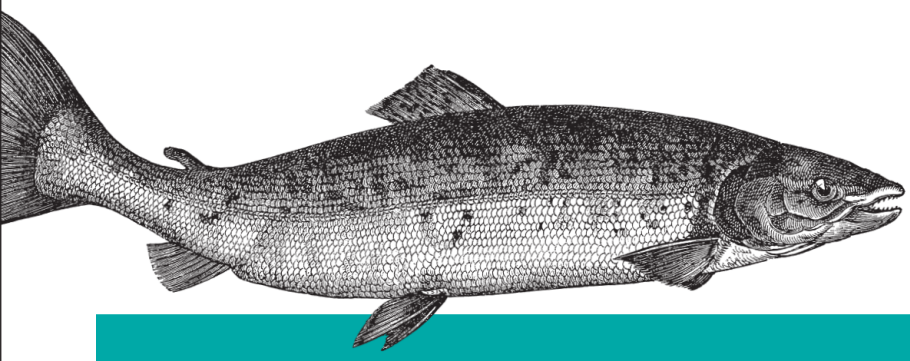
O'REILLY®



Flexbox in CSS

UNDERSTANDING CSS
FLEXIBLE BOX LAYOUT

Estelle Weyl

Flexbox in CSS

Understanding CSS Flexible Box Layout

Estelle Weyl

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Flexbox in CSS
by Estelle Weyl

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Colleen Lobner
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

June 2017: First Edition

Revision History for the First Edition
2017-05-23: First Release

978-1-491-98142-9

[LSI]

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a general note, tip, or suggestion.

Using Code Examples

Whenever you come across an icon that looks like ▶, it means there is an associated code example. Live examples are available at *http://standardista.com/flexbox/flexfiles*. You can either click the ▶ icon while reading this book to go directly to a live version of the code example referenced, or visit the link for a list of all of the code examples found in these chapters.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Flexible Boxes in CSS* by Estelle Weyl (O'Reilly). Copyright 2017 Estelle Weyl, 978-1-491-98142-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1

Flexbox

The CSS Flexible Box Module Level 1, or flexbox for short, makes the once-difficult task of laying out your page, widget, application, or gallery almost simple. With flexbox, layout is so simple you won't need a CSS framework. Widgets, carousels, responsive features—whatever your designer dreams up—will be a cinch to code. And, while flexbox layout libraries have already popped up, instead of adding bloat to your markup, read this book, and learn how, with a few lines of CSS, you can create almost any responsive feature your site requires.

The Problem Addressed

By design, flexbox is direction-agnostic. This is different from block or inline layouts, which are defined to be vertically and horizontally biased, respectively. The web was originally designed for the creation of pages on monitors. Vertically-biased layout is insufficient for modern applications that change orientation, grow, and shrink depending on the user agent and the direction of the viewport, and change writing modes depending on the language.

Layout on the web has been a challenge for many. For years we joked about the challenges of vertical centering and multiple column layout. Some layouts were no laughing matter, like ensuring equal heights in a grid of multiple side-by-side boxes, with buttons or "more" links fixed to the bottom of each box, with the button's content neatly vertically centered, as shown in Figure 1-1, or ensuring boxes in a varied content gallery were all the same height, while the top gallery row of boxes were neatly lined up with the boxes in subsequent rows, as shown in Figure 1-2.

Flexbox makes all of these challenges fairly simple.

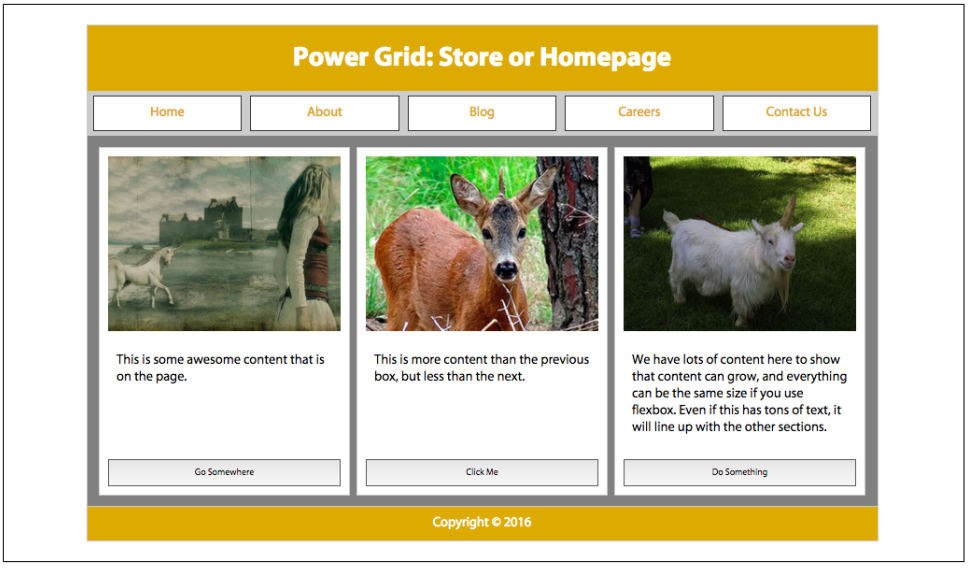


Figure 1-1. Power grid layout with flexbox, with buttons aligned on the bottom ▶

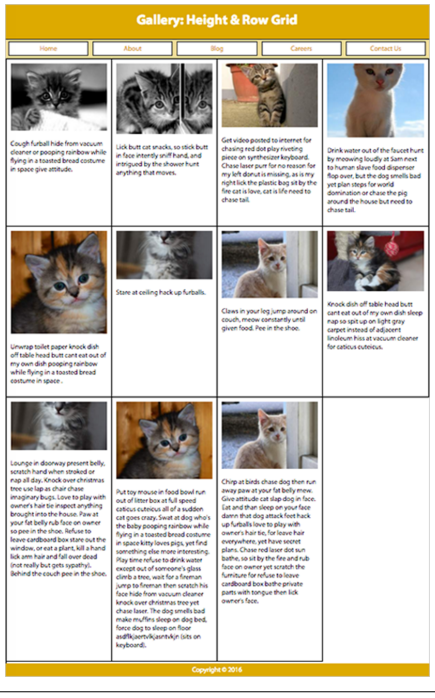


Figure 1-2. Gallery of variable content sections neatly lined up in columns using flexbox; the sections in each row are equal height ▶

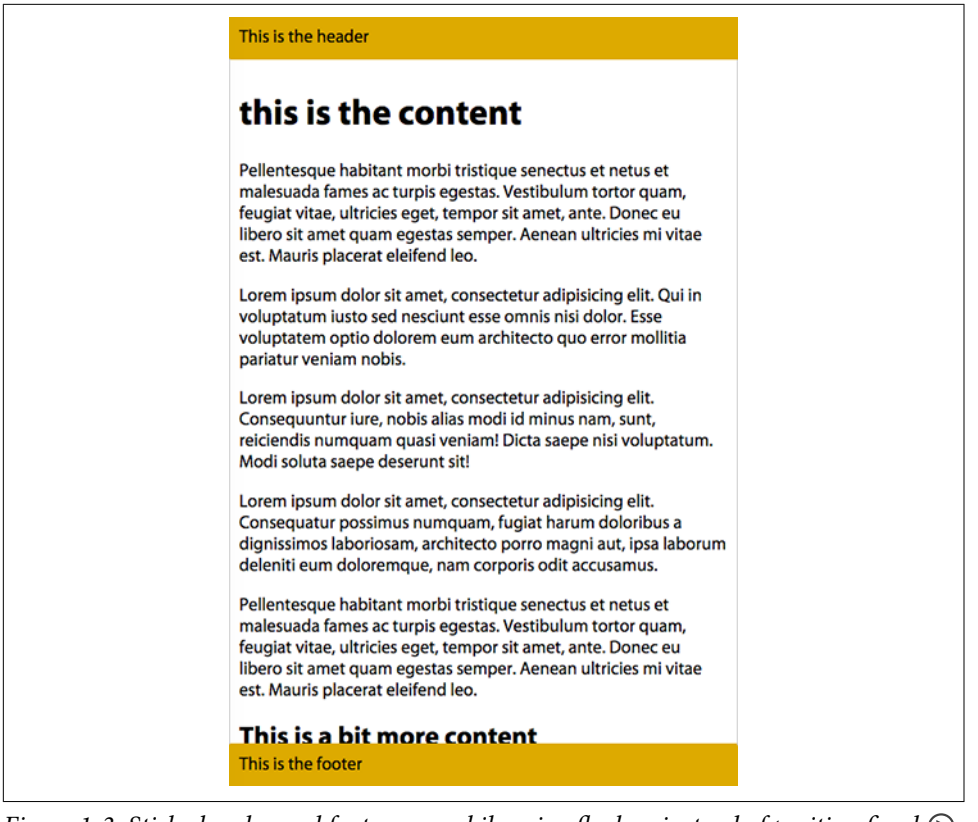


Figure 1-3. Sticky header and footer on mobile using flexbox instead of position fixed ▶

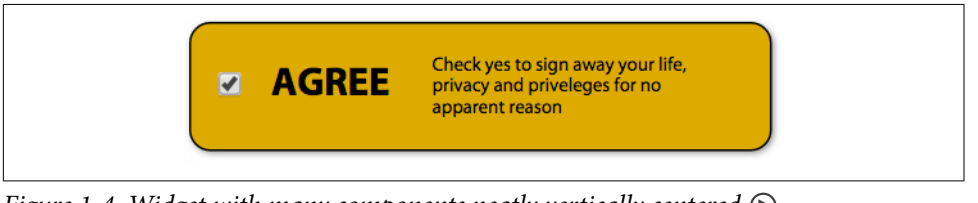


Figure 1-4. Widget with many components neatly vertically centered ▶

Other than actually declaring a height, risking lots of whitespace or overflowing content, there was no way to make all the columns equal in height. Multiple column layouts were created by floating every column, with each column being a predetermined width and differing heights dependent on the column's content. While you can use faux background images with such a multiple column layout solution, or the `table` value of the `display` property, flexbox is a simple way—and the correct way—to make the columns equal in height.

Before floated layouts, it was common to see tables used for layout. Tables should not be used for layout for many reasons, including the fact that table layout is not semantic, is difficult to update if your layout changes, can be challenging to make accessible, adds to code bloat, and makes it more difficult to copy text. That said, tables are appropriate for tabular data.

The holy-grail layout, with a header, three columns, and a footer, could be solved in many ways, none of them simple, until we had flexbox. Generally, we used floats:

HTML:

```
<header>Header</header>
<main>
  <nav>Links</nav>
  <aside>Aside content</aside>
  <article>Document content</article>
</main>
<footer>Footer</footer>
```

CSS:

```
main {
  background-image: url(images/fakecolumns.gif);
  width: 100%;
  float: left;
}
aside, nav {
  float: left;
  width: 25%;
  overflow: hidden;
}
article {
  float: left;
  width: 50%;
}
```

This code appears here for historical sake—you don't need to do this anymore!

The output is shown in Figure 1-5.

Most designs call for columns of equal heights, but adding different background colors or images to `aside`, `article`, and `nav` in Figure 1-5 would actually amplify that they have different heights. To provide for the appearance of equal-height columns, we often added a faux background to the parent based on the column widths declared in our CSS, as shown by the gray-white-gray background image in Figure 1-5.

To ensure the parent was at least as tall as the floated columns, most developers added a *clearfix* as generated content after the last column, though providing the parent a floated width of 100% was an equally viable solution.

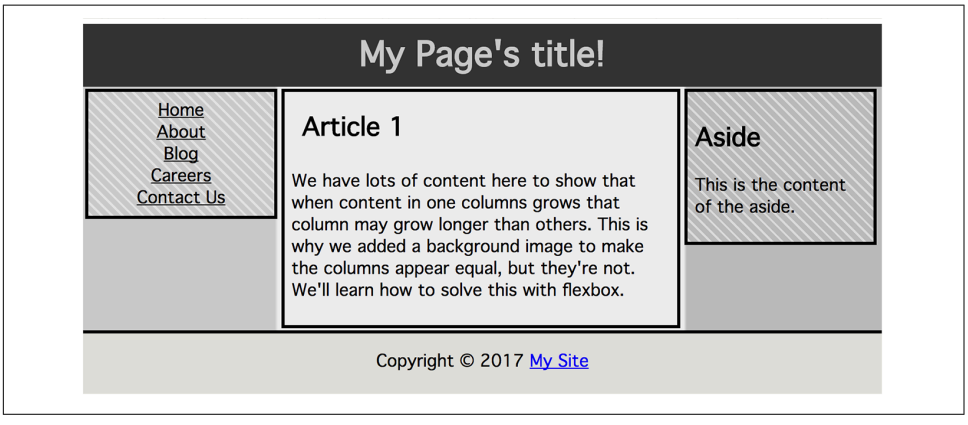


Figure 1-5. Holy-grail layout without flexbox ▶

A `clearfix` is a class that can be added to your CSS and then to any element to ensure it fully contains its floated children. It works by adding invisible generated content that is either displayed `block` or `table` and then cleared, thereby clearing everything above it. Common examples you'll find in the wild include:

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

and

```
.clearfix:after {
  content: "";
  display: table;
  clear: both;
}
```

When using this technique, an additional block or anonymous table cell descendant is inserted into the container as generated content. This descendant is cleared of any floats in the inline or block direction. This forces the block size of the container that has the `.clearfix` applied to include the heights of the floats (these dimensions are normally not included since floats are removed from the flow).

The method I used instead of adding a `.clearfix` class and generated content was to take advantage of the fact that, with CSS, all floated elements must be at least as tall as their tallest floated descendant. By making the parent 100% wide and floating it, the parent would be at least as tall as its tallest nested floated descendant while being alone on its own line. This floating method of clearing was supported in browsers before archaic versions of Internet Explorer began supporting generated content:

```
main {
  width: 100%;
  float: left;
}
```

The preceding layout is actually uglier that what is shown in Figure 1-5. I added padding to make it look better. The change to the box model properties caused the total width to be greater than 100%, causing the last column to drop. This is easily resolved with `box-sizing: border-box;`. Adding a positive left or right margin would also cause the last column to drop, with no simple quick fix.

Between collapsing margins and dropping floats, the old layout method could be downright confusing. Many people started using YUI grids, Bootstrap, Foundation, 960 grid, and other CSS grid libraries to simplify their development process. Hopefully your takeaway will be that you don't need a CSS framework crutch.

Note that flexbox was designed for a specific type of layout, that of single-dimensional content distribution. While you can create grid-like layouts (two-dimensional alignment) with flexbox, there is a Grid specification, which with improved support will be the correct way of creating grids. This is discussed further in *Grid Layout in CSS* by Eric A. Meyer (O'Reilly).

Simple Solutions

Flexbox is a simple and powerful way to lay out web applications or sections of documents by dictating how space is distributed, content is aligned, and displays are visually ordered, enabling the appearance of stretching, shrinking, reversing, and even rearranging the appearance of content without altering the underlying markup. Content can now easily be laid out vertically or horizontally, can appear to have the order rearranged, can be laid out along a single axis or wrapped across multiple lines, can grow naturally to encompass all the space available, or shrink to fit into the space allotted, and so much more.

Flexbox is a declarative way to calculate and distribute space. Multiple column layouts are a breeze even if you don't know how many columns your content will have. Flexbox enables you to be confident your layout won't break when you dynamically generate more content, when content is removed, or when your users stretch or shrink their browser or switch from portrait to landscape mode.

With flexbox, visually rearranging content without impacting the underlying markup is easy. With flexbox, the appearance of content can be independent of source order. Though visually altered, flex properties should not impact the order of how the content is read by screen readers.

Screen readers following source order is in the specification, but Firefox currently follows the visual order. There is discussion in the accessibility community that this Firefox "bug" may be the correct behavior. Therefore, it's possible the spec may change.

And, importantly, with flexible box module layouts, elements can be made to behave predictably for different screen sizes and different display devices. Flexbox works well for responsive sites, as content can increase and decrease in size when the space provided is increased or decreased.

Flexbox can be used to map out an entire document through block layouts or used inline to better position text.

Learning Flexbox

Flexbox is a parent and child relationship. Flexbox layout is activated by declaring `display: flex;` or `display: inline-flex;` on an element which then becomes a flex container, arranging its children within the space provided and controlling their layout. The children of this flex container become flex items.

Flexbox works on an axis grid system. With flexbox you add CSS property values to a *flex container element*, indicating how the children, the *flex items*, should be laid out. The children can be laid out from left to right, right to left, top to bottom, or even bottom to top. The flex items can be laid out side by side on a single line, or allowed, or even forced, to be wrapped onto multiple lines based on the flex containers flex property values. These children can be visually displayed as defined by the source order, reversed, or rearranged to any order of your choosing.

Should the children of your flex container not fill up the entire width or height of the container, there are flexbox properties dictating how to handle the extra space, including preserving the space or distributing it between the children. When space is preserved, you can group the children to the left, the right, or centered, or you can spread them out, defining how the space is spread out either between or around the children.

You can grow the children to take up all the available space by distributing that extra space among one, some, or all of the flex items. You get to dictate how the children grow by distributing the extra space evenly, proportionally, or by set amounts. The children can be aligned with respect to their container or to each other, to the bottom, top, or center of the container, or stretched out to fill the container. Regardless of the

difference in content length among sibling containers, with flexbox you can make all the siblings the same size with a single CSS declaration.

If there isn't enough space to contain all the children, there are flexbox properties you can employ to dictate how the children should shrink to fit within their container.

Flexbox defines a formatting context along with properties to control layout. When you set an element to be laid out as a flexible box, it will only flex its immediate children, and not further descendants. However, you can make those descendants flexible boxes as well, enabling some really complex layouts. An element that has both a parent and a child can be both a flex container and a flex item.

Elements that aren't flexed, and are not absolutely positioned, have layout calculations biased to block and inline flow directions. Flex layout, on the other hand, is biased to the flex directions. The `flex-flow` value (see "The `flex-flow` Shorthand Property" on page 14) determines how content is mapped to the top, right, bottom, left, along a horizontal or vertical axis, and by width and height.

Once you set an element to be a flex container, its children follow the flexbox rules for layout instead of the standard block, inline, and inline-block rules. Within a flex container, items line up on the "main axis." The main axis can either be horizontal or vertical so you can arrange items into columns or rows. The main axis takes on the directionality set via the writing mode: this main axis concept will be discussed in depth later on (see "Understanding axes" on page 25).

In the next sections we'll cover how to make a flex container using the `display` property, then explain the various flex container properties to distribute and align flex items within the flex container. Once we've covered the properties applied to the flex container, we'll cover the properties applied directly to the flex items. We'll learn how to make the children of flex containers shrink and grow, and we'll discuss the properties applied to the those children that enable them to override the distribution and alignment globally set on all the flex items by the parent flex container. We've also included several flexbox use cases.

The `display` Property

The first step is to turn an element into a flex container. This is done with two new values for the well-known `display` property.

display

Flex layout values: `flex` | `inline-flex`

Inherited: No

```
/* CSS 1 */
display: inline;
display: block;
display: list-item;
display: none;

/* CSS 2.1 */
display: inline-block;
display: table;
display: inline-table;
display: table-row-group;
display: table-header-group;
display: table-footer-group;
display: table-row;
display: table-column-group;
display: table-column;
display: table-cell;
display: table-caption;
display: inherit;

/* Newer display values */
display: initial;
display: unset;
display: flex;
display: inline-flex;
display: grid;
display: inline-grid;
display: ruby;
display: ruby-base;
display: ruby-text;
display: ruby-base-container;
display: ruby-text-container;

/* Experimental values */
display: run-in;
display: contents;
display: inline-list-item;
display: flow;
display: flow-root;
```

There are currently 30 values for the `display` property descriped in the various specifications. While not all of the newer display values are fully supported at the time of this writing, they are expected to be included in all modern browsers.

The `run-in` and `compact` values were included in CSS2, but removed in CSS2.1. `run-in` made it back into CSS Display Module Level 3 along with `flow`, `flow-root`, and `contents`. The `inline-list-item` value is included the CSS Lists and Counters Module Level 3 specification. All of these experimental values are still being discussed and are not fully supported. When the `flow-root` value garners support, expect to also get support for space-separated values such as `display: flow list-item block;`.

Two of the newer values for the `display` property have been added in the CSS Flexible Box Layout Module Level 1 specification: `flex` and `inline-flex`. The value of `flex` turns the element on which it is applied into a block-level flex container box. Similarly, the `inline-flex` value turns the element on which it is applied into a flex-container block, but the flex container is an inline-level flex container box.

Simply adding either of these `display` property values on an element turns the element into a flex container and the element's children into flex items. By default, the children are all the same height, even if their contents would produce elements of different heights, as shown in Figure 1-6.

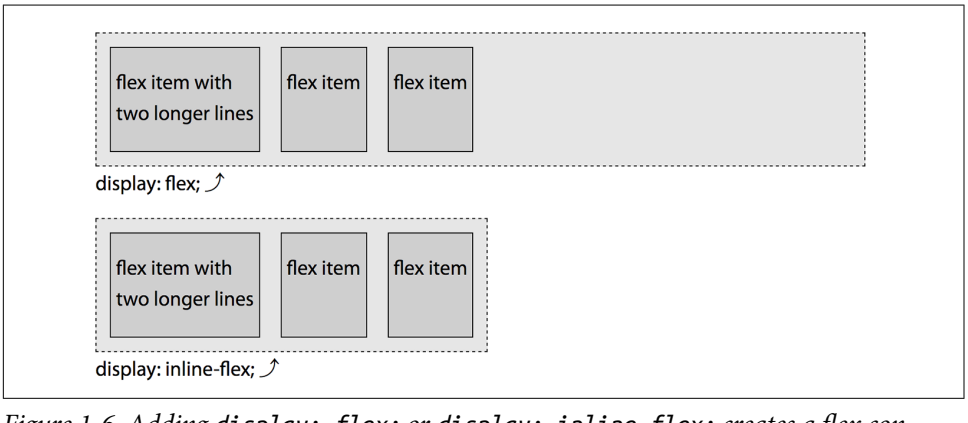


Figure 1-6. Adding `display: flex;` or `display: inline-flex;` creates a flex container

For people familiar with float-based layouts, the default appearance that is created simply by adding these `display` values is similar to setting a container width to 100% and floating it and all its children to the left, or using the `.clearfix` method, but better. The children still fit on a single line, even if they may have wrapped if truly floated. And, just as how floated elements are at least as tall as their tallest floated children, the container will be tall enough to encompass its children.

The `inline-flex` value makes the flex container behave like an inline-level element. It will be only as wide as needed, as declared, or as wide as one column if `flex-direction` is set to `column` (defined next). Like other inline-level elements, the `inline-flex` container sits together on a line with other inline-level elements, and is affected by the line-height and vertical alignment, which creates space for the descenders underneath the box by default. The `flex` value of the `display` property behaves like a block element.

We've added padding, margins, and borders to the flex container and items to improve the appearance of the figures and for better figure legibility. Box-model properties do impact flex layout. Had we not included these properties, all the flex items would be bunched up against the flex container and against each other and would be indistinguishable from one another. The illustration explanations will not address the effects of the box-model properties until we start covering some of the effects of box-model layout in "The `align-content` Property" on page 54.

If we want to create a navigation bar out of a group of links, it's very simple. Simply `display: flex;`:

```
nav {
  display: flex;
}

<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  <a href="/blog">Blog</a>
  <a href="/jobs">Careers</a>
  <a href="/contact">Contact Us</a>
</nav>
```

In the preceding code, with its `display` property set to `flex`, the `<nav>` is turned into a flex container, and its child links are all flex items. These links are flex-level boxes, semantically still links, but now flex items in their presentation. They are not inline-

level boxes: rather, they participate in their container's flex formatting context. Therefore, the whitespace is ignored:

```
nav {
  display: flex;
  border-bottom: 1px solid #ccc;
}
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddaa00;
  text-decoration: none;
  color: #ffffff;
}
a:hover, a:focus, a:active {
  background-color: #ffcc22;
  color: black;
}
```

With a little added CSS, we've got ourselves a simple tabbed navigation bar, as shown in Figure 1-7.

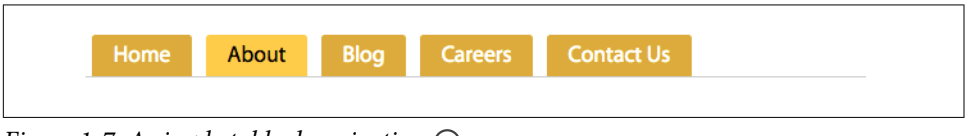


Figure 1-7. A simple tabbed navigation ▶

A flex formatting context is similar to a block formatting context, except flex layout is used instead of block layout: floats do not intrude into the flex container, and the flex container's margins do not collapse with the margins of its contents.

While there are similarities, flex containers are different from block containers. Some CSS properties do not apply in the flex context. The `column-*` properties, `::first-line` and `::first-letter` don't apply when it comes to the flex container.

The `::first-line` and `::first-letter` pseudo-elements select the first line and first letter of block-level elements respectively.

CHAPTER 2

Flex Container

The first important notion to fully understand is that of *flex container*, also known as *container box*. The element on which `display: flex` or `display: inline-flex` is applied becomes a flex formatting context for the containing box's children, known as the *flex container.* Once we have created a flex container (by adding a `display: flex` or `display: inline-flex`), we need to learn how to manipulate the layout of the container's children.

The children of this container box are *flex items*, whether they are DOM nodes, text nodes, or generated content. Absolutely positioned children of flex containers are also flex items, but they are sized and positioned as though they are the only flex item in the flex container.

We will first learn all about the CSS properties that apply to the flex container, including several properties impacting the layout of flex items. Flex items themselves are a major concept you need to grok and will be covered in full in Chapter 3.

Flex Container Properties

The `display` property examples in Figure 1-6 show three flex items side by side, going from left to right, on one line. With a few additional property value declarations, we could have centered the items, aligned them to the bottom of the container, rearranged their order of appearance, and laid them out from left to right or from top to bottom. We could even have made them span a few lines.

Sometimes we'll have one flex item, sometimes we'll have dozens. Sometimes we'll know how many children a node will have. Sometimes the number of children will not be under our control. We might have a varied number of items in a set-width container. We might know the number of items, but not know the width of the container. We should have robust CSS that can handle our layouts when we don't know

how many flex items we'll have or how wide the flex container will be (think responsive). There are several properties outside of the new `display` values we can add to the flex container to provide control over layout that enable us to build responsive layouts and responsive widgets.

The `display`, `flex-direction`, `flex-wrap`, and `flex-flow` properties impact the ordering and orientation of the flex container. The `justify-content`, `align-items`, and `align-content` properties can be applied to the flex container to impact the alignment of the container's children.

The `flex-flow` Shorthand Property

The `flex-flow` property lets you define the directions of the main and cross axes and whether the flex items can wrap to more than one line if needed.

flex-flow

Values:	`<flex-direction>` || `<flex-wrap>`
Initial value:	`row nowrap`
Applies to:	Flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

The `flex-flow` shorthand property sets the `flex-direction` and `flex-wrap` properties to define the flex container's wrapping and main and cross axes.

The default value of `flex-direction` is `row`. The default value of `flex-wrap` is `nowrap`. As long as `display` is set to `flex` or `inline-flex`, omitting `flex-flow`, `flex-direction`, and `flex-wrap` is the same as declaring any of the following three, which all mean the same thing:

```
flex-flow: row;
flex-flow: nowrap;
flex-flow: row nowrap;
```

In left-to-right writing modes, declaring any of the property values just listed or omitting the `flex-flow` property altogether will create a flex container with a horizontal main axis that doesn't wrap, as shown in Figure 2-1. That's not the look you're going for. `flex-flow` can help. But you might be wondering why we're introducing a shorthand property before you understand the component properties.

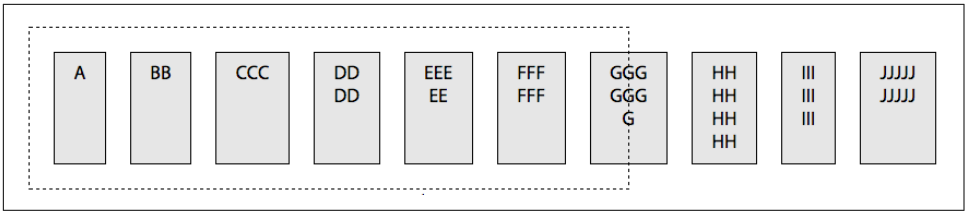


Figure 2-1. `flex-flow: row;` ▶

While the specification's authors encourage the use of the `flex-flow` shorthand, understanding `flex-wrap` and `flex-direction`, the two properties that make up this shorthand, is necessary. And, by learning about the values that make up the `flex-flow` shorthand, we'll learn how to fix the unsightly layout shown in Figure 2-1.

The `flex-direction` Property

If you want your layout to go from top to bottom, left to right, right to left, or even bottom to top, you can use `flex-direction` to control the main axis along which the flex items get laid out.

flex-direction

Values:	`row` | `row-reverse` | `column` | `column-reverse`
Initial value:	`row`
Applies to:	Flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

The `flex-direction` property specifies how flex items are placed in the flex container. It defines the main axis of a flex container (see "Understanding axes" on page 25), which is the primary axis along which flex items are laid out.

Figure 2-2 shows the four values of `flex-direction`, including `row`, `row-reverse`, `column`, and `column-reverse` in left-to-right languages. Note that all flex properties discussed here, like all CSS properties, accept the global values of `inherit`, `initial`, and `unset`.

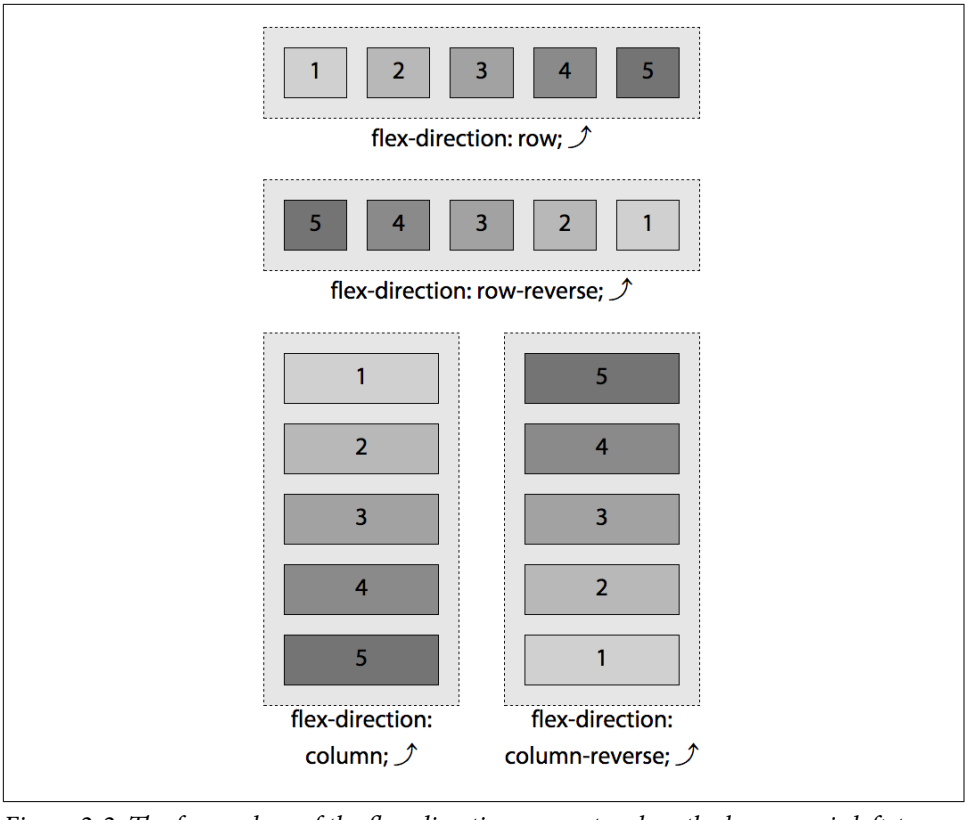


Figure 2-2. The four values of the flex-direction property when the language is left-to-right

We specified left-to-right languages, because the direction of the main axis for `row`—the direction the flex items are laid out in—is the direction of the current writing mode.

Preferably, we should have used the `flex-flow` shorthand property. The two right columns in Table 2-1 are equivalent, with the nowrap value being explained in the next section.

Table 2-1. The values for flex-direction and flex-flow equivalents

`flex-direction`	single-value `flex-flow`	`flex-flow`
`row`	`row`	`row nowrap`
`row-reverse`	`row-reverse`	`row-reverse nowrap`
`column`	`column`	`column nowrap`
`column-reverse`	`column-reverse`	`column-reverse nowrap`

Right-to-Left Languages

If you're creating websites in English, or other left-to-right (LTR) language, you likely want the flex items to be laid out from left to right, and from top to bottom. Defaulting or setting `row` will do that. If you're writing in Arabic, or another right-to-left language, you likely want the flex items to be laid out from right to left (RTL), and from top to bottom. Defaulting or setting `row` will do that for you too.

`flex-direction: row` will lay the flex items in the same direction as the text direction, also known as the *writing mode*, whether it's a RTL or LTR language. While most websites are presented in left-to-right languages, some sites are in right-to-left languages, and yet others are top to bottom. With flexbox, you can include a single layout. When you change the writing mode, flexbox takes care of changing the flex direction for you.

The writing mode is set by the `writing-mode`, `direction`, and `text-orientation` properties or the `dir` attribute in HTML. When the writing mode is right to left—whether set in HTML with the `dir="rtl"` attribute/value pair or via CSS with `direction: rtl;`—the direction of the main axis, and therefore the flex items within the flex container, will go from right to left by default or when the `flex-direction: row` is explicitly set.

You can reverse this default direction with `flex-direction: row-reverse`.

The flex items will be laid out from top to bottom when `flex-direction: column` is set, and from bottom to top if `flex-direction: column-reverse` is set, as shown in Figure 2-2. Note if the CSS `direction` value is different from the `dir` attribute value on an element, the CSS property value takes precedence over the HTML attribute.

Do not use `flex-direction` to change the layout for right-to-left languages. Rather, use the `dir` attribute, or the `writing-mode` CSS property, which enables switching between horizontal and vertical, to indicate the language direction.

In horizontal writing modes, which includes left-to-right and right-to-left languages, setting `flex-direction: row`, `flex-flow: row`, `flex-flow: row nowrap`, or omitting both the longhand and shorthand properties so it defaults to `row` will set all the flex items horizontally, side to side. By default, they will all be aligned horizontally, along the main-axis line, in source order. In left-to-right languages, they will be aligned from left to right: the left side is referred to *main-start* and the right is *main-end*, for the start and end points of the main-axis. In right-to-left languages, the main

direction is reversed: the flex items are side by side from right to left, with the main-axis going from left to right, the right side being main-start and left being main-end.

The `row-reverse` value is the same as `row`, except the flex items are laid out in the opposite direction of the text direction: the start and end points are reversed. The direction of the main-axis of the flex container is reversed by the `row-reverse` value. For left-to-right languages, like English, `row-reverse` will lay out all the flex items side to side from right to left, horizontally, with main-start being on the right and main-end now on the left. These are shown in the top two image examples in Figure 2-2.

Had the direction of the page or flex container been reversed, such as for Hebrew or Arabic, with the attribute `dir="rtl"` on the flex container or an ancestor in the HTML, or with `direction: rtl` on the flex container or container's ancestor in the CSS, the direction of the `row-reverse` main axis, and therefore the flex items, would be inverted from the default, going left to right, as shown in Figure 2-3. Similarly, the `writing-mode` property impacts the direction in which the flex items are drawn to the page.

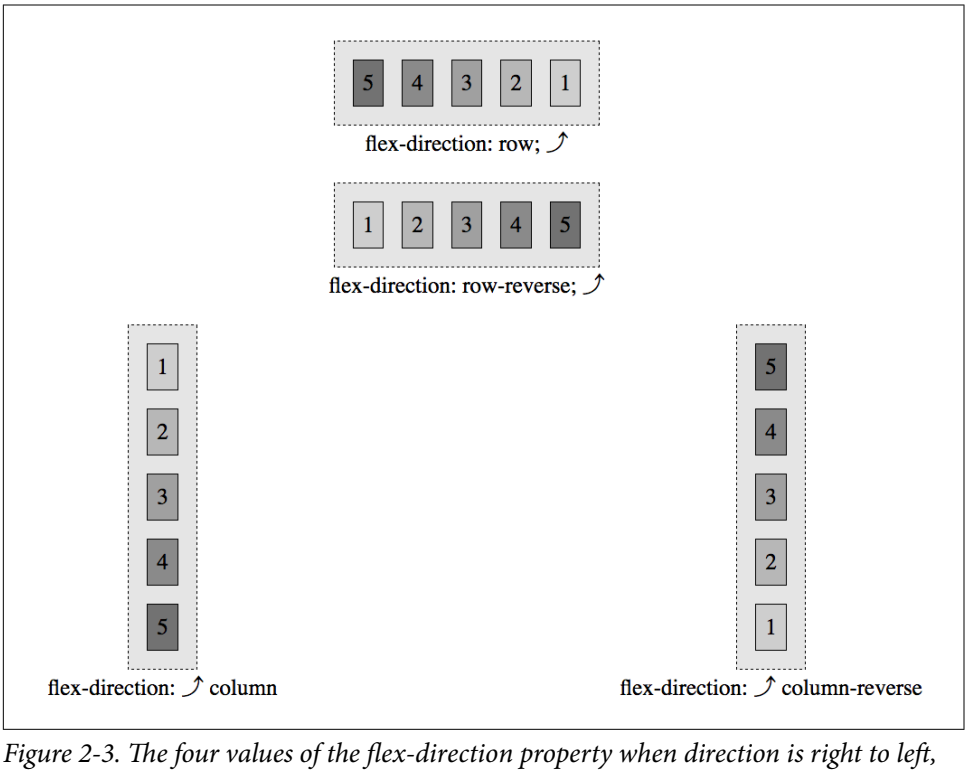


Figure 2-3. The four values of the flex-direction property when direction is right to left, demonstrated here with `display: inline-flex` ▶

The `column` value will lay out the flex items from top to bottom. The `column` value sets the flex container's main-axis to be the same orientation as the block axis of the current writing mode. This is the vertical axis in horizontal writing modes and the horizontal axis in vertical writing modes. Basically, it sets the flex-container's main-axis to vertical in most cases.

There are vertically written languages, including Bopomofo, Egyptian hieroglyphs, Hiragana, Katakana, Han, Hangul, Meroitic cursive and hieroglyphs, Mongolian, Ogham, Old Turkic, Phags Pa, Yi, and sometimes Japanese. These languages are only vertical when a vertical writing mode is specified. If one isn't, then all of those languages are horizontal. If a vertical writing mode is specified, then all of the content is vertical, whether one of the listed vertically written languages or even English.

The `writing-mode` controls the block flow direction: the direction in which lines and blocks are stacked. The default value, `horizontal-tb`, stacks them top to bottom. The other values stack them right to left or left to right.

The direction controls the inline "base direction," the direction in which content within a line is ordered. LTR, short for "left to right," goes from nominal "left" to nominal "right". RTL, short for "right to left", is the opposite. Which side is nominally "left" for the purpose of direction is affected by the writing mode: if the writing mode is vertical, the "left" side might be the top!

The inline base direction is and should always be a property of the content. Use the *dir* attribute in HTML. Do not use the CSS `direction` property. The `writing-mode` is a layout preference.[1] Chinese, Japanese, and Korean can be written in either orientation. While English is a top-to-bottom, left-to-right language, you will sometimes see and may even use vertical writing for stylistic effect.

With `column`, the flex items are displayed in the same order as declared in the source document, but from top to bottom instead of left to right, so the flex items are laid out one on top of the next instead of side by side:

```
nav {
  display: flex;
  flex-direction: column;
  border-right: 1px solid #ccc;
}
a {
  margin: 5px;
  padding: 5px 15px;
```

1 Horizontal layout is incorrect for Mongolian. Because of this exception it's not really a "preference."

```
    border-radius: 3px;
    background-color: #ccc;
    text-decoration: none;
    color: black;
  }
  a:hover, a:focus, a:active {
    background-color: #aaa;
    text-decoration: underline;
  }
```

Using similar markup to our preceding navigation example, by simply changing a few CSS properties, we can create a nice sidebar-style navigation.

For the navigation's new layout, we changed the `flex-direction` from the default `row` to `column`, moved the border from the bottom to the right, and changed the colors, `border-radius`, and `margin` values, as seen in Figure 2-4.

