Introduction to the Box Model

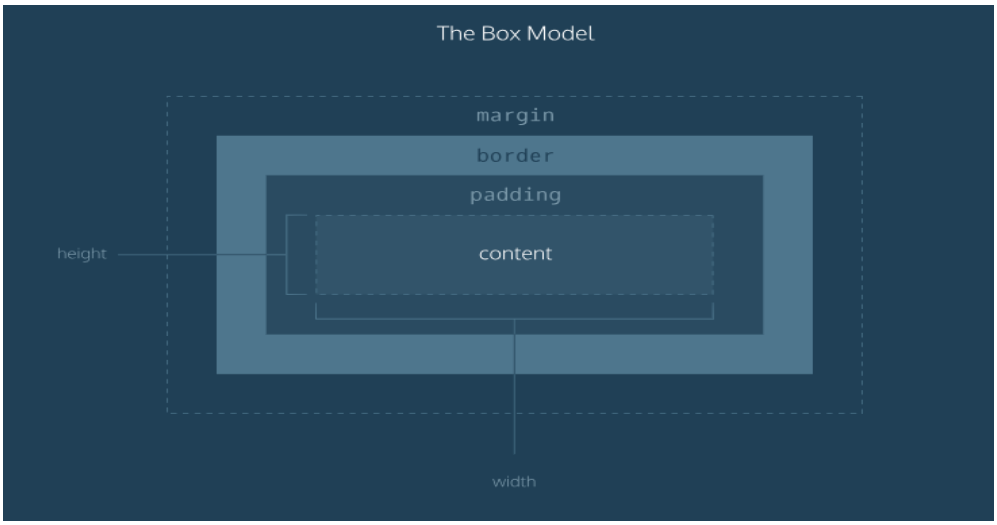
If you have used HTML and CSS, you have unknowingly seen aspects of the box model. For example, if you have set the background color of an element, you may have noticed that the color was applied not only to the area directly behind the element, but also to the area to the right of the element. Also, if you have aligned text, you know it is aligned relative to something. What is that something?

All elements on a web page are interpreted by the browser as "living" inside of a box. This is what is meant by the box model.

For example, when you change the background color of an element, you change the background color of its entire box

1. **The dimensions of an element's box.**
2. **The borders of an element's box.**
3. **The paddings of an element's box.**
4. **The margins of an element's box.**

The box model comprises the set of properties which define parts of an element that take up space on a web page.



How to insert text on image

```
  <div id="banner">
    <div class="content">
      <h1>Conservation Efforts at Lake Tahoe Being Praised by Nation's
Leaders</h1>
    </div>
  </div>
```

```
#banner {
    background-image: url("https://s3.amazonaws.com/learncodehtml-
content/htmlcss1-img_tahoe.jpeg");
    background-size: cover;
    background-position: bottom center;
    width: 100%;
    height:700px;
  }
```

```
#banner .content h1 {
    position: relative;
    top: 50px;
    margin: 0 auto;
    width:400px;
    border:3px solid  white;
  }
```

Height and Width

When the width and height of an element are set in pixels, it will be the same size on all devices—an element that fills a laptop screen will overflow a mobile screen.

```
p {
    height: 80px;
```

```
    width: 240px;
  }
```

Borders

A *border* is a line that surrounds an element, like a frame around a painting.

1. width — **The thickness of the border , it can be set in pixels or** thin, medium, **or** thick.
2. style — **The design of the border, there are 10 different styles. styles include:** none, dotted, **and** solid.
3. color — **The color of the border. 140 built-in color keywords.**

```
p {
    border: 3px solid coral;
  }
```

The default border is `medium none color`

Border Radius

You can modify the corners of an element's border box with the `border-radius` **property.**

```
div.container {
border: 3px solid rgb(22, 77, 100); border-radius: 5px;
 }
```

You can create a border that is a perfect circle by setting the radius equal to the height of the box, or to `100%`**.**

```
div.container {
height: 60px;
width: 60px;
 border: 3px solid rgb(22, 77, 100);
border-radius: 100%; }
```

The code in the example above creates a `<div>` **that is a perfect circle.**

Padding I

The space between the contents of a box and the borders of a box is known as *padding*.

```
p.content-header {
    border: 3px solid coral;
    padding: 10px;
  }
```

The code in this example puts 10 pixels of space between the content of the paragraph (the text) and the borders, on all four sides.

If you want to be more specific about the amount of padding on each side of a box's content, you can use the following properties:

1. padding-top
2. padding-right
3. padding-bottom
4. padding-left

```
p.content-header {
    border: 3px solid fuschia;
    padding-bottom: 10px;
    padding-top: 10px;
    padding-left: 10px;
    padding-right: 10px;
  }
```

Padding II

Another implementation of the `padding` property .

```
p.content-header {
    border: 3px solid grey;
    padding: 6px 11px 4px 9px;
  }
```

In order, it specifies the amount of padding on the top (6 pixels), right (11 pixels), bottom (4 pixels), and left (9 pixels) sides of the content.

Top/left equal and right/left padding equal then,

```
p.content-header {
     padding: 5px 10px;
  }
```

The first value, `5px`, for the top and bottom sides .The second value, `10px`, for the left and right sides of the content.

Margins I

Margin refers to the space directly outside of the box. The `margin` property is used to specify the size of this space.

```
p {
    border: 1px solid aquamarine;
    margin: 20px;
  }
```

The code in the example above will place 20 pixels of space on the outside of the paragraph's box on all four sides. This means that other HTML elements on the page cannot come within 20 pixels of the paragraph's border.

1. margin-top
2. margin-right
3. margin-bottom
4. margin-left

more flexible:

```
p {
    border: 3px solid DarkSlateGrey;
    margin-right: 15px;
    margin-left: 15px;
    margin-top: 15px;
    margin-bottom: 15px;
  }
```

Margins II

A similar implementation of the margin property is used to specify exactly how much margin there should be on each side of the box in a single declaration

```
p {
    margin: 6px 10px 5px 12px;
  }
```

In order, it specifies the amount of margin on the top (6 pixels), right (10 pixels), bottom (5 pixels), and left (12 pixels) sides of the box.

shortcut:

```
p {
    margin: 6px 12px;
  }
```

The first value, `6px`, sets a margin value for the top and bottom of the box. The second value, `12px`, sets a margin value for the left and right sides of the box.

Auto

The `margin` property also lets you center content.

```
div {
    margin: 0 auto;
  }
```

In the example above, `margin: 0 auto;` will center the `div`s in their containing elements. The `0`sets the top and bottom margins to 0 pixels. The `auto` value instructs the browser to adjust the left and right margins until the element is centered within its containing element.

The `div` elements in the example above should center within an element that fills the page, but this doesn't occur. Why?

In order to center an element, a width must be set for that element. Otherwise, the width of the div will be automatically set to the full width of its containing element, like the `<body>`, for example. It's not possible to center an element that takes up the full width of the page.

```
div.headline {
    width: 400px;
    margin: 0 auto;
  }
```

In the example above, the width of the div is set to 400 pixels, which is less than the width of most screens. This will cause the div to center within a containing element that is greater than 400 pixels wide.

```
<div id="main" class="content">full site content</div>

#main {
    padding: 40px;
    text-align: center;
    width: 400px;
    margin: 0 auto;
  }
```

Margin Collapse

One additional difference is that top and bottom margins, also called vertical margins, *collapse*, while top and bottom padding does not.

Margins collapse between adjacent elements. In simple terms, this means that for adjacent **vertical**block-level elements in the normal document flow, only the **margin** of the element with the largest**margin** value will be honored, while the **margin** of the element with the smaller **margin** value will be**collapsed** to zero.

Horizontal margins (left and right), like padding, are always displayed and added together. For example, if two divs with ids `#div-one` and `#div-two`, are next to each other, they will be as far apart as the sum of their adjacent margins.

```
#img-one {
    margin-right: 20px;
  }

  #img-two {
    margin-left: 20px;
  }
```

In this example, the space between the `#img-one` and `#img-two` borders is 40 pixels. The right margin of `#img-one` (`20px`) and the left margin of `#img-two` (`20px`) add to make a total margin of 40 pixels.

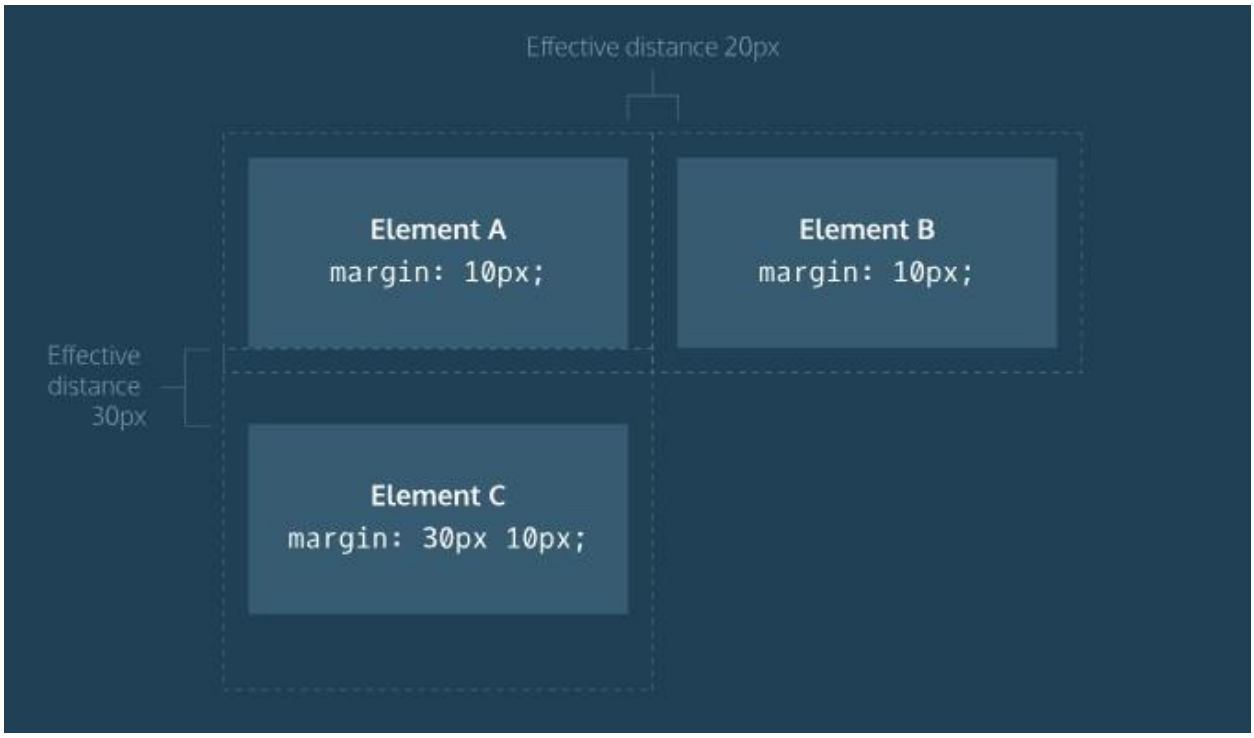
Unlike horizontal margins, vertical margins do not add. Instead, the larger of the two vertical margins sets the distance between adjacent elements.

```
#img-one {
```

```
  margin-bottom: 30px;
}

#img-two {
  margin-top: 20px;
}
```

In this example, the vertical margin between the `#img-one` and `#img-two` elements is 30 pixels. Although the sum of the margins is 50 pixels, the margin collapses so the spacing is only dependent on the `#img-one` bottom margin.



Minimum and Maximum Height and Width

Because a web page can be viewed through displays of differing screen size, the content on the web page can suffer from those changes in size.

1. `min-width` — **this property ensures a minimum width of an element's box.**
2. `max-width` — **this property ensures a maximum width of an element's box.**

```
p {
    min-width: 300px;
```

```
    max-width: 600px;
  }
```

In the example above, the width of all paragraphs will not shrink below 300 pixels, nor will the width exceed 600 pixels.

Content, like text, can become difficult to read when a browser window is narrowed or expanded. These two properties ensure that content is legible by limiting the minimum and maximum widths of an element.

You can also limit the minimum and maximum *height* of an element.

1. `min-height` — **this property ensures a minimum height for an element's box.**
2. `max-height` — **this property ensures a maximum height of an element's box.**

```
p {
    min-height: 150px;
    max-height: 300px;
  }
```

In the example above, the height of all paragraphs will not shrink below 150 pixels and the height will not exceed 300 pixels.

Overflow

The `overflow` property controls what happens to content that spills, or overflows, outside its box. It can be set to one of the following values:

- `hidden` **- when set to this value, any content that overflows will be hidden from view.**
- `scroll` **- when set to this value, a scrollbar will be added to the element's box so that the rest of the content can be viewed by scrolling.**
- `visible` **- when set to this value, the overflow content will be displayed outside of the containing element. Note, this is the default value.**

```
p {
    overflow: scroll;
  }
```

The overflow property is set on a parent element to instruct a web browser how to render child elements. For example, if a div's overflow property is set to `scroll`, all children of this div will display overflowing content with a scroll bar.

Resetting Defaults

```
* {
    margin: 0;
    padding: 0;
  }
```

The code in the example above resets the default margin and padding values of all HTML elements. It is often the first CSS rule in an external stylesheet.

Note that both properties are both set to `0`. When these properties are set to `0`, they do not require a unit of measurement.

Visibility

Elements can be hidden from view with the `visibility` property.

The `visibility` property can be set to one of the following values:

1. hidden — **hides an element.**
2. visible — **displays an element.**

```
<ul>
    <li>Explore</li>
    <li>Connect</li>
    <li class="future">Donate</li>
  <ul>
```

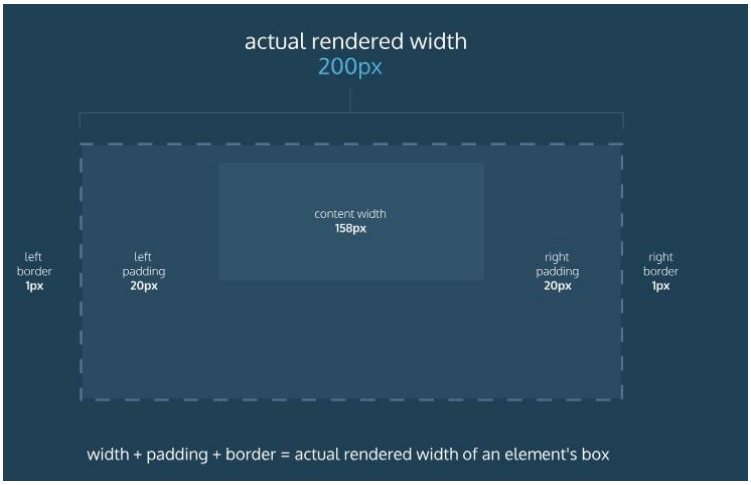
```
.future {
    visibility: hidden;
  }
```

In the example above, the list item with a class of `future` will be hidden from view in the browser.

Keep in mind, however, that users can still view the contents of the list item (e.g., `Donate`) by viewing the source code in their browser. Furthermore, the web page will *only* hide the contents of the element. It will still leave an empty space where the element is intended to display.

Note: **What's the difference between `display: none` and `visibility: hidden`? An element with `display: none` will be completely removed from the web page. An element with `visibility: hidden`, however, will not be visible on the web page, but the space reserved for it will.**

Box Model: Content-Box



Many properties in CSS have a default value and don't have to be explicitly set in the stylesheet.

For example, the default `font-weight` of text is `normal`, but this property-value pair is not typically specified in a stylesheet.

The same can be said about the box model that browsers assume. In CSS, the `box-sizing` property controls the type of box model the browser should use when interpreting a web page.

