Flexbox

for responsive web design

How to build websites that work on any browser and any device

Harvey Berman

Flexbox for Responsive Web Design

By Harvey Berman

Copyright © 2016 Harvey Berman
All Rights Reserved

Table of Contents

Preface

Part 1. Flexbox Tutorial

- Chapter 1. Introduction to Flexbox
- Chapter 2. Flexbox Terminology
- Chapter 3. Flex Containers
- Chapter 4. Item Flow and Order
- Chapter 5. Item Alignment
- Chapter 6 Item Flexibility

Part 2. Browser Support Strategies

- Chapter7. Vendor Prefixes
- Chapter 8. Progressive Enhancement

Part 3. Responsive Web Design

- Chapter 9. Media Queries
- Chapter 10. Responsive Text
- Chapter 11. Responsive Images
- Chapter 12. Setting the Viewport

Part 4 Responsive Web Pages

Chapter 13. Responsive Navigation

Chapter 14. Responsive Footer

Chapter 15. Responsive Layout

Chapter 16. Responsive Web Page

Appendix A. Understanding Flex-Direction

Appendix B. Autoprefixer Options

Appendix C. Checkbox Hack

Preface

Responsive web design is all about creating flexible websites that work well on phones, tablets, computers, and other devices. The key word here is “flexible”.

The CSS flexible box model, known as flexbox, is a new layout tool that can help you build flexible websites painlessly. It is powerful, easy to learn, and easy to use.

About This Book

Flexbox for Responsive Web Design is a practical guide that explains the principles, benefits, and challenges of responsive web design with flexbox. It has four parts:

- A clear, compact tutorial covering the language of flexbox – core concepts, terminology, and syntax.
- Effective strategies for dealing with older browsers that don't support flexbox.
- A user-friendly introduction to responsive web design – theory and practice in "aw shucks" language.
- Step-by-step instruction for building responsive web pages with flexbox.

Topics are explained in clear, easy-to-understand language. Key points are reinforced with real-world examples. And code is provided so you can see exactly how everything works.

Who This Book Is For

This book is for web designers who want to build responsive websites that work on any browser and any device. The only requirement is a basic understanding of HTML and CSS.

What You Will Learn

As you progress through the book, you will learn to:

- Position content where you want with flexbox – no floats, no workarounds, and no hacks.
- Generate responsive text that is readable on any screen – the right font size, line-height, and line length.
- Serve responsive images efficiently – the right image at the right size – using simple HTML and CSS.
- Build flexible navigation panels that work great on desktops, tablets, and mobile phones.
- Design responsive web pages - pages that adjust automatically to different devices and screen sizes.

By the end of the book, you will have a customizable template for a fully-

responsive web page powered by flexbox – a tangible product that you can use for your own projects.

Source Code

All of the source code used in this book can be downloaded at http://flexbox-rwd.com/download.aspx. Code is organized by chapter, so you can easily find what you are looking for.

Why This Book?

There is nothing hard about building responsive websites with flexbox. But there are a few moving parts. This book focuses on the most important parts, so you can quickly acquire the skills you need to design responsive websites that work reliably on all browsers and all devices.

Part 1. Flexbox Tutorial

Responsive websites are flexible websites. They work well on smart phones, tablets, desktop computers, or any other device.

For years, designers relied on floats, hacks, and clever javascript to make sites flexible. Now, there is a better way – the CSS flexible box model, known as flexbox. In this book, we advocate using flexbox to build responsive websites.

Before you can use flexbox, though, you need to understand flexbox; so we begin with a tutorial covering the following topics:

- Chapter 1. Introduction to Flexbox
- Chapter 2. Flexbox Terminology
- Chapter 3. Flex Containers
- Chapter 4. Item Flow and Order
- Chapter 5. Item Alignment
- Chapter 6. Item Flexibility

Chapter 1. Introduction to Flexbox

If you want to build a responsive website, you really need to know about flexbox.