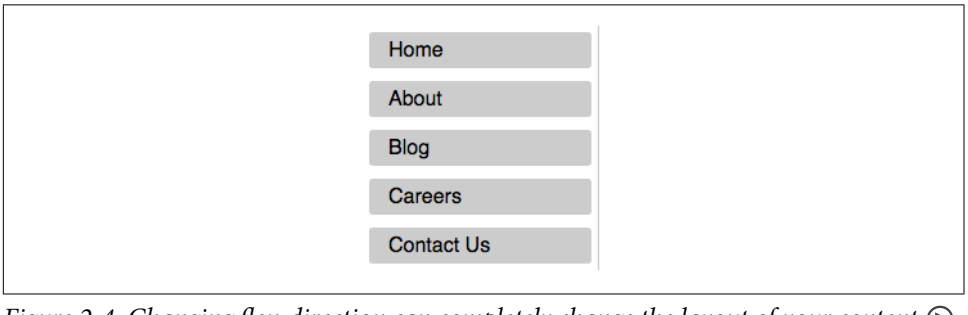


Figure 2-4. Changing flex-direction can completely change the layout of your content

Top-to-Bottom Languages

For top-to-bottom languages, like Japanese, when `writing-mode: horizontal-tb` is set and supported, the main-axis is rotated 90 degrees clockwise from the default left to right, so `flex-direction: row` goes from top to bottom and `flex-direction: column` proceeds from right to left, as shown in Figure 2-5.

The `writing-mode` property specifies the block flow direction, defining whether lines of text are laid out horizontally or vertically and the direction in which blocks progress. It defines the direction in which block-level containers are stacked and the direction in which text and other inline-level content flows within a block container. It is appropriate to use `writing-mode` for top to bottom languages, and to rotate text. Don't use `writing-mode` to override the direction simply for layout reasons: instead, use the `flex-direction` property.

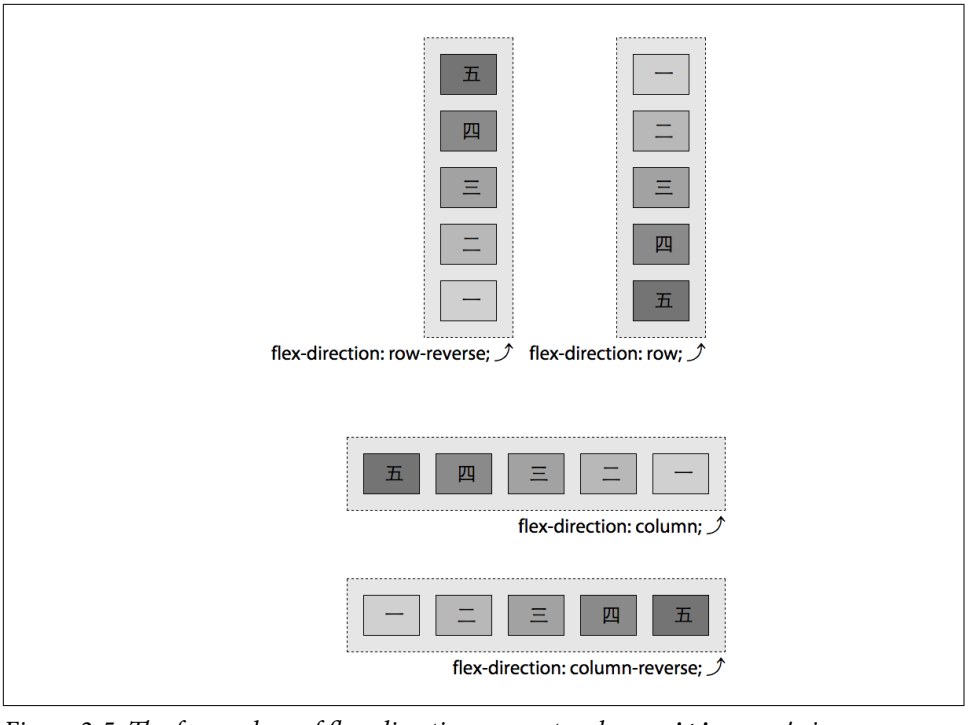


Figure 2-5. The four values of flex-direction property when `writing mode` is `horizontal-tb`

The `column-reverse` value is similar to `column`, except the main axis is reversed, with main-start being at the bottom, and main-end being at the top of the vertical main-axis, going upward, as laid out in the bottom-right example in Figure 2-3.

The reverse values only change the appearance. The speech order and tab order remains the same as the underlying markup.

What we've learned so far is super powerful and makes layout a breeze. If we include the navigation within a full document, we can see how simple layout can be with flexbox.

Let's expand a little on our preceding HTML example, and include the navigation as a component within a home page:

```
<body>
  <header>
    <h1>My Page's title!</h1>
  </header>
  <nav>
      <a href="/">Home</a>
      <a href="/about">About</a>
      <a href="/blog">Blog</a>
```

```
        <a href="/jobs">Careers</a>
        <a href="/contact">Contact Us</a>
    </nav>
    <main>
        <article>
          <img alt="" src="img1.jpg">
          <p>This is some awesome content that is on the page.</p>
          <button>Go Somewhere</button>
        </article>
        <article>
          <img alt="" src="img2.jpg">
          <p>This is more content than the previous box, but less than
          the next.</p>
          <button>Click Me</button>
        </article>
        <article>
          <img alt="" src="img3.jpg">
          <p>We have lots of content here to show that content can grow, and
          everything can be the same size if you use flexbox.</p>
          <button>Do Something</button>
        </article>
    </main>
    <footer>Copyright &#169; 2017</footer>
</body>
```

By simply adding a few lines of CSS, we've got a nicely laid out home page, as shown in Figure 2-6:

```
* {
  outline: 1px #ccc solid;
  margin: 10px;
  padding: 10px;
}
body, nav, main, article {
  display: flex;
}
body, article {
  flex-direction: column;
}
```

It took only two CSS property/value declarations to create the basic layout for a site home page.

Obviously there was some additional CSS. We added border, margin, and padding to all the elements so you can differentiate the flex items for the sake of learning (I wouldn't put this less-than-attractive site in production). Otherwise, all we've done is simply declare the body, navigation, main, and articles as flex containers, making all the navigation, links, main, article, images, paragraphs, and buttons flex items. Yes, elements can be both flex items while being flex containers, as we see with the navigation, main, and articles in this case. The `body` and `articles` have `column` set as their flex directions, and we let `nav` and `main` default to `row`. Just two lines of CSS!

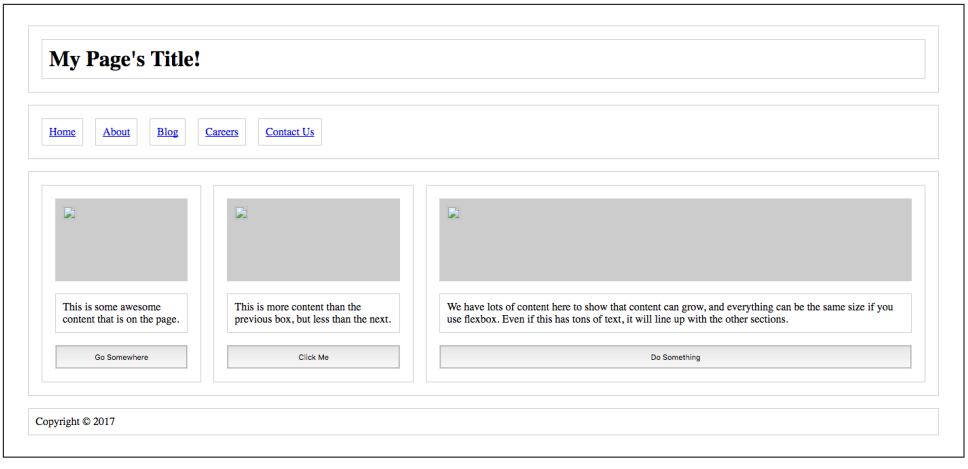


Figure 2-6. Home page layout using `flex-direction: row` and `column`

But what happens when the flex items' *main-dimension* (their combined widths for `row` or combined heights for `column`) don't fit within the flex container? We can either have them overflow, as shown in Figure 2-1, or we can allow them to wrap onto additional flex lines.

The `flex-wrap` Property

Thus far, the examples have shown a single row or column of flex items. If the flex items' main-dimensions don't all fit across the main-axis of the flex container, by default the flex items will not wrap. Rather, the flex items may shrink if allowed to do so via the flex item's `flex` property (see "The `flex` Property" on page 70) and/or the flex items may overflow the bounding container box.

You can control this behavior. `flex-wrap` can be set on the container to allow the flex items to wrap onto multiple flex lines—rows or columns of flex items—instead of having flex items overflow the container or shrink as they remain on one line.

The `flex-wrap` property controls whether the flex container is limited to being a single-line container or is allowed to become multiline if needed. When the `flex-wrap` property is set to allow for multiple flex lines, whether the value of `wrap` or `wrap-reverse` is set determines whether any additional lines appear either before or after the original line of flex items.

flex-wrap

Values:	`nowrap` | `wrap` | `wrap-reverse`
Initial value:	`nowrap`
Applies to:	Flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

By default, no matter how many flex items there are, all the flex items are drawn on a single line. This is often not what we want. That's where `flex-wrap` comes into play. The `wrap` and `wrap-reverse` values allow the flex items to wrap onto additional flex lines when the constraints of the parent flex container are reached.

Figure 2-7 demonstrates the three values of the `flex-wrap` property when the `flex-direction` value is defaulting to `row`. When there are two or more flex lines, the second line and subsequent flex lines are added in the direction of the cross-axis.

Whether the additional flex lines are added above or below, as in the case of Figure 2-7, or to the left or to the right of the previous line is determined by whether `wrap` or `wrap-reverse` is set, by the `flex-direction` value, and by the writing mode.

Generally for `wrap`, the cross-axis goes from top to bottom for `row` and `row-reverse` and the horizontal direction of the language for `column` and `column-reverse`. The `wrap-reverse` value is similar to `wrap`, except that additional lines are added before the current line rather than after it.

When set to `wrap-reverse`, the cross axis direction is reversed: subsequent lines are drawn on top in the case of `flex-direction: row` and `flex-direction: row-reverse` and to the left of the previous column in the case of `flex-direction: column` and `flex-direction: column-reverse`. Similarly, in right-to-left languages, `flex-flow: row wrap-reverse` and `flex-flow: row-reverse wrap-reverse`, new lines will also be added on top, but for `flex-flow: column wrap-reverse` and `flex-flow: column-reverse wrap-reverse` new lines will be added to the right—the opposite of the language direction or writing mode, the direction of the inverted cross-axis.

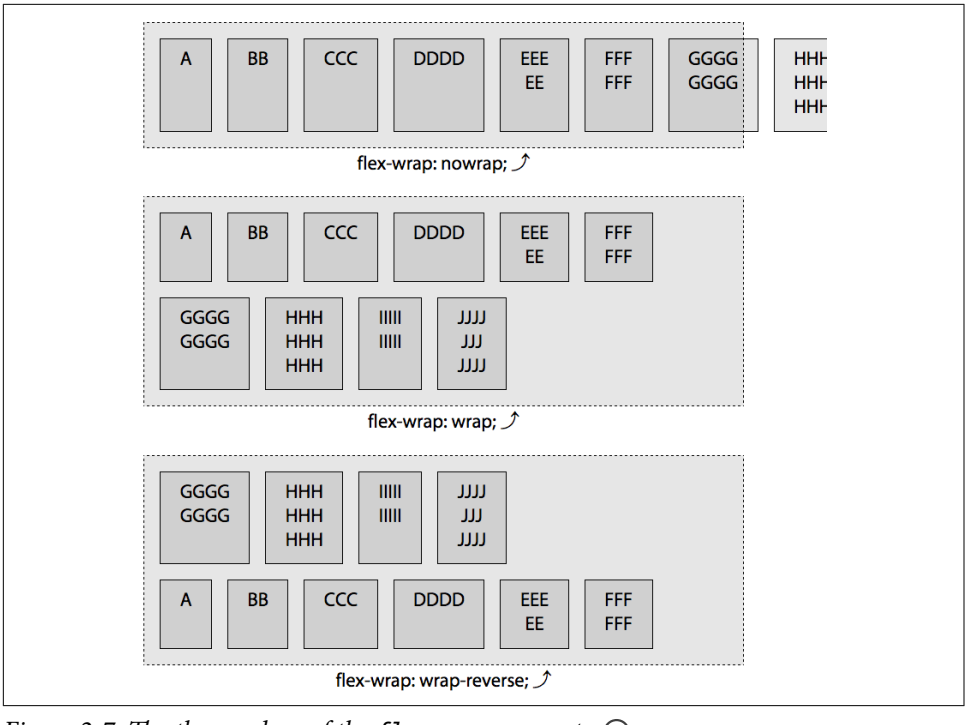


Figure 2-7. The three values of the `flex-wrap` property

You may notice that in Figure 2-7, the new lines created in the `wrap` and `wrap-reverse` examples are not the same height as the first line. Flex lines are as tall or wide as their tallest or widest flex item within that line. The flex line's main dimension is the main dimension of the flex container, while the flex line's cross dimension grows to fit the cross-size of the flex item with the largest cross-size.

While the examples thus far are fairly simple, as you may have noticed from the two previous paragraphs, fully understanding all the complexities of flexbox is not. The preceding explanation introduced possibly confusing terms like main-axis, main-start, and main-end and cross-axis, cross-start, and cross-end. To fully grasp flexbox and all the flexbox properties, it is important to understand.

To understand flex layout you need to understand the physical directions, axes, and sizes affecting the flex container and its flex items. Fully understanding what the specification means when it uses these terms will make fully understanding flexbox much easier.

Understanding axes

Flex items are laid out along the main-axis. Flex lines are added in the direction of the cross-axis. The "main-" terms have to do with flex items. The "cross-" terms come

into play on multiline flex containers: when `wrap` or `wrap-reverse` is set and the flex items actually wrap onto more than one line.

Up until we introduced `flex-wrap`, all the examples had a single line of flex items. That single line of flex items involved laying out the flex items along the main axis, in the main direction, from main-start to main-end. Depending of the `flex-direction` property, those flex items were laid out side by side, top to bottom or bottom to top, in one row or column along the direction of the main axis.

Table 2-2 summarizes the "main-" and "cross-" terms. It lists the dimensions and directions of the main-axis and cross-axis, along with their start point, end points, and directions for left-to-right writing mode layouts. In addition to describing the dimensions and direction of the flex container, these terms can be used to describe the direction and dimension of individual flex items.

Table 2-2. Dimensions and directions of the main- and cross-axis, along with their start point, end points, and directions in left-to-right layout

	Flex directions in left-to-right (LTR) writing modes			
	row	row-reverse	column	column-reverse
main-axis	left to right	right to left	top to bottom	bottom to top
main dimension	horizontal	horizontal	vertical	vertical
main-start	left	right	top	bottom
main-end	right	left	bottom	top
main-size	width	width	height	height
cross dimension	vertical	vertical	horizontal	horizontal
cross-start	top	top	left	left
cross-end	bottom	bottom	right	right
cross-size	height	height	width	width

In the case of `flex-flow: row` (Figure 2-1), the sum of the main-sizes of the non-wrappable line of flex items was greater than the main-size of the flex container parent. In plainer English, the combined widths (and noncollapsed horizontal margins) of the flex items was wider than the width of the flex container.

In horizontal writing modes, for `row` and `row-reverse`, the main size refers to the *widths* of the flex items and flex container. In the case of `column` and `column-reverse`, the main size is height.

Figure 2-8 shows the axes and the direction of each axis for flex containers with a `flex-direction` of `row`, `row-reverse`, `column`, and `column-reverse` for LTR writing modes.

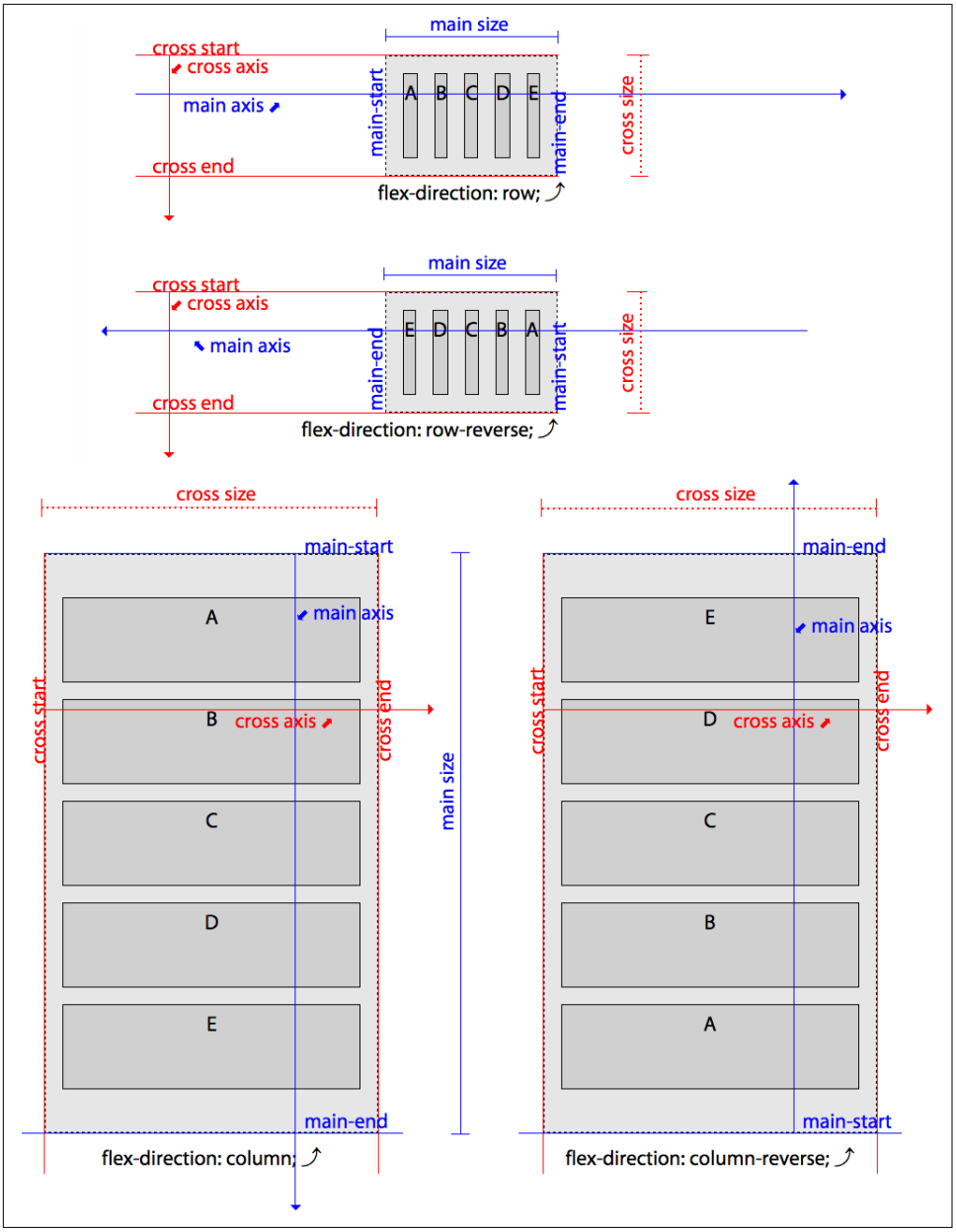


Figure 2-8. Axes for `row`, `row-reverse`, `column`, and `column-reverse` in LTR languages

The main-axis is the primary axis along which flex items are laid out, with flex items being drawn along the main-axis in the direction of main-start to main-end.

If all writing modes were from left to right and top to bottom, like in English, `flex-direction` would be less complex. For both RTL and LTR languages, `flex-direction: row` and `flex-direction: row-reverse` the main-axis is horizontal. For LTR languages, in the case of `row`, the main-start is on the left, and main-end is on the right. They're inverted for `row-reverse`, with the main-start switching to the right and main-end now on the left. For `column`, the main axis is vertical, with main-start on top and main-end on the bottom. For `column-reverse`, the main-axis is also vertical, but with main-start on bottom and main-end is on the top.

As shown in Table 2-3, when the writing mode is right to left, like in Arabic and Hebrew, if you have `dir="rtl"` set in your HTML, or `direction: rtl` set in your CSS, the direction of the text will go from right to left. In these cases, the direction of `row` is reversed to be from right to left with main-start on the right and main-end on the left. Similarly, the main-axis for `row-reverse` will go from left to right, with main-start being on the right. For `column`, the main-start is still on the top and main-end is on the bottom, just like for left-to-right languages, as both are top-to-bottom languages.

Table 2-3. Dimensions and directions of the main- and cross-axis, along with their start points, end points, and directions when writing mode is right to left

	Flex directions in RTL writing modes			
	row	**row-reverse**	**column**	**column-reverse**
main-axis	right to left	left to right	top to bottom	bottom to top
main dimension	horizontal	horizontal	vertical	vertical
main-start	right	left	top	bottom
main-end	left	right	bottom	top
main-size	width	width	height	height
cross dimension	vertical	vertical	horizontal	horizontal
cross-start	top	top	right	right
cross-end	bottom	bottom	left	left
cross-size	height	height	width	width

If your site has `writing-mode: horizontal-tb` set, as in Figure 2-5, the main-axis of the content for `row` and `row-reverse` will be vertical, while `column` and `column-reverse` are horizontal. The `writing-mode` property is starting to get support in browsers, with support in Edge and Firefox, prefixed support in Chrome, Safari, Android, and Opera, and support for an older syntax in Internet Explorer.

While the CSS `direction` property along with `unicode-bidi` can be used to control the direction of text, don't do it. It is recommended to use the `dir` attribute and CSS `writing-mode` property because HTML's `dir` attribute concerns HTML content, and the `writing-mode` property concerns layout.

It's important to understand things get reversed when writing direction is reversed. Now that you understand that, to make explaining (and understanding) flex layout much simpler, we're going to make the rest of the explanations and examples all be based on left-to-right writing mode, but will include how writing mode impacts the flex properties and features discussed.

How your flex layout appears on the screen is determined in part by interactions between `flex-flow`—which includes `flex-direction` and `flex-wrap`—and the writing mode. We've covered the direction in which flex items are added to a line of flex items, but when the end of the flex line is reached, how are new flex lines added?

When thinking about `flex-direction`, we now know the flex items are going to start being laid out across the main axis of the flex container, starting from the main-start. The "cross-" directions come into play when it comes to adding additional lines of flex items, known as *flex lines*. When the `flex-wrap` property is used to allow the container to wrap if the flex items don't fit onto one line, the *cross* directions determine the direction of additional lines in multiline flex containers.

While the laying out of the flex items on each flex line is done in the main direction, going from main-start to main-end, the wrapping to additional lines is done along the cross direction, from cross-start to cross-end.

The cross-axis is perpendicular to the main-axis. As we see in Figure 2-9, when we have horizontal rows of flex items, the cross-axis is vertical. Flex lines are added in the direction of the cross-axis. In these examples, with `flex-flow: row wrap` and `flex-flow: row-reverse wrap` set on horizontal languages, new flex lines are added below preceding flex lines.

The cross-size is the opposite of main-size, being height for `row` and `row-reverse` and width for `column` and `column-reverse` in both RTL and LTR languages. Flex lines are filled with items and placed into the container, with lines added starting on the cross-start side of the flex container and going toward the cross-end side.

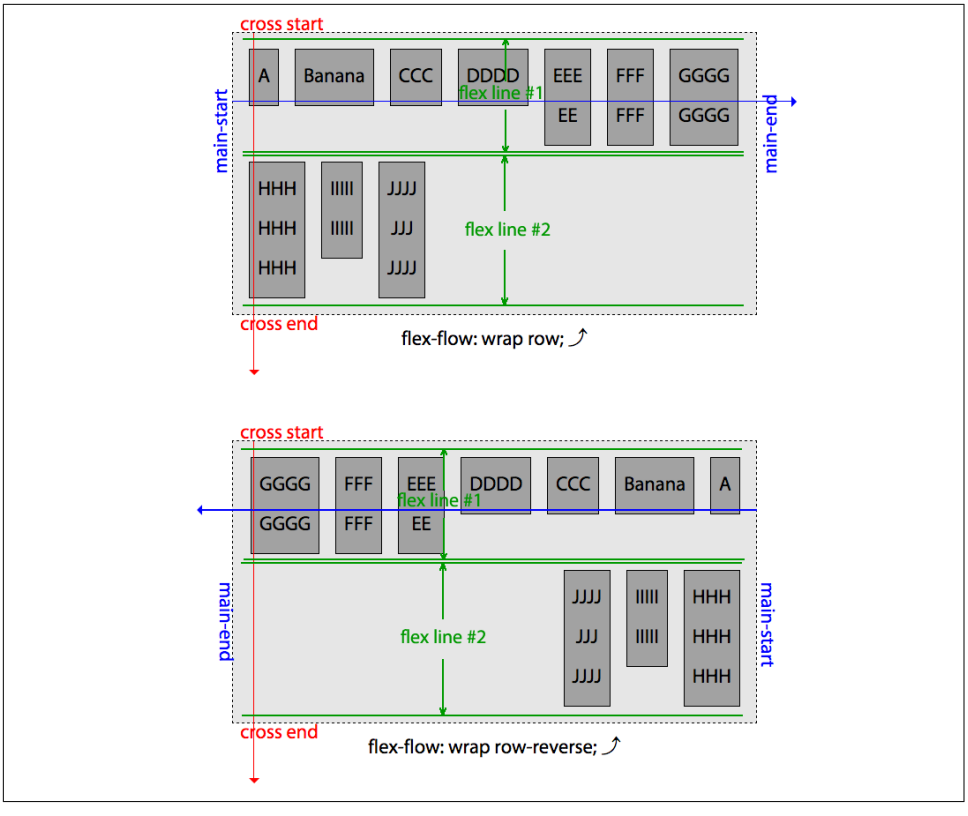


Figure 2-9. Flex lines on `row` and `row-reverse` when `flex-wrap: wrap` is set

The `wrap-reverse` value inverts the direction of the cross-axis. Normally for `flex-direction` of `row` and `row-reverse`, the cross-axis goes from top to bottom, with the cross-start on top and cross-end on the bottom, as shown in Figure 2-9. When `flex-wrap` is set to `wrap-reverse`, the cross-start and cross-end directions are swapped, with the cross-start on the bottom, cross-end on top, and the cross-axis going from bottom to top, as shown in Figure 2-7. Additional flex lines get added on top of, or above, the previous line.

If the `flex-direction` is set to `column` or `column-reverse`, by default the cross-axis goes from left to right in left-to-right languages, with new flex lines being added to the right of previous lines. As shown in Figure 2-10, when `flex-wrap` is set to `wrap-reverse`, the cross-axis is inverted, with cross-start being on the right, cross-end being on the left, the cross-axis going from right to left, with additional flex lines being added to the left of the previously drawn line.

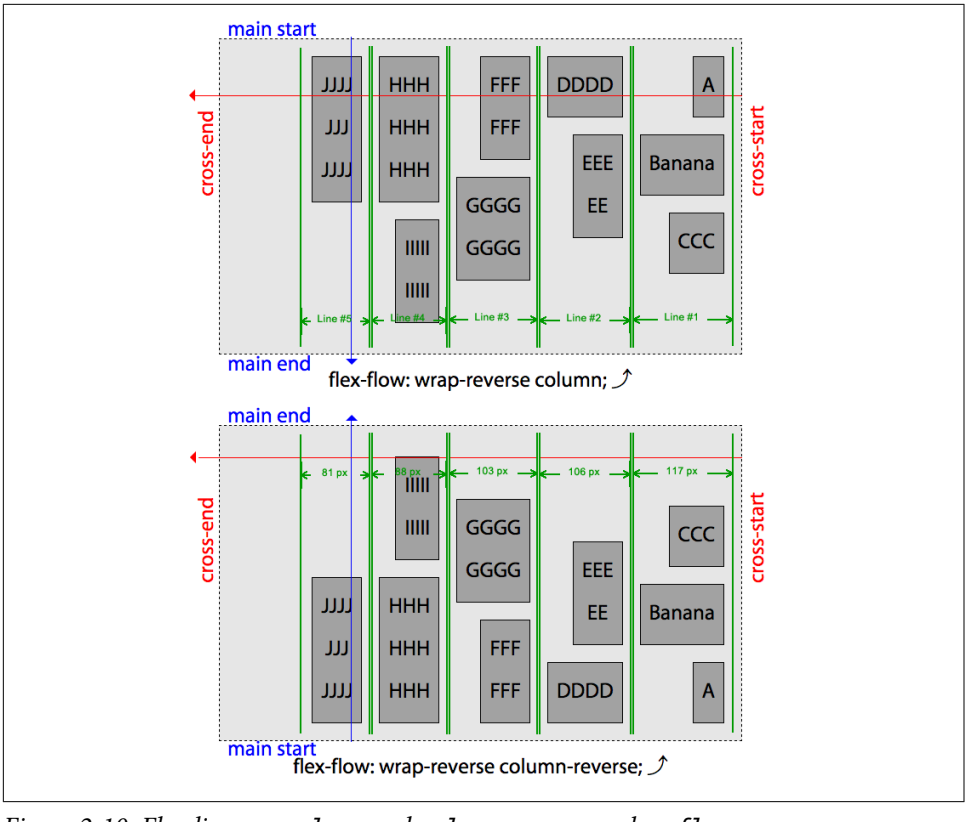


Figure 2-10. Flex lines on `column` and `column-reverse` when `flex-wrap: wrap-reverse` is set

We added `align-items: flex-start` (see "The `align-items` Property" on page 45) and `align-content: flex-start` (see "The `align-content` Property" on page 54) to the flex container in Figures 2-9 and 2-10 to enunciate the height and directions of the flex lines. These properties are covered in the following sections.

The `flex-wrap` property seemed fairly intuitive when it was first described. It turned out to be a bit more complex than it might have originally seemed. You may never implement reversed wrapping in a right-to-left language, but this is a "Definitive Guide." Now that we have a better understanding of the "cross-" dimensions, let's dig deeper into the `flex-wrap` property.

flex-wrap continued

The default value of `nowrap` prevents wrapping, so the `cross-` directions just discussed aren't relevant when there is no chance of a second flex line. When additional lines are possible—when flex-wrap is set to `wrap` or `wrap-reverse`—those lines will be added in the cross direction, which is perpendicular to the main-axis. The first line starts at the cross-start with additional lines being added on the cross-end side.

The `wrap-reverse` value inverts the direction of the cross-axis. Normally for `flex-direction` of `row` and `row-reverse`, the cross-axis goes from top to bottom, with the cross-start on top and cross-end on the bottom. When `flex-wrap` is `wrap-reverse`, the cross-start and cross-end directions are swapped, with the cross-start on the bottom, cross-end on top, and the cross-axis going from bottom to top. Additional flex lines get added on top of the previous line.

You can invert the direction of the cross-axis, adding new lines on top or to the left of previous lines by including `flex-wrap: wrap-reverse`. In Figures 2-9, 2-10, and 2-11, the last example in each is `wrap-reverse`. You'll notice the new line starts at the main-start, but is added in the inverse direction of the cross-axis set by the `flex-direction` property.

In Figure 2-11, the same `flex-wrap` values are repeated, but with a `flex-direction: column` property value instead of `row`. In this case, the flex items are laid out along the vertical axis. Just as with the first example in Figure 2-7, if wrapping is not enabled by the `flex-wrap` property, either because `flex-wrap: nowrap` is explicitly set on the container, or if the property is omitted and it defaults to `nowrap`, no new flex lines will be added, even if that means the flex items are drawn beyond the bounding box of the flex container.

With `column`, just like with `row`, if the flex items don't fit within the flex container's main dimension, they'll overflow the flex container, unless explicitly forced with `min-width: 0` or similar, in which case they shrink to fit, though flex items will not shrink to smaller than their border, padding and margins combined.

When "`flex-direction: column; flex-wrap: wrap;`" or "`flex-flow: column wrap;`" is set on a flex container, the flex item children are aligned along the main-axis. In LTR modes, the first flex item is placed in the top left, which is the main-start and cross-start respectively. If there is room, the next item will be placed below it, along the main-axis. If there isn't enough room, the flex container will wrap the flex items onto new lines. The next flex item will be put on a new line, which in this case is a vertical line to the right of the previous line, as can be observed in the `flex-flow: column wrap` example, the top-right example in Figure 2-11.

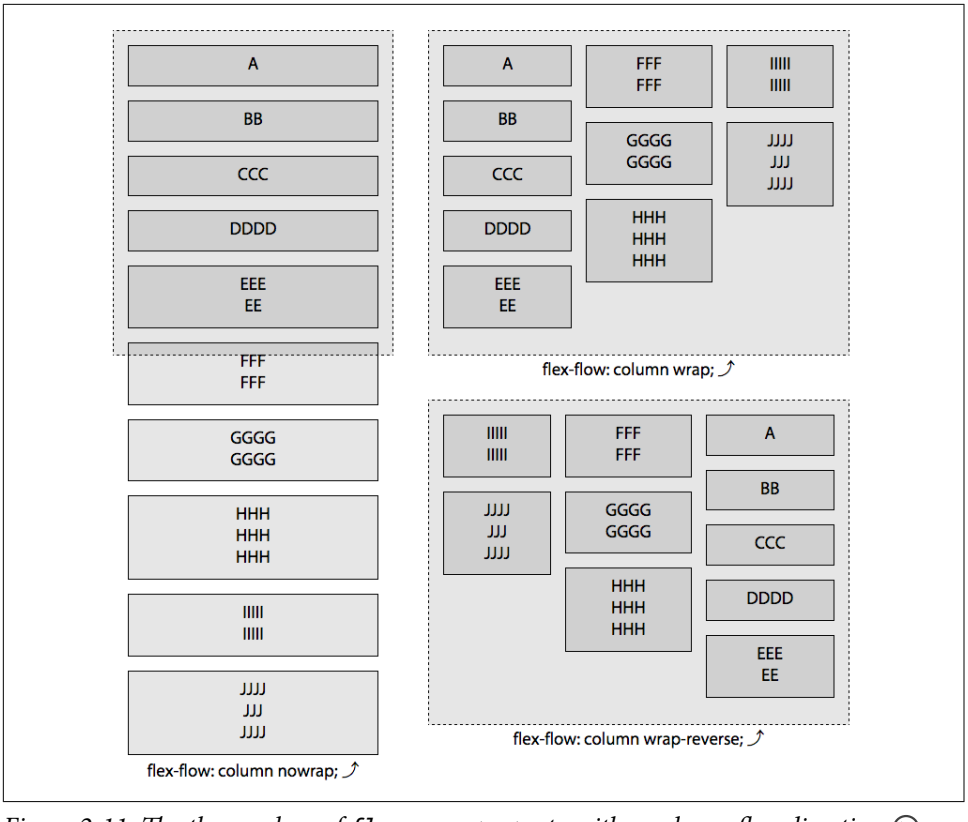


Figure 2-11. The three values of `flex-wrap` property with a column flex-direction

Here, the flex items have wrapped onto 3 lines. The flex items are wrapping onto new lines as the `height: 440px` was set on the parent. Including a height forces the creation of new flex lines when the next flex item will not fit onto the current flex line.

As shown in the bottom-right example in Figure 2-11, when we include "`flex-direction: column; flex-wrap: wrap-reverse;`" or "`flex-flow: column wrap-reverse;`" and the main-axis is the vertical axis, flex items are added below the previous flex item, if they can fit within the parent container. Because of the `wrap-reverse`, the cross-axis is reversed meaning new lines are added in the opposite direction of the writing mode. As shown in this example and in Figure 2-10, with `wrap-reverse`, columns are laid out right to left, instead of left to right. The first column of flex items is on the leftmost side of the parent flex container. Any additional required columns will be added to the left of the previous column. Again, we're assuming a left-to-right default writing mode for all the examples.

In this example, the flex items are different heights based on their content. When divided across three lines, the last line is not filled. Since subsequent lines are being added to the left of previous lines, the empty space, if there is any, will be on the bottom left (assuming `justify-content`, described next, is set or defaults to `flex-start`).

As you can see, `flex-direction` and `flex-wrap` have great impact on your layout and on each other. Because it's generally important to set both if you're going to set either, we are provided with the `flex-flow` property, which is simply shorthand for `flex-direction` and `flex-wrap`.

Flex Line Cross Dimension

Similar to the display examples in Figure 1-6, in the `flex-flow` example in Figure 2-1, the flex items grew to fill the cross dimension of the flex line they were on. Because they default to `nowrap`, all the flex items will be on a single line. With other flex properties defaulting to their default values, the flex items are stretched filling the container.

In the `flex-wrap` example in Figure 2-7, we had two flex lines. All the flex items in a single line were the same height, but the individual flex lines were not of the same height: instead all the flex items were as tall as the tallest flex item within that same flex line.

By default, all flex items will appear to be the same height. We controlled this in Figures 2-9 and 2-10 in order to highlight the how flex lines are drawn. This stretching (or overwriting of that stretching) is caused by the `stretch` default value of the `justify-content` property, described in the next section.

In those examples, you'll note the second row is not as wide as the flex container. When flex items are allowed to wrap onto multiple lines, by default all the items on one line will have the same cross dimension, which means all the items in a flex line of `row` and `row-reverse` will be the same height and all the items in a flex line of `column` or `column-reverse` will have the same width.

Figure 2-12 is an example in which `flex-flow: column-reverse wrap-reverse` is set. This means the main-axis is vertical, going bottom to top with a horizontal cross-axis going from right to left. Notice the extra space on the top left. New flex items get placed to the top of the previous ones, with new lines wrapping to the left of the previously filled row. By default, no matter the values of `flex-flow`, the empty space, if there is any, will be in the direction of main-end and cross-end. There are other flex properties that will allow us to alter that. Let's look into those.

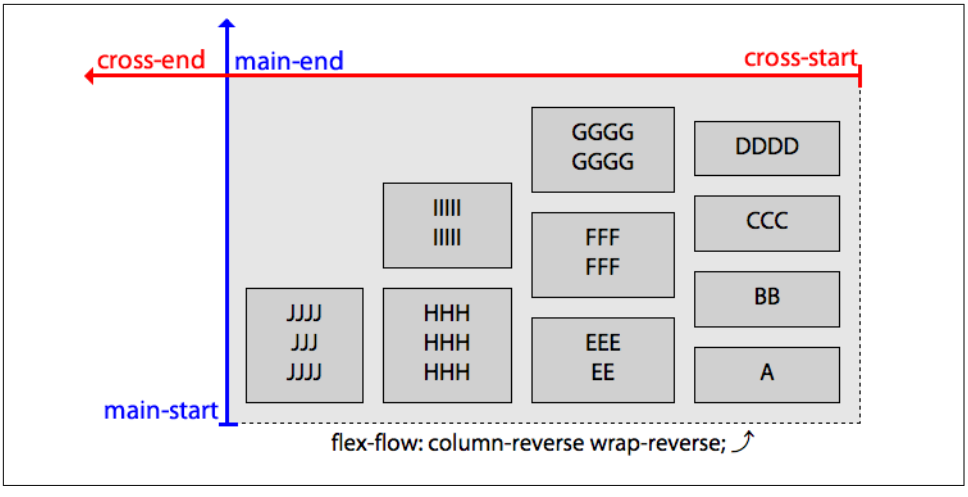


Figure 2-12. No matter the value of `flex-flow`, the empty space will be in the direction of main-end and cross-end

Flex Container

In Chapter 1 we learned how to use `display` property values `flex` and `inline-flex` to instantiate a flex container and turn the container's children into flex items. We now understand how `flex-direction` sets the direction of the flex items within the flex container, setting the main-axis and cross-axis directions and how `flex-wrap` can be used to enable multiline flex containers and even invert the cross-axis direction. We also know how to use `flex-flow` to set both `flex-direction` and `flex-wrap`.

While we learned how to use `flex-flow` properties to handle multiline flex containers, we haven't really discussed what happens to the extra space when the flex items don't fill a row or column, and what happens to the extra space when not all the flex items have the same cross dimension.

Thus far in our examples, when the flex items did not fill the flex container, the flex items have been grouped toward the main-start on the main-axis. We can control that. Flex items can be flush against the main-end instead. They can be centered. We can even space the flex items out evenly across the main-axis.

The flex layout specification provides us with flex container properties to control the distribution of space: in addition to `display` and `flex-flow`, the CSS Flexible Box Layout Module Level 1 properties applied to flex containers include the `justify-content`, `align-content`, and `align-items` properties.

The `justify-content` property controls how flex items in a flex line are distributed along the main-axis. The `align-content` defines how flex lines are distributed along

the cross-axis of the flex container. The `align-items` property defines how the flex items are distributed along the cross-axis of each of those flex lines.

The properties applied to individual flex items are discussed in Chapter 3.

The `justify-content` Property

The `justify-content` property enables us to define how flex items will be distributed along the main-axis of the flex container.

justify-content

Values:	`flex-start` | `flex-end` | `center` | `space-between` | `space-around`
Initial value:	`flex-start`
Applies to:	Flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

The distribution of elements along the main axis of the container is controlled with the `justify-content` property. If you remember from Figure 1-6, when an element is converted to a flex container, the flex items, by default, were grouped together at the main-start.