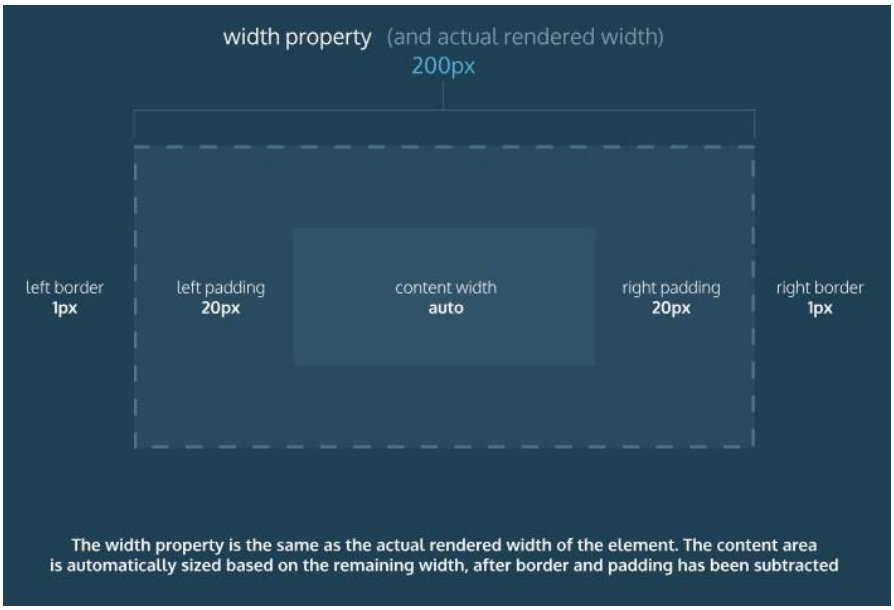
The default value of this property is `content-box`. This is the same box model that is affected by border thickness and padding.

Box Model: Border-Box

Fortunately, we can reset the entire box model and specify a new one: `border-box`

```
* {
    box-sizing: border-box;
  }
```

in the example above resets the box model to `border-box` for all HTML elements. This new box model (the universal selector (`*`) targets all elements on the web page and sets their box model to `border-box` model.) avoids the dimensional issues that exist in the former box model.



The height and width of the box will remain fixed. The border thickness and padding will be included inside of the box, which means the overall dimensions of the box do not change.

```
<h1>Hello World</h1>
```

.css file

```
* {
  box-sizing: border-box;
}

h1 {
  border: 1px solid black;
  height: 200px;
  width: 300px;
  padding: 10px;
}
```

the height of the box would remain at 200 pixels and the width would remain at 300 pixels. The border thickness and padding would remain entirely *inside* of the box

Flow of HTML

A browser will render(to show) the elements of an HTML document that has no CSS from left to right, top to bottom, in the same order as they exist in the document. This is called the *flow* of elements in HTML.

In addition to the properties that it provides to style HTML elements, CSS includes properties that change how a browser *positions* elements.

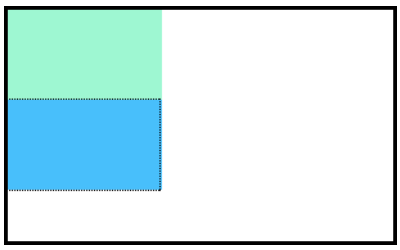
FIVE PROPERTIES

- position
- display
- z-index
- float
- clear

Each of these properties will allow us to position and view elements on a web page.

Position

Take a look at the *block-level* elements in the image below:



Block-level elements like these boxes create a *block* the full width of their parent elements, and they prevent other elements from appearing in the same horizontal space.

.CSS

```
.boxes {
  width: 120px;
  height: 70px;
}
```

HTML:

```
<div class="boxes"></div>
<div class="boxes"></div>
```

Notice the block-level elements in the image above take up their own line of space and therefore don't overlap each other. In the browser to the right you can see block-level elements also consistently appear on the left side of the browser. This is the default *position* for block-level elements.

The default position can be changed by setting its `position` property. It takes four values:

1. static - **the default value (it does not need to be specified)**
2. relative
3. absolute
4. fixed

Position: Relative

This is First way to modify the default position of an element to its `position` property to `relative`.

```
.box-bottom {
  background-color: DeepSkyBlue;
  position: relative;
}
```

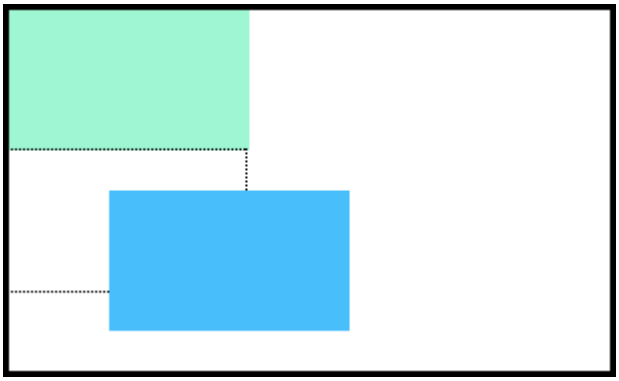
Although the code in the example above instructs the browser to expect a relative positioning of the div, it does not specify where the div should be positioned on the page.

```
.box-bottom {
  background-color: DeepSkyBlue;
  position: relative;
  top: 20px;
  left: 50px;
}
```

Above example uses two of the offset properties. The valid offset properties are:

1. top - **moves the element down.**
2. bottom - **moves the element up.**
3. left - **moves the element right.**
4. right - **moves the element left.**

In the example above, the **will be moved down 20 pixels and to the right 50 pixels from its default static position. The image below displays the new position of the box. The dotted line represents where the statically positioned (default) box was positioned.**



Units for offset properties can be specified in pixels, ems, or percentages. Note that offset properties will not work if the value of the element's `position` **property is the default** `static`.

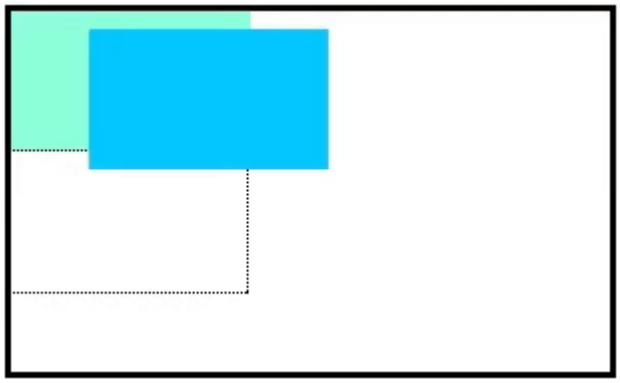
Position: Absolute

Another way of modifying the position of an element is by setting its position to `absolute`.

When an element's position is set to `absolute` **all other elements on the page will** *ignore* **the element and act like it is not present on the page. The element will be positioned relative to its closest positioned parent element.**

```
.box-bottom {
  background-color: DeepSkyBlue;
  position:  absolute;
  top: 20px;
  left: 50px;
}
```

In the example above, the `.box-bottom` `<div>` **will be moved down and right from the top left corner of the view. If offset properties weren't specified, the top box would be entirely covered by the bottom box. Take a look at the gif below:**



The bottom box in this image (colored blue) is displaced from the top left corner of its container. It is 20 pixels lower and 50 pixels to the right of the top box.

In the next exercise, we will compare the scrolling of `absolute` **elements with** `fixed` **elements.**

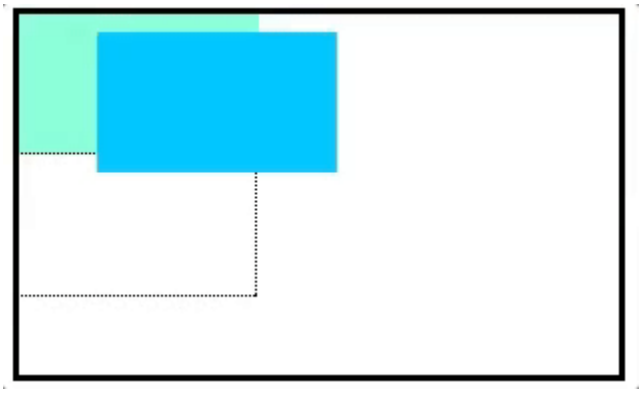
Position: Fixed

When an element's position is set to absolute**, as in the last exercise, the element will scroll with the rest of the document when a user scrolls.**

We can *fix* **an element to a specific position on the page (regardless of user scrolling) by setting its position to** fixed**.**

```
.box-bottom {
  background-color: DeepSkyBlue;
  position: fixed;
  top: 20px;
  left: 50px;
}
```

In the example above, the `.box-bottom` `<div>` **will remain fixed to its position no matter where the user scrolls on the page, like in the image below:**



https://s3.amazonaws.com/codecademy-content/courses/web-101/unit-7/alex-clark-experiment/fixed.gif

This technique is often used for navigation bars on a web page.

Z-Index

When boxes on a web page have a combination of different positions, the boxes (and therefore, their content) can overlap with each other, making the content difficult to read or consume.

```
.box-top {
  background-color: Aquamarine;
}

.box-bottom {
  background-color: DeepSkyBlue;
  position: absolute;
  top: 20px;
  left: 50px;
}
```

In the example above, the `.box-bottom` `<div>` ignores the `.box-top` `<div>` and overlaps it as a user scrolls.

The `z-index` property controls how far "back" or how far "forward" an element should appear on the web page when elements overlap. This can be thought of the *depth* of elements, with deeper elements appearing behind shallower elements.

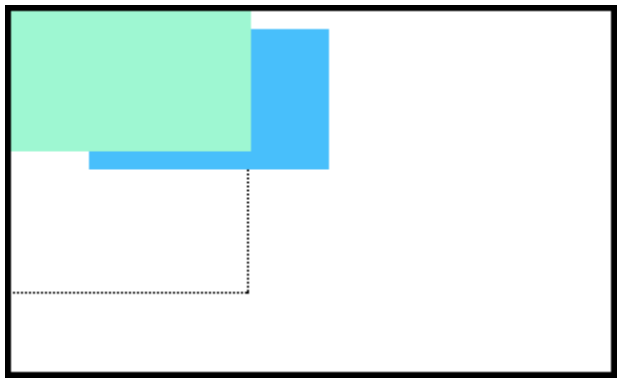
The `z-index` property accepts integer values. Depending on their values, the integers instruct the browser on the order in which elements should be displayed on the web page.

```
.box-top {
  background-color: Aquamarine;
  position: relative;
  z-index: 2;
}

.box-bottom {
  background-color: DeepSkyBlue;
  position: absolute;
  top: 20px;
  left: 50px;
```

```
  z-index: 1;
}
```

In the example above, we set the `.box-top` position to relative and the z-index to 2. We changed position to `relative`, because the z-index property does *not* work on static elements. The z-index of `2` moves the `.box-top` element forward, because it is greater than the `.box-bottom` z-index, `1`. See the example image below:



In the image above, you can see the top box is moved in front of the bottom box.

Inline Display

Every HTML element has a default `display` value that dictates if it can share horizontal space with other elements.

Some elements fill the entire browser from left to right regardless of the size of their content.

Other elements only take up as much horizontal space as their content requires and can be directly next to other elements. There are three values for the `display` property: `inline`, `block`, and `inline-block`.

```
does not start a new line and cannot be sized using the height and width
properties.
```

The default `display` for some tags, such as `<em>`, `<strong>`, and `<a>`, is called *inline*. Inline elements have a box that wraps tightly around their content, only taking up the amount of space necessary to display their content and not requiring a new line after each element. `inline` elements cannot be altered in size with the `height` or `width` CSS

properties.

This is most easily demonstrated with a simple example. First, some simple CSS that we'll be using:

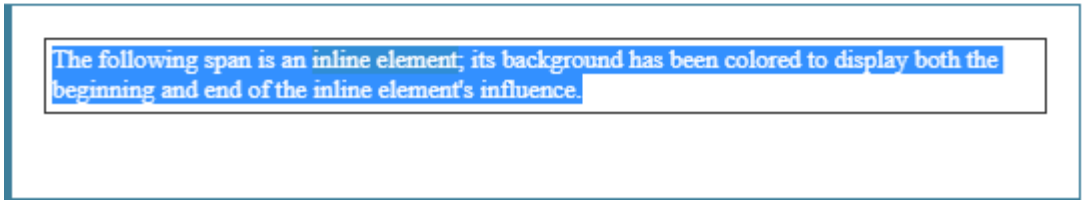
```
.highlight {
  background-color:#ee3;
}
```

Inline

Let's look at the following example which demonstrates an inline element:

```
<div>The following span is an <span class="highlight">inline element</span>
its background has been colored to display both the beginning and end of
the inline element's influence.</div>
```

In this example, the `<div>` block-level element contains some text. Within that text is a `<span>` element, which is an inline element. Because the `<span>` element is inline, the paragraph correctly renders as a single, unbroken text flow, like this:



To learn more about *inline* elements, click

https://developer.mozilla.org/en-US/docs/Web/HTML/Inline_elements

The CSS `display` property provides the ability to make any element an inline element. This includes elements that are not inline by default such as paragraphs, divs, and headings.

List of "inline" elements

The following elements are inline by default (although block and inline elements are no longer defined in HTML 5, use content categories instead):

`<a>`	`<i>`	`<script>`
`<abbr>`	`<iframe>`	`<select>`
`<acronym>`	`<img>`	`<slot>`
`<audio>` (if has visible controls)	`<input>`	`<small>`
`<b>`	`<ins>`	`<span>`
`<bdi>`	`<kbd>`	`<strong>`
`<bdo>`	`<label>`	`<sub>`
`<big>`	`<map>`	`<sup>`
` `	`<mark>`	`<svg>`
`<button>`	`<meter>`	`<template>`
`<canvas>`	`<noscript>`	`<textarea>`
`<cite>`	`<object>`	`<time>`
`<code>`	`<output>`	`<u>`
`<data>`	`<picture>`	`<tt>`
`<datalist>`	`<progress>`	`<var>`
`<del>`	`<q>`	`<video>`
`<dfn>`	`<ruby>`	`<wbr>`
`<em>`	`<s>`	
`<embed>`	`<samp>`	

Block Display

Some elements are not displayed in the same line as the content around them. These are called *block-level* elements. These elements fill the entire width of the page by default, but their `width` property can also be set. Unless otherwise specified, they are the height necessary to accommodate their content.

Elements that are block-level by default include all levels of heading elements (`<h1>` through `<h6>`), `<p>`, `<div>` and `<footer>`. For a complete list of block level elements, visit

starts a new line and can be sized using the height and width properties.

https://developer.mozilla.org/en-US/docs/Web/HTML/Block-level_elements

Elements

The following is a complete list of all HTML "block-level" elements (although "block-level" is not technically defined for elements that are new in HTML5).

`<address>`
Contact information.

`<article>`
Article content.

`<aside>`
Aside content.

`<blockquote>`
Long ("block") quotation.

`<details>`
Disclosure widget.

`<dialog>`
Dialog box.

`<dd>`
Describes a term in a description list.

`<div>`
Document division.

`<dl>`
Description list.

`<dt>`
Description list term.

`<fieldset>`
Field set label.

`<figcaption>`
Figure caption.

`<figure>`
Groups media content with a caption (see `<figcaption>`).

`<footer>`
Section or page footer.

`<form>`
Input form.

`<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>`, `<h6>`
Heading levels 1-6.

`<header>`
Section or page header.

`<hgroup>`
Groups header information.

`<hr>`
Horizontal rule (dividing line).

`<li>`
List item.

`<main>`
Contains the central content unique to this document.

`<nav>`
Contains navigation links.

`<ol>`
Ordered list.

`<p>`
Paragraph.

`<pre>`
Preformatted text.

`<section>`
Section of a web page.

`<table>`
Table.

`<ul>`
Unordered list.

Inline-Block Display

The third value for the display **property is** inline-block**. Inline-block display combines features of both inline and block elements. Inline-block elements can appear next to each other and we can specify their dimensions using the** width **and** height **properties. Images are the best example of default inline-block elements.**

For example, **s in the CSS below will be displayed on the same line and with the specified dimensions:**

```
<div  class="rectangle">
   <p>I'm a rectangle!</p>
</div>
<div  class="rectangle">
   <p>So am  I!</p>
</div>
```

```
<div class="rectangle">
  <p>Me three!</p>
</div>
.rectangle {
  display: inline-block;
  width: 200px;
  height: 300px;
}
```

In the example above, there are three rectangular divs that each contain a paragraph of text. The .rectangle **s will all appear inline (provided there is enough space from left to right) with a width of 200 pixels and height of 300 pixels, even though the text inside of them may not require 200 pixels by 300 pixels of space.**

```
does not start a new line and can be sized using the height and width properties.
```

Float

So far, you've learned how to specify the exact position of an element using offset properties. If you're simply interested in moving an element as far left or as far right as possible on the page, you can use the float **property.**

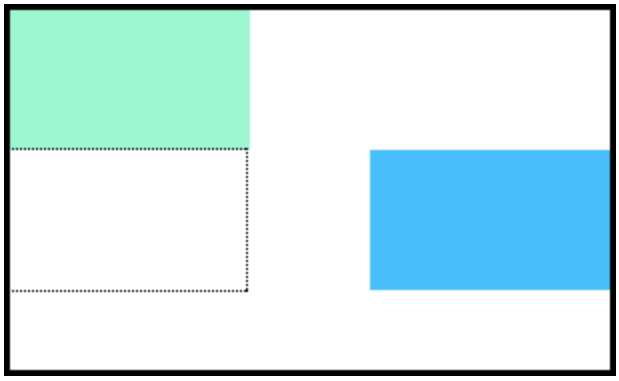
The float **property can be set to one of two values:**

1. left **- this value will move, or float, elements as far left as possible.**
2. right **- this value will move elements as far right as possible.**

```
.boxes {
  width: 120px;
  height: 70px;
}

.box-bottom {
  background-color: DeepSkyBlue;
  float:  right;
}
```

In the example above, we float the .box-bottom **element to the** right**. This works for static and relative positioned elements. See the result of the code below:**



Floated elements must have a width specified, as in the example above. Otherwise, the element will assume the full width of its containing element, and changing the float value will not yield any visible results.

