Web Design and Programming

Full reference material

For Class 3

Dr. Peeraya Sripian

8 May 2024

Adapted from Robbins, J. N. (2007). *Learning Web Design: A Beginner's Guide to (X) HTML, StyleSheets, and Web Graphics*. " O'Reilly Media, Inc.".

Table of content

- CSS Introduction
- Adding Styles to the Document
- Document Structure
- The cascade property
- CSS Unit of measurement
- Formatting text
- Specificity
- Colors and Background
- Box model
- CSS Layout with FLEXBOX
- CSS Layout with GRID


CSS
Introduction

CSS

- W3C standard for defining the presentation of documents written in HTML

- A separate language with its own syntax.

- HTML – document structure
- CSS – Presentation (how it is delivered to the user, whether on a computer screen, cell phone, printed on paper, etc)

The Benefits of CSS

- Precise type and **layout control**
- **Less work:** Change look of the whole site with one edit
- **Accessibility:** Markup stays semantic
- **Flexibility:** The same HTML markup can be made to appear in dramatically different ways

Style Separate from Structure

- These pages have the exact same HTML source but different style sheets:

(csszengarden.com)

How Style Sheets Work

1. Start with a marked up document (like HTML, but could be another XML markup language).
2. Write styles for how you want elements to look using CSS syntax.
3. Attach the styles to the document (there are a number of ways).
4. The browser uses your instructions when rendering the elements.

Style Rules

Each rule *selects* an element and *declares* how it should display.

```
h1 { color: green; }
```

```
<h1>This is H1</h1>
```

This rule selects all **`h1`** elements and declares that they should be green.

This is H1

```
strong { color: red; font-style: italic; }
```

```
<p>I wrote up a streamlined adaptation of his recipe that requires <strong>(strong) much</strong> less time and serves 6 people instead of <em>five</em>times that amount.</p>
```

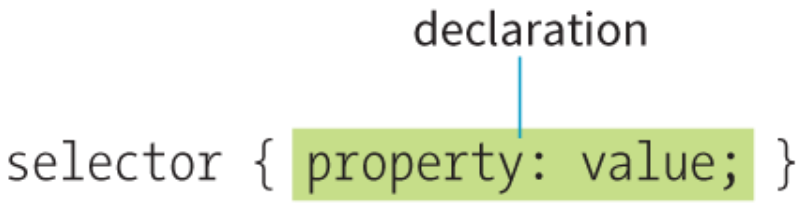
This rule selects all `strong` inline elements and declares that they should be red and in an italic font.

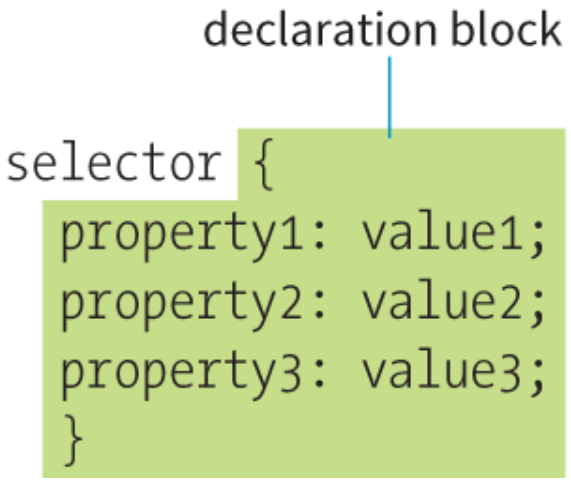
I wrote up a streamlined adaptation of his recipe that requires ***(strong) much*** less time and serves 6 people instead of *five*times that amount.

Style Rule Structure

- A style rule is made up of a selector a declaration.

- The declaration is one or more property / value pairs.





Selectors

- There are many types of selectors. Here are just two examples:

```
p {property: value;}
```

Element type selector: Selects all elements of this type (`p`) in the document.

```
#intro {property: value}
```

ID selector (indicated by the `#` symbol) selects by ID value. In the example, an element with an **id of "intro"** would be selected.

Declarations

- The **declaration** is made up of a **property/value pair** contained in curly brackets **{ }:**
  - **selector {property: value;}**

- **Example**

```
h2 { color: red;
     font-size: 2em;
     margin-left: 30px;
     opacity: .5;
   }
```

This is H2

Declarations (cont’d)

- End each declaration with a semicolon to keep it separate from the next declaration.
- White space is ignored, so you can stack declarations to make them easier to read.
- **Properties** are defined in the CSS specifications.
- **Values** are dependent on the type of property:
  - Measurements
  - Color values
  - Keywords
  - More

CSS Comments

```
/* comment goes here */
```

- Content between `/*` and `*/` will be ignored by the browser.
- Useful for leaving notes or section label in the style sheet.
- Can be used within rules to temporarily hide style declarations in the design process.

```
<style>
/*This is how you write comment in CSS
<style> tag or .css file*/
  h1
  {
    color:green;
  }

  h2{
    color:red;
    font-size: 2em;
    margin-left: 30px;
    opacity: .5;
  }
  strong{
    color:red; font-style:italic;
  }
</style>
```

Adding Styles to the Document

- There are three ways to attach a style sheet to a document:

- **External style sheets**

  A separate, text-only *.css* file associated with the document with the `link` element or `@import` rule

- **Embedded style sheets**

  Styles are listed in the `head` of the HTML document in the `style` element.

- **Inline styles**

  Properties and values are added to an individual element with the `style` attribute.

External Style Sheets

- Store styles in a separate ***.css*** file and attach to the document via **`<link>`** or **`@import`**
- Most efficient method: Change styles in multiple documents by editing one *.css* file
- A *.css* document is a simple text document (may begin with **`@charset`** to identify character set)

menu.css

```
@charset "UTF-8"

body { font-family: Georgia, serif;
  font-size: 100%;
  line-height: 175%; }

h1 { font-size: 1.5em;
   color: purple;}

dt { font-weight: bold; }

strong { font-style: italic; }

h2 { font: bold 1em Georgia, serif;
  text-transform: uppercase;
  letter-spacing: 8px;
  color: purple;}

dt strong { color: maroon; }

header p { font-style: italic; color: gray;}

header, h2, #appetizers p { text-align: center; }

#appetizers p, { font-style: italic; }

.price { font-style: italic;
  font-family: Georgia, serif; }

.label { font-weight: bold;
  font-variant: small-caps;
  font-style: normal; }

p.warning, sup { font-size: x-small;
  color: red; }
```

Attaching a Style Sheet with the link Element

- The **`link`** element defines a relationship between the current document and an external resource.
- It goes in the **`head`** of the document.
- Use the **`rel`** attribute to say it's a style sheet. Use **`href`** to provide the URL of the *.css* file (relative to the current document):

```
<head>
  <title>Titles are required.</title>
  <link rel="stylesheet" href="/path/stylesheet.css">
</head>
```

Attaching a Style Sheet with an @import rule

- An **`@import`** rule imports the contents of an external style sheet into another style sheet (either a *.css* document or embedded with **`style`**):

```
<head>
  <style>
     @import url("/path/stylesheet.css");
     p { font-face: Verdana;}
  </style>
  <title>Titles are required.</title>
</head>
```

External Style Sheets

- The style rules are saved in a separate text-only *.css* file and attached via **`link`** or **`@import`**.

Via **`link`** element in HTML:

```
<head>
  <title>Titles are require</title>
  <link rel="stylesheet"
href="/path/example.css">
</head>
```

Via **`@import`** rule in a style sheet:

```
<head>
  <title>Titles are required</title>
  <style>
    @import url("/path/example.css");
    p {font-face: Verdana;}
  </style>
</head>
```

Inline Styles

- Apply a style declaration to a single element with the **`style`** *attribute*:

```
<p style="font-size: large;">Paragraph text...</p>
```

To add multiple properties, separate them with semicolons:

```
<h3 style="color: red; margin-top: 30px;">Intro</h3>
```

Embedded Style Sheets

- Embedded style sheets are placed in the **`head`** of the document via the **`style`** element:

```
<head>
  <title>Titles are required</title>
  <style>
    /* style rules go here */
  </style>
</head>
```

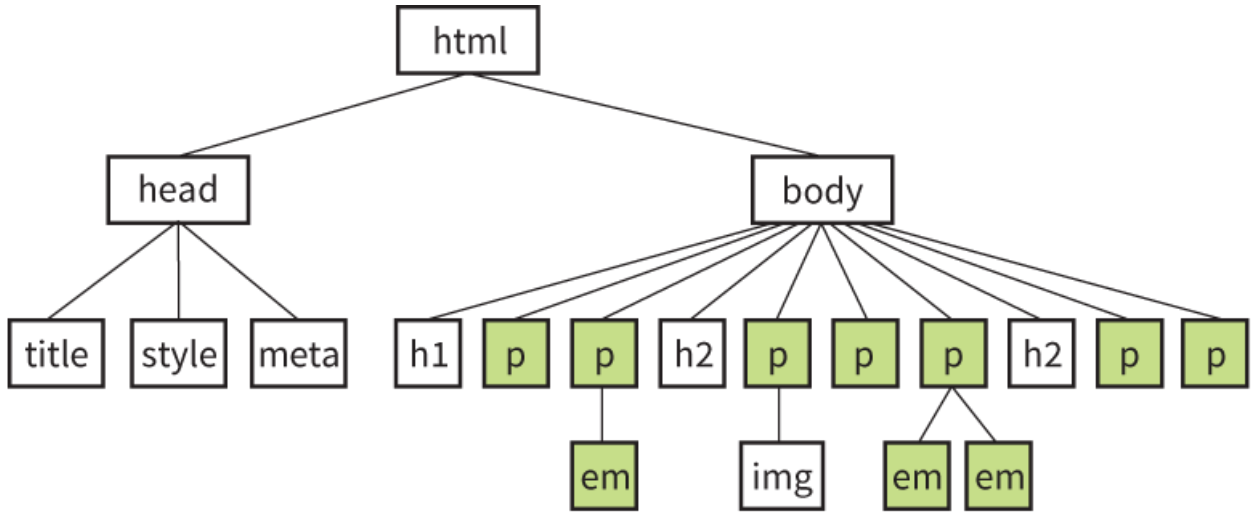
Document Structure

- Documents have an implicit structure.
- We give certain relationships names, as if they’re a family:
- All the elements contained in a given element are its descendants.
- An element that is directly contained within another element is the child of that element.
- The containing element is the parent of the contained element.
- Two elements with the same parent are siblings.

Inheritance

- Many properties applied to elements are passed down to the elements they contain. This is called inheritance.

- For example, applying a sans-serif font to a `p` element causes the `em` element it contains to be sans-serif as well:




```
p {font-size: large; font-family: sans-serif;}
```

Inheritance (cont'd)

- **Some properties inherit; others do not.**
  Properties related to text usually inherit; properties related to layout generally don't.

- Styles explicitly applied to specific elements override inherited styles.

- You'll learn to use inheritance strategically to keep your style rules simple.

The cascade property

It is called “cascade” for a reason!

The Cascade

- The **cascade** refers to the system for resolving conflicts when several styles apply to the same element.
- Style information is passed down (it “cascades” down) until overwritten by a style rule with more **weight**.
- Weight is considered based on:
    - **Priority** of style rule source
    - **Specificity** of the selector
    - **Rule order**

The Cascade: Priority

- Style rules from sources higher in this list override rules from sources listed below them.

1. Any style marked as **!important** by the **user** (to accommodate potential accessibility settings)
2. Any style marked **!important** by the **author** (of the web page)
3. Author styles (style sheets created in web site production)
4. User styles (added by the reader)
5. User agent styles (browser defaults)

The Cascade: Specificity

- When two rules in a single style sheet conflict, the **type of selector is used to determine which rule has more weight**.

- For example, ID selectors are more specific than general element selectors.

The Cascade: Rule Order

- When two rules have equal weight, rule order is used. **Whichever rule appears last "wins."**

```
<style>
  p {color: red;}
  p {color: blue;}
  p {color: green;}
</style>
```

In this example, paragraphs would be green.

- Styles may come in from external style sheets, embedded style rules, and inline styles. The style rule that gets parsed last (the one closest to the content) will apply.

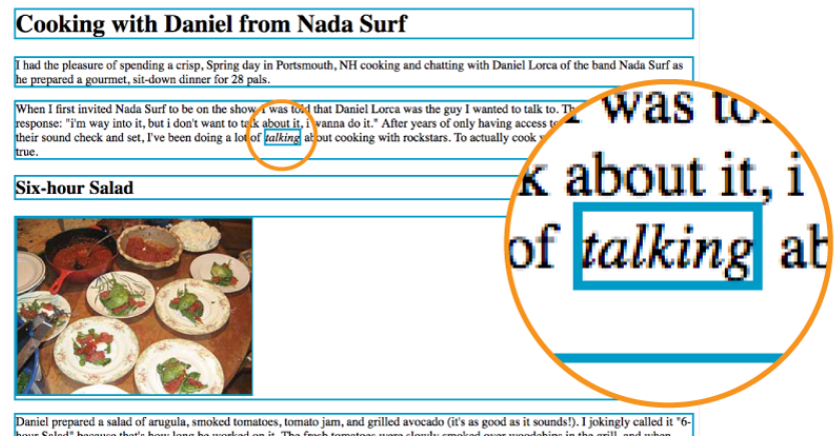
The Box Model

- Browsers see every element on the page as being contained in a little rectangular box. **Block elements and inline elements participate in the box model.**
- In this example, a blue border is added to all elements.

```
h1 { border: 1px solid blue; }
h2 { border: 1px solid blue; }
p  { border: 1px solid blue; }
em { border: 1px solid blue; }
img{ border: 1px solid blue; }
```

Equivalent to

```
h1, h2, p, em, img { border: 1px solid blue; }
```

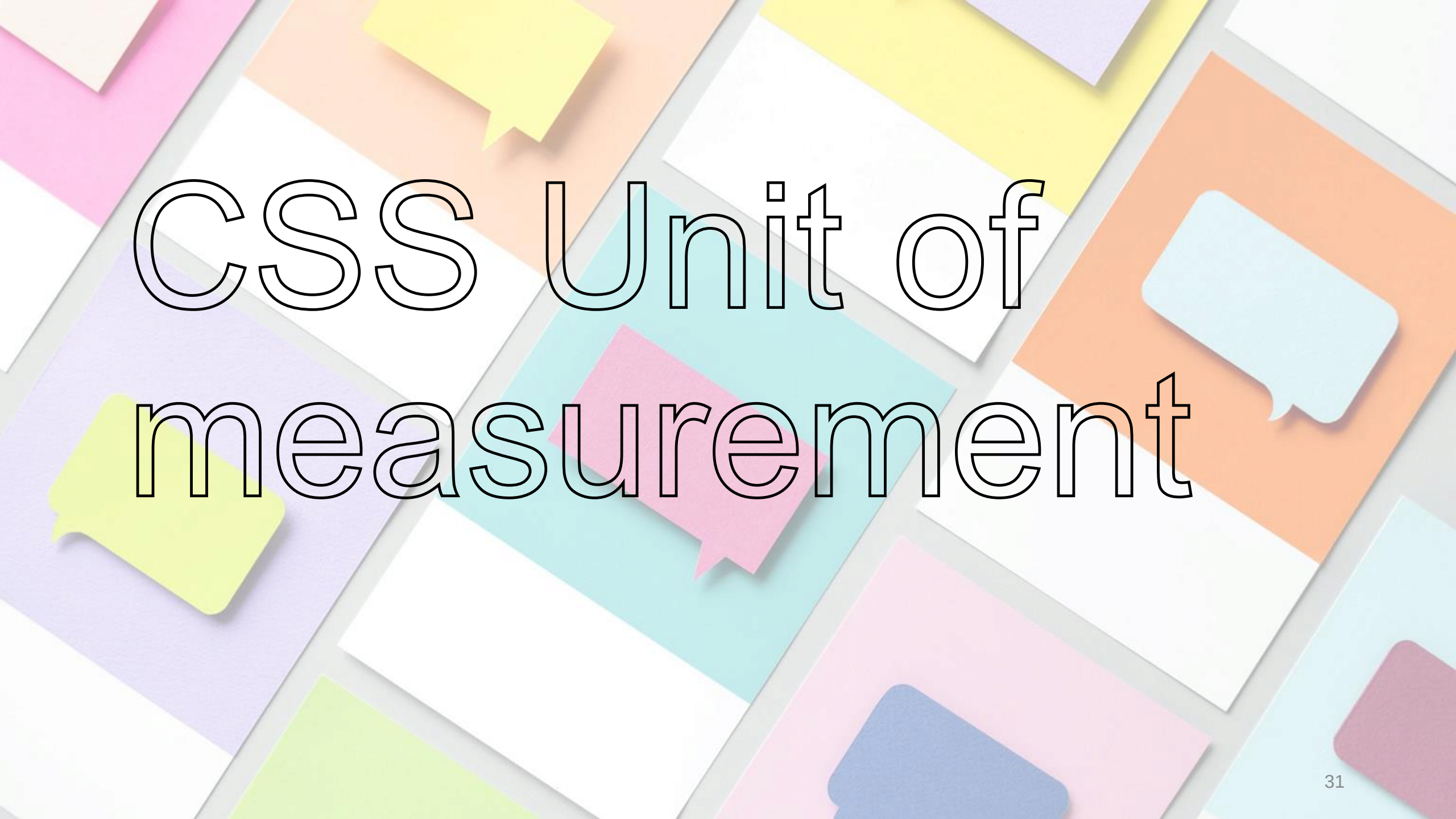


Grouped Selectors

Note: Border is one of the properties that are not inherited

The Box Model (cont'd)

- The **box model** is the foundation of CSS page layout.
- Apply properties such as **borders**, **margins**, **padding**, and **backgrounds** to element boxes.
- Position, move, grow, and shrink boxes to create fixed or flexible page layouts.


CSS Unit of measurement

Full details on CSS Values and Units Module
http://www.w3.org/TR/css3-values/

CSS Units of Measurement

- CSS provides a variety ways to specify measurements:

Absolute units

- Have predefined meanings or real-world equivalents

Relative units

- Based on the size of something else, such as the default text size or the size of the parent element

Percentages

- Calculated relative to another value, such as the size of the parent element

Absolute Units

- With the exception of pixels, absolute units are not appropriate for web design:

`px`	pixel
`in`	inches
`mm`	millimeters
`cm`	centimeters
`q`	1/4 millimeter
`pt`	points (1/72 inch)
`pc`	pica (1 pica = 12 points = 1/6 inch)

Relative Units

Good for most web measurements

- Relative units are based on the size of something else:

em	a unit equal to the current font size
ex	x-height, equal to the height of a lowercase *x*
rem	root em, equal to the font size of the `html` element
ch	zero width, equal to the width of a zero (0)
vw	viewport width unit (equal to 1/100 of viewport width)
vh	viewport height unit (1/100 of viewport height)
vmin	viewport minimum unit (value of `vh` or `vw`, whichever is smaller)
vmax	viewport maximum unit (value of `vh` or `vw`, whichever is larger)

RELATIVE UNITS

The rem Unit

- The **`rem`** **(root em)** unit is based on the font size of the root **`html`** element, whatever that happens to be.

- **Default** in modern browsers: Root font size is 16 pixels, so a **`rem`** **= a 16-pixel unit,** **`10rem`** **= 160 px (for default)**

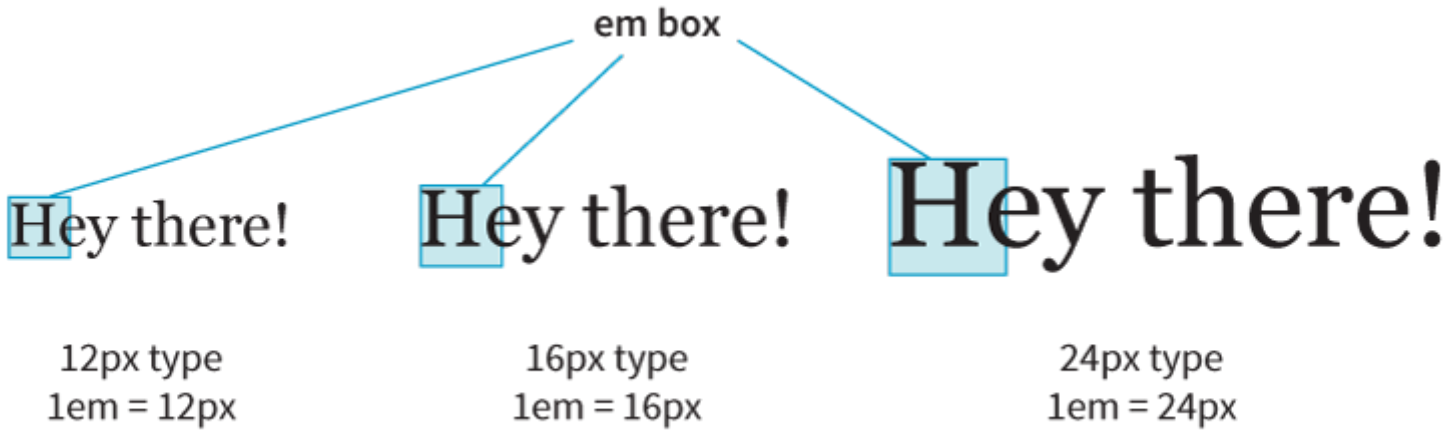
- If the root font size of the document changes, so does the size of a rem (and that's good for keeping elements proportional).

- EX: if a user changes the base font size (in his browser) to 24 px, **`10rem`** would be 240px

RELATIVE UNITS

The em Unit

- The `em` unit is traditionally based on the width of a capital letter *M* in the font.
- When the font size is 16 pixels,1em = 16 pixels, 2em = 32 pixels, and so on.



NOTE: Because they're based on the font size of the current element, the size of an em may not be consistent across a page.

RELATIVE UNITS

Viewport Percentage Lengths (vw/vh)

- Viewport width (**vw**) and viewport height (**vh**) units are relative to the size of the viewport (browser window):
  - **vh** = 1/100th width of viewport
  - **vh** = 1/100th height of viewport

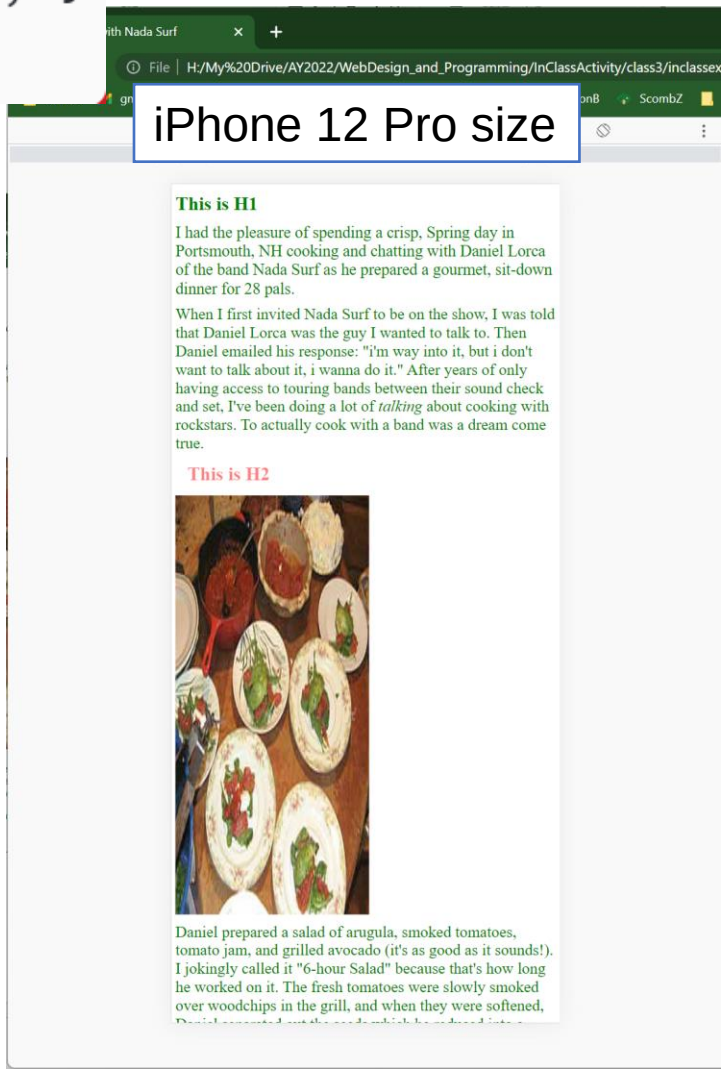
- They’re useful for making an element fill the viewport or a specified percentage of it. This image will be 50% the width and height of the viewport:

```
img { width: 50vw; height: 50vh; }
```

```
img { width: 50vw; height: 50vh; }
```

Regular browser in PC screen

iPhone 12 Pro size



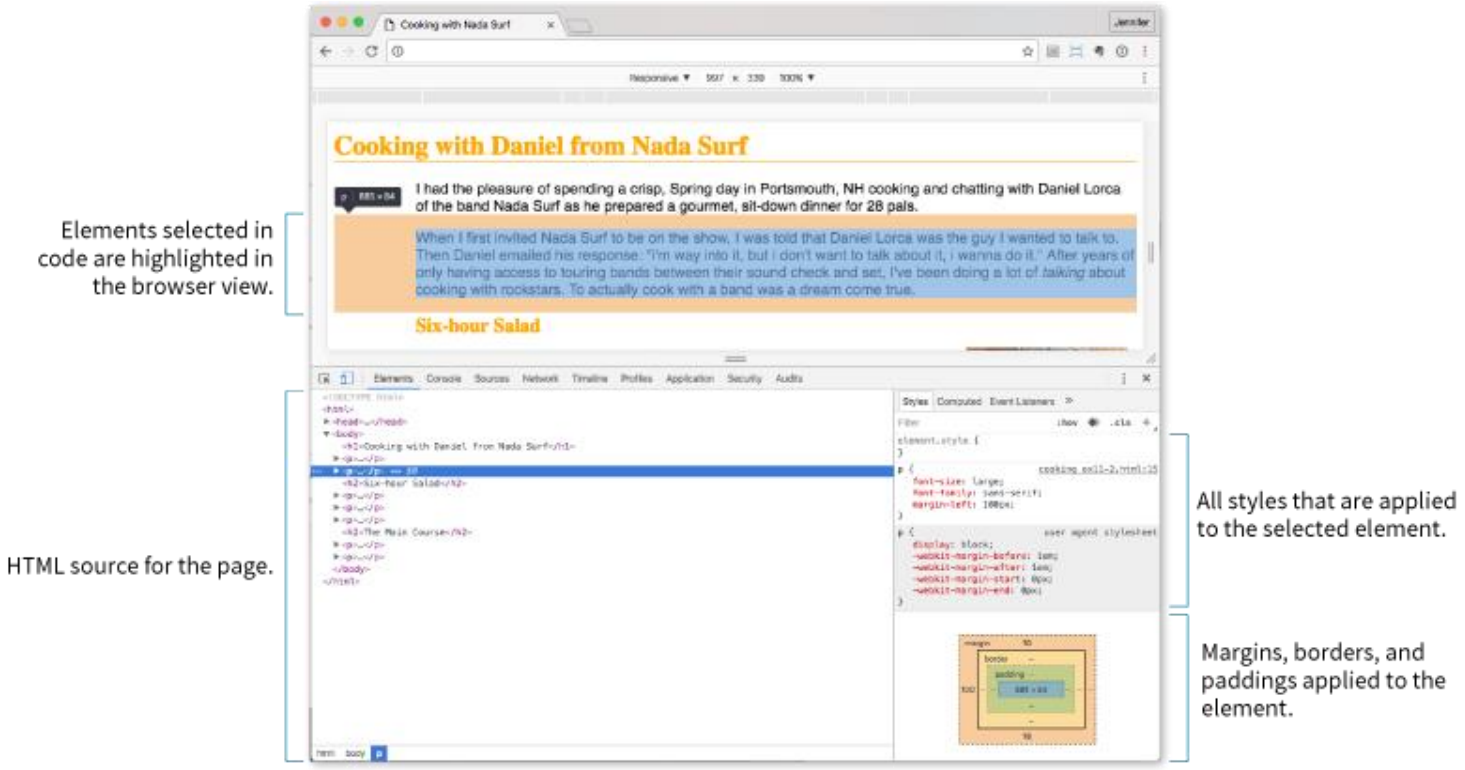
Browser Developer Tools

Major browsers have built-in tools that aid development:

- HTML, CSS, and JavaScript inspectors
- Network speed reports
- Animation tools
- Other helpful features

Browser Developer Tools (cont'd)

Chrome DevTools (View > Developer > Developer Tools)



Firefox, Safari, Opera, and Microsoft Edge also have developer tools.