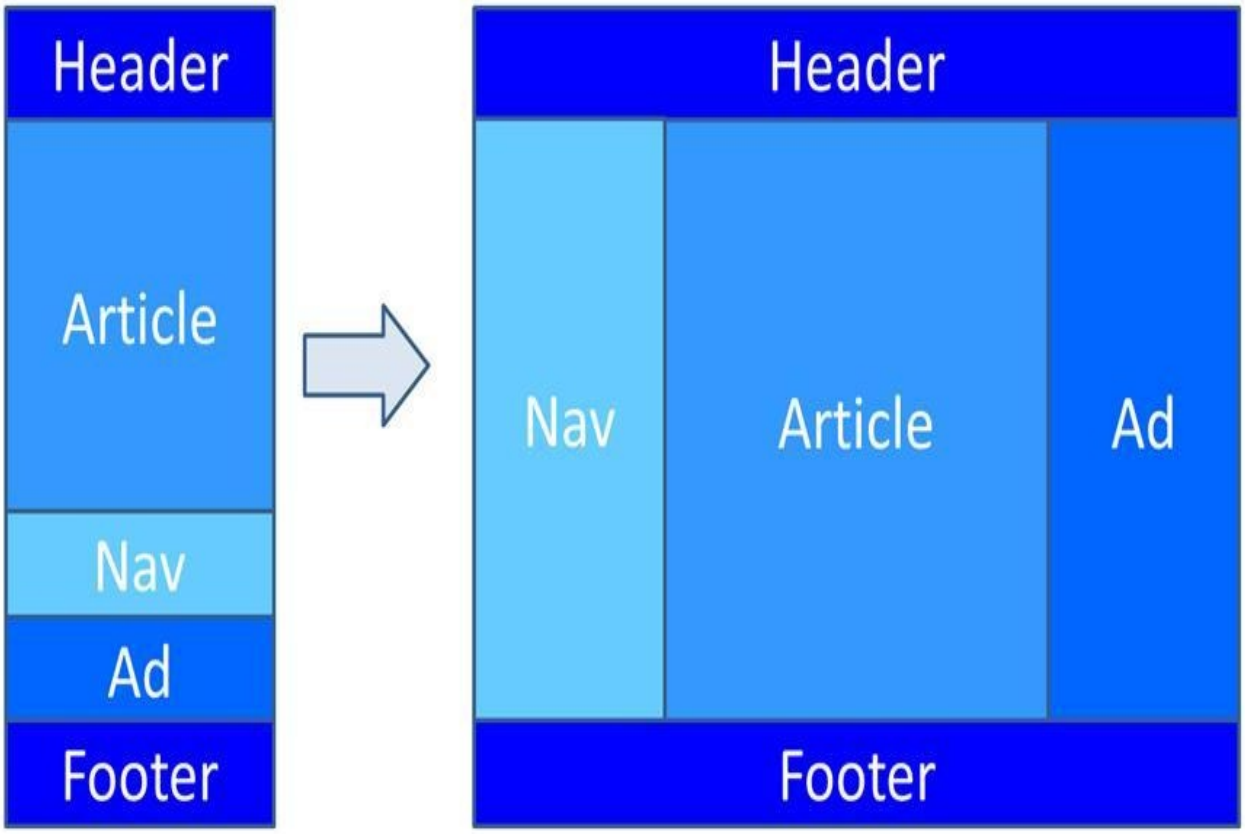
What is Flexbox?

Flexbox is the common name for the CSS3 flexible box layout module. This module defines a new layout mode, called flex layout. What, you may ask, is a layout mode?

A layout mode is a set of rules that determine the size and position of web elements, based on their relationships with siblings and ancestors. There are several types of layout modes:

- **block.** Designed for laying out documents.
- **inline.** Designed for laying out text.
- **table.** Designed for laying out tables.
- **positioned.** Designed for positioning elements independent of other elements.
- **flex.** Designed for laying out web components that can be resized smoothly.

The following figure illustrates a common scenario where layout mode comes into play. The figure shows a web page that changes layout dynamically to accommodate different screen resolutions:



When the screen is small, the web page is displayed in a single column; and when the screen is big, the page expands to fill three columns. A web designer could probably use any layout mode to produce the change shown – one-column for small screens and three-columns for large screens – but savvy designers use flexbox.

Why Use Flexbox?

The main reason to use flexbox is this: complex layouts are easier to produce with flexbox than without flexbox. For example, with flexbox it is easy to:

- Align content vertically within a parent element.
- Create equal-height columns for grid-based layouts.
- Reorder content for different screen resolutions and different devices.

These things can also be done with traditional CSS using other layout modes, but they require floats, fancy javascript, and/or clever hacks.

Flexbox, in contrast, is a true page layout tool. Designers can build complicated layouts painlessly with just a few lines of code – no floats, no workarounds, and no hacks.

> ***NOTE***
>
> ***For a good description of common design problems that were once hard or impossible to solve with CSS alone, visit***

https://philipwalton.github.io/solved-by-flexbox/.

Philip Walton shows (with code) how each problem is now "trivially easy" to solve with flexbox.

Flexbox and Responsive Web Design

Flexbox is particularly useful for responsive web design. When you build a responsive website, you build a site that adapts itself to the device on which it is displayed. It shrinks to fit the small screen of a smart phone and expands to fill the large screen of a desktop monitor.

With flexbox, components on a web page can be made to:

- Expand as the browser viewport gets bigger.
- Shrink as the viewport gets smaller.
- Stay the same size, regardless of viewport behavior.

This degree of layout control comes at no cost to performance. There are no add-on frameworks to compete for precious resources and slow things down. Everything is accomplished with simple CSS, using the flexbox layout model; and response is lightning-quick.

