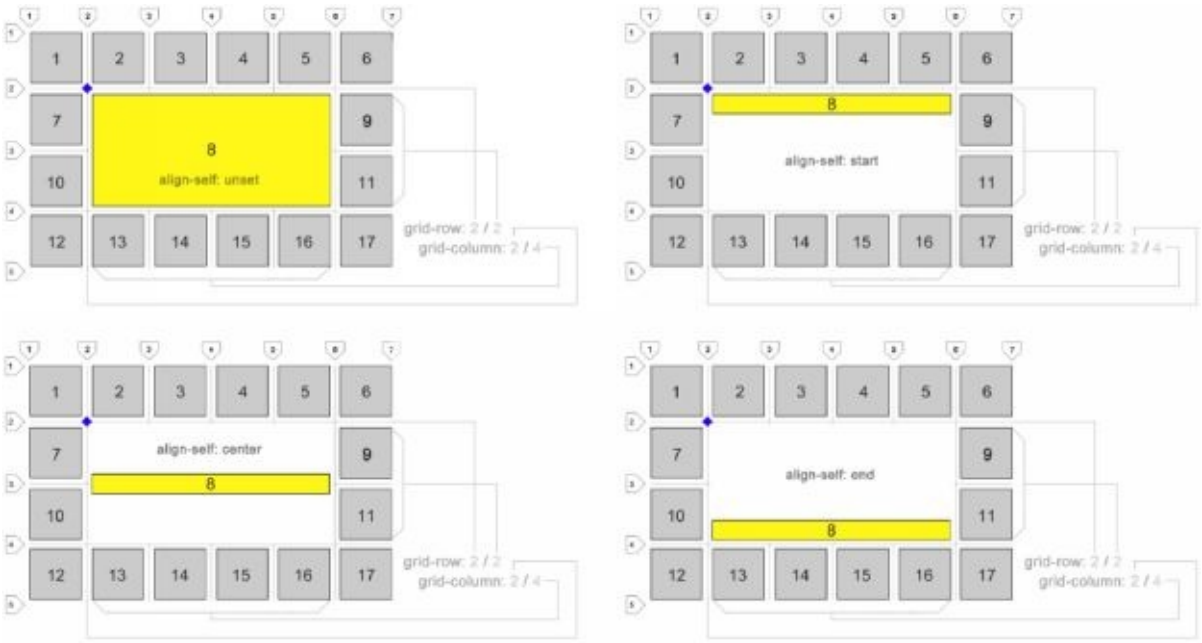



**Figure 217:** Using **align-self** it is possible to align the item's content with **start**, **center** and **end** values.

You can use values **start**, **center** and **end**.

Note, however there aren't **top** and **bottom** values for **align-self**.

# justify-self

Another property that does the same thing but horizontally is **justify-self**:

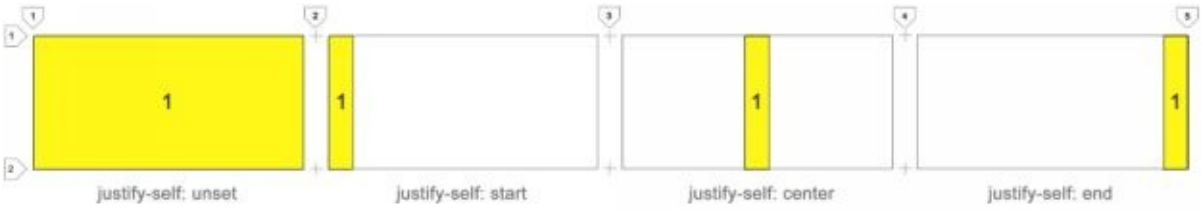



**Figure 218:** CSS grid item property justify-self in action using unset, start, center and end values.

You can use **start|left** or **end|right** values interchangeably here.