Formatting text

Designing Text

- Styling text on the web is tricky because you don’t have control over how the text displays for the user:
  - They may not have the font you specify.
  - They may have their text set larger or smaller than you designed it.

- Best practices allow for flexibility in text specification.

Typesetting Terminology

- A **typeface** is a set of characters with a single design (example: `Garamond`).
- A **font** is a particular variation of the typeface with a specific weight, slant, or ornamentation (example: `Garamond Bold Italic`).
- In traditional metal type, each size was a separate font (example: `12-point Garamond Bold Italic`).
- On a computer, fonts are generally stored in individual font files.

CSS Basic Font Properties

- CSS font properties deal with specifying the shapes of the characters themselves:
  - `font-family`
  - `font-size`
  - `font-weight`
  - `font-style`
  - `font-variant`
  - `font` (a shorthand that includes settings for all of the above)

Specifying the Font Family

`font-family`

Values: One or more font family names, separated by commas

Example:

```
body { font-family: Arial; }
var { font-family: Courier, monospace; }
```

Specifying the Font Family (cont'd)

- Font names must be capitalized (except generic font families).
- Use commas to separate multiple font names.
- If the name has a character space, it must appear within quotation marks:

```
p { font-family: "Duru Sans", Verdana, sans-serif; }
```

Using Fonts on the Web

- The font must be available on the user's machine for it to display.
- The best practice is to provide a list of options. The browser uses the first one that is available.
- Start with the font you want and then provide backup options ending with a generic font family, as shown here:

```
p { font-family: "Duru Sans", Verdana, sans-serif; }
```

- You can also download a web font with the page, but it adds to the download and display time.

Generic Font Families

- Generic font families instruct the browser to use an available font from one of five stylistic categories: **serif**, **sans-serif**, **monospace**, **cursive**, **fantasy**
- Generic font families are often used as the last backup option.

Generic Font Families (cont'd)

Specifying Font Size

`font-size`

Values:

- CSS length units
- Percentage value
- Absolute keywords (`xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`)
- Relative keywords (`larger`, `smaller`)

Example:

```
h1 { font-size: 2.5rem; }
```

Specifying Font Size (cont'd)

- Most common sizing methods:
  - `rem` and `em` units
  - `percentages` (based on the inherited font size for that element)
  - `pixels` (px) can be used, but they're not flexible.

See also CSS Unit of measurement

Font Size: rem Units

- The `rem` (**root em**) is equal to the font size of the `html` (root) element.
- In browsers, the **default** root size is 16 pixels, so: **1 rem = 16 pixels**.
- If the font size of the root is changed, rem measurements change too.
- !!! Old browsers *do not* support rem units (IE8 and earlier).

See also CSS Unit of measurement

Font Size: em Units

- The `em` **unit** is based on the current font size of the element.
- The default font size is 16 pixels. **By default,** `1em` **= 16 pixels.**
- But if you change the font size of the element, the size of its em unit changes too.
- Ems may be different sizes in different parts of the document and may compound larger or smaller when elements are nested.
- This makes ems a little tricky to use, although they are better supported than rem units.

Font Weight (Boldness)

`font-weight`

Values: `normal`, `bold`, `bolder`, `lighter`, `100`, `200`, `300`, `400`, `500`, `600`, `700`, `800`, `900`

Example:

```
h1 { font-weight: normal; }
span.new { font-weight: bold; }
```

- Most common values are normal and bold.
- Numerical values are useful when using a font with multiple weights.

Font Style (Italics)

`font-style`

Values: `normal`, `italic`, `oblique`

Example:

```
cite { font-style: italic; }
```

- Makes text italic, normal, or oblique (slanted, but generally the same as italics).

Small Caps

`font-variant`

Values (in CSS2.1): `normal`, `small-caps`

Example:

```
abbr { font-variant: small-caps; }
```

- Small caps are a separate font design that uses small uppercase characters in place of lowercase letters.
- They help acronyms and other strings of capital letters blend in with the weight of the surrounding text.

Condensed and Extended Text

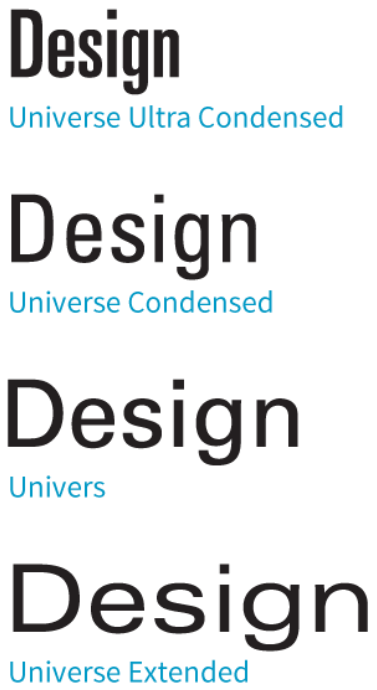
`font-stretch`

Values (in CSS2.1): `normal`, `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `semi-expanded`, `expanded`, `extra-expanded`, `ultra-expanded`

Example:

```
abbr { font-variant: small-caps; }
```

- Tells the browser to select a normal, condensed, or extended font variation from a typeface if it is available



The Shortcut font Property

`font`

Values (in CSS2.1): A list of values for all the individual properties, in this order:

```
{font: style weight stretch variant size/line-height font-family}
```

At minimum, it must contain **`font-size`** and **`font-family`**, in that order. Other values are optional and may appear in any order prior to **`font-size`**.

Example:

```
p { font: 1em sans-serif; }
h3 { font: oblique bold small-caps 1.5em Verdana, sans-serif; }
```

Advanced Typography

Font-variant- XXX
Access to special characters (glyphs) in fonts

The **CSS3 Font Module** offers properties for fine-tuned typography control, including:

- Ligatures
- Superscript and subscript
- Alternate characters (such as a swash design for an *S*)
- Proportional font sizing using x-height
- Kerning
- OpenType font features

Text Line Treatments

Some properties control whole lines of text:

- Line height (`line-height`)
- Indents (`text-indent`)
- Horizontal alignment (`text-align`)

Line Height

`line-height`

Values: *`number`*, *`length`*, *`percentage`*, `normal`

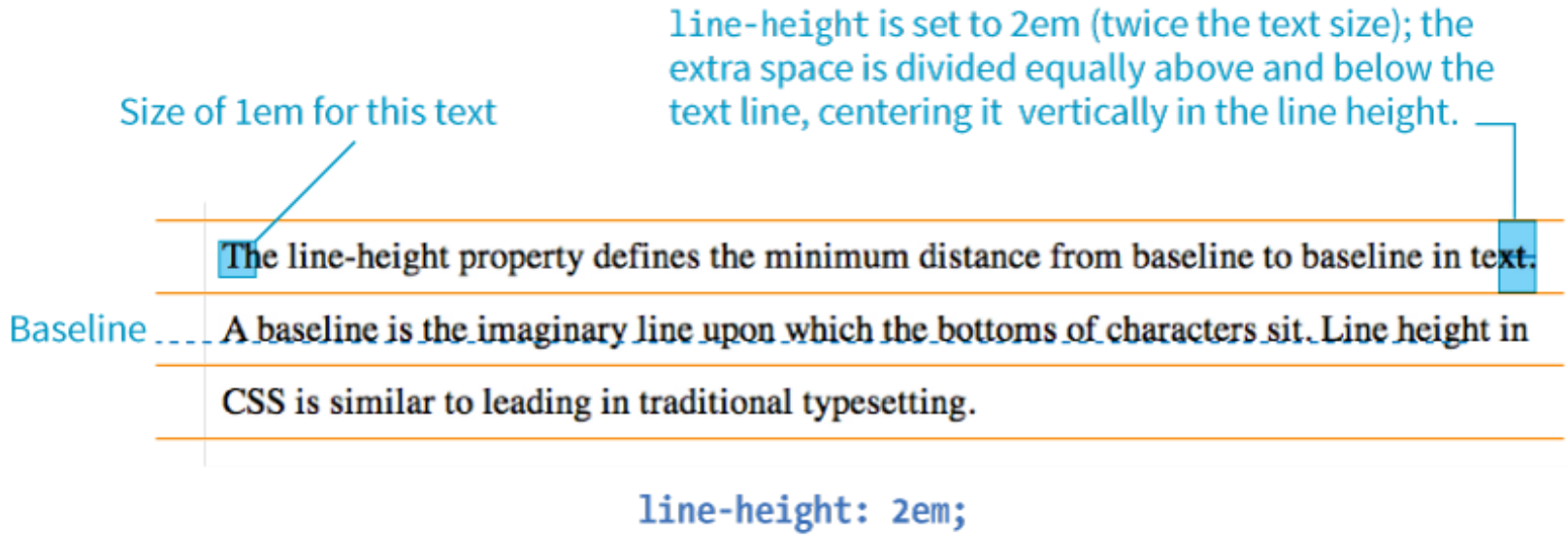
Example:

```
p { line-height: 1.4em; }
```

- Line height defines the minimum distance from baseline to baseline in text.

Line Height (cont'd.)

- The **baseline** is the imaginary line upon which the bottoms of characters sit.
- If a large character or image is on a line, the line height expands to accommodate it.



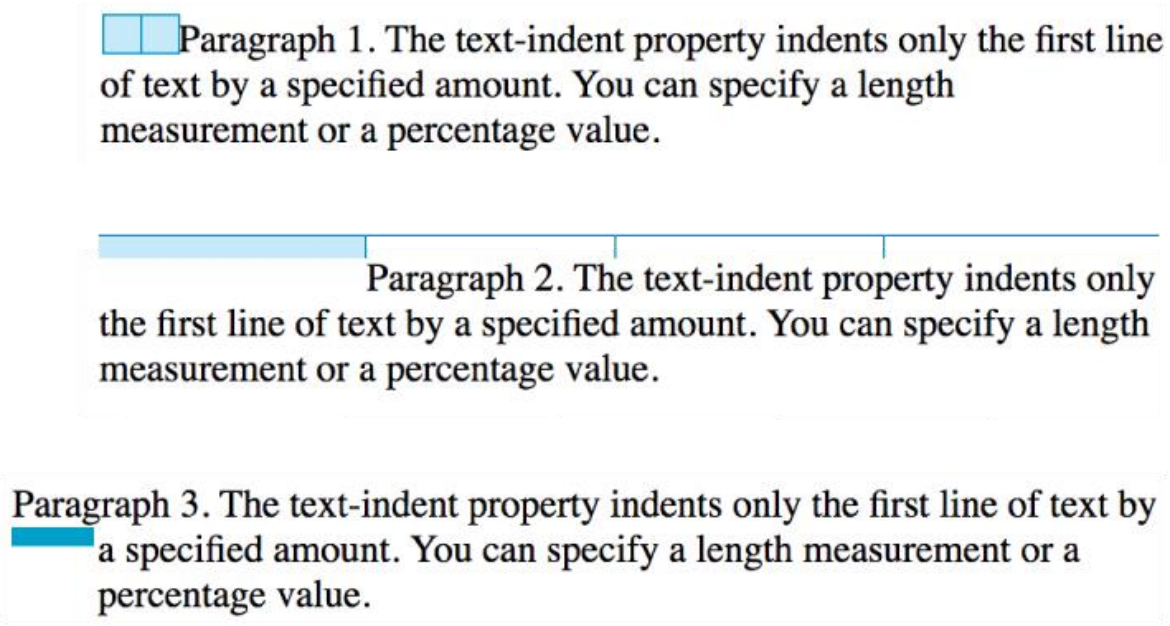
Indents

`text-indent`

Values: *`Length`*, *`percentage`*

Examples:

```
p {text-indent: 2em; }
p {text-indent: 25%; }
p {text-indent: -35px; }
```



Horizontal Text Alignment

`text-align`

Values: `left`, `right`, `center`, `justify`, `start`, `end`

Examples:

```
text-align: left;
```

Paragraph 1. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page. The resulting text behavior of the various values should be fairly intuitive.

```
text-align: right;
```

Paragraph 2. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.

```
text-align: center;
```

Paragraph 3. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.

```
text-align: justify;
```

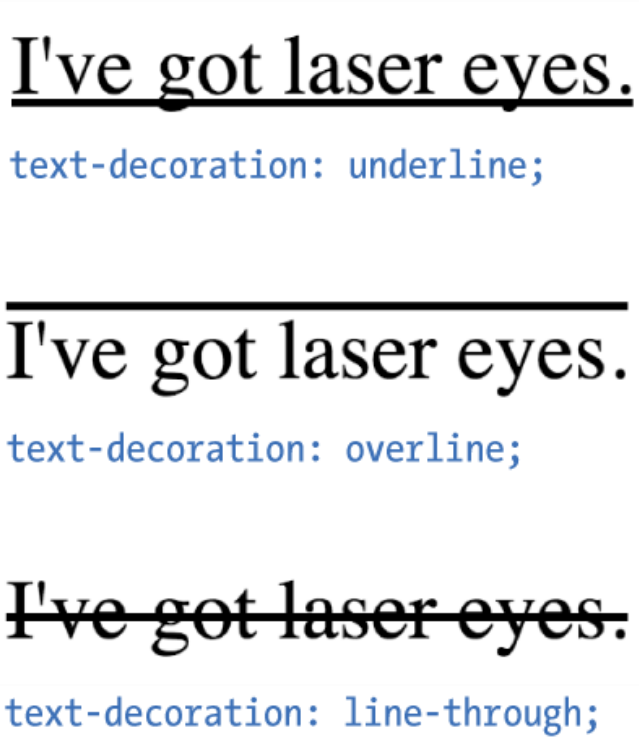
Paragraph 4. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.

Underlines (Text Decoration)

`text-decoration`

Values: `none`, `underline`, `overline`, `line-through`, `blink`

Examples:



NOTE:
`text-decoration` is often used to turn **off** underlines under links:

```
a {
  text-decoration: none;
}
```

Text Decoration Tips

- If you turn off underlines under links, be sure there is another visual cue to compensate.

- Underlining text that is not a link may be misleading. Consider italics instead.

- Don’t use `blink`. Browsers don’t support it anyway.

Capitalization

`text-transform`

Values:

`none`, `capitalize`, `lowercase`, `uppercase`, `full-width`

Examples:

`text-transform: none;` *(as it was typed in the source)*	And I know what you're thinking.
`text-transform: capitalize;`	And I Know What You're Thinking.
`text-transform: lowercase;`	and i know what you're thinking.
`text-transform: uppercase;`	AND I KNOW WHAT YOU'RE THINKING.

Spacing

letter-spacing

Values: *length*, normal

word-spacing

Values: *length*, normal

Examples:

B l a c k G o o s e B i s t r o S u m m e r M e n u

```
p { letter-spacing: 8px; }
```

Black Goose Bistro Summer Menu

```
p { word-spacing: 1.5em; }
```

Text Shadow

`text-shadow`

Values: *'horizontal-offset' 'vertical-offset' 'blur-radius' 'color'*, none

The value is two offset measurements, an optional blur radius, and a color value (with no commas between).

Example:

List Style Properties

There are three properties for affecting the display of lists:

- **`list-style-type`**
  Chooses the type of list marker

- **`list-style-position`**
  Sets the position of the marker relative to the list element box

- **`list-style-image`**
  Allows you to specify your own image for use as a bullet

LIST STYLES

Choosing a Marker

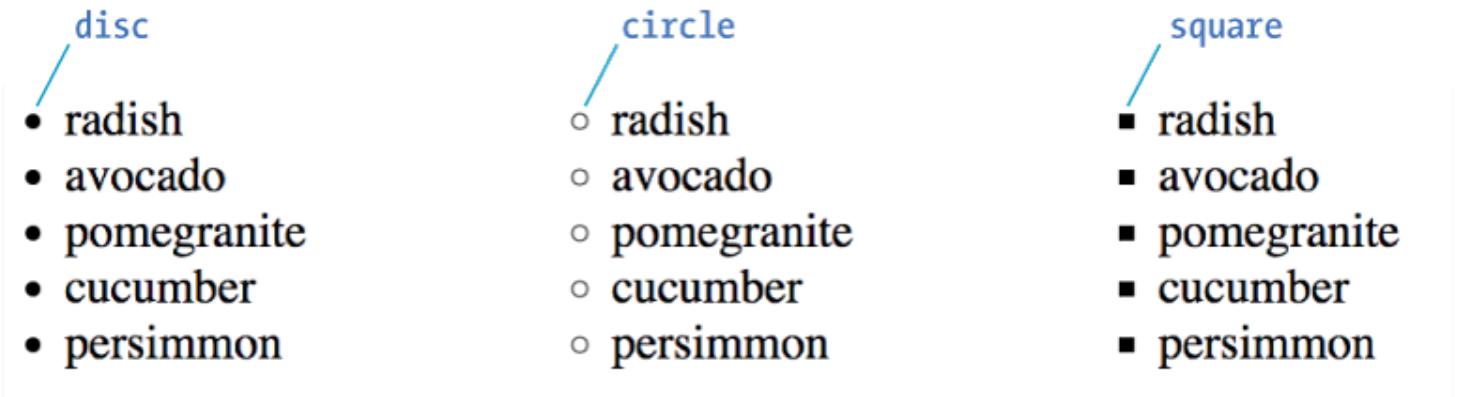
`list-style-type`

Values:

`none`, `disc`, `circle`, `square`, `decimal`, `decimal-leading-zero`, `lower-alpha`, `upper-alpha`, `lower-latin`, `upper-latin`, `lower-roman`, `upper-roman`, `lower-greek`

Unordered lists:

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



LIST STYLES

Choosing a Marker

Ordered lists:

```
ol { list-style-type: keyword; }
```

Keyword	System
`decimal`	1, 2, 3, 4, 5…
`decimal-leading-zero`	01, 02, 03, 04, 05…
`lower-alpha`	a, b, c, d, e…
`upper-alpha`	A, B, C, D, E…
`lower-latin`	a, b, c, d, e… (same as lower-alpha)
`upper-latin`	A, B, C, D, E… (same as upper-alpha)
`lower-roman`	i, ii, iii, iv, v…
`upper-roman`	I, II, III, IV, V…
`lower-greek`	α, β, γ, δ, ε…

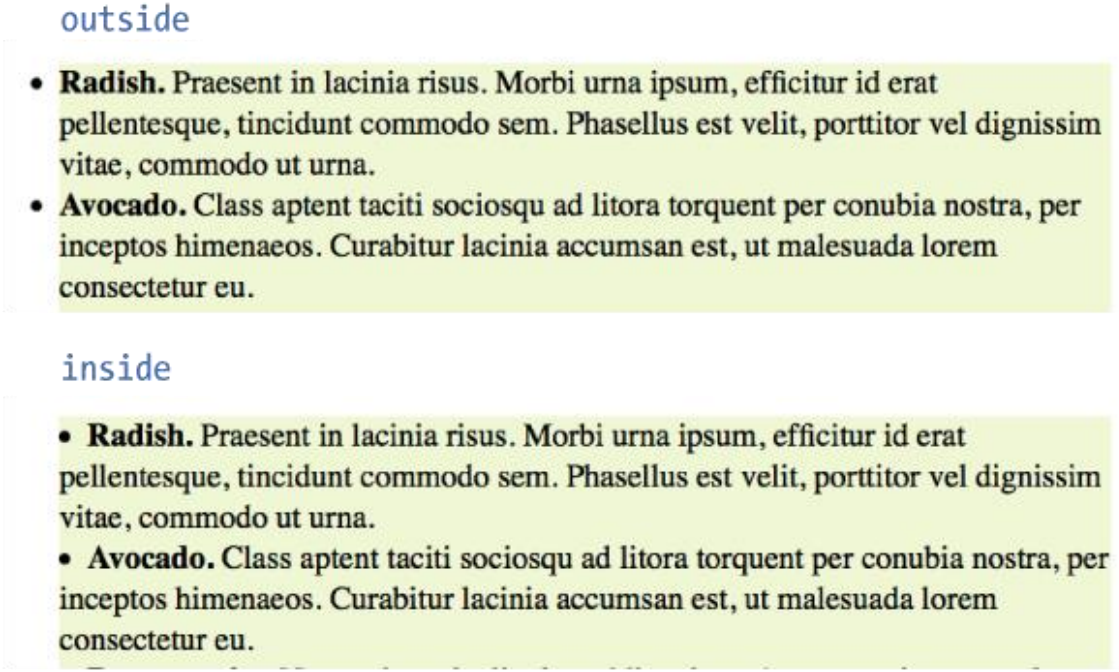
LIST STYLES

Marker Position

`list-style-position`

Values: `inside`, `outside`, `hanging`

Positions the marker relative to the content area:



LIST STYLES

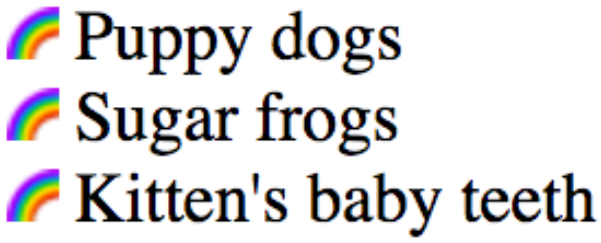
Custom Bullets

`list-style-image`

Values: `url(location)`, `none`

Example:

```
ul {
  list-style-type: disc;
  list-style-image: url(/images/rainbow.gif);
  list-style-position: outside;
}
```



More Selector Types

- Descendent selectors
- ID selectors
- Class selectors
- Universal selector

Descendent Selectors

- A **descendent selector** targets elements contained in another element.
- It's a kind of **contextual selector** (it selects based on relationship to another element).
- It's indicated in **a list separated by a character space**.

```
ol a {font-weight: bold;}
```

(only the links (**a**) in ordered lists (**ol**) would be bold)

```
h1 em {color: red;}
```

(only emphasized text in h1s would be red)

A space between element names means that the 2nd element must be contained within the first

Descendent Selectors (cont'd)

- They can appear as part of a grouped selector:

```
h1 em, h2 em, h3 em {color: red;}
```

(only emphasized text in **h1**, **h2**, and **h3** elements)

- They can be several layers deep:

```
ol a em {font-variant: small-caps;}
```

(only emphasized text in links in ordered lists)

ID Selectors

- **ID selectors** (indicated by a # symbol) target elements based on the value of their ID attributes:

```
<li id="primary">Primary color t-shirt</li>
```

- To target just that item:

```
li#primary {color: olive;}
```

- To omit the element name:

```
#primary {color: olive;}
```

- It can be used as part of a compound or contextual selector:

```
#intro a { text-decoration: none;}
```

Class Selectors

- **Class selectors** (indicated by a **.** symbol) select elements based on the value of their class attributes:

```
p.special { color: orange;}
```

(All paragraphs with the class name "special" would be orange.)

- To target *all* element types that share a class name, omit the element name in the selector:

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

(All elements with the class “hilight” would have a yellow background.)