Clear

The float **property can also be used to float multiple elements at once. However, when multiple floated elements have different heights, it can affect their layout on the page. Specifically, elements can "bump" into each other and not allow other elements to properly move to the left or right.**

The clear **property specifies how elements should behave when they bump into each other on the page. It can take on one of the following values:**

1. left **— the left side of the element will not touch any other element within the same containing element.**
2. right **— the right side of the element will not touch any other element within the same containing element.**
3. both **— neither side of the element will touch any other element within the same containing element.**
4. none **— the element can touch either side.**

```
div {
  width: 200px;
  float:  left;
}

div.special {
  clear:  left;
}
```

In the example above, all **s on the page are floated to the left side. The element with class** special **did not move all the way to the left because a taller** **blocked its positioning. By setting its** clear **property to** left**, the** special **will be moved all the way to the left side of the page.**

CSS COLOR

Introduction to Color

CSS supports a wide variety of colors. These include *named colors*, **like** blue, black, **and** LimeGreen, **along with colors described by a numeric value. Using a numeric system allows us to take advantage of the whole spectrum of colors that browsers support.**

Colors in CSS can be described in three different ways:

- *Named colors* — **English words that describe colors, also called** *keyword colors*
- *RGB* — **numeric values that describe a mix of red, green, and blue**
- *HSL* — **numeric values that describe a mix of hue, saturation, and lightness**

Foreground vs Background

it's important to make two distinctions about color.

1. **The foreground color**
2. **The background color**

Foreground color is the color that an element appears in. For example, when a heading is styled to appear green, the *foreground color* **of the heading has been styled.**

Conversely, when a heading is styled so that its background appears yellow, the *background color* **of the heading has been styled**

Two properties:

1. color **- this property styles an element's foreground color.**
2. background-color **- this property styles an element's background color.**

```
h1 {
  color: Red;
  background-color: Blue;
}
```

In the example above, the text of the heading will appear in red, and the background of the heading will appear blue.

Hexadecimal

One syntax that we can use to specify colors is called *hexadecimal*. Colors specified using this system are called *hex colors*. A hex color begins with a hash character (`#`) which is followed by three or six characters. The characters represent values for red, blue and green.

```
DarkSeaGreen: #8FBC8F
Sienna:       #A0522D
SaddleBrown:  #8B4513
Brown:        #A52A2A
Black:        #000000 or #000
White:        #FFFFFF or #FFF
Aqua:         #00FFFF or #0FF
```

In the example above, you may notice that there are both letters and numbers in the values. This is because the hexadecimal number system has 16 digits (0-15) instead of 10 (0-9) like you are used to. To represent 10-15, we use A-F. Here is a list of many different colors and their hex values.

Notice that `Black`, `White`, and `Aqua` are all represented with both three characters and six characters. This can be done with hex colors whose number pairs are the same characters.

RGB Colors

There is another syntax for representing RGB values that uses decimal numbers. It looks like this:

```
h1 {
  color: rgb(23, 45, 23);
}
```

Here, each of the three values represents a color component, and each can have a decimal number value from 0 to 255. The first number represents the amount of red, the second is green, and the third is blue. These colors are exactly the same as hex, but with a different syntax and a different number system.

Both are hex and decimal color representations are equivalent.

Hex and RGB

The hexadecimal and RGB color system can represent many more colors than the small set of CSS named colors.

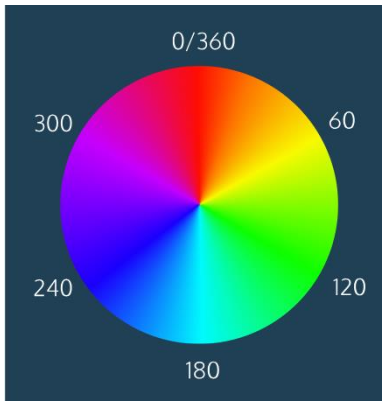
In both hex and decimal, we have three values, one for each color. Each can be one of 256 values. Specifically, `256 * 256 * 256 = 16,777,216`. That is the amount of colors we can now represent. Compare that to the 147 named CSS colors!

Hue, Saturation, and Lightness

The RGB color scheme is convenient because it's very close to how computers represent colors internally. There's another equally powerful system called the hue-saturation-lightness color scheme.Syntax similar but,

The first number represents the degree of the hue, and can be between 0 and 360. The second and third numbers are percentages representing saturation and lightness respectively. Here is an example:

```
color: hsl(120, 60%, 70%);
```



Saturation refers to the intensity or purity of the color. If you imagine a line segment drawn from the center of the color wheel to the perimeter, the saturation is a point on that line segment.

If you spin that line segment to different angles, you'll see how that saturation looks for different hues. The saturation increases towards 100% as the point gets closer to the edge (the color becomes more rich). The saturation decreases towards 0% as the point gets closer to the center (the color becomes more gray).

Lightness refers to how light or dark the color is. Halfway, or 50%, is normal lightness. Sliding the dimmer up towards 100% makes the color lighter, closer to white. Sliding the dimmer down towards 0% makes the color darker, closer to black.

Opacity and Alpha

We'll change the *opacity*, or the amount of transparency, of some colors so that some or all of the bottom elements are visible through a covering element.

To use opacity in the HSL color scheme, use `hsla` .

```
color: hsla(34, 100%, 50%, 0.1);
```

The fourth value (which we have not seen before) is the *alpha*. This last value is sometimes called the opacity.

Alpha is a decimal number from zero to one. If alpha is zero, the color will be completely transparent. If alpha is one, the color will be opaque. The value for half transparent would be `0.5`.

To use opacity in the rgb color scheme, Use `rgba`.

```
color: rgba(234, 45, 98, 0.33);
```

Alpha can only be used with HSL and RGB colors; we cannot add the alpha value to `color: green` `color: #FFFFF`.

There is, however, a named color keyword for zero opacity, `transparent`. It's equivalent to `rgba(0, 0, 0, 0)`. It's used like any other color keyword:

```
color: transparent;
```

CSS TYPOGRAPHY

Typography

The art of arranging text on a page. In particular, we'll look at how to style fonts with CSS to make them legible and appealing and how to add external fonts to your web pages.

Some of the most important information a user will see on a web page will be textual.

Font Family

The phrase "type of font" refers to the technical term typeface, or *font family*.

Use the `font-family` property to change the typeface of text on your web page,

```
h1 {
  font-family: Garamond;
}
```

In the example above, the font family for all main heading elements has been set to `Garamond`.

When setting typefaces on a web page, keep the following points in mind:

1. The font specified in a stylesheet must be installed on a user's computer in order for that font to display when a user visit the web page.
2. The default typeface for all most browsers is Times New Roman.
3. It's a good practice to limit the number of typefaces used on a web page to 2 or 3.
4. When the name of a typeface consists of more than one word, it must be enclosed in double quotes

```
h1 {
  font-family: "Courier New";
}
```

Font Weight

It's common to bold important headings or keywords. It can be done with the `font-weight` property.

```
p {
  font-weight: bold;
```

```
}
```

```
p {
  font-weight: normal;
}
```

By default, the `font-weight` of most text elements is set to `normal`.

Font Weight II

The `font-weight` property can also be assigned a number value to style text on a numeric scale ranging from 100 to 900.

There are a number of default font weights that we can use:

1. `400` is the default `font-weight` of most text.
2. `700` signifies a bold `font-weight`.
3. `300` signifies a light `font-weight`.

```
header {
  font-weight: 800;
}

footer {
  font-weight: 200;
}
```

It's important to note that not all fonts can be assigned a numeric `font-weight`.

Font Style

You can also italicize text with the `font-style`property.

```
h3 {
  font-style: italic;
}
```

The `italic` value causes text to appear in italics. The `font-style` property also has a `normal` value which is the default.

Word Spacing

You can also increase the spacing between words in a body of text, technically known as *word spacing*.

```
h1 {
  word-spacing: 0.3em;
}
```

The default amount of space between words is usually `0.25em`.

Note, again, that the preferred unit is ems.

Letter Spacing

The technical term for adjusting the spacing between letters is called "kerning". Kerning can be adjusted with the `letter-spacing` property in CSS.

```
h1 {
  letter-spacing: 0.3em;
}
```

Text Transformation

Text can also be styled to appear in either all uppercase or lowercase with the `text-transform` property.

```
h1 {
  text-transform: uppercase;
}
```

Text Alignment

Text always appears on the left side of the browser.

To move, or align, text, we can use the `text-align` property.

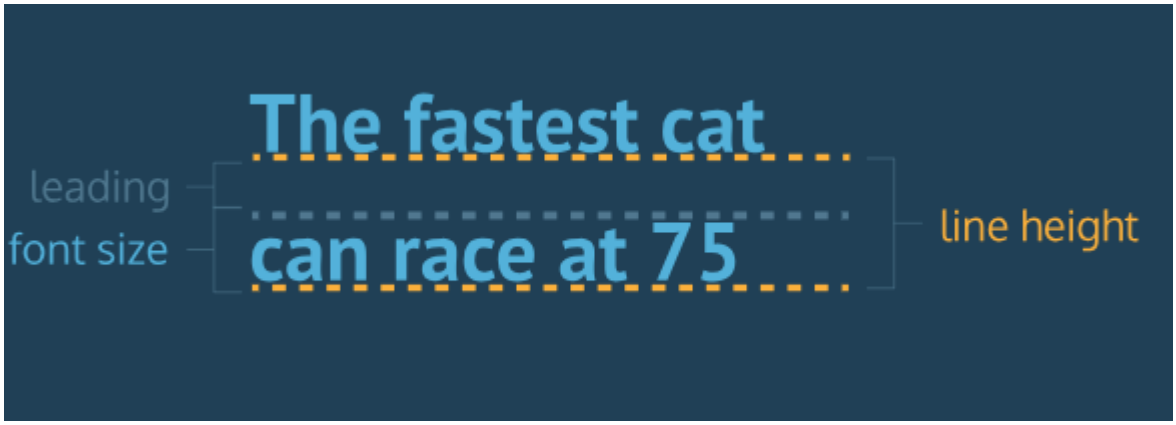
```
h1 {
  text-align: right;
}
```

The `text-align` property can be set to one of the following three values:

1. `left` - aligns text to the left hand side of the browser.
2. `center` - centers text.
3. `right` - aligns text to the right hand side of the browser.

Line Height Anatomy

Another property that we can set for text is `line-height`. This property modifies the *leading* of text.



Line Height

We often modify `line-height` to make text on a web page easier to read. When text is styled to appear larger, the vertical spacing between lines of text can decrease, creating text that is difficult to read, particularly in paragraphs.

Line heights can take one of several values:

1. A unitless number, such as `1.2`. This number is an absolute value that will compute the line height as a ratio of the font size.
2. A number specified by unit, such as `12px`. This number can be any valid CSS unit, such as pixels, percents, ems, or rems.

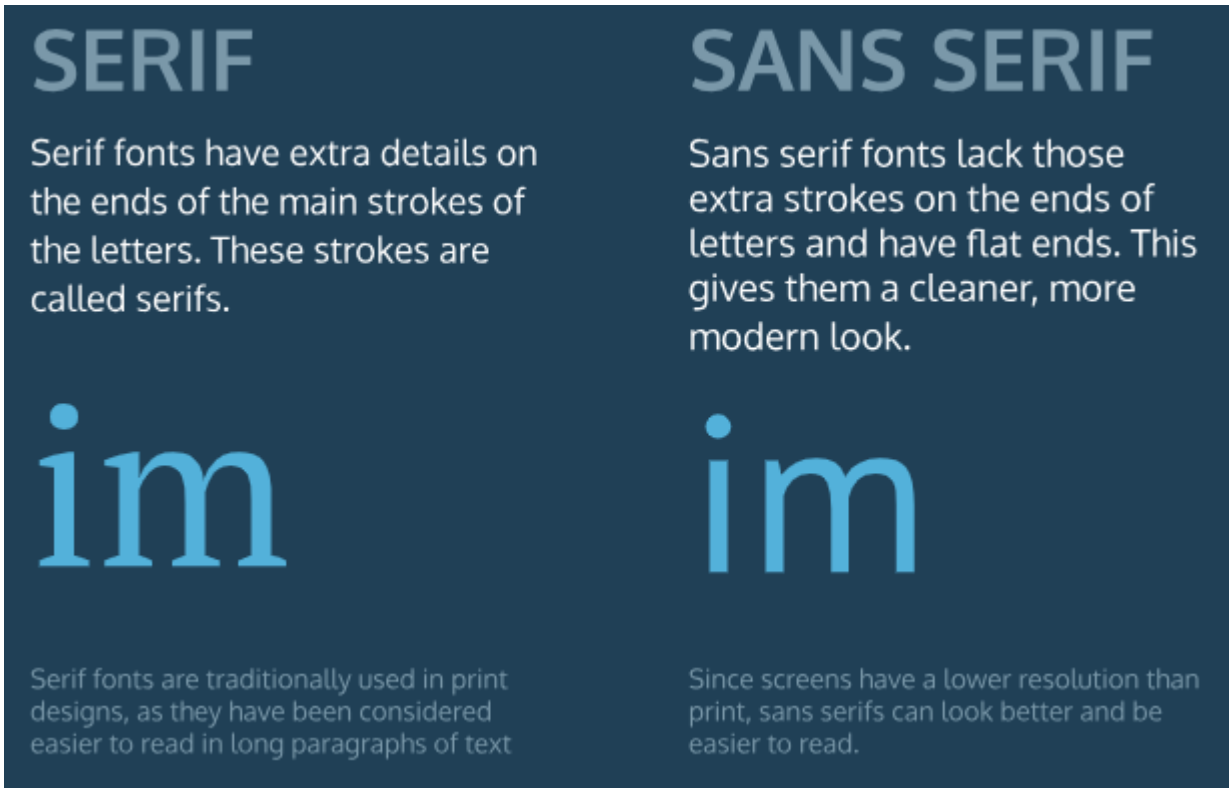
If we change the font size, a unitless `line-height` would automatically readjust, whereas the pixel value would remain static.

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

Serif and Sans Serif

You'll set some text to be *serif* and some text to be *sans-serif*.

1. Serif —Letter have taken extra space. Examples include fonts like Times New Roman or Georgia, among others.
2. Sans-Serif —Letters have straight, flat edges, like Arial or Helvetica.



Fallback Fonts

These pre-installed fonts serve as *fallback fonts* if the stylesheet specifies a font which is not installed on a user's computer.

Syntax is required:

```css
h1 {
  font-family: "Garamond", "Times", serif;
}
```

The CSS rule above says:

1. Use the Garamond font for all `<h1>` elements on the web page.
2. If Garamond is not available, use the Times font.
3. If Garamond and Times are not available, use any serif font pre-installed on the user's computer.

The fonts specified after Garamond are the fallback fonts (`Times`, `serif`). Fallback fonts help ensure a consistent experience for the diverse audience of users that visit a site.

Linking Fonts I

With the number of fonts available with modern typography, it is unrealistic to expect users to have all fonts installed on their computers.

New fonts are often centralized in directories made available for public use. We refer to these fonts as *non-user fonts*. Google Fonts

(https://fonts.google.com/) is one such directory of thousands of open-source fonts, available for free use. Google Fonts gives us a way to retrieve the link for a single font, multiple fonts, or multiple fonts with the `font-weight` and `font-style` properties.