# Template Areas

Template areas provide a way to refer to an isolated part of your grid by a predefined name. This name cannot include spaces. Use - instead. Each set of row names is enclosed in double quotes. You can further separate these sets of row names either by a line break or by space to create columns as shown in the example below.
Although only 5 items are present template are names can logistically occupy places not yet filled with items:

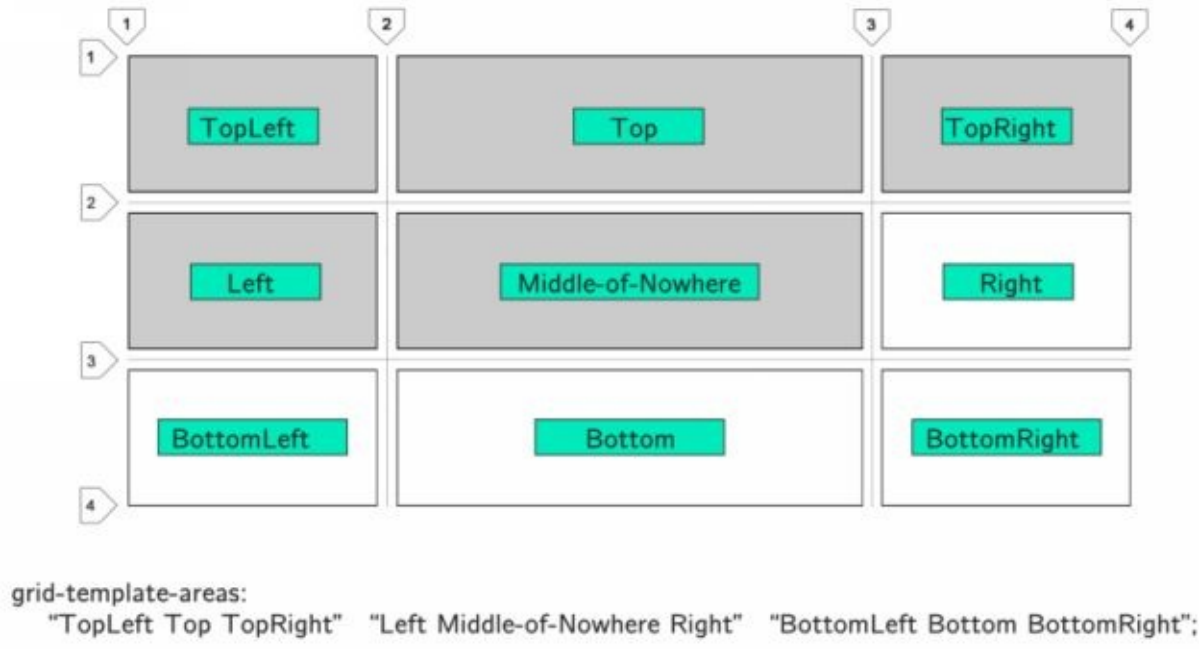



**Figure 219:** Example of specifying template areas with **grid-template-areas** property.

You can specify an area for any of the row and column as long as you separate the set of each consequent row by a space, and provide names for each row using double quotes. Within double quotes, each item is separated by space. This means ***no spaces are allowed in template area names***.
Similar principle to specifying row and column size is followed here to name all of the areas in the grid. Just separate them by space or tab. This syntax simply allows us to intuitively name our template areas.

But things get a lot more convenient when you start combining areas with the same name across multiple containers. Here I named 3 items in the left column Left and 3 items in the right column Right. CSS grid template areas automatically combined them to occupy the same space by name.