Universal Selector

- The **universal element selector** (*****) matches any element, like a wildcard in programming languages:

```
* {border: 1px solid gray;}
```

(puts a 1-pixel gray border around every element in the document)

- Can be used as part of contextual selectors:

```
#intro * {border: 1px solid gray;}
```

(selects all elements contained within an element with the ID **`intro`**)

Specificity Basics

- **Specificity** refers to a system for sorting out which selectors have more weight when resolving style rule conflicts.
- **More specific selectors have more weight.**
- In simplified terms, it works like this:
  - **Inline styles** with the `style` attribute are more specific than (and will override…)
  - **ID selectors**, which are more specific than (and will override…)
  - **Class selectors**, which are more specific than (and will override…)
  - **Individual element selectors**

Calculating Specificity

- There is a system used to calculate specificity. Start by drawing three boxes:

[ ] [ ] [ ]

For each style rule:

1. Count the **IDs** in the selector and put that number in the first box.
2. Count the **class** and pseudo-class selectors and put the number in the second box.
3. Count the **element** names and put the number in the third box

[ID] [class] [elements]

4. **The first box that is not a tie determines which selector wins.**

Calculating Specificity (cont'd)

Example:

The second one has a class selector and the first one doesn't, therefore the second one is more specific and has more weight.

The lime color applies to `h1`s when they have the class name "special."

Using Specificity

Use specificity strategically to take advantage of overrides:

```
p { line-height: 1.2em; }                    [0] [0] [1]
```

(sets the line-height for all paragraphs)

```
blockquote p { line-height: 1em; }           [0] [0] [2]
```

(more specific selector changes line-height when the paragraph is in a blockquote)

```
p.intro { line-height: 2em; }                [0] [1] [1]
```

(paragraphs with the class “intro” have a line-height of 2em, even when they’re in a blockquote. A class name in the selector has more weight than two element names.)

Colors and Background

Ways to specify color

- By color names

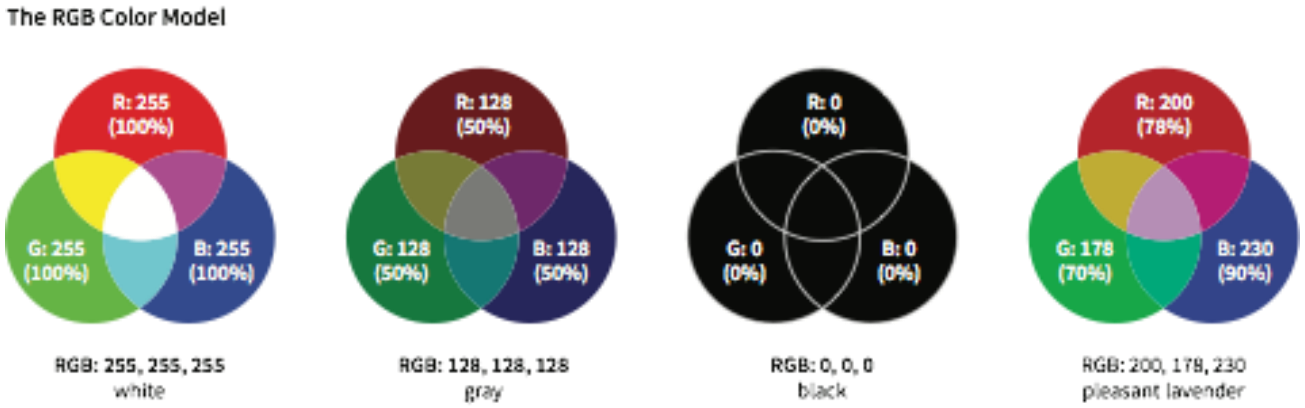
```
h1 { color: red; }
h2 { color: darkviolet; }
body { background-color: papayawhip; }
```

- By numeric values
  - RGB (Red Green Blue)
  - RGBa + Alpha (opacity-transparency)
  - HSL (Hue, Saturation, Lightness)
  - HSLa + Alpha (opacity-transparency)

RGB Color

- The **RGB color model** mixes color with red, green, and blue light.

- Each channel can have 256 shades, for millions of color options.



4 Styles of RGB specification

- RGB values (0 to 255):
  `rgb(200,178,230)`

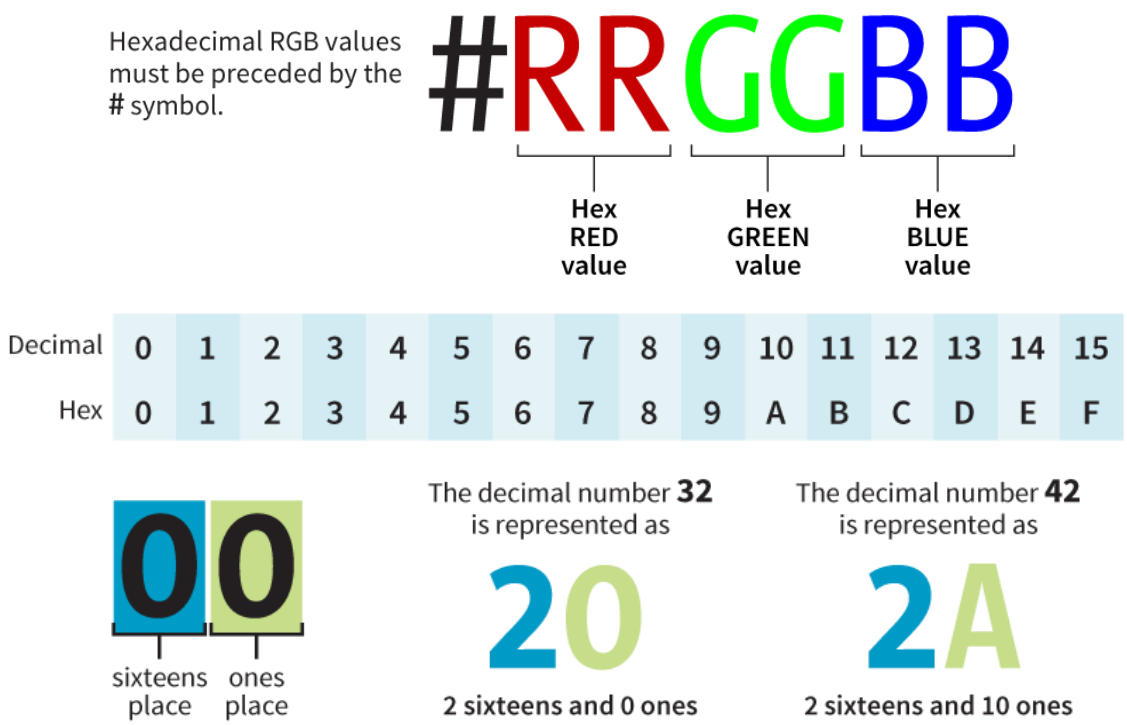
- Percentage values:
  `rgb(78%,70%,90%)`

- Hexadecimal values:
  `#C8B2E6`

- Condensed hexadecimal values (for double-digits only):
  `#F06` is the same as `#FF0066`

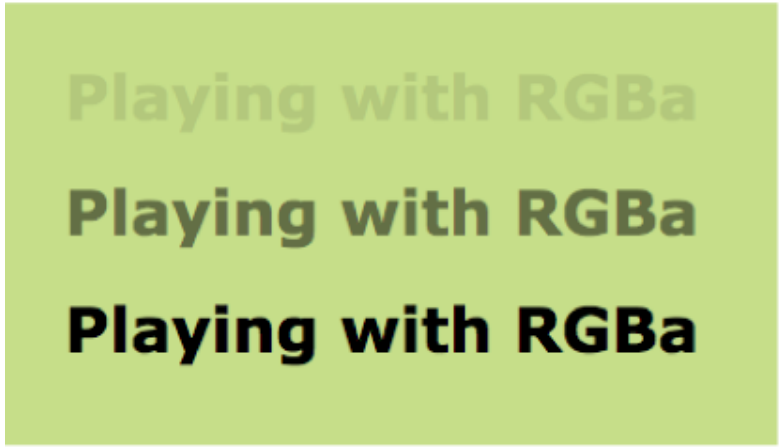
Hexadecimal RGB Values

Red, green, and blue values converted to **hexadecimal** and preceded by the # symbol.



RGBa Color

- **RGB + an alpha channel** for transparency
- The first three values are RGB. The fourth is the transparency level from 0 (transparent) to 1 (opaque).



HSL and HSLa

- Colors described by values for **hue** (°), **saturation** (%), and **luminosity** (%):

```
hsl(180,50%,75%)
```

- Hue specifies the position on a color wheel (in degrees) that has red at 0°, green at 120°, and blue at 240°.
- HSL is less commonly used than RGB, but some find it more intuitive.
- **HSLa** adds an alpha value for transparency.

Foreground Color

`color`

Values: *`Color value`* (named or numeric)

Example:
```
blockquote {border: 4px dashed; color: green;}
```

The **foreground** of an element consists of its text and border (if one is specified).

> In the latitude of central New England, cabbages are not secure from injury from frost with less than a foot of earth thrown over the heads. In mild winters a covering of half that depth will be sufficient; but as we have no prophets to foretell our mild winters, a foot of earth is safer than six inches.

Background Color

`background-color`

Values: *`Color value`* (named or numeric)

Example:

```
blockquote {border: 4px dashed;  background-color: green;}
```

The **background painting area** of an element fills the area behind the text to the outer edge of the border.

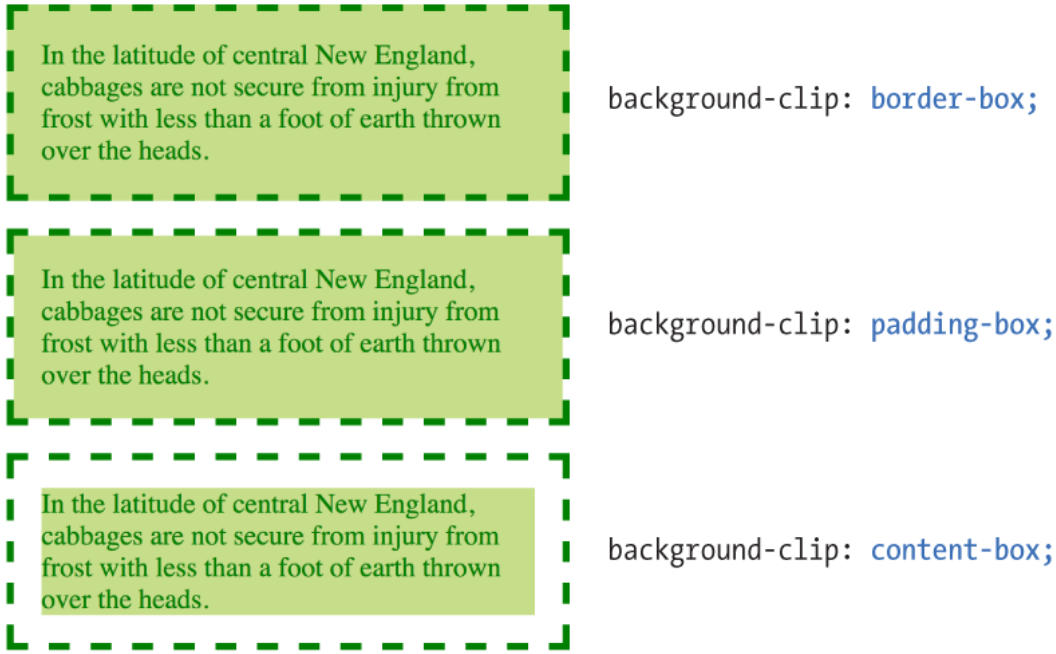
> In the latitude of central New England, cabbages are not secure from injury from frost with less than a foot of earth thrown over the heads. In mild winters a covering of half that depth will be sufficient; but as we have no prophets to foretell our mild winters, a foot of earth is safer than six inches.

Clipping the Background

`background-clip`

Values: `border-box`, `padding-box`, `content-box`

Specifies where the background painting area ends.



Opacity

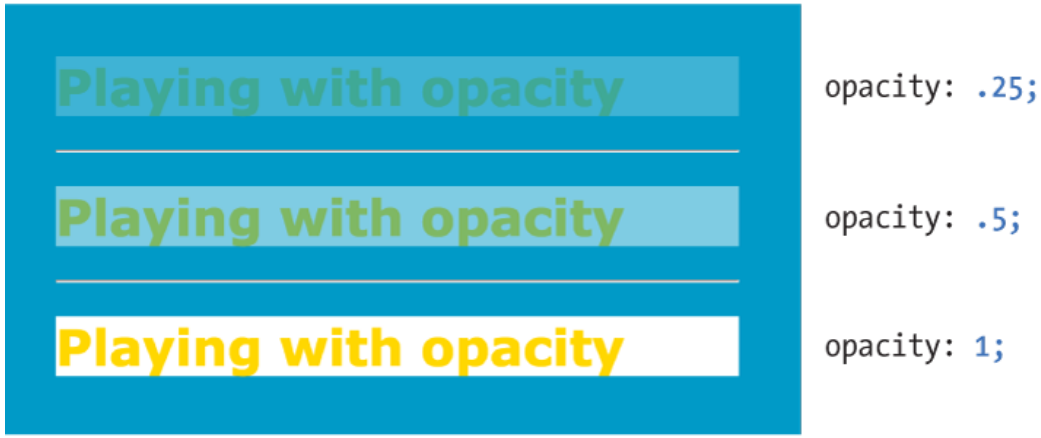
`opacity`

Values: *`number`* (0 to 1)

Example:

```
h1 {color: gold; background: white; opacity: .25;}
```

Specifies the transparency level from 0 (transparent) to 1 (opaque):



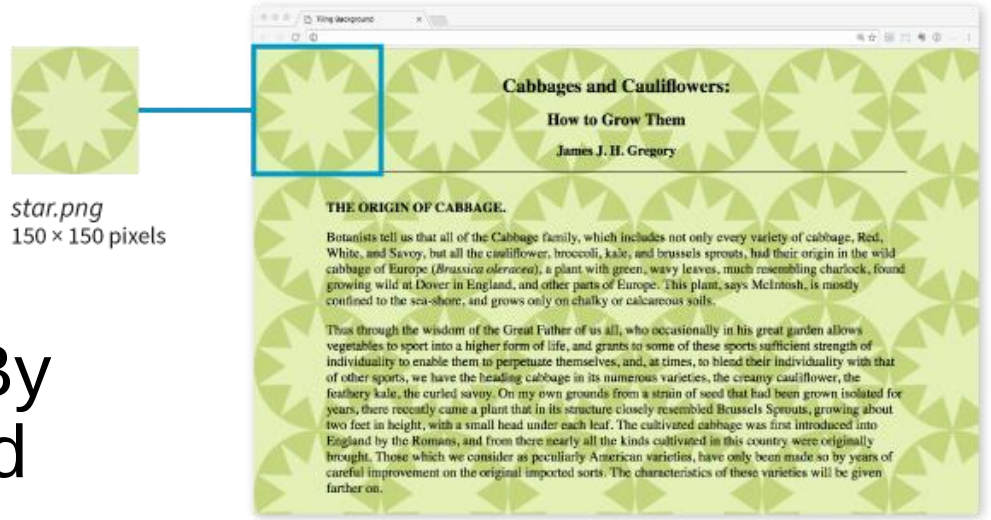
Tiling Background Images

background-image

Values: `url(`*location of image*`)`, `none`

Example:

```
body {background-image: url(star.png);}
```



Locates an image to be used as a tiling background image behind an element. By default, it starts at the top, left corner and repeats horizontally and vertically:

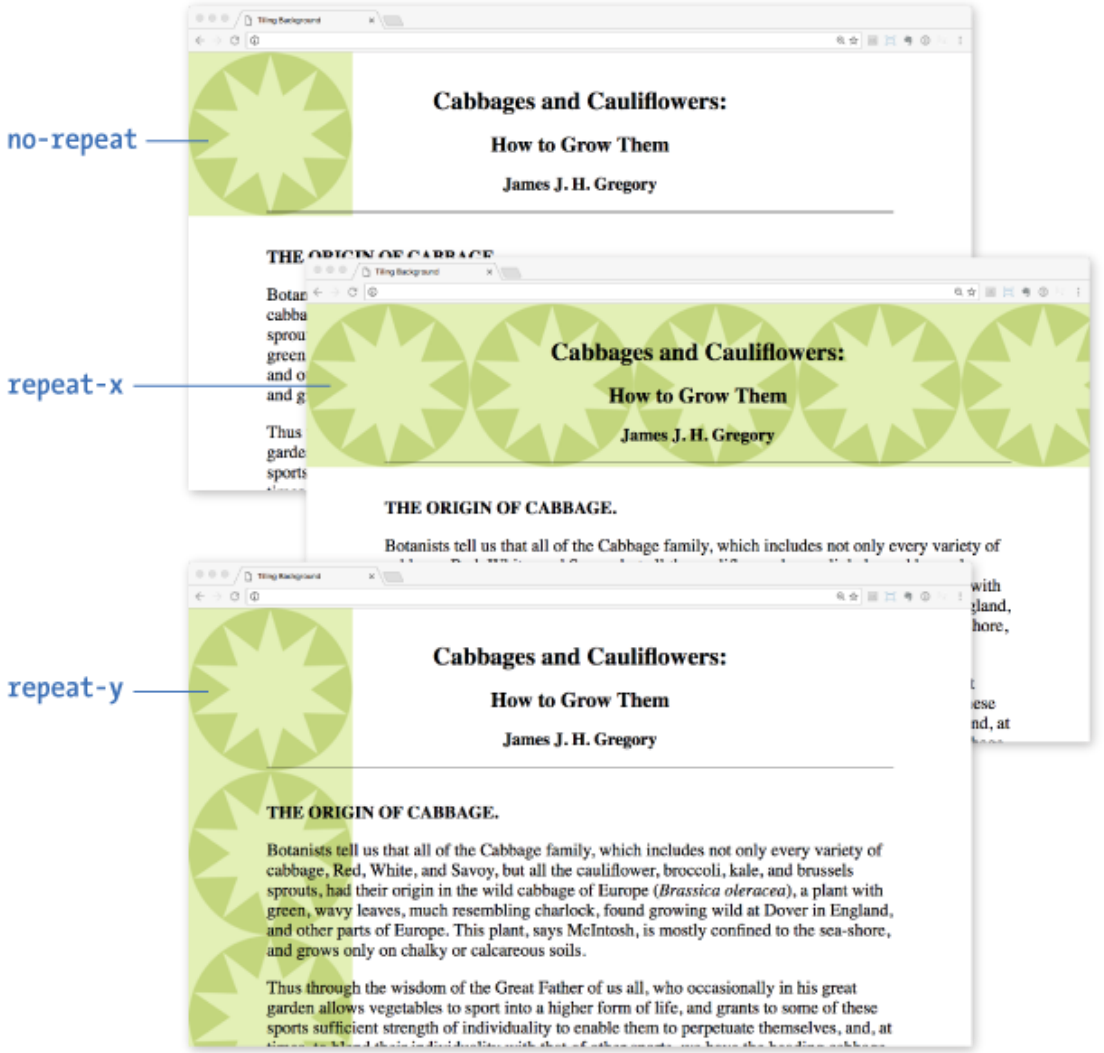
Background Repeating

`background-repeat`

Values:
`repeat`, `no-repeat`, `repeat-x`, `repeat-y`, `space`, `round`

Specifies how the background image repeats and can restrict it to tiling in one direction or not at all:

- **`repeat-x`**: Tiles horizontally only
- **`repeat-y`**: Tiles vertically only
- **`space`**: Adds space around images so they fit in the window with no clipping
- **`round`**: Distorts the image so it fits without clipping



Background Position

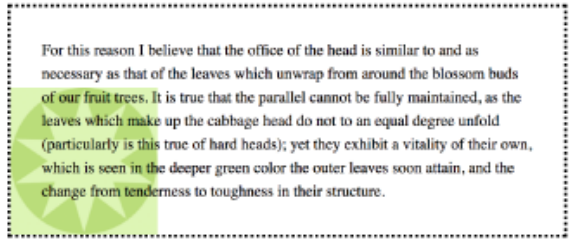
`background-position`

Values:
`Length`, *`percentage`*, `left`, `center`, `right`, `top`, `bottom`

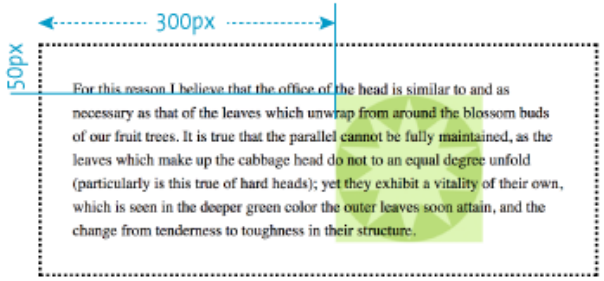
Specifies the position of the **origin image**, the first image that is placed in the background from which tiling images extend.

Examples (horizontal position goes first):

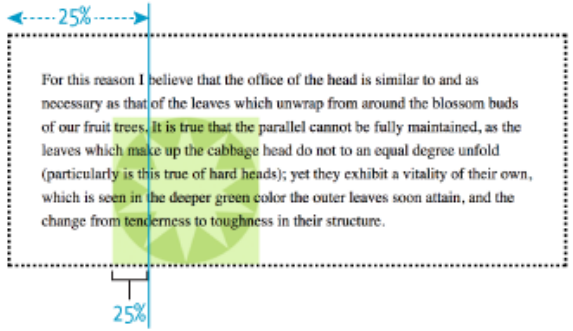
```
background-position: left bottom;
background-position: 300px 100px;
background-position: 25% 100%;
```



`background-position: left bottom;`



`background-position: 300px 50px;`



`background-position: 25% 100%;`

Background Attachment

`background-attachment`

Values: `scroll`, `fixed`, `local`

Specifies whether the background image scrolls with the content or stays in a fixed position relative to the viewport.

A large non-repeating background image in the **body** of the document.

`background-attachment: scroll;`

By default, the background image is attached to the **body** element and scrolls off the page when the page content scrolls.

`background-attachment: fixed;`

When **`background-attachment`** is set to **`fixed`**, the image stays in its position relative to the browser viewing area and does not scroll with the content.

Background Size

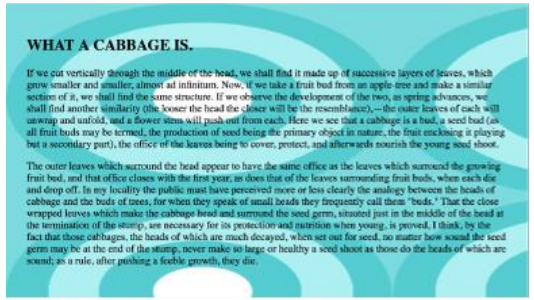
`background-size`

Values:

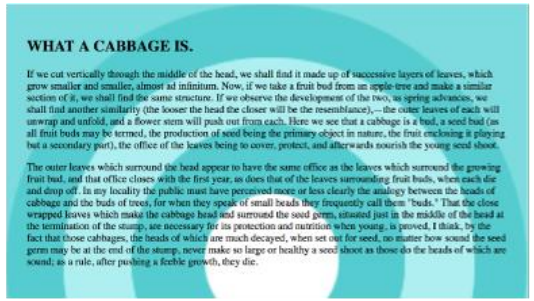
`Length`, *`percentage`*, `auto`, `cover`, `contain`

target.png
300 × 300 pixels

Specifies the size of the tiling image:

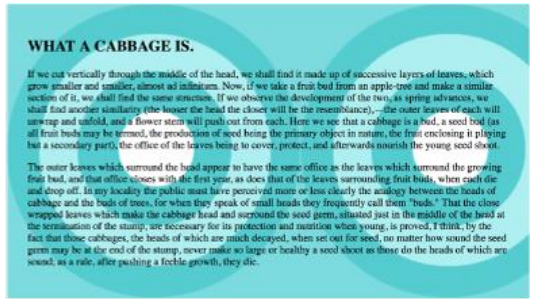


`background-size: 600px 300px;`



`background-size: cover;`

The entire background area of the element is covered, and the image maintains its proportions even if it is clipped.



`background-size: contain;`

The image is sized proportionally so it fits entirely in the element. There may be room left over for tiling (as shown).

Shorthand background Property

`background`

Values:
`background-color background-image background-repeat background-attachment background-position background-clip background-origin background-size`

- Specifies all background properties in one declaration

```
background: white url(star.png) no-repeat top center fixed;
```

- Properties are optional and may appear in any order
- Properties not represented reset to their defaults—be careful it doesn't overwrite previous background settings.