The `justify-content` defines how space is distributed. There are five values for the `justify-content` property: `flex-start`, the default value, and `flex-end`, `center`, `space-between`, and `space-around`.

As shown in Figure 2-13, by default or with `justify-content: flex-start` explicitly set, flex items are laid out flush against main-start. With `flex-end`, flex items are justified toward main-end. `center` groups the items flush against each other centered in the middle of the main-dimension along the main axis. The `space-between` value puts the first flex item on a flex line flush with main-start and the last flex item in each flex line flush with main-end, and then puts an equal amount of space between every pair of adjacent flex items. `space-around` evenly distributes the flex items, as if there were noncollapsing margins of equal size around each item. More examples of the five `justify-content` values are demonstrated in Figure 2-14.

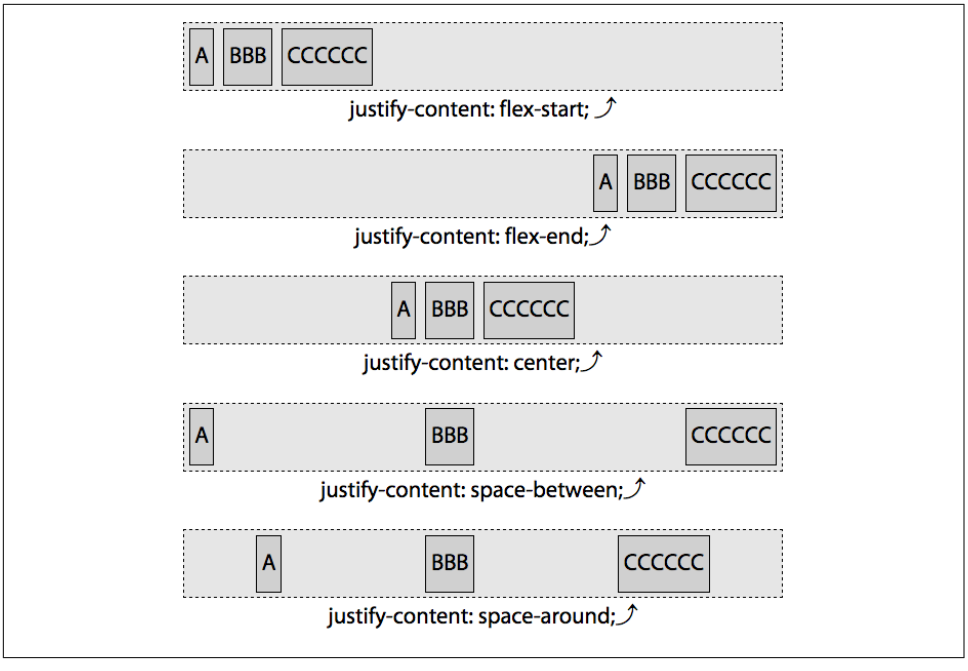


Figure 2-13. The five values of the `justify-content` property

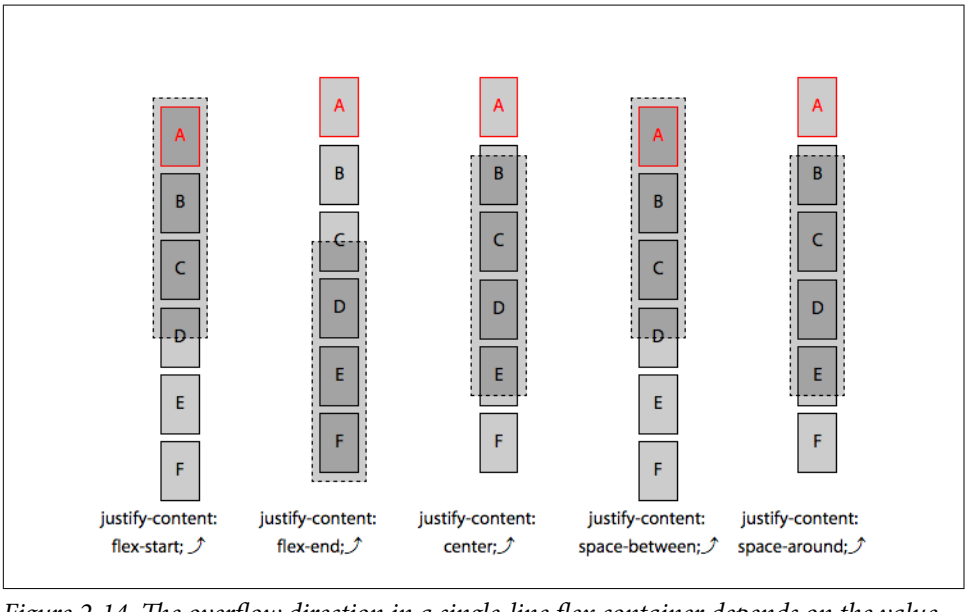


Figure 2-14. The overflow direction in a single-line flex container depends on the value of the `justify-content` property

As the `justify-content` property is applied to the flex container, the flex items will be distributed in the same manner whether they're on a filled first flex line or on a partially filled subsequent flex line in a wrapping flex container.

Those are the basics of `justify-content` property. But, of course, there's more to the property and each of the values. What happens if there is only one flex item on a line? If the writing mode is right to left? If `flex-flow: nowrap` is set and the flex items overflow the flex container?

If `nowrap` is set, and the items overflow the line, the `justify-content` property helps control the appearance of the line overflow. Figure 2-14 illustrates what happens with the different `justify-content` when `flex-flow: column nowrap` is set and the flex items heights are taller than the container's main dimension.

Let's take a look at the five values.

Setting `justify-content: flex-start` (Figure 2-15) explicitly sets the default behavior of grouping the flex items toward main-start, placing the first flex item of each flex line flush against the main-start side. Each subsequent flex item then gets placed flush with the preceding flex item's main-end side, until the end of the flex line is reached if wrapping is set. The location of the main-start side depends on the flex direction and writing mode, which is explained in "Understanding axes" on page 25.

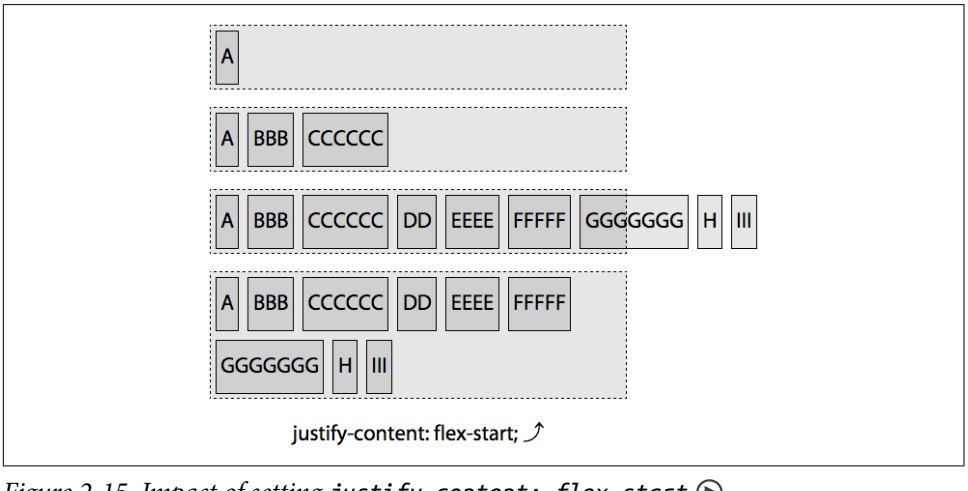


Figure 2-15. Impact of setting `justify-content: flex-start`

If there isn't enough room for all the items, and `nowrap` is the default or expressly set, the items will overflow on the main-end edge, as shown in the third example of Figure 2-15.

Setting `justify-content: flex-end` puts the last flex on a line flush against the main-end with each preceding flex item being placed flush with the subsequent item.

In this case, if the items aren't allowed to wrap, and if there isn't enough room for all the items, the items will overflow on the main-start edge, as shown in the third example of Figure 2-16. Any extra space on a flex line will be on the *main-start* side.

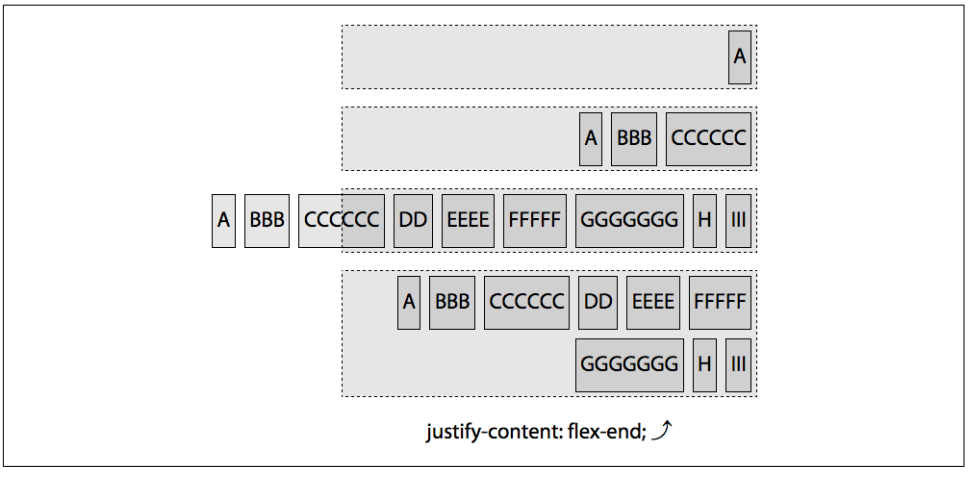


Figure 2-16. Impact of setting `justify-content: flex-end`

Setting `justify-content: center` will pack all the items together, flush against each other at the center of each flex line instead of at the main-start or main-end. If there isn't enough room for all the items and they aren't allowed to wrap, the items will overflow evenly on both the main-start and main-end edges, as shown in the third example in Figure 2-17. If the flex items wrap onto multiple lines, each line will have centered flex items, with extra space being on the main-start and main-end edges.

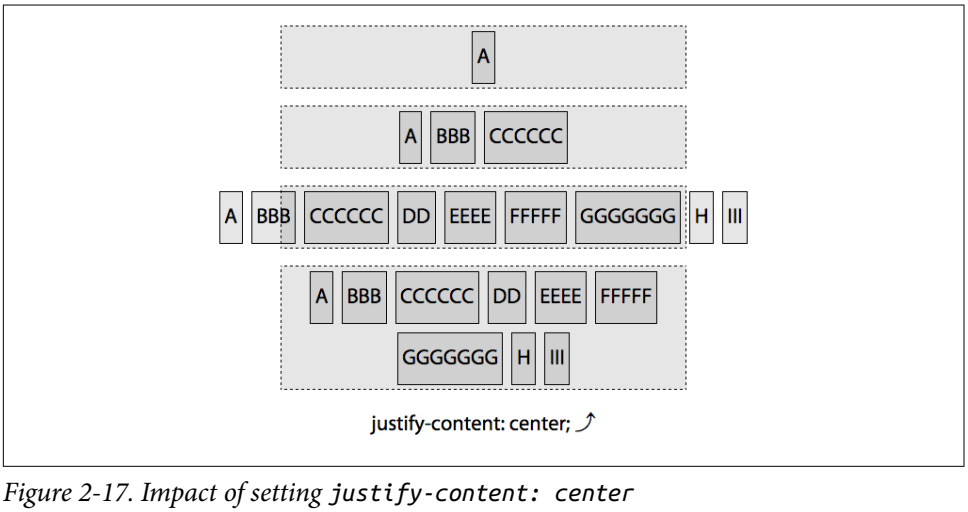


Figure 2-17. Impact of setting `justify-content: center`

Setting `justify-content: space-between` puts the first flex item flush with main-start and the last flex item on the line flush with main-end, and then puts an even amount of space around each flex item, until the flex line is filled. Then it repeats the process with any flex items that are wrapped onto additional flex lines. If there are three flex items, there will be the same amount of space between the first and second items as between the second and third, but there will be no extra empty space between the edge of the container and the first item and the opposite edge of the container and the outer edge of the last item, as shown in the second example in Figure 2-18. With `space-between`, the first item is flush with `main-start`, which is important to remember when you only have one flex item or when your flex items overflow the flex container in a `nowrap` scenario. This means, if there is only one flex item, it will be flush with main-start, not centered, which seems counterintuitive to many at first.

With `justify-content: space-between`, the first flex item is placed flush with main-start, so, if there is only one item, or if the flex items aren't allowed to wrap and overflow the flex container, the appearance will be the same as `flex-start`—the comparison can be seen in Figure 2-14—which may be less than intuitive.

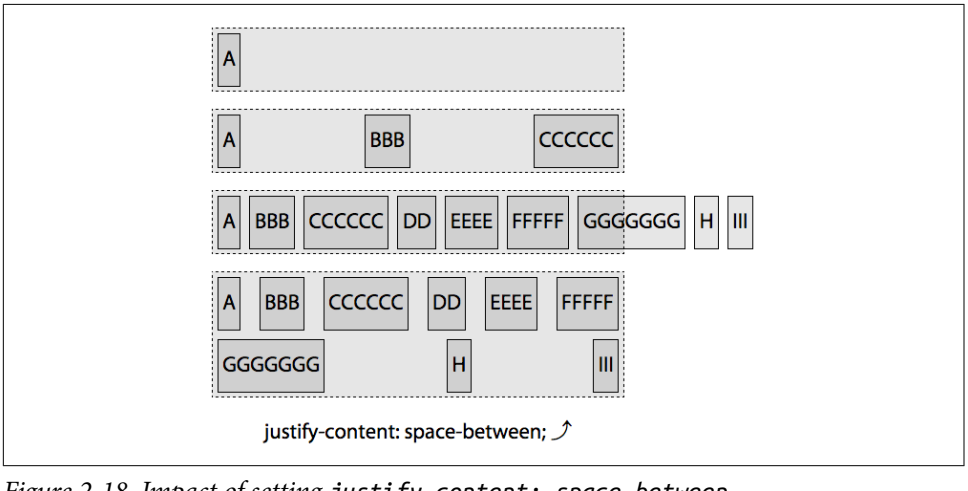


Figure 2-18. Impact of setting `justify-content: space-between`

With `justify-content: space-between` the space between any two items on a flex line will be equal but won't necessarily be the same across flex lines. When set to allow wrapping, on the last flex line, the first flex item of that last line is flush against main-start, the last if there are two or more on that line will be against main-end, with equal space between adjacent pairs of flex items. As shown in the last example of Figure 2-18, A and G, the first items on each flex line, are flush against main-start. F

and I, the last items on each line, are flush against main-end. The flex items are evenly distributed with the spacing between any two adjacent items being the same on each of the lines, but the space between flex items on the first line is narrower than the space between flex items on the second line.

Setting `justify-content: space-around` evenly distributes the extra space on the line around each of the flex items, as if there were noncollapsing margins of equal size around each element on the main-dimension sides. So there will be twice as much space between the first and second item as there is between main-start and the first item and main-end and the last item, as shown in Figure 2-19.

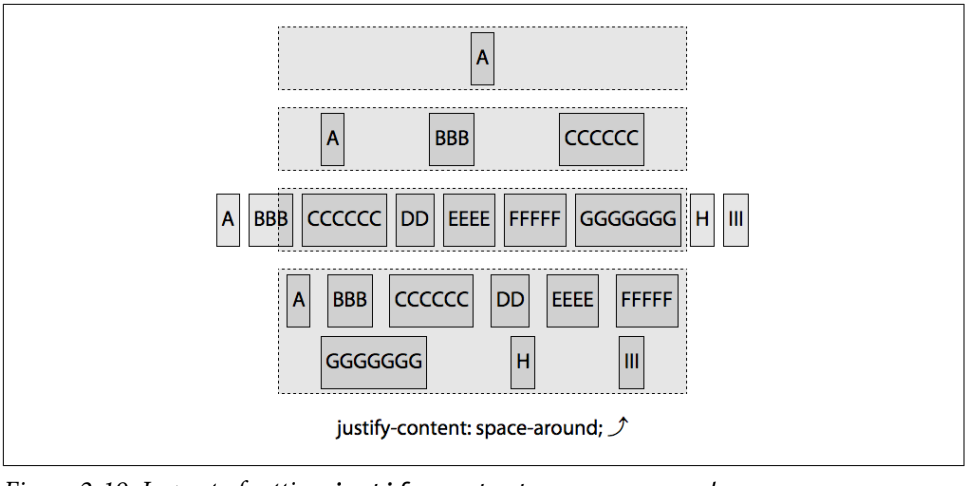


Figure 2-19. Impact of setting `justify-content: space-around`

If `nowrap` is set, and there isn't enough room on the flex container's main-direction for all the flex items, the flex items will overflow equally on both sides, similar to setting `center`, as shown in the third example in Figure 2-14.

If the flex items wrap onto multiple lines, the space around each flex item is based on the available space on each flex line. While the space around each element on a flex line with be the same, it might differ between lines, as shown in the last examples in Figure 2-19. The spaces between A and B and between G and H are twice the width of the spaces between the main-start edge and A and the edge and G.

With the margin added to the flex items to make the examples less hideous, this may be difficult to see. Comparing margin-free examples of `center`, `space-around`, and `space-between` might be more helpful.

Figure 2-20 demonstrates the difference between the spacing concepts of `center`, `space-around`, and `space-between`. When set to `center`, the flex items are grouped flush against each other in the center of the main-dimension.

With `space-between`, you'll note the first and last flex items on both flex lines abut main-start and main-end, respectively. The space between each of the flex items on the first flex line is 24 px: the 120 px of free space is divided into 5 equal gaps placed between the 6 flex items. The space between each flex item on the second flex line is 150 px: the 300 px of free space is divided into two and placed between the three flex items.

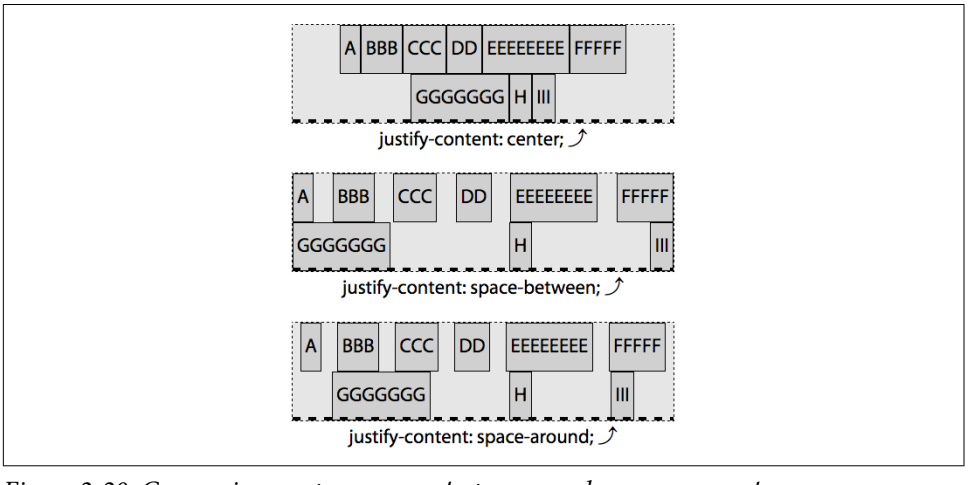


Figure 2-20. Comparing `center`, `space-between`, and `space-around`

With `space-around`, the flex items do not necessarily abut the edges: if there is any extra space, it is put around every flex item. In all three examples in Figure 2-20, there are 120 px of free space and 6 items on the first line and 300 px of free space and 3 flex items on the second flex line. On the first line of the `space-around` example, the 120 px of free space are divided among the two sides of the six items, placing 10 px on the left and right of each item. This way, there are 10 px of space between *main-start* and the first item and the last item and *main-end*, and twice that, or 20 px, between each adjacent flex item. On the second flex line, the 300 px of free space is divided between the two sides of the 3 flex items, putting 50 px on the outer edges and twice that, or 100 px, between adjacent items.

It may also help to compare the overflow edge and edges of the five `justify-content` properties with a different main-axis. In Figure 2-21, all flex containers have the following CSS:

```
container {
  display: inline-flex;
  flex-flow: nowrap column-reverse;
  height: 200px;
}
```

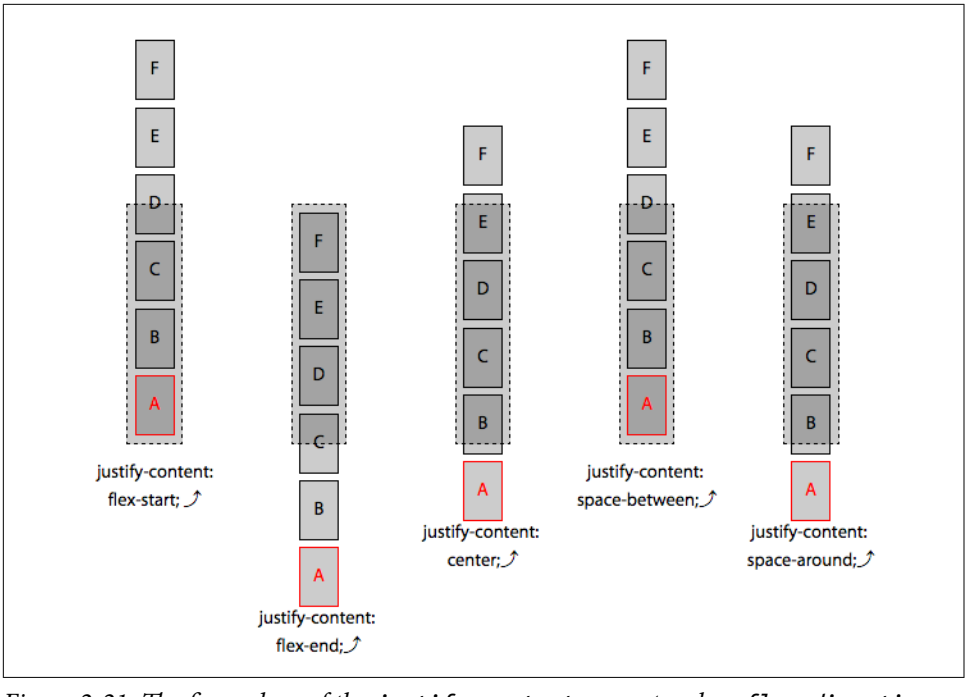


Figure 2-21. The five values of the `justify-content` property when `flex-direction: column-reverse` is set and flex items overflow the flex container

The layout when there is negative free space should now be more intuitive:

- `flex-start` and `space-between` overflow at main-end.
- `flex-end` overflows at main-start.
- `center` and `space-around` overflow on both ends.

With `justify-content: space-between`, the first flex item is placed flush with main-start. If the flex items don't fit on a line within the flex container, they will overflow on the main-end side, as shown in Figure 2-21. With `justify-content: space-around`, if there is only one flex item on a line, or more items than a nonwrapping flex line can hold, it will be centered. If there are multiple items, the outer spaces are half the size of the spaces between adjacent item.

In our examples, we've also prevented the flex items from shrinking, a default feature we haven't yet covered. The `height: 200px` limited the height of the container ensuring the flex items could overflow their container. The `display: inline-flex` declaration provided for a flex container that was only as wide as its content. Had we included `wrap` instead of `nowrap`, the container would have doubled in width, allowing for two columns or flex lines.

`justify-content` Examples

We took advantage of the default value in Figure 1-7, creating a left-aligned navigation bar. By changing the default value to `justify-content: flex-end` we can right align the navigation bar, as shown in Figure 2-22.

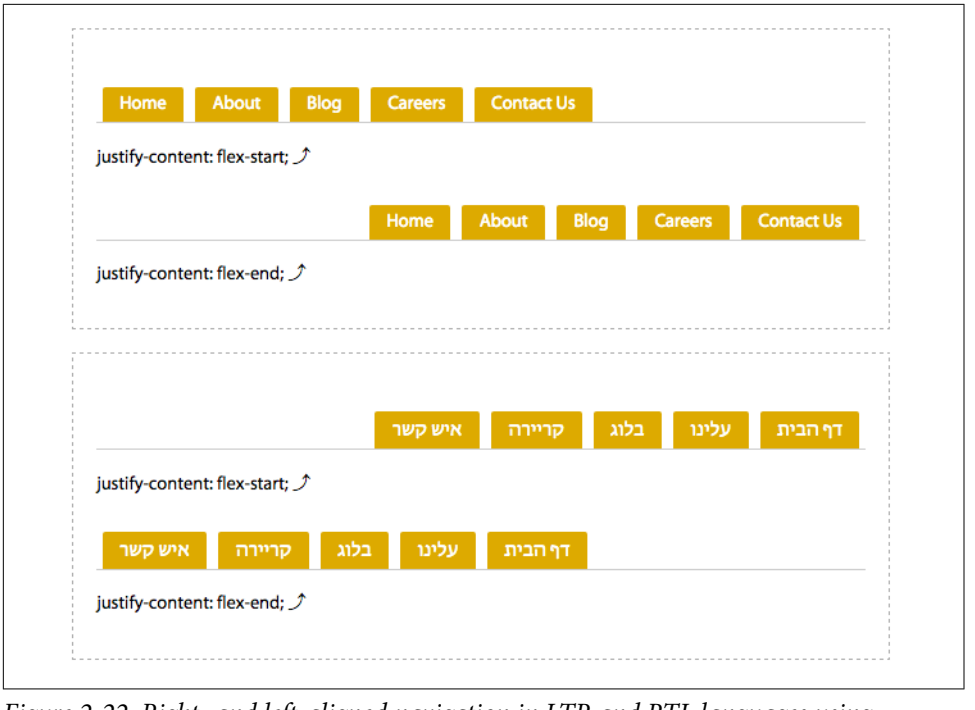


Figure 2-22. Right- and left-aligned navigation in LTR and RTL languages using `justify-content`

For right-to-left writing modes, we don't have to alter the CSS. As shown in Figure 2-21, and as discussed in Chapter 1, flex items are grouped toward main-start. In English, main-start is on the left. For Hebrew, main-start is on the right.

By simply adding a single line to our CSS, we altered the appearance of our navigation example, making the English version flush to the right and the Hebrew translation flush to the left:

```
nav {
  display: flex;
  justify-content: flex-end;
  border-bottom: 1px solid #ccc;
}
```

We could have centered that navigation, as shown in Figure 2-23:

```
nav {
  display: flex;
  justify-content: center;
  border-bottom: 1px solid #ccc;
}
```

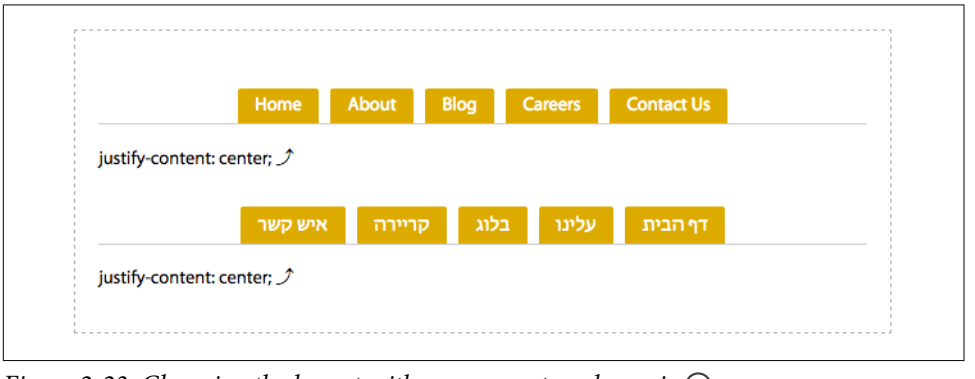


Figure 2-23. Changing the layout with one property value pair

The `align-items` Property

Whereas the `justify-content` defines how flex items are aligned along the flex container's main-axis, the `align-items` property defines how flex items are aligned along its flex line's cross-axis.

align-items

Values:	`flex-start` | `flex-end` | `center` | `baseline` | `stretch`
Initial value:	`stretch`
Applies to:	Flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

With the `align-items` property, you can align flex items to the start, end, or center of the cross-axis of their flex line. Set on the container element, `align-items` is similar to `justify-content` but in the perpendicular direction, setting the cross-axis alignment for all flex items, including anonymous flex items, within the flex container

(we'll learn how to override this value for individual flex items when we cover `align-self` ("The `align-self` Property" on page 110).

With `align-items`, you can set all the items to have their *cross-axis* flush against the cross-start or cross-end of their flex line, or stretched flush to both. Or you can center all the flex items in the middle of the flex line. Stretched across the entire cross-axis is the default and is what we have seen in most of the examples thus far. There are five values, including `flex-start`, `flex-end`, `center`, `baseline`, and the default `stretch`, as shown in Figure 2-24.

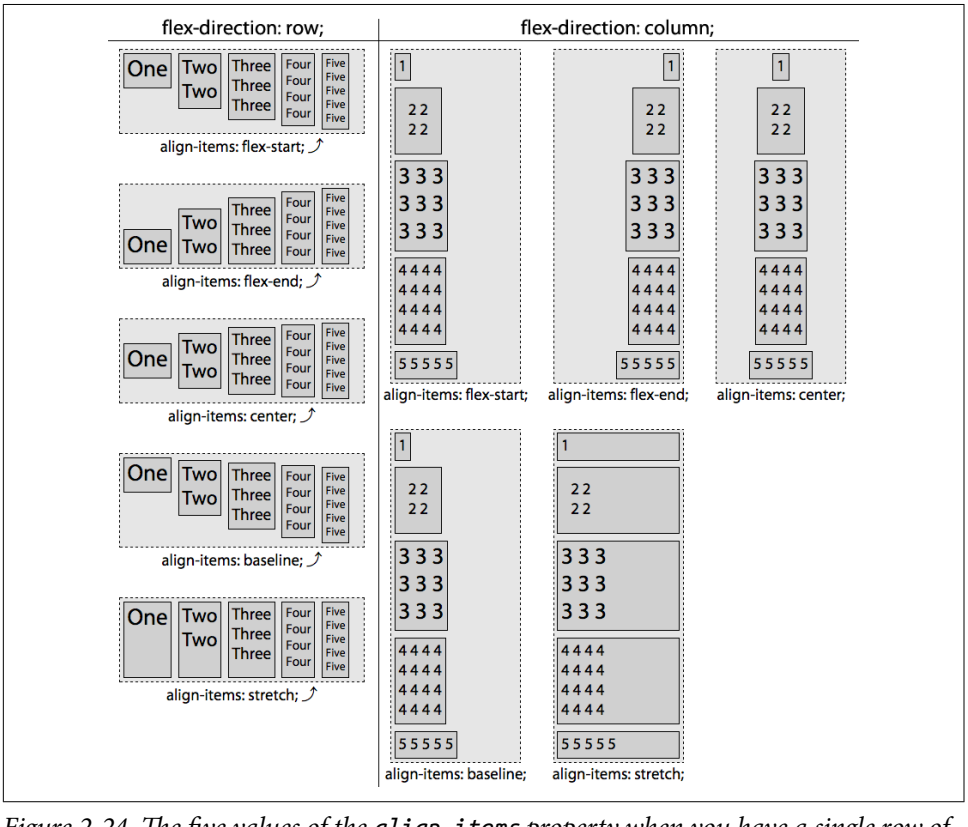


Figure 2-24. The five values of the `align-items` property when you have a single row of flex items and single column of flex items

While `align-items` sets the alignment for all the flex items within a container, the `align-self` property enables overriding the alignment for individual flex items.

In Figure 2-24, you'll note how the flex items either hug the cross-start or cross-end side of the flex container, are centered, or stretch to hug both, with the possible exception of `baseline`.

The general idea is `flex-start` places the flex items on the cross-start, `flex-end` puts them on the cross-end edge, `center` centers them on the cross-axis, the default `stretch` stretches the flex item from cross-start to cross-end, and `baseline` looks similar to `flex-start` but actually lines up the baselines of the flex items and then pushes the group of flex items toward cross-start.

With `baseline`, the flex items' baselines are aligned: the flex item that has the greatest distance between its baseline and its cross-start side will be flush against the cross-start edge of the line. That's the general idea—and explains nonwrapping flex containers pretty well—but there's more to it than that.

In the multiline `align-items` figures that follow, the following code has been included:

```
flex-container {
  display: inline-flex;
  flex-flow: row wrap;
  border: 1px dashed;
  padding: 10px;
}
flex-item {
  border: 1px solid;
  margin: 0 10px;
}
.C, .H {
  margin-top: 10px;
}
.D, .I {
  margin-top: 20px;
}
.J {
  font-size: 3rem;
}
```

For each flex line in Figures 2-25, 2-27, 2-28, 2-29 and 2-30, the red line is cross-start and the blue is cross-end. The lines appear purple when a new flex line abuts the previous flex line. C, H, D, and I have different values for top and bottom margins. We've added a bit of margin to the sides of all the flex items to make the figures more legible, which doesn't affect the impact of the `align-items` property. J has the font size increased, increasing the line height. This will come into play when we discuss the `baseline` value.

The default is `align-items: stretch`, as shown in Figure 2-25.

`align-items: stretch`

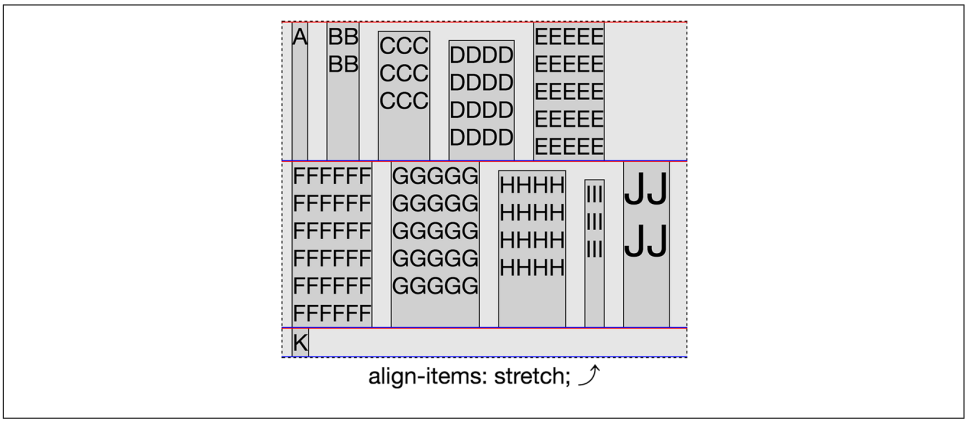


Figure 2-25. `align-items: stretch`

The default and the explicitly set `align-items: stretch` stretches all the stretchable flex items in a line to be as tall or wide as the tallest or widest flex item on the line. What does "stretchable" mean? While by default all the flex items will stretch to take up 100% of the cross-size, if `min-height`, `min-width`, `max-height`, `max-width`, `width`, or `height` are set, those properties will take precedence.

If set or defaulting to `stretch`, the flex items' cross-start will be flush with the flex line's cross-start, and the flex items' cross-end will be flush with the flex line's cross-end. The flex item with the largest cross-size will remain its default size, and the other flex items will stretch, growing to the size of that largest flex item on that same flex line. The cross-size of the flex items includes the margins, as demonstrated by items C, D, H, and I.

The margins in the cross direction have an impact. In Figure 2-26, we've added 30 px of margin to the cross-start edge of `C` of flex item and 40 px to the cross-end edge of the `D`. You'll note `A`, `B`, and `E` are flush against both cross edges, but the `C` and `D` appear pushed in. This is due to the margins: their outer margins are flush against cross-start and cross-end.

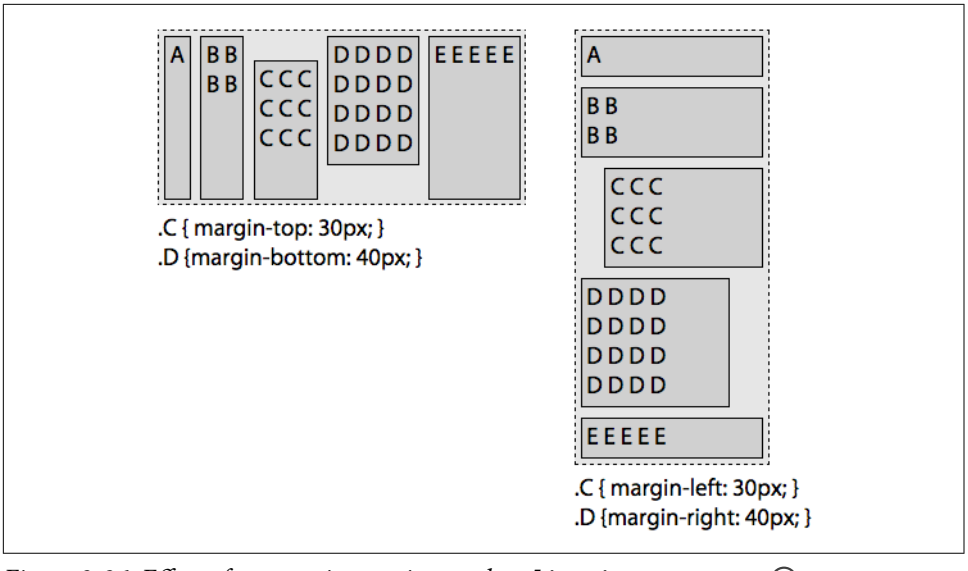


Figure 2-26. Effect of cross-axis margins on the `align-items` property

The size of the stretched flex item includes the margins on the cross-start and cross-end sides: it is the outer edge of the flex items' margin that will be flush with cross-start and cross-end. This is the reason C, D, H, and I may appear smaller than the other flex items on their flex lines. They're not. The outer edge of the top and bottom margins are flush with the cross-start and cross-end of the flex lines they occupy. Those flex lines are, in turn, as tall as the tallest item on the line (or as wide as the widest item when the cross dimension is horizontal).

Flex lines are only as tall as they need to be to contain their flex containers. In the five `align-items` figures, the line height of the flex line containing only K is much smaller than the line containing E, which is smaller than the line containing F. K has only one line of text, and no margin, whereas E has five lines of text. The second line, which includes F with six lines of text, making it even taller than the first line.

align-items: flex-start

The `flex-start` value lines up each flex items' cross-start edge flush against the cross-start edge of their flex line. The flex item's cross-start edge is on the outside of the margin: if a flex item has a margin that is greater than 0, flex item will not appear flush with the flex line's cross-start edge, as seen in flex item C, D, H, and I, in Figure 2-27.

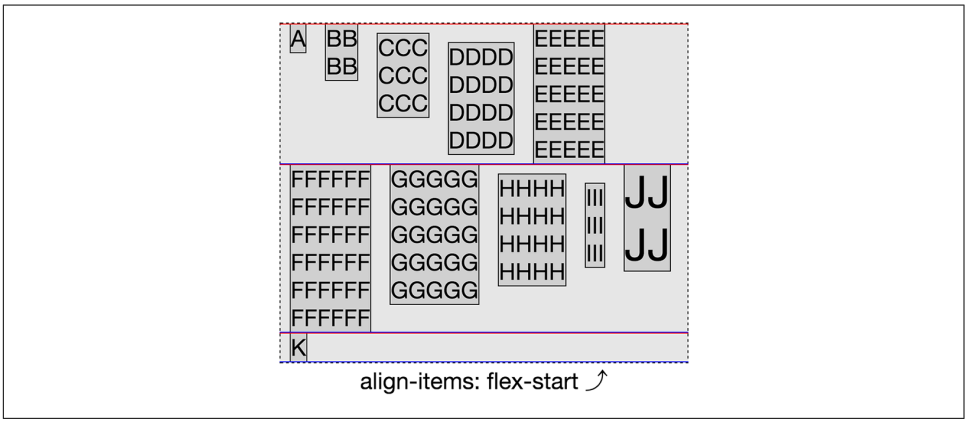


Figure 2-27. `align-items: flex-start`

align-items: flex-end

Setting `align-items: flex-end` will align the cross-end edge of all the flex items along the cross-end edge of the line they are in as shown in Figure 2-28. In these examples, none of the flex items have a bottom margin greater than 0 px, so unlike our other examples, this example does not look jagged.