The Challenge of Flexbox

Flexbox must be complicated, right? Nope! Flexbox is easy to learn and easy to use. If you have a working knowledge of HTML and CSS, you can use flexbox effectively.

So, what's the problem?

The main problem with flexbox is browser compatibility. Although flexbox is recognized by all modern browsers, some people still use older browsers (like Internet Explorer 9) that do not fully support flexbox. Dang!

NOTE

As this is being written, 81% of browsers worldwide fully support flexbox. For a current assessment of browser support for flexbox, visit http://caniuse.com/#search=flex.

The current state of browser support is the main challenge to flexbox. On one hand, we want to take advantage of the cool layout capabilities that we get with flexbox. On the other hand, we want websites to be fully functional on all

browsers.

Here’s the bottom line: Flexbox can be used today if you only support modern browsers. If you support older browsers as well, you need to employ one or both of the following strategies:

- Add vendor prefixes to stylesheets.
- Design with progressive enhancement.

These topics are covered in Chapters 7 and 8. For now, the main takeaway is this: The challenge posed by browser support for flexbox should not be a deal-breaker. Effective strategies exist to use flexbox and still deliver fully-functional websites to any browser.

Next Steps

In the next chapters, we will cover flexbox terminology and flexbox syntax. You’ll see how flexbox facilitates layout control, and you’ll learn to cope with older browsers that don’t support flexbox.

By the end of this book, you will know how to build a responsive web page that works on all browsers and all devices, with flexbox.

This sounds like fun, so let’s get started.

Chapter 2. Flexbox Terminology

To use flexbox effectively, you need to speak the language. In this chapter, we introduce some flexbox terminology – jargon that you need to understand to use flexbox effectively.

The Flex Container and Flex Items

A flexbox is made up of a “flex container” and its “flex items”, so it is important to know how they are related.

- The flex container is a parent element.
- Flex items are children of the parent element.

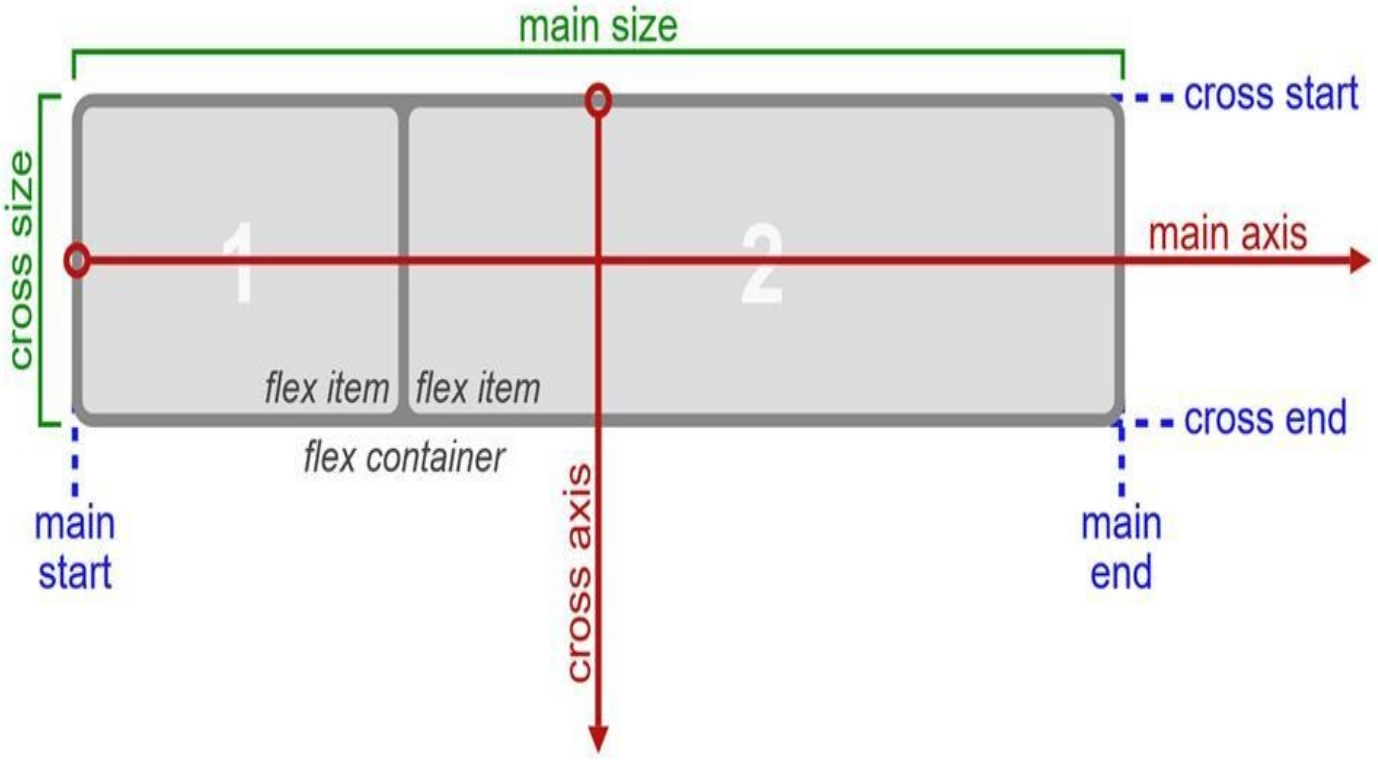
Consider the code below. The <header> element is a potential flex container, and its children (the elements) are potential flex items.

```
<header class = “container”>
    <div class = “item”>Flex item 1</div>
    <div class = “item”>Flex item 2</div>
    <div class = “item”>Flex item 3</div>
</header>
```