Multiple Background Images

- You can place more than one background image in a single image (separated by commas):

```
body {
  background:
    url(image1.png) left top no-repeat,
    url(image2.png) center center no-repeat,
    url(image3.png) right bottom no-repeat;
  }
```

More with colors

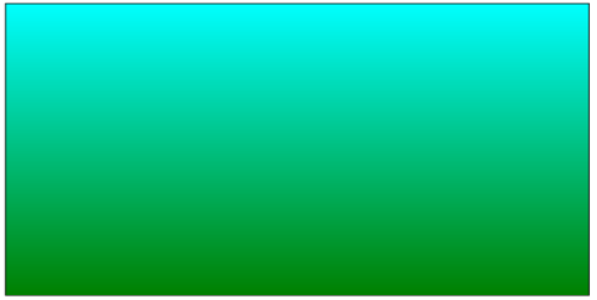
- Gradient fills
  - Linear
  - Radial

- Be careful with vendor prefix (Depending on browser, experimental implementation of properties)

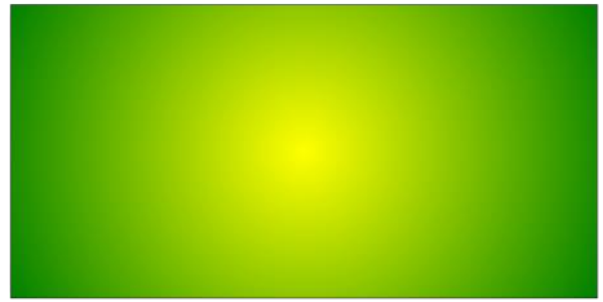
- Tips: use a gradient generator like

https://cssgradient.io/

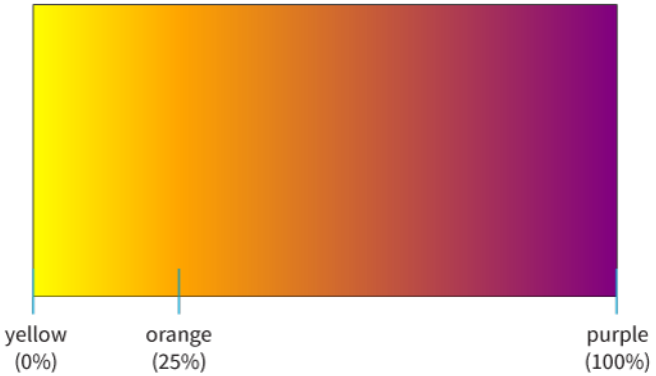
```
linear-gradient(180deg, aqua, green);
or
linear-gradient(to bottom, aqua, green);
```



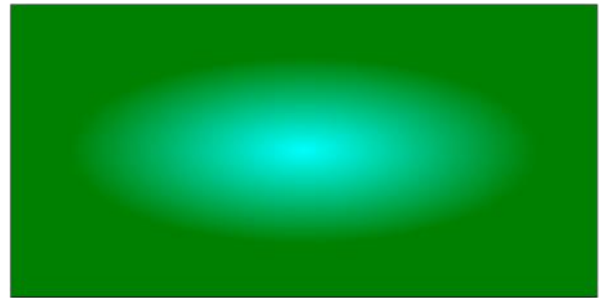
```
radial-gradient(yellow, green);
```



```
linear-gradient(90deg, yellow, orange 25%, purple);
```




```
radial-gradient(200px 80px, aqua, green);
```



More Selector Types

Pseudo-class selectors

Pseudo-element selectors

Attribute selectors

Pseudo-Class Selectors

Treat elements in a certain state as belonging to the same class

Link Pseudo-classes

`:link`	Applies style to unvisited (unclicked) links
`:visited`	Applies style to visited links

User Action Pseudo-classes

`:focus`	Applies when element is selected for input
`:hover`	Applies when the mouse pointer is over the element
`:active`	Applies when the element (such as a link or button) is in the process of being clicked or tapped

Pseudo-classes (cont'd)

Pseudo-classes must appear in the following order:

```
a { text-decoration: none; }  /* turns underlines off for all links */
a:link { color: maroon; }
a:visited { color: gray; }
a:focus { color: maroon; background-color: #ffd9d9; }
a:hover { color: maroon; background-color: #ffd9d9; }
a:active { color: red; background-color: #ffd9d9; }
```

a:link

Links are maroon and not underlined.

a:focus
a:hover

While the mouse is over the link or when the link has focus, the pink background color appears.

a:active

As the mouse button is being pressed, the link turns bright red.

a:visited

After that link has been visited, the link is gray.

More Pseudo-Class Selectors

Structural pseudo-classes

These allow selection based on where the element is in the structure of the document (the document tree):

- `:root`
- `:empty`
- `:first-child`
- `:last-child`
- `:only-child`
- `:first-of-type`
- `:last-of-type`
- `:only-of-type`
- `:nth-child()`
- `:nth-last-child()`
- `:nth-of-type()`
- `:nth-last-of-type()`

Input pseudo-classes

These selectors apply to states that are typical for form inputs:

- `:enabled`
- `:disabled`
- `:checked`

Location pseudo-classes (in addition to :link and :visited)

- `:target` (fragment identifier)

Linguistic pseudo-class

- `:lang()`

Logical pseudo-class

- `:not()`

Pseudo-Element Selectors

Applies styles to elements not explicitly marked up in the source.

::first-line

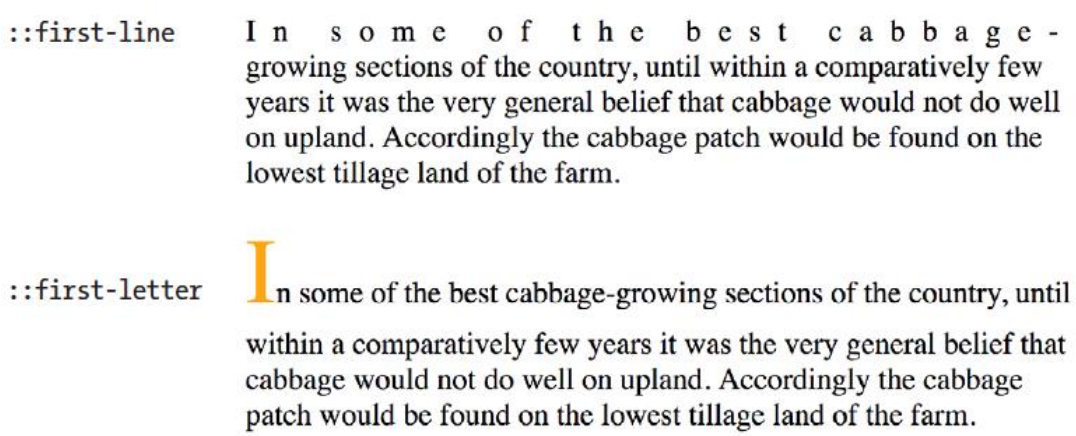
Applies a style to the first line of an element:

```
p:first-line {letter-spacing: 9px;}
```

::first-letter

Applies a style to the first letter of an element:

```
p:first-letter { font-size 300%; color: orange;}
```



Pseudo-Element Selectors (cont'd.)

The `::before` and `::after` pseudo-elements insert generated content before or after a specified element.

`::before`

Inserts copy (provided with the **content** property) before an element and applies style properties to it as specified

`::after`

Inserts copy (provided with the **`content`** property) after an element and applies style properties to it as specified

Generated Content Example

 We are required to warn you that undercooked food is a health risk. Thank you.

The style sheet:

```
p.warning::before {
  content: url(exclamation.png);
  margin-right: 6px;
}
p.warning::after {
  content: " Thank you.";
  color: red;
}
```

The markup:

```
<p class="warning">We are required to warn you that undercooked food is a health risk.</p>
```

Attribute Selectors

Targets elements based on attribute names or values. There are eight types:

Simple attribute selector
Matches an element with a given attribute:

`E[attribute]`

```
img[title] { border: 3px solid;}
```

(Matches every `img` element that has a title *attribute)*

Attribute Selectors (cont.)

```
img[title="first grade"] {border: 3px solid;}
```
Select any image that has title equal to “first grade”

```
img[title~="grade"] {border: 3px solid;}
```
Select any image that has title *with* word “grade”

```
[hreflang|="en"] {border: 3px solid;}
```
Match any link that points to a doc. written in English

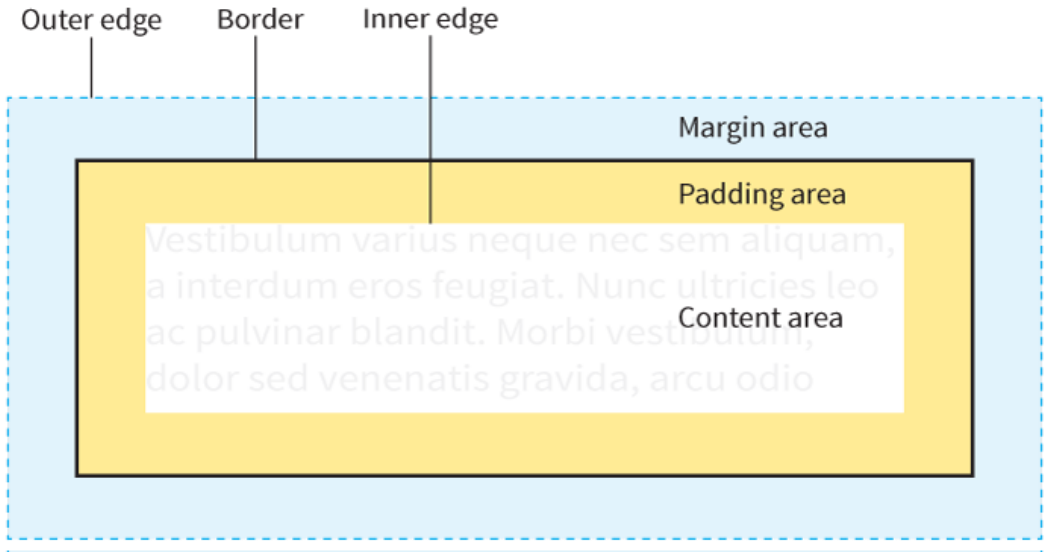
```
img[src^=“/images/icons”]{border: 3px solid;}
```
Apply the style only in images path beginning with /images/icons

```
a[href$=”.pdf”]{border-bottom: 3px solid;}
```
Apply the style to a element that only link to pdf file

```
img[title*=“February”]{border: 3px solid;}
```
Look for any image with ”February” in the title

Box model

- Fundamental concepts of CSS
- Every element in a document generates a box



The parts of an Element Box

NOTE: The margin is indicated with a blue shade and outline, but is invisible in the layout.

Specifying Box Dimensions

`width`

Values: *`Length`*, *`percentage`*, `auto`

`height`

Values: *`Length`*, *`percentage`*, `auto`

Specify the dimensions of an element box with **`width`** and **`height`** properties

Box Sizing Models

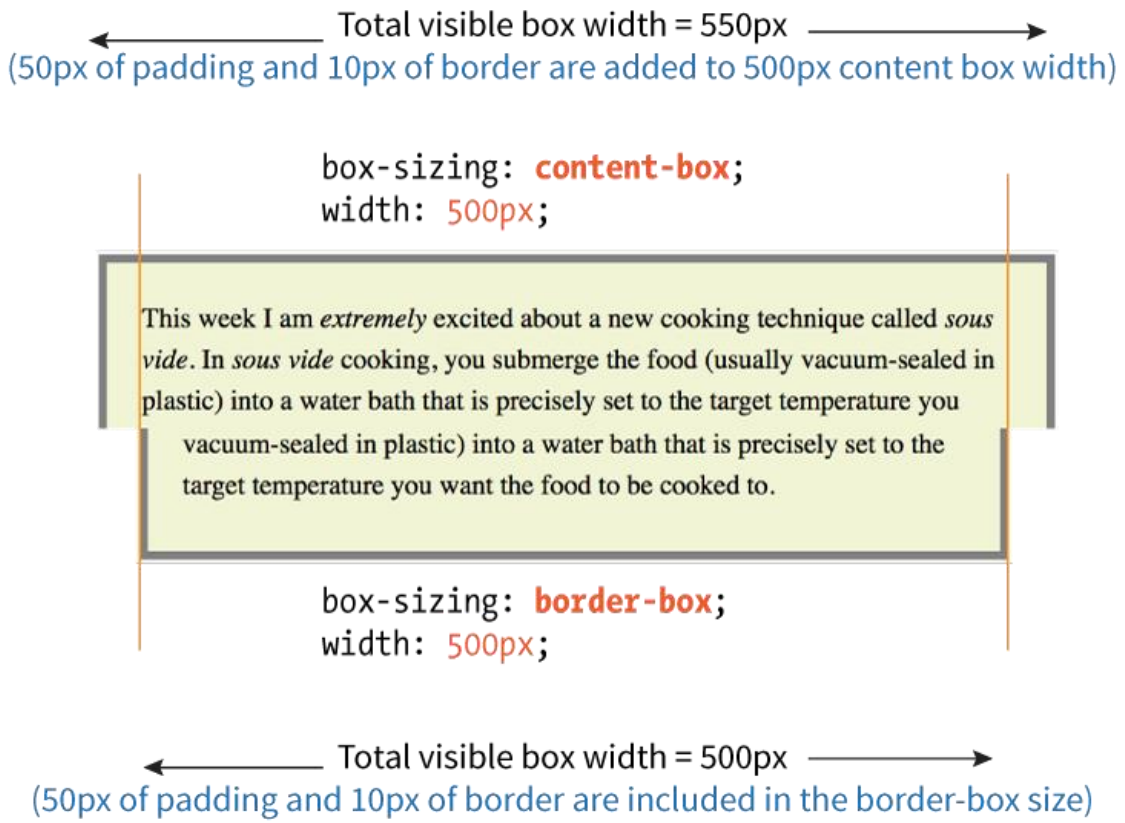
`box-sizing`

Values: `content-box`, `border-box`

There are two methods for sizing an element box, specified with the `box-sizing` attribute:

Content-box sizing (default)
Applies `width` and `height` values to the content box only

Border-box sizing
Applies `width` and `height` values to the border box (including the padding and content)

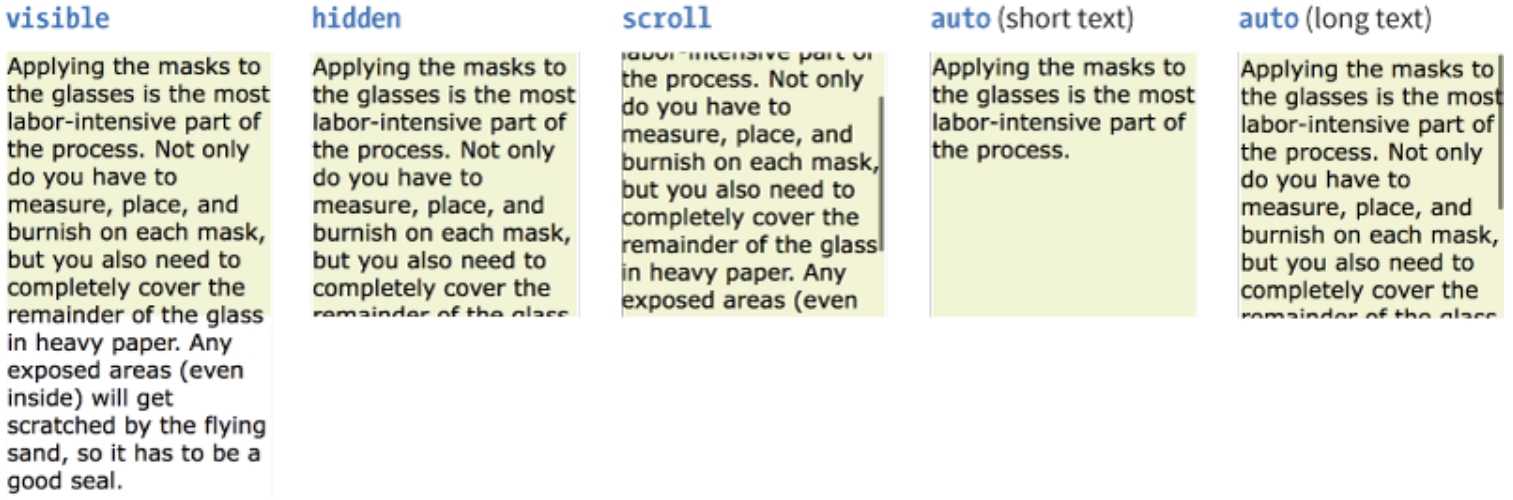


Overflow

`overflow`

Values: `visible`, `hidden`, `scroll`, `auto`

Specifies what to do when content doesn't fit in a sized element box:



Padding

padding, padding-top, padding-right, padding-bottom, padding-left

Values: *Length*, *percentage*

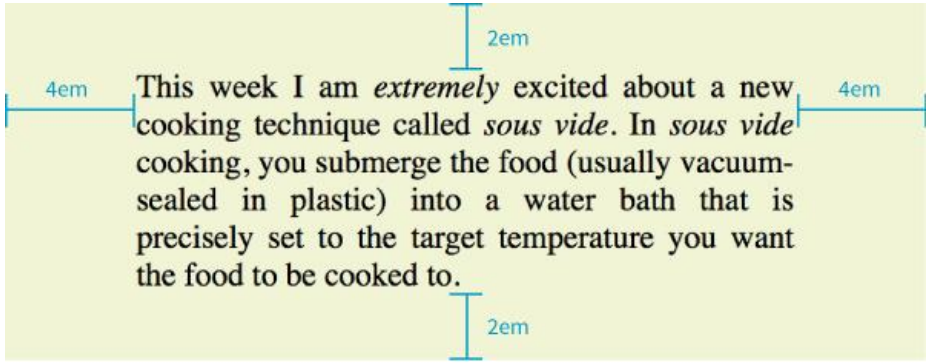
An amount of space between the content area and the border (or the space the border would be if one isn't specified).

You can add padding to one side at a time, or on all sides with the `padding` shorthand property.

```
blockquote {
  padding-top: 2em;
  padding-right: 4em;
  padding-bottom: 2em;
  padding-left: 4em;
  background-color: #D098D4; /*light green*/
}
```

OR use Shorthand Padding Property (clockwise order only: *top right bottom left* ***TRouBLe***)

```
  padding: 2em 4em 2em 4em;
```



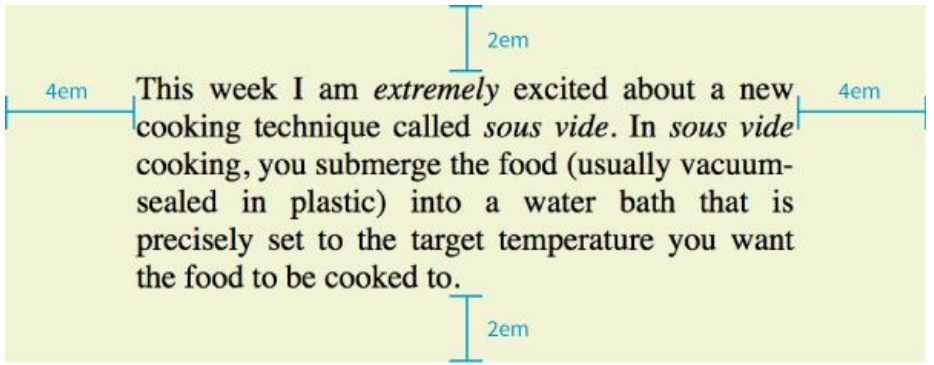
Shorthand padding Property (cont'd)

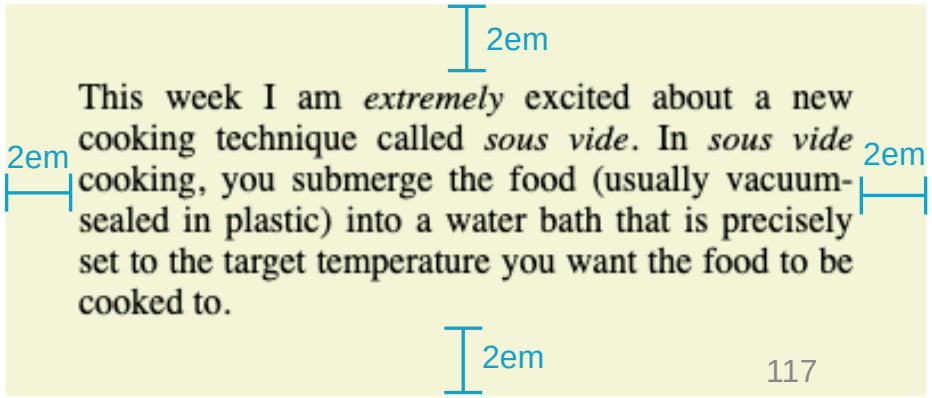
```
/*padding: top right bottom left;*/
padding: 2em 4em 2em 4em;

/*padding: top right+left bottom;*/
padding: 2em 4em 2em;

/*padding: top+bottom  right+left;*/
padding: 2em 4em;

/*padding: all sides;*/
padding: 2em;
/*(2em padding all around)*/
```





Borders

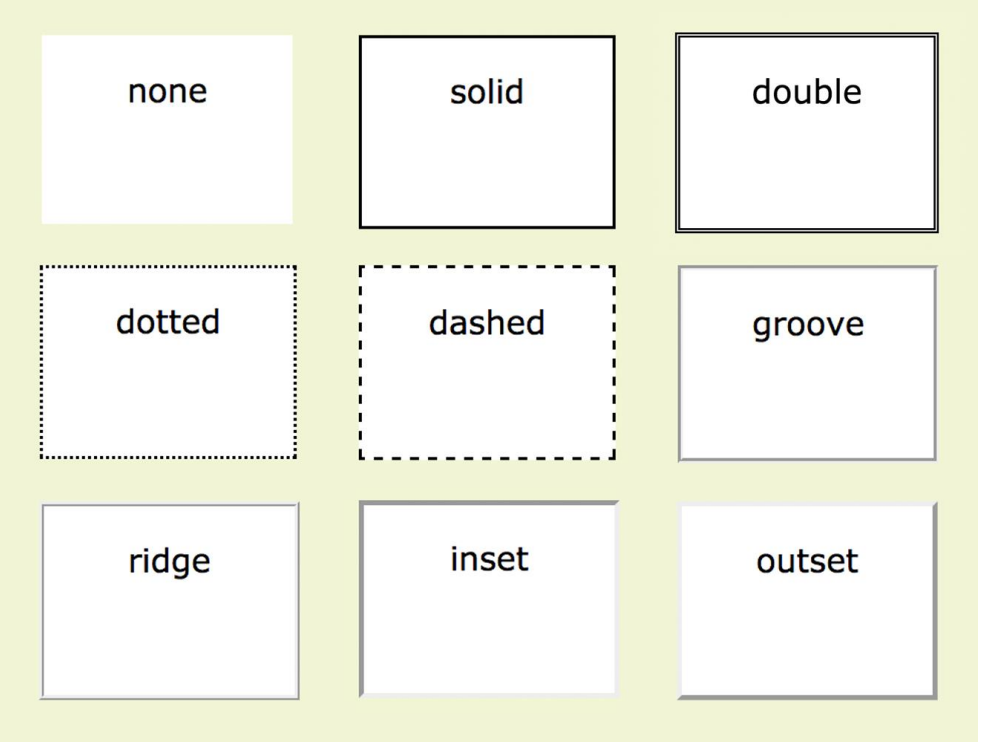
- A **border** is a line drawn around the content area and its (optional) padding.
- The thickness of the border is included in the dimensions of the element box.
- You define **style**, **width** (thickness), and **color** for borders.
- Borders can be applied to single sides or all around

Border Style

`border-style`,
`border-top-style`, `border-right-style`,
`border-bottom-style`, `border-left-style`

Values: `none`, `solid`, `hidden`, `dotted`, `dashed`, `double`, `groove`, `ridge`, `inset`, `outset`

NOTE: The default is `none`, so if you don't define a border style, it won't appear.



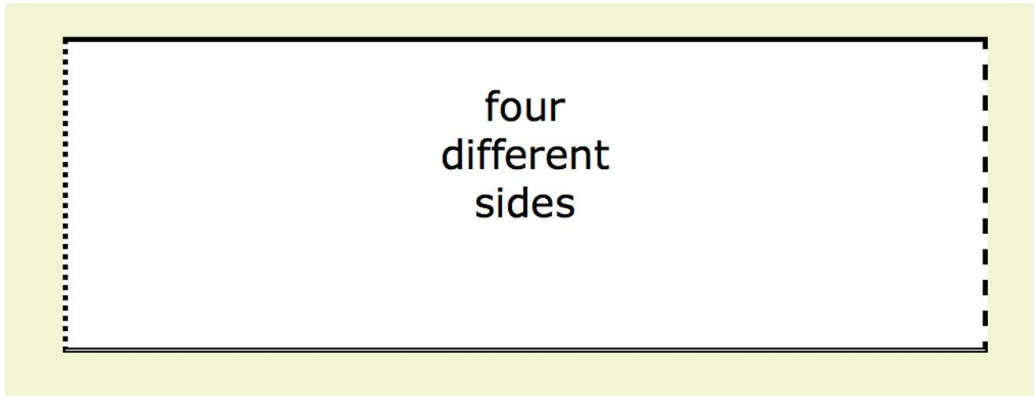
Border Style

`border-style`

The **`border-style`** shorthand uses the clockwise (TRouBLe) shorthand order. The following rules have the same effect:

```
div#silly {
  border-top-style: solid;
  border-right-style: dashed;
  border-bottom-style: double;
  border-left-style: dotted;
  width: 300px;
  height: 100px;
}

div#silly {
  border-style: solid dashed double dotted;
}
```



Border Width, Color

border-width,
border-top-width, **border-right-width**,
border-bottom-width, **border-left-width**

Values: ***Length***, **thin**, **medium**, **thick**

The **border-width** shorthand uses the clockwise (TRouBLe) order:

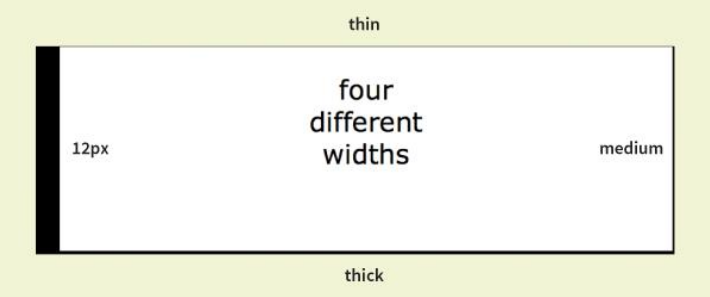
border-color,
border-top-color, **border-right-color**,
border-bottom-color, **border-left-color**

Values: ***Color value*** *(named or numeric)*

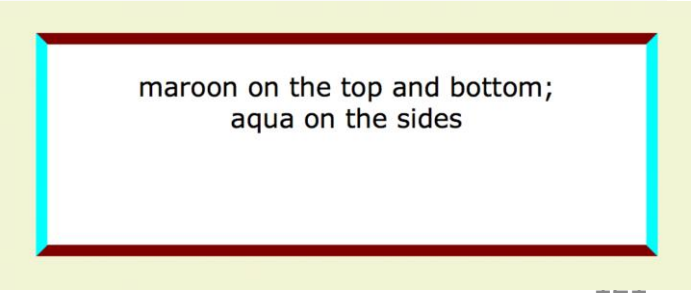
The **border-color** properties override the **color** property

NOTE: The **border-style** property is required for the border to be rendered.

```
div#help {
  border-width: thin medium thick 12px;
  border-style: solid;
  width: 300px;
  height: 100px;
}
```




```
div#special {
  border-color: maroon aqua;
  border-style: solid;
  border-width: 6px;
  width: 300px;
  height: 100px;
}
```



Border Shorthand Properties

`border`,
`border-top`, `border-right`,
`border-bottom`, `border-left`

Values: *`border-style border-width border-color`*

Combine style, width, and color values in shorthand properties for each side or all around (`border`):

```
p.example {
  border: 2px dotted aqua;
}
```

This is the example of border: 2px dotted aqua;

NOTE: The `border-style` property must be included in the shorthand for the border to be rendered.