Linking Fonts II

When we have the link to the font of our choice, we can add the font to the `<head>` section of the HTML document, using the `<link>` tag and the `href`.

Let's take a look at a few examples:

1. A single linked font, `Droid`

```
<head>
  <link href="https://fonts.googleapis.com/css?family=Droid+Serif"
type="text/css" rel="stylesheet">
</head>
```

2. Multiple linked fonts, `Droid+Serif|Playfair+Display`

```
<head>
  <link
href="https://fonts.googleapis.com/css?family=Droid+Serif|Playfair+Displa
y" type="text/css" rel="stylesheet">
</head>
```

3. Multiple linked fonts, along with weights and styles. Here `Droid Serif` has font weights of `400`, `700`, and `700i`, while `Playfair Display` has font weights of `400`, `700`, and `900i`:

```
<head>
  <link
href="https://fonts.googleapis.com/css?family=Droid+Serif:400,700,700i|Pl
ayfair+Display:400,700,900i" rel="stylesheet">
</head>
```

Font-Face I

There are other ways to link non-user fonts that don't require the use of the `<link>` tag in the HTML document. CSS offers a way to import fonts directly into stylesheets with the `@font-face` property.

https://fonts.googleapis.com

To load fonts with the `@font-face` property:

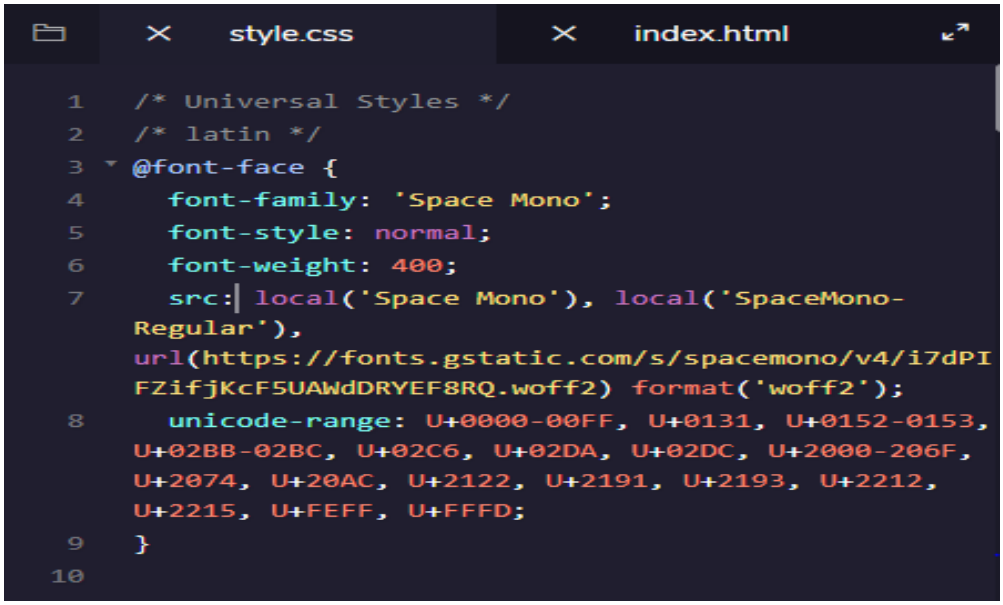
1. Instead of using the font's link in the HTML document, enter the link into the URL bar in the browser.

https://fonts.googleapis.com/css?family=Space+Mono:400,700

2. The browser will load the CSS rules. You will need to focus on the rules that are directly labeled as `/* latin */`. Some of the latin rules are on separate lines. You will need each of these.

```
/* latin */
@font-face {
  font-family: 'Space Mono';
  font-style: normal;
  font-weight: 400;
  src: local('Space Mono'), local('SpaceMono-Regular'), url(https://fonts.gstatic.com/s/spacemono/v4/i7dPIFZifjKcF5UAWdDRYEF8RQ.woff2) format('woff2');
  unicode-range: U+0000-00FF, U+0131, U+0152-0153, U+02BB-02BC, U+02C6, U+02DA, U+02DC, U+2000-206F, U+2074, U+20AC, U+2122, U+2191, U+2193, U+2212, U+2215, U+FEFF, U+FFFD;
}
/* vietnamese */
```

3. Copy each of the CSS rules labeled latin, and paste the rules from the browser to the top of **style.css**.



It is important to stress the need to copy the `@font-face` rules to the top of the stylesheet for the font to load correctly in the project.

Font-Face III

While Google Fonts and other resources can broaden font selection, you may wish to use an entirely different font or abstain from using a font from an external service.

We can modify our `@font-face` rule to use local font files as well. We can supply the user with the desired font family and host it along with our site instead of depending on a different site.

```
@font-face {
  font-family: "Roboto";
  src: url(fonts/Roboto.woff2) format('woff2'),
       url(fonts/Roboto.woff) format('woff'),
       url(fonts/Roboto.tff) format('truetype');
}
```

Here, you'll notice:

1. The main difference is the use of a relative filepath instead of a web URL.
2. We add a format for each file to specify which font to use. Different browsers support different font types, so providing multiple font file options will support more browsers.

As of now `.woff2` appears to be the way of the future, due to greatly reduced file sizes and improved performance, but many browsers still don't support it. There are lots of great sources to find fonts to use locally, such as https://www.fontsquirrel.com/

.CSS top of file :

```
@font-face {
  font-family: "Glegoo";
  src: url(../fonts/Glegoo-Regular.ttf) format('truetype');
}
```

```
.banner p {
  border-top: 1px solid #fff;
  border-bottom: 1px solid #fff;
  padding: 10px;
  color: #ffffff;
  font-weight: bold;
  line-height: 1.4;
  font-family: "Glegoo";
  font-size: 20px;
}
```

CSS : Material

https://developer.mozilla.org/en-US/docs/Web/CSS

CODE CHALLENGE: CSS DESIGN

Round Image

Embrace web trends by changing a square image to a circular one.

```
img {
    height: 150px;
    margin: 0 auto;
   border-radius: 50%;
  }
```

Combinator

Use a CSS combinator to make a ruleset more specific.

```
</article>
<p class="recipe">
    Here's the recipe:
 </p>
</article>
```

```
article .recipe {
    font-weight: bold;
  }
```

Position

Positioning an element is an important part of shaping the layout of a web page.

```
<body>
    <div class="circle" id="blue">
    </div>
    <div class="circle" id="red">
    </div>
  </body>
.circle {border-radius: 50%;width: 200px;height: 200px; opacity: 0.5;}
#red {  background-color: rgb(0, 0, 255);}
#blue { background-color: rgb(255, 0, 0); position:absolute;}
```

Learn Responsive Design

SIZING ELEMENTS

Relative Measurements

Modern technology allows users to browse the Internet via multiple devices, such as desktop monitors, mobile phones, tablets, and more. Devices of different screen sizes, however, pose a problem for web developers: how can we ensure that a website is readable and visually appealing across all devices, regardless of screen size?

Responsive design refers to the ability of a website to resize and reorganize its content based on:

1. The size of other content on the website.
2. The size of the screen the website is being viewed on.

With CSS, you can avoid hard coded measurements and use *relative measurements* instead.

Em

Today, the em represents the size of the base font being used. For example, if the base font of a browser is 16 pixels (which is normally the default size of text in a browser), then 1 em is equal to 16 pixels. 2 ems would equal 32 pixels, and so on.

```
.heading {
  font-size: 2em;
}
```

```
.splash-section {
  font-size: 18px;
}

.splash-section h1 {
  font-size: 1.5em;
}
```

The example above shows how to use ems without relying on the default font size of the browser. Instead, a base font size (`18px`) is defined for all text within the `splash-section`element. The second CSS rule will set the font size of all `h1` elements inside of `splash-section`relative to the base font of `splash-section` (18 pixels). The resulting font size of `h1` elements will be 27 pixels.

Rem

The second relative unit of measurement in CSS is the *rem*, coded as `rem`.

Rem stands for *root em*. It acts similar to em, but instead of checking parent elements to size font, it checks the *root element*. The root element is the `<html>` tag.

Most browsers set the font size of `<html>` to 16 pixels, so by default rem measurements will be compared to that value. To set a different font size for the root element, you can add a CSS rule.

```
html {
  font-size: 20px;
}

h1 {
  font-size: 2rem;
}
```

In the example above, the font size of the root element, `<html>`, is set to 20 pixels. All subsequent rem measurements will now be compared to that value and the size of `h1`elements in the example will be 40 pixels.

Percentages: Height & Width

To size non-text HTML elements relative to their parent elements on the page you can use *percentages*.

```
.main {
  height: 300px;
  width: 500px;
}

.main .subsection {
  height: 50%;
  width: 50%;
}
```

In the example above, `.main` and `.subsection`each represent divs. The `.subsection` div is nested within the `.main` div. Note that the dimensions of the parent div (`.main`) have been set to a height of 300 pixels and a width of 500 pixels.

When percentages are used, elements are sized relative to the dimensions of their parent element (also known as a container). Therefore, the dimensions of the `.subsection` div will be 150 pixels tall and 250 pixels wide. Be careful, a child element's dimensions may be set erroneously(mistaken) if the dimensions of its parent element aren't set first.

Note: Because the box model includes padding, borders, and margins, setting an element's `width`to `100%` may cause content to overflow its parent container. While tempting, `100%` should only be used when content will not have padding, border, or margin.

Percentages: Padding & Margin

Percentages can also be used to set the padding and margin of elements.

When height and width are set using percentages, you learned that the dimensions of child elements are calculated based on the dimensions of the parent element.

When percentages are used to set padding and margin, however, they are calculated based only on the *width* of the parent element.

For example, when a property like `margin-left` is set using a percentage (say `50%`), the element will be moved halfway to the right in the parent container (as opposed to the child element receiving a margin half of its parent's margin).

Vertical padding and margin are also calculated based on the width of the parent. Why? Consider the following scenario:

1. A container div is defined, but its height is not set (meaning it's flat).
2. The container then has a child element added within. The child element *does* have a set height. This causes the height of its parent container to stretch to that height.
3. The child element requires a change, and its height is modified. This causes the parent container's height to also stretch to the new height. This cycle occurs endlessly whenever the child element's height is changed!

In the scenario above, an unset height (the parent's) results in a constantly changing height due to changes to the child element. This is why vertical padding and margin are based on the width of the parent, and not the height.

Note: When using relative sizing, ems and rems should be used to size text and dimensions on the page related to text size (i.e. padding around text). This creates a consistent layout based on text size. Otherwise, percentages should be used.

Width: Minimum & Maximum

Although relative measurements provide consistent layouts across devices of different screen sizes, elements on a website can lose their integrity when they become too small or large. You can limit how wide an element becomes with the following properties:

1. `min-width` — ensures a minimum width for an element.
2. `max-width` — ensures a maximum width for an element.

```
p {
  min-width: 300px;
  max-width: 600px;
}
```

In the example above, when the browser is resized, the width of paragraph elements will not fall below 300 pixels, nor will their width exceed 600 pixels.

When a browser window is narrowed or widened, text can become either very compressed or very spread out, making it difficult to read. These two properties ensure that content is legible by limiting the minimum and maximum widths.

Note: The unit of pixels is used to ensure hard limits on the dimensions of the element(s).

Height: Minimum & Maximum

You can also limit the minimum and maximum *height* of an element.

1. `min-height` — ensures a minimum height for an element's box.
2. `max-height` — ensures a maximum height for an element's box.

```
p {
  min-height: 150px;
  max-height: 300px;
}
```

In the example above, the height of all paragraphs will not shrink below 150 pixels and the height will not exceed 300 pixels.

What will happen to the contents of an element if the `max-height` property is set too low for that element? It's possible that content will overflow outside of the element, resulting in content that is not legible.

Scaling Images and Videos

Many websites contain a variety of different media, like images and videos. When a website contains such media, it's important to make sure that it is scaled proportionally so that users can correctly view it.

```
.container {
  width: 50%;
  height: 200px;
  overflow: hidden;
}

.container img {
  max-width: 100%;
  height: auto;
  display: block;
}
```

In the example above, `.container` represents a container div. It is set to a width of `50%` (half of the browser's width, in this example) and a height of 200 pixels.
Setting `overflow` to `hidden`ensures that any content with dimensions larger than the container will be hidden from view.

The second CSS rule ensures that images scale with the width of the container.
The `height`property is set to `auto`, meaning an image's height will *automatically* scale proportionally with the width. Finally, the last line will display images as block level elements (rather than inline-block, their default state). This will prevent images from attempting to align with other content on the page (like text), which can add unintended margin to the images.

It's worth memorizing the entire example above. It represents a *very common* design pattern used to scale images and videos proportionally.

Note: The example above scales the width of an image (or video) to the width of a container. If the image is larger than the container, the vertical portion of the image will overflow and will not display. To swap this behavior, you can set `max-height` to `100%` and `width` to `auto` (essentially swapping the values). This will scale the *height* of the image with the height of the container instead. If the image is larger than the container, the horizontal portion of the image will overflow and not display.

Scaling Background Images

Background images of HTML elements can also be scaled responsively using CSS properties.

```
body {
  background-image: url('#');
  background-repeat: no-repeat;
  background-position: center;
  background-size: cover;
}
```

In the example above, the first CSS declaration sets the background image (`#` is a placeholder for an image URL in this example). The second declaration instructs the CSS compiler to not repeat the image (by default, images will repeat). The third declaration centers the image within the element.

The final declaration, however, is the focus of the example above. It's what scales the background image. The image will *cover* the entire background of the element, all while keeping the image in proportion. If the dimensions of the image exceed the dimensions of the container then only a portion of the image will display.

MEDIA QUERIES

Responsive Web Design

When someone visits a website, it's possible they are viewing it on a phone, tablet, computer, or even a TV monitor. Because screen sizes can vary greatly across different devices, it's important for websites to resize and reorganize their content to best fit screens of all sizes.

When a website doesn't respond to different screen sizes, the website may look odd or become indecipherable on certain devices. This usually occurs on smaller screens, like phones. When a website responds to the size of the screen it's viewed on, it’s called a *responsive* website.

Media Queries

CSS uses *media queries* to adapt a website's content to different screen sizes. With media queries, CSS can detect the size of the current screen and apply different CSS styles depending on the width of the screen.

```
Example:1

@media only screen and (max-width: 480px) {
  body {
    font-size: 12px;
  }
}

+-----------------------------------------------------------------+

Example:2
@media only screen and (max-width: 480px) {
  .page-title {
    width: 270px;
  }
}

/*When the screen is less than 480px wide, give the .page-title
class a width of 270px.
```

```
This will make the .page-title element appear more clearly on small screens. */
```

The example above demonstrates how a media query is applied. The media query defines a rule for screens smaller than 480 pixels (approximately the width of many smartphones in landscape orientation).

Let's break this example down into its parts:

1. `@media` — This keyword begins a media query rule and instructs the CSS compiler on how to parse the rest of the rule.
2. `only screen` — Indicates what types of devices should use this rule. In early attempts to target different devices, CSS incorporated different media types (screen, print, handheld). The rationale was that by knowing the media type, the proper CSS rules could be applied. However, "handheld" and "screen" devices began to occupy a much wider range of sizes and having only one CSS rule per media device was not sufficient. `screen` is the media type always used for displaying content, no matter the type of device. The `only` keyword is added to indicate that this rule only applies to one media type (`screen`).
3. `and (max-width : 480px)` — This part of the rule is called a *media feature*, and instructs the CSS compiler to apply the CSS styles to devices with a width of 480 pixels or smaller. Media features are the conditions that must be met in order to render the CSS within a media query.
4. CSS rules are nested inside of the media query's curly braces. The rules will be applied when the media query is met. In the example above, the text in the `body` element is set to a `font-size` of `12px` when the user's screen is less than 480px.

Range

Specific screen sizes can be targeted by setting multiple width and height media features. `min-width` and `min-height` are used to set the minimum width and minimum height, respectively. Conversely, `max-width` and `max-height` set the maximum width and maximum height, respectively.

By using multiple widths and heights, a range can be set for a media query.

```
@media only screen and (min-width: 320px) and (max-width: 480px) {
    /* ruleset for 320px - 480px */
  .gallery-item .thumbnail{
    width:95%;
  }
}
```

The example above would apply its CSS rules only when the screen size is between 320 pixels and 480 pixels. Notice the use of a second `and`keyword after the `min-width` media feature. This allows us to chain two requirements together.