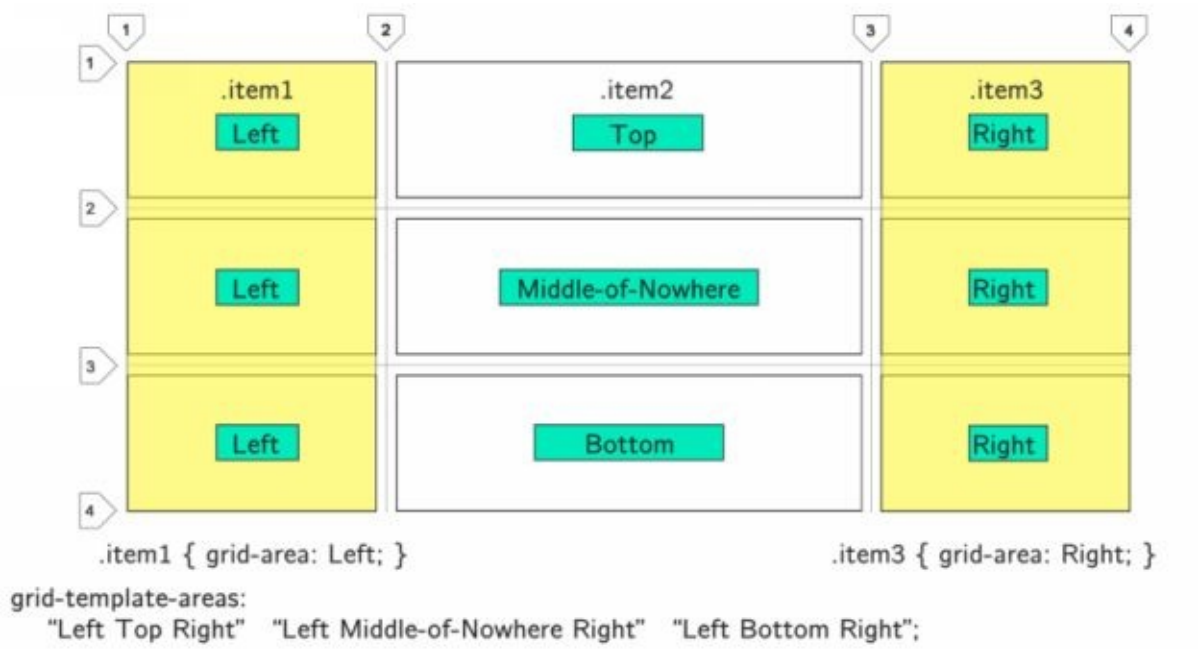



**Figure 220:** Spanning template areas across multiple grid cells."

Simply name your columns and rows, and adjacent blocks will merge" into larger areas. Just make sure to keep them rectangular! It's important to make sure that areas consist of items aligned into larger ***rectangle*** areas. ***Doing Tetris blocks here will not work***. Straying from the rule of always keeping your areas rectangular is likely to break the CSS grid and/or produce unpredictable outcome.

# Naming Grid Lines

Working with numbers (and negative numbers) can become redundant over time especially when dealing with complex grids. You can name grid lines with whatever you want using **[**name**]** brackets right before size value.

To name the first grid line you can: **grid-template-column:[left] 100px** Likewise for rows it is: **grid-template-row:[top] 100px**.

You can name multiple grid lines. The **[]** brackets are inserted at an intuitive place in the set. Exactly where the grid line (a.k.a. gap) would appear: **grid-template-columns:[left] 5px 5px [middle] 5px 5px [right]**

Now you can use the names left, middle and right to refer to your grid lines when creating columns and rows that need to reach that area:

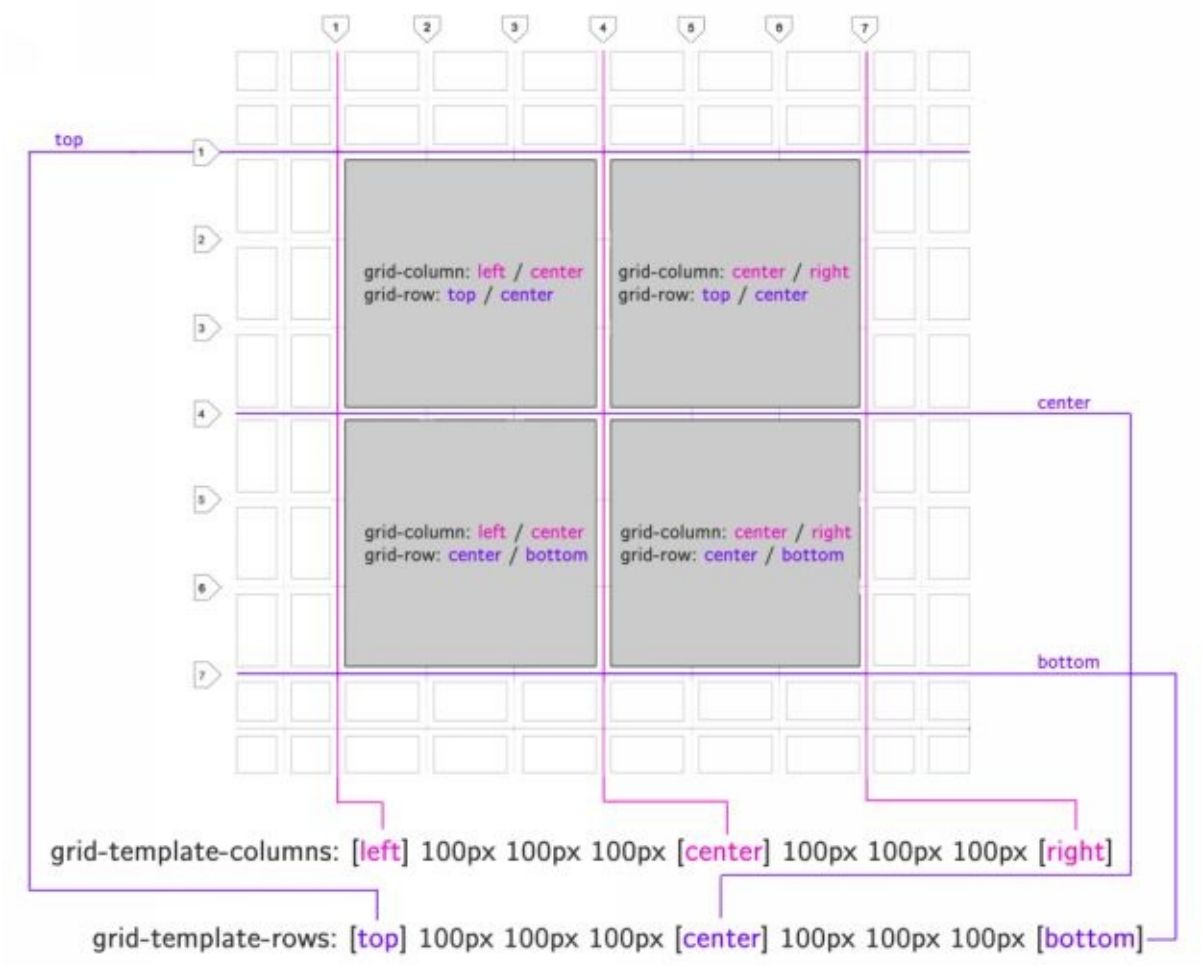



**Figure 221:** Instead of using numbers it is possible to name and refer to lines in between grid's cells as values to your properties **grid-template-columns** and **grid-template-rows**. Note how each span in the diagram refers to named lines and not the gap's numbers. You can use any name you want.

Naming gap lines creates a more meaningful experience. It's a lot better to think of the middle line as *center* (or *middle*) instead of 4. This tutorial covered almost everything there is to CSS grid using visual diagrams.

cleardoublepage In conclusion... *remember...*
***The music is not in the notes, but in the silence between -- Wolfgang Amadeus Mozart*** This seems to be true of CSS grid also. And so many other things!

Nearly 8 weeks have been spent drawing diagrams representing pretty much every single thing you can possibly do with CSS grid. And you've just looked at (and hopefully learned from) them all.
Of course, I assume the possibility that a few things were missed here and there. It's impossible to document absolutely every possible case. And I will be glad for anyone in the community to point it out so this book can be improved in the future editions with even more useful examples.

# Tesla CSS Art

Although CSS language was designed primarily for helping with the creation of websites and web application layouts, some talented UI designers have pushed it to its absolute limit! Some argue that there is little practical use in doing this. But the fact remains... these artists create challenging designs using deep knowledge of CSS properties and values.

Below is a CSS model of Tesla in space, designed by Sasha Tran (@sa_sha26 on Twitter) exclusively for this book!


**Figure 222:** Tesla in space, designed entirely in CSS by Sasha Tran (@sa_sha26) you should follow her on Twitter if you want to stay in touch with a talented UI designer!

The remaining pages of this book will describe, in great detail, how each separate part of this car was created, which CSS properties were used, etc.

Making CSS art can be a challenge, even for web designers. We're taking everything we've learned so far in this book and putting it into action!

It's all about how skillful you are with the CSS properties: **overflow:hidden**, **transform:rotate**, **box-shadow** and **border-radius**.