Flex Layout Points of Reference

On a ship, port and starboard are nautical terms for left and right, respectively. Port is the left-hand side of a vessel, facing forward, and starboard is the right-hand side.

In the same way, points of reference exist for a flex container and flex items. Luckily, the Worldwide Web Consortium (W3C) has prepared a chart to illustrate the meaning of key points of reference for flex containers. That chart, found at https://drafts.csswg.org/css-flexbox/, is reproduced below.



Direction

The direction in which flex items are laid out determines whether the container is a row flex container or a column flex container. In **row flex containers**, items are laid out left to right or right to left. In **column flex containers**, items are laid out top to bottom or bottom to top. In the figure above, flex items are laid out left to right, so the container shown in the figure is a row flex container.

The **main axis** is determined by the direction in which flex items are laid out. If items are laid out in rows (left to right or right to left), the main axis is the horizontal axis. If they are laid out in columns (top to bottom or bottom to top), the main axis is the vertical axis. The **cross axis** is the axis perpendicular to the main axis.

In the figure above, the main axis is the horizontal axis, so the cross axis is the vertical axis.

Location: Start and End Points

The direction in which flex items are laid out determines the **main start** side of the flex container or of a flex item. If items are laid out left to right, the main start is the left side; if items are laid out right to left, the main start is the right side. Similarly, if items are laid out top to bottom, the main start is the top side; if items are laid out bottom to top, the main start is the bottom side. The **main end** is the side opposite the main start.

In the figure above, items are laid out left to right, so main start is the left side, and the main end is on the right side.

Size

The **main size** of a flex container or a flex item is its width or height, whichever runs along the main axis. The **cross size** of a flex item is its width or height, whichever runs along the cross axis.

In the figure above, the main size of the flex items would be their width, since their widths run along the main axis. And the cross size would be their height, since height runs along the cross axis.