The example above can be written using two separate rules as well:

```
@media only screen and (min-width: 320px) {
    /* ruleset for 320px - 479px */
}


@media only screen and (min-width: 480px) {
    /* ruleset for > 480px */
}
```

The first media query in the example above will apply CSS rules when the size of the screen meets or exceeds 320 pixels. The second media query will apply CSS rules when the size of the screen meets or exceeds 480 pixels, meaning that it will override the CSS rules present in the first media query.

Both examples above are valid.

Dots Per Inch (DPI)

Another media feature we can target is screen resolution. Many times we will want to supply higher quality media (images, video, etc.) only to users with screens that can support high resolution media.

Targeting screen resolution also helps users avoid downloading high resolution (large file size) images that their screen may not be able to properly display.

To target by resolution, we can use the `min-resolution` and `max-resolution` media features. These media features accept a resolution value in either dots per inch (dpi) or dots per centimeter (dpc). Learn more about resolution measurements here.

```
@media only screen and (min-resolution: 300dpi) {
    /* CSS for high resolution screens */
}
```

The media query in the example above targets high resolution screens by making sure the screen resolution is at least 300 dots per inch. If the screen resolution query is met, then we can use CSS to display high resolution images and other media.

```
@media only screen and (min-resolution: 150dpi) {
    .logo{
      background-image: url("../img/spaceship@2x.png");
    }
```

```
    /* CSS for high resolution screens */
  }
```

And Operator

We chained multiple media features of the same type in one media query by using the `and` operator. It allowed us to create a range by using `min-width` and `max-width` in the same media query.

The `and` operator can be used to require multiple media features. Therefore, we can use the `and`operator to require both a `max-width` of `480px`*and* to have a `min-resolution` of `300dpi`.

For example:

```
@media only screen and (max-width: 480px) and (min-resolution: 300dpi) {
    /* CSS ruleset */
}
```

By placing the `and` operator between the two media features, the browser will require both media features to be true before it renders the CSS within the media query.
The `and` operator can be used to chain as many media features as necessary.

```
@media only screen and (max-width: 480px) and (min-resolution: 150dpi) {
    /* CSS ruleset */
  .main .page-description {
 font-size:20px;
  }
}
```

Comma Separated List

If only one of multiple media features in a media query must be met, media features can be separated in a comma separated list.

For example, if we needed to apply a style when only one of the below is true:

- The screen is more than 480 pixels wide
- The screen is in landscape mode

We could write:

```
@media only screen and (min-width: 480px), (orientation: landscape) {
    /* CSS ruleset */
}
```

In the example above, we used a comma (`,`) to separate multiple rules. The example above requires only one of the media features to be true for its CSS to apply.

Note that the second media feature is `orientation`. The `orientation` media feature detects if the page has more width than height. If a page is wider, it's considered `landscape`, and if a page is taller, it's considered `portrait`.

```
@media only screen and (min-width: 320px) and (max-width: 480px),(orientation
:portrait) {
    .gallery-item .thumbnail {
      width: 95%;
    }
  }
```

Breakpoints

The points at which media queries are set are called *breakpoints*. Breakpoints are the screen sizes at which your web page does not appear properly. For example, if we want to target tablets that are in landscape orientation, we can create the following breakpoint:

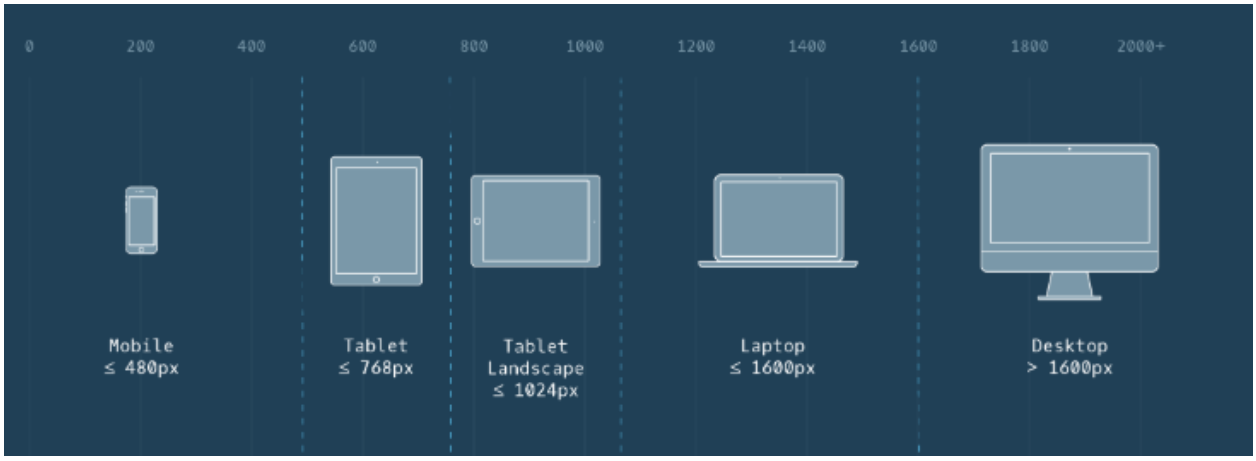
```
-@media only screen and (min-width: 768px) and (max-width: 1024px) and
(orientation: landscape) {
    /* CSS ruleset */
}
```

The example above creates a screen size range the size of a tablet in landscape mode and also identifies the orientation.

However, setting breakpoints for every device imaginable would be incredibly difficult because there are many devices of differing shapes and sizes. In addition, new devices are released with new screen sizes every year.

Rather than set breakpoints based on specific devices, the best practice is to resize your browser to view where the website naturally breaks based on its content. The dimensions at which the layout breaks or looks odd become your media query breakpoints. Within those breakpoints, we can adjust the CSS to make the page resize and reorganize.

By observing the dimensions at which a website naturally breaks, you can set media query breakpoints that create the best possible user experience on a project by project basis, rather than forcing every project to fit a certain screen size. Different projects have different needs, and creating a responsive design should be no different.




```
@media only screen and (min-width: 768px) and (max-width: 1024px) and (orientation: landscape) {
    .page-title,
    .page-description {
      float: left;
      width: 20em;
    }

    .page-description {
        text-align: left;
    }
  }
```

This CSS will make the page title and description float to the left of the gallery images. Resize the browser to observe these changes at various screen widths.

1. **What are the maximum dimensions of a div with the following properties AND minimum dimensions?**

- **600 pixels tall and 900 pixels wide is max**
- **150 pixels tall and 300 pixels wide. Is min**

2. **When percentages are used to size padding and width, dimensions are set relative to which of the following?**

- **The width of the parent element**

3. **Which of the following property/value pairs will ensure that the background image of the div in the code below will scale proportionally?**

```
div {
    background-image: url('#');
    background-repeat: no-repeat;
    background-position: center;
    /* ? */
  }
```

- `background-size: cover;`

4. **What is the difference between an em and a rem?**

- **An em sizes relative to a parent element's base font, whereas rems size relative to the root element's font size.**

5. **In the code below, how many pixels should the font size of the root element be set to in order to set the size of main headings to 36 pixels?**

```
html {
    font-size: ?
  }

  h1 {
    font-size: 2rem;
  }
```

- **18 pixels**

6. **The following code will size all h1 elements in "splash-section" to how many pixels?**

```
html {
    font-size: 12px;
  }

  .splash-section {
    font-size: 16px;
  }

  .splash-section h1 {
    font-size: 1.5em;
  }
```

- **24 PIXELS**

7. **The code below will display images in which of the following ways?**

```
.container {
    width: 50%;
    height: 200px;
    overflow: hidden;
  }

  .container img {
    max-width: 100%;
    height: auto;
    display: block;
  }
```

- **Images will shrink to the full width of their container, scale proportionally, and display partially if the image dimensions exceed container dimensions.**

8. **The "height" property in the code below should be set to which of the following values to ensure that the image remains proportional to its original size?**

```
img {
    width: 500px;
    height: ?
  }
```

- Auto

9. **Which of the following requirements must be met so that the media query below applies its corresponding CSS rules?**

```
@media only screen and (min-width: 320px), (orientation: landscape) {
    /* CSS rules */
  }
```

- **A browser with a width above 320px (or) an orientation of landscape.**

FLEXBOX

What is Flexbox?

flexbox or Flexible Box Layout, a new tool developed for CSS3 that greatly simplifies how to position elements. While flexbox is not meant to lay out entire pages, it is useful for positioning elements, whether individually or in groups.

There are two important components to a flexbox layout: *flex containers* and *flex items*. A flex container is an element on a page that contains flex items. All direct child elements of a flex container are flex items.

To designate an element as a flex container, set the element's `display` property to `flex` or `inline-flex`. Once an item is a flex container, there are several properties we can use to specify how its children behave.

1. `justify-content`
2. `align-items`
3. `flex-grow`
4. `flex-shrink`
5. `flex-basis`
6. `flex`
7. `flex-wrap`
8. `align-content`
9. `flex-direction`
10. `flex-flow`

Main Example:-----

```
<body>
  <h1>Display: Flex</h1>
  <div class="container" id="flex">
    <div class="box">
      <h2>1</h2>
    </div>
    <div class="box">
      <h2>2</h2>
    </div>
    <div class="box">
      <h2>3</h2>
    </div>
  </div>
  <h1>Display: Block</h1>
  <div class="container" id="block">
    <div class="box">
      <h2>1</h2>
    </div>
```

```
    <div class="box">
      <h2>2</h2>
    </div>
    <div class="box">
      <h2>3</h2>
    </div>
  </div>
</body>
```

```
body {
    margin: 0;
    border: 0;
    font-family: 'Roboto Mono', monospace;
  }

  h1 {
    font-size: 18px;
    text-align: center;
  }

  h2 {
    font-size: 16px;
    text-align: center;
  }

  .container {
    background-color: whitesmoke;
  }

  .box {
    background-color: Dodgerblue;
    height: 100px;
    width: 100px;
    border: 1px solid lightgrey;
  }

  #flex {

  }
```

display: flex

Any element can be a flex container. Flex containers are helpful tools for creating websites that respond to changes in screen sizes. Child elements of flex containers will change size and location in response to the size and position of their parent container.

For an element to become a flex container, its `display` property must be set to `flex`.

```
div.container {
  display: flex;
}
```

In the example above, all divs with the class `container` are flex containers. If they have children, the children are flex items. A div with the declaration `display: flex;` will remain block level — no other elements will appear on the same line as it.

However, it will change the behavior of its child elements. Child elements will not begin on new lines.

Main-Example :----

```
    #flex {
  Display : flex;
  }
```

inline-flex

If we didn't want div elements to be block-level elements, we would use `display: inline`. Flexbox, however, provides the `inline-flex`value for the `display` attribute, which allows us to create flex containers that are also inline elements.

```
<div class="container">
  <p>I’m inside of a flex container!</p>
  <p>A flex container’s children are flex items!</p>
</div>
<div class="container">
  <p>I’m also a flex item!</p>
  <p>Me too!</p>
</div>
.container {
  width: 200px;
  height: 200px;
  display: inline-flex;
}
```

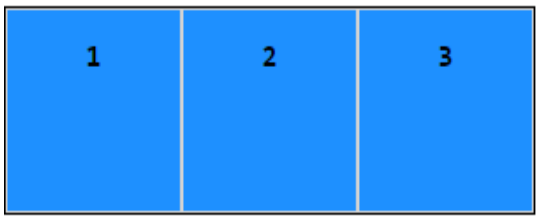
In the example above, there are two container divs. Without a width, each div would stretch the entire width of the page. The paragraphs within each div would also display on top of each other because paragraphs are block-level elements.

If we give flex property to the parent all it's children will become inline by-default but not the parent. But if we give parent inline-flex property then the parent will also become inline.

Main-Example :----

```
  .container {
    background-color: whitesmoke;
    border:solid 1px black;
  }
```

```
  #flex {
  display:inline-flex;
  }
```



When we change the value of the `display`property to `inline-flex`, the divs will display inline with each other if the page is wide enough.

```
<div class="container">
  <div class="child">
    <h1>1</h1>
  </div>
  <div class="child">
    <h1>2</h1>
  </div>
</div>
.container {
  width: 200px;
}

.child {
  display: inline-flex;
  width: 150px;
  height: auto;
```

```
}
```

In the example above, the `.child` divs will take up more width (300 pixels) than the `container`div allows (200 pixels). The `.child` divs will shrink to accommodate the container's size. In later exercises, we will explore several ways to handle this.

justify-content

when we changed the `display` value of parent containers to `flex` or `inline-flex`, all of the child elements (flex items) moved toward the upper left corner of the parent container. This is the default behaviour of flex containers and their children. We can specify how flex items spread out from left to right, along the *main axis*.

To position the items from left to right, we use a property called `justify-content`.

```
.container {
  display: flex;
  justify-content: flex-end;
}
```

we set the value of `justify-content` to `flex-end`. This will cause all of the flex items to shift to the right side of the flex container.

There are five values for the `justify-content`property:

1. `flex-start` — all items will be positioned in order starting, from the left of the parent container, with no extra space between or before them.
2. `flex-end` — all items will be positioned in order, with the last item starting on the right side of the parent container, with no extra space between or after them.
3. `center` — all items will be positioned in order, in the center of the parent container with no extra space before, between, or after them.
4. `space-around` — items will be positioned with equal space before and after each item, resulting in double the space between elements.
5. `space-between` — items will be positioned with equal space between them, but no extra space before the first or after the last elements.

In the definitions above, "no extra space" means that margins and borders will be respected, but no more space (than is specified in the style rule for the particular element) will be added between elements. The size of each individual flex item is not changed by this property.

```
#flexstart {
justify-content: flex-start;
}

#flexend {
justify-content: flex-end;
}

#center {
justify-content: center;
}

#spacearound {
justify-content: space-around;
}

#spacebetween {
justify-content: space-between;
}
```

align-items

It is also possible to align flex items vertically within the container. The `align-items` property makes it possible to space flex items vertically.

```
.container {
  align-items: baseline;
}
```

In the example above, the `align-items` property is set to `baseline`. This means that the baseline of the content of each item will be aligned.

There are five values we can use for the `align-items` property:

1. `flex-start` — all elements will be positioned at the top of the parent container.
2. `flex-end` — all elements will be positioned at the bottom of the parent container.
3. `center` — the center of all elements will be positioned halfway between the top and bottom of the parent container.
4. `baseline` — the bottom of the content of all items will be aligned with each other.

5. `stretch` — if possible, the items will stretch from top to bottom of the container (this is the default value; elements with a specified height will not stretch; elements with a minimum height or no height specified will stretch).

These five values tell the elements how to behave along the *cross axis* of the parent container. In these examples, the cross axis stretches from top to bottom of the container.

You might be unfamiliar with the `min-height` and `max-height` properties, but you have used `height`and `width` before. `min-height`, `max-height`, `min-width`, and `max-width` are properties that ensure an element is at least a certain size or at most a certain size.

```
#flexstart {
align-items: flex-start;
}

#flexend {
align-items: flex-end;
}

#center {
align-items: center;

}

#baseline {
align-items: baseline;
}
```

flex-grow

In flex – Inline , we learned that all flex items shrink proportionally when the flex container is too small. However, if the parent container is larger than necessary then the flex items will not stretch by default. The `flex-grow` property allows us to specify if items should grow to fill a container and also which items should grow proportionally more or less than others.

```
<div class="container">
  <div class="side">
    <h1>I’m on the side of the flex container!</h1>
  </div>
  <div class="center">
    <h1>I'm in the center of the flex container!</h1>
  </div>
  <div class=”side”>
    <h1>I'm on the other side of the flex container!</h1>
```

```
  </div>
</div>
.container {
  display: flex;
}

.side {
  width: 100px;
  flex-grow: 1;
}

.center {
  width: 100px;
  flex-grow: 2;
}
```

In the example above, the `.container` div has a `display` value of `flex`, so its three child divs will be positioned next to each other. If there is additional space in the `.container` div (in this case, if it is wider than 300 pixels), the flex items will grow to fill it. The `.center` div will stretch twice as much as the `.side` divs. For example, if there were 60 additional pixels of space, the `center` div would absorb 30 pixels and the `side`divs would absorb 15 pixels each.

If a `max-width` is set for an element, it will not grow larger than that even if there is more space for it to absorb.