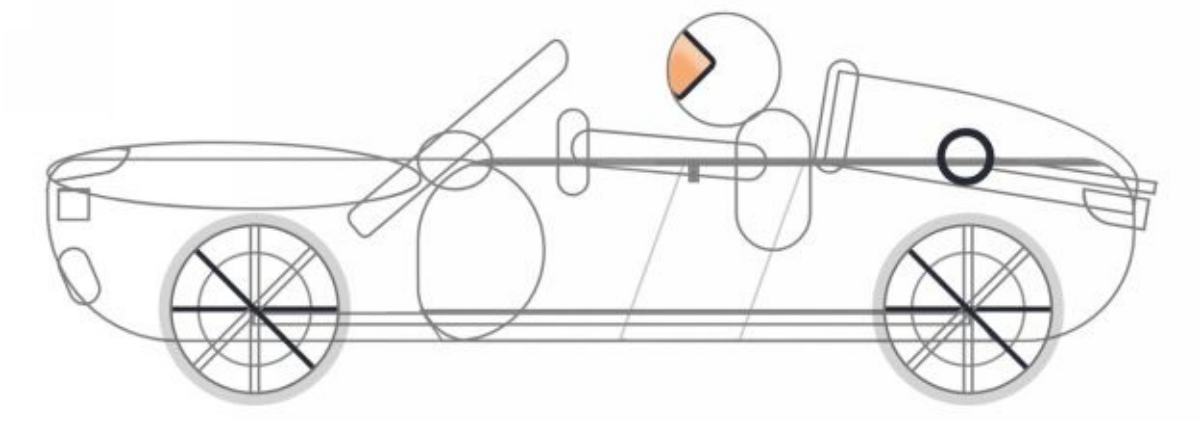


**Figure 223:** By making all backgrounds transparent you can clearly see the Tesla's composition, consisting of several HTML ** ** elements.

On the pages that follow we'll break down each significant element of the car to demonstrate how it was created.