Border Radius (Rounded Corners)

`border-radius`

Values: *1, 2, 3, or 4* `length` *or* `percentage values`

- The `border-radius` property rounds off the corners of an element.
- The value is a length or percentage value reflecting the radius of the curve.
- Providing one value makes all the corners the same.
- Four values are applied clockwise, starting from the top-left corner.

```
p {
  width: 200px;
  height: 100px;
  background: darkorange;
}
```

border-radius: 1em;

border-top-right-radius: 50px;

border-radius: 50px;

```
border-top-left-radius: 1em;
border-top-right-radius: 2em;
border-bottom-right-radius: 1em;
border-bottom-left: 2em;
```

or

```
border-radius: 1em 2em;
```

Margins

margin,
margin-top,
margin-right,
margin-bottom,
margin-left

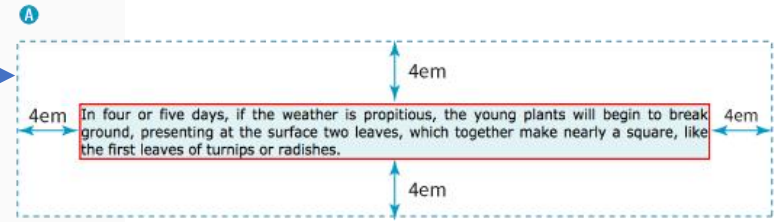
Values: *Length*, *percentage*

Margin is

- An amount of space added on the outside of the border.
- Prevent elements from bumping into one another or the edge of the viewport.

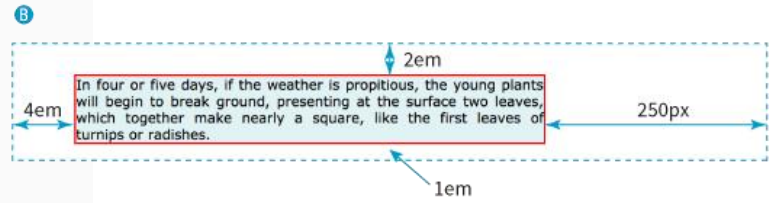
A

```
p#A {
    margin: 4em;
    border: 2px solid red;
    background: #e2f3f5;
}
```



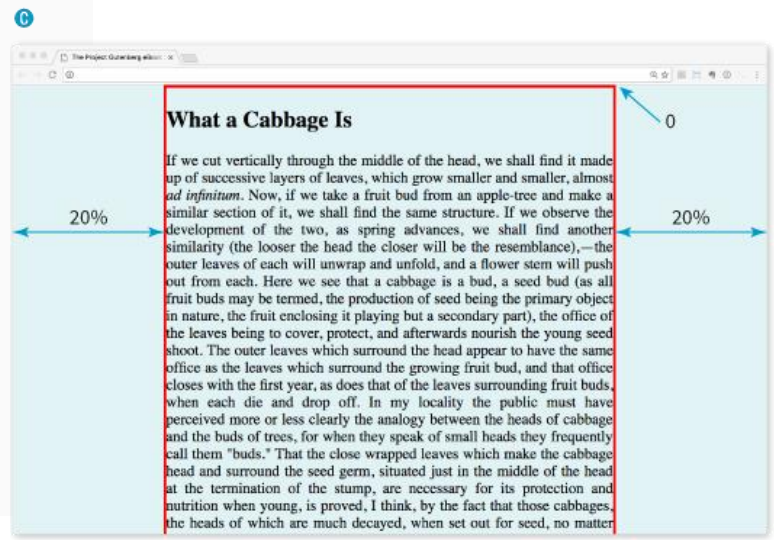
B

```
p#B {
    margin-top: 2em;
    margin-right: 250px;
    margin-bottom: 1em;
    margin-left: 4em;
    border: 2px solid red;
    background: #e2f3f5;
}
```



C

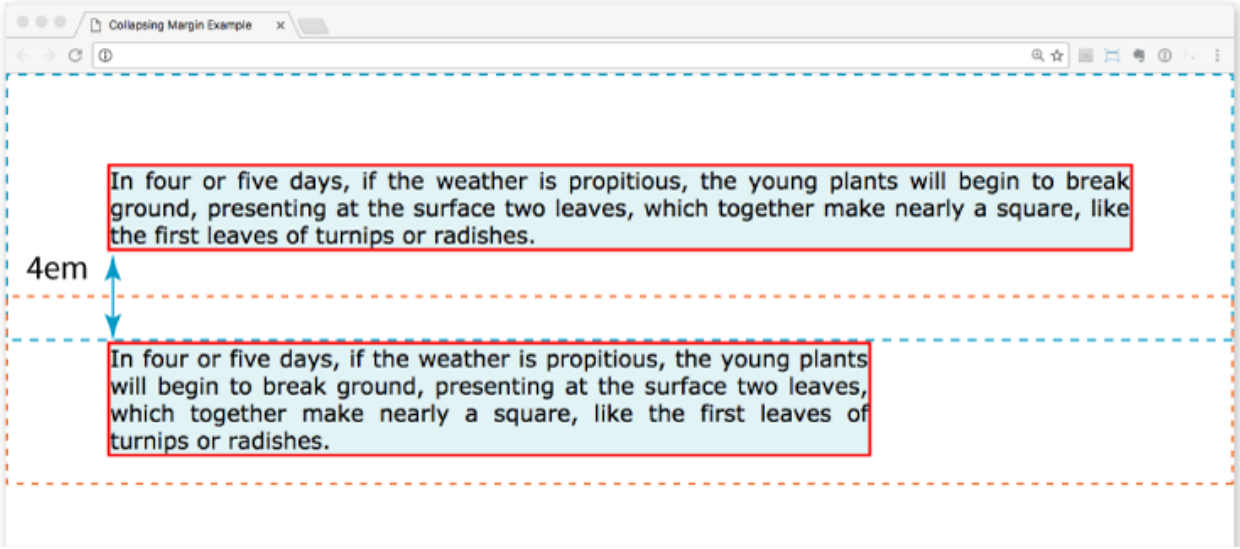
```
body {
    margin: 0 20%;
    border: 3px solid red;
    background-color: #e2f3f5;
}
```



Margins (cont'd)

- Top and bottom margins of neighboring elements **collapse** (they overlap instead of accumulating).

- The top element has a bottom margin of `4em`. The bottom element has a top margin of 2em. The resulting margin is 4em (the largest value).



Assigning Display Types

`display`

Values: `inline` | `block` | `run-in` | `flex` | `grid` | `flow` | `flow-root` | `list-item` | `table` | `table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container` | `inline-block` | `inline-table` | `inline-flex` | `inline-grid` | `contents` | `none`

Assigns a **display type** that determines how the element box behaves in layouts.

Examples:

```
nav li { display: inline; }

nav li a { display: block; }

nav li a { display: none; }
```

- Make `li` (normally block elements) into inline elements so they line up in a horizontal menu:
- Make an anchor (`a`) element (normally inline) display as a block so you can give it a width and height:
- To prevent certain content in the source document from displaying in specific media. Be careful that it would still download.

Box Drop Shadows

box-shadow

Values: *horizontal-offset vertical-offset blur-distance spread-distance color inset*, none

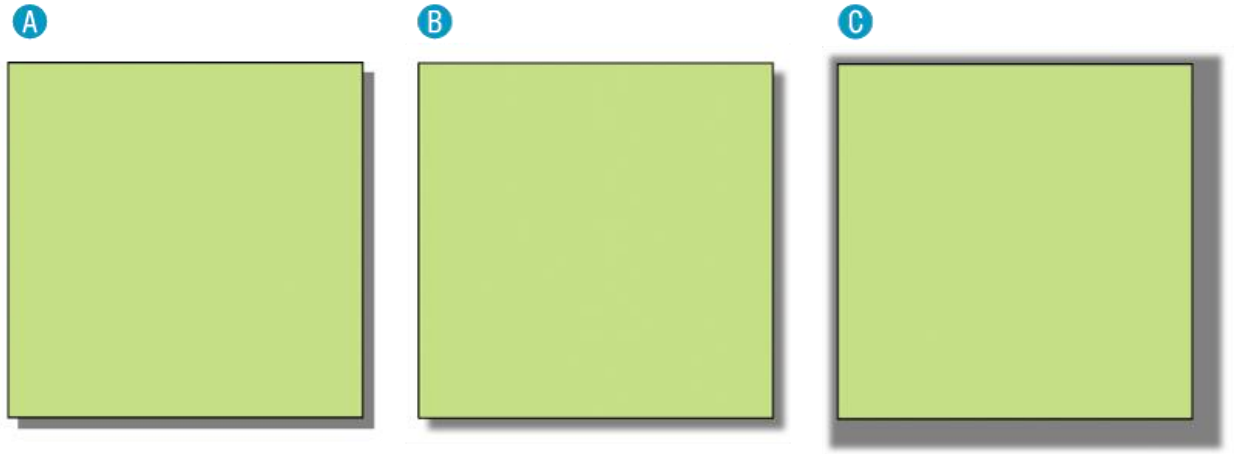
Applies a drop shadow around the visible element box.

The values are a horizontal and vertical offset, optional blur and spread values (in pixels), a color value, and the option to make it appear inset.

A
```
box-shadow: 6px 6px gray;
```

B
```
box-shadow: 6px 6px 5px gray;
/* 5 pixel blur */
```

C
```
box-shadow: 6px 6px 5px 10px gray;
/* 5px blur,10px spread */
```

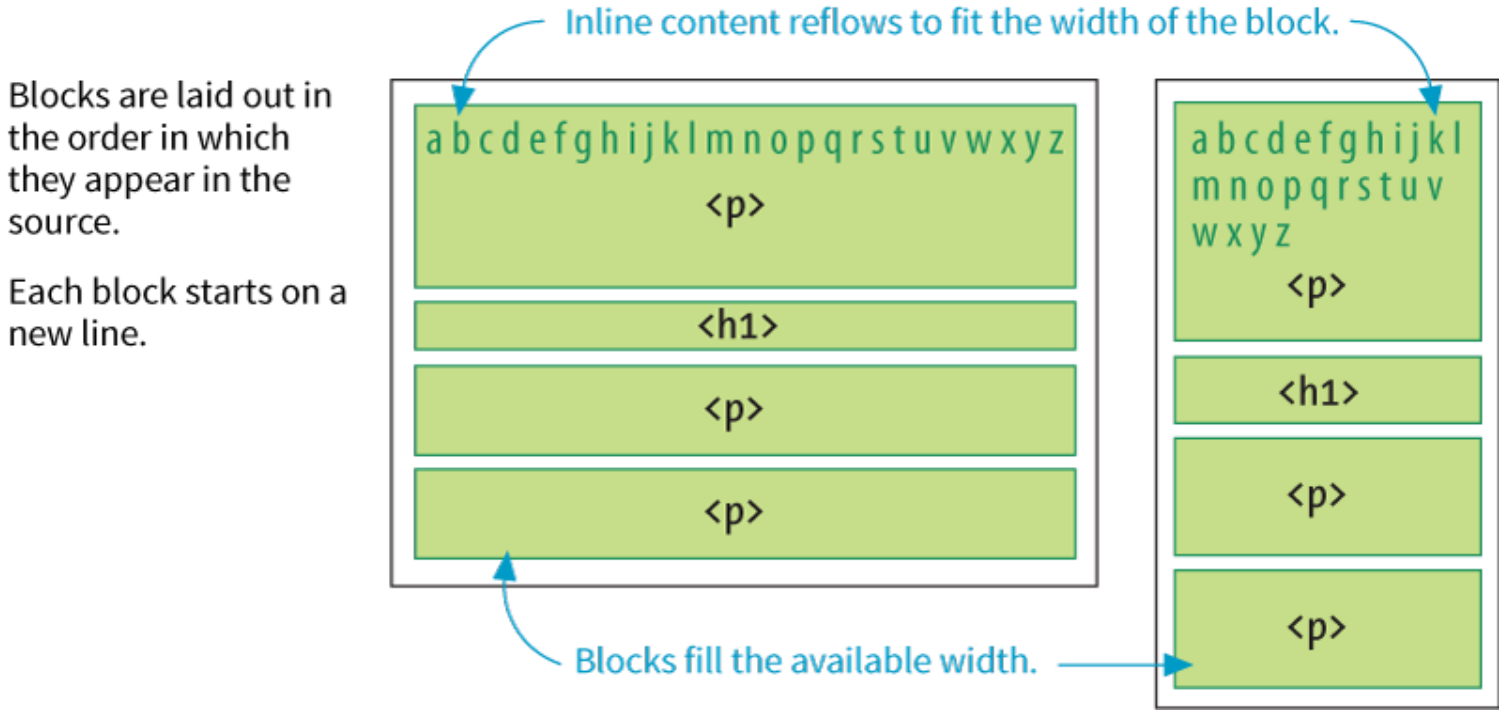


Floating and positioning

- Floating elements to the left and right
- Clearing floated elements
- Containing floated elements
- Creating text-wrap shapes
- Relative Positioning
- Absolute positioning and containing blocks
- Fixed positioning

Normal Flow

- In the **normal flow**, elements are laid out from **top to bottom** in the order in which they appear in the source and from **left to right** (in left-to-right reading languages).



Floating

`float`

Values: `left`, `right`, `none`

Moves an element as far as possible to the left or right and allows the following content to wrap around it:

```
img { float: right; }
```

Inline image floated to the right

Image moves over, and text wraps around it

After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub. Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid.

Note:

- Floated elements are removed from the normal flow but **influence** the surrounding content.
- Floated elements stay **within** the content area of the element that contains it.
- Margins are **always maintained** (they don't collapse) on all sides of floated elements.
- You must provide a **width** for a floated text element (because default width is `auto`).
- Elements **don't float higher** than their reference in the source.

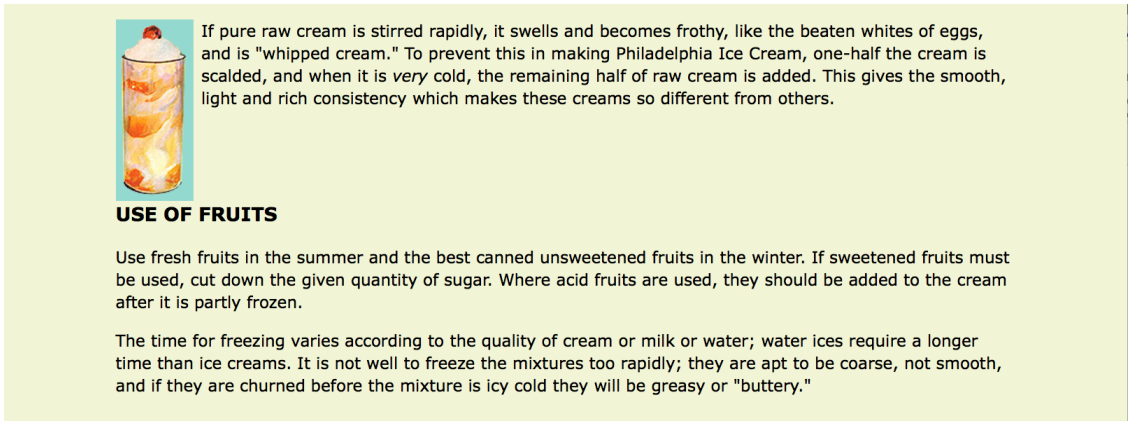
Clearing Floated Elements

`clear`

Values: `left`, `right`, `both`, `none`

Prevents an element from appearing next to a floated element and forces it to start against the next available “clear” space

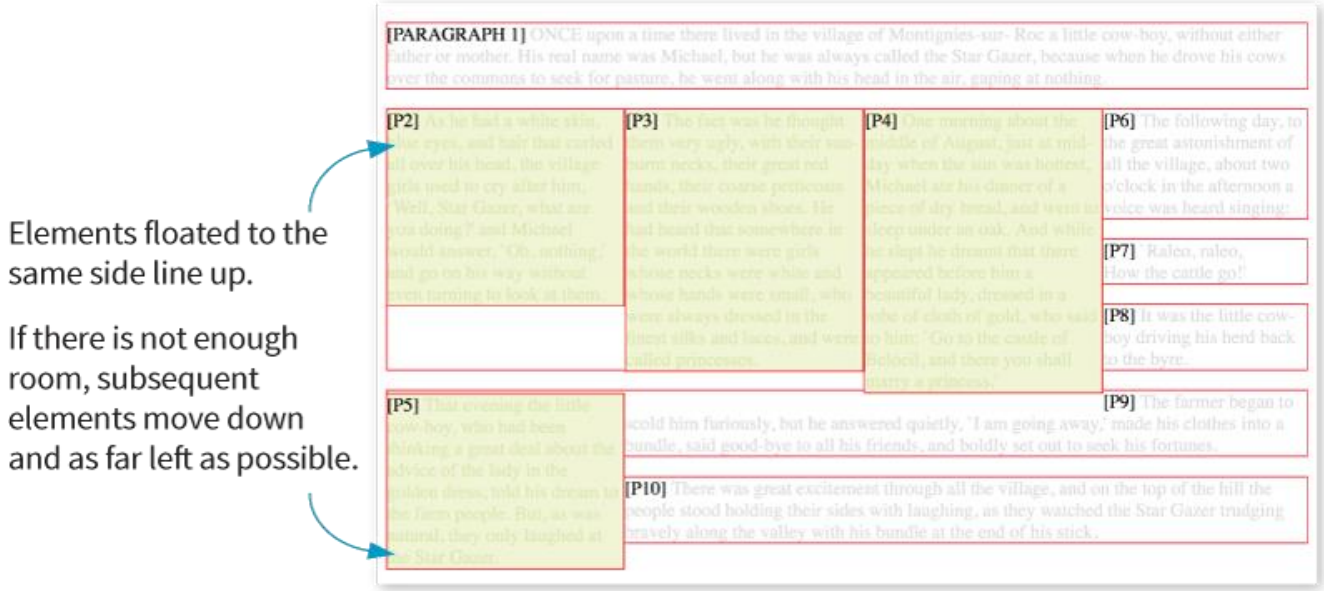
```
img {
  float: left;
  margin-right: .5em;
}
h2 {
  clear: left;
  margin-top: 2em;
}
```



(The `h2` is “cleared” and starts below the floated element.)

Floating Multiple Elements

- When you float multiple elements, browsers follow rules in the spec to ensure they don't overlap.
- Floated elements will be placed as far left or right (as specified) and as high up as space allows.

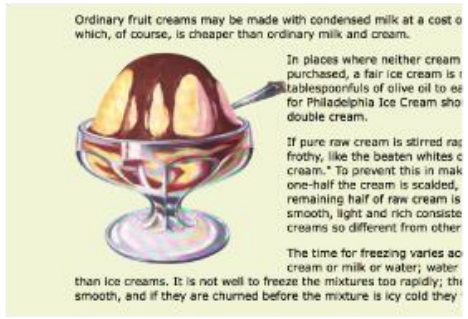


CSS Shapes (Text Wrap)

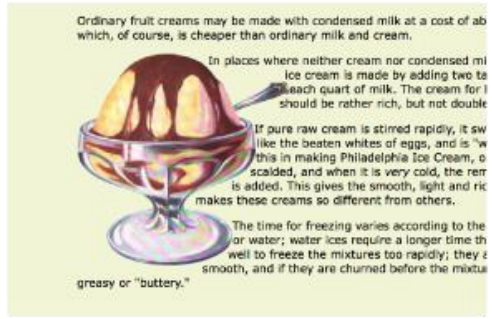
shape-outside

Values: `none, circle(), ellipse(), polygon(), url(), [margin-box | padding-box | content-box ]`

Changes the shape of the text wrap to a circle or ellipse, a complex path, or based on transparent areas in an image



Default text wrap



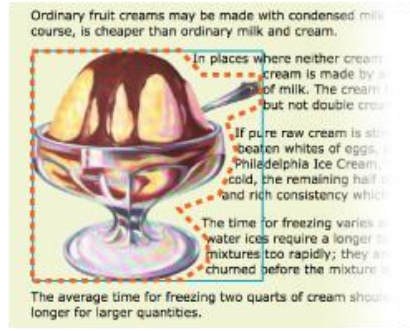
Using the transparent areas of the image as a guide:
`shape-outside: url(sundae.png);`

Using circle() notation:
`shape-outside: circle(200px);`

shape-margin

Values: *length, percentage*

Specifies an amount of space to hold between the image and the wrapped text



The edges of the image (blue) and polygon path (dotted orange) revealed.

Using a path:

```
shape-outside: polygon(0px 0px, 186px 0px,
225px 34px, 300px 34px, 300px 66px, 255px 88px,
267px 127px, 246px 178px, 192px 211px, 226px
236px, 226px 273px, 209px 300px, 0px 300px);
```

Positioning

`position`

Values: `static`, `relative`, `absolute`, `fixed`, `sticky`

Indicates that an element is to be positioned and specifies which positioning method to use

- **Static:** The default position in the flow
- **Relative:** Moves the element relative to its original position
- **Absolute:** Removes the element from the flow and places it relative to the viewport or other containing element
- **Fixed:** Element stays in one position relative to the viewport
- **Sticky:** Element starts in the flow but stays fixed once it scrolls to a particular position relative to the viewport

`top, right, bottom, left`

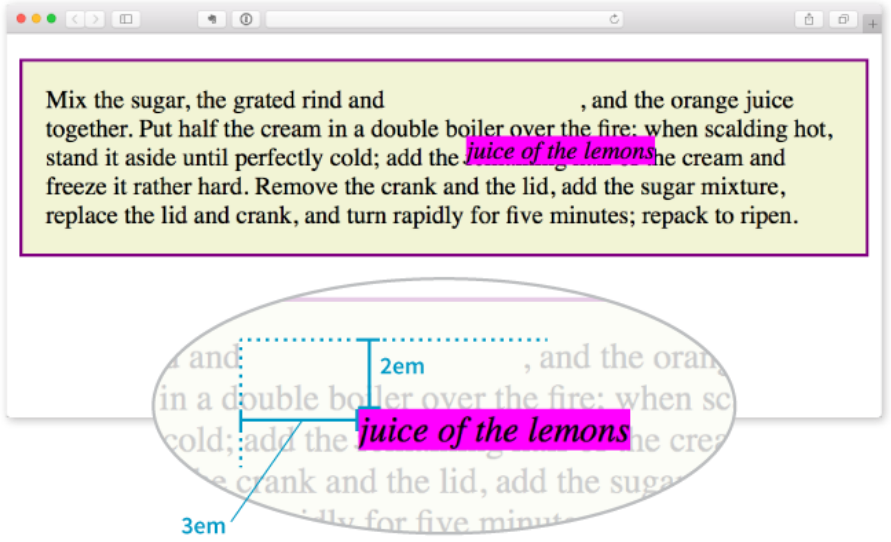
Values: *`length`*, *`percentage`*, `auto`

Offset properties that provide the distance the element should be moved *away* from that respective edge

Relative Positioning

- Moves the element **relative** to its original position
- The space it originally occupied is preserved.

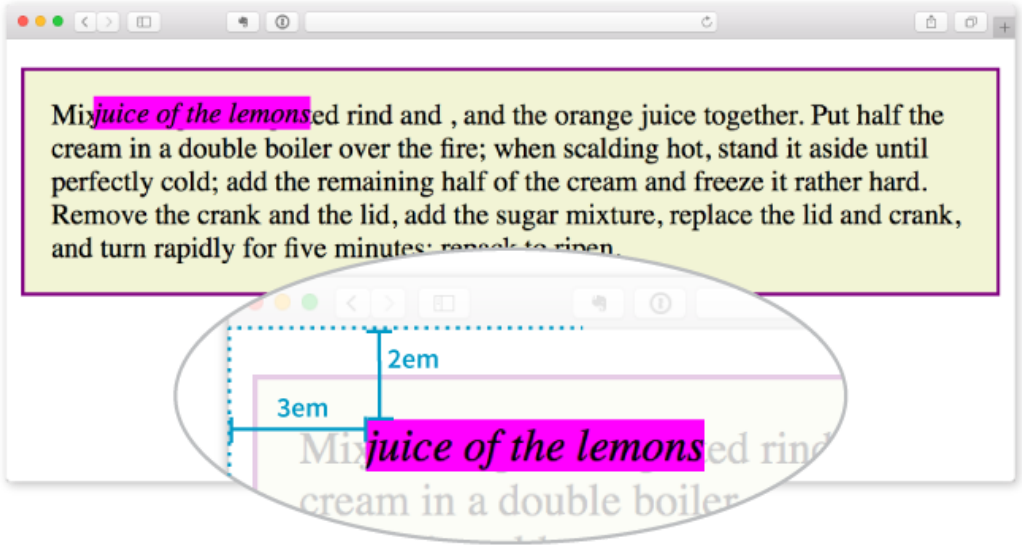
```
em {
  position: relative;
  top: 2em; /* moves it down */
  left: 3em; /* moves it right */
  background-color: fuchsia;
}
```



Absolute Positioning

- Moves the element relative to the **viewport** or **containing block** element
- The space it originally occupied is closed up.