Flex Lines

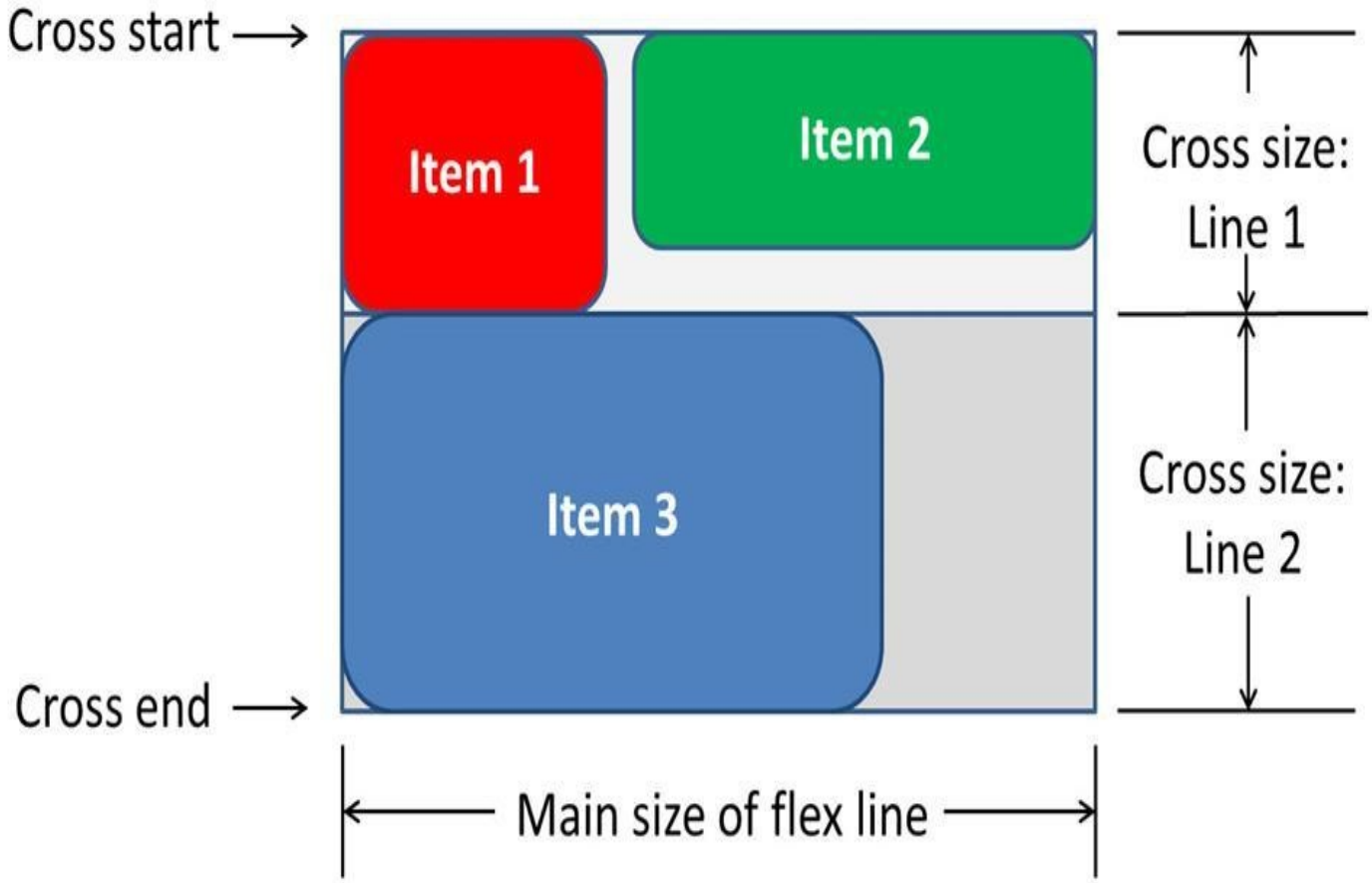
Flex lines are hypothetical containers used to group and align flex items in a flex container. With respect to flex lines, there are two types of flex containers:

- **Single-line flex container**. Displays flex items in a single line, even if items overflow the container.

- **Multi-line flex container**. Displays flex items on one line or on multiple lines, as needed to prevent items from overflowing the container.

The main distinction between a single-line flex container and a multi-line flex container is the fact that a single-line container can only display flex items on one line. A multi-line flex container can display flex items on one line as well; but it can also display items on multiple lines, if required to prevent items from overflowing the container.

The figure below shows a multi-line flex container with three flex items and two flex lines.



When a flex container holds multiple lines, the lines are stacked along the cross axis. The side of the flex container where the first line of flex items is displayed is called the **cross start**. The side where the last line of flex items is displayed is called the **cross end**.

In a single-line flex container the cross size of the flex line equals the cross size of the flex container. In a multi-line flex container, the cross size of each line is equal to the minimum size needed to contain flex items displayed on that line. The main size of a line equals the main size of its flex container.

In the figure above, notice that the container is too small to hold all three flex items on a single line, so it displayed Item 3 on a second line. That is what multi-line containers do.

In Chapter 4, we'll show how the CSS property "flex-wrap" is used to create single-line and multi-line flex containers.

Next Steps

Now that you speak the language of flexbox, let's move on to the fun stuff – coding.

> ***NOTE***
>
> ***Over the years, flexbox has been through several iterations and, even now, is technically a work in progress. Nevertheless, flexbox is supported today by all modern browsers and the current version is***

considered relatively stable.

To see a W3C report on the current state of the progress, visit https://drafts.csswg.org/css-flexbox/.

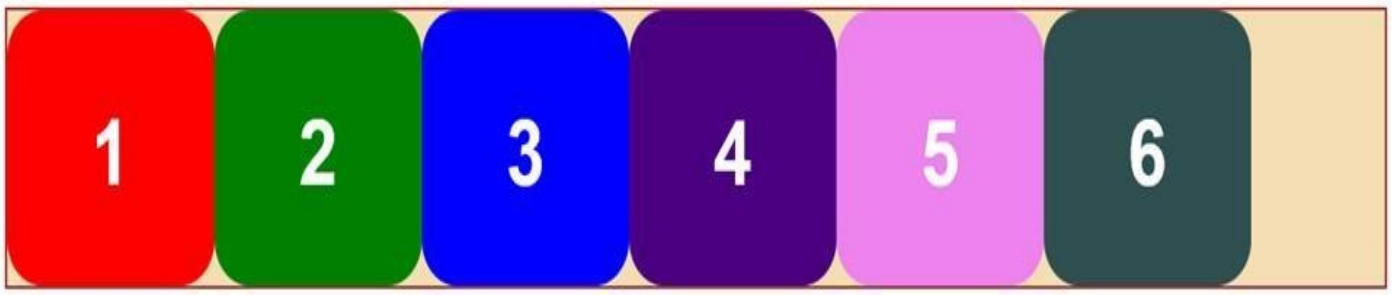
Chapter 3. Flex Containers

The heart of a flex layout is the flex container.