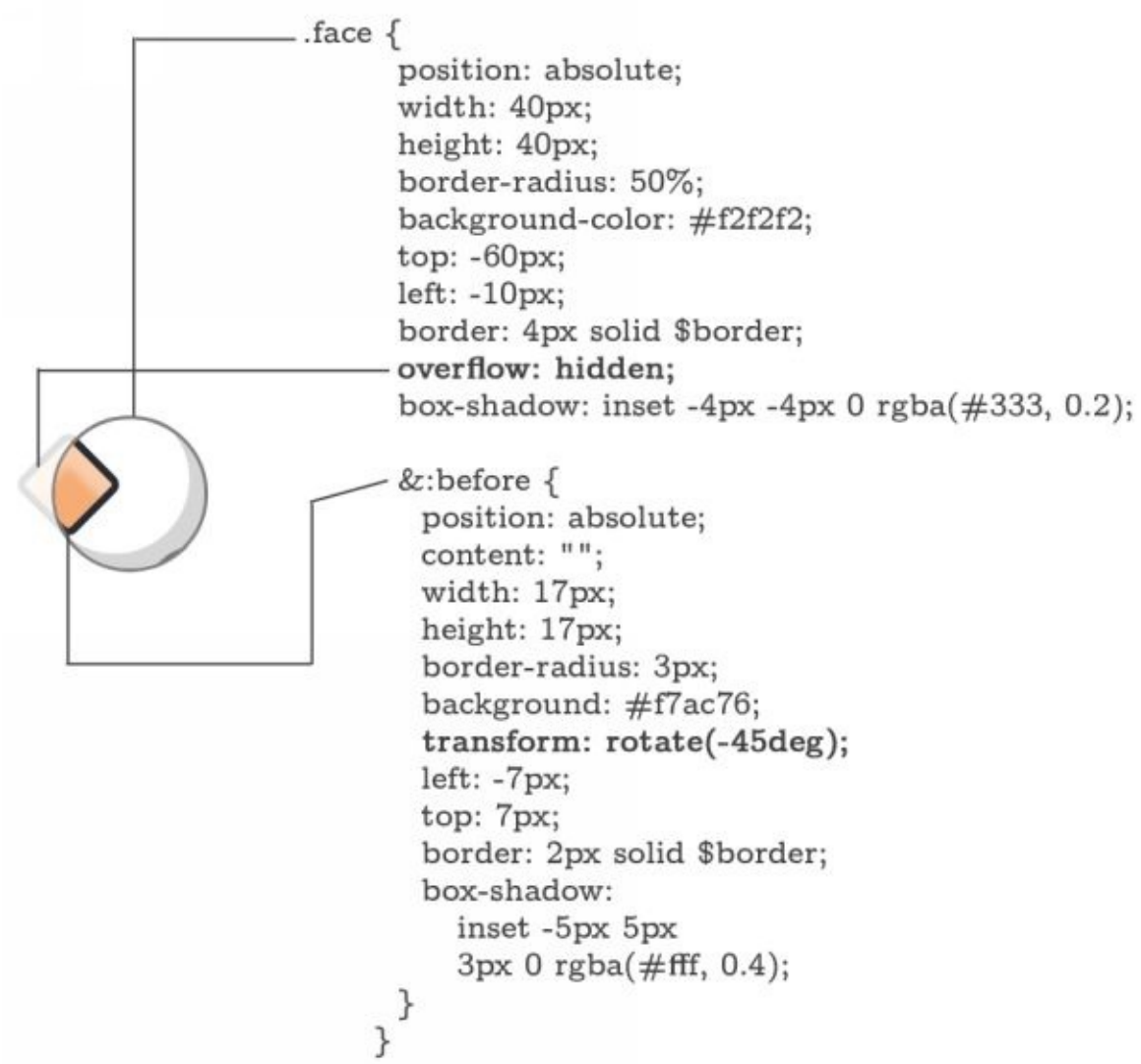
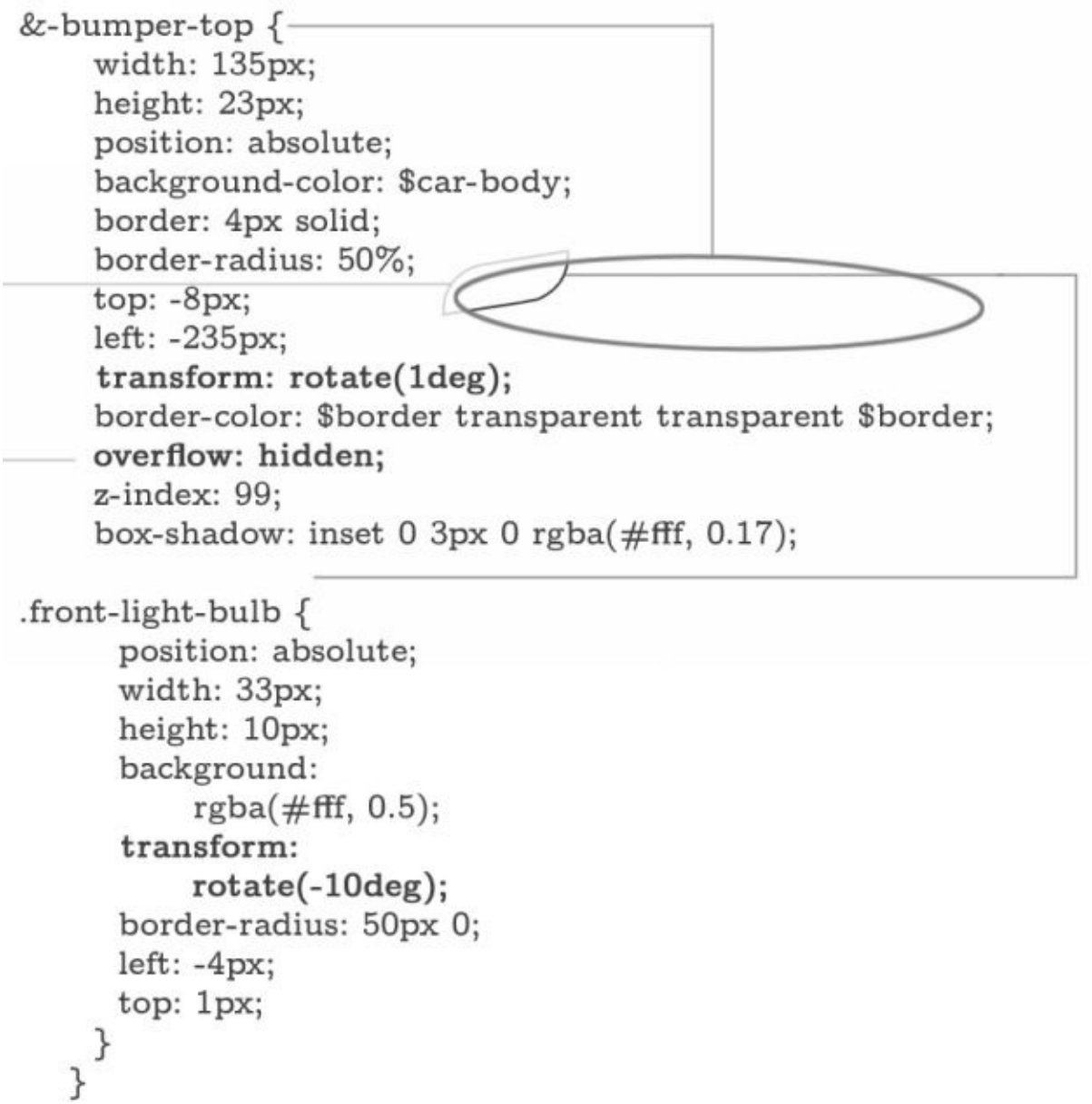