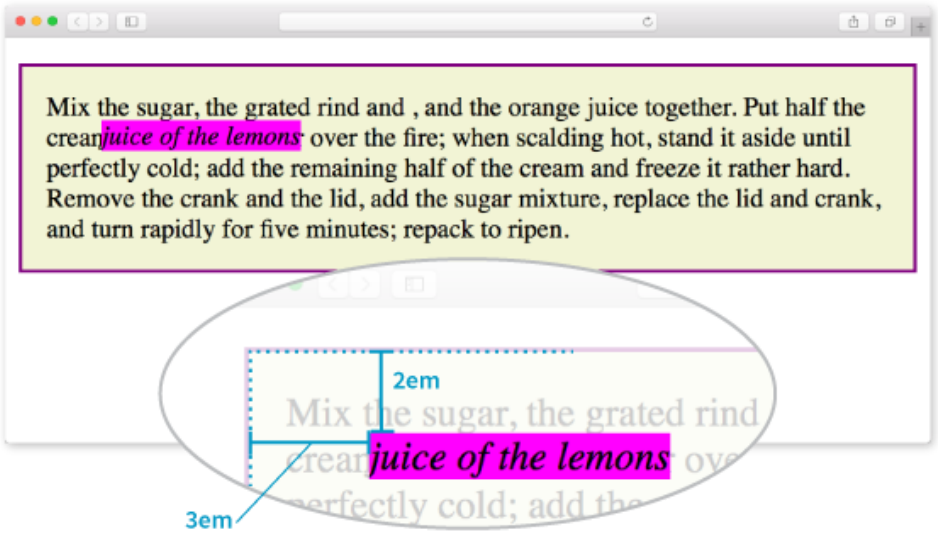
```
em {
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}
```



Containing Blocks

- A positioned element serves as a **containing block** (or *positioning context*) for the elements it contains.
- If a positioned element has an ancestor that has its `position` set to `relative`, `absolute`, or `fixed`, then its position will be **relative** to that containing block element.
- If a positioned element is *not* contained within another positioned element, then it is placed relative to the initial containing block (the `html` element) and the viewport.

```
p {
  position: relative;
  padding: 15px;
  background-color: #F2F5D5;
  border: 2px solid purple;
}a
em {
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}a
```



The relatively positioned **p** element acts as a containing block for the **em** element.

Specifying Position

- Position can be specified in **length measurements** (like pixels) or percentages.

- The measurement moves it **away** from the positioning offset property provided (i.e., `top: 200px;` moves the element **DOWN** from the top edge).

- Be careful **not to overspecify**. Two offset properties are usually enough.

Stacking Order

`z-index`

Values: *`Number`*, `auto`

By default, elements later in the document source order stack on top of preceding elements.

You can change the stacking order with the `z-index` property. Higher values stack on top of lower values.

CSS Layout with FLEXBOX

Flexbox Container
Rows and Columns (Direction)
Flex Flow (Direction + Wrap)
Aligning on the Main Axis
Aligning on the Cross Axis
Aligning with Margins
Specifying How Items “Flex”
Changing Item Order
Browser Support for Flexbox

About Flexbox

- **Flexbox** is a display mode that lays out elements along **one axis** (horizontal or vertical).
- Useful for menu options, galleries, product listings, etc.
- Items in a flexbox can expand, shrink, and/or wrap onto multiple lines, making it a great tool for responsive layouts.
- Items can be reordered, so they aren't tied to the source order.
- Flexbox can be used for individual components on a page or the whole page layout.

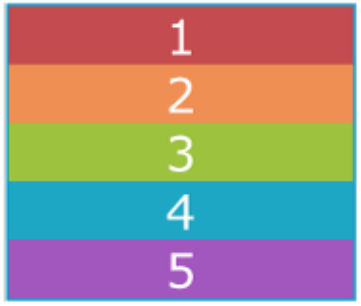
Flexbox Container

```
display: flex
```

- To turn on Flexbox mode, set the element's `display` to `flex`.
- This makes the element a **flexbox container**.
- All of its direct children become **flex items** in that container.
- By default, items line up in the writing direction of the document (left to right rows in left-to-right reading languages).

Flexbox Container (cont'd)

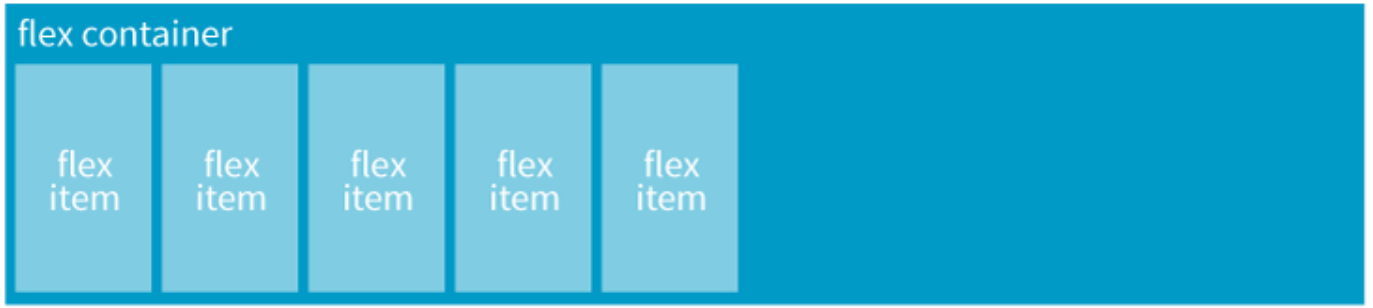
By default, the `divs` display as block elements, stacking up vertically. Turning on flexbox mode makes them line up in a row.



block layout mode

```
display: flex;
```

flexbox layout mode

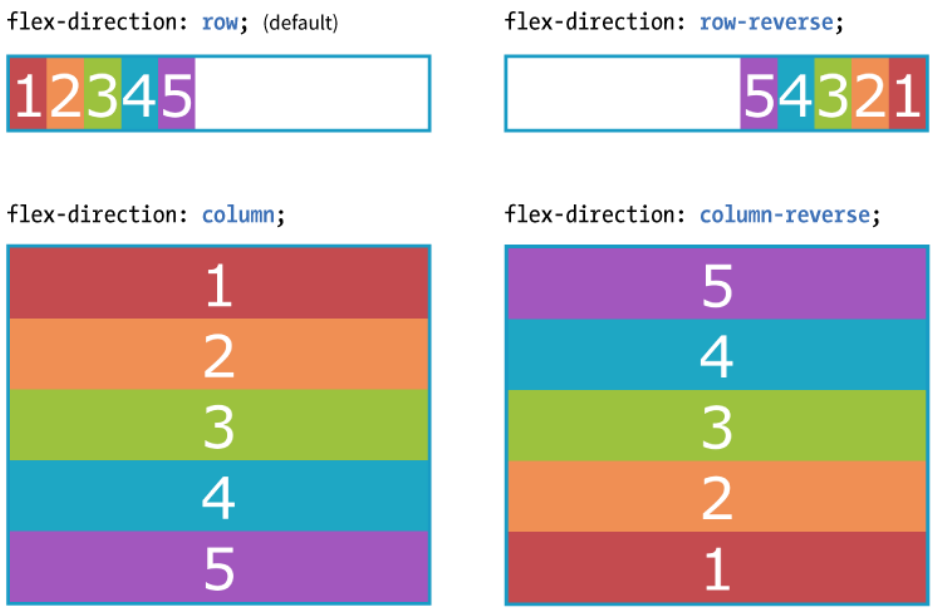


Rows and Columns (Direction)

`flex-direction`

Values: `row`, `column`, `row-reverse`, `column-reverse`

The default value is `row` (for L-to-R languages), but you can change the direction so items flow in columns or in reverse order:



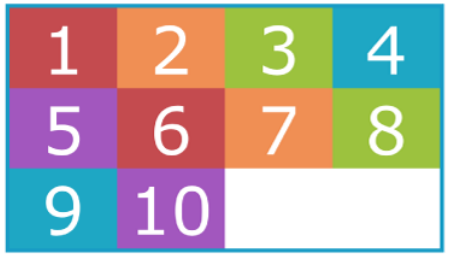
Wrapping Flex Lines

`flex-wrap`

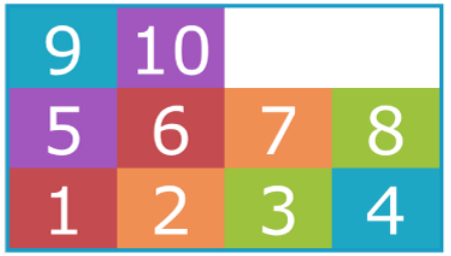
Values: `wrap`, `nowrap`, `wrap-reverse`

Flex items line up on one axis, but you can allow that axis to wrap onto multiple lines with the `flex-wrap` property:

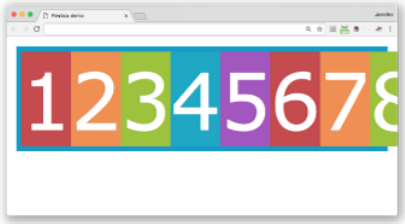
`flex-wrap: wrap;`



`flex-wrap: wrap-reverse;`



`flex-wrap: nowrap;` (default)



When wrapping is disabled, flex items squish if there is not enough room, and if they can't squish any further, may get cut off if there is not enough room in the viewport.

Flex Flow (Direction + Wrap)

`flex-flow`

Values: *`Flex-direction flex-flow`*

The shorthand `flex-flow` property specifies both direction and wrap in one declaration.

Example

```
#container {
  display: flex;
  height: 350px;
  flex-flow: column wrap;
}
```

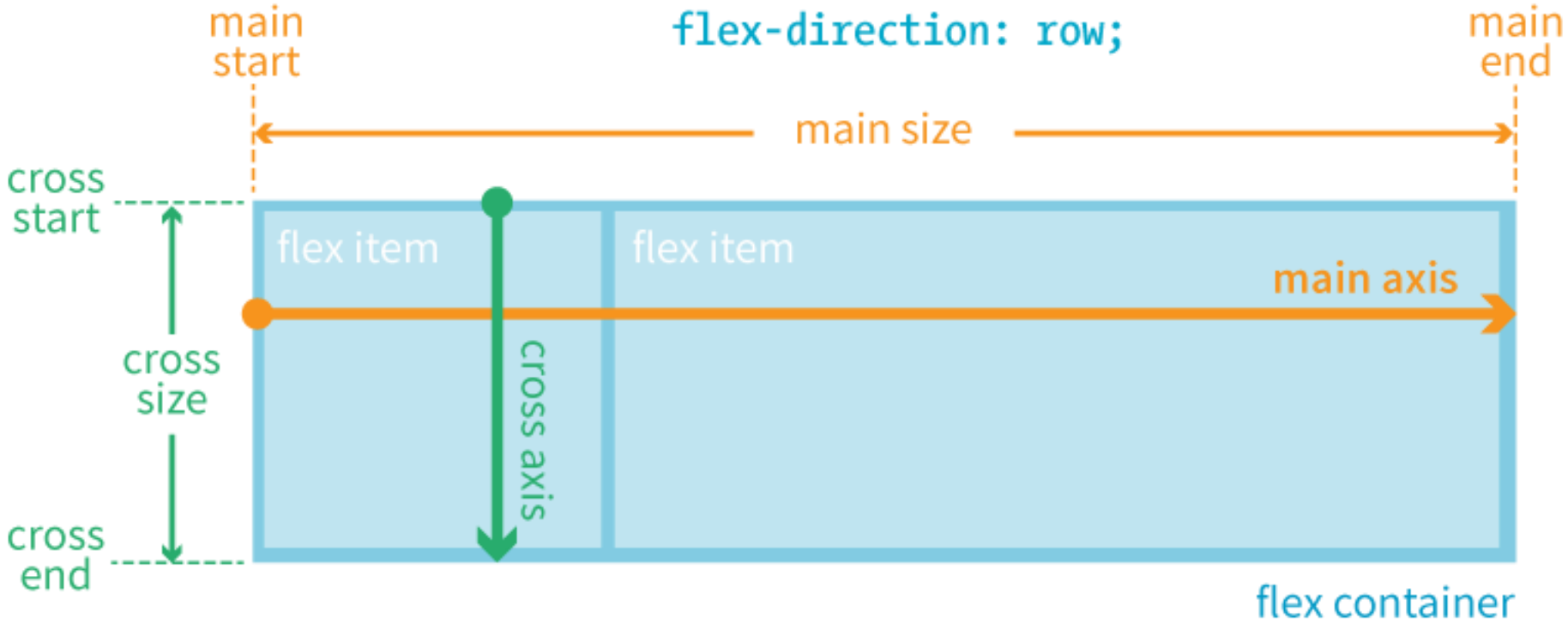
Flexbox Alignment Terminology

- Flexbox is **“direction-agnostic”** so we talk in terms of *main axis* and *cross axis* instead of rows and columns.
- The **main axis** runs in whatever direction the flow has been set.
- The **cross axis** runs perpendicular to the main axis.
- Both axes have a **start**, **end**, and **size**.

ROW: Main and Cross Axes

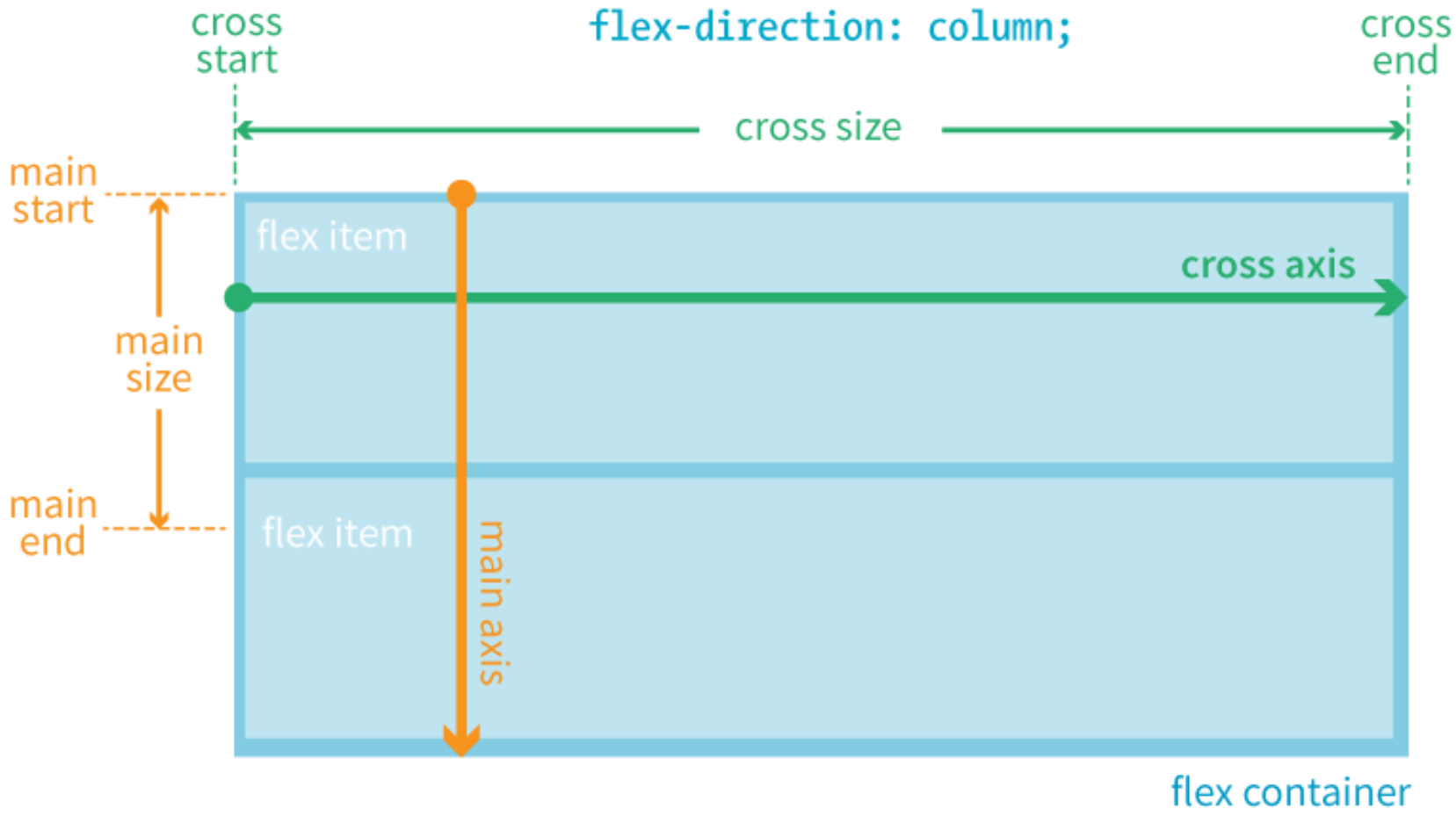
FOR LANGUAGES THAT READ HORIZONTALLY FROM LEFT TO RIGHT:

When `flex-direction` is set to `row`, the main axis is horizontal and the cross axis is vertical.



COLUMN: Main and Cross Axes

When `flex-direction` is set to `column`, the main axis is vertical and the cross axis is horizontal.



Aligning on the Main Axis

`justify-content`

Values: `flex-start`, `flex-end`, `center`, `space-between`, `space-around`

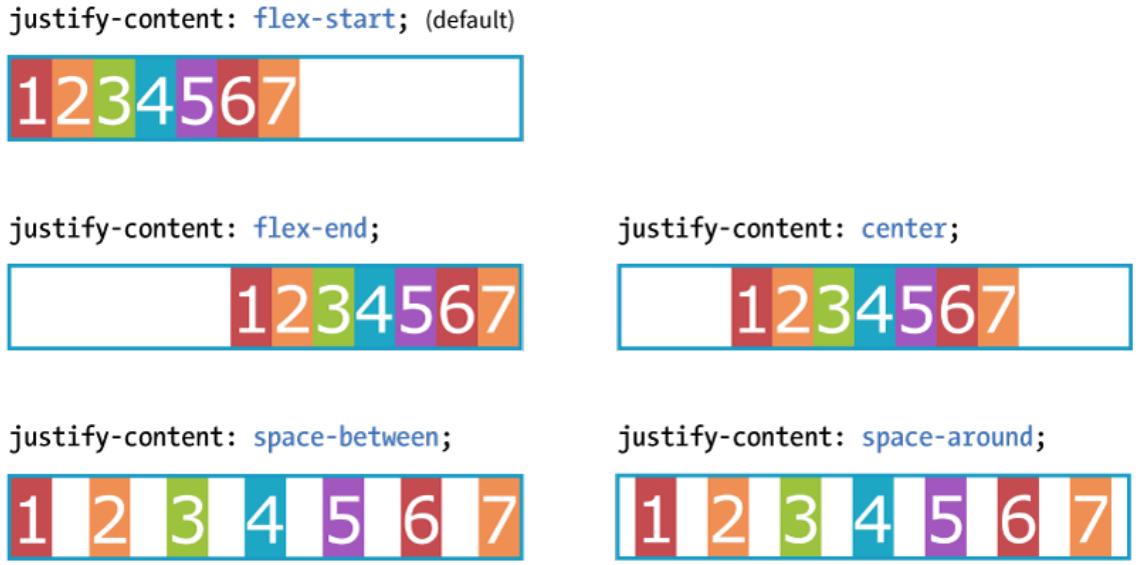
When there is space left over on the **main axis**, you can specify how the items align with the `justify-content` property (notice we say *start* and *end* instead of left/right or top/bottom).

The `justify-content` property applies to the **flex container**.

Example:

```
#container {
  display: flex;
  justify-content: flex-start;
}
```

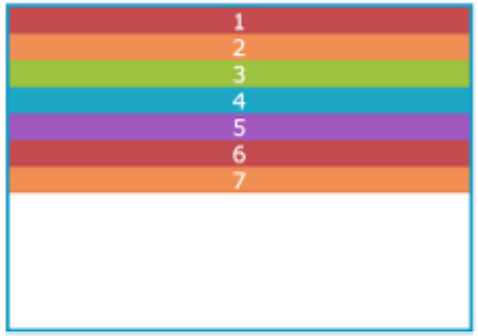
When the direction is **row**, and the **main axis is horizontal**



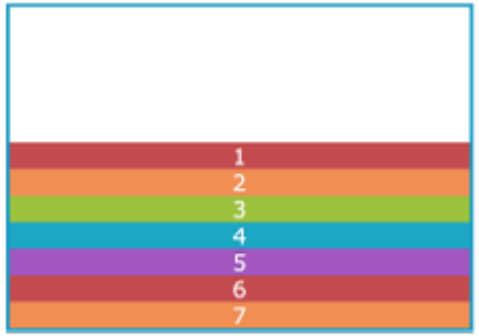
Aligning on the Main Axis (cont'd.)

When the direction is **column**, and the **main axis is vertical**

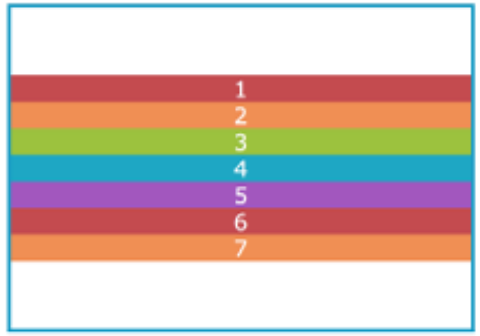
`justify-content: flex-start;` (default)



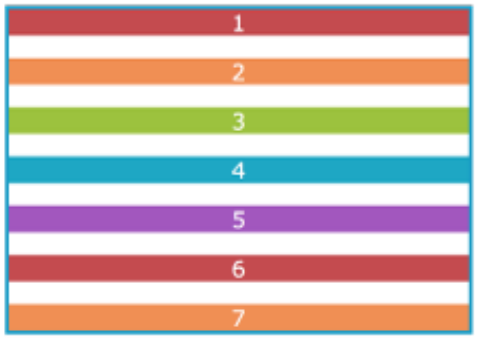
`justify-content: flex-end;`



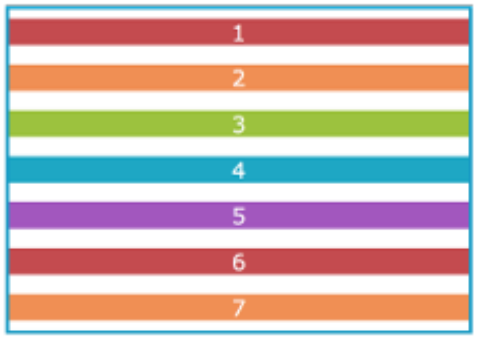
`justify-content: center;`



`justify-content: space-between;`



`justify-content: space-around;`



NOTE: I needed to specify a height on the container to create **extra space** on the main axis. By default, it's just high enough to contain the content.

A WORD FROM THE AUTHOR

“Keeping the main and cross axes straight in your mind when changing between rows and columns is one of the trickiest parts of using Flexbox.

Once you master that, you’ve got it!”

—Jennifer Robbins

Aligning on the Cross Axis

`align-items`

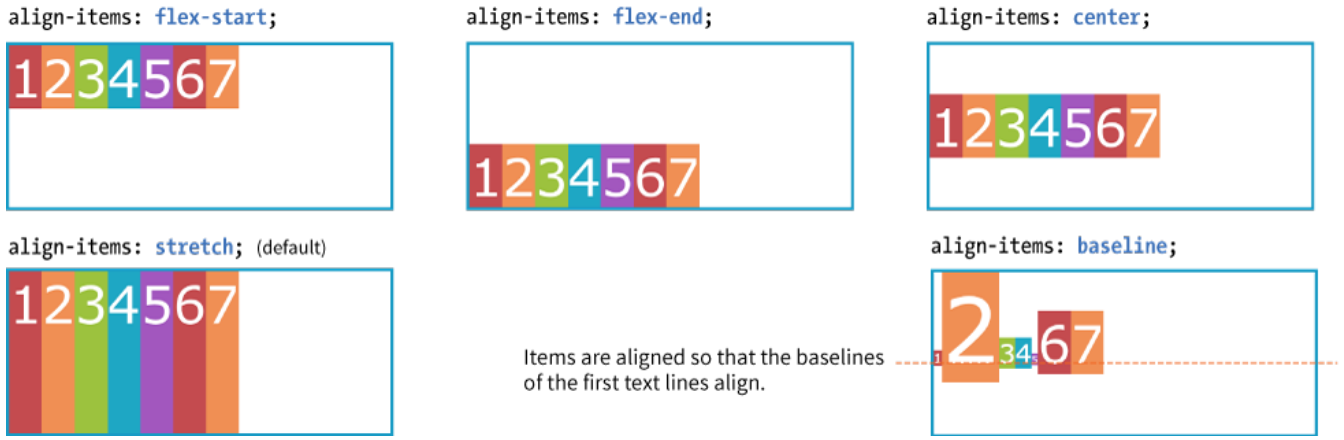
Values: `flex-start`, `flex-end`, `center`, `baseline`, `stretch`

When there is space left over on the **cross axis**, you can specify how the items align with the **`align-items`** property.

The **`align-items`** property applies to the **flex container**.

Example:

```
#container {
  display: flex;
  flex-direction: row;
  height: 200px;
  align-items: flex-start;
}
```



When the direction is **row**, the main axis is horizontal, and the **cross axis is vertical**. (need to specify extra height in the container)

Aligning on the CROSS Axis (cont'd)

`align-self`

Values: `flex-start`, `flex-end`, `center`, `baseline`, `stretch`

Aligns an **individual item** on the cross axis. This is useful if one or more items should override the `align-items` setting for the container.

The `align-self` property applies to the **flex item**.

Example:

```
.box4 {
  align-self: flex-end;
}
```

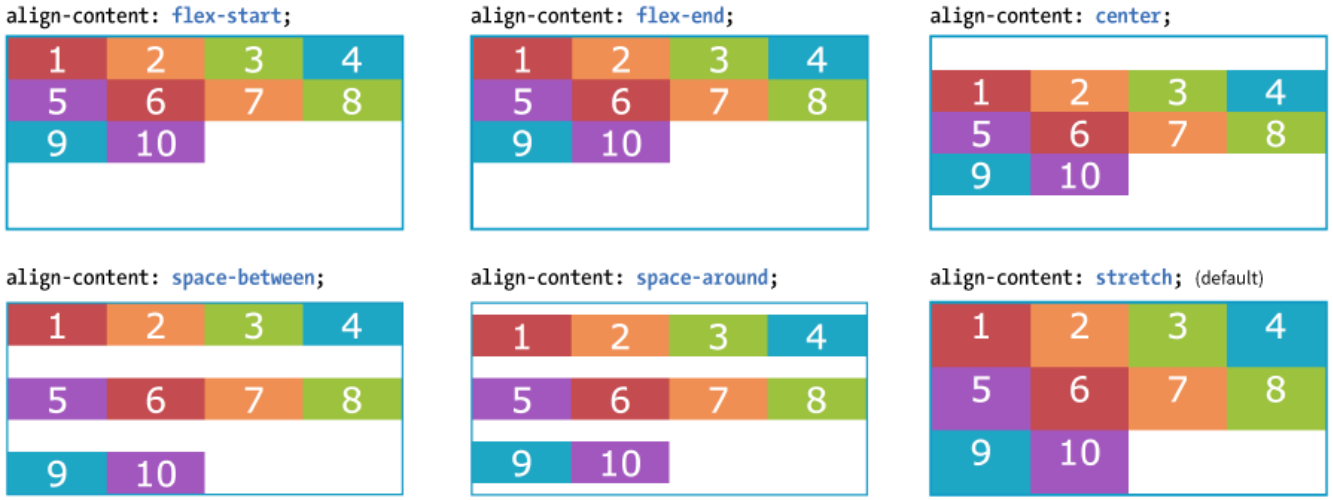
Aligning on the CROSS Axis (cont'd)

`align-content`

Values: `flex-start`, `flex-end`, `center`, `space-around`, `space-between`, `stretch`

When lines are set to **wrap** and there is extra space on the cross axis, use `align-content` to **align the lines of content**.