This property — `flex-grow` — is the first we have learned that is declared on flex items.

flex-shrink

Just as the `flex-grow` property proportionally stretches flex items, the `flex-shrink` property can be used to specify which elements will shrink and in what proportions.

You may have noticed in earlier exercises that flex items shrank when the flex container was too small, even though we had not declared the property. This is because the default value of `flex-shrink` is `1`. However, flex items do not grow unless the `flex-grow` property is declared because the default value of `flex-grow` is `0`.

```
<div class="container">
  <div class="side">
    <h1>I'm on the side of the flex container!</h1>
  </div>
  <div class="center">
    <h1>I'm in the center of the flex container!</h1>
  </div>
  <div class="side">
    <h1>I'm on the other side of the flex container!</h1>
  </div>
</div>
```

```
.container {
  display: flex;
}

.side {
  width: 100px;
  flex-shrink: 1;
}

.center {
  width: 100px;
  flex-shrink: 2;
}
```

In the example above, the `.center` div will shrink twice as much as the `.side` divs if the `.container` div is too small to fit the elements within it. If the content is 60 pixels too large for the flex container that surrounds it, the `.center`div will shrink by 30 pixels and the outer divs will shrink by 15 pixels each. Margins are unaffected by `flex-grow` and `flex-shrink`.

Keep in mind, minimum and maximum widths will take precedence over `flex-grow` and `flex-shrink`. As with `flex-grow`, `flex-shrink` will only be employed if the parent container is too small or the browser is adjusted.

Example : GROW & SHRINK

```
<!DOCTYPE html>
<html>

<head>
  <title>Flex Grow and Shrink</title>
<style>
body {
  margin: 0;
  border: 0;
  font-family: 'Roboto Mono', monospace;
}

h1 {
  font-size: 18px;
}

h2 {
  font-size: 16px;
}

h1,
h2 {
  text-align: center;
```

```
}

.top,
.middle,
.bottom {
  width: 100px;
  height: 100px;
  background-color: Dodgerblue;
  border: 2px solid lightgrey;
  margin: 10px 30px;
}

.top.side {
 flex-grow:1;
  flex-shrink:2;
}

.top.center {
  flex-grow:1;
}

.middle.side {
flex-shrink:0;
}

.middle.center {
flex-grow:1;}

.bottom.side {
flex-grow:1;
}
.bottom.center {
  flex-grow:2;
    flex-shrink:2;
}
#top,
#middle,
#bottom {
  display: flex;
  background-color: Whitesmoke;
  justify-content: center;
  min-height: 200px;
  align-items: center;
}
</style>
</head>
<body>
  <h1>Step 1</h1>
  <div id="top">
```

```
    <div class="top side">
      <h2>1</h2>
    </div>
    <div class="top center">
      <h2>2</h2>
    </div>
    <div class="top side">
      <h2>3</h2>
    </div>
  </div>
  <h1>Step 2</h1>
  <div id="middle">
    <div class="middle side">
      <h2>1</h2>
    </div>
    <div class="middle center">
      <h2>2</h2>
    </div>
    <div class="middle side">
      <h2>3</h2>
    </div>
  </div>
  <h1>Step 3</h1>
  <div id="bottom">
    <div class="bottom side">
      <h2>1</h2>
    </div>
    <div class="bottom center">
      <h2>2</h2>
    </div>
    <div class="bottom side">
      <h2>3</h2>
    </div>
  </div>
</body>
</html>
```

flex-basis

In the previous two exercises, the dimensions of the divs were determined by heights and widths set with CSS. Another way of specifying the width of a flex item is with the `flex-basis` property. `flex-basis` allows us to specify the width of an item before it stretches or shrinks.

```
<div class="container">
  <div class=”side”>
    <h1>Left side!</h1>
```

```
  </div>
  <div class="center">
    <h1>Center!</h1>
  </div>
  <div class="side">
    <h1>Right side!</h1>
  </div>
</div>
.container {
  display: flex;
}

.side {
  flex-grow: 1;
  flex-basis: 100px;
}

.center {
  flex-grow: 2;
  flex-basis: 150px;
}
```

In the example above, the `.side` divs will be 100 pixels wide and the `.center` div will be 150 pixels wide if the `.container` div has just the right amount of space (350 pixels, plus a little extra for margins and borders). If the `.container` div is larger, the `.center` div will absorb twice as much space as the `.side` divs.

The same would hold true if we assigned `flex-shrink` values to the divs above as well.

flex

The `flex` property provides a convenient way for specifying how elements stretch and shrink, while simplifying the CSS required. The `flex`property allows you to declare `flex-grow`, `flex-shrink`, and `flex-basis` all in one line.

Note: The `flex` *property* is different from the `flex` *value* used for the `display` property.

```
.big {
  flex-grow: 2;
  flex-shrink: 1;
  flex-basis: 150px;
}

.small {
  flex-grow: 1;
  flex-shrink: 2;
  flex-basis: 100px;
}
```

In the example above, all elements with class `big`will grow twice as much as elements with class `small`. Keep in mind, this doesn't mean `big`items will be twice as big as `small` items, they'll just take up more of the extra space.

These three properties in one line.

```
.big {
  flex: 2 1 150px;
}

.small {
  flex: 1 2 100px;
}
```

In the example above, we use the `flex` property to declare the values for `flex-grow`, `flex-shrink`, and `flex-basis` (in that order) all in one line.

```
.big {
 flex: 2 1;
}
```

In the example above, we use the `flex` property to declare `flex-grow` and `flex-shrink`, but not `flex-basis`.

```
.small {
  flex: 1 20px;
}
```

In the example above, we use the `flex` property to declare `flex-grow` and `flex-basis`. Note that there is no way to set only `flex-shrink` and `flex-basis` using 2 values.

flex-wrap

Sometimes, we don't want our content to shrink to fit its container. Instead, we might want flex items to move to the next line when necessary. This can be declared with the `flex-wrap`property. The `flex-wrap` property can accept three values:

1. `wrap` — child elements of a flex container that don't fit into a row will move down to the next line
2. `wrap-reverse` — the same functionality as `wrap`, but the order of rows within a flex container is reversed (for example, in a 2-row flexbox, the first row from a `wrap` container will become the second in `wrap-reverse` and the second row from the `wrap` container will become the first in `wrap-reverse`)
3. `nowrap` — prevents items from wrapping; this is the default value and is only necessary to override a wrap value set by a different CSS rule.

Note: The `flex-wrap` property is declared on flex *containers*.

```
.container {
  background-color: dodgerblue;
```

```
  display: flex;
  align-items: center;
  min-height: 125px;
  justify-content:space-around;
}

.box {
  background-color: whitesmoke;
  border: 1px solid white;
  width: 100px;
  height: 100px;
}

#wrap {
flex-wrap:wrap;
}
/*
#nowrap {
flex-wrap:nowrap;
}

#reverse {
flex-wrap:wrap-reverse;
}
*/

<body>
  <h1>Flex-Wrap: Wrap</h1>
  <div class="container" id="wrap">
    <div class="box">
      <h3>1</h3>
    </div>
    <div class="box">
      <h3>2</h3>
    </div>
    <div class="box">
      <h3>3</h3>
    </div>
</div>
</body>
```

Align-content

Now that elements can wrap to the next line, we might have multiple rows of flex items within the same container.

We already used the `align-items` property to space flex items from the top to the bottom of a flex container. `align-items` is for aligning elements within a single row. If a flex

container has multiple rows of content, we can use `align-content` to space the rows from top to bottom.

`align-content` accepts six values:

1. `flex-start` — all rows of elements will be positioned at the top of the parent container with no extra space between.
2. `flex-end` — all rows of elements will be positioned at the bottom of the parent container with no extra space between.
3. `center` — all rows of elements will be positioned at the center of the parent element with no extra space between.
4. `space-between` — all rows of elements will be spaced evenly from the top to the bottom of the container with no space above the first or below the last.
5. `space-around` — all rows of elements will be spaced evenly from the top to the bottom of the container with the same amount of space at the top and bottom and between each element.
6. `stretch` — if a minimum height or no height is specified, the rows of elements will stretch to fill the parent container from top to bottom (default value).

```
<div class="container">
  <div class=”child”>
    <h1>1</h1>
  </div>
  <div class="child">
    <h1>2</h1>
  </div>
  <div class="child">
    <h1>3</h1>
  </div>
  <div class="child">
    <h1>4</h1>
  </div>
</div>
.container {
  display: flex;
  width: 400px;
  height: 400px;
  flex-wrap: wrap;
  align-content: space-around;
}

.child {
  width: 150px;
  height: 150px;
}
```

In the example above, there are four flex items inside of a flex container. The flex items are set to be 150 pixels wide each, but the parent container is only 400 pixels wide. This means that no more than two elements can be displayed inline. The other two elements will wrap

to the next line and there will be two rows of `divs` inside of the flex container. The `align-content` property is set to the value of `space-around`, which means the two rows of divs will be evenly spaced from top to bottom of the parent container with equal space before the first row and after the second, with double space between the rows.

Below, we will see each of the properties in action!

Note: The `align-content` property is declared on flex containers.

flex-direction

Up to this point, we've only covered flex items that stretch and shrink horizontally and wrap vertically. As previously stated, flex containers have two axes: a *major axis* and a *cross axis*. By default, the major axis is horizontal and the cross axis is vertical.

The major axis is used to position flex items with the following properties:

1. `justify-content`
2. `flex-wrap`
3. `flex-grow`
4. `flex-shrink`

The cross axis is used to position flex items with the following properties:

1. `align-items`
2. `align-content`

The major axis and cross axis are interchangeable. We can switch them using the `flex-direction` property. If we add the `flex-direction` property and give it a value of `column`, the flex items will be ordered vertically, not horizontally.

```
<div class="container">
  <div class="item">
    <h1>1</h1>
  </div>
  <div class="item">
    <h1>2</h1>
  </div>
  <div class="item">
    <h1>3</h1>
  </div>
  <div class="item">
    <h1>4</h1>
  </div>
  <div class="item">
```

```
    <h1>5</h1>
  </div>
</div>
.container {
  display: flex;
  flex-direction: column;
  width: 1000px;
}
.item {
  height: 100px;
  width: 100px;
}
```

In the example above, the five divs will be positioned in a vertical column. All of these divs could fit in one horizontal row. However, the `column` value tells the browser to stack the divs one on top of the other. As explained above, properties like `justify-content` will not behave the way they did in previous examples.

The `flex-direction` property can accept four values:

1. `row` — elements will be positioned from left to right across the parent element starting from the top left corner (default).
2. `row-reverse` — elements will be positioned from right to left across the parent element starting from the top right corner.
3. `column` — elements will be positioned from top to bottom of the parent element starting from the top left corner.
4. `column-reverse` — elements will be positioned from the bottom to the top of the parent element starting from the bottom left corner.

Below, we'll investigate how these work.

Note: The `flex-direction` property is declared on flex containers.

flex-flow

Like the `flex` property, the `flex-flow` property is used to declare both the `flex-wrap` and `flex-direction` properties in one line.

```
.container {
  display: flex;
  flex-wrap: wrap;
  flex-direction: column;
}
```

In the example above, we take two lines to accomplish what can be done with one.

```
.container {
  display: flex;
  flex-flow: column wrap;
```

```
}
```

In the example above, the first value in the `flex-flow` declaration is a `flex-direction` value and the second is a `flex-wrap` value. All values for `flex-direction` and `flex-wrap` are accepted.

Note: The `flex-flow` property is declared on flex containers.

Nested Flexboxes

So far, we've had multiple flex containers on the same page to explore flex item positioning. It is also possible to position flex containers inside of one another.

```
<div class="container">
  <div class="left">
    <img class="small" src="#"/>
    <img class="small" src="#"/>
    <img class="small" src="#" />
  </div>
  <div class="right">
    <img class="big" src="#" />
  </div>
</div>
.container {
  display: flex;
  justify-content: center;
  align-items: center;
}

.left {
  display: inline-flex;
  flex: 2 1 200px;
  flex-direction: column;
}

.right {
  display: inline-flex;
  flex: 1 2 400px;
  align-items: center;
}

.small {
  height: 200px;
  width: auto;
}

.large {
  height: 600px;
  width: auto;
}
```

In the example above, a div with three smaller images will display from top to bottom on the left of the page (`.left`). There is also a div with one large image that will display on the right side of the page (`.right`). The left div has a smaller `flex-basis` but stretches to fill more extra space; the right div has a larger `flex-basis` but stretches to fill less extra space. Both divs are flex items *and* flex containers. The items have properties that dictate how they will be positioned in the parent container and how their flex item children will be positioned in them.

We'll use the same formatting above to layout the simple page to the right.

Introduction to Grids

On the box model, Flexbox, and CSS display and positioning properties explain three possible ways to approach layout.

New powerful tool called *CSS Grid*. The ***grid* can be used to lay out entire web pages**. **Whereas Flexbox is mostly useful for positioning** items in a one-dimensional layout, **CSS grid is most useful for two-dimensional layouts,** providing many tools for aligning and moving elements across both rows and columns.

Use these properties to create grid layouts:

- `grid-template-columns`
- `grid-template-rows`
- `grid-template`
- `grid-template-area`
- `grid-gap`
- `grid-row-start` / `grid-row-end`
- `grid-column-start` / `grid-column-end`
- `grid-area`

Creating a Grid

To set up a grid, you need to have both a *grid container* and *grid items*. The grid container will be a parent element that contains grid items as children and applies overarching styling and positioning to them.

HTML element into a grid container, you must set the element's `display` property to `grid` (for a block-level grid) or `inline-grid` (for an inline grid).

```
.grid {
    border: 2px blue solid;
    width: 400px;
    height: 500px;
    display:grid;
    /*Right now, we haven't specified the number of rows or columns in our grid, so every item is sitting on a new row. */

  }
```

Creating Columns

By default, grids contain only one column. If you were to start adding items, each item would be put on a new row; that's not much of a grid! To change this, we need to explicitly define the number of rows and columns in our grid. It works according to width.

We can define the columns of our grid by using this property `grid-template-columns`.

```
.grid {
  display: grid;
  width: 500px;
  grid-template-columns: 100px 200px;
}
```

This property creates two changes. First, it defines the number of columns in the grid; in this case, there are two. Second, it sets the width of each column. The first column will be 100 pixels wide and the second column will be 200 pixels wide.

We can also define the size of our columns as a percentage of the entire grid's width.

```
.grid {
  display: grid;
  width: 1000px;
  grid-template-columns: 20% 50%;
}
```

In this example, the grid is 1000 pixels wide. Therefore, the first column will be 200 pixels wide because it is set to be 20% of the grid's width. The second column will be 500 pixels wide.

We can also mix and match these two units. In the example below, there are three columns of width 20 pixels, 40 pixels, and 60 pixels:

```
.grid {
  display: grid;
  width: 100px;
  grid-template-columns: 20px 40% 60px;
}
```

Notice that in this example, the total width of our columns (120 pixels) exceeds the width of the grid (100 pixels). This might make our grid cover other elements on the page! We can avoid overflow .

Creating Rows

Use the property `grid-template-rows`. It works according to height.

This property is almost identical to `grid-template-columns`.

```
.grid {
  display: grid;
  width: 1000px;
  height: 500px;
  grid-template-columns: 100px 200px;
  grid-template-rows: 10% 20% 600px;
}
```

This grid has two columns and three rows. `grid-template-rows` defines the number of rows and sets each row's height. In this example, the first row is 50 pixels tall (10% of 500), the second row is 100 pixels tall (20% of 500), and the third row is 600 pixels tall.

When using percentages in these two properties, remember that rows are defined as a percentage of the grid's height, and columns are defined as a percentage of its width.

Grid Template

The property `grid-template` can replace the previous two CSS properties. Both `grid-template-rows` and `grid-template-columns` are nowhere to be found in the following code!

```
.grid {
  display: grid;
  width: 1000px;
  height: 500px;
  grid-template: 200px 300px / 20% 10% 70%;
// first height(row) then width(column)
}
```

When using `grid-template`, the values before the slash will determine the size of each row. The values after the slash determine the size of each column. In this example, we've made two rows and three columns of varying sizes.

Fraction

You may already be familiar with several types of responsive units such as percentages (`%`), `em`s and `rem`s. CSS Grid introduced a new relative sizing unit — `fr`, like fraction.

By using the `fr` unit, we can define the size of columns and rows as a fraction of the grid's length and width.