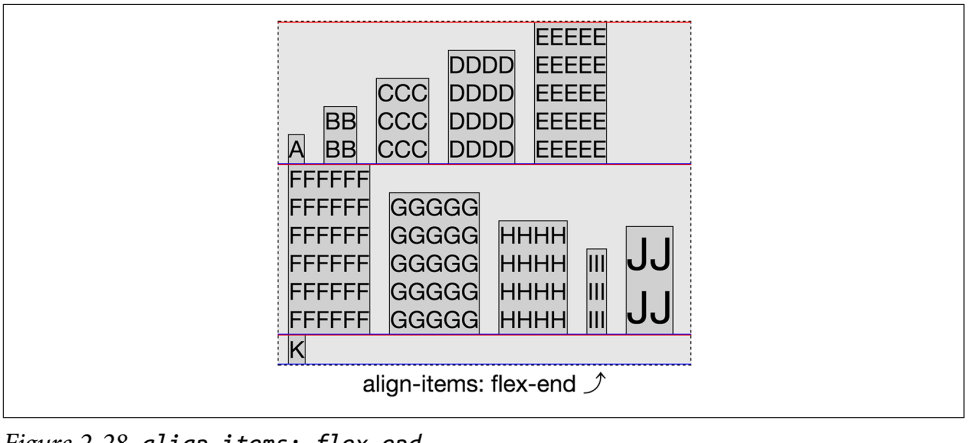


Figure 2-28. `align-items: flex-end`

`align-items: center`

As shown in Figure 2-29, setting `align-items: center` will center the flex items' cross-size along the middle point of the cross-axis of the line. The center is the midpoint between the outer edges of the margin, remembering flex item margins do not collapse. Because the cross-edge margins for C, D, H, and I are not symmetrical, the flex items do not appear centered along the cross-axis, even though they are. In LTR and RTL languages, in the case of `flex-direction: row` and `row-reverse`, the midpoint is the midpoint of the top margin, top border, top-padding, content or height, bottom padding, bottom border, and bottom margin. For `flex-direction: column`, and `column-reverse`, the midpoint is the midpoint of the left margin, left border, left-padding, content or width, right padding, right border, and right margin.

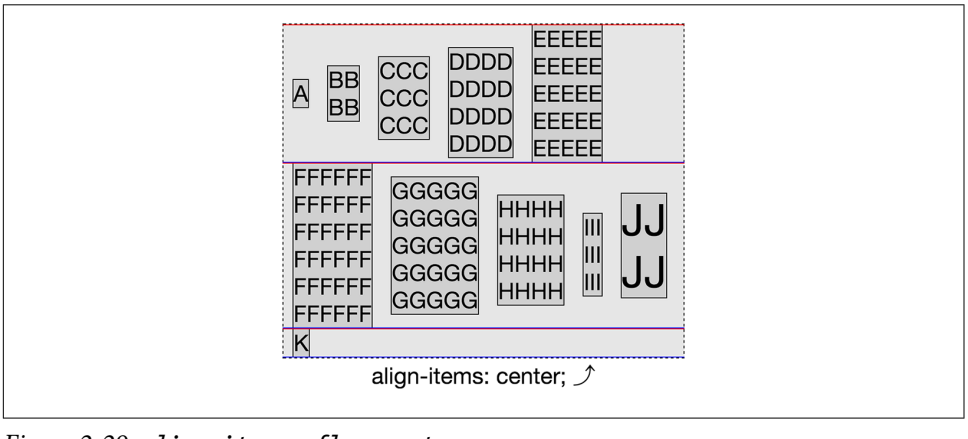


Figure 2-29. `align-items: flex-center`

Flex items may overflow their parent flex container along in the main-axis if `nowrap` is set and the main-size is constrained. Similarly, if the flex container's cross-size is constrained, the contents may overflow the flex container's cross-start and/or cross-end edge. The direction of the overflow is not determined by the `align-items` property, but rather by the `align-content` property, discussed next. The `align-items` aligns the flex items within the flex line and does not directly impact the overflow direction of the flex items within in the container.

`align-items: baseline`

The `baseline` value may be a little more confusing. With `baseline`, the flex items in each line are all aligned at their baselines, which is basically the bottom of the first line of text, if there is any. The flex item on each flex line with the biggest distance between its baseline and its cross-start margin edge is placed flush against the cross-start edge of the line, with all other flex items' baselines lined up with the baseline of that flex item.

Instead of aligning the cross-start of each flex item flush against the cross-start of each line, the flex items are aligned so their baselines align. In many implementations, `baseline` will look like `flex-start`, but will differ from the `flex-start` value if the flex items have different margins, padding, or border on the cross-start side, or if the first lines of content of the flex items don't all have the same line heights.

You'll notice that A, B, C, D, and E all seem aligned at top. What you may have missed is that they are not flush to the top—they are not flush against the red line. D has a top margin of 20 px. The outer edge of D's top margin is flush against the cross-start of the flex line, which is flush with the top of the flex container. As previously noted, the distance between the cross-start line and baseline is determined by the item on the line that has the biggest distance between its outer margin on its cross-start side and its baseline. In the Figure 2-30 baseline example, the items with the largest distance on their own flex lines are D, J, and K. These items' outer margins on the cross-start side are placed flush against the cross-start edge of their respective lines, and the other items in the same flex lines have their baseline lined up with D, J, and K's baseline.

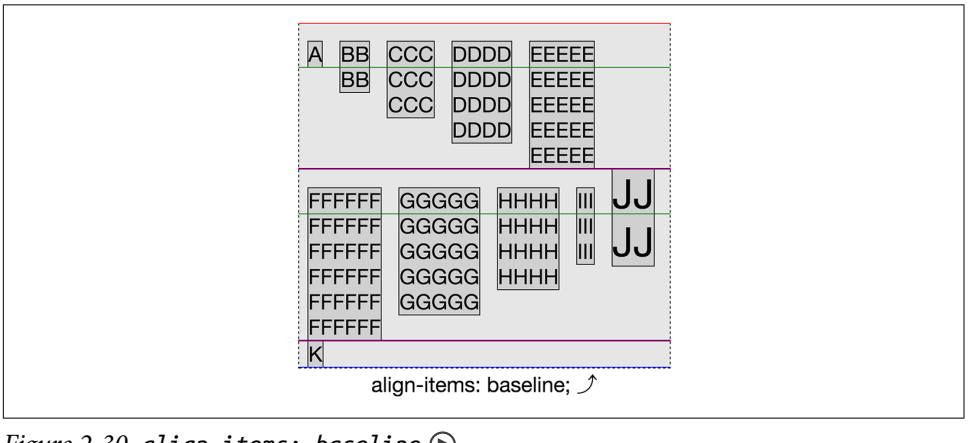


Figure 2-30. `align-items: baseline`

As A, B, C, D, and E all have the same line height, and D has the tallest top margin, they are aligned with D's baseline, and D's top margin is flush against the cross-start edge. When it comes to the first line, because they all have the same line height, border, and padding, it looks like they're lined up like `flex-start`, but they are actually a little lower, accommodating for D's top margin. The green line denotes where the baseline is.

In all the examples, we increased the font size for J to `3rem` to create a flex item that had a different baseline from all the other flex items. Only when `align-items: baseline` is set does this impact the flex item alignment within the flex line, as shown in Figure 2-30. When `align-items: baseline` is set, the baselines of all the items in the second flex line are aligned with J's baseline, as J is the flex item with the greatest space between the outer top margin and the bottom of the first line of text. Again, the green line denotes the approximate location of the baseline.

Additional Notes

The `align-items` property is set on the flex container and impacts all of the flex items within that flex container. If you want to change the alignment of one or more flex items, but not all, you can include the `align-self` property on the flex items you would like to align differently. The `align-self` takes the same values as `align-items`, and is discussed in Chapter 3.

You cannot override the alignment for anonymous flex items (nonempty text node children of flex containers): their `align-self` always matches the value of `align-items` of their parent flex container.

In the `align-items` examples, the flex container's cross-size was as tall as it needed to be. No `height` was declared on the container, so it defaulted to `height: auto`. Because of this, the flex container grew to fit the content. You may have noticed the example flex containers were all the same height, and the flex line heights were the same across all examples.

Had the cross-size, in this case the height, been set to a specific size, there may have been extra space at cross-end, or not enough space to fit the content. Flexbox allows us to control the alignment of flex lines with the `align-content` property. The `align-content` property is the last property we need to focus on that applies to the flex container (versus the flex items). The `align-content` property only impacts flex line alignment in multiline flex containers.

The `align-content` Property

The `align-content` property aligns a flex container's lines within a flex container that has extra space in the cross-axis direction, and dictates which direction will have overflow when there is not enough room to fit the flex lines.

align-content

Values:	`flex-start` | `flex-end` | `center` | `space-between` | `space-around` | `stretch`
Initial value:	`stretch`
Applies to:	Multiline flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	No

The `align-content` property allows us to dictate how any extra cross-direction space in a flex container is distributed between and around flex lines. This is different from the previously discussed `align-items` property which dictates flex item positioning within each flex line.

The `align-content` property determines how flex lines are aligned within a multiline flex container. When there is extra space in the cross-axis direction, you can dictate whether the lines are grouped at cross-start, cross-end, centered, or stretched to take up all the available space, or distributed across the flex container with the extra space distributed between or around the flex lines.

Think of `align-content` as similar to how `justify-content` aligns individual items along the main-axis of the flex container, but for flex lines across the cross-axis of the container. This property only applies to multiline flex containers, having no effect on nonwrapping and otherwise single-line flex containers.

Figure 2-31 demonstrates the six possible values of the `align-content` property.

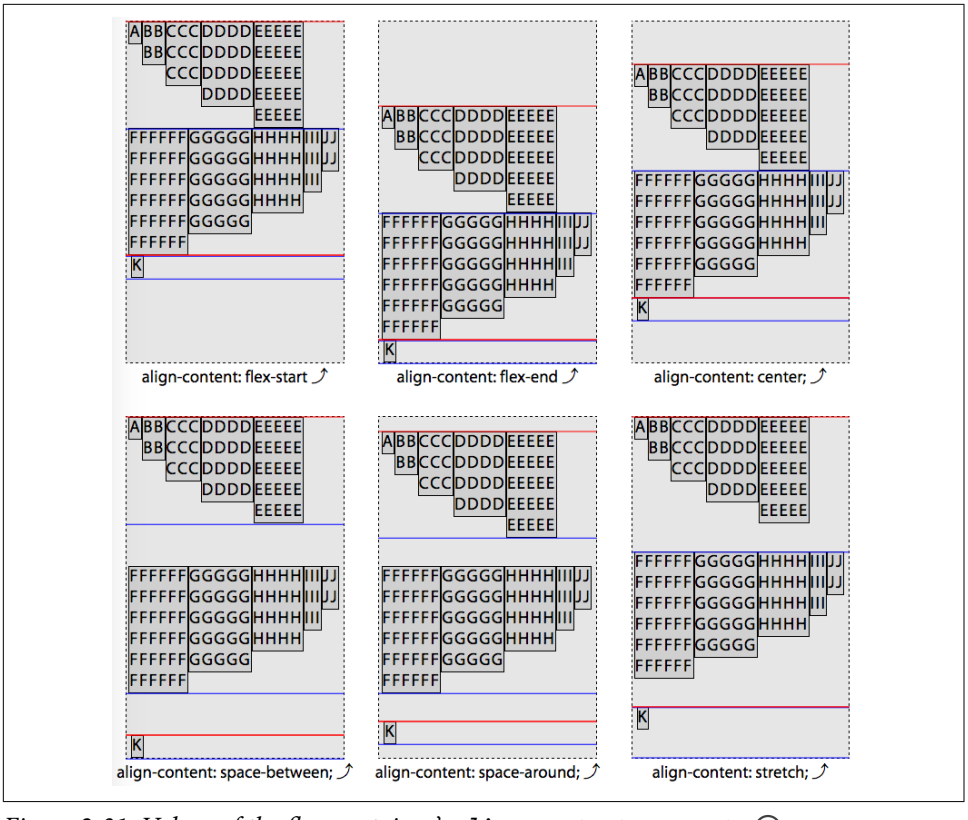


Figure 2-31. Values of the flex container's `align-content` property

We've used the following CSS as the base for the preceding six examples, with no margins on the flex items:

```
flex-container {
  display: flex;
  flex-flow: row wrap;
  align-items: flex-start;
  border: 1px dashed;
  height: 480px;
}
```

Distribution of extra space

In Figure 2-31, each flex container has three flex lines. With a height of 480 px, the flex container is taller than the default combined heights of the 3 flex lines. The tallest items in each line—E, F, and K—are 150 px, 180 px, and 30 px, respectively, for a combined total of 360 px. Each flex container has an extra 120 px of free space in the cross-size direction. The cross-start side of each flex line is denoted with a red line, the cross-end side with a blue line; they may appear purple when they overlap. With

five of the `align-items` values, the free space is distributed outside of the flex lines, as is more apparent in Figure 2-32. With `stretch`, the extra space is evenly distributed between all the flex lines, increasing their cross-size.

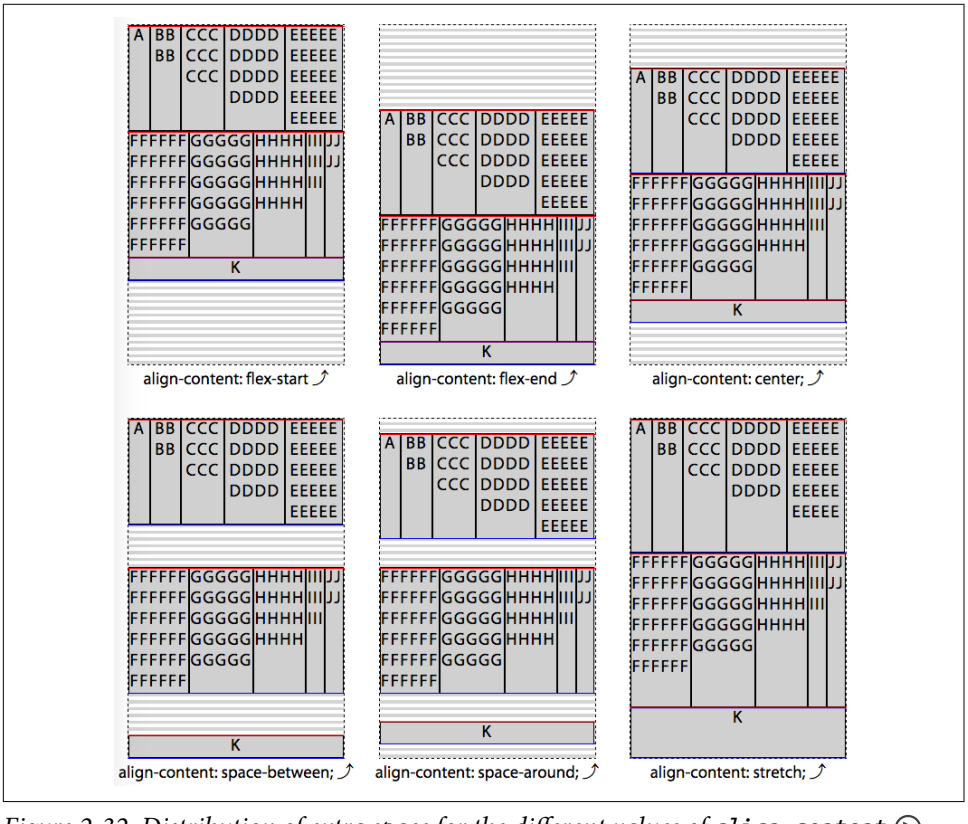


Figure 2-32. Distribution of extra space for the different values of `align-content`

Figure 2-32 reiterates the six possible values of the `align-content` property, with `align-items: stretch` and `flex: 1` set to allow the flex items to grow to take up their entire lines to make the impact of the `align-content` values more apparent (and generally, to be less of an eyesore):

```
flex-container {
  display: flex;
  flex-flow: row wrap;
  align-items: stretch;
  border: 1px dashed;
  height: 480px;
}
flex-items {
  flex: 1;
}
```

Just like in the previous example, with a height of 480 px, and with flex lines 150 px, 180 px, and 30 px tall, we have 120 px of free space along the cross-direction distributed differently depending on the value of the `align-content` property:

```
480 - (150 + 180 + 30) = 120
```

As shown in the first examples in both Figures 2-31 and 2-32, with `flex-start` the 120 px is on the cross-end side. With `flex-end` the extra 120 px of available space is at the cross-start side. With `center`, the lines are centered, with 60 px of extra space at both the cross-start and cross-end sides, as shown in the top-right example of both figures. With `space-between`, there is 60 px between adjacent pairs of flex lines, as shown in the bottom-left examples. With `space-around`, the space is evenly distributed around each line: the 120 px is distributed evenly, putting 20 px of non-collapsed space on the cross-start and cross-end sides of each flex line, so there are 20 px of extra space at the cross-start and cross-end sides of the flex container and 40 px of space between adjacent flex lines.

The `stretch` value is different: with `stretch` the lines stretch with the extra space evenly distributed among the flex lines rather than between them. In this case, 40 px was added to each of the flex lines. You'll note in the sixth example in both Figures 2-31 and 2-32, there is no area within the container that is not occupied by a flex line. Stretch is the default value, as you likely want to fill all the available space.

If there isn't enough room for all the lines, they will overflow at cross-start, cross-end, or both, depending on the value of the `align-content` property, as shown in Figure 2-33.

Figure 2-33 shows the size `align-content` values when the flex lines overflow their parent flex container. The only difference in the CSS between this and Figure 2-31 is the height of the flex container. We defined `height: 240px` to create flex containers not tall enough to encompass all their child flex items:

```
flex-container {
  display: flex;
  flex-flow: row wrap;
  align-items: flex-start;
  border: 1px dashed;
  height: 240px;
}
```

If the flex lines overflow the flex container, `flex-start`, `space-between`, and `stretch` overflow the cross-end side, `stretch` and `center` overflow evenly both the cross-end and cross-start sides, and only `flex-end` overflows only on the cross-start side.

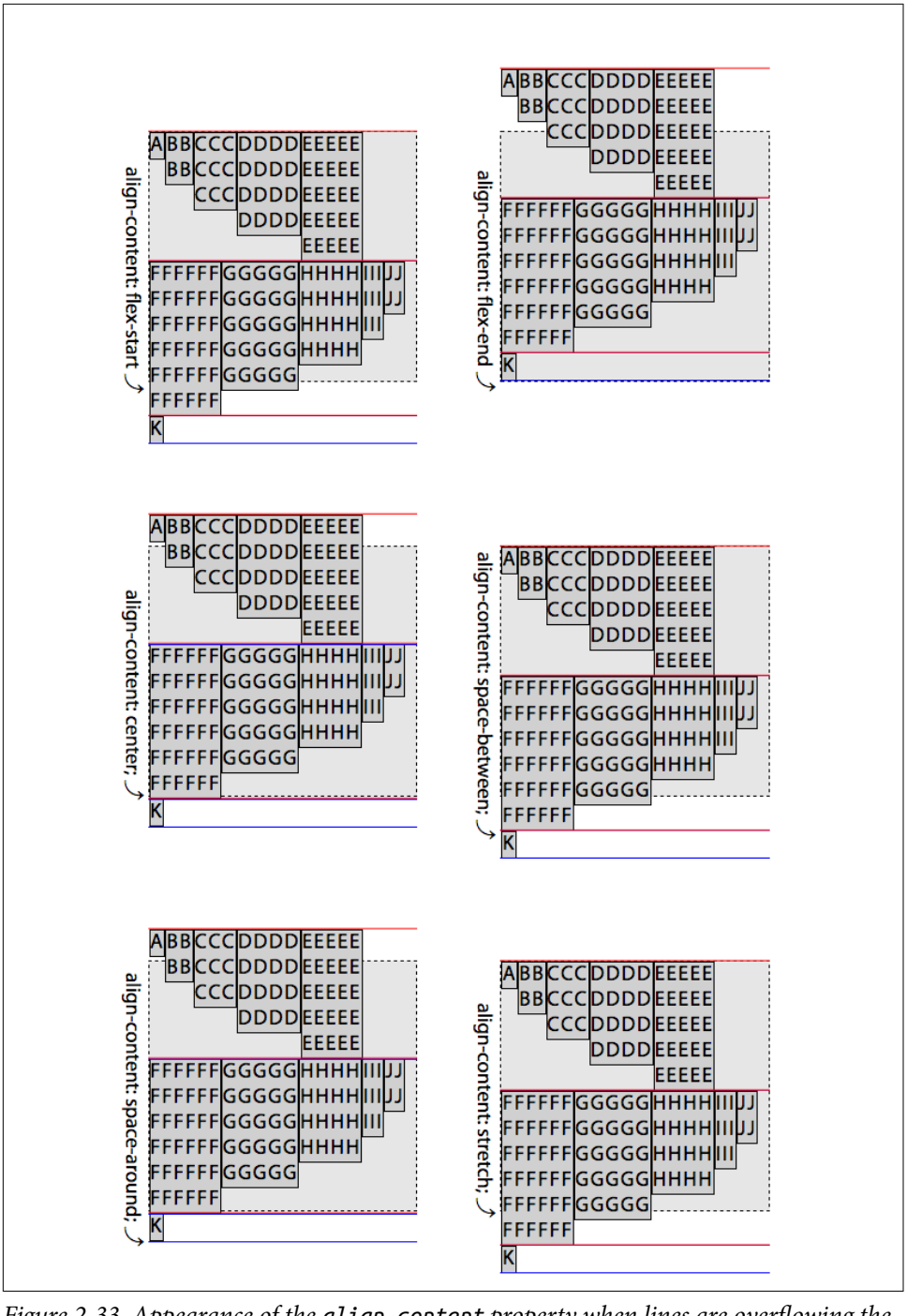


Figure 2-33. Appearance of the `align-content` property when lines are overflowing the container

`align-content: flex-start`

With `align-content: flex-start`, the cross-start edge of the first line of flex items will be flush against cross-start, with each subsequent line placed flush against the preceding line, as shown in Figure 2-34.

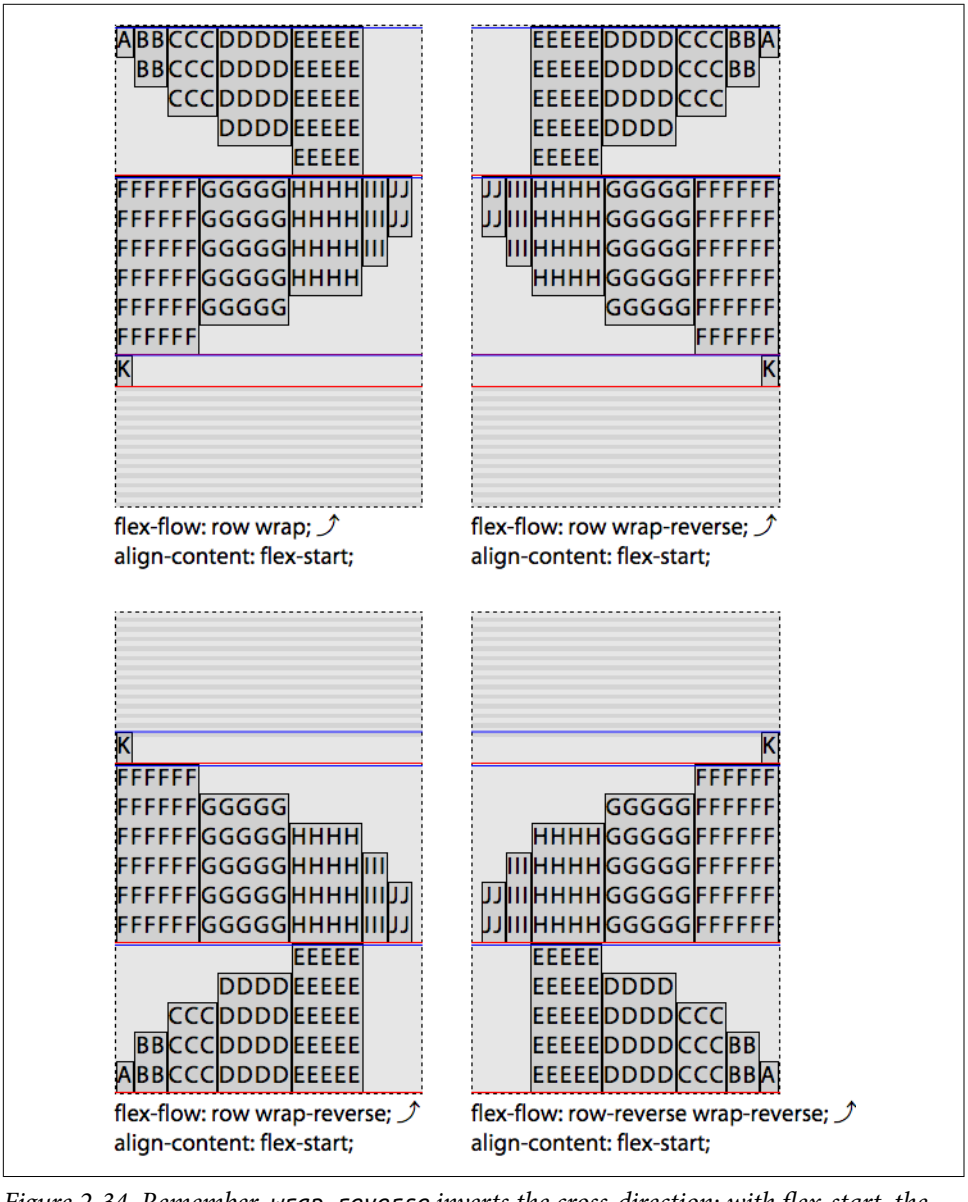


Figure 2-34. Remember, `wrap-reverse` inverts the cross-direction; with flex-start, the excess space or overflowing lines, if any, is always at cross-end ▶

In other words, all the flex lines will be grouped together at the cross-start edge of the flex container. Note all the extra space (the extra 120 px in this case) is at cross-end. Remembering the cross-direction is inverted with `flex-direction: wrap-reverse` (see "The `flex-direction` Property" on page 15), as shown in Figure 2-15. If there isn't enough room for all the lines, they will overflow at cross-end.

`align-content: flex-end`

With `align-content: flex-end`, the cross-end edge of the last line of flex items is placed flush with the cross-end edge of the flex container, and each preceding line is placed flush with the subsequent line, leaving all the extra whitespace at cross-start. In other words, all the flex lines will be grouped flush against the cross-end edge of the flex container. Should the content overflow the flex container, it will do so on cross-start side.

`align-content: center`

Declaring `align-content: center` groups the flex lines together, so they are flush against each other, just like they are grouped together with `flex-start` and `flex-end`, but in the center of the flex container.

`center`, `flex-start`, and `flex-end` all have each line's cross-end being flush against the subsequent line's cross-start. Instead of all the lines being grouped at the container cross-start as in `flex-start` or at cross-end as in `flex-end`, with `align-content: center` the group of lines is centered between the flex container's cross-start and cross-end, with equal amounts of empty space—60 px on each side in this case—between the cross-start edge of the flex container and the cross-start edge of the first flex line, and between the cross-end edge of the flex container and the cross-end edge of the last flex line, as shown in the top-right example in 2-31 and 2-32. If the flex items overflow the flex container, the lines will overflow equally in both directions past the cross-start and cross-end edges of the container, as shown in the center-left example in Figure 2-33.

`align-content: space-between`

When `align-content: space-between` is set, the flex lines are evenly distributed in the flex container. The *even distribution* is based on the available space, not the size of the lines; the extra space is divided equally among the lines, not proportionally.

If we remember back to `justify-content: space-between`, the first flex item is flush against main-start. The second flex item, if there is one, is flush against main-end. The rest of the flex items, if there are any, are spread out with equal amounts of the free space distributed between the flex items. This is similar to how `align-content: space-between` works. If there is more than one flex line, the first line will be flush against the container's cross-start, the last line will be flush against the container's cross-end, and the available extra space is distributed evenly between the additional lines, if there are any. The extra space is distributed evenly, not proportionally. The space between any two flex lines within the flex container is equal, even if the cross-sizes of the multiple flex lines differ.

Only flex containers with multiple lines can have free space in the cross-axis for lines to be aligned in. If there is only one line, the `align-content` property will not impact the distribution of the content. In flex containers with a single line of flex items, the lone line stretches to fill all of the available space.

The middle line, if there is an odd number of lines, is not necessarily centered, as the lines don't necessarily all have equivalent cross dimensions. Rather, the spacing between any two adjacent lines is the same: there is the same amount of space between the first and second line as there is between any other adjacent flex line, as shown in the bottom left example in Figure 2-32. In this case, we have 120 px total of free space, which gets divided equally, with half, or 60 px, between the first and second flex lines, and 60 px between the second and third flex lines:

```
120px / 2 = 60px
```

If there isn't enough space to encompass all the flex lines, the free space is negative and `align-content: space-between` appearance is identical to `flex-start`, with the overflow on the cross-end side.

`align-content: space around`

The `space-around` value distributes the lines within a multiline flex container evenly, as if all the flex lines had equal, noncollapsing margins on both the cross-start and cross-end sides. Because there is an equal distribution of the extra available space around each line, the space between the edges of the container and the first and last flex lines is half the size of the distance between any two flex lines. The distribution of the extra space is shown in Figure 2-35.

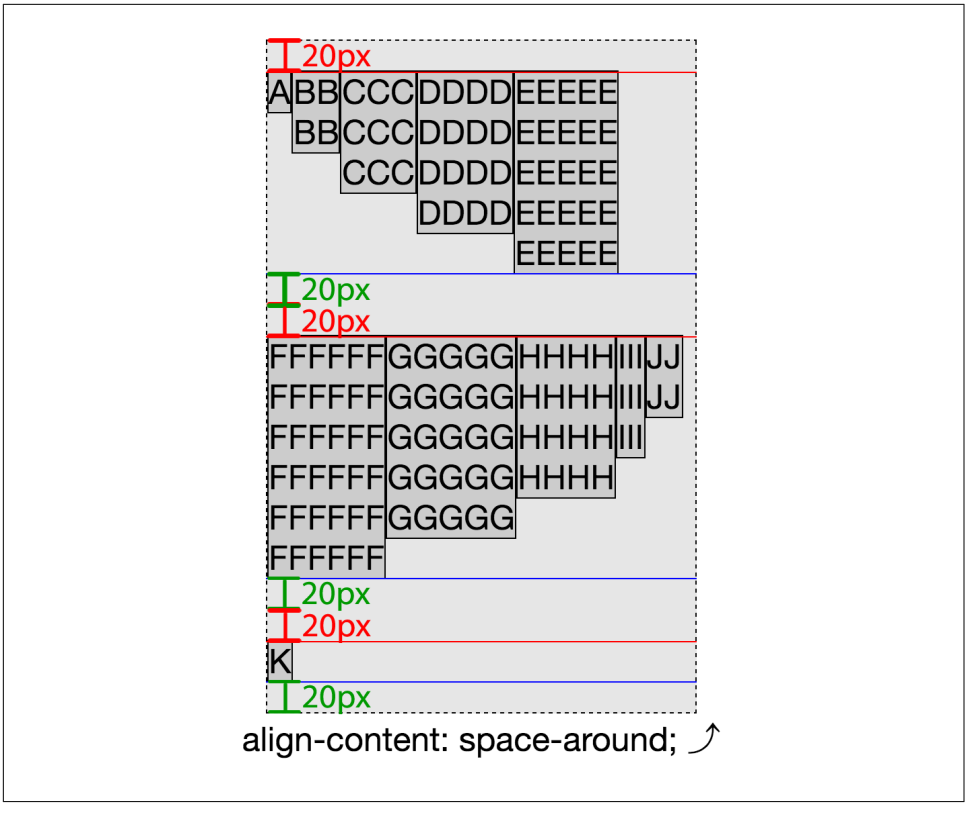


Figure 2-35. Distribution of free space when `align-content: space-around` is set

In this case we have 120 px of free space distributed to 3 flex lines, with half of each flex lines extra space being added to the cross-start edge and half being added to the cross-end edge:

```
(120px / 3 lines) / 2 sides = 20px per side
```

In this example, there is 20 px on the outer sides of the flex lines, with twice that between any two adjacent flex lines. There is 20 px between the cross-start edge and the first line, 20 px between the cross-end edge and the last line, and 40 px (twice 20 px) between the first and second flex lines and the second and third flex lines.

If there isn't enough room for the flex lines and they overflow the flex container, the free space is negative, and `align-content: space-around` value appears identical to `center`, with the overflow evenly distributed between the cross-start and cross-end sides.

`align-content: stretch`

Omitting the `align-content` property, or setting it explicitly to the `align-content: stretch` default value, makes the lines stretch to take up all the extra available space.

If you take a close look at Figure 2-32, you'll note that `stretch` and `space-between` are actually quite different. In `space-between`, the extra space is distributed *between* the lines. In `stretch`, the extra space is divided into the number of lines and added *to* the lines. The original lines were 150 px, 180 px, and 30 px tall, respectively. When set to stretch, the extra 120 px is added to the lines in equal amounts. As we have three flex lines and 120 px of extra space, each flex line grows by 40 px, giving us flex lines that are 190 px, 220 px, and 70 px, respectively, as shown in Figure 2-36.

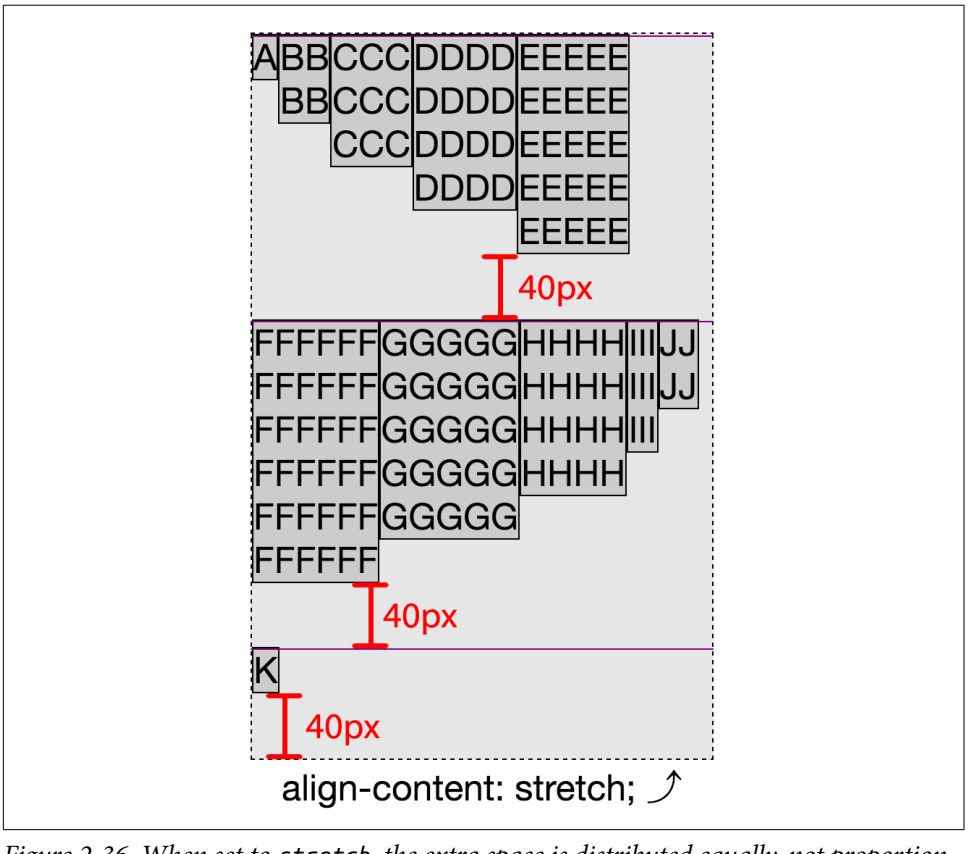


Figure 2-36. When set to `stretch`, the extra space is distributed equally, not proportionally, to each line

That extra space, in this example, appears at the cross-end of each individual line since we included `align-items: flex-start`. Had we used `align-items: flex-end` instead, the extra space in each line would have been apparent at the individual lines' flex-start.

If there isn't enough space to fit all the flex lines, the lines will behave as if set to `flex-start`, with the first flex line flush against the cross-start edge of the flex container.

We have been taking a look at properties of the flex container. It's time to take a look at properties directly applied to flex items.

CHAPTER 3

Flex Items

In the previous chapters, we learned how to globally lay out all the flex items within a flex container by adding flexbox property values to that container. The flexible box layout specification provides several additional properties applicable directly to flex items. With these flex item–specific properties, we can more precisely control the layout of individual flex containers' non-anonymous flex items.

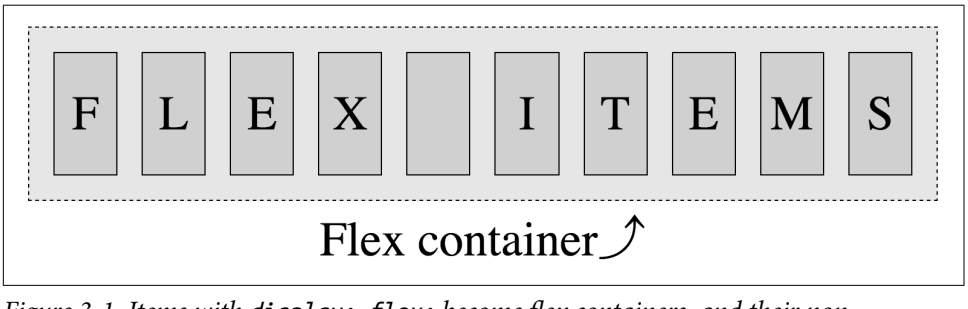


Figure 3-1. Items with `display: flex;` become flex containers, and their non-absolutely positioned children become flex items ▶

Now that we have a good understanding of the properties applicable to the flex container, it's time to focus on properties applicable to the flex items.

What Are Flex Items?

We create flex containers simply by adding a `display: flex` or `display: inline-flex` to an element that has child nodes. The children of those flex container are called *flex items*, be they DOM nodes, nonempty text nodes, or generated content.

When it comes to text-node children of flex containers, if the text node is not empty (containing content other than whitespace) it will be wrapped in an anonymous flex item, behaving like its flex item siblings. While these anonymous flex items do inherit all the flex properties set by the flex container, just like their DOM node siblings, they are not directly targetable with CSS. Therefore, we can't directly set any of the flex item–specific properties on them.

Generated content can be targeted directly; therefore all the properties discussed in this chapter apply equally to generated content as they do to element nodes.

Whitespace-only text nodes within a flex container are ignored, as if their `display` property were set to `none`, as the following code example shows:

```
nav ul {
  display: flex;
}

<nav>
  <ul>
    <li><a href="#1">Link 1</a></li>
    <li><a href="#2">Link 2</a></li>
    <li><a href="#3">Link 3</a></li>
    <li><a href="#4">Link 4</a></li>
    <li><a href="#5">Link 5</a></li>
  </ul>
</nav>
```

In the preceding code, with its `display` property set to `flex`, the unordered list is the flex container, and its child list items are all flex items. These list items, being flex items, are flex-level boxes, semantically still list items, but not list items in their presentation. They are not block-level boxes either. Rather, they participate in their container's flex formatting context. The whitespace is ignored. The links, which are descendants of the flex items, are not impacted by in inclusion of flex display on their parent's parent.

Flex Item Features

The margins of flex items do not collapse. The `float` and `clear` properties don't have an effect on flex items, and do not take a flex item out of flow. Additionally, `vertical-align` has no effect on a flex item. However, the `float` property can still affect box generation by influencing the `display` property's computed value, as the following code example shows:

```
aside {
  display: flex;
}
img {
  float: left;
}
```

```
<aside>
    <!-- this is a comment -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="Foo Master">
    Some text
</aside>
```

In this example, the `aside` is the flex container. The comment and whitespace-only text nodes are ignored. The text node containing "some text" is wrapped in an anonymous flex item. The header, image, and text node containing "some text" are all flex items. As the image is a flex item, the `float` is ignored. Even though images and text nodes are inline-level nodes, being flex items, as long as they are not absolutely positioned, they are blockified:

```
aside {
  display: flex;
}

<aside>
    <!-- a comment -->
    <h1>Header</h1>

    <img src="images/foo.jpg" alt="foo master">
    Some text <a href="foo.html">with a link</a> and more text
</aside>
```

In the last example, the markup is similar to the code in the second example, with the addition of a link within the nonempty text node. In this case, we are creating five flex items. The comment and whitespace-only text nodes are ignored. The header, the image, the text node before the link, the link, and the text node after the link are all flex items. The text nodes containing "some text" and "and more text" are wrapped in anonymous flex items. As these two text node children of the `<aside>` are not contiguous, they are wrapped in separate anonymous flex items. The header, image, and link, being DOM nodes, can be targeted directly with CSS. The anonymous flex containers are not directly targetable.

Absolute positioning

While a value of `float: left` or `float: right` on the child of a flex container does not float the item—as the child is a flex item and the `float` is ignored—setting `position: absolute` is a different story. The absolutely-positioned children of flex containers, just like any other absolutely-positioned element, are taken out of the flow of the document.

They do not get converted to flex items. They are not in the document flow. They do not participate in flex layout. However, they can be impacted by the properties set on

the flex container, just like a child can be impacted by a parent element that isn't a flex container. In addition to inheriting inheritable properties as they would had the parent not been a flex container, the parent's properties impact the origin of the positioning.

The absolutely positioned child of a flex container is impacted by both the `justify-content` value of the parent flex container and its own `align-self` value, if there is one. For example, if you set `align-content: center;` on the absolutely positioned child, it will by default be centered on the flex container parent's cross-axis. The `order` property may not impact where the absolutely-positioned flex container child is drawn, but it does impact the order of when it is drawn in relation to its siblings.