In this chapter, we'll show how to create a flex container. And we'll explain the difference between block-level flex containers and inline flex containers.

Boxes, Boxes, Boxes

Throughout this chapter and the next three chapters, we'll illustrate each new property with a simple example, so you can see how everything works. All of the examples are variations on a theme – small boxes with rounded corners displayed within a larger box with square corners. The small boxes are flex items and the larger box is a flex container, like this:



Here's the HTML markup that produced the large box and the smaller, numbered boxes.

HTML

```
<div class="container container-flex">
    <span class="item color1">1</span >
    <span class="item color2">2</span >
    <span class="item color3">3</span >
    <span class="item color4">4</span >
    <span class="item color5">5</span >
    <span class="item color6">6</span >
</div>
```

The large box is a div element, and the smaller boxes are span elements. Notice that the div element is a parent element and the span elements are all children of the parent. All flexbox layouts are built in this way – a parent element for the container and child elements for the flex items.

We'll use this markup or very similar markup for all of the examples in this chapter and in the next three chapters. This is pretty simple HTML, so we won't always repeat it for each example. If there is ever any question about the markup that we used, the source code for every example is available at http://flexbox-

rwd.com/download.aspx.

And here are the CSS declarations used to style boxes in our examples. You've probably used style rules like these many times in the past.

CSS

```
html { box-sizing: border-box; }
*, *:before, *:after { box-sizing: inherit; }
.container {
    background-color: wheat;
    font-size: 2rem;
    font-family: arial;
    font-weight: bold;
    width: 1200px;
    text-align: center
    border: solid 2px brown; }
.item {
    padding: 10px;
    color: white;
    border-radius: 30px;
    height: 180px;
    width: 180px;
    padding-top:60px; }
.color1 {background-color: red;}
.color2 {background-color: green;}
.color3 {background-color: blue;}
.color4 {background-color: indigo;}
.color5 {background-color: violet;}
.color6 {background-color: darkslategray; }
```

NOTE

Throughout this book, we set the box-sizing property equal to border-box. For a clear explanation of why this makes sense, see https://css-tricks.com/box-sizing/.

We'll use these styles or very similar styles for all of the examples in this chapter and in the next three chapters, so we won't be repeating them for each example. Again, if there is ever any question about the CSS that we used, the source code is available at http://flexbox-rwd.com/download.aspx.

Since the focus of this book is flexbox, we'll put all of the flexbox properties for the flex container in a separate class, called the "container-flex" class: and we will show the "container-flex" class for each example. To see how we're using flexbox in any example, just look at the container-flex class.

CSS

```
.container-flex {
    display: flex;     /* Declare a flex container */
}
```

Here, the "container-flex" class has only one CSS declaration – a display property with the value of "flex". We'll add other declarations as we move through the flexbox tutorial, but the display property is a good place to begin.

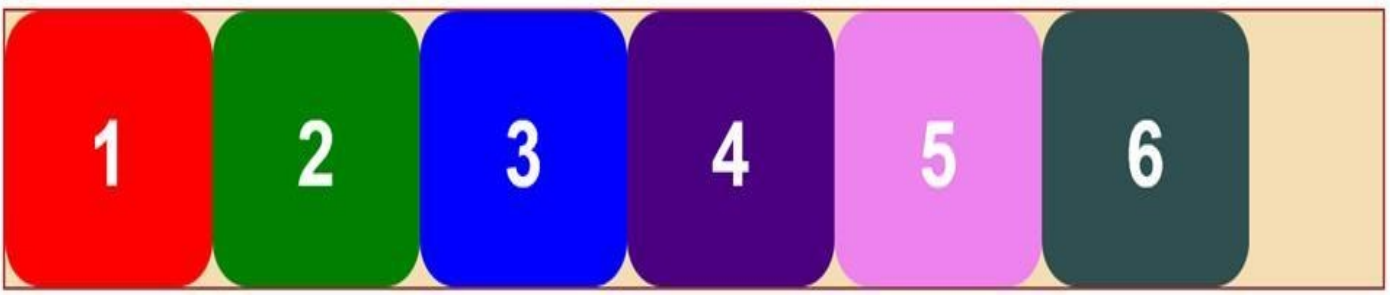
The Display Property

If you've worked with CSS, you're probably already familiar with its display property. Changing values for the display property of an HTML object affects the layout of surrounding content. For example,

- **display: block** makes the HTML object a block element. Block elements generally start a new line. Sibling elements appear on different lines.
- **display: inline** makes the HTML object an inline element. Inline elements do not start a new line, and siblings can appear on the same line.