The `align-content` property applies to the **flex container**.



Aligning with Margins

- Use a margin (set to auto) to put extra space on the side of particular flex items.

- **Example:** Adding an auto margin to the right of the first flex item (the `li` with the logo) pushes the remaining `li` to the right:


```
ul {
    display: flex;
    align-items:    center;
    ...
}
li.logo {
    margin-right: auto;
}
```

Specifying How Items “Flex”

`flex`

Values: `none`, `'flex-grow flex-shrink flex-basis'`

- Items can resize (`flex`) to fill the available space on the main axis in the container.
- The `flex` property identifies how much an item can grow and shrink and identifies a starting size
- It distributes extra space in the container *within* items (compared to `justify-content` that distributes space *between and around* items).

flex Property Example

`flex` is a shorthand for separate `flex-grow`, `flex-shrink`, and `flex-basis` properties.

The values 1 and 0 work like on/off switches.

```
li {
  flex: 1 0 200px;
}
```

In this example, list items in the flex container start at 200 pixels wide, are permitted to expand wider (`flex-grow: 1`), and are not permitted to shrink (`flex-shrink: 0`).

NOTE: The spec recommends always using the `flex` property and using individual properties only for overrides.

Expanding Items (flex-grow)

`flex-grow`

Values: *Number*

Specifies whether and in what proportion an item may stretch larger.
1 allows expansion; 0 prevents it.

`flex-grow` is applied to the **flex item element**.

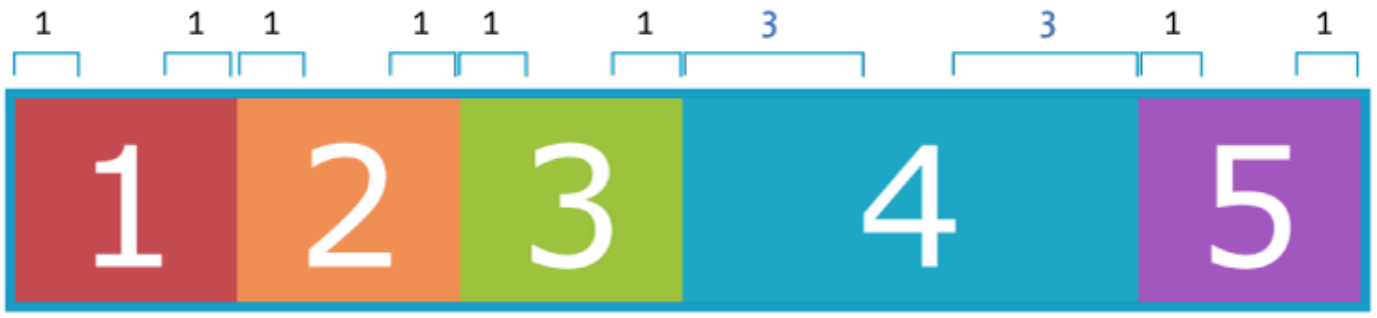
Expanding Items (cont'd)

Relative Flex

When the `flex-basis` *has a value* ***other than 0***, higher integer values act as a ratio that applies more space within that item.

Example: A value of `3` assigns **three times more space** to box4 than items with a `flex-grow` value of `1`. (Note that it isn't necessarily 3x as wide as the other items.)

```
.box4 { flex: 3 1 auto; }
```



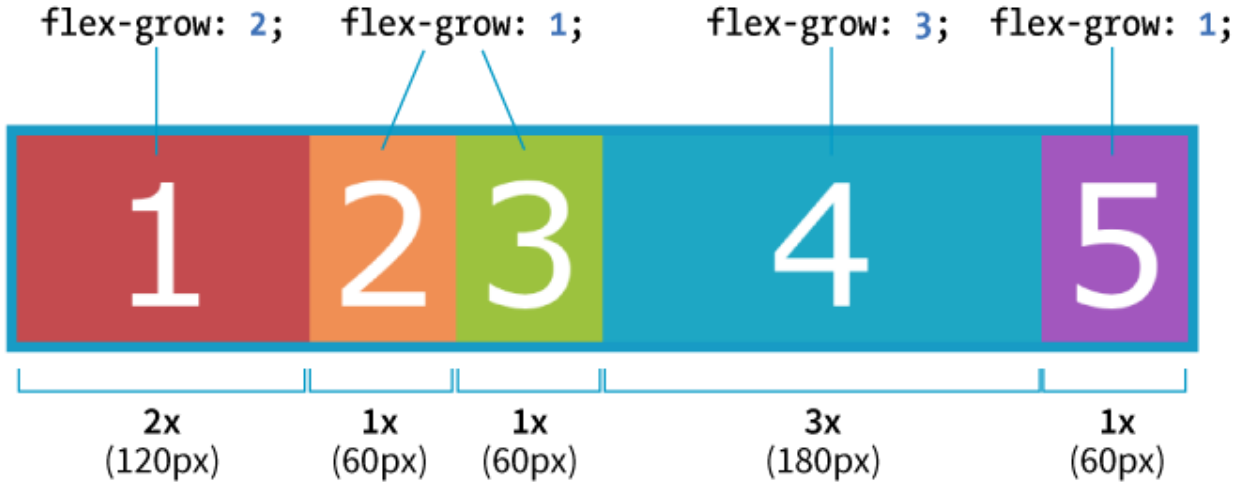
Expanding Items (cont'd.)

Absolute Flex

When the `flex-basis` *is* ***0***, items get sized proportionally according to the flex ratio.

Example: A value of `3` makes "box4" **3x as wide** as the others when `flex-basis: 0`.

```
.box4 { flex: 3 1 0%; }
```



Shortcut flex Values

- `flex: initial` (same as `flex: 0 1 auto;`)
  Prevents the item from growing, but allows it to shrink to fit the container
- `flex: auto` (same as `flex: 1 1 auto;`)
  Allows items to be fully flexible as needed. Size is based on the width/height properties.
- `flex: none` (same as `flex: 0 0 auto;`)
  Creates a completely **inflexible item** while sizing it to the width/height properties.
- `flex: integer` (same as `flex: integer 1 0px;`)
  Creates a flexible item with **absolute flex** (so `flex-grow` integer values are applied proportionally)

Changing Item Order

`order`

Values: ***Number***

Specifies the order in which a particular item should appear in the flow (independent of the HTML source order):

`order` is applied to the **flex item element**.

The default is 0. Items with the same order value are placed according to their order in the source.

Items with different order values are arranged from lowest to highest.

The specific number value doesn't matter; only how it relates to other values (like z-index) matters.

Changing Item Order (cont'd)

Example:

"box3" has a higher order value (1) than the others with default order of 0. It appears last in the line even though it's third in the markup:

```
.box3 {
  order: 1;
}
```

Changing Item Order (cont'd)

Ordinal groups

Items that share the same order value are called an ordinal group.

Ordinal groups stick together and are arranged from lowest value to highest:

```
.box2, .box3 {
  order: 1;
}
```

Browser Support for Flexbox

The Flexbox spec changed over the years and was implemented by browsers along the way:

- **Current version (2012):** `display: flex;`
  Supported by all current desktop and mobile browser versions
- **“Tweener” version (2011):** `display: flexbox;`
  Supported by IE10 only
- **Old version (2009):** `display: box;`
  Supported by Chrome <21, Safari 3.1–6, Firefox 2–21; iOS 3.2–6.1, Android 2.1–4.3

Browser Support (cont'd)

- To ensure that Flexbox works across all supporting browsers, you need a lot of vendor prefixes and redundant declarations.

- Use a tool like **Autoprefixer** to generate all that code for you (autoprefixer.github.io).

Flexbox Property Review

Flex container properties

```
display
flex-flow
    flex-direction
    flex-wrap
justify-content
align-items
align-content
```

Flex item properties

```
align-self
flex
    flex-grow
    flex-shrink
    flex-basis
order
```

CSS Layout with GRID

Grid terminology
Grid display type
Creating the grid template
Naming grid areas
Placing grid items
Implicit grid behavior
Grid spacing and alignment

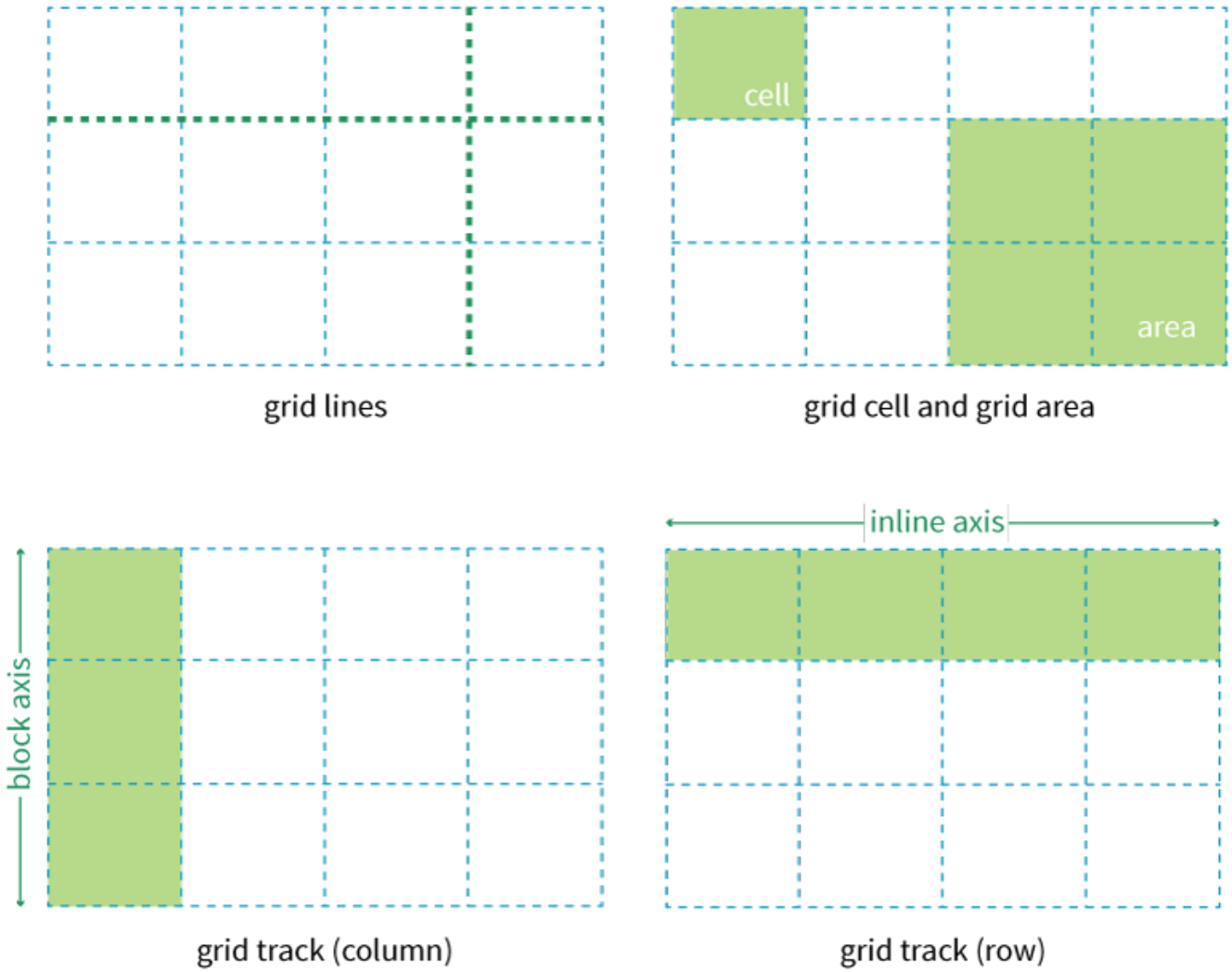
Grid vs. Flexbox

- Flexbox lays out elements on **one axis only**
- Grid lay out elements in rows and column – completely flexible to fit a variety of screen size or mimic a print page layout
- Grid Layout Module is one of the more **complex** specs in CSS
- Here is for a good head start to Grid

How CSS Grids Work

1. Set an element's `display` to `grid` to establish a **grid container**. Its children become **grid items**.
2. Set up the columns and rows for the grid (explicitly or with rules for how they are created on the fly).
3. Assign each grid item to an area on the grid (or let them flow in automatically in sequential order).

Grid Terminology



Creating a Grid Container

- To make an element a **grid container**, set its **`display`** property to **`grid`**.

- All of its children automatically become **grid items**.

The markup

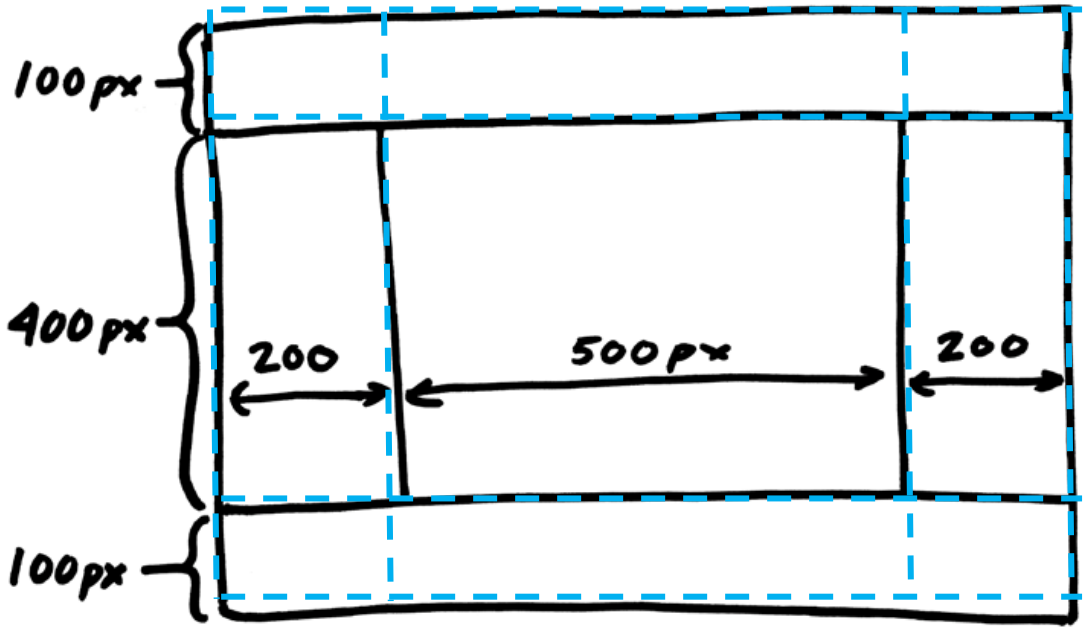
```
<div id="layout">
  <div id="one">One</div>
  <div id="two">Two</div>
  <div id="three">Three</div>
  <div id="four">Four</div>
  <div id="five">Five</div>
</div>
```

The styles

```
#layout {
  display: grid;
}
```

Setting up the grid

- Create a sketch of how you like your final grid to look



The blue dotted lines shows how many rows and column the grid requires to create this structure

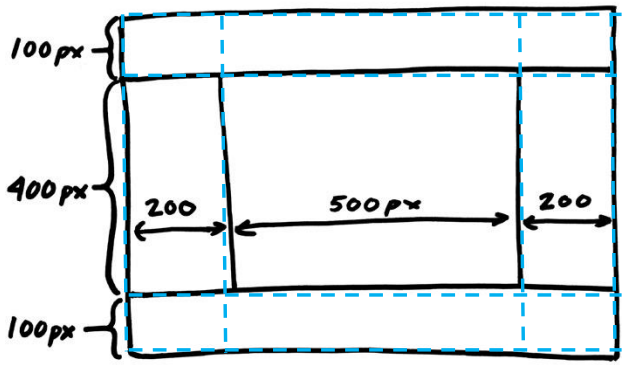
Defining Row and Column Tracks

`grid-template-rows`
`grid-template-columns`

Values: `none`, *list of* **track** *sizes and optional line names*

- The value of `grid-tempate-rows` is a list of the *heights* of each row track in the grid.
- The value of `grid-template-columns` is a list of the *widths* of each column track in the grid.

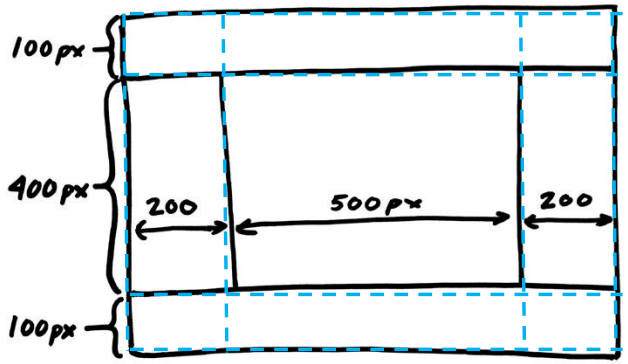
```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 500px 200px;
}
```

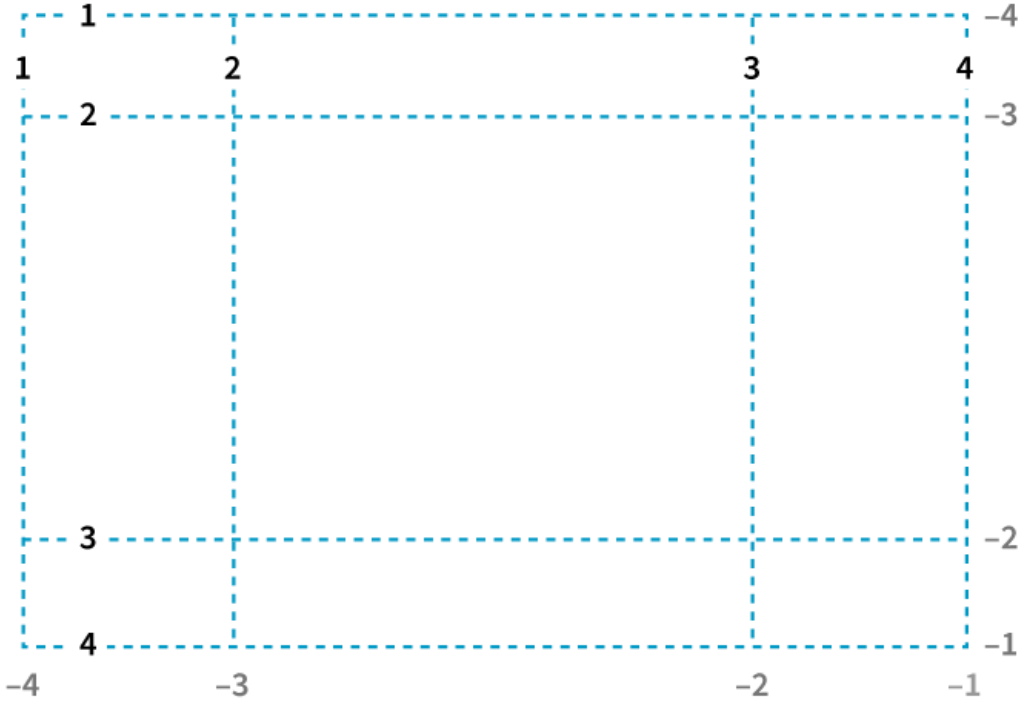


- The number of sizes provided determines the number of rows/columns in the grid.
- This grid in the example above has 3 rows and 3 columns.

Grid Line Numbers

Browsers assign a number to every grid line automatically, starting with **1** from the beginning of each row and column track and also starting with **–1** from the end.

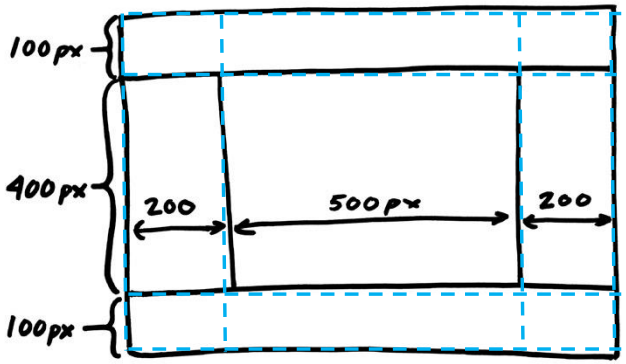




Grid Line Names

- **You can also assign names to lines** to make them more intuitive to reference later.
- Grid line names are added in square brackets in the position they appear relative to the tracks.
- To give a line more than one name, include all the names in brackets, separated by spaces.

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [header-end content-start]
400px [content-end footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 500px [links] 200px;
}
```



Track Size Values

The CSS Grid spec provides a *lot* of ways to specify the width and height of a track. Some of these ways allow tracks to adapt to available space and/or to the content they contain:

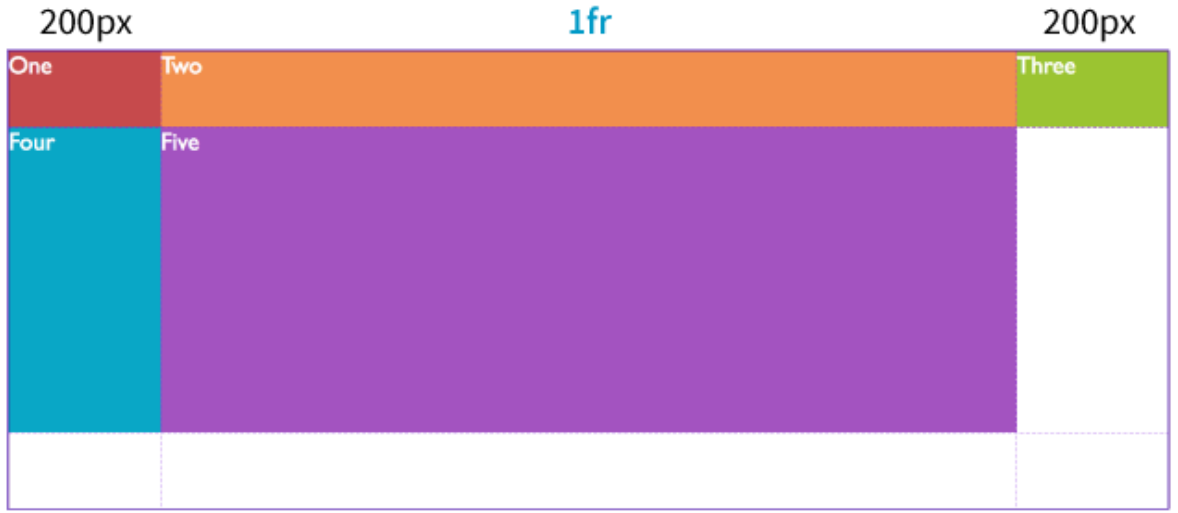
- Lengths (such as `px` or `em`)
- Percentage values (`%`)
- **Fractional units (`fr`)**
- `minmax()`
- `min-content`, `max-content`
- `auto`
- `fit-content()`

Tips: You can try to use Firefox's CSS Grid Inspector and Layout Panel, or some extension of Chrome (eg: Gridman) to see Grid layout of a specific web page

Fractional Units (fr)

The Grid-specific fractional unit (**fr**) expands and contracts based on available space:

```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 1fr 200px;
}
```



Size Range with minmax()

- The `minmax()` function constricts the size range for the track by setting a minimum and maximum dimension.
- It’s used in place of a specific track size.
- This rule sets the middle column to at least 15em but never more than 45em:

```
grid-template-columns: 200px minmax(15em, 45em) 200px;
```

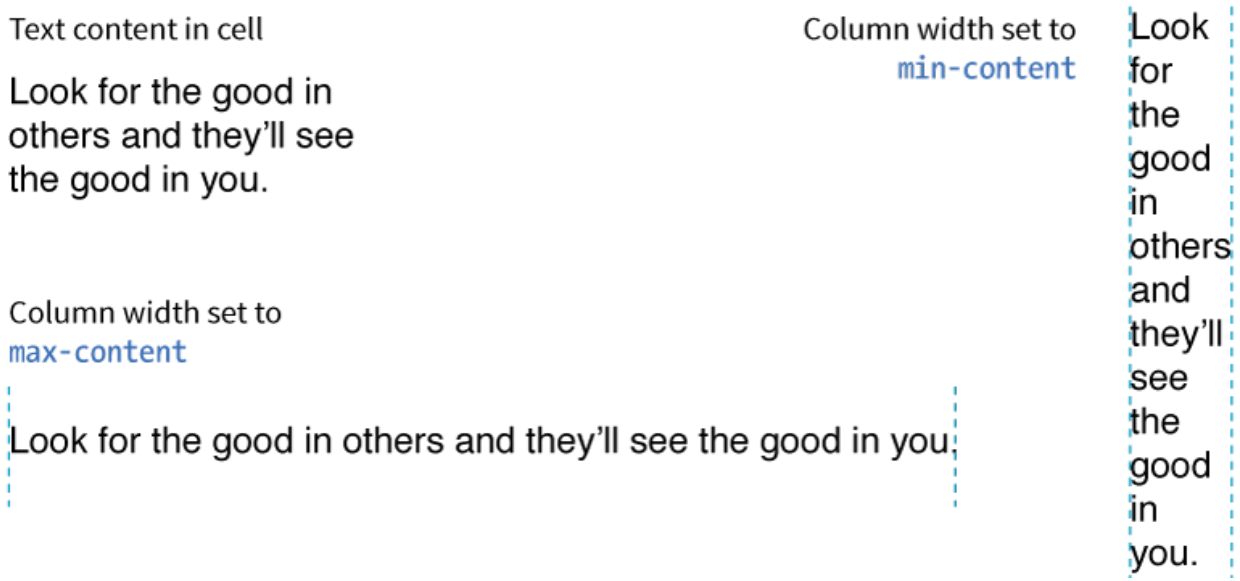
Content-based sizing

`min-content` is the smallest that a track can be depending on the content.

`max-content` allots the maximum amount of space needed for a content.

`auto` lets the browser take care of it.

** If you are not sure which to use, start with `auto` for content-based sizing.



Repeating Track Sizes

The shortcut **`repeat()`** function lets you repeat patterns in track sizes:

`repeat(#,` *`track pattern`*`)`

The first number is the number of repetitions. The track sizes after the comma provide the pattern:

This