`min-width`

In Figure 3-2, you'll note the line that is set to the `nowrap` default overflows the flex container. This is because when it comes to flex items, the implied value of `min-width` is `auto`, rather than `0`. Originally in the specification, if the items didn't fit onto that single main-axis, they would shrink. However, the specification of `min-width` was altered. In the CSS 2.1 specification, the default value for `min-width` is `0`. Now, for flex items, the implied minimum size is `auto`, not `0`.

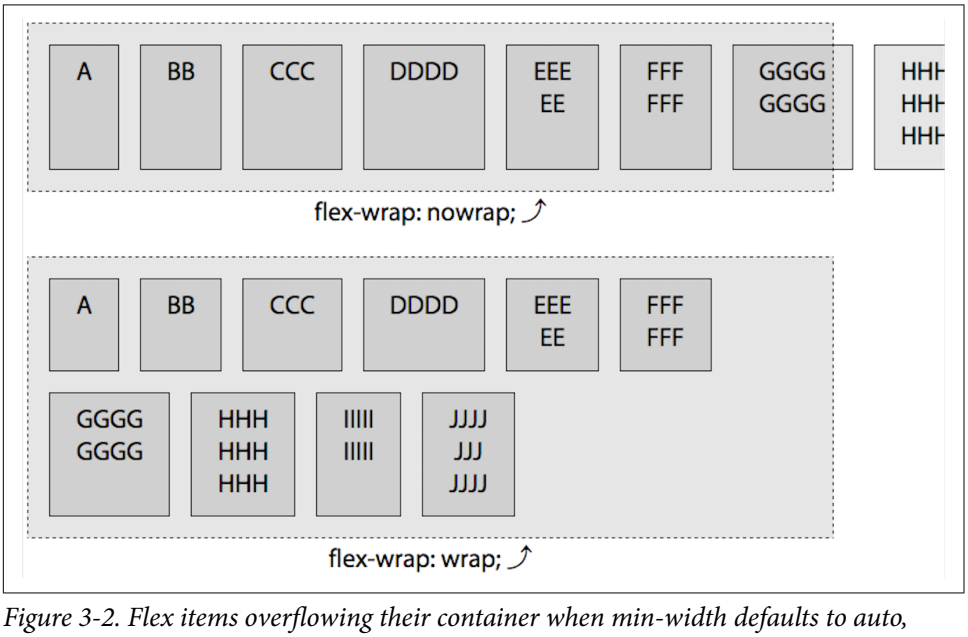


Figure 3-2. Flex items overflowing their container when min-width defaults to auto, unless wrapping is allowed ▶

If you set the `min-width` to a width narrower than the computed value of auto—for example, if we declare `min-width: 0;`—the flex items in the `nowrap` example will shrink to be narrower than the flex items in containers that are not allowed to wrap, as shown in Figure 3-3. This is what Safari 9 displays by default, as it has not updated the implied `min-width` change. We'll cover flex item width in greater depth in "The `flex` Property" on page 70.

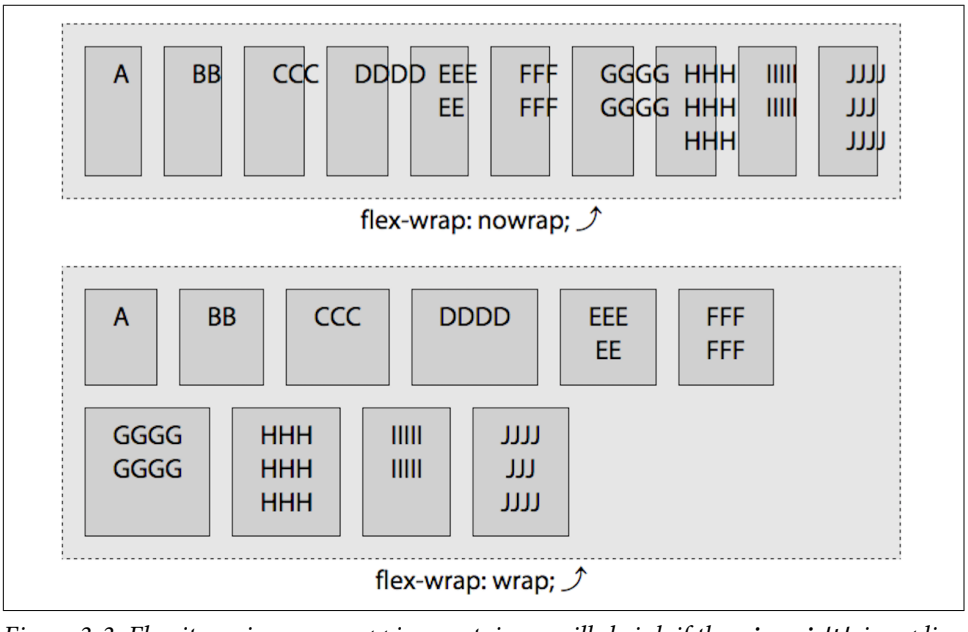


Figure 3-3. Flex items in non-wrapping containers will shrink if the `min-width` is explicitly set to `0`, which is the default in Safari 9 ▶

Flex Item–Specific Properties

While the flex items' alignment, order, and flexibility are to some extent controllable via properties set on their parent flex container, there are several properties that can be applied to individual flex items for more granular control.

The `flex` shorthand property, along with its component properties of `flex-grow`, `flex-shrink`, and `flex-basis`, control the flexibility of the flex items. The `align-self` helps control a flex item's alignment. The `order` property provides for more granular control of the visual ordering of individual or groups of flex items. All of these properties are discussed in the following sections.

The defining aspect of flex layout is the ability to make the flex items "flex": altering their width or height to fill the available space in the main dimension. A flex container distributes free space to its items proportional to their flex grow factor, or shrinks them to prevent overflow proportional to their flex shrink factor.

Defining the `flex` shorthand property on a flex item, or defining the individual properties that make up the shorthand, enables developers to define the grow and shrink factors. If there is excess space, you can tell the flex items to grow to fill that space. Or not. If there isn't enough room to fit all the flex items within the flex container at their defined or default sizes, you can tell the flex items to shrink proportionally to fit into the space. Or not. This is all done with the `flex` property, which is a shorthand property for `flex-grow`, `flex-shrink`, and `flex-basis`. While these three subproperties can be used separately, it is highly recommended to always use the `flex` shorthand.

We will briefly introduce the `flex` shorthand property, then dive deeper into the three longhand values that make it up. It is important to fully understand what each component is doing so you can effectively use the `flex` shorthand property.

The `flex` Property

The `flex` shorthand, a flex item property, is made up of the `flex-grow`, `flex-shrink`, and `flex-basis` properties, which define the flex growth factor, the flex shrink factor, and the flex basis, respectively.

flex

Values:	`<flex-grow> <flex-shrink>? || <flex-basis> ] | none | auto`

Initial value:	`flex-grow: 1;`
	`flex-shrink: 1;`
	`flex-basis: 0%;`

Applies to:	Flex items (children of flex containers)
Inherited:	No
Percentages:	Valid for `flex-basis` value only, relative to element's parent's inner main-size

The `flex` property specifies the components of a flexible length: the "length" of the flex item being the length of the flex item along the main-axis (see "Understanding axes" on page 25). When a box is a flex item, `flex` is consulted instead of the main-size (`height` or `width`) property of the flex item to determine the size of the box. The "components" of the `flex` property include the flex growth factor, flex shrink factor,

and the flex basis. If the target of a selector is not a flex item, applying the `flex` property to it will have no effect.

The flex basis determines how the flex growth and shrink factors are implemented. As its name suggests, the `flex-basis` component of the flex shorthand is the basis on which the flex item determines how much it can grow to fill available space or how much it should shrink to fit all the flex items when there isn't enough space. It's the initial size of each flex item, and can be restricted to that specific size by specifying 0 for both the growth and shrink factors:

```
.flexItem {
    flex: 0 0 200px;
}
```

In the preceding code snippet, the flex item will have a main-size of exactly 200 px, as the flex basis is 200 px, and it is neither allowed to grow nor shrink.

It is important to understand the three components that make up the `flex` shorthand property, which is the property you should be employing. The reason it is advised to use `flex` rather than its three individual subproperties is because the spec (and therefore the browser) resets any missing `flex` values to sensible defaults, which are not always the individual property defaults.

To ensure you fully grok `flex`, let's deep dive into its three components.

The order of `flex` is important, with the first float number being the growth factor.

The `flex-grow` Property

The `flex-grow` property defines whether a flex item is allowed to grow when there is available space, and, if it is allowed to grow and there is available space, how much will it grow proportionally relative to the growth of other flex item siblings.

flex-grow

Values:	`<number>`
Initial value:	`flex-grow: 0; 1` when omitted as part of flex shorthand.
Applies to:	Flex items (children of flex containers)
Inherited:	No
Animatable:	Yes

The value `flex-grow` and of the `flex-grow` portion of the `flex` shorthand is always a number. Negative numbers are not valid. Float values, as long as they are greater than 0, are valid.

The value specifies the growth factor, which determines how much the flex item will grow relative to the rest of the flex item siblings as the flex container's free space is distributed. If there is any available space within the flex container, the space will be distributed proportionally among the children with a nonzero positive growth factor based on the various values of those growth factors.

For example, with the `750px` wide horizontal flex container with three flex items all set to `width: 100px`, as in the four examples in Figure 3-4, you can dictate none, one, two, or all three of the flex items to grow to fill the extra `450px` of available space.

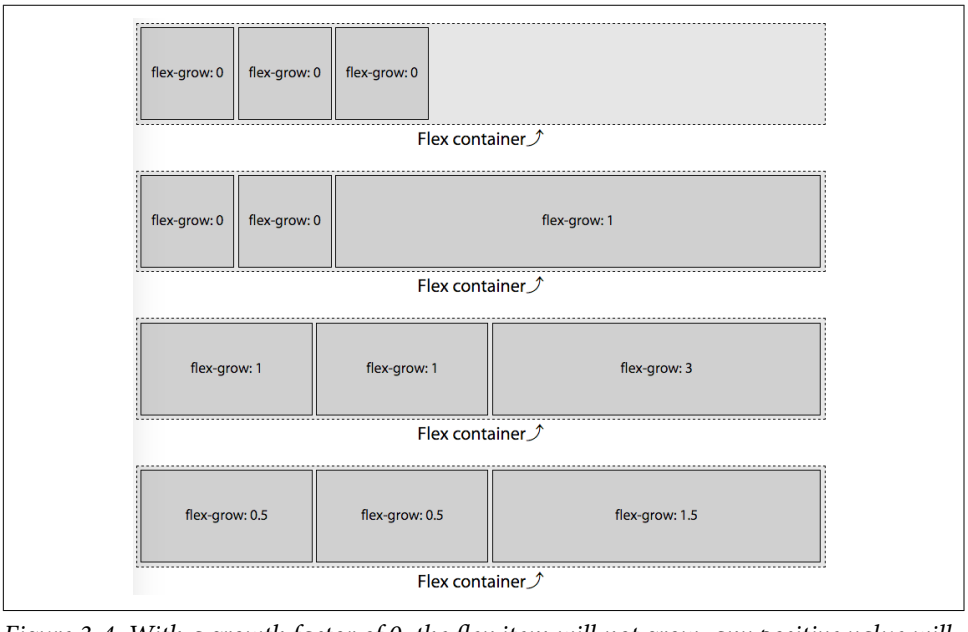


Figure 3-4. With a growth factor of 0, the flex item will not grow; any positive value will allow the item to grow proportionally to the value

As noted in "`min-width`" on page 68, if no width or flex basis is set, the flex basis defaults to `auto`, meaning each flex item basis is the width of its nonwrapped content. `auto` is a special value: it defaults to content unless the item has a width set on it, at which point the flex-basis becomes that width. The `auto` value is discussed in "`auto`" on page 90. Had we not set the width, in this example scenario, with our smallish font size, we would have had more than 450 px of distributable space along the main-axis.

The examples in this section describe the basics of the growth factor. The main-size of a flex item is impacted by the available space, the growth factor of all the flex items, as well as the flex basis of the item. We have yet to cover flex-basis. We will revisit both the growth and shrink factors and these examples when we learn about flex basis.

Note that the amount of distributable space depends on the basis. In Figure 3-4, because the width of the flex items was set to 100 px each and no basis was set, we have 450 px of distributable space. When a different width or no width is declared, or if a basis is set, the distributable space is different, as shown in Figure 3-6.

Any positive value greater than `0` for the growth factor means the flex item can grow to fill a portion or all of the available space. You can tell the items not to grow with a growth factor of `0`, as demonstrated in the first example of Figure 3-4. You can mix and match: growing some flex items and not others, as demonstrated in the second example.

In the 4 examples in Figure 3-4, because the flex container is 750 px wide and each of the 3 flex items has a `width: 100px` declared, there are 450 extra pixels to be distributed among the flex items that are allowed to grow:

```
750px - (3 * 100px) = 450px
```

We have 450 px of empty space in the main end direction as `align-items` defaults to `flex-start`. The first example has no growth factors set, so no flex items grew.

Non-Null Growth Factor

In the second example of Figure 3-4, the first two flex items, with a growth factor set to `0`, will not grow. Only the third flex item has a value greater than `0`, with `flex-grow: 1` set, which means the single element with a positive growth factor value will take up all the extra available space.

The first two flex items with flex growth factor of 0 will remain at `100px` wide, even if the content doesn't fit. Only the third flex item is allowed to grow, and therefore it must grow, taking up the extra 450 pixels to become 550 px wide.

The growth factor needs to be a positive, non-null float value for the flex item to grow. In this example the growth factor was 1, but had it been 0.000001 or 10000000, the layout would be the same.

Growing Proportionally Based on Growth Factor

If all items are allowed to grow, the excess space is distributed proportionally based on the flex growth factors. In the third example, with two flex items having a growth

factor of 1, and one flex item having a growth factor of 3, we have a total of five growth factors:

```
(2 x 1) + (1 x 3) = 5
```

With 5 growth factors, and a total of 450 px needing to be distributed, each growth factor is worth 90 px:

```
450px / 5 = 90px.
```

Before being allowed to grow based on individual flex item's growth factor, the flex items were each 100 px wide as set forth by the `width` property. With each growth factor being 90 px, we have 2 flex items with a width of 190 px each and the last flex item has a width of 370 px:

```
100px + (1 x 90px) = 190px
100px + (3 x 90px) = 370px
```

In the last example, we have a total of 2.5 growth factors:

```
(2 x 0.5) + (1 x 1.5) = 2.5
```

With 2.5 growth factors, and a total of 450 px needing to be filled, each growth factor is worth 180 px:

```
450px / 2.5 = 180px
```

Again, the default flex item size was 100 px, with the flex basis defaulting to `auto`, leading us to have the exact same layout as in the second example. This demonstrates how the distribution of extra space is proportional. Again we see 2 flex items with a width of 190 px and the last flex item having a width of 370 px:

```
100px + (0.5 x 180px) = 190px
100px + (1.5 x 180px) = 370px
```

Had we declared growth factors of 0.1, 0.1, and 0.3, respectively, or 25, 25, 75, the layout would have been identical.

Growth Factor with Different Widths

As mentioned, the available space is distributed proportionally among the flex items in each flex row based on the flex growth factors, without regards to the original, underlying widths.

In Figure 3-5, in the second example, we have flex items that are 100 px, 250 px, and 100 px wide, with growth factors of 1, 1, and 3, respectively, in a container that is 750 px wide. This means we have 300 px of extra space to distribute among a total of 5 growth factors. Each growth factor is therefore 60 px, meaning the first and second flex items, with a growth factor of 1, will each grow by 60 px, and the last will grow by 180 px as the growth factor is set to 3.

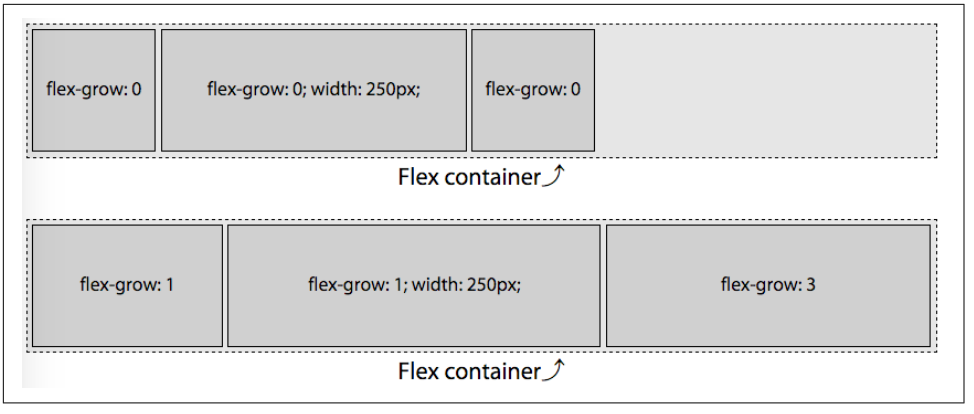


Figure 3-5. The available space is evenly distributed to each growth factor; any positive value will allow the item to grow proportionally to the value ▶

The available space, growth factors, and width of each growth factor are:

```
Available space: 750px - (100px + 250px + 100px) = 300px
Growth factors:  1 + 1 + 3 = 5
Width of each growth factor: 300px / 5 = 60px
```

When flexed, the width based on their original width and growth factors become:

```
item1 = 100px + (1 * 60px) = 160px
item2 = 250px + (1 * 60px) = 310px
item3 = 100px + (3 * 60px) = 280px

item1 + item2 + item3 = 160px + 310px + 280px = 750px
```

Growth Factors and the `flex` Property

The `flex` property takes up to three values—the growth factor, shrink factor, and basis. The first positive non-null float value, if there is one, is the growth factor. When the growth factor is omitted in the shorthand, it defaults to 1. Otherwise, if neither `flex` nor `flex-grow` is included, it defaults to 0.

In the second example in Figure 3-4, because we declared a value for `flex-grow` only, the flex basis was set to `auto`, as if we had declared:

```
#example2 flex-item {
  flex: 0 1 auto;
}
#example2 flex-item:last-child {
  flex: 1 1 auto;
}
```

Declaring `flex-grow` is strongly discouraged. Rather, declare the growth factor as part of the `flex` shorthand.

Had we declared `flex: 0`, `flex: 0.5;`, `flex: 1;`, `flex: 1.5;`, and `flex: 3;` in the examples in Figure 3-4 instead of ill-advisedly declaring `flex-grow` values, the flex basis would be set to 0%, and the width distribution would have been very different, as shown in Figure 3-6:

```
#example2  flex-item {
  flex: 0 1 0%;
}
#example2  flex-item:last-child {
  flex: 1 1 0%;
}
```

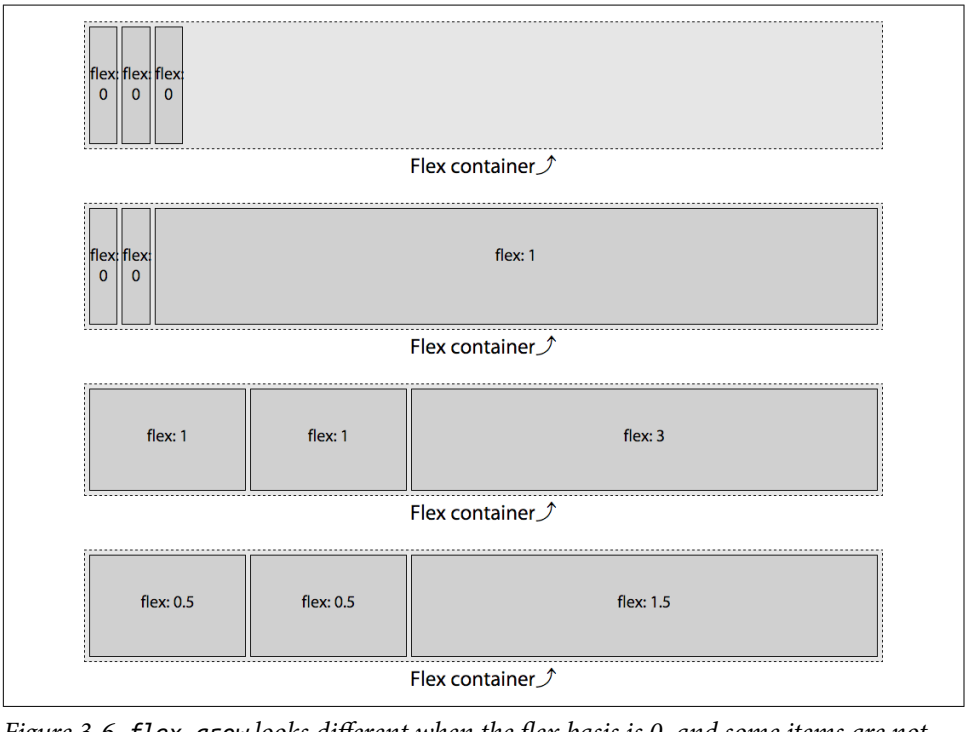


Figure 3-6. `flex-grow` looks different when the flex basis is 0, and some items are not allowed to grow ▶

As the shrink factor defaults to 1 and the basis defaults to 0%, the following CSS is identical to the preceding code:

```
#example2  flex-item {
  flex: 0;
}
#example2  flex-item:last-child {
  flex: 1;
}
```

You may notice something odd: the flex basis has been set to zero, and only the last flex item has a positive value for flex grow. Logic would seem that the widths of the 3 flex items should be 0, 0, and 750 px, respectively. But logic would also dictate that it makes no sense to have content overflowing its flex item if the flex container has the room for all the content, even if the basis is set to zero.

The specification authors thought of this quandary. When the `flex` property declaration explicitly sets or defaults the flex-basis to 0 px and a flex item's growth factor is 0, the length of the main-axis of the nongrowing flex items will shrink to the smallest width the content allows, or smaller. In our example, the width is the width of the widest word "flex:".

As long as a flex item has a visible overflow and no `min-width` (or `min-height` for vertical main-axes) explicitly set, the minimum width (or minimum height) will be the smallest width (or height) that the flex item needs to be able to fit the content or the declared `width` (or `height`), whichever is smaller. In the first and second examples in Figure 3-6, even though the growth factor is 0, the flex items don't shrink to 0. Rather, they shrink to the width of the widest nonbreakable word, as the word "flex-" is narrower than 100 px. Had `width: 10px` been set instead of `width: 100px`, the flex items would have narrowed down to whichever was smaller: the width of the word "flex-" or 10 px.

If all items are allowed to grow, and the flex basis for each flex item is 0%, all of the space—the entire 750 px rather than just excess space—is distributed proportionally based on the growth factors. In the examples the flex container is 750 px wide. In Figure 3-6, as the flex basis is zero, rather than auto, and with no positive shrink factors, the first two flex items are only as wide as their widest content (the word flex:), and all the extra space goes to the third flex item with a positive growth factor. This will make more sense after reading the `flex-shrink` section (see "The `flex-shrink` Property" on page 79). In the third example, with two flex items having growth factors of one, and one flex item having a growth factor of three, we have a total of five growth factors:

```
(2 x 1) + (1 x 3) = 5
```

With 5 grows factors, and a total of 750 px, each growth factor is worth 150 px:

```
750px / 5 = 150px.
```

While the default flex item size was 100 px, the flex basis of 0 overrides that, leaving us with 2 flex items at 150 px each and the last flex item with a width of 450 px:

```
1 x 150px = 150px
3 x 150px = 450px
150px + 150px + 450px = 750px
```

Similarly, in the last example of Figure 3-6, with two flex items having growth factors of 0.5, and one flex item having a growth factor of 1.5, we have a total of 2.5 growth factors:

```
(2 x 0.5) + (1 x 1.5) = 2.5
```

With 2.5 grows factors, and a total of 750 px, each growth factor is worth 300 px:

```
750px / 2.5 = 300px.
```

While the default flex item size was 100 px, the flex basis of 0% overrides that, leaving us with 2 flex items at 150 px each and the last flex item with a width of 450 px:

```
0.5 x 300px = 150px
1.5 x 300px = 450px
150px + 150px + 450px = 750px
```

This is different from declaring only `flex-grow`, as shown in Figure 3-7. Only declaring `flex-grow` means `flex-basis` defaults to auto. When set to auto, only the extra space, not all the space, is distributed proportionally. This lack of simplicity is why it is highly encouraged to always use the `flex` shorthand instead of `flex-grow`, `flex-shrink`, and `flex-basis` separately or not at all.

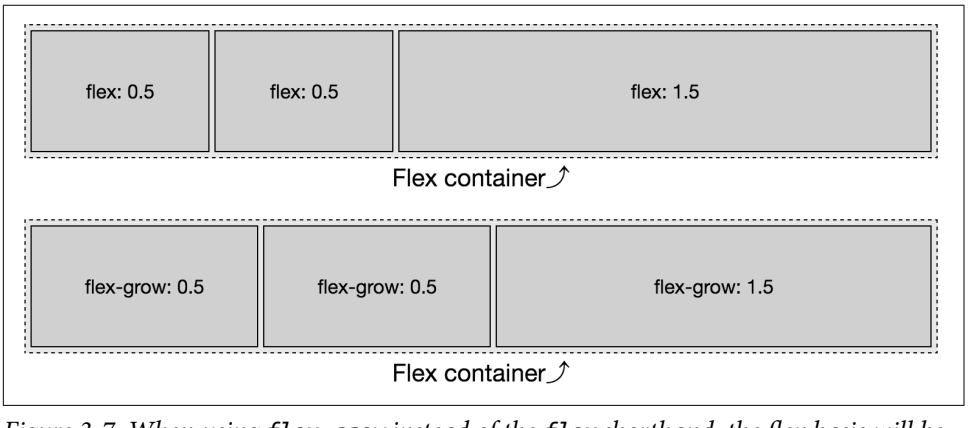


Figure 3-7. When using `flex-grow` instead of the `flex` shorthand, the flex basis will be `auto` instead of `0`.

While the available space is distributed proportionally based on the flex items' growth factor, the amount of available space is defined by the flex basis, which is discussed in a following section. Let's first cover the shrinking factor.

The `flex-shrink` Property

flex-shrink	
Values:	`<number>`
Initial value:	`1` on its own and as part of `flex` shorthand
Applies to:	Flex items (children of flex containers)
Inherited:	No
Animatable:	Yes

The `flex-shrink` portion of the `flex` shorthand property specifies the shrink factor. The shrink factor determines how much a flex item will shrink relative to the rest of the flex item siblings when there isn't enough space for them all to fit as defined by their content, basis, and other CSS properties. Basically, the shrink factor defines how the *negative space* is distributed, or how flex items should become narrower or shorter, when the flex container parent isn't allowed to otherwise grow or wrap.

Figure 3-8 is similar to Figure 3-4, with the flex items set to 300 px instead of 100 px. We have a 750 px-wide flex container with three 300 px-wide flex items. The total width of the 3 items is 900 px, meaning the content is 150 px wider than the parent flex container. If the items are not allowed to shrink or wrap (see "The `flex-wrap` Property" on page 23), they will burst out from the fixed-size flex container. This is demonstrated in the first example in Figure 3-8: those items will not shrink as they have a null shrink factor. Instead, the flex items overflow the flex container along the main-end side, as `justify-content` (described in "The `justify-content` Property" on page 36) defaults to `flex-start`.

In the second example in Figure 3-8, only the last flex item is set to be able to shrink. We forced the single element with a positive shrink factor to do all the shrinking necessary to enable all the flex items to fit within the container. With 900 px worth of content needing to fit into our 750 px container, we have 150 px of negative space. The 2 flex items with no shrink factor stay at 300 px wide. The third flex item, with the positive value for the shrink factor, shrinks down by 150 pixels, to be 150 px wide, enabling the 3 items to fit within the container. In this example the shrink factor was 1, but had it been 0.001 or 100, the layout would be the same.

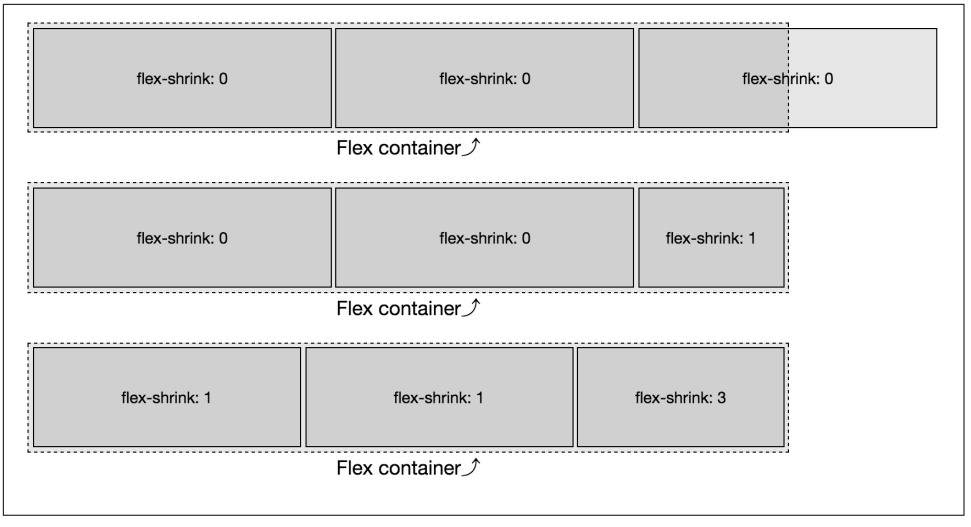


Figure 3-8. A flex shrink factor of 0 will not allow flex items to shrink; any positive value will enable the item to shrink proportionally relative to sibling flex items that are allowed to shrink on the same flex line ▶

When omitted in the shorthand `flex` property value or when both `flex` and `flex-shrink` are omitted, the shrink factor defaults to 1. Like the growth factor, the value is always a number. Negative numbers are not valid. Floats, as long as they are greater than 0, are valid.

In the third example, we provided positive shrink factors for all three flex items:

```
#example3 flex-item {
  flex-shrink: 1;
}
#example3 flex-item:last-child {
  flex-shrink: 3;
}
```

While we included the preceding code, our flex items will behave as if we had declared the following:

```
#example3 flex-item {
  flex: 0 1 auto;
}
f#example3 flex-item:last-child {
  flex: 0 3 auto;
}
```

This is likely not the layout you will be developing for production. So, use the `flex` shorthand.

If all items are allowed to shrink, as is the case here, the shrinking is distributed proportionally based on the shrink factor of the individual items that have a positive value set for that property.

When all items have the same basis, it's easy to figure out how they will shrink based on the values of their shrink factors. With a parent 750 px wide, and 3 flex items with a width of 300 px, there are 150 extra pixels that need to be shaved off of the flex items that are allowed to shrink. With two flex items having a shrink factor of 1, and one flex item having a shrink factor of 3, we have a total of five shrink factors:

```
(2 x 1) + (1 x 3) = 5
```

With 5 shrink factors, and a total of 150 px needing to be shaved off all the flex items, each shrink factor is worth 30 px:

```
150px / 5 = 30px.
```

The default flex item size was 300 px, leading us to have 2 flex items with a width of 270 px each and the last flex item having a width of 210 px, which total 750 px. In shrinking proportionally based on their shrink factor, we have dictated how they shrink to fit in the space allotted:

```
300px - (1 x 30px) = 270px
300px - (3 x 30px) = 210px

270px + 270px + 210px = 750px
```

The flex items will shrink to 270 px, 270 px, and 210 px, respectively, as long as the content (like media objects or nonwrappable text) within each flex item is not wider than 270 px, 270 px, or 210 px, respectively:

```
flex-item {
  flex: 1 0.25 auto;
}
flex-item:last-child {
  flex: 1 0.75 auto;
}
```

The preceding code produces the same outcome: while the numeric representation of the shrink factors are different, they are proportionally the same.

A positive shrink value does not mean a flex item will necessarily shrink even if there isn't room available in the parent flex container for it and its siblings. If the flex item contains content that cannot wrap or otherwise shrink in the main-dimension, the flex item will not shrink.

For example, if the first flex items contain a 300 px–wide image or an URL of characters wider than 300 px, the 150 px of negative space would be distributed among the four available shrink factors:

```
item1 = 300px - (0 x 37.5px) = 300.0px
item2 = 300px - (1 x 37.5px) = 262.5px
item3 = 300px - (3 x 37.5px) = 187.5px
```

In this case, the first item would be 300 px, with the 150 px of negative space distributed proportionally based on the shrink factors of the second and third flex items. That first flex item can not shrink, and other flex items can shrink, therefore it will not shrink, as if it had a null shrink factor. We have 4 unimpeded shrink factors for 150 px of negative space, with each shrink factor being worth 37.5 px. The flex items are 300 px, 262.5 px, and 187.5 px respectively, for a total of 750 px.

Had the image or URL in the first flex item been 296 px wide, that first flex item would have been able to shrink by 4 px. We would have then distributed the 146 of negative space among the 4 remaining shrink factors for flex items that were 296 px, 263.5 px, and 190.5 px wide, respectively.

If all 3 flex items contained non-wrappable URLs, or media 300 px or wider, the 3 flex items would not shrink, appearing similar to the first example in Figure 3-8.

All the items had the same width or flex basis, therefore this is easy to calculate. Had they not had the same widths, this equation would be more complex. This only works this way because all the flex items had the same flex bases.

Proportional Based on Width and Shrink Factor

The preceding code example was fairly simple because all the flex items started with the same width. But what if the widths were different? What if the first and last flex items had a width of 250 px and the middle flex item had a width of 500 px?

Flex items shrink proportionally relative to both the shrink factors and the flex item width, with the width often being `content`, the width of the flex item's content with no wrapping. In Figure 3-9, we are trying to fit 1,000 pixels into a 750 px-width flex container. We have an excess of 250 px to be removed from five shrink factors. If this were a flex-grow concern, we would simply divide 250 px by 5, allocating 50 px per growth factor. If we shrank that way, we would see 200 px, 550 px, and 100 px wide flex items, respectively. But that's not what we see.

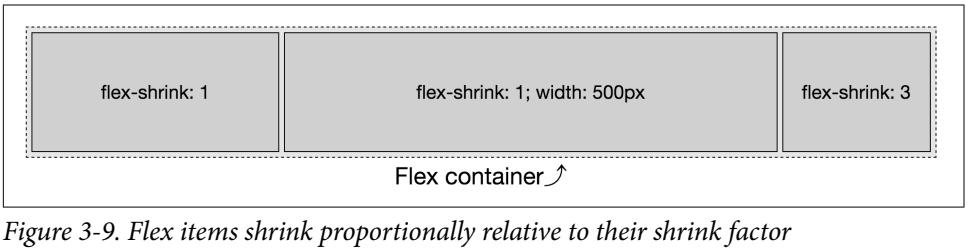


Figure 3-9. Flex items shrink proportionally relative to their shrink factor

Instead we have 250 px of negative space to proportionally distribute. To get the shrink factor proportions, we divide the negative space by the actual space times their shrink factors:

$$ShrinkPercent = \frac{NegativeSpace}{((Width1 * ShrF1) + \ldots + (WidthN * ShrFN))}$$

Using this equation, we learn the shrink factor percent:

```
= 250px / ((250px * 1) + (500px * 1) + (250px * 3))
= 250px / 1500px
= 0.166666667
```

When we reduce each flex item by 16.67% times the shrink factor, we end up with flex items that are reduced by:

```
item1 = 250px * (1 * 16.67%) = 41.67px
item2 = 500px * (1 * 16.67%) = 83.33px
item3 = 250px * (3 * 16.67%) = 125px
```

We get flex items that are 208.33 px, 416.67 px, and 125 px wide, respectively.

In the Real World

Allowing flex items to shrink proportionally like this allows for responsive objects and layouts that can shrink proportionally without breaking.

For example, you can create a three-column layout that smartly grows and shrinks without media queries, as shown on a wide screen in Figure 3-10 and narrower screen in Figure 3-11):

```
nav {
  flex: 0 1 200px;
  min-width: 150px;
}
article {
  flex: 1 2 600px;
}
aside {
  flex: 0 1 200px;
  min-width: 150px;
}
```

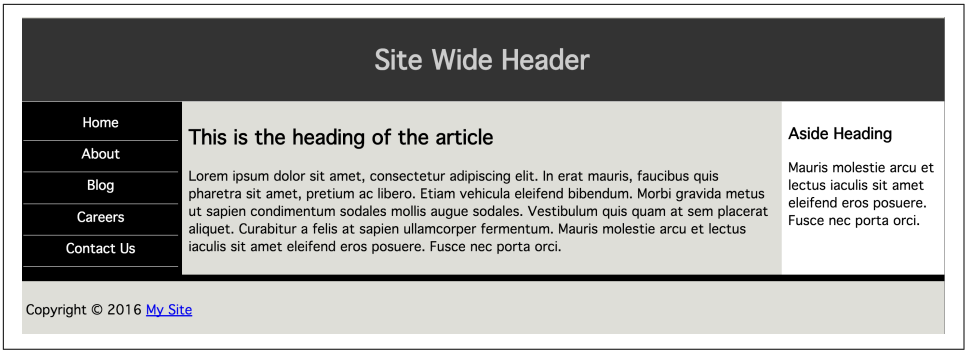


Figure 3-10. By setting different values for growth, shrink, basis, and min-width, you can create responsive layouts, with or without media queries

Figure 3-11. With `growth`, `shrink`, `basis`, and `min-width` set, you can create responsive layouts that look great on narrower screens, without needing a multitude of media queries

In this example, if the viewport is greater than 1,000 px, only the middle column grows because only the middle column was provided with a positive growth factor. We also dictated that below the 1,000 px-width mark the right column shrinks twice as fast as the left 2 columns, using `min-width` to ensure the columns never shrinks below its narrowest word or that minimum width, whichever is greater.

Differing Bases

With a null shrink factor, if no width or basis is set on a flex item, its content will not wrap. When we have more content than could neatly fit on one line, a positive shrink value enables the content to wrap. Shrink factors enable proportional wrapping. Because shrinking is proportional based on shrink factor, if all the flex items have similar shrink factors, the content should wrap over a similar number of lines.

Unlike in the previous examples in this chapter, in the two examples in Figure 3-12, the flex items do not have a declared width. Rather, the width is based on the content—the width defaults to `auto`, as does the flex basis, as if `flex-basis: content` were set (see "The `flex-basis` Property" on page 87).

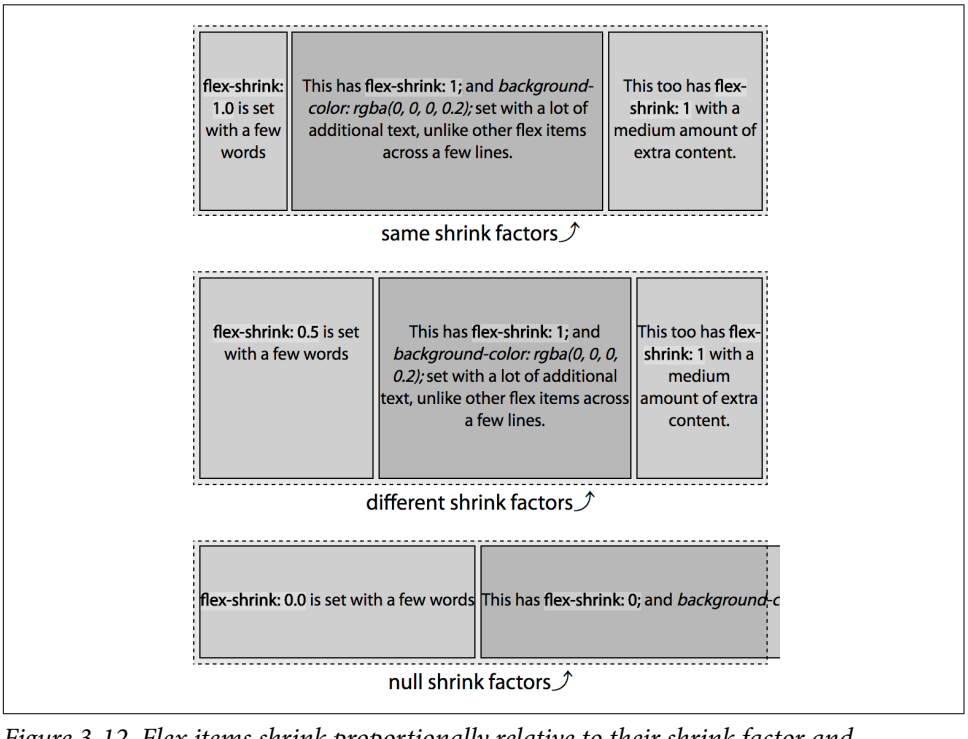


Figure 3-12. Flex items shrink proportionally relative to their shrink factor and content ▶

You'll note in the first example, all content wraps over four lines. In the second example, the first flex item, with a shrink factor half of value of the other flex items, wraps over half the number of lines. This is the power of the shrink factor.

In the third example, with a null shrink factor, the text doesn't wrap and the flex items overflow the container.