This unit was specifically created for use in CSS Grid. Using `fr` makes it easier to prevent grid items from overflowing the boundaries of the grid. Consider the code below:

```
css .grid { display: grid; width: 1000px; height: 400px; grid-template: 2fr 1fr 1fr / 1fr 3fr 1fr; }
```

In this example, the grid will have three rows and three columns. The rows are splitting up the available 400 pixels of height into four parts. The first row gets two of those parts, the second row gets one, and the third row gets one. Therefore the first row is 200 pixels tall, and the second and third rows are 100 pixels tall.

Each column's width is a fraction of the available space. In this case, the available space is split into five parts. The first column gets one-fifth of the space, the second column gets three-fifths, and the last column gets one-fifth. Since the total width is 1000 pixels, this means that the columns will have widths of 200 pixels, 600 pixels, and 200 pixels respectively.

It is possible to use `fr` with other units as well. When this happens, each `fr` represents a fraction of the *available* space.

```
css .grid { display: grid; width: 100px; grid-template-columns: 1fr 60px 1fr; }
```

In this example, 60 pixels are taken up by the second column. Therefore the first and third columns have 40 available to split between them. Since each gets one fraction of the total, they both end up being 20 pixels wide.

Repeat

The properties that define the number of rows and columns in a grid can take a function as a value. `repeat()` is one of these functions. The `repeat()` function was created specifically for CSS Grid.

```
.grid {
  display: grid;
  width: 300px;
  grid-template-columns: repeat(3, 100px);
}```

The repeat function will duplicate the specifications for rows or columns a given number of times. In the example above, using the repeat function will make the grid have three columns that are each 100 pixels wide. It is the same as writing:
```css
grid-template-columns: 100px 100px 100px;
```

Repeat is particularly useful with `fr`. For example, `repeat(5, 1fr)` would split your table into five equal rows or columns.

Finally, the second parameter of `repeat()` can have multiple values.

```
grid-template-columns: repeat(2, 20px 50px) // one slap at a-time repeat 2 time :P
```

This code will create four columns where the first and third columns will be 20 pixels wide and the second and fourth will be 50 pixels wide.

Example :

```
.grid {
    display: grid;
    border: 2px blue solid;
    width: 400px;
    height: 500px;
    grid-template: repeat(3, 1fr) / 3fr 50% 1fr;
  }
```

minmax

So far, all of the grids that we have worked with have been a fixed size. The grid in our example has been 400 pixels wide and 500 pixels tall. But sometimes you might want a grid to resize based on the size of your web browser.

You might want to prevent a row or column from getting too big or too small.
```

```
.grid {
  display: grid;
  grid-template-columns: 100px minmax(100px, 500px) 100px;
}```

In this example, the first and third columns will always be 100 pixels wide, no matter the size of the grid. The second column, however, will vary in size as the overall grid resizes. The second column will always be between 100 and 500 pixels wide.
```

Example:-

```
<body>
    <div class="grid">
      <div class="box a">A</div>
      <div class="box b">B</div>
      <div class="box c">C</div>
      <div class="box d">D</div>
      <div class="box e">E</div>
      <div class="box f">F</div>
      <div class="box f">G</div>
    </div>
  </body>
```

```
.grid {
    display: grid;
    border: 2px blue solid;

    height: 500px;
    grid-template: repeat(3, 1fr) / 3fr minmax(50px,300px) 1fr;
  }

.box {
  background-color: beige;
  color: black;
  border-radius: 5px;
  border: 2px dodgerblue solid;
}
```

- Delete the `width` property from `.grid`.
- If you resize your browser, you will see the grid change size with the window.
- Using `minmax()`, change the second column to be between 50 pixels and 300 pixels.

Grid Gap

The CSS properties `grid-row-gap` and `grid-column-gap` will put blank space between every row and column in the grid.

`css .grid { display: grid; width: 320px; grid-template-columns: repeat(3, 1fr); grid-column-gap: 10px; }`

It is important to note that `grid-gap` does not add space at the **beginning or end** of the grid. In the example code, our grid will have three columns with two ten-pixel gaps between them.

Let's quickly calculate how wide these columns are. Remember that using `fr` considers all of the *available* space. The grid is 320 pixels wide and **20 of those pixels are taken up by the two grid gaps**. Therefore each column takes a piece of the 300 available pixels. Each column gets `1fr`, so the columns are evenly divided into thirds (or 100 pixels each).

Finally, there is a **CSS property** `grid-gap` that can set the row and column gap at the same time. `grid-gap: 20px 10px;` will set the distance between rows to 20 pixels and the distance between columns to 10 pixels. Unlike other CSS grid properties, this shorthand does not take a `/`between values! **If only one value is given, it will set the column gap and the row gap to that value.**

```
.grid {
  display: grid;
  border: 2px blue solid;
  width: 1300px;
  height: 600px;
  grid-template: repeat(3, 1fr) / repeat(3, 1fr);
  grid-gap: 20px 20px;
}

.box {
  background-color: beige;
  color: black;
  border-radius: 5px;
  border: 2px dodgerblue solid;
}
```

Multiple Row Items

Using the CSS properties `grid-row-start` and `grid-row-end`, we can make single grid items take up multiple rows. Remember, we are no longer applying CSS to the outer grid container; we're adding CSS to the elements sitting inside the grid!

```
.item {
  grid-row-start: 5;
  grid-row-end: 9;
}
```

In this example, the HTML element of class `item`will take up two rows in the grid, rows 1 and 2. The values that `grid-row-start` and `grid-row-end`accept are *grid lines*.

Row grid lines and column grid lines start at 1 and end at a value that is 1 greater than the number of rows or columns the grid has. For example, if a grid has 5 rows, the grid row lines range from 1 to 6. If a grid has 8 columns, the grid row lines range from 1 to 9.

The value for `grid-row-start` should be the row at which you want the grid item to begin. The value for `grid-row-end` should be one greater than the row at which you want the grid item to end. An element that covers rows 2, 3, and 4 should have these declarations: `grid-row-start: 2` and `grid-row-end: 5`.

It is possible for the value of `grid-row-start` to be greater than that of `grid-row-end`. Both properties can also each have negative values. Consult the documentation to learn more about how to use these features.

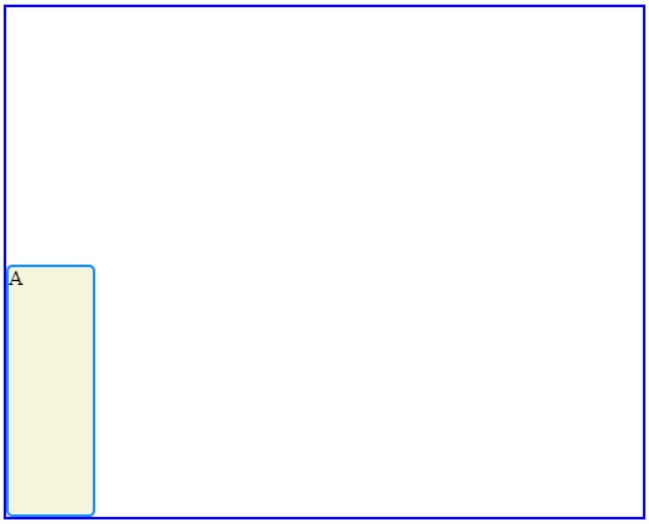
```
<body>
    <div class="grid">
      <div class="box a">A</div>
    </div>
  </body>
```

```
.grid {
    display: grid;
    border: 2px blue solid;
    height: 500px;
    width:500px;
    grid-template: repeat(4, 1fr 2fr) / repeat(4, 3fr 2fr);
    grid-gap: 5px;
  }

  .a {
    grid-row-start: 5;
    grid-row-end: 7;
  }

  .box {
    background-color: beige;
```

```
    color: black;
    border-radius: 5px;
    border: 2px dodgerblue solid;
  }
```



Grid Row

We can use the property `grid-row` as shorthand for `grid-row-start` and `grid-row-end`. The following two code blocks will produce the same output:

```
.item {
  grid-row-start: 4;
  grid-row-end: 6;
}
.item {
  grid-row: 4 / 6;
}
```

This code should look similar to the way `grid-template` is shorthand for `grid-template-rows`and `grid-template-columns`. In this case, the starting row goes before the "/" and the ending row goes after it. Again, the ending row is exclusive; this grid item will occupy rows four and five.

When an item spans multiple rows or columns using these properties, it will also include the `grid-gap` if any exists. For example, if an item spans two rows of height 100 pixels and there is a ten-pixel grid-gap, then the item will have a total height of 210 pixels.

Grid Column

The previous three properties also exist for columns. `grid-column-start`, `grid-column-end` and `grid-column` work identically to the row properties. These properties allow a grid item to span multiple columns.

When using these properties, we can use the keyword `span` to start or end a column or row relative to its other end. Look at how `span` is used in the code below:

```
.item {
  grid-column: 4 / span 2;
}
```

This is telling the `item` element to begin in column four and take up two columns of space. So `item` would occupy columns four and five. It produces the same result as the following code blocks:

```
.item {
  grid-column: 4 / 6;
}
.item {
  grid-column-start: 4;
  grid-column-end: span 2;
}
.item {
  grid-column-start: span 2;
  grid-column-end: 6;
}
```

`span` is a useful keyword, because it avoids off-by-one errors (miscalculating the ending grid line) you might make when determining the ending grid line of an element. If you know where you want your grid item to start and how long it should be, use `span`!

```
<body>
    <div class="grid">
      <div class="box a">A</div>
      <div class="box b">B</div>
    </div>
  </body>
```

```
.grid {
    display: grid;
    border: 2px blue solid;
    height: 500px;
    width:500px;
    grid-template: repeat(4, 1fr 2fr) / repeat(4, 3fr 2fr);
    grid-gap: 5px;
  }
  .a {
    grid-row: 4 / span 1 ;
    grid-column: 4 / span 2;
  }

  .b {
    grid-column: 2 / span 6;
```

```
    grid-row: 2 / span 2;
  }
  .box {
    background-color: beige;
    color: black;
    border-radius: 5px;
    border: 2px dodgerblue solid;
  }
```

Grid Area

We've already been able to use `grid-row` and `grid-column` as shorthand for properties like `grid-row-start` and `grid-row-end`. We can refactor even more using the property `grid-area`. This property will set the starting and ending positions for both the rows and columns of an item.

```
.item {
  grid-area: 2 / 3 / 4 / span 5;
}
```

`grid-area` takes four values separated by slashes. The order is important! This is how `grid-area` will interpret those values.

1. `grid-row-start`
2. `grid-column-start`
3. `grid-row-end`
4. `grid-column-end`

In the above example, the item will occupy rows two and three and columns three through eight. Using `grid-area` is an easy way to place items exactly where you want them in a grid.

```
<div class="grid">
    <div class="box a">A</div>
    <div class="box b">B</div>
    <div class="box c">C</div>
  </div>
```

```
.a {
    grid-area: 5 / 1 / span 2 / span 2;
  }

  .b {
    grid-area: 2 / 2 / span 3 / span 6;
  }

  .c {
    grid-area: 6 / 8 / span 3 / span 1;
```

```
  }
```

Example:-

```
<body>
    <div class="grid">
      <div class="box a">A</div>
      <div class="box b">B</div>
      <div class="box c">C</div>
      <div class="box d">D</div>
      <div class="box e">E</div>
    </div>
  </body>
```

```
<style>
.grid {
  border: 2px blue solid;
  height: 500px;
  width: 500px;
 display:grid;
  grid-template-columns: 25% 25% 2fr 1fr ;
   grid-template-rows: 200px 200px;
  grid-gap:10px 15px;
}

.box {
  background-color: beige;
  color: black;
  border-radius: 5px;
  border: 2px dodgerblue solid;
}
.a {
  grid-column: 1 / span 2;
   grid-row: 1 /  3;
}
</style>
```

How tall will the .item element be if the following CSS is in effect?

```
.layout {
    grid-template-columns: 200px 200px 200px;
    grid-template-rows: 100px 100px 200px 100px;
    grid-gap: 10px;
  }
  .item {
    grid-column-start: 2;
    grid-column-end: 3;
    grid-row-start: 2;
    grid-row-end: 4;
  }
```

`310px`

How wide will the .item element be if the following CSS is in effect?

```
.layout {
    grid-template-columns: 200px 200px 200px;
    grid-template-rows: 100px 100px 200px 100px;
    grid-gap: 10px;
  }
  .item {
    grid-column-start: 2;
    grid-column-end: 3;
    grid-row-start: 2;
    grid-row-end: 4;
  }
```

200px

Which of the following would make a grid with 3 columns where the middle column takes up 60% of the space, the first column takes up 1/3 of the remaining space and the last column takes up 2/3 of the remaining space?

`grid-template-columns: 1fr 60% 2fr;`

Introduction

Now we can create a two-dimensional grid-based layout for your web pages! This time we will learn the following additional properties that you can use to harness the power of CSS Grid Layout:

- `grid-template-areas`
- `justify-items`
- `justify-content`
- `justify-self`
- `align-items`
- `align-content`
- `align-self`
- `grid-auto-rows`
- `grid-auto-columns`
- `grid-auto-flow`

The *explicit* and *implicit*grids and *grid axes*.

Grid Template Areas

The `grid-template-areas` property allows you to name sections of your web page to use as values in the `grid-row-start`, `grid-row-end`, `grid-col-start`,`grid-col-end`, and `grid-area` properties.

```
<div class="container">
  <header>Welcome!</header>
  <nav>Links!</nav>
  <section class="info">Info!</section>
  <section class="services">Services!</section>
  <footer>Contact us!</footer>
</div>
.container {
  display: grid;
  max-width: 900px;
  position: relative;
  margin: auto;
  grid-template-areas: "head head"
                       "nav nav"
                       "info services"
                       "footer footer";
  grid-template-rows: 300px 120px 800px 120px;
  grid-template-columns: 1fr 3fr;
}

header {
  grid-area: head;
}
```

```css
nav {
  grid-area: nav;
}

.info {
  grid-area: info;
}

.services {
  grid-area: services;
}

footer {
  grid-area: footer;
}
```

You may want to expand this section of the website to view the code above more clearly.

1. In the example above, the HTML creates a web page with five distinct parts.
2. The `grid-template-areas` declaration in the `.container` rule set creates a 2-column, 4-row layout.
3. The `grid-template-rows` declaration specifies the height of each of the four rows from top to bottom: 300 pixels, 120 pixels, 800 pixels, and 120 pixels.
4. The `grid-template-columns` declaration uses the `fr` value to cause the left column to use one fourth of **the available space on the page and the right column to use three-fourths of the available space on the page.**
5. In each rule set below `.container`, we use the `grid-area` property to tell that section to cover the portion of the page specified. The `header` element spans the first row and both columns. The `nav` element spans the second row and both columns. The element with class `.info` spans the third row and left column. The element with class `.services`spans the third row and right column. The `footer` element spans the bottom row and both columns.
6. That's it! An entire page laid out in 40 lines of code.

This property is declared on grid containers.

You can see what you'll be building in this exercise here.

Example:2

1. In **style.css**, add the `grid-template-areas`property to the `.container` rule set. Its value should create a 2-column, 4-row layout with these areas:

- header (spans two columns in the first row)
- nav (spans two columns in the second row)
- left (spans one column on the left in the third row)
- right (spans one column on the right in the third row)

- footer (spans two columns in the fourth row)

2. Follow the same pattern for the header, `nav`, `.left`, `.right`, and `footer` rule sets in **style.css**. and give sizes to rows and column.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Grid Stuff</title>
  <link rel="stylesheet" type="text/css" href="style.css" />
  <style>
  .container {
  display: grid;
  max-width: 900px;
  position: relative;
  margin: auto;
  grid-gap: 10px;
  grid-template-areas: "header header"
                                          "nav        nav"
                                          "left       right"
                                          "footer footer";
  grid-template-columns: 200px 400px;
  grid-template-rows: 150px 200px 600px 200px;
}

h1, h2 {
  font-family: monospace;
  text-align: center;
}

header {
  background-color: dodgerblue;
  grid-area: header;
}

nav {
  background-color: beige;
  grid-area: nav;
}

.left {
  background-color: dodgerblue;
  grid-area: left;
}
```

```
.right {
  background-color: beige;
  grid-area: right;
}

footer {
  background-color: dodgerblue;
  grid-area: footer;
}
  </style>
</head>

<body>
  <div class="container">
    <header>
      <h1>Header</h1>
    </header>
    <nav>
      <h1>Nav</h1>
    </nav>
    <section class="left">
      <h2>Left</h2>
    </section>
    <section class="right">
      <h2>Right</h2>
    </section>
    <footer>
      <h1>Footer</h1>
    </footer>
  </div>
</body>

</html>
```

Overlapping Elements

Another powerful feature of CSS Grid Layout is the ability to easily overlap elements.

When overlapping elements, it is generally easiest to use grid line names and the `grid-area`property.