**Figure 224:** The helmet consists of a circle and the orange face shield which is just a *nested*, rotated square with white *inner* box shadow, that cuts off at the radius line because **.face** is set to **overflow:hidden**.

Note how **&:before** is *nested* inside **.face** using { brackets }. This is accomplished by using the SASS extension (Syntactically Awesome Style Sheets). I recommend looking more into it at **http://sass-lang.com**. It is also briefly discussed at the very beginning of this book.

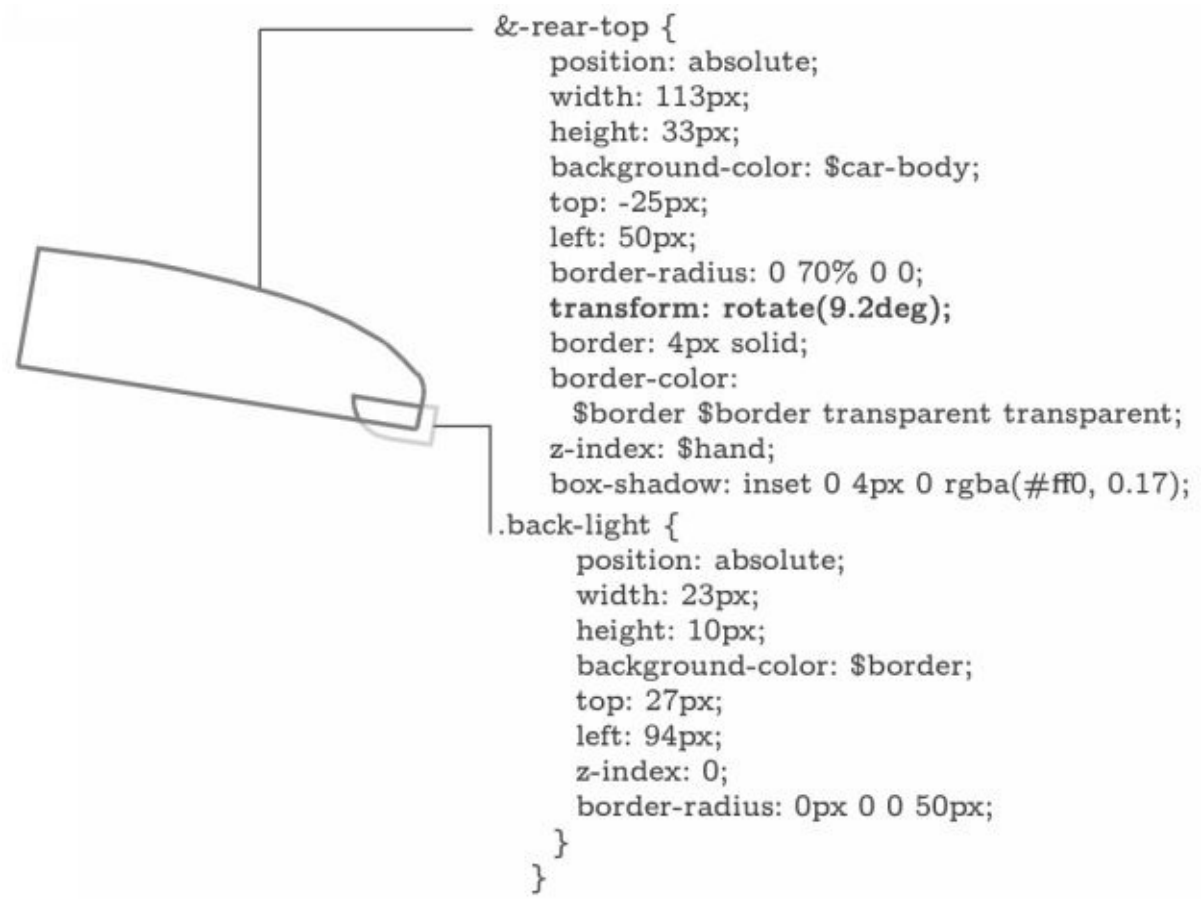
Of course you can still rewrite this in standard vanilla CSS by replacing **&:before** and the brackets by a separate element with its own ***id*** or ***class***.



```
&-bumper-top {
      width: 135px;
      height: 23px;
      position: absolute;
      background-color: $car-body;
      border: 4px solid;
      border-radius: 50%;
      top: -8px;
      left: -235px;
      transform: rotate(1deg);
      border-color: $border transparent transparent $border;
      overflow: hidden;
      z-index: 99;
      box-shadow: inset 0 3px 0 rgba(#fff, 0.17);

.front-light-bulb {
        position: absolute;
        width: 33px;
        height: 10px;
        background:
            rgba(#fff, 0.5);
        transform:
            rotate(-10deg);
        border-radius: 50px 0;
        left: -4px;
        top: 1px;
      }
    }
```