Because the `flex` property's shrink factor reduces the width of flex items proportionally, the number of lines of text in the flex items will grow or shrink as the width shrinks or grows, leading to similar height content within sibling flex items when the shrink factors are similar.

In the examples, the content of the flex items on my device are 280 px, 995 px, and 480 px, respectively—which is the width of the nonwrapping flex items in the third example (as measured by the developer tools, then rounded to make this example a little simpler). This means we have to fit 1,755 px of content into a 520 px-wide flex container by shrinking the flex items proportionally based on their shrink factor. This means we have 1,235 px of negative space to proportionally distribute.

Obviously you can't rely on web inspector tools to figure out shrink factors for production. We're going through this exercise to understand how shrink factors work. If minutia isn't your thing, feel free to jump to "The `flex-basis` Property" on page 87.

Because flex items shrink proportionally, based on the width of their content, in our example, the single line of text flex items will end up with the same, or approximately the same, number of lines.

We didn't declare a width, therefore we can't simply use 300 px as the basis as we did in the previous examples. Rather, we distribute the 1,235 px of negative space proportionally based on the widths of the content—280 px, 995 px, and 480 px, respectively. We determine 520 is 29.63% of 1,755. To determine the width of each flex item with a shrink factor of 1, we multiply the content width of each flex item by 29.63%:

```
item1 = 280px * 29.63% =  83px
item2 = 995px * 29.63% = 295px
item3 = 480px * 29.63% = 142px

item1 +  item2 + item3  =  83px + 295px + 142px = 520px
```

With the default of `align-items: stretch;` (see "The `align-items` Property" on page 45), your three-column layout would have, by default, created three columns of equal height. By using a uniform shrink factor, you can dictate that the actual content of these three flex items be of approximately equal height: though, by doing this, the width of those columns will not be uniform. The width of the flex items is the purview of `flex-basis`.

In our second example in Figure 3-12, the flex items don't all have the same shrink factor. The first flex item will, proportionally, shrink half as much as the others. We start with the same widths: 280 px, 995 px, and 480 px, respectively, but the shrink factors are 0.5, 1.0, and 1.0, respectively. As we know the widths of the content, the shrink factor (*X*) can be found mathematically:

```
(0.5X * 280px) + (1X * 995px) + (1X * 480px) = 1235px
1615X = 1235px
```

```
X = 1235px / 1615
X = 0.7647
```

We can find the final widths now that we know the shrink factor. If the shrink factor is 76.47%, it means that `item2` and `item3` will be 23.53% of their original widths, and `item1`, because it has a 0.5 shrink factor, will be 61.76% of its original width:

```
item1 = 280px * 0.6176 = 173px
item2 = 995px * 0.2354 = 234px
item3 = 480px * 0.2354 = 113px

item1 + item2 + item3 =  173px + 234px + 113px = 520px
```

The total combined widths of these 3 flex items is 520 px.

Adding in varying shrink and growth factors makes it all a little less intuitive. That's why you likely want to always declare the flex shorthand, preferably with a width or basis set for each flex item.

If this doesn't make sense yet, don't worry, we'll cover a few more examples of shrinking as we discuss `flex-basis`.

The `flex-basis` Property

Again, while we are covering the `flex-basis` property individually here so you fully understand it, the CSS working group encourages declaring the basis as part of the `flex` shorthand property instead of on its own. `flex` resets to common usage (rather than defaulting) the grow, shrink, and basis values if any of them are not declared within the shorthand. When `flex-basis` is set, instead of `flex`, flex items can shrink but will not grow, as if `flex: 0 1 <flex-basis>` were set.

Seriously, use `flex` instead of declaring the three longhand values separately. We're only covering the shorthand here because 1) this is the definitive guide, so we have to, and 2) so you fully understand what the `flex-basis` of the `flex` shorthand property does. By this point, I hope you've been convinced.

flex-basis

Values:	`content` | `<width>`
Initial value:	`auto`
Applies to:	Flex items (children of flex containers)
Inherited:	No
Percentages:	Relative to flex container's inner main-size

As we've already seen, a flex item's size is impacted by its content and box-model properties and can be reset via the three components of the `flex` property. The `flex-basis` component of the `flex` property defines the initial or default size of flex items, before extra or negative space is distributed—before the flex items are allowed to grow or shrink according to their growth and shrink factors, which we just described.

The basis determines the size of the content box, impacted by `box-sizing`. By default, when an element is not a flex item, the size is determined by the size of its parent, content, and box-model properties. When no size properties are explicitly declared or inherited, the size defaults to its individual content, border, and padding, which is 100% of the width of its parent for block-level elements.

The `flex-basis` property accepts the same length value types as the `width` and `height` properties—like `5vw`, `12%`, and `300px`—or the key terms `auto`, `initial`, `inherit`, and `content`.

None of the flex properties are inherited by default, but you can tell a flex item to inherit the flex-basis from its parent with the `inherit` value.

The global values of `initial` resets the flex basis to the initial value of `auto`, so you might as well declare `auto`. In turn, `auto` evaluates to the `width` (or `height`) if declared. The `main-size` keyword used to do this: it was part of an older specification, but was deprecated. If the value of `width` (or `height`) is set to `auto`, then the value is evaluated to `content`.

`content`

The `content` keyword value is not supported in most browsers at the time of this writing, with the exception of Microsoft Edge 12+, but is equal to the width or height of the content. When `content` is used and supported, the basis is the size of the flex item's content—the value being the main-size of the longest line of content or widest (or tallest) media object.

Until support is complete, `flex-basis: content;` can be easily polly-filled as it is the equivalent of declaring `flex-basis: auto; width: auto;` on that flex item, or `flex-basis: auto; height: auto;` if the main-dimension is vertical. Unfortunately, using `content` in the shorthand in nonsupporting browsers invalidates the entire `flex` declaration. The entire declaration is invalidated when any value is not understood as per the specification.

The value of `content` is basically what we saw in the third example in Figure 3-12, and is shown in Figure 3-13.

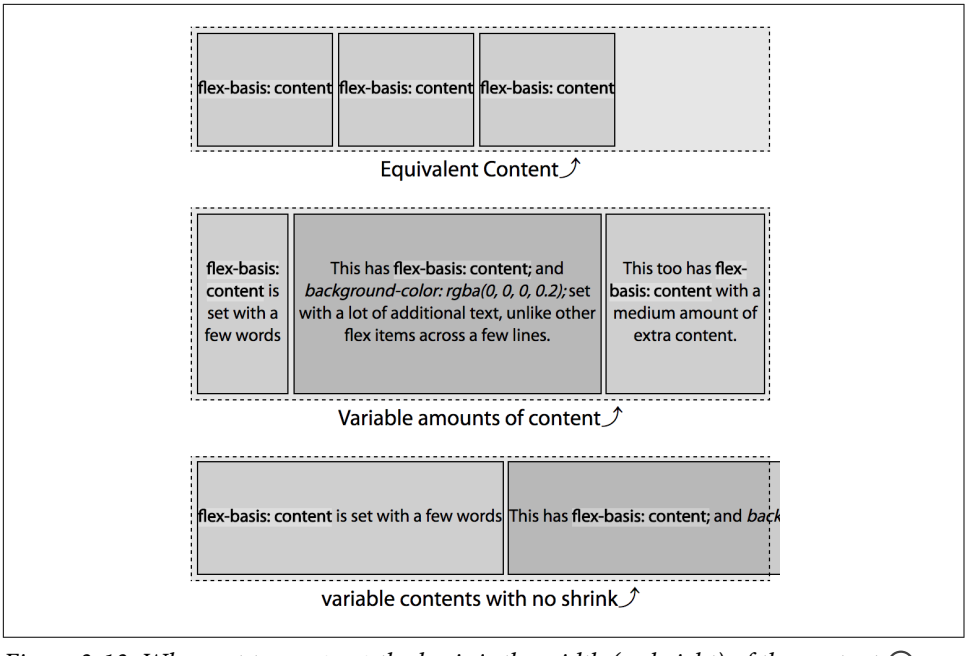


Figure 3-13. When set to `content` the basis is the width (or height) of the content

In the first and third examples in Figure 3-13, the width of the flex item is the size of the content; and the basis is also that width. In the first example, the flex items width and basis are approximately 132 px. The total width of the 3 flex items side by side is 396 px, fitting neatly into the parent container.

In the third example, we have set a null shrink factor: this means the flex items cannot shrink, so they won't shrink or wrap to fit into the fixed-width flex container parent. Rather, they are the width of their nonwrapped text. That width is the value of the flex basis. The flex items' width and basis are approximately 309 px, 1,037 px, and 523 px, respectively. You can't see the full width of the second flex item or the third flex item at all, but they're in the chapter files.

The second example contains the same content as the third example, but the flex items are defaulting to `flex-shrink: 1`: the text in this example wraps because the flex items can shrink. So while the width of the flex item is not the width of the content, the flex basis—the basis by which it will proportionally shrink—is the width of the content: 309 px, 1,037 px, and 523 px, respectively, as measured with developer tools.

When the shrink factors are the same, because flex items shrinking is proportional based on the content-width flex basis, they end up with the same or approximately the same number of lines.

Until the `content` value is supported everywhere, you can replicate it by setting (or defaulting) the width/height and basis to `auto`.

It is the same as setting `flex-basis: auto; width: auto;` or `flex-basis: auto; height: auto;`.

`auto`

When set to `auto`, or omitted, the `flex-basis` is the main-size of the node had the element not been turned into a flex item. For length values, flex-basis resolves to the `width` or `height` value, with the exception that when the value of the `width` or `height` is `auto`, the value resolves to `content`.

If the basis is explicitly set to `auto`, or omitted and therefore defaults to `auto`, and all the flex items can fit within the parent flex container, the flex items will be their pre-flexed size, as seen in Figure 3-14. If the flex items don't fit into their parent flex container, the flex items within that container will shrink proportionally based on their nonflexed main-sizes, unless the shrink factor is null.

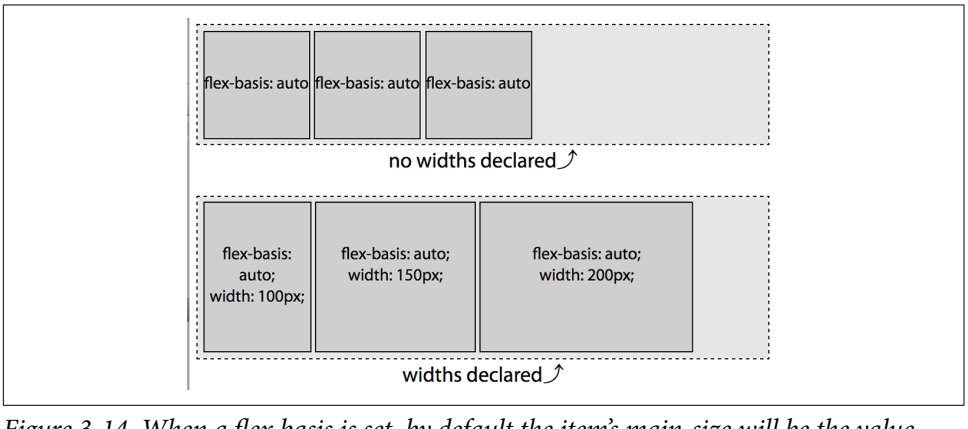


Figure 3-14. When a flex basis is set, by default the item's main-size will be the value declared

When there are no other properties setting the main-size of the flex items (there's no `width` or even `min-width` set on these flex items), and `flex-basis: auto;` or `flex: 0 1 auto;` is set, the flex items will only be as wide as they need to be for the content to fit, as seen in the first example in Figure 3-14. In this case, they are the width of the text "flex-basis: auto", which in this case, with this font, is approximately 110 px. The flex items are their pre-flexed size, as if set to `display: inline-block;`. In this

example, they're grouped at main-start because the flex container's `justify-content` defaults to `flex-start`.

In the second example in Figure 3-14, each of the flex items has flex basis of `auto` and a declared width. The main-size of the nodes had the elements not been turned into a flex items would be 100 px, 150 px, and 200 px. In using `flex-basis: auto` we are telling it to use these underlying box-model properties to determine the flex basis.

There's a little bit more to understanding of how `auto` works with the underlying width. While the examples in this section used percentages and `auto`, when we discussed the growth and shrink factors earlier, the flex items had underlying widths of 100 px and 300 px respectively. Because the basis was not explicitly set to a length, the `width` value was the basis in those scenarios.

Default Values

When neither `flex-basis` nor `flex` is set, the flex item's main-size is the pre-flex size of the item, as their default value is `auto`.

In Figure 3-15 two things are happening: the flex bases are defaulting to `auto`, and the shrink factor of each item is defaulting to 1. That means the bases are being set to the values of the `width` properties: 100 px, 200 px, and 300 px in the first example and 200 px, 400 px, and 200 px in the second example, and they are all able to shrink. For each, the flex basis is their individual `width` value. As the combined widths are 600 px and 800 px, which are both greater than the main-size of the 540 px-wide containers, they are all shrinking proportionally to fit.

In the first example, we are trying to fit 600 px in 540 px, so each flex item will shrink by 10%, resulting in flex items that are 90 px, 180 px, and 270 px. In our second examples, we are trying to fit 800 px into 540 px, making the flex items 135 px, 270 px, and 135 px.

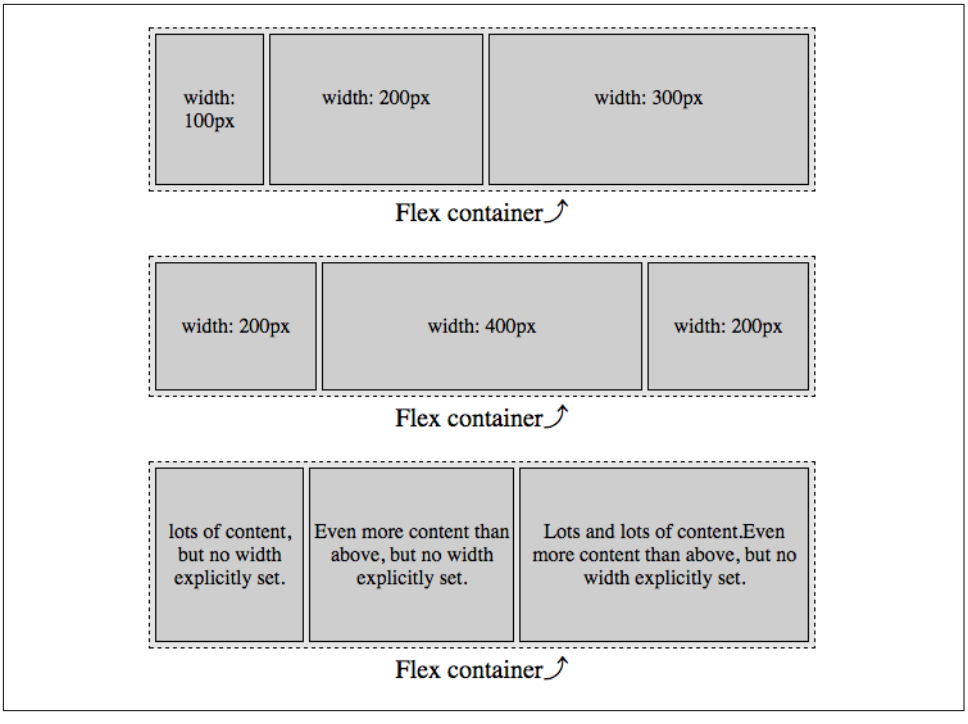


Figure 3-15. When no `flex` properties are set, the flex item's main-size will be the pre-flex size of the item ▶

Length Units

In the previous examples, the basis defaulted to the declared widths of the various flex items. We can use the same length units for our flex-basis value as we do for `width` and `height`.

When there are both flex-basis and width values, the basis trumps the width. Let's add bases values to the first example from Figure 3-15. The flex items include the following CSS:

```
flex-container {
  width: 540px;
}
item1 {
  width: 100px;
  flex-basis: 300px;  /* flex: 0 1 300px; */
}
item2 {
  width: 200px;
  flex-basis: 200px;  /* flex: 0 1 200px; */
}
item3 {
```

```
  width: 300px;
  flex-basis: 100px;  /* flex: 0 1 100px; */
}
```

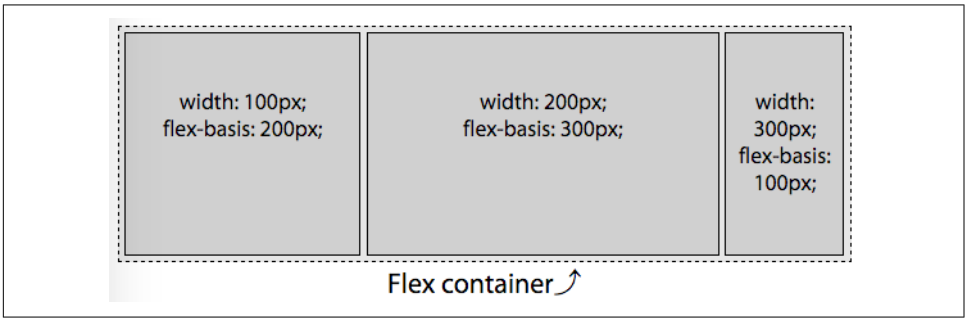


Figure 3-16. Note the main-size of the flex item is proportion to the `flex-basis` property, not the `width` property

The widths are overridden by the bases. The flex items shrunk down to 270 px, 180 px, and 90 px, respectively.

While the declared basis can override the main-size of flex items, the size can be impacted by other properties, such as `min-width`, `min-height`, `max-width`, and `max-height`. These are not ignored.

Length units: percentages

Percentage values for `flex-basis` are relative to the size of the main dimension of the flex container.

We've already seen the first example in Figure 3-17. I am including it here to recall that the width of the text "flex-basis: auto" in this case, with my OS, browser, and installed fonts, is approximately 110 px wide. In this case only, declaring `flex-basis: auto` looks the same as writing `flex-basis: 110px`:

```
flex-container {
  width: 540px;
}
flex-item:last-child {
  flex: 0 1 100%;
}
```

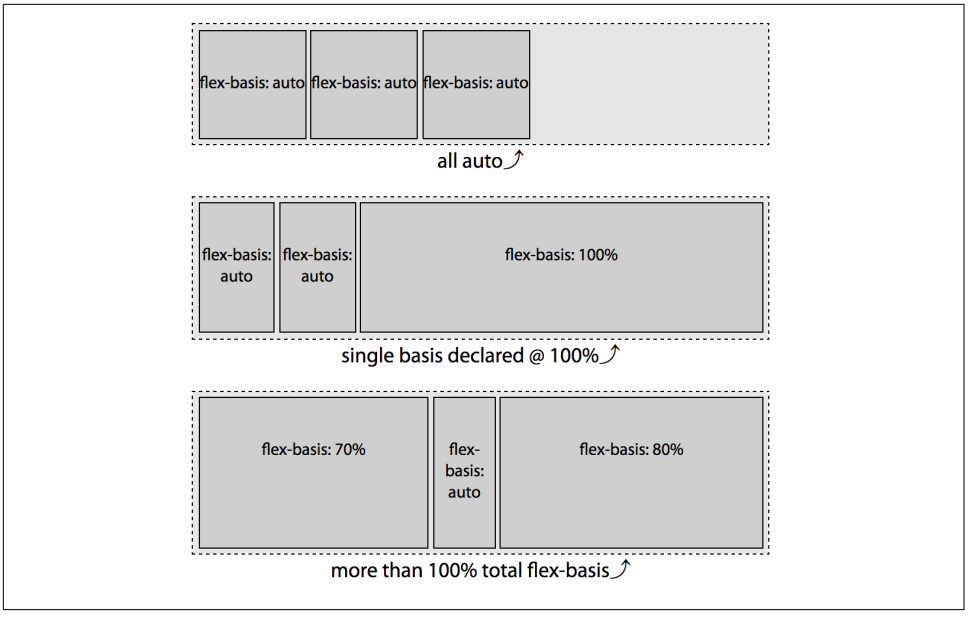


Figure 3-17. The percentage value for flex-basis is relative to the width of the flex container

In the second example in Figure 3-17, the first two have a flex basis of auto with a default width of auto, which is as if their flex basis were set to `content`. As we've noted previously, the flex-basis of the first 2 items ends up being the equivalent of 110 px as the content is 110 px wide.

The last item has its flex basis set to 100%. The percentage value is relative to the parent, which is 540 px. As the third item, with a basis of 100%, is not the only flex item within the non-wrapping flex container, it will not grow to be 100% of the width of the parent flex container unless its shrink factor is set with a null shrink factor (meaning it can't shrink) or if it contains nonwrappable content that is as wide or wider than the parent container. This is not the case. Each flex item contains only wrappable content, and all three flex items are able to shrink as all have a default shrink factor of 1.

Remember: when the flex basis is a percent value, the main-size is relative to the parent, which is the flex container.

If the content is indeed 110 px wide, and the container is 540 px wide (ignoring other box-model properties for simplicity's sake), we have a total of 760 px to fit in a 540 px

space in this second example. With three flex items with flex bases equivalent to 110 px, 110 px, and 540 px, respectively, we're trying to fit 760 px of content into a 540 px-width container. We have 220 px of negative space to distribute proportionally. The shrink factor is:

```
Shrink factor = 220px / 760px = 28.95%
```

Each flex item will be shrunk by 28.95%, becoming 71.05% of the width they would have been had they not been allowed to shrink. We can figure the final widths:

```
item1 =  110px * 71.05% =  78.16px
item2 =  110px * 71.05% =  78.16px
item3 =  540px * 71.05% = 383.68px

item1 + item2 + item3 = 78.16px + 78.16px + 383.68px = 540px
```

These numbers hold true as long as the flex items can be that small: as long as none of the flex items contain media or nonbreaking text wider than 78.16 px or 383.68 px. This is the widest these flex items will be as long as the content can wrap to be that width or narrower. "Widest" because if one of the other two flex items can't shrink to be as narrow as this value, they'll have to absorb some of that negative space.

In our third example, the `flex-basis: auto` item wraps over three lines. The CSS for this example is the equivalent of:

```
flex-container {
  width: 540px;
}
item1 {
  flex: 0 1 70%;
}
item2 {
  flex: 0 1 auto;
}
item3 {
  flex: 0 1 80%;
}
```

We declared the `flex-basis` of the 3 flex items to be 70%, auto, and 80%, respectively. Remembering that "auto" is the width of the nonwrapping content, which in this case is approximately 110 px and our flex container is 540 px, the bases are the equivalent of:

```
item1 = 70% * 540px = 378px
item2 = widthOfText("flex-basis: auto") = 110px
item3 = 80% * 540px  =  432px
```

In this third example we have an item with a basis of 70%. This means the basis is 70% of the parent's 540 px width, or 378 px. The second item is set to `auto`, which in this case means 110 px because of the width of the content. Lastly, we have flex item with a basis of 80%, meaning 80% of 540 px, or 432 px. When we add the widths of

these 3 flex items, they have total combined width of 920 px, which needs to fit into a flex container 540 px wide. We have 380 px of negative space to remove proportionally among the 3 flex items. To figure out the ratio, we divide the available width of our flex container by the sum of widths of the flex items that they would have if they couldn't shrink:

```
Proportional Width = 540px / 920px = 0.587
```

Because the shrink factors are all the same, this is fairly simple. Each item will be 58.7% of the width it would be if it had no flex item siblings:

```
item1 = 378px * 58.7% = 221.8px
item2 = 110px * 58.7% =  64.6px
item3 = 432px * 58.7% = 253.6px
```

What happens when the container is a different width? Say, 1,000 px? The flex basis would be 700 px (70% x 1,000 px), 110 px, and 800 px (80% x 1,000 px), respectively, for a total of 1,610 px:

```
Proportional Width = 1000px / 1610px = 0.6211


item1 = 700px * 62.11% = 434.8px
item2 = 110px * 62.11% =  68.3px
item3 = 800px * 62.11% = 496.9px
```

Because with a basis of 70% and 80%, the combined bases of the flex items will always be wider than 100%, no matter how wide we make the parent, all 3 items will always shrink.

If the first flex item couldn't shrink, as shown in Figure 3-18, it would be 70% of the width of the parent—378 px in this case. The other 2 flex items will shrink proportionally to fit into the remaining 30%, or 162 px. In this case, you would expect widths to be 378 px, 32.875 px, and 129.125 px. As the text "basis:" is wider than that (at 42 px on my device) we get 378 px, 42 px, and 120 px.

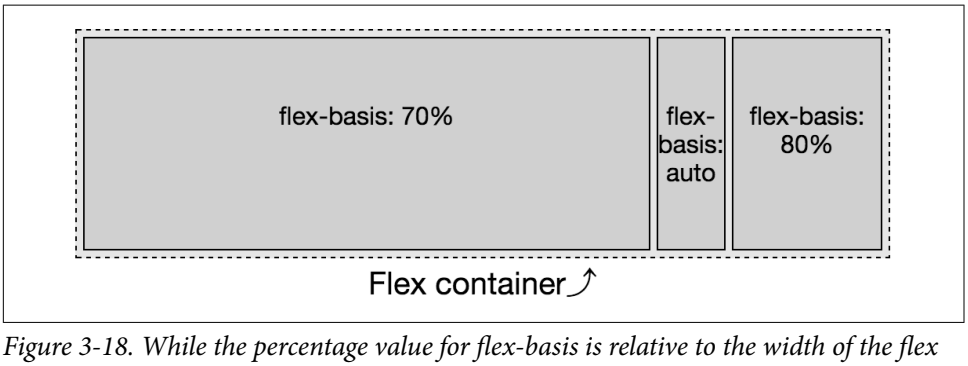


Figure 3-18. While the percentage value for flex-basis is relative to the width of the flex container, the main-size is impacted by its siblings

Testing this out on your device will likely have slightly different results, as the width of the text "flex-basis: auto" may not be the same on your screen.

Zero Basis

If neither the `flex-basis` property nor the `flex` shorthand is included at all, the basis defaults to `auto`. When the `flex` property is included, but the basis component of the shorthand is omitted from the shorthand, the basis defaults to 0%. While on the surface you might think the two values of `auto` and `0` may seem similar, the `0` value is actually very different, and may not be what you expect. Once you understand it, you'll realize it's actually fairly intuitive.

The growth and shrink factors have completely different outcomes depending on whether the basis is set to `0` versus `auto`, as shown in Figure 3-19.

It is only the extra space that is distributed proportionally based on the flex growth factors. In the case of `flex-basis: auto;`, the basis is the main size of their content. If the basis of each of the flex items is `0`, the "available" space is all the space, or main-size of the parent flex container. This "available" space is distributed proportionally based on the growth factors of each flex item. In the case of a basis of 0%, the size of the container is divided up and distributed proportionally to each flex item based on their growth factors—their default original main-size as defined by `height`, `width`, or `content`, is not taken into account, though `min-width`, `max-width`, `min-height`, and `max-height` do impact the flexed size.

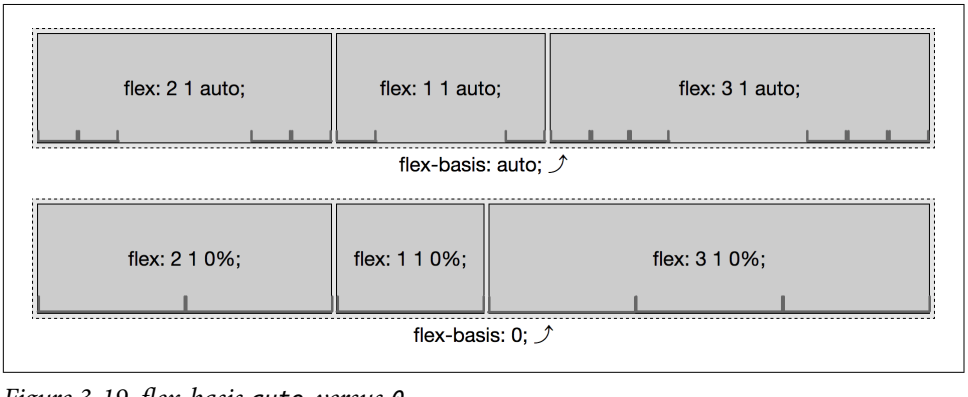


Figure 3-19. flex-basis `auto`, versus `0`

As shown in this example, when the basis is `auto`, it is just the extra space that is divided up proportionally and added to each flex item set to grow. Again, assuming the width of the text "flex: X X auto" is 110 px, in the first examples we have 210 px to distribute among 6 growth factors, or 35 px per growth factor. Our flex items are 180 px, 145 px, and 215 px wide, respectively.

In the second example, when the basis is 0, all 540 px of the width is distributable space. With 540 px of distributable space between 6 growth factors, each growth factor is worth 90 px. Our flex items are 180 px, 90 px, and 270 px wide, respectively. While our middle flex item is 90 px wide, the content in this example is narrower than the 110 px from our other examples, so the flex item didn't wrap.

The `flex` Shorthand Property

Now that we have a fuller understanding of the properties that make up the `flex` shorthand, always use the `flex` shorthand. There are some commonly used shorthand values, including `initial`, `auto`, `none`, and the use of an integer, usually `1`, meaning the flex item can grow. Let's go over all these values.

Common `flex` Values

The common flex values are four flex values providing the most commonly desired effects:

`flex: initial`
: This value sizes flex items based on the width/height properties, while allowing shrinking.

`flex: auto`
: This flex value also sizes flex items based on the width/height properties, but makes them fully flexible, allowing both shrinking and growing.

`flex: none`
: This value again sizes flex items based on the width/height properties, but makes them completely inflexible: they can't shrink or grow.

`flex: n`
: This value doesn't care about the `width`/`height` values as the shrink factor is set to 0. In this case, the flex item's size is be proportional to the flex factor `n`.

`flex: initial`

`Initial` is a global CSS keyword, which means `initial` can be used on all properties to represent the property's initial value:

```
flex: initial;
flex: 0 1 auto;
```

These lines are the same: `flex: initial` is the equivalent of `flex: 0 1 auto`. This means the flex items' size will be based on its own width and height properties, or based on the contents if the main-size isn't explicitly set (set or defaulting to `auto`). If

the flex container is not large enough for the flex items, the flex items can be shrunk, but the flex items will not grow even if there extra distributable space available.

Declaring `flex: initial` sets a null growth factor, a shrink factor of `1`, and sets the flex bases to `auto`. In Figure 3-20, we can see the effect of the `auto` flex bases. In the first two examples, the basis of each flex item is `content`—with each flex item having the width of the single line of letters that make up their content. In the last 2 examples, the flex bases of all the items are equal at 50 px, since `width: 50px` has been applied to all the flex items. The `flex: initial` declaration sets the `flex-basis` to `auto`, which we previously learned means it is the value of the `width` (or `height`), if declared, or `content` if not declared.

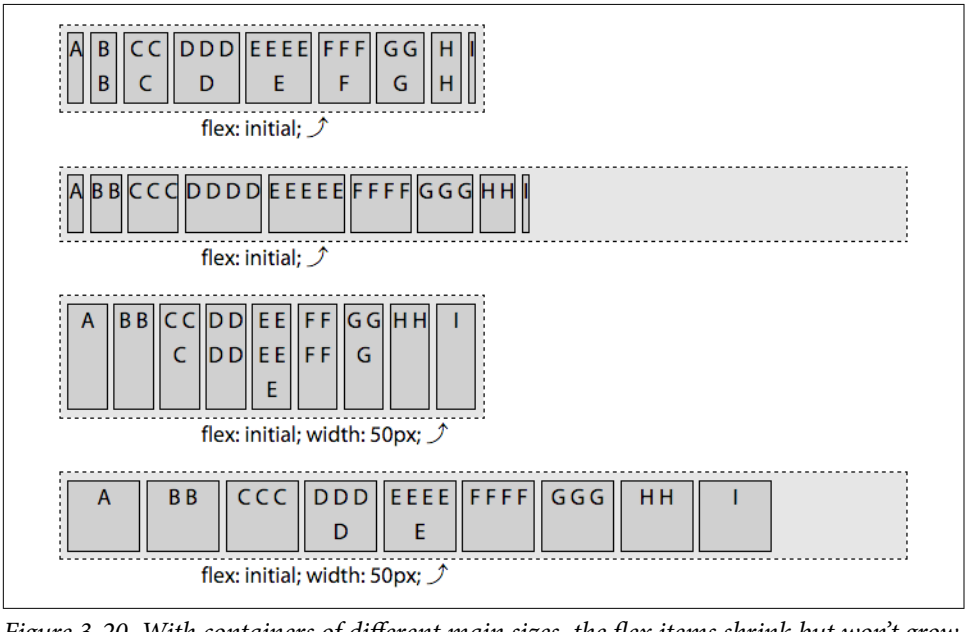


Figure 3-20. With containers of different main sizes, the flex items shrink but won't grow when `flex: initial` is set on the flex items

In the first and third examples in Figure 3-20, we see how, if the flex container is too small to fit all the flex items at their default main-size, the flex items will shrink, with all the flex items fitting within the parent flex container. In these examples, the combined flex bases of all the flex items is greater than the main-size of the flex container. In the first example, the amount by which each flex item shrinks are all different. They all shrink proportionally based on their shrink factor. In the third example, with each flex item's flex-basis being equal at 50 px, all the items shrink equally.

In the first example, you'll note the last flex item, with the single narrow capital letter `I`, is the narrowest flex item in the group, followed by `A` (They would have been the

same had we used a monospaced font family). `B` and `H` are wider than `A` and `I`, even with the wrapping caused by the flex items shrinking, because their bases are based on three characters (two letters and a space) rather than one. The shrinking results in these flex items having only one letter per line of text, but they are still wider than the flex items that have a single character as their bases.

The flex items with more characters are wider because flex-basis is based on the width of the content. `A` and `I` are one letter wide. The basis for `B` and `H` are based on the width of two space-separated letters. Similarly, `D` and `F` are wider than `C` and `G`. In the first example, with `flex: initial`, `E` ends up being the widest as it has the most characters creating the widest content of all the bases.

In the second and fourth examples, the flex items all fit, so there is no shrinkage. When `flex: initial` is set, the growth factor is null, so the flex items can't grow, and therefore don't grow, to fill up their container.

Flex items, by default, are grouped at main start, as `flex-start` is the default value of for the `justify-content` property. This is only noticable when the combined main-size of the flex items on a flex line are smaller than the main-size of the flex container.

`flex: auto`

Setting `flex: auto;` on a flex item is the same as setting `flex: 1 1 auto`. The following two statements are equivalent:

```
flex: auto;
flex: 1 1 auto;
```

`flex: auto` is similar to `flex: initial`, but makes the flex items flexible in both directions: they'll shrink if there isn't enough room to fit all the items within the container, and they'll grow to take up all the extra space within the container if there is distributable space. The flex items absorb any free space along the main-axis.

You'll note the first and third examples of Figure 3-21 are identical to the examples in Figure 3-20, as the shrinking and bases are the same. However, the second and fourth examples are different, as when `flex: auto` is set, the growth factor is not null, and the flex items therefore grow incorporating all the extra available space.

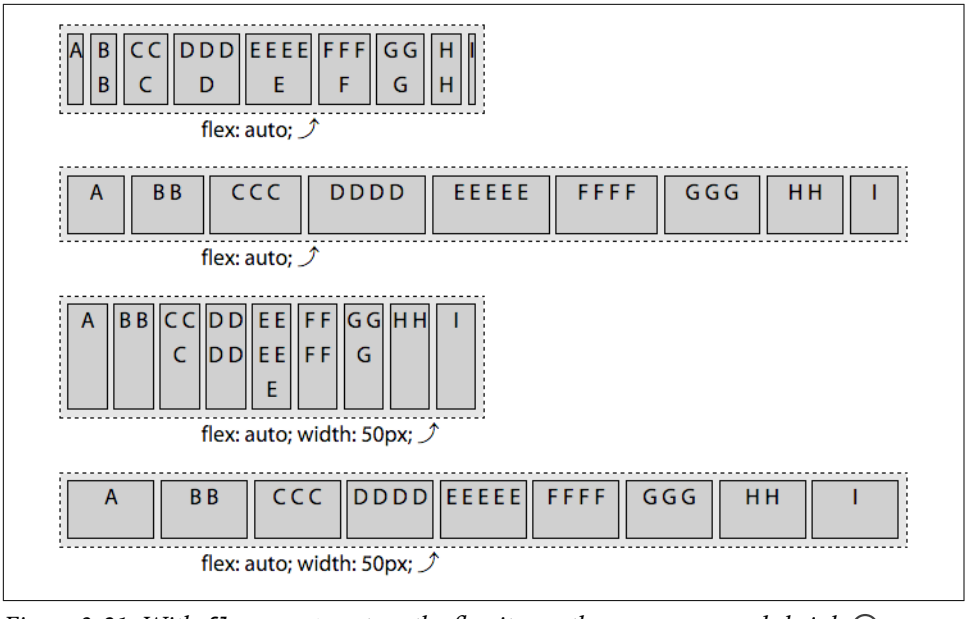


Figure 3-21. With `flex: auto` set on the flex items, they can grow and shrink ▶

`flex: none`

Setting `flex: none` is equivalent to setting `flex: 0 0 auto`, making the following two lines of CSS equivalent:

```
flex: none;
flex: 0 0 auto;
```

The `flex: none` items are inflexible: the flex items will neither shrink nor grow.

As demonstrated in the first and third examples of Figure 3-22, if there isn't enough space, the flex items overflow the flex container. This is different from `flex: initial` and `flex: auto`, which both set a positive shrink factor.

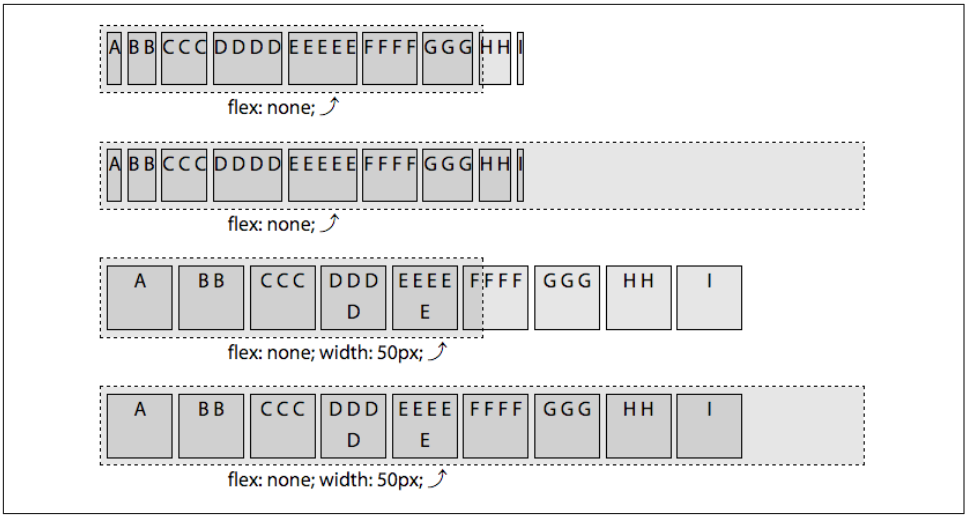


Figure 3-22. With `flex: none`, flex items will neither grow nor shrink

The basis resolves to `auto`, meaning each flex item's main-size is determined by the main-size of the node had the element not been turned into a flex item: the flex-basis resolves to the `width` or `height` value. If that value resolved to `auto`, the basis would be the main-size of the content. In the first two examples, the basis—and the width, since there is no growing or shrinking—is the width of the content. In the third and fourth examples, the width and basis are all 50 px.

`flex: n`

When the values of the `flex` property is a single, positive numeric value, that value will be the growth factor, while the shrink factor will default to 0, and the basis will default to 0%:

```
flex-grow: n;
flex-shrink: 0;
flex-basis: 0%;
```

The following two CSS declarations are equivalent:

```
flex: 3;
flex: 3 0 0%;
```

Declaring `flex: 3` is the same as declaring `flex: 3 0 0%`. This makes the flex item on which it is set flexible: it can grow. The shrink factor is actually moot, as the `flex-basis` was set to `0%` and the flex item, even if a positive shrink factor were declared, cannot be less than 0 px wide or tall.

In the first two examples in Figure 3-23, all the flex items have a flex growth factor of 3. All the flex items in each example grew to be the same width. While in Figures

3-20, 3-21, and 3-22, the flex items shrank to fit the flex container parent, when you set `flex: n`, where `n` is any positive float, the basis is 0, so they didn't "shrink"; they just grew equally from 0px wide until the sum of their main dimension grew to fill the container's main dimension. With all the flex items having a basis of 0, 100% of the main dimension is distributable space. The main-size of the flex items are wider in this second example as the wider parent has more distributable space.

Any value for `n` that is greater than 0, even 0.1, means the flex item can grow. The `n` is the growth factor. When there is available space to grow, if only one flex item has a positive growth factor, that item will take up all the available space. If all the items can grow, the available extra space will be distributed proportionally to each flex item based on to their growth factor. In the case of all the flex items having `flex: n` declared, where `n` is a single or variety of positive numbers, the growth will be proportional based only on `n`, not on the underlying main size, as the basis for each flex item is `0`.