```
<div class="container">
  <div class="info">Info!</div>
  <img src="#" />
  <div class="services">Services!</div>
</div>
.container {
  display: grid;
  grid-template: repeat(8, 200px) / repeat(6, 100px);
}

.info {
  grid-area: 1 / 1 / 9 / 4;
}

.services {
  grid-area: 1 / 4 / 9 / 7;
}

img {
  grid-area: 2 / 3 / 5 / 5;
  z-index: 5;
}
```

In the example above, there is a grid container with eight rows and six columns. There are three grid items within the container — a `<div>` with the class `info`, a `<div>` with the class `services`, and an image.

The `info` section covers all eight rows and the first three columns.
The `services` section covers all eight rows and the last three columns.

The image spans the 2nd, 3rd, and 4th rows and the 3rd and 4th columns.

The z-index property tells the browser to render the image element on top of the `services` and `info` sections so that it is visible.

Example:2

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
```

```
  <title>Grid Stuff</title>
  <link rel="stylesheet" type="text/css" href="style.css" />
  <style>
  .container {
  display: grid;
  max-width: 900px;
  position: relative;
  margin: auto;
  grid-gap: 10px;
  grid-template: repeat(12, 1fr) / repeat(6, 1fr);
}

h1, h2, h3 {
  font-family: monospace;
  text-align: center;
}

header {
  background-color: dodgerblue;
  grid-area: 1 / 1 / 3 / 7;
}

nav {
  background-color: beige;
  grid-area: 3 / 1 / 4 / 7;
}

.left {
  background-color: dodgerblue;
  grid-area:4 / 1 / 9 / 5;
}

.right {
  background-color: beige;
    grid-area:4 / 5 / 9 / 7;
}

.overlap {
  background-color: lightcoral;
  grid-area:6 / 4 / 8 / 6;
  z-index:5;
}

footer {
  background-color: dodgerblue;
  grid-area:9 / 1 / 13 / 7;
}
  </style>
</head>
```

```
<body>
  <div class="container">
    <header>
      <h1>Header</h1>
    </header>
    <nav>
      <h1>Nav</h1>
    </nav>
    <section class="left">
      <h2>Left</h2>
    </section>
    <div class="overlap">
      <h3>Overlap!</h3>
    </div>
    <section class="right">
      <h2>Right</h2>
    </section>
    <footer>
      <h1>Footer</h1>
    </footer>
  </div>
</body>

</html>
```

Justify Items

We have referred to "two-dimensional grid-based layout" .There are two axes in a grid layout — the *column*(or block) axis and the *row* (or inline) axis.

The column axis stretches from top to bottom across the web page.

The row axis stretches from left to right across the web page.

In the following four exercises, we will learn and use properties that rely on an understanding of grid axes.

`justify-items` is a property that positions grid items along the inline, or row, axis. This means that it positions items from left to right across the web page.

`justify-items` accepts these values:

- `start` — aligns grid items to the left side of the grid area
- `end` — aligns grid items to the right side of the grid area
- `center` — aligns grid items to the center of the grid area
- `stretch` — stretches all items to fill the grid area

There are several other values that `justify-items`accepts, which you can read about on the https://developer.mozilla.org/en-US/docs/Web/CSS/justify-items#Values

The definitions for these values can also be found in the https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Justifying_Items_on_the_Inline_or_Row_Axis

It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

```
<main>
  <div class="card">Card 1</div>
  <div class="card">Card 2</div>
  <div class="card">Card 3</div>
</main>
main {
  display: grid;
  grid-template-columns: repeat(3, 400px);
  justify-items: center;
}
```

In the example above, we use `justify-items` to adjust the positioning of some elements on this web page.

1. There is a grid container with three columns that are each 400 pixels wide.
2. The container has three grid items that do not have a specified width.

3. Without setting the `justify-items` property, these elements will span the width of the column they are in (400 pixels).
4. By setting the `justify-items` property to `center`, the `.card` `<div>`s will be centered inside of their columns. They will only be as wide as necessary to contain their content (the words Card 1, etc).
5. If we specify a width for the `.card` elements, they will not stretch the width of their column.

This property is declared on grid containers.

Justify Content

In the previous exercise, we learned how to position elements within their columns. we will learn how to position a grid within its parent element.

We can use `justify-content` to position the entire grid along the row axis.

It accepts these values:

- `start` — aligns the grid to the left side of the grid container
- `end` — aligns the grid to the right side of the grid container
- `center` — centers the grid horizontally in the grid container
- `stretch` — stretches the grid items to increase the size of the grid to expand horizontally across the container
- `space-around` — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element
- `space-between` — includes an equal amount of space between grid items and no space at either end
- `space-evenly` — places an even amount of space between grid items and at either end

There are several other values that `justify-content` accepts, which you can read about on the https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_the_grid_tracks_on_the_block_or_column_axis

. The definitions for these values can also be found in the https://developer.mozilla.org/en-US/docs/Web/CSS/justify-content#Values

It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

```
<main>
  <div class="left">Left</div>
  <div class="right">Right</div>
</main>
main {
  display: grid;
  width: 1000px;
  grid-template-columns: 300px 300px;
  grid-template-areas: "left right";
  justify-content: center;
}
```

1. In the example above, the grid container is 1000 pixels wide, but we only specified two columns that are 300 pixels each. This will leave 400 pixels of unused space in the grid container.
2. `justify-content: center;` positions the columns in the center of the grid, leaving 200 pixels on the right and 200 pixels on the left of the grid.

```
main {
  display: grid;
  grid-template-columns: 250px 250px;
  grid-template-rows: repeat(3, 450px);
  grid-gap: 20px;
  margin-top: 44px;
  grid-auto-rows: 500px;
/*items comes in center in inside respective comumns*/
  justify-items:center;


  /*items are centered the columns according to the width of the page*/

  justify-content:center;

}
```

Align Items

how to position grid items from top to bottom!

`align-items` is a property that positions grid items along the block, or column axis. This means that it positions items from top to bottom.

`align-items` accepts these values:

- `start` — aligns grid items to the top side of the grid area
- `end` — aligns grid items to the bottom side of the grid area
- `center` — aligns grid items to the center of the grid area
- `stretch` — stretches all items to fill the grid area

There are several other values that `align-items`accepts, which you can read about on the https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout/Box_Alignment_in_CSS_Grid_Layout#Aligning_items_on_the_block_or_column_Axis

The definitions for these values can also be found in the https://developer.mozilla.org/en-US/docs/Web/CSS/justify-items#Values

It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

```
<main>
  <div class="card">Card 1</div>
  <div class="card">Card 2</div>
  <div class="card">Card 3</div>
</main>
main {
  display: grid;
  grid-template-rows: repeat(3, 400px);
  align-items: center;
}
```

In the example above, we use `align-items` to adjust the positioning of some elements on this web page.

1. There is a grid container with three rows that are 400 pixels tall.
2. The container has three grid items that do not have a specified width.
3. Without setting the `align-items` property, these elements will span the height of the row they are in (400 pixels).
4. By setting the `align-items` property to `center`, the `.card` `<div>`s will be centered vertically inside of their rows. They will only be as tall as necessary to contain their content (the words Card 1, etc).
5. If we specify a height for the `.card` elements, they will not stretch the height of their row even if `align-items: stretch;` is set.

This property is declared on grid containers.

Example:

```
main {
  display: grid;
```

```
  grid-template-columns: 250px 250px;
  grid-template-rows: repeat(3, 450px);
  grid-gap: 20px;
  margin-top: 44px;
  grid-auto-rows: 500px;
  /*items are centered in their respective colums*/
  justify-items:center;
   justify-content:center;
   align-items:stretch;

}
```

Align Content

In the previous exercise, we positioned grid items within their rows. `align-content` positions the rows along the column axis, or from top to bottom.

It accepts these positional values:

- `start` — aligns the grid to the top of the grid container
- `end` — aligns the grid to the bottom of the grid container
- `center` — centers the grid vertically in the grid container
- `stretch` — stretches the grid items to increase the size of the grid to expand vertically across the container
- `space-around` — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element
- `space-between` — includes an equal amount of space between grid items and no space at either end
- `space-evenly` — places an even amount of space between grid items and at either end

There are several other values that `align-content`accepts, which you can read about on the Mozilla Developer Network. The definitions for these values can also be found in the documentation. It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

```
<main>
  <div class="top">Top</div>
  <div class="bottom">Bottom</div>
</main>
main {
  display: grid;
  height: 600px;
  rows: 200px 200px;
```

```
  grid-template-areas: "top"
                       "bottom";
  align-content: center;
}
```

1. In the example above, the grid container is 600 pixels tall, but we only specified two rows that are 200 pixels each. This will leave 200 pixels of unused space in the grid container.
2. `align-content: center;` positions the rows in the center of the grid, leaving 100 pixels at the top and 100 pixels at the bottom of the grid.

This property is declared on grid containers.

Justify Self and Align Self

The `justify-items` and `align-items` properties specify how all grid items contained within a single container will position themselves along the row and column axes, respectively.

`justify-self` specifies how an individual element should position itself with respect to the row axis. This property will override `justify-items` for any item on which it is declared.

`align-self` specifies how an individual element should position itself with respect to the column axis. This property will override `align-items` for any item on which it is declared.

They both accept these four properties:

- `start` — positions grid items on the left side/top of the grid area
- `end` — positions grid items on the right side/bottom of the grid area
- `center` — positions grid items on the center of the grid area
- `stretch` — positions grid items to fill the grid area (default)

`align-self` and `justify-self` accept the same values as `align-items` and `justify-items`.

These properties are declared on grid items.

```
<div class="recipe a">
    <img src="https://s3.amazonaws.com/codecademy-content/courses/learn-css-
grid/lesson-ii/image-1.png" />
    <h2>CHOCOLATE MOUSSE</h2>
    <img src="https://s3.amazonaws.com/codecademy-content/courses/learn-css-
grid/lesson-ii/time.svg" class="time" />
    <p class="mins">20 mins</p>
    <p class="description">
      This delicious chocolate mousse will delight dinner guests of all ages!</p>
  </div>
```

```
.a {
    /*
    align-self:end;
  */}

  .c {
    /*
    justify-self:start;
  */
  }
```

Implicit vs. Explicit Grid

So far, we have been explicitly defining the dimensions and quantities of our grid elements using various properties. This works well in many cases, such as a landing page for a business that will display a specific amount of information at all times.

However, there are instances in which we don't know how much information we're going to display. For example, consider online shopping. Often, these web pages include the option at the bottom of the search results to display a certain quantity of results or to display ALL results on a single page. When displaying all results, the web developer can't know in advance how many elements will be in the search results each time.

What happens if the developer has specified a 3-column, 5-row grid (for a total of 15 items), but the search results return 30?

Something called the *implicit* grid takes over. The implicit grid is an algorithm built into the specification for CSS Grid that determines default behavior for the placement of elements when there are more than fit into the grid specified by the CSS.

The default behavior of the implicit grid is as follows: items fill up rows first, adding new rows as necessary. New grid rows will only be tall enough to contain the content within them. Now, we will learn how to change this default behavior.

Grid Auto Rows and Grid Auto Columns

CSS Grid provides two properties to specify the size of grid tracks added implicitly: `grid-auto-rows` and `grid-auto-columns`.

`grid-auto-rows` specifies the height of implicitly added grid rows. `grid-auto-columns` specifies the width of implicitly added grid columns.

`grid-auto-rows` and `grid-auto-columns` accept the same values as their explicit counterparts, `grid-template-rows` and `grid-template-columns`:

- pixels (`px`)
- percentages (`%`)
- fractions (`fr`)
- the `repeat()` function

```
<body>
  <div>Part 1</div>
  <div>Part 2</div>
  <div>Part 3</div>
  <div>Part 4</div>
  <div>Part 5</div>
</body>
body {
  display: grid;
  grid: repeat(2, 100px) / repeat(2, 150px);
  grid-auto-rows: 50px;
}
```

In the example above, there are 5 `<div>`s. However, in the `section` rule set, we only specify a 2-row, 2-column grid — four grid cells.

The fifth `<div>` will be added to an implicit row that will be 50 pixels tall.

If we did not specify `grid-auto-rows`, the rows would be auto-adjusted to the height of the content of the grid items.

These properties are declared on grid containers.

Example:2

```
<html>
<head>
<style>
  body {
    display: grid;
    grid: repeat(2, 100px) / repeat(2, 100px);
    grid-auto-rows: 50px;
```

```
  }
.a {
border: 1px solid red;
}
</style>
</head>
<body>
    <div class="a">explict</div>
    <div class="a">explict</div>
    <div class="a">explict</div>
    <div class="a">explict</div>
    <div class="a">Part 5</div>
    <div class="a">Part 3</div>
    <div class="a">Part 4</div>
    <div class="a">Part 5</div>
        <div class="a">Part 1</div>
    <div class="a">Part 2</div>
    <div class="a">Part 3</div>
    <div class="a">Part 4</div>
    <div class="a">Part 5</div>
    <div class="a">Part 3</div>
    <div class="a">Part 4</div>
    <div class="a">Part 5</div>
  </body>
</html>
```

Grid Auto Columns

```
<!DOCTYPE html>
<html>
<head>
<style>
.grid-container {
  display: inline-grid;
  grid-template-columns: auto auto auto;
  background-color: #2196F3;
  padding: 10px;
}

.grid-item {
  background-color: rgba(255, 255, 255, 0.8);
  border: 1px solid rgba(0, 0, 0, 0.8);
  padding: 20px;
  font-size: 30px;
  text-align: center;
```

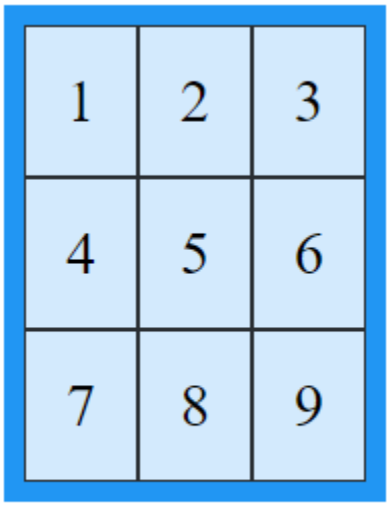
```
}
</style>
</head>
<body>

<h1>The display Property:</h1>

<div class="grid-container">
  <div class="grid-item">1</div>
  <div class="grid-item">2</div>
  <div class="grid-item">3</div>
  <div class="grid-item">4</div>
  <div class="grid-item">5</div>
  <div class="grid-item">6</div>
  <div class="grid-item">7</div>
  <div class="grid-item">8</div>
  <div class="grid-item">9</div>
</div>

<p>Set the <em>display</em> property to <em>inline-grid</em> to make an inline grid container.</p>


</body>
</html>
```



Set the *display* property to *inline-grid* to make an inline grid container.

Grid Auto Flow

In addition to setting the dimensions of implicitly-added rows and columns, we can specify the order in which they are rendered.

`grid-auto-flow` specifies whether new elements should be added to rows or columns.

`grid-auto-flow` accepts these values:

- `row` — specifies the new elements should fill rows from left to right and create new rows when there are too many elements (default)
- `column` — specifies the new elements should fill columns from top to bottom and create new columns when there are too many elements
- `dense` — this keyword invokes an algorithm that attempts to fill holes earlier in the grid layout if smaller elements are added

You can pair `row` and `column` with `dense`, like this: `grid-auto-flow: row dense;`.

This property is declared on grid containers.

Which of the following properties sizes implicit grid rows?

`grid-auto-rows`

Which of the following can be applied to a grid item (as opposed to the grid container)?

`align-self`

Imagine we have a grid with 4 items in it, with the following CSS properties. What width would the 3rd column be?

```
.grid {
    grid-auto-columns: 100px 200px;
    grid-auto-flow: column;
  }
```

`100px`

Awesome! The boxes will fill up implicitly created columns which alternate between being `100px` and `200px`. Odd rows will be `100px` in size.