caption{The hood is a long oval element rotated by just 1 degree. In the same way as the helmet's face shield, the light bulb is hidden within the parent by using **overflow:hidden**
. Hiding the overflow is what helps us get away with creating more complex, irregular shapes that closely describe real-life objects.}



```
&-rear-top {
    position: absolute;
    width: 113px;
    height: 33px;
    background-color: $car-body;
    top: -25px;
    left: 50px;
    border-radius: 0 70% 0 0;
    transform: rotate(9.2deg);
    border: 4px solid;
    border-color:
      $border $border transparent transparent;
    z-index: $hand;
    box-shadow: inset 0 4px 0 rgba(#ff0, 0.17);
.back-light {
      position: absolute;
      width: 23px;
      height: 10px;
      background-color: $border;
      top: 27px;
      left: 94px;
      z-index: 0;
      border-radius: 0px 0 0 50px;
    }
  }
```

**Figure 225:** The importance of **overflow:hidden** in creating CSS art cannot be understated. The back light is using absolutely the same technique as the previous two examples. The back of the car is a *rotated*rectangle with just one of the corners rounded. Here you just have to follow your artist's instinct, in order to create shapes that match your preference and a sense of style.

```
&-fender {
    position: absolute;
    top: -2px;
    left: -100px;
    width: 260px;
    height: 65px;
    border-radius: 30px 20px 40px 20px;
    background-color: #ce4038;
    border: 4px solid;
    border-color: $border;
    z-index: $car-rear;
    overflow: hidden;
    box-shadow: inset 0 4px 0 rgba(#fff, 0.17),
      inset -5px -4px 0 rgba(#333, 0.2);
```

**Figure 226:** The base of the car that stretches toward its back is simply a large rectangular **div** element with rounded corners and an *inner* **box-shadow**.

```
&-tire {
      .front,.rear {
        width: 60px;
        height: 60px;
        background: $border;
        position: absolute;
        border-radius: 50%;
        top: 22px;
        z-index: $tire;
        display: flex;
        justify-content: center;
        align-items: center;

        &:before {
          position: absolute;
          width: 60px;
          height: 60px;
          content: "";
          border: 5px solid #333;
          opacity: 0.2;
          border-radius: 50%;
        }
      }
    }
```

**Figure 227:** Again using SCSS

And there you have it! I've only talked about the key CSS properties often used to create CSS art. To avoid redundancy some of the most obvious ones were skipped. For example, it is assumed you already know how to use **top**, **left**, **width** and **height** properties.

To see the original CSS code for the Tesla on **codepen.io** visit the following URL: **https://codepen.io/sashatran/pen/gvVWKJ**