Flexbox defines two new values that you can assign to the display property of an HTML object. You can assign the value "flex" or "inline-flex" to an object. This identifies the object as a flex container. And it turns children of the flex container into flex items.

Consider our first example – six numbered boxes contained within a bigger un-numbered box.



Here, again, is the HTML markup that produced the container and the six boxes.

HTML

```
<div class="container container-flex">
    <span class=" item color1">1</ span >
    <span class=" item color2">2</span >
    <span class=" item color3">3</span >
    <span class=" item color4">4</span >
    <span class=" item color5">5</span >
    <span class=" item color6">6</span >
</div>
```

And here is the CSS "container-flex" class.

CSS

```
.container-flex {
    display: flex;     /* This declares a flex container */
}
```

The "display: flex" declaration applies to the parent element in our HTML. It turns that parent element into a flex container, and it makes the parent's children flex items. In this example, the children are elements.

Flex vs. Inline-Flex

We noted that the CSS display property can take two new values: flex and inline-flex. Both values turn a parent element into a flex container, and both values turn children of that element into flex items.

So what's the difference between flex and inline-flex? Flex makes the flex container a block element, and inline-flex makes it an inline element. When two inline flex containers are siblings, they can appear on the same line; but when two block-level flex containers are siblings, they will appear on separate lines. Let's illustrate this with an example.

Here's some HTML markup that produces two containers, each with three boxes. Notice that each element is a parent element and the elements are all children of a parent.

HTML

```
<div class="container container-flex">
    <span class="item color1">1</span >
    <span class="item color2">2</span >
    <span class="item color3">3</span >
```

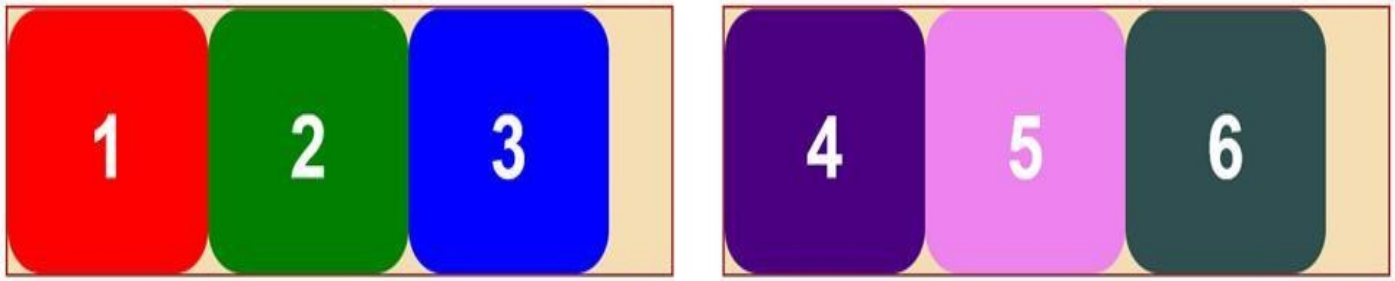
```
</div>
<div class="container container-flex">
    <span class="item color4">4</span >
    <span class="item color5">5</span >
    <span class="item color6">6</span >
</div>
```

We'll use much the same CSS for this example as we used previously, except that we'll give the display property a value of "inline-flex".

CSS

```
.container-flex {
    display: inline-flex;      /* This declares an inline flex container */
}
```

This markup, with a display value of "inline-flex", produces two flex containers next to each other on the same line.

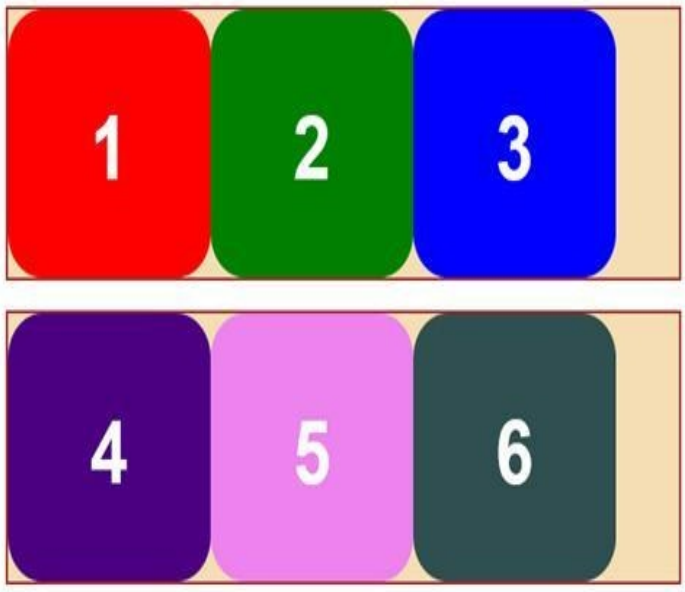


What happens if we change the display value from "inline-flex" to "flex"?