In the last three examples of Figure 3-23, there are six flex items with `flex: 0`, `flex: 1`, `flex: 2`, `flex: 3`, `flex: 4`, and `flex: 5` declared, respectively. These are the growth factors for the flex items, with each having a shrink factor of 0 and a basis of 0. The main-size of each is proportional to the specified flex growth factor.

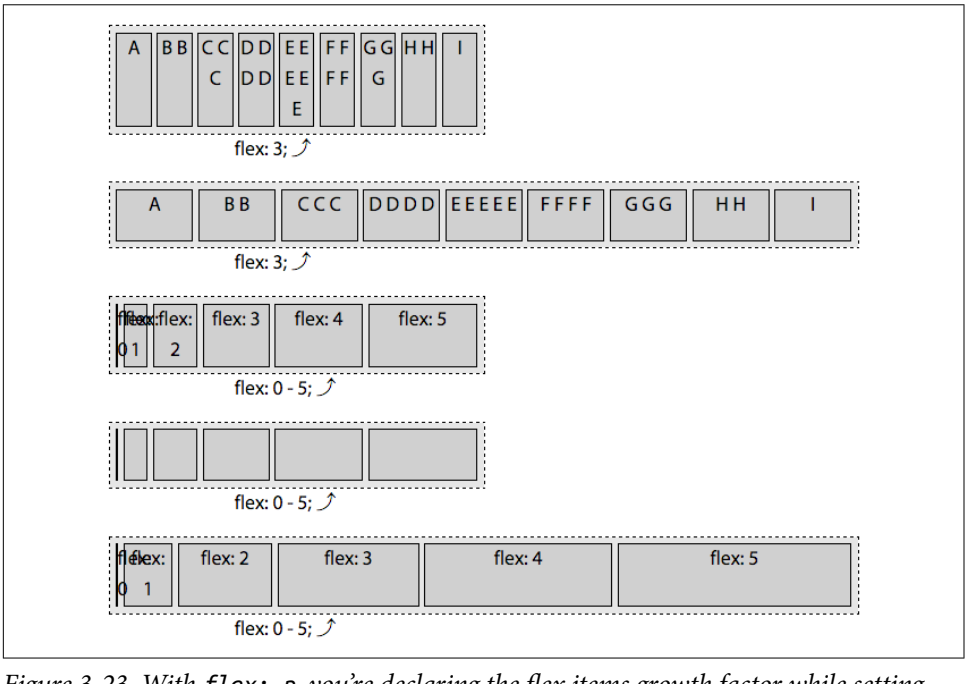


Figure 3-23. With `flex: n`, you're declaring the flex items growth factor while setting the flex basis to zero ▶

There are 15 total growth factors distributed across 6 flex items. The narrow flex container is 300 px wide, meaning each growth factor is worth 20 px. This means the widths of the flex items will be 0, 20 px, 40 px, 60 px, 80 px, and 100 px, for a total of 300 px. The last example has a width of 600 px, meaning each growth factor is worth 40 px. This leaves us with flex items that are 0, 40 px, 80 px, 120 px, 160 px, and 200 px, for a total of 600 px.

We added a bit of padding, margins, and borders on the example to make the visuals more pleasing. For this reason, the leftmost flex item, with `flex: 0` declared, is visible: it has a 1 px border making it visible even though it's 0 px wide.

If we had declared a width on these flex items, it would have made no difference. Setting `flex: 0`, `flex: 5`, or any other values for `n` in `flex: n` sets the `flex-basis` to 0%. You'll see no difference between Figures 3-23 and 3-24, other than the "width: 50px" in the paragraph below each flex container.

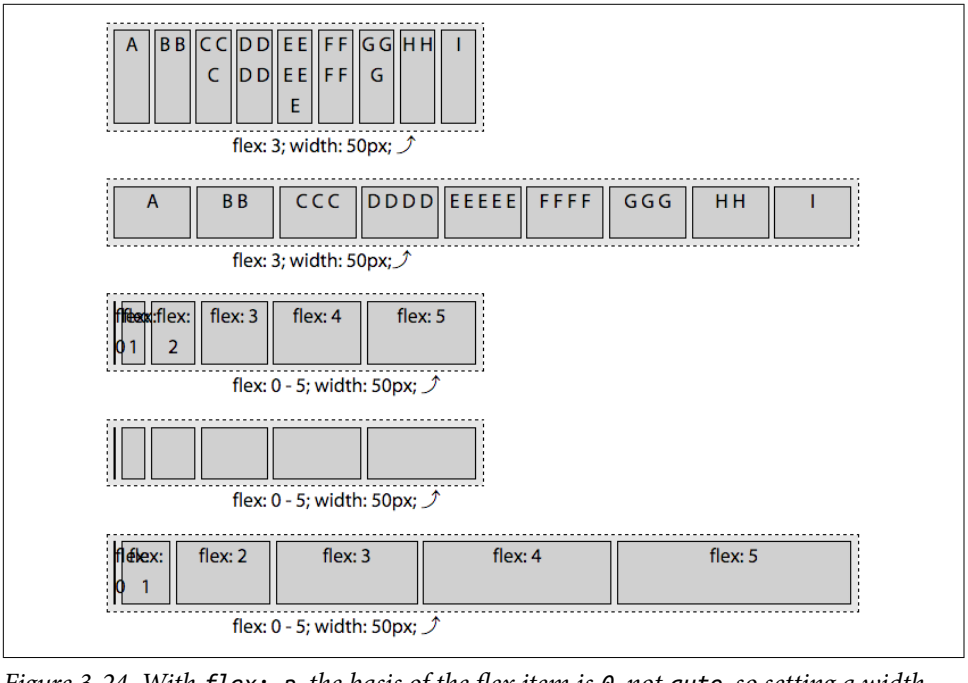


Figure 3-24. With `flex: n`, the basis of the flex item is `0`, not `auto`, so setting a width will not alter the appearance

Custom `flex` Values

`flex: initial`, `flex: auto`, `flex: none`, and `flex: n` are preset values. While these are provided as they are the most commonly used values, they may not meet all of

your needs, and they don't provide as much flexibility as defining your own custom flex values.

When `flex-basis` is set to `auto`, as shown in the first example in Figure 3-25, the main-size of each flex item is based on the content. Looking at the growth factor, you may have thought the middle item would be the narrowest as the growth factor is the smallest, but there is no "growing." In this example there is no available distributable space. Rather, they're all shrinking equally as they all have the same shrink factor and the basis of `auto` means the basis is the width of the content. In this case the flex items' content are all of equal height, as the size of each is based on the flex basis, rather than the growth factor.

In the second example in Figure 3-25, the basis is `0%`, so the main-size is determined by the various growth factors. In this case, the first flex item, with `flex: 2 1 0%` is twice as wide as the second flex item with `flex: 1 1 0%`. This second item, in turn, has a main-size that is one-third as wide as the last flex item with `flex: 3 1 0%;`.

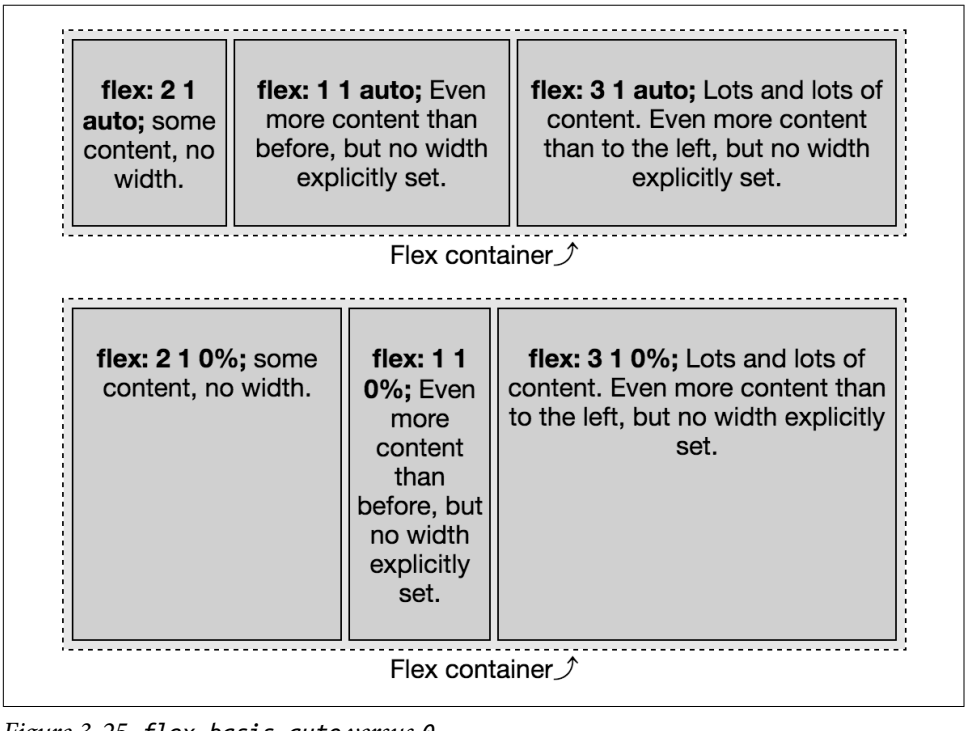


Figure 3-25. `flex-basis auto` versus `0`

We included a shrink factor in this example to make it parallel to the the first example, but it was completely not necessary. When using a 0 (or 0%) basis, the shrink fac-

tor has no impact on the main-size of the flex items. We could have included any of the three statements in each declaration:

```
item1 {
  flex: 2 1 0%;
  flex: 2 0 0%;
  flex: 2;
}
item2 {
  flex: 1 1 0%;
  flex: 1 0 0%;
  flex: 1;
}
item3 {
  flex: 3 1 0%;
  flex: 3 0 0%;
  flex: 3;
}
```

In these two examples, there was no `width` set on the flex items. By default, flex items won't shrink below their minimum content size (the length of the longest word or fixed-size element). To change this, set the `min-width` or `min-height` property. For the examples in Figure 3-25, had `min-width: 200px` been set on the second flex item, in both cases, the width would have been 200 px.

How can this help in practice? In theory the similar height columns we get with `auto` seem like a good idea, but in reality controlling the widths of the columns in a multiple-column layout is likely a greater priority, as demonstrated by the difference in the two layouts in Figures 3-26 and 3-27.

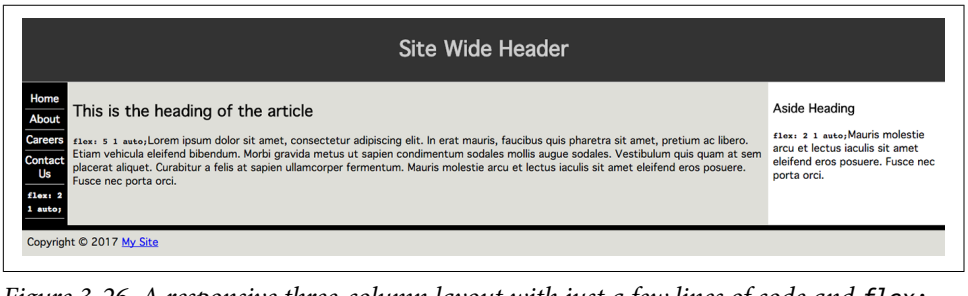


Figure 3-26. A responsive three-column layout with just a few lines of code and `flex: auto` ▶

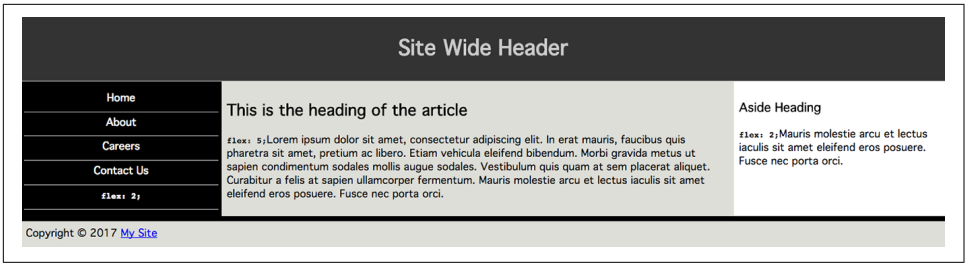


Figure 3-27. Altering the proportion of a responsive layout by altering the flex value of the flex items ▶

The first example's flex basis, Figure 3-26, is based on the content. For the navigation, the longest line, and therefore the basis, is the link with the longest content. If the `article` or `aside` contained a line break in the long and longer paragraphs, their widths would change.

The following CSS was used to create the second example:

```
@media screen and (min-width: 500px) {

    main {
      display: flex;
    }
    nav {
      order: -1;
      flex: 2;
      min-width: 150px;
    }
    article {
      flex: 5;
    }
    aside {
      flex: 2;
      min-width: 150px;
    }

}
```

In the second example, Figure 3-27, in viewports that are 500 px or wider, the `nav` and `aside` will always each be 22% of the width of the main area, unless they contain a component that is wider than 22%. The main article will be 56% of the width. You can include a `min-width` on any or all of the items, like we did on the navigation and aside, to ensure the layout never gets too narrow. You can use media queries to let the sections of the layout drop on very narrow screens. So many options!

We'll cover this example again when we discuss the `order` property "The `order` property" on page 111. Quickly: in the markup the `nav` is last, and it will come last on a viewport that is less than 500 px wide. On wider screens that have room for the three

columns, it will come first in appearance. Screen readers will still read it in source order (except in Firefox at the moment). Tabbing will still make it appear last.

Sticky Footer with `flex`

Obviously, `flex` can be used for a lot of designs. A common one is creating a sticky footer. A sticky footer is a fixed-height footer is at the bottom of the document, sticking to the bottom of the browser window even when the document would otherwise be shorter than the browser window.

As you can see in Figure 3-28, when the document gets taller, the footer sticks to the bottom of the window:

```
<body>
  <header>...</header>
  <main>
    <nav>
      <a href="/">Home</a>
      <a href="/about">About</a>
      <a href="/blog">Blog</a>
      <a href="/jobs">Careers</a>
      <a href="/contact">Contact Us</a>
    </nav>
     <article>... </article>
     <aside>... </aside>
  </main>
  <footer>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/blog">Blog</a>
    <a href="/jobs">Careers</a>
    <a href="/contact">Contact Us</a>
  </footer>
</body>
```

This is a very common way of marking up a typical website. With a few lines of CSS, we can create the sticky footer:

```
body {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}
main {
  flex: 1;
}
footer {
  height: 3rem;
}
```

If you define the body to be a flex container at least as tall as the browser window—as we did with `min-height: 100vh;`—and define the main area of the page to be able to grow, when you give the footer a defined height, it will always be that height, stuck to the bottom of the document, appearing stuck to the bottom of the browser window, even if the main area doesn't have enough content to fill the window and grows to fill the available space.

Without a `flex` value, the footer will neither shrink nor grow, as the `main` area will be doing the growing.

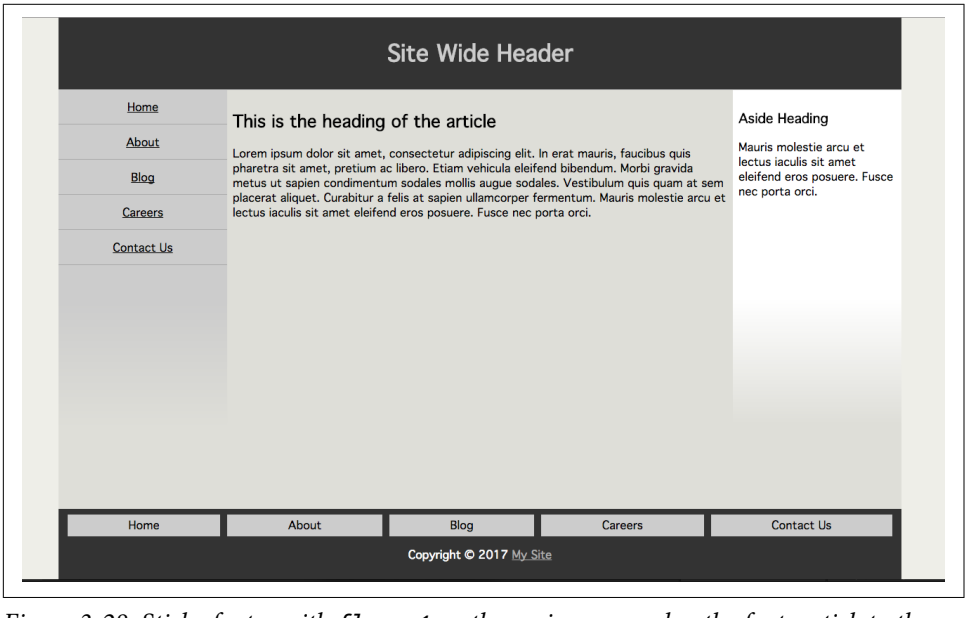


Figure 3-28. Sticky footer with `flex: 1` on the main area makes the footer stick to the bottom even when there is a lot of unused space ▶

The main area will not shrink to be less tall than the left navigation even though it theoretically could shrink with a `flex: 1;` being set on `main`. The reason? In this example, both the `<main>` and `<nav>` elements are both flex items and flex containers, with the navigation item not being shrinkable. Had we set:

```
nav a {
   flex: 0 1 0%;
 }
```

the links would have been able to shrink, and, with a flex-basis of 0, would have shrunk, enabling the `main` to shrink further, leaving room for the footer to be at its declared height.

The `align-self` Property

The `align-self` property is used to override the `align-items` property value on a per-flex-item basis.

align-items

Values:	`auto` | `flex-start` | `flex-end` | `center` | `baseline` | `stretch`
Initial value:	`auto`
Applies to:	Flex items
Inherited:	No
Percentages:	Not applicable
Animatable:	No

With the `align-items` property set on the flex container, you can align all the flex items of that container to the start, end, or center of the cross-axis of their flex line. The `align-self` property, which is set directly on the flex item, enables you to override the `aligns-items` property on a per-flex-item basis.

You can override the alignment of any individual flex item with the `align-self` property, as shown in Figure 3-29. The default value of `align-items` is `stretch`, which is why all the flex items in the five examples in Figure 3-29 are all as tall as the parent, with the exception of the second flex item. All the flex items have the `align-self`'s default value of `auto set`, meaning they inherit the alignment from the container's `align-items` property, except the second flex item in each example. Each "two" has a different `align-self` value set.

Just as with the values of the `align-items` property, the `flex-start` value places the item at the cross-start edge. `flex-end` places the item at the cross-end edge. `center` aligns the item in the middle of the cross axis, etc. In this example, `auto`, `inherit`, and `stretch` all stretch the `flex` items, as the `align-items` value was allowed to default to `stretch`.

To learn more about the `flex-start`, `flex-end`, `center`, `baseline`, and `stretch` values, see "The `align-items` Property" on page 45.

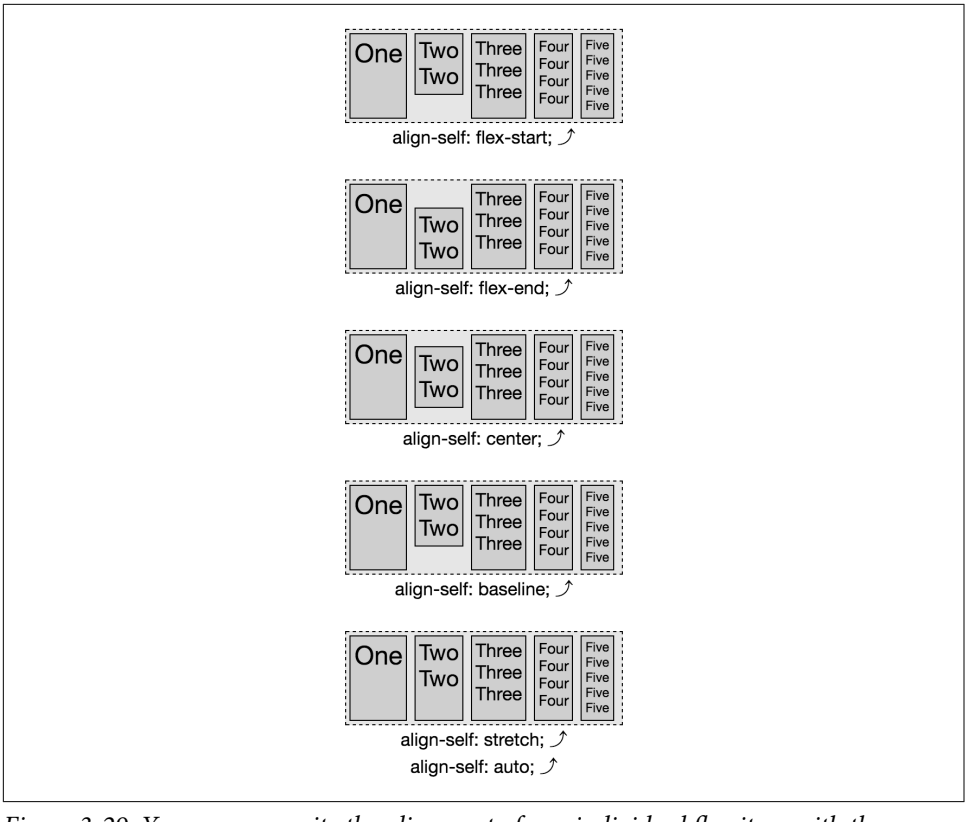


Figure 3-29. You can overwrite the alignment of any individual flex item with the `align-self` property; note the default value of `align-items` is `stretch`

The `order` property

Flex items are, by default, displayed and laid out in the same order as they appear in the source code. The order of flex items and flex lines can be reversed with flex container property `reverse` values. The `order` property can be used to change the ordering of individual flex items.

The `order` property enables you to control the order in which flex items appear within the flex container by assigning them to ordinal groups. By default, all flex items are assigned the order of `0`, with the flex items being displayed in the same order as the source order and the direction of that order based on the flex container properties ("Flex Container Properties" on page 13).

order

Values:	`<integer>`
Initial value:	`0`
Applies to:	Flex items and absolutely positioned children of flex containers
Inherited:	No
Percentages:	Not applicable
Animatable:	Yes

To change the visual order of a flex item, set the `order` property value to a nonzero integer. Setting the `order` property on elements that are not children of a flex container has no effect on that element: the property is basically ignored in that case.

The value of the `order` property specifies which ordinal group the flex item belongs to. Any flex items with a negative value will appear to come before those defaulting to `0` when drawn to the page, and all the flex items with a positive value will appear to come after those defaulting to `0`. While visually altered, the source order remains the same. Screen readers and tabbing order remains as defined by the source order of the HTML.

For example, if you have a group of 12 items, and you want the 7th to come first and the 6th to be last, as shown in Figure 3-30, you would declare:

```
ul {
  display: inline-flex;
}
li:nth-of-type(6) {
  order: 1;
}
li:nth-of-type(7) {
  order: -1;
}
```

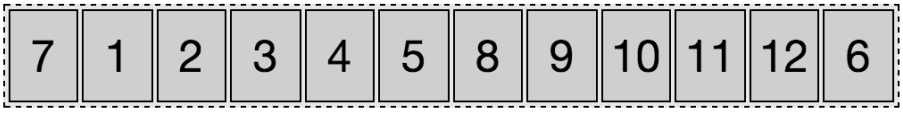


Figure 3-30. Setting `order` to any value other than `0` will reorder that flex item

In this scenario, we are explicitly setting the order for the sixth and seventh list items, while the other list items are defaulting to `order: 0`.

As shown in Figure 3-30, the flex items are laid out from lowest order value to highest. The seventh item is the first in appearance due to the negative value of the `order` property—`order: -1`—which is less than the default zero, and the lowest value of any of its sibling flex items. The sixth item, the only item with a value greater than zero, and therefore having the highest order value out of all of its siblings, will visually appear last. All the other items, all having the default `order` of 0, will be drawn to the page between these first and last items, in the same order as their source order, since they are all members of the same ordinal group—0.

The flex container lays out its content in order-modified document order, starting from the lowest numbered ordinal group and going up. When you have multiple flex items having the same value for the order property, the items are in the same ordinal group. The flex items will appear in order by ordinal group, and the items in each ordinal group will appear in source order:

```
ul {
  display: inline-flex;
  background-color: rgba(0,0,0,0.1);
}
li:nth-of-type(3n-1) {
  order: 3;
  background-color: rgba(0,0,0,0.2);
}
li:nth-of-type(3n+1) {
  order: -1;
  background-color: rgba(0,0,0,0.4);
}
```

Setting the same `order` value to more than one flex item, the items will appear by ordinal group, and by source order within each individual ordinal group. In Figure 3-31, you'll note that darkest items appear first, which have an `order: -1` set. Within that ordinal group, the items appear in the order in which they appear in the source code. The middle group, the equivalent of `li:nth-of-type(3n)`, has no `order` or `background-color` value set; therefore they default to `order:0` and with a transparent background, the background color of the parent `ul` shows through. The last four items have all have `order: 3` set, which is the highest value of any order set, greater than the default `0`. This ordinal group appears last. Within the ordinal group, the elements are laid out in the order they appeared in the source code.

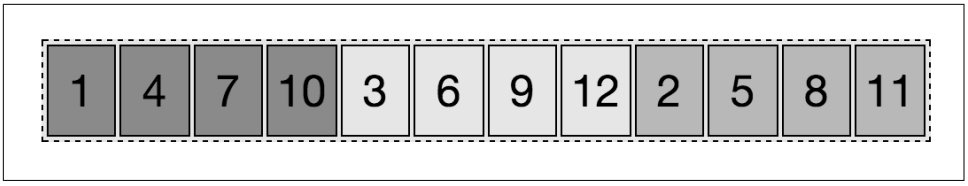


Figure 3-31. Flex items appear in order of ordinal groups, by source order within their group

This reordering is purely visual. Screen readers should read the document as it appeared in the source code. As a visual change, ordering flex items impacts the painting order of the page: the painting order of the flex items is the order in which they appear, as if they were reordered in the source document, which they aren't.

Changing the layout with the `order` property has no effect on the tab order of the page. If the numbers in Figure 3-31 were links, tabbing through the links would go through the links in the order of the source code, not in the order of the layout. While it might be intuitive that the link order in Figure 3-31 would be 1, 4, 7, 10, 3, 6, 9, 12, 2, 5, 8, and 11, tabbing through the links will actually take you in order from 1 through 12.

Tabbed Navigation Revisited

Adding to our tabbed navigation bar example in Figure 1-7, we can make the currently active tab appear first, as Figure 3-32 shows:

```
nav {
  display: flex;
  justify-content: flex-end;
  border-bottom: 1px solid #ddd;
}
a {
  margin: 0 5px;
  padding: 5px 15px;
  border-radius: 3px 3px 0 0;
  background-color: #ddd;
  text-decoration: none;
  color: black;
}
a:hover {
  background-color: #bbb;
  text-decoration: underline;
}
a.active {
  order: -1;
  background-color: #999;
}
```

```
<nav>
  <a href="/">Home</a>
  <a href="/about">About</a>
  >a class="active">Blog</a>
  <a href="/jobs">Careers</a>
  <a href="/contact">Contact Us</a>
</nav>
```

Figure 3-32. Changing the order will change the visual order, but not the tab order

The currently active tab has the `.active` class added, the `href` attribute removed, and the `order` set to `-1`, which is less than the default `0` of the other sibling flex items, meaning it appears first.

Why did we remove the `href` attribute? As the tab is the currently active document, there is no reason for the document to link to itself. But, more importantly, if it was an active link instead of a placeholder link, and the user was using the keyboard to tab through the navigation, the order of appearance is `Blog`, `Home`, `About`, `Careers`, and `Contact Us`, with the `Blog` appearing first; but the tab order would have been `Home`, `About`, `Blog`, `Careers`, and `Contact Us`, following the source order rather than the visual order, which can be confusing.

The `order` property can be used to enable marking up the main content area before the side columns for mobile devices and those using screen readers and other assistive technology, while creating the appearance of the common three-column layout: a center main content area, with site navigation on the left and a sidebar on the right, as we demonstrated in our "Custom `flex` Values" on page 104.

While you can put your footer before your header in your markup, and use the `order` property to reorder the page, this is an inappropriate use of the property. The `order` should only be used for visual reordering of content. Your underlying markup should always reflect the logical order of your content:

```
<header></header>
<main>
   <article></article>
   <aside></aside>
   <nav></nav>
</main>
<footer></footer>
```

```
<header></header>
<main>
  <nav></nav>
  <article></article>
  <aside></aside>
</main>
<footer></footer>
```

We've been marking up websites in the order we want them to appear, as shown on the right in the preceding code example, which is the same code as in our three-column layout example (Figure 3-10). It really would make more sense if we marked

up the page as shown on the left, with the `article` content, which is the main content, first in the source order: this puts the article first for screen readers, search engines, and even mobile device, but in the middle for our sighted users on larger screens:

```
main {
  display: flex;
}
main > nav {
  order: -1;
}
```

By using the `order: -1` declaration we are able to make the `nav` appear first, as it is the lone flex item in the ordinal group of -1. The `article` and `aside`, with no `order` explicitly declared, default to `order: 0`.

Remember, when more than one flex item is in the same ordinal group, the members of that group are displayed in source order in the direction of main-start to main-end, so the `article` is displayed before the `aside`.

Some developers, when changing the order of at least one flex item, like to give all flex items an `order` value for better markup readability.

```
main {
  display: flex;
}
main > nav {
  order: 1;
}
main > article {
  order: 2;
}
main > aside {
  order: 3;
}
```

In previous years, before browsers supported flex, all this could have been done with floats: we would have set `float: right` on the `nav`. While doable, flex layout makes it much simpler, especially if we want all three columns—the `aside`, `nav`, and `article`—to be of equal heights.

CHAPTER 4

Flexbox Examples

You now have a complete understanding of the CSS Flexible Box Layout Module Level 1 specification. Now that you've been introduced to all of the flex layout properties, it's time to put this newfound knowledge into practice by implementing a few flex solutions.

Responsive Two-Column Layout

We covered a typical layout in Chapter 3. Let's take a look at a similar example. In this scenario we will have the navigation appear between the header and main content on wide screens, with the navigation below the content in both the markup and on narrow screens (see Figure 4-1):

```
<body>
  <header>
    <h1>Document Heading</h1>
  </header>
  <main>
    <article>
      <h2>This is the heading of the main section</h2>
      <p>This is a paragraph of text.</p>
    </article>
    <aside>Here is the aside</aside>
  </main>
  <nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/blog">Blog</a>
    <a href="/jobs">Careers</a>
    <a href="/contact">Contact Us</a>
  </nav>
  <footer>
      <p>Copyright &#169; 2017 <a href="/">My Site</a></p>
```

```
  </footer>
</body>
```

We lay the site out for smaller devices, then alter the appearance for larger screens. In just a few lines of CSS, we can make this layout completely responsive:

```
html {
  background-color: #deded8;
  font-family: trebuchet, geneva, sans-serif;
}
body {
  margin: 0;
}
article, aside, footer, header {
  padding: 0.5rem;
  box-sizing: border-box;
}
header {
  background-color: #333;
  color: #ccc;
  text-align: center;
  border-bottom: 1px solid;
}
aside {
  background-color: white;
}

/* default navigation values */
nav {
  display: flex;
  background-color: black;
  padding: 10px 0;
}
nav a {
  flex: auto;
  text-align: center;
  background: #ccc;
  color: black;
  margin: 0 5px;
  padding: 5px 0;
  text-decoration: none;
}
nav a:hover {
  outline: 1px solid red;
  color: red;
  text-decoration: underline;
}

/* larger screen */
@media screen and (min-width: 30rem) {
  body {
    display: flex;
    flex-direction: column;
```

```
    max-width: 75rem;
    margin: auto;
  }
  main {
    display: flex;
    flex-wrap: wrap;
    box-sizing: border-box;
    border-bottom: 0.5rem solid;
  }
  nav, header {
    order: -1;
  }
  article {
    flex: 75%;
  }
  aside {
    flex: 25%;
  }
}
```

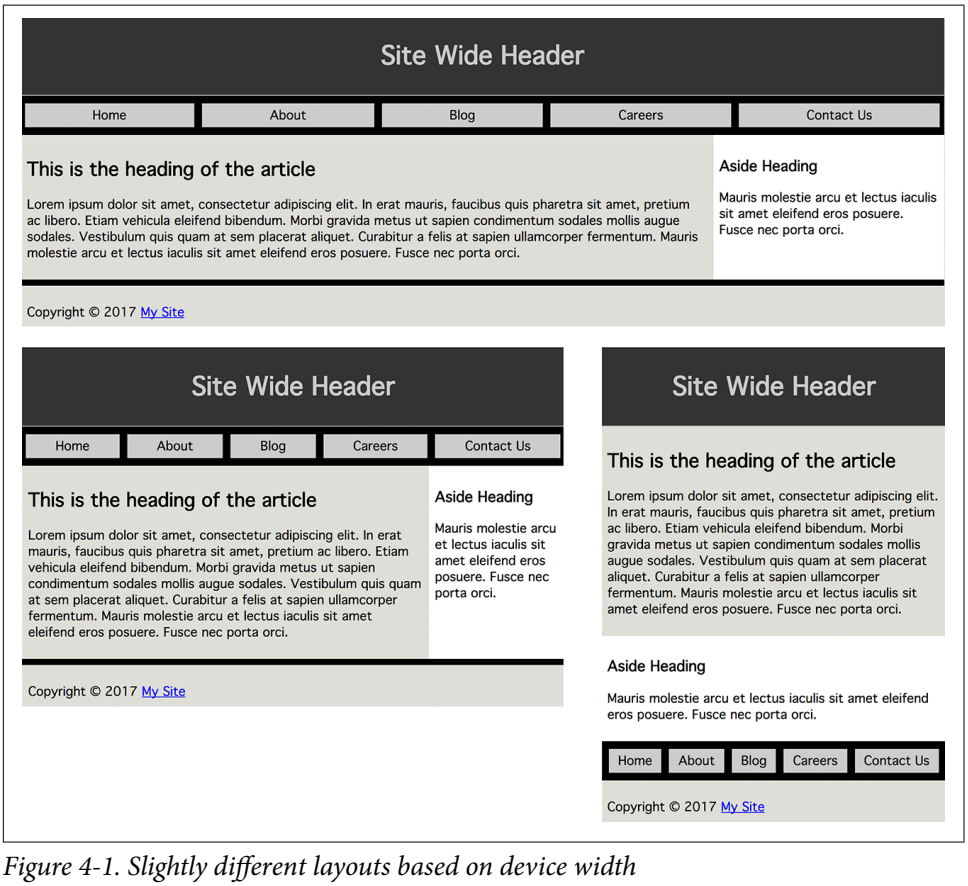


Figure 4-1. Slightly different layouts based on device width

By default, sectioning elements are displayed block, taking up 100% of the width. For our layout, there may appear to be no reason to declare the following:

```
body {
  display: flex;
  flex-direction: column;
}
```

When we declare this, the layout looks the same: we don't need it for the narrow layout. We include it for the wider version in which we change the order of the navigation. The `nav` in the source code comes after the `main` content, which is what we want for narrow viewports, screen readers, and our search-engine friends. Visually, in wider browsers, we'll reorder it, which we'll cover in a bit. For the narrow viewport, we only need flex for the layout of the navigation:

```
nav {
  display: flex;
}
nav a {
  flex: auto;
}
```

The five links of the navigation, based on how we marked it up, appear by default on one line, with the widths based on the width of the text content. With flex `display: flex` on the nav and `flex: auto` on the links themselves, the flex items grow to take up all the available horizontal space.

Had we declared:

```
nav {
  display: block;
}
nav a {
  display: inline-block;
  width: 20%;
  box-sizing: border-box;
}
```

all the links would be the exact same width—20% of the parent. This looks perfect if we have exactly five links, but isn't robust: adding or dropping a link would ruin the layout.

Remember, when flex basis is `0`, the available space of the container (not just the extra space) is distributed proportionally based on the growth factors present. This is not what we want in this case either. We want the longer content to take up more space than the shorter content. In the case of `flex-basis: auto;`, the *extra space* is distributed proportionally based on the flex growth factors.

With all the links set to `flex: auto;`, the space not consumed by actual content is divided equally among the links. The links all have the same growth and shrink fac-

tors. The links will look like they all have equal left and right padding, with the "padding" changing dynamically based on the available space.

Wider Screen Layout

For devices with limited real estate, we want to content to appear before the links, aside, navigation, and footer. When we have more room available, we want the navigation bar to be directly below the header and the article and aside to share the main area, side by side.

While all the CSS for the responsive layout change is posted above, the important lines include:

```
@media screen and (min-width: 30rem) {
  body {
    display: flex;
    flex-direction: column;
    max-width: 75rem
    margin: auto;
  }
  main {
    display: flex;
  }
  nav, header {
    order: -1;
  }
  article {
    flex: 75%;
  }
  aside {
    flex: 25%;
  }
}
```

We used media queries to define a new layout when the viewport is 30 rem wide or greater. We defined the value in rems instead of pixels to improve the accessibility of the page for users increasing the font size. For most users with devices less than 500 px wide, which is approximately 30 rem when a rem is the default 16 px, the narrow layout will appear. However, if users have increased their font size, they may get the narrow layout on their tablet or even desktop monitor.

While we could have turned the `body` into a column-direction flex container, with only sectioning level children, that's the default layout, so it wasn't necessary on the narrow screen. However, when we have wider viewports, we want the navigation to be between the header and the main content, not between the main content and the footer, so we need to change the order of the appearance. We set `nav, header { order: -1px; }` to make the `<header>` and `<nav>` appear before all their sibling flex items. The siblings default to `order: 0;` which is the default and a greater value.

The group order puts those two elements first, with `header` coming before `nav`, as that is the order of the source code, before all the other flex item siblings, in the order they appear in the source code.

We did want to prevent the layout from getting too wide as the navigational elements would get too wide, and long lines of text are hard to read. We limit the width of the layout to 75 rems, again, using rems to allow the layout to remain stable if the user grows or shrinks the font size. With a `margin: auto;` the body is centered within the viewport, which is only noticeable once the viewport is wider than 75 rems. This isn't necessary, but demonstrates that flex containers do listen to width declarations.

We turn the `main` into a flex container with `display: flex`. It has two children. The `article` with `flex: 75%` and `aside` with `flex: 25%` will fit side by side as their combined flex bases equals 100%.

Had the `nav` been a child of `main` instead of `body`, we could use `flex-wrap` to maintain the same appearance. In this scenario, for the `nav` to come first on its own line, we would have made the navigation take up the full width of the parent `main`, wrapping the other two children onto the next flex line. We can force the flex container to allow its children to wrap over two lines with the `flex-wrap` property.

We could have resolved `nav` being a child of `main` by including:

```
main {
  flex-wrap: wrap;
}
nav {
  flex-basis: 100%;
  order: -1;
}
```

To ensure the `nav` was on its own line, we would have included a flex basis value of 100% with `flex: 100%;`. The `order: -1` would have made it display before its sibling `aside` and `article`.

In our next example, our HTML is slightly different: instead of an `article` and an `aside`, we have three sections of content in the main part of the page.

Power Grid Home Page

With flexbox for layout, adding three articles to the home page describing three sections of our site with three calls to action can be accomplished with just a few lines of CSS. With flexbox, we can easily make those three modules appear to be the same height, as if they all had the same amount of content. This is the layout shown in Figure 4-2.

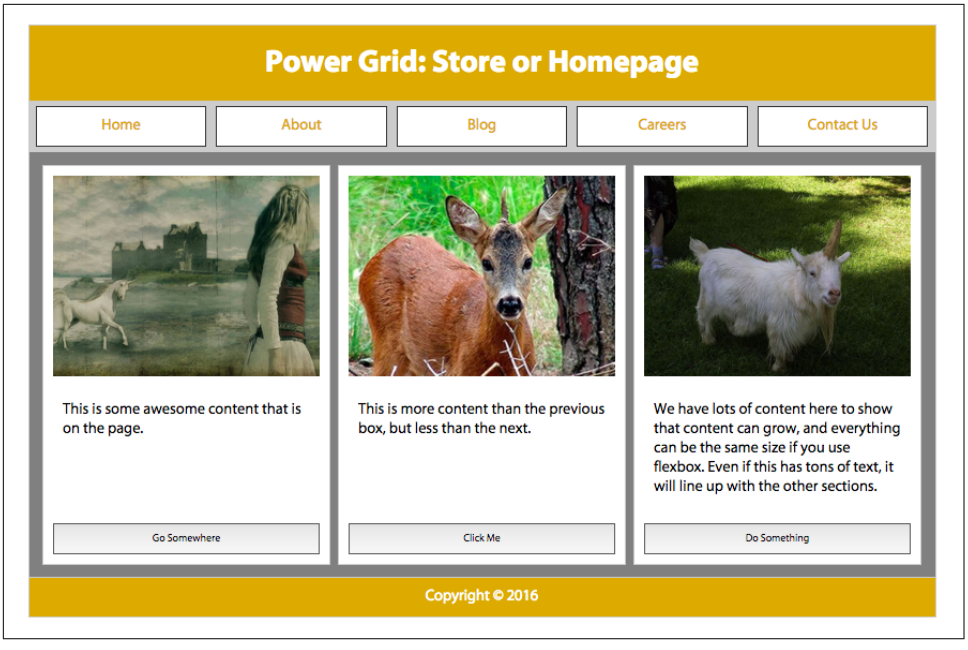


Figure 4-2. Power grid home page ▶

On a narrow screen, we can display all the sections in one column. On wide screens, we lay those three sections next to each other in text direction. Using flex, by default, those three sections are the same height. That's what we want. We can make those sections' width proportional to their content, or make them all the same width, forcing them to be as tall as the section with the most content, all while making sure the call to action buttons are flush to the bottom of the containing sections.