```
grid-template-columns: 200px 20px 1fr 20px 1fr 20px 1fr
20px 1fr 20px 1fr 20px 1fr 200px;
```

Is equivalent to

```
grid-template-columns: 200px repeat(5, 20px 1fr) 200px;
```

(Here repeat() is used in a longer sequence of track sizes.
It repeats the track sizes 20px 1fr 5 times.)

Repeating Track Sizes (cont'd.)

You can let the browser figure out how many times a repeated pattern will fit with **`auto-fill`** and **`auto-fit`** values instead of a number:

```
grid-template-rows: repeat(auto-fill, 15em);
```

`auto-fill` creates as many tracks as will fit in the available space, even if there's not enough content to fill all the tracks. (Fills the row with as many columns

`auto-fit` creates as many tracks as will fit, dropping empty tracks from the layout. (Fits the currently available rows into the space by expanding them)

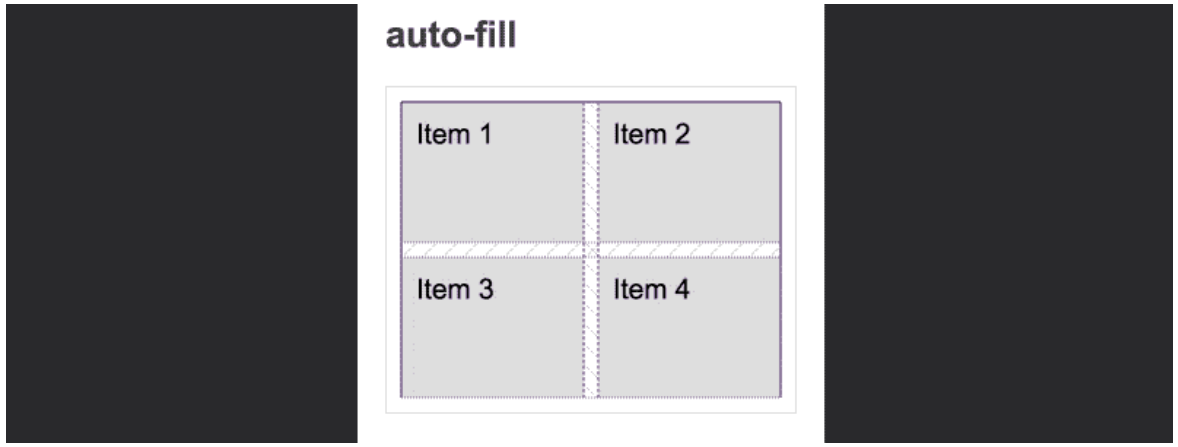
NOTE: If there's leftover space in the container, it's distributed according to the provided vertical and horizontal alignment values.

auto-fill and auto-fit demo

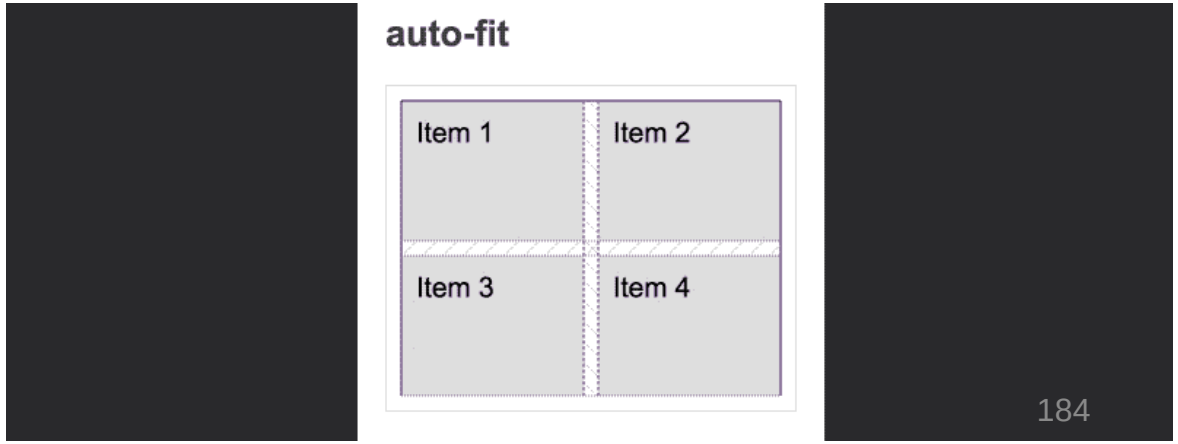
- Auto-fill will fill the extra space with a new empty grid
  So if the width of the parent element is widen, there is extra space, a new grid will be populated

- When using auto-fit, the extra space in the parent element is eliminated to be `0px`.
  Therefore, a grid item width of `1fr` will fill the extra space

```
grid-template-columns: repeat(auto-fill,
minmax(100px, 1fr));
```




```
grid-template-columns: repeat(auto-fit,
minmax(100px, 1fr));
```



Giving Names to Grid Areas

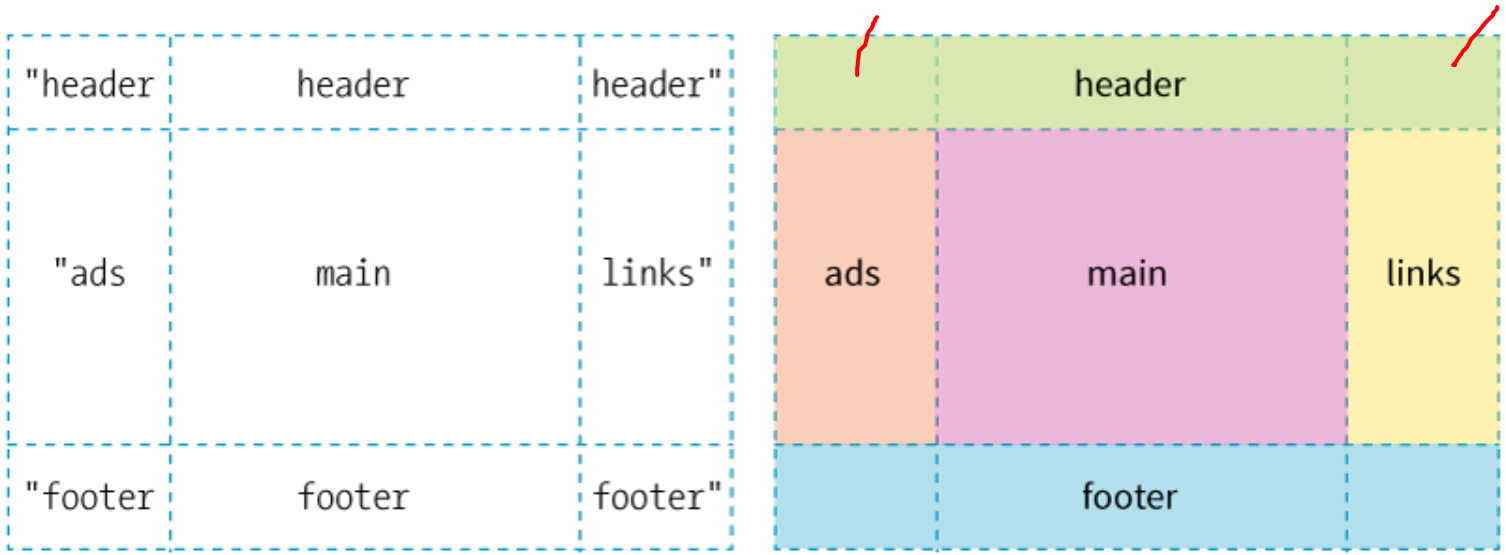
`grid-template-areas`

Values: `none`, *series of area names by row*

- `grid-template-areas` lets you assign names to areas in the grid to make it easier to place items in that area later.
- The value is a list of names for every cell in the grid, **listed by row**.
- When neighboring cells share a name, they **form a grid area** with that name.

Giving Names to Grid Areas (cont'd)

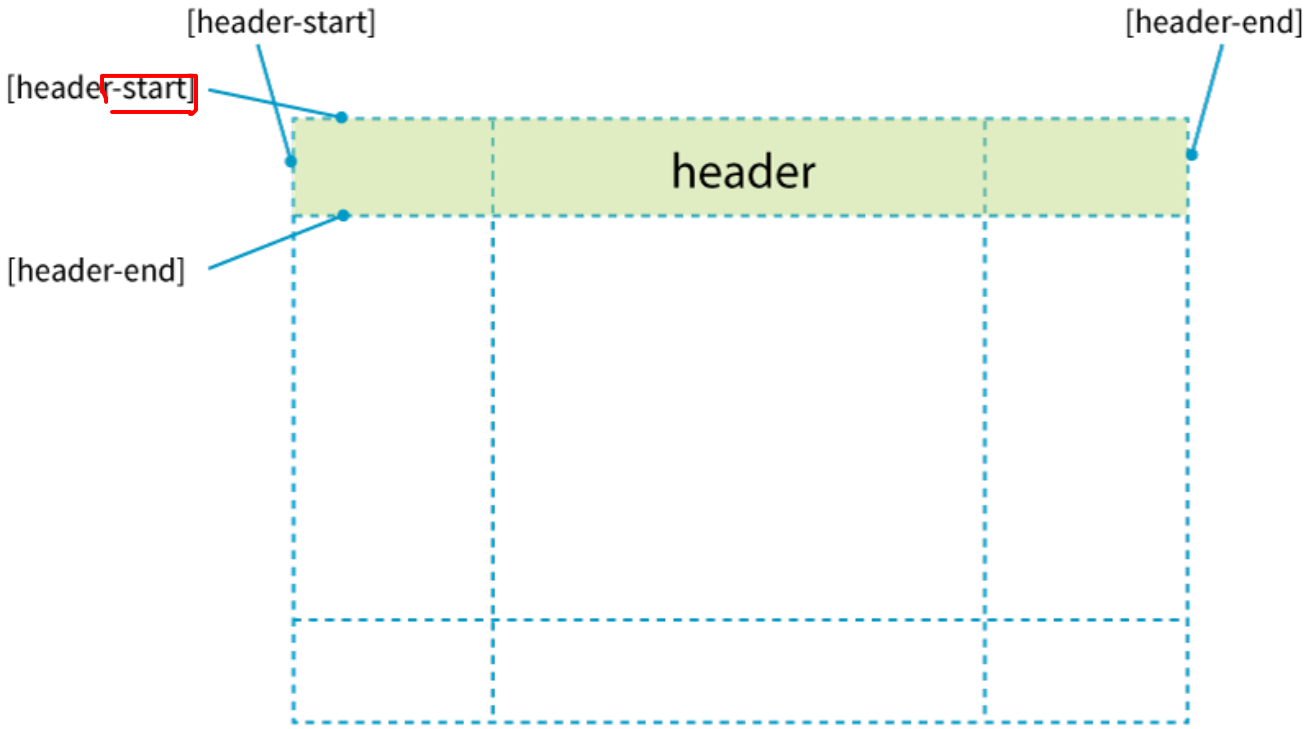
```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer"
}
```



Giving Names to Grid Areas (cont'd)

- Assigning names to lines with -start and -end suffixes creates an area name **implicitly**.

- Similarly, when you specify an area name with **`grid-template-areas`**, line names with -start and -end suffixes are **implicitly generated**.



The grid Shorthand Property

grid

Values: `none`, *`row info/column info`*

The `grid` shorthand sets values for `grid-template-rows`, `grid-template-columns`, and `grid-template-areas`.

NOTE: The `grid` shorthand is available, but the word on the street is that it's more difficult to use than separate template properties.

Example grid: rows / columns

```
#layout {
  display: grid;
  grid: 100px 400px 100px / 200px 1fr 200px;
}
```


See more about Grid and shorthand property at /https://css-tricks.com/snippets/css/complete-guide-grid

Quite complicated method

Placing Items Using Grid Lines

Specify the bordering lines of the target grid area

```
grid-row-start
grid-row-end
grid-column-start
grid-column-end
```

Values: `auto`, *`"line name"`*, `span` *`number`*, `span` *`"line name"`*, `number` *`"line name"`*

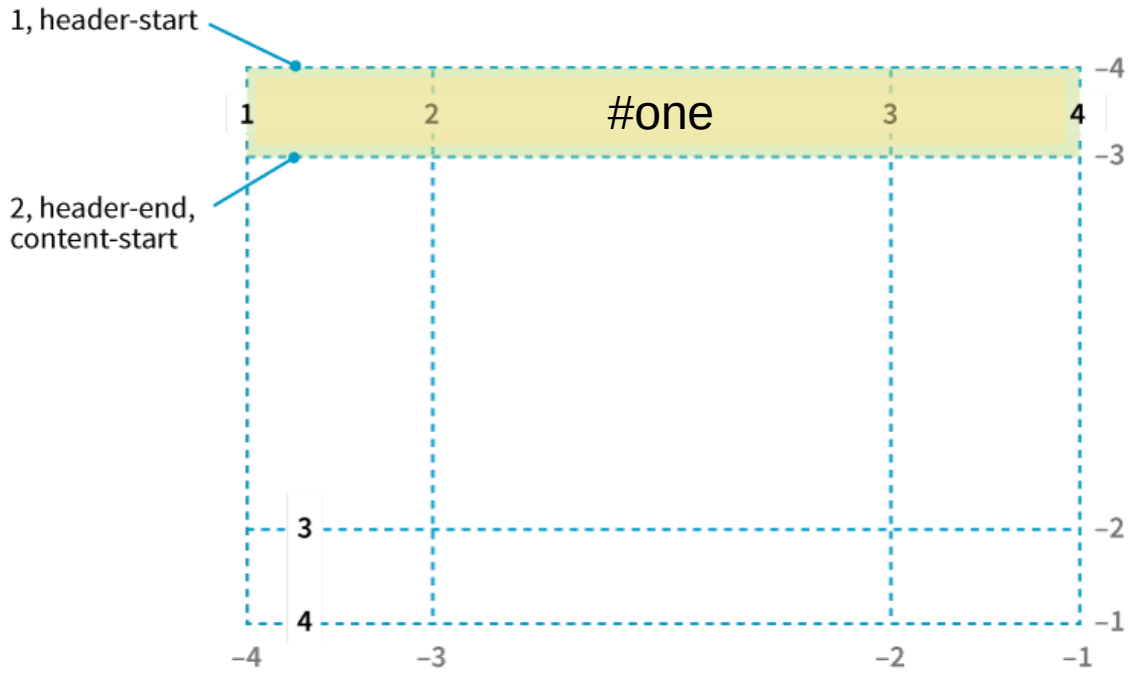
- These properties position grid items on the grid by referencing the grid lines where they begin and end.
- The property is applied to the **grid item** element.

Quite complicated method

Placing Items on the Grid (cont'd)

By line number:

```
#one {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 4;
}
```



Using a span:

Span a certain number of cells

```
#one {
  ...
  grid-column-start: 1;
  grid-column-end: span 3;
}
```

Starting from the last grid line and spanning backward:

-1 is 1 line from the last line

```
#one {
  ...
  grid-column-start: span 3;
  grid-column-end: -1;
}
```

By line name:

```
#one {
  grid-row-start: header-
start;
  grid-row-end: header-end;
  …
}
```

Placing Items on the Grid Using Areas

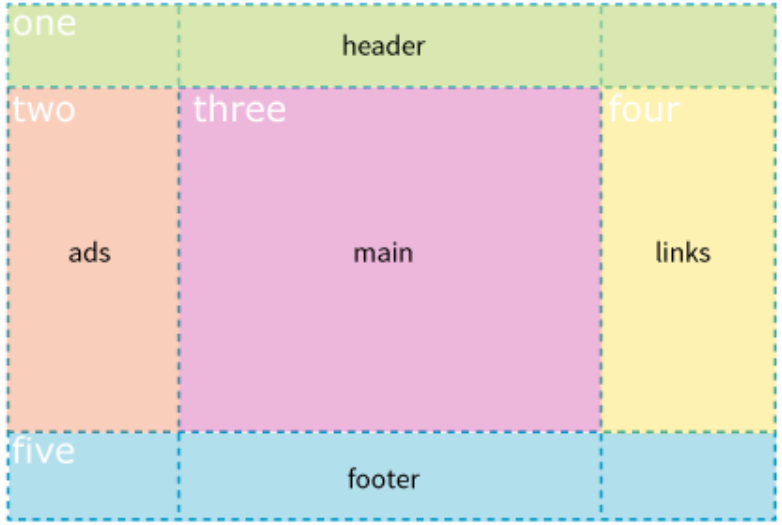
The easy method

`grid-area`

Values: *`Area name, 1 to 4 line identifiers`*

Positions an item in an area created with `grid-template-areas`:

```
#one { grid-area: header; }
#two { grid-area: ads; }
#three { grid-area: main; }
#four { grid-area: links; }
#five { grid-area: footer; }
```



Implicit Grid Behavior

The Grid Layout system does some things for you **automatically** (**implicit behavior**):

- Generating “-start” and “-end” line names when you name an area (and vice versa)
- Flowing items into grid cells sequentially if you don’t explicitly place them
- Adding rows and columns on the fly as needed to fit items

Automatically Generated Tracks

`grid-auto-rows`
`grid-auto-columns`

Values: *List of track sizes*

Provide one or more track sizes for automated tracks. If you provide more than one value, it acts as a repeating pattern.

Example:

Column widths are set explicitly with a template, but columns will be generated automatically with a height of 200 pixels:

```
grid-template-columns: repeat(3, 1fr);
grid-auto-rows: 200px;
```

Flow Direction and Density

`grid-auto-flow`

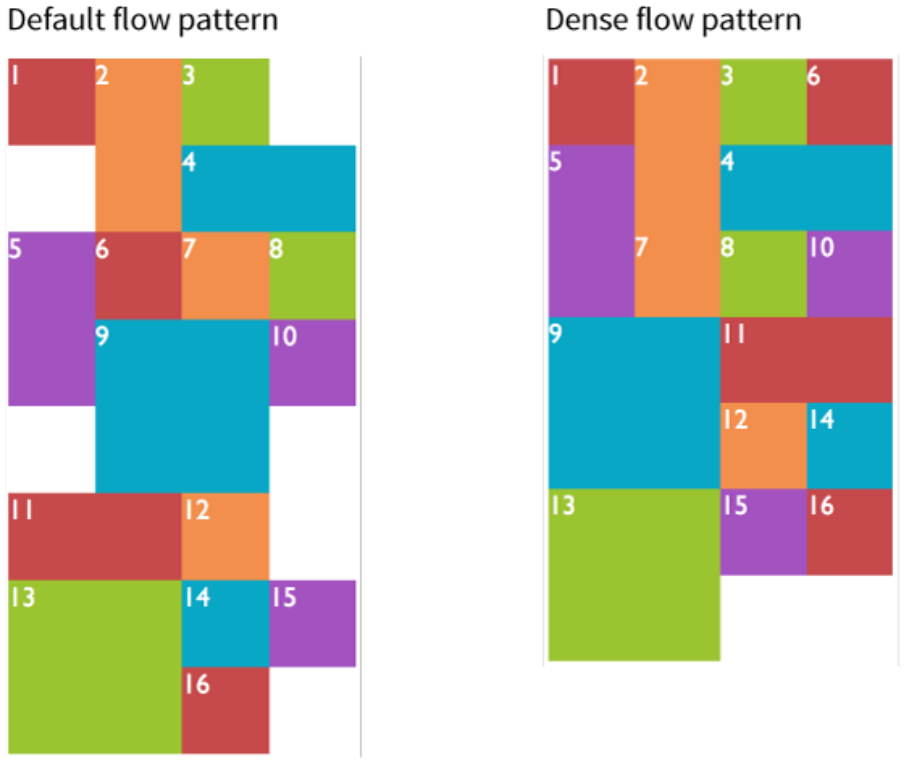
Values: `row` *or* `column`, `dense` *(optional)*

Specifies whether you'd like items to flow in by `row` or `column`. The default is the writing direction of the document.

Example:

```
#listings {
  display: grid;
  grid-auto-flow: column dense;
}
```

The dense keyword instructs the browser to fill in the grid as densely as possible, allowing items to appear out of order.



The Grid Shorthand Property (Revisited)

- Use the `auto-flow` keyword in the shorthand `grid` property to indicate that the rows or columns should be generated automatically.

- **Example:**
  Columns are established explicitly, but the rows generate automatically. *(Remember, row information goes before the slash.)*

```
grid: auto-flow 200px / repeat(3, 1fr);
```

- Because `auto-flow` is included with row information, `grid-auto-flow` is set to `row`.

Spacing Between Tracks

`grid-row-gap`

`grid-column-gap`

Values: *`Length`* *(must not be negative)*

`grid-gap`

Values: *`grid-row-gap grid-column-gap`*

Adds space between the row and/or columns tracks of the grid

NOTE: These property names will be changing to *`row-gap`*, *`column-gap`*, and *`gap`*, but the new names are not yet supported.

Space Between Tracks (cont'd)

- If you want equal space between all tracks in a grid, use a gap instead of creating additional spacing tracks:

```
grid-gap: 20px 50px;
```

(Adds 20px space between rows and 50px between columns)

Item Alignment

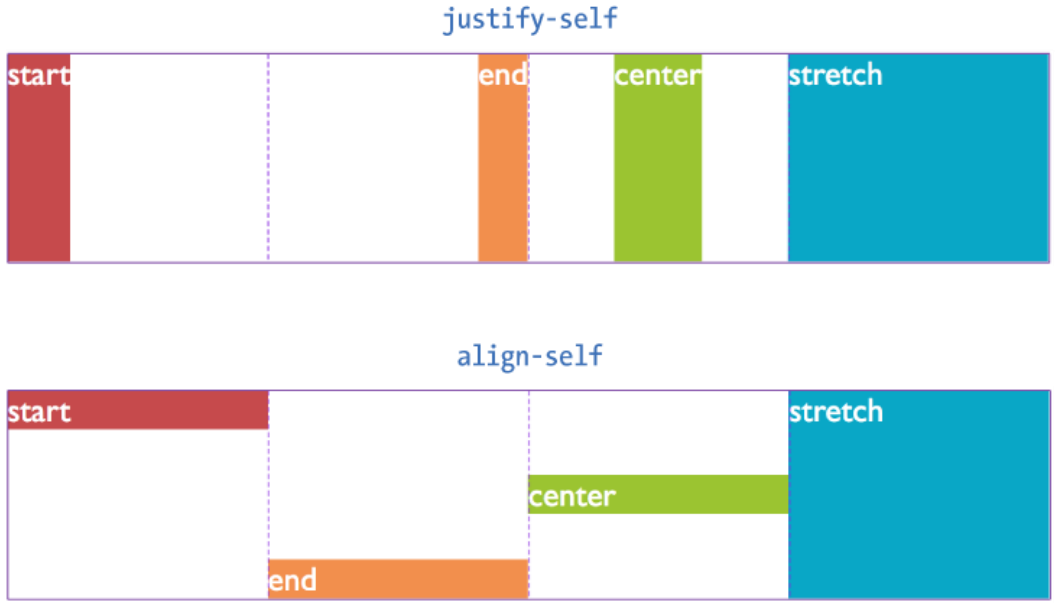
`justify-self`
`align-self`

Values: `start`, `end`, `center`, `left`, `right`, `self-start`, `self-end`, `stretch`, `normal`, `auto`

When an item element doesn't fill its grid cell, you can specify how it should be aligned within the cell.

`justify-self` aligns on the **inline** axis (horizontal for L-to-R languages).

`align-self` aligns on the **block** (vertical) axis.



NOTE: These properties are applied to the individual **grid item** element.

Aligning All the Items

`justify-items`
`align-items`

Values: `start`, `end`, `center`, `left`, `right`, `self-start`, `self-end`, `stretch`, `normal`, `auto`

These properties align items in their cells all at once. They are applied to the **grid container**.

`justify-items` aligns on the **inline** axis.

`align-items` aligns on the **block** (vertical) axis.

Track Alignment

`justify-content`
`align-content`

Values: `start`, `end`, `center`, `left`, `right`, `stretch`, `space-around`, `space-between`, `space-evenly`

When the **grid tracks do not fill the entire container**, you can specify how tracks align.

`justify-content` aligns on the **inline** axis (horizontal for L-to-R languages).

`align-content` aligns on the **block** (vertical) axis.

Track Alignment (cont'd)

justify-content:

start

end

center

space-around

space-between

space-evenly

align-content:

start

end

center

space-around

space-between

space-evenly

NOTE: These properties are applied to the **grid container**.

Grid Property Review

Grid container properties

```
display: grid | inline-grid
grid
   grid-template
      grid-template-rows
      grid-template-columns
      grid-template-areas
  grid-auto-rows
  grid-auto-columns
  grid-auto-flow
grid-gap
  grid-row-gap
  grid-column-gap
justify-items
align-items
justify-content
align-content
```

Grid item properties

```
grid-column
   grid-column-start
   grid-column-end
grid-row
   grid-row-start
   grid-row-end
grid-area
justify-self
align-self
order  (not part of Grid Module)
z-index  (not part of Grid Module)
```