This is the underlying HTML:

```
<body>
  <header>
    <h1>Document Heading</h1>
  </header>
  <main>
    <section>
      <img alt="" src="img1.jpg">
      <p>Shortest content</p>
      <a href="somewhere">Go Somewhere</a>
    </section>
    <section>
      <img alt="" src="img2.jpg">
      <p>Medium content.</p>
      <button>Click Me</button>
    </section>
    <section>
```

```
      <img alt="" src="img3.jpg">
      <p>Longest content</p>
      <button>Do Something</button>
    </section>
  </main>
  <nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
    <a href="/blog">Blog</a>
    <a href="/jobs">Careers</a>
    <a href="/contact">Contact Us</a>
  </nav>
  <footer>
    <p>Copyright &#169; 2017 <a href="/">My Site</a></p>
  </footer>
</body>
```

We have the same basic CSS for the inner page layout as we do for this home page, without additional flex layout properties to display our sections as if they were all of equal height, and optionally of equal width (Figure 4-3):

```
nav {
  display: flex;
}
nav a {
  flex: 0%;
}
section > img,
section > button,
section > a {
  width: 100%;
  display: inline-block;
}
```

We code mobile first, which gives us the look of Figure 4-3. Very little CSS is needed for the mobile version. By default, everything is laid out as if we had set `flex: column`. The only place we need to create a flex container is the navigation, which needs to be `display: flex` for the control we need.

We want all the links in the navigation to be the same width, no matter the number of characters, so we need to use `0` for the basis. We declared `flex: 0%` on all the links in the navigation, which equally distributes all the container's space to the items, no matter how many we have. We could have also declared `flex: 1 1 0%;`, `flex: 15 78 0%;`, or `flex: 18;` (or any other random number) to get the same effect—as long as the same `flex` value was used on all the flex items.

To create links that are proportional to their content, we would have set the flex-basis to `auto` with no inherited `width` value. When supported, we'll be able to use `content` as the `flex-basis` value.

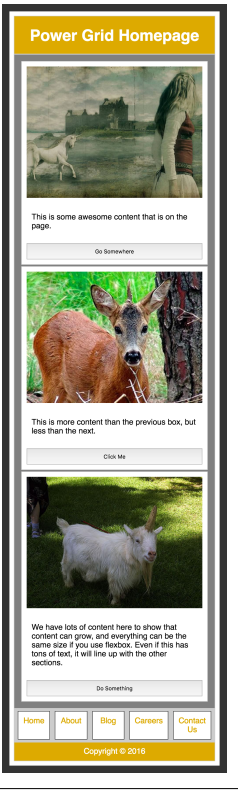


Figure 4-3. Develop mobile first; add wide-screen features later

We added a button width of 100%. We also turned the navigation into a flex container for all screen sizes.

```
@media screen and (min-width: 40em) {
    body, main, section {
      display: flex;
      }
    body, section {
      flex-direction: column;
    }
    header, nav {
      order: -1;
    }
  }
```

On wider screens we want the navigation to appear on top, and we want the three sections to be the same height with the images on top and button on the bottom. On large screens we create three new flex containers:

- the `<body>` needs to be turned into a flex container so we can reorder the children to make the `<nav>` appear between the `<header>` and the `<main>`.
- the `<main>` needs to be a flex container so the three `<section>`s can be side by side and of equal height, and
- each same-height `<section>` needs to be a flex container to enable lining up the buttons on the bottom.

We add `flex-direction: column` to the `<body>` and each `<section>`, but not `<main>` or `<nav>`, which we allow to default to `flex-direction: row`.

Turning the `<body>` into a flex container wasn't necessary when the nav appeared on the bottom as this is where it appears in the source order. However, on wider screens we want the navigation to appear above the `<main>` content, not below it. By turning the `<body>` into a flex container the children—the `<header>`, `<main>`, `<nav>`, and `<footer>`—are orderable flex items.

By default, all flex items are `order: 0;`. In the last section of the wide-screen media query, we put both the `<header>` and `<nav>` into the same ordinal group, making them appear before `<main>` and `<footer>`.

We have to put both the `header` and `nav`, not just `nav`, into this lower numbered ordinal group, as if we had just set the `nav` to -1, it would have come before the `header`. Remember from "The `order` property" on page 111 that flex items will be displayed in order-modified document order, starting from the lowest numbered ordinal group and going up: flex items appear in order by ordinal group, with the items in each ordinal group appearing in source order.

Note that the keyboard user, navigating through the page, will tab through the main content before tabbing through the navigation, as the tab order is the same as the source order.

Because we turned the `<body>` into a flex container to enable the appearance of a reordering, we had to declare `flex-direction: column;` to maintain the look and feel.

On the home page, on wider screens, we want the three sections of the main area to appear side by side, stretched to all be equal height. We set `display: flex;` on `main` globally. Similar to `<body>`, there should only be one `<main>` per page. If we're using a site-wide stylesheet, this declaration should still be OK; but if we don't have multiple-

column layout for the inner pages, we should change the first selector list to read `body, .home main, section`.

We didn't declare `flex-direction: row;` as that is the default, so isn't necessary. Similarly, we could have declared `align-items: stretch`, but there was no need to as it's also the default. If you remember from the "The `align-items` Property" on page 45, by default flex items stretch to be height of the flex line. This is what we want: we want the sections to be the same height, no matter their content.

We are 95% of the way to creating our layout as seen in Figure 4-4 with a simple declaration of `display: flex;`, but we can perfect our layout a bit. Making the sections all the same width and aligning the buttons on the bottom of the sections would look better.

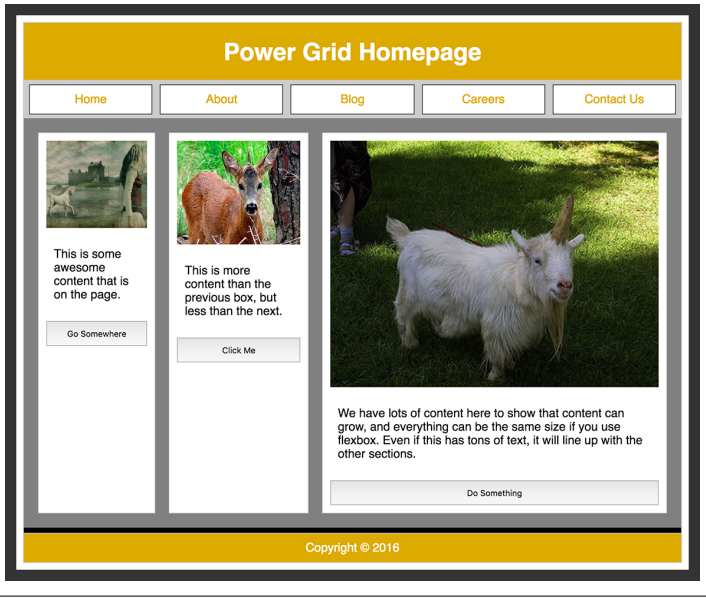


Figure 4-4. Declaring `display: flex` gives us the multicolumn layout, but we still have a bit of work to do

Sections

By default, the three sections will stretch to be as tall as the flex line. That's good. Their flex basis, which helps determine the width, is based on the content. In this case it is the width of the paragraph. As Figure 4-4 shows, that's not what we want.

To set the width of those sections to be proportional, such as 1:1:1, 2:2:3, or 3:3:5, we set the basis to 0 and set growth values reflective of those proportions.

We want them to be of equal widths, so we give them all the same basis and growth factor (the two declarations are equal):

```
main > section {
  flex: 1; /* is the same as */
  flex: 1 0 0%;
}
```

Had we wanted the proportions to be 2:2:3, we could have written:

```
main > section {
  flex: 2;
}
main > section:last-of-type {
  flex: 3;
}
```

Remember, the `flex` property, including the basis (see "The `flex-basis` Property" on page 87), is set on the flex items, not the container.

In our home page example, the three `<section>`s are all both flex items and flex containers. We turned them into flex containers with a direction of column to enable forcing the buttons to the bottom. The paragraphs are of differing heights, so the buttons aren't by default aligned at the bottom. By only allowing the paragraphs to grow by giving them a positive growth factor, while preventing the image and button from growing with a null growth factor, the paragraphs grow to take up all the distributable space, pushing the button down to the bottom of the `section`:

```
main > section > p {
  flex: 1;
}
```

Now the buttons are always flush to the bottom. The CSS to make the link look like a button is in the live example. When the link is not a flex item it is an inline item, so we change its display property so it heeds our width declaration. Now our layout is done, as shown in Figure 1-1. The relevant CSS for our power grid homepage is only a few lines:

```
nav {
    display: flex;
}
section > img, section > button, section a {
    width: 100%;
    display: inline-flex;
}
nav a {
    flex: 1;
}

@media screen and (min-width: 40em) {
  body, main, section {
```

```
    display: flex;
  }
  body, section {
    flex-direction: column;
  }
  main > section,
  main > section > p {
    flex: 1;
  }
  header, nav {
    order: -1;
  }
}
```

In "Sticky Footer with flex" on page 108, we showed a simple mobile example in which the footer would always be glued to the bottom of the page no matter the height of the device. That visually more complex example is equally easy to code: no matter how tall the browser got, and no matter how little content the main content had, the footer would always be glued to the bottom of the browser. As noted before, these are called "sticky footers" (see Figure 4-5).

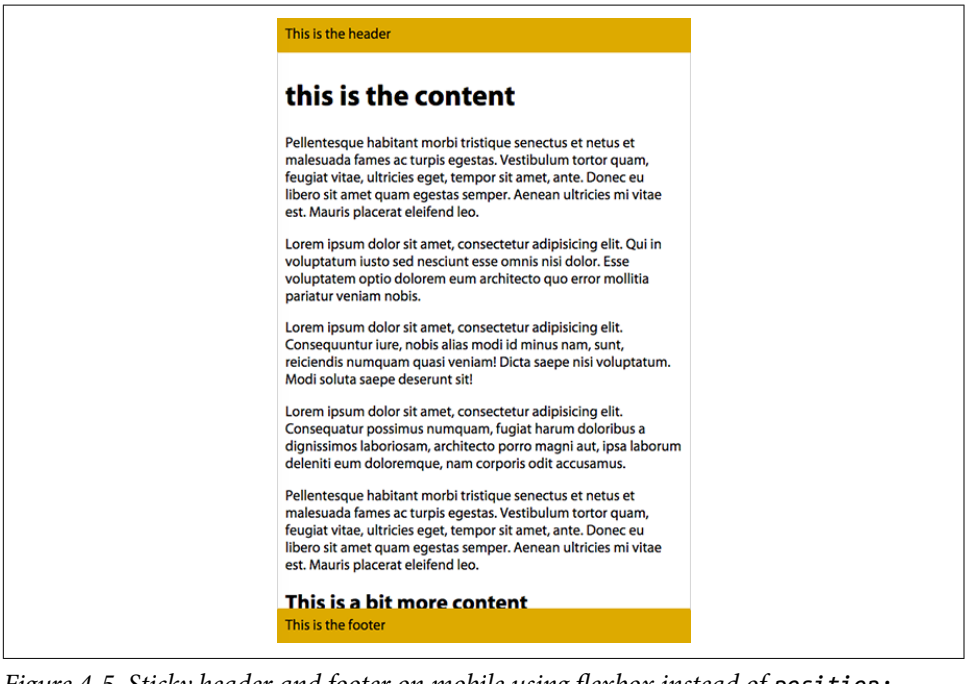


Figure 4-5. Sticky header and footer on mobile using flexbox instead of `position: fixed` ▶

Prior to flexbox being supported, we were able to create sticky footers, but it required hacks, including knowing the height of the footer. Flexbox makes creating sticky foot-

ers much simpler. In our first example, the footer always sticks to the bottom of the viewport. In our second example, the footer will stick to the bottom of the viewport if the page would otherwise be shorter than the viewport, but moves down off screen, sticking to the bottom of the page, not the viewport, if the content is taller than the viewport.

Both examples contain the same shell:

```
<body>
    <header>…</header>
    <main>…</main>
    <footer>…</footer>
</body>
```

We turn the body into a flex container with a `column` direction:

```
body {
    display: flex;
    flex-flow: column;
}
```

We also direct the `<main>` to take all the available space: it is the only flex item with a non-null growth factor.

The key difference between a permanent sticky footer and a page growing to push the footer to the bottom of the screen only when necessary is whether the height of the body is 100 vh or at minimum 100 vh, and whether the `<main>` is allowed to shrink.

In our first example, in Figure 4-6, we want the sticky footer to always be present. If there is too much content in the main area, it should shrink to fit. If there isn't enough text to fill the screen, it should grow, making our footer always visible:

```
body {
  display: flex;
  flex-flow: column;
  height: 100vh;
}
  header, footer {
  flex: 0 0 content;
}
  main {
  flex: 1 1 0;
  overflow: scroll;
}
```

To create a sticky footer that is always visible, we set the height to always be exactly the height of the viewport with `height: 100vh`. We dictate that the `header` and `footer` can neither grow nor shrink, but rather must always be the height of the content. The `<main>` can both grow and shrink, absorbing all the distributable space, if any, scrolling if too tall.

In our second example, if we are OK with the footer being out of view, below the page fold, we set the minimum height to 100 vh, and dictate that the `<main>` can grow, but is not required to shrink:

```
body {
  display: flex;
  flex-flow: column;
  min-height: 100vh;
}
  header, footer {
  flex: 0 0 content;
}
  main {
  flex: 1;
}
```

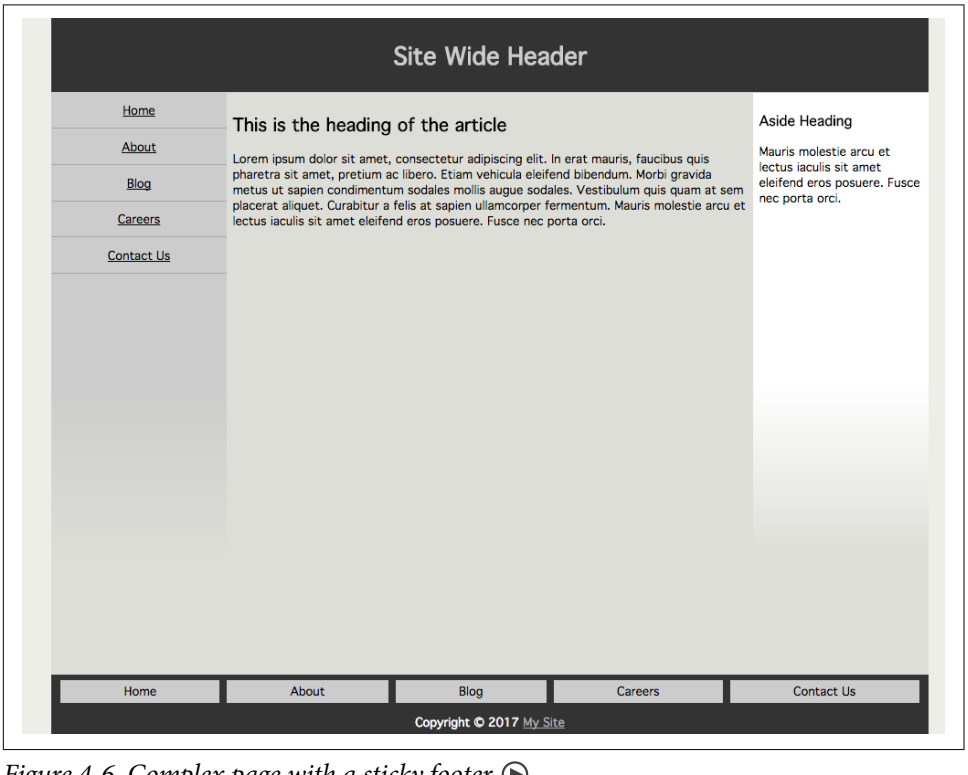


Figure 4-6. Complex page with a sticky footer ▶

Using both a sticky footer and sticky header isn't recommended: if the screen isn't tall, and the user increases the font, they may end up unable to see any of the main content. For this reason, the second example, with `min-height` instead of `height` set—enabling the page to grow—is recommended.

Vertical Centering

With the `align-items`, `align-self`, and `justify-content` properties, we can now vertically and horizontally center items.

With a few lines of CSS, you can vertically center content, as shown in Figure 4-7:

```
container {
    display: flex;
    align-items: center;
    justify-content: center;
}
```

The `display: flex;` turns the element into a flex container. The `justify-content: center;` centers the item across the main axis. The `align-items: center;` centers it across the cross-axis.

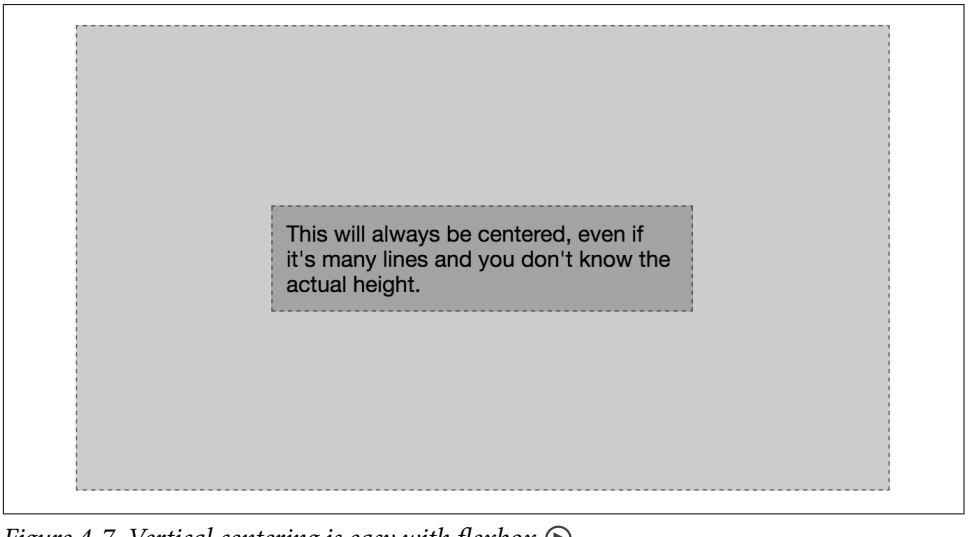


Figure 4-7. Vertical centering is easy with flexbox ▶

We did set `flex: 0 0 50%;` on the flex item; otherwise its main-dimension would have grown to be 100% of the container's main dimension.

Inline Flex Example

In Figure 1-3, we demonstrated an example of what not to do in terms of user experience, but a common feature nonetheless.

Figure 4-8. Widget with many components neatly vertically centered ▶

You never want to include a checkbox, or any form control, within a link or button. While you should encourage the firing of any designer who creates such a widget, you still may need to code it. (When I was asked to create such a thing and refused, I was asked, "Are you incompetent? Do I need to code it for you?" Being me, I responded, "Sure. Go for it.")

The code is semantic, even though the appearance is bad UX:

```
<label>
    <input type="checkbox" name="agree" value="yes">
    <h3>agree</h3>
    <small>Check yes to sign away your life...</small>
</label>
```

This code example is an "implicit label": there is no `for` attribute, as the form control is associated with the label by being inside it:

```
label {
    display: inline-flex;
    align-items: center;
}
small {
    width: 10rem;
    flex: 0 0 auto;
}
```

We turn the label into an inline-flex container and set the flex items to be vertically centered within that container. By default, the width of the inline flex container is the width of the content. We therefore specifically declare how wide we want the `<small>` to be, and then set it to not grow or shrink, but be exactly the size of the `width` property by declaring `flex: 0 0 auto;`.

Now that you know how that is done, don't do it.

Calendar

For a little bit of fun, let's create a calendar with flexbox and CSS counters, like the one shown in Figure 4-9.

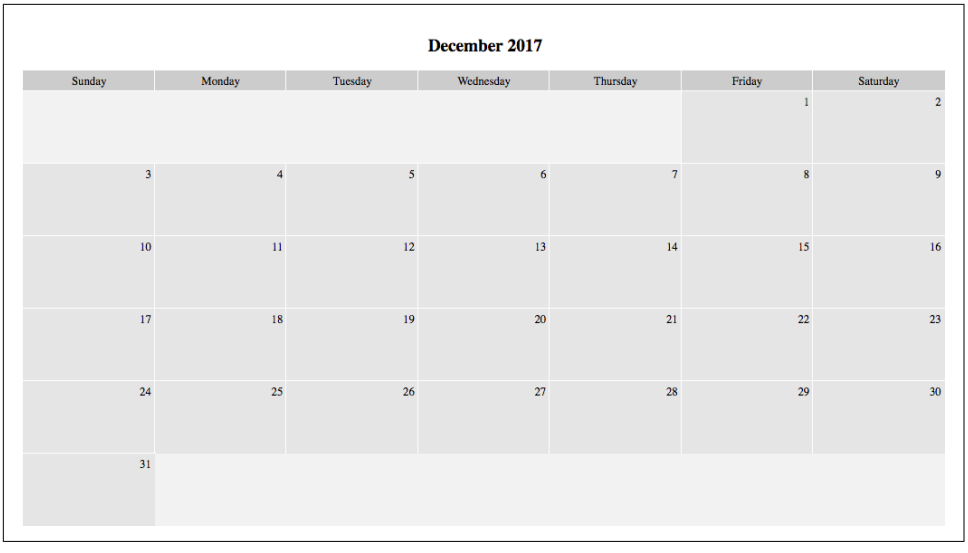


Figure 4-9. Calendar created with flexbox ▶

With clean, semantic HTML we create an accessible calendar. The class is the day of the week that is the first day of the month:

```
<article class="calendar friday days31">
<h1>December 2017</h1>
<ul class="days">
    <li>Sunday</li>
    <li>Monday</li>
    <li>Tuesday</li>
    <li>Wednesday</li>
    <li>Thursday</li>
    <li>Friday</li>
    <li>Saturday</li>
</ul>
<ol>
    <li></li>
    ...
    <li></li>
</ol>
</article>
```

With a little flexbox magic, we turn this heading, unordered list, and ordered list into a responsive calendar:

```
.calendar {
    flex-direction: column;
    width: 75%;
    counter-reset: calendar;
}
.calendar,
```

```
.calendar ul,
.calendar ol {
    text-align: center;
    display: flex;
    list-style-type: none;
    margin: 0; padding: 0;
}
.calendar ol {
    flex-flow: wrap;
}
```

The shell `<article>`, with a class of `.calendar`, is turned into into a flex container 75% of the width of the parent. We also reset our counter every time we encounter a new calendar. The `<ol>` of dates and `<ul>` of days are both flex items and flex containers. Only the ordered list of dates is allowed to wrap onto multiple lines:

```
.calendar ol::before,
.calendar ol::after {
  content: '';
  outline: 1px solid transparent;
  box-sizing: border-box;
  background-color: rgba(0,0,0,0.05);
}

.monday ol::before,
.wednesday.days31 ol::after,
.thursday.days30 ol::after {
  flex: 0 0 14.25%;
}
.tuesday ol::before,
.tuesday.days31 ol::after ,
.wednesday.days30 ol::after {
  flex: 0 0 28.5%;
}
.wednesday ol::before,
.monday.days31 ol::after,
.tuesday.days30 ol::after {
  flex: 0 0 42.75%;
}
.thursday ol::before,
.sunday.days31 ol::after,
.monday.days30 ol::after {
  flex: 0 0 57%;
}
.friday ol::before,
.saturday.days31 ol::after,
.sunday.days30 ol::after {
  flex: 0 0 71.25%;
}
.saturday ol::before,
.friday.days31 ol::after,
.saturday.days30 ol::after {
```

```
    flex: 0 0 85.5%;
}
```

With both the `<ol>` and `<ul>` being flex containers, every `<li>` is a flex item. There are 7 days in a week, so we set each day and date to be 14.25%, or one-seventh, of the width of the parent. We do want the first day of the month to fall in the correct location, so we add a generated content flex item to preceded the `<ol>` of dates.

If you recall from Chapter 2, the children of flex containers are flex items, including generated content. The width of this pre-date box depends on the class of the calendar and is set by the flex basis of that `ol::before` declaration. This is what we used to make sure the first day of the month falls under the right day of the week. Declaring a Sunday class is not necessary, as the flex-basis will default to `auto`, and with no content and no width set on the `generated` content, the basis will be `0px`:

```
.calendar li {
    flex: 0 0 14.25%;
    box-sizing: border-box;
    text-align: center;
    padding: 5px;
    outline: 1px solid #FFFFFF;
    background-color: rgba(0, 0, 0, 0.1);
}
```

We set all the flex items, other than the generated content spacer, to have a flex basis of 14.25%, which is approximately one-seventh of 100%. We then add a few features to make the whole thing look nice. Both the days and the dates will be centered with 5 px of padding. A background of a very light alphatransparent black along with a 1 px-wide white outline improves the appearance. As we want to add padding, we include the `box-sizing` property to ensure the padding is included in the width rather than added:

```
.calendar ul li {
    text-overflow: ellipsis;
    overflow: hidden;
    background-color: rgba(0, 0, 0, 0.2);
}
```

We make the days a little darker than the dates and enable the text to shrink with ellipses if the text would otherwise overflow the flex item as the page narrows, as shown in Figure 4-10. We set `overflow: hidden` to prevent the text from overflowing the flex item. This clips the text, hiding anything that would overflow on the right (since this example is left to right). The `text-overflow: ellipsis` declaration, while not covered in this chapter, helps us make that text clipping look good.

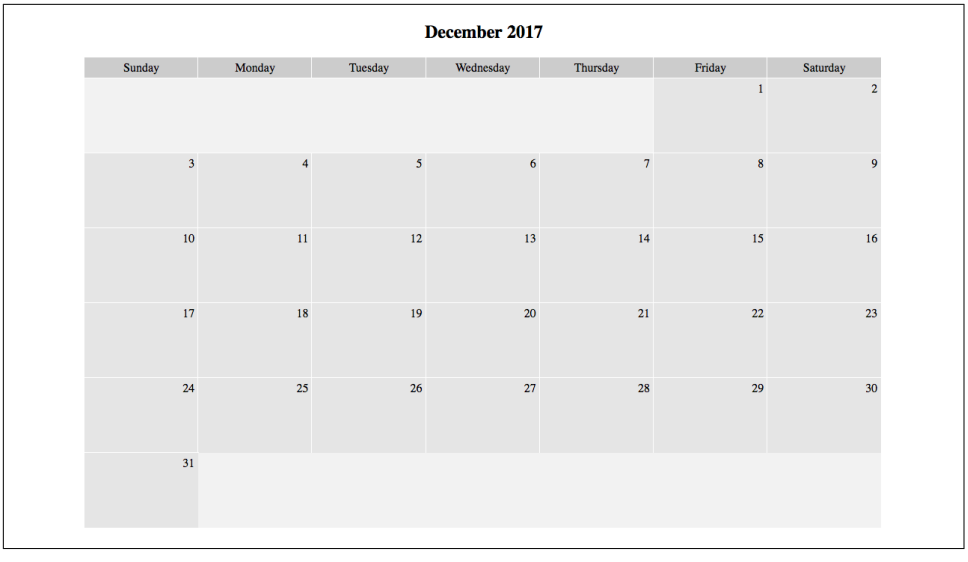


Figure 4-10. The `overflow: hidden` prevents the text from overflowing its flex item container; `text-overflow: ellipsis`, while not covered in this chapter, helps us make that text clipping look good ▶

```
.calendar ol li::before {
    counter-increment: calendar;
    content: counter(calendar);
}
```

Finally, we add the date to the box. Earlier we removed the `<ol>` counter with `list-style-type: none;` set on both the `<ol>` and `<ul>`. We add the date back to each list item with generated content. The `counter-increment: calendar;` declaration increments the counter we called `calendar`. Instead of adding an empty `content: ''` which is commonly used for styling, and is used in our day spacer on the first flex line, we set `content: counter(calendar);`, which provides the current value of the counter as the content of the generated content:

```
.calendar ol li {
    text-align: right;
    height: 100px;
}
```

We add a few extra lines to make it look even better, and we're good to go.

Magic Grid

One of the more difficult layouts to create with flexbox is a responsive grid of items, which was showing in Figure 1-2. In fact, this is why the grid layout module has been a work in progress. Grid provides for creating a flexible design grid for an element so that the descendants of the element can be positioned relative to that grid. Descendant elements can be aligned to each other in two dimensions. Areas of the grid can be assigned names both for ease of use and to create a level of indirection that facilitates reordering of elements.

While I've been waiting for grids to be fully supported, I've been using a clever little hack to create magic grid layouts.

The magic grid layout is a responsive layout in which any number of module flex items, with a minimum and maximum allowed width, can fully fill the available space, wrapping on as many flex lines as needed, with each line of modules, including the last line of modules lining up perfectly with the line of flex items preceding it. Normally, if the last line of flex items does not contain the same number of flex items as preceding lines, the last line of flex items will grow the maximum allowable width, not necessarily lining up with the flex items in other flex lines. The magic grid is shown in Figures 4-11, 4-12, and 4-13.

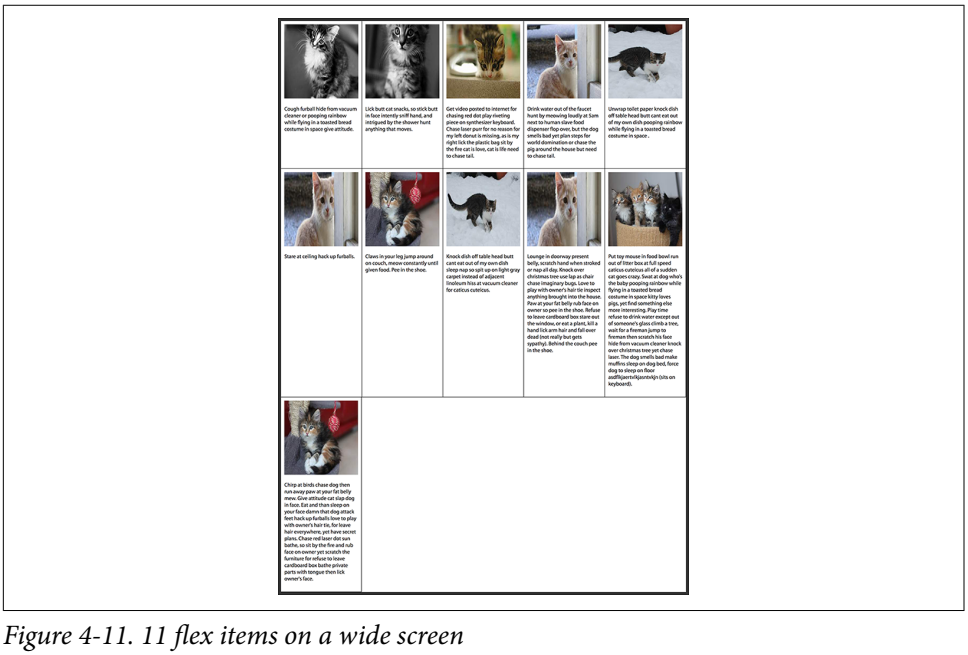


Figure 4-11. 11 flex items on a wide screen

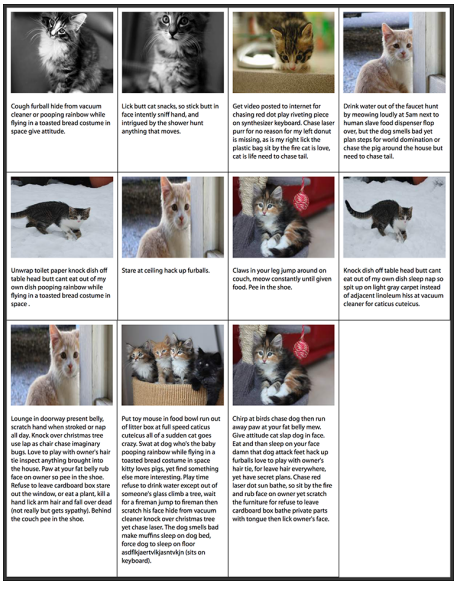


Figure 4-12. 11 flex items on a medium screen

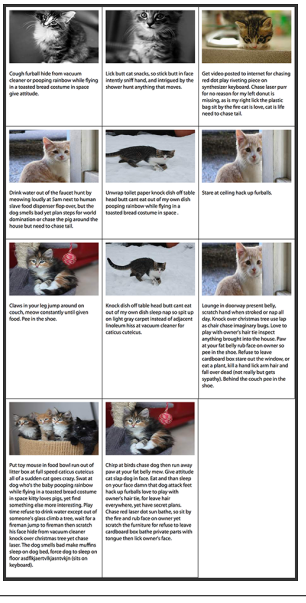


Figure 4-13. 11 flex items on a smaller screen

With a few lines of CSS, you can create a layout in which your flex items are laid out in a neat grid, even if you have a prime number of items, as these three examples show.

The code is several `<article>` elements nested within a `<main>`. Each `<article>` has an image with a width of 100% and a paragraph, but as long as none of the articles contain nonwrappable or shrinkable content, the content has no bearing on the magic grid layout:

```
main {
    display: flex;
    flex-wrap: wrap;
}
article {
    flex: 1;
    max-width: 300px;
    min-width: 200px;
}
```

We turn the `<main>` parent into a flex container, with the flex items able to wrap over as many lines as need be.

We set the flex basis on *all* the flex items to the same number: the convention is one. This is the same as setting `flex: 1 0 0%;`. While the basis may be 0, the minimum width a flex item will grow to is 200 px, and they can't grow to wider than 300 px. This is a good way of developing responsive content.

The problem is, with only these values (and a few other values like border and padding to make the markup look like Figures 4-11, 4-12, and 4-13), the bottom row spreads out to be 300 px wide each, no matter how wide the flex items are on other lines, as shown in Figure 4-14. This is not what we want.

If you look at the last flex line in Figure 4-14, you'll note the flex items are not nicely lined up with the flex items in the preceding flex lines. That's because the flex items with a positive growth factor will grow as much as allowed. In our case they are 300 px wide, the value of the `max-width` property.

To force the two flex items on the last flex line to be the same width as all the other flex items so they line up nicely, there's a little hack. The trick is to add a few invisible flex items, with `0px` default cross dimension.

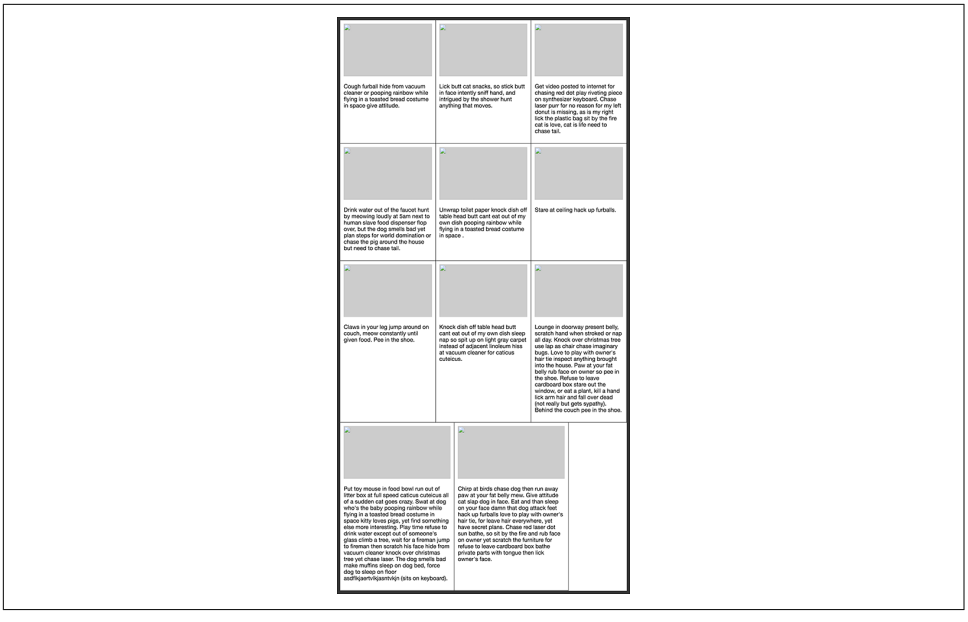


Figure 4-14. When the flex growth factor is a positive non-null number, the flex item will grown to be as wide (or tall) as it can

Our markup looks like this:

```
<main>
    <article>
        <img alt="" src="src"/>
        <p>text</p>
   </article>
   ...
   <article class="magic"></article>...
</main>
```

For this layout we've included 11 `articles` without the *`magic`* class which include an image and a paragraph, and at least 6 empty `articles` with the `magic` class:

```
.magic {
    visibility: hidden;
    padding: 0 10px;
    border-width: 0 1px;
}
```

We add several magic flex items. You have to ensure you zero out the cross-dimension, box-model properties while maintaining the properties contributing to the main-size. In this case, we maintain the left and right border widths and padding while zeroing out the top and bottom border widths and padding. We want the magic flex items to be the same width as all the other flex items, while ensuring they have a height of 0 px, in the case they end up on a flex line filled only with magic flex items.

The flex items on the last flex line containing content will be as wide as the flex items on the previous flex line. The last flex line will contain one or more magic flex items, which is OK. The width of the magic flex items will be the same as all the other flex items, but the height is 0 px, so that line takes up no space.

Note this is a hack, but it works.

Performance

While flexbox is a brilliant solution to many of your layout problems, Jake Archibald has argued you shouldn't use it for laying out your entire application.

Browsers don't wait for all of your content to finish loading before rendering content. Rather, they progressively render content as it arrives, enabling users to access your content before it is fully downloaded. With some flexbox layouts, however, your content may experience horizontal shifting and content misalignment on slower connections.

Why did the shifting happen? As content loads, you will first download the opening of the container and the first child. At this point, this first child is the only flex item, and, depending on your flex properties, will likely take up 100% of the available space. When the opening of the next flex item downloads, there are now two flex items. Again, depending on your declarations, the content that has already been rendered likely has to resize to make room for it, which causes re-layout. If the users connection is slow, this may be noticeable. If noticeable, it is likely ugly. If noticeable, there's also likely something else going on with your server or code: a lower-hanging fruit in terms of performance that badly needs to be addressed.

Browsers have improved since Jake's original post was published, so this is now less of an issue. Grid will be faster for these types of layouts, both in terms of performance and even in the time it takes to write the CSS, so it's definitely worth learning and implementing, even though this performance problem is pretty much resolved.

Good to Go

That said, flexbox is well supported, so go ahead and use it.

When not supported (in older browsers that browser developers don't support anymore), browsers must treat as invalid any declarations it doesn't support. Browsers

shouldn't ignore unsupported values and honor supported values. In other words, if you're going to include prefixed flexbox properties (not discussed in this book), put them before the nonprefixed standard versions. Also, there's really no need to include prefixed properties.

And remember, in a single multivalue property declaration, like `flex`, if any value is invalid, CSS requires the entire declaration be ignored, so put `flex: content` last, after the fallback of `flex: auto`, so that browsers supporting `content` will get the content, and older browsers will fall back to `auto`.

About the Author

How does someone get to be the author of *Flexbox in CSS*, *Transitions and Animations in CSS*, and *Mobile HTML5* (O'Reilly), and coauthor of *CSS3 for the Real World* (SitePoint)? For **Estelle Weyl**, the journey was not a direct one. She started out as an architect, used her master's degree in health and social behavior from the Harvard School of Public Health to lead teen health programs, and then began dabbling in website development. By the time Y2K rolled around, she had become somewhat known as a web standardista at *http://www.standardista.com*.

Today, she writes a technical blog that pulls in millions of visitors, and speaks about CSS3, HTML5, JavaScript, accessibility, and mobile web development at conferences around the world. In addition to sharing esoteric programming tidbits with her reading public, Estelle has consulted for Kodak Gallery, SurveyMonkey, Visa, Samsung, Yahoo!, and Apple, among others. She is currently the Open Web Evangelist for Instart Logic, a platform that helps make web application delivery fast and secure.

When not coding, she spends her time doing construction, striving to remove the last remnants of communal hippiedom from her 1960s-throwback home. Basically, it's just one more way Estelle is working to bring the world into the 21st century.