CSS

```
.container-flex {
    display: flex;      /* This declares a block-level flex container */
}
```

When we use a display value of "flex", we still see the same two flex containers; but this time they're stacked one on top of the other.



Here’s what’s going on. When we assign a value of “flex” or “inline-flex” to the CSS display property of an HTML element, we make that element a flex container; and we make its children flex items. A value of “flex” creates a block-level flex container, and a value of “inline-flex” creates (duh) an inline flex container.

> ***Block-level containers are displayed on separate lines. Inline containers can be displayed on the same line, if space is available. https://drafts.csswg.org/css-flexbox/.***

Chapter 4. Item Flow and Order

This chapter explains how to control the direction in which items flow in a flex container and the order in which they appear.

Flex items can be displayed in rows or columns in a flex container. And they can be laid out in any order – left to right, right to left, top to bottom, or bottom to top. This degree of control is provided through three new CSS properties: “flex-direction”, “flex-wrap”, and “order”.

Flex-Direction

The flex-direction property identifies the main axis of the flex container and determines how flex items are laid out along that axis. Flex-direction can take four values:

- **row (default):** Flex items are laid out in a row from left to right.
- **row-reverse**: Flex items are laid out in a row from right to left.
- **column**: Flex items are laid out in a column from top to bottom.
- **column-reverse**: Flex items are laid out in a column from bottom to top.

Here’s some HTML markup that that we’ll use for these examples.

HTML

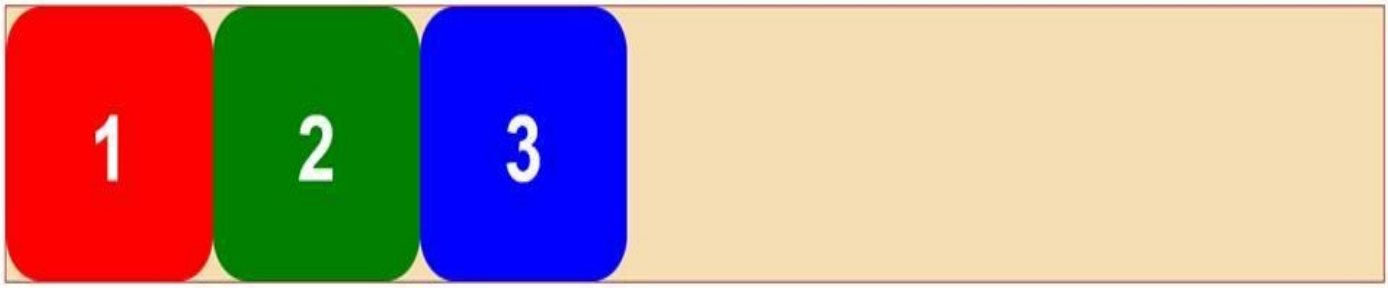
```
<div class=“container container-flex”>
    <span class=“item color1”>1</span >
    <span class=“item color2”>2</span >
    <span class=“item color3”>3</span >
</div>
```

And here is the CSS declaration block for the container-flex class:

CSS

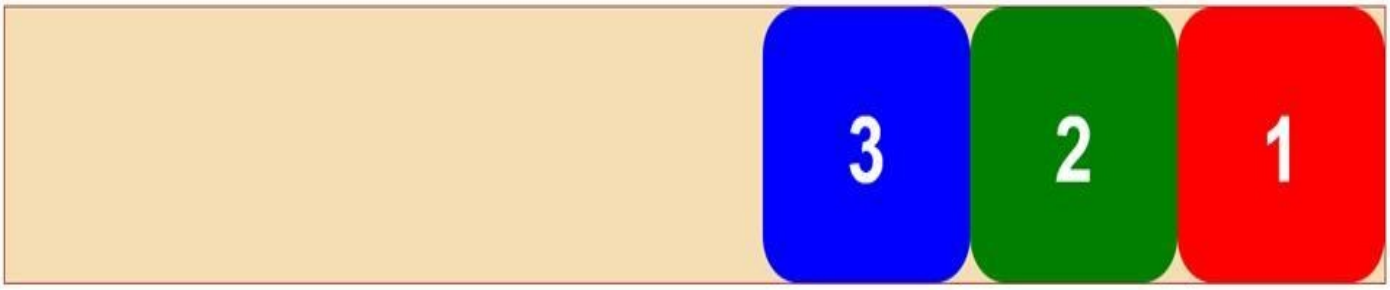
```
.container-flex {
    display: flex;
    flex-direction: row;      /* Items go left to right in a row */
}
```

Here, we are directing that items within the flex container be displayed from left to right in a row, which produces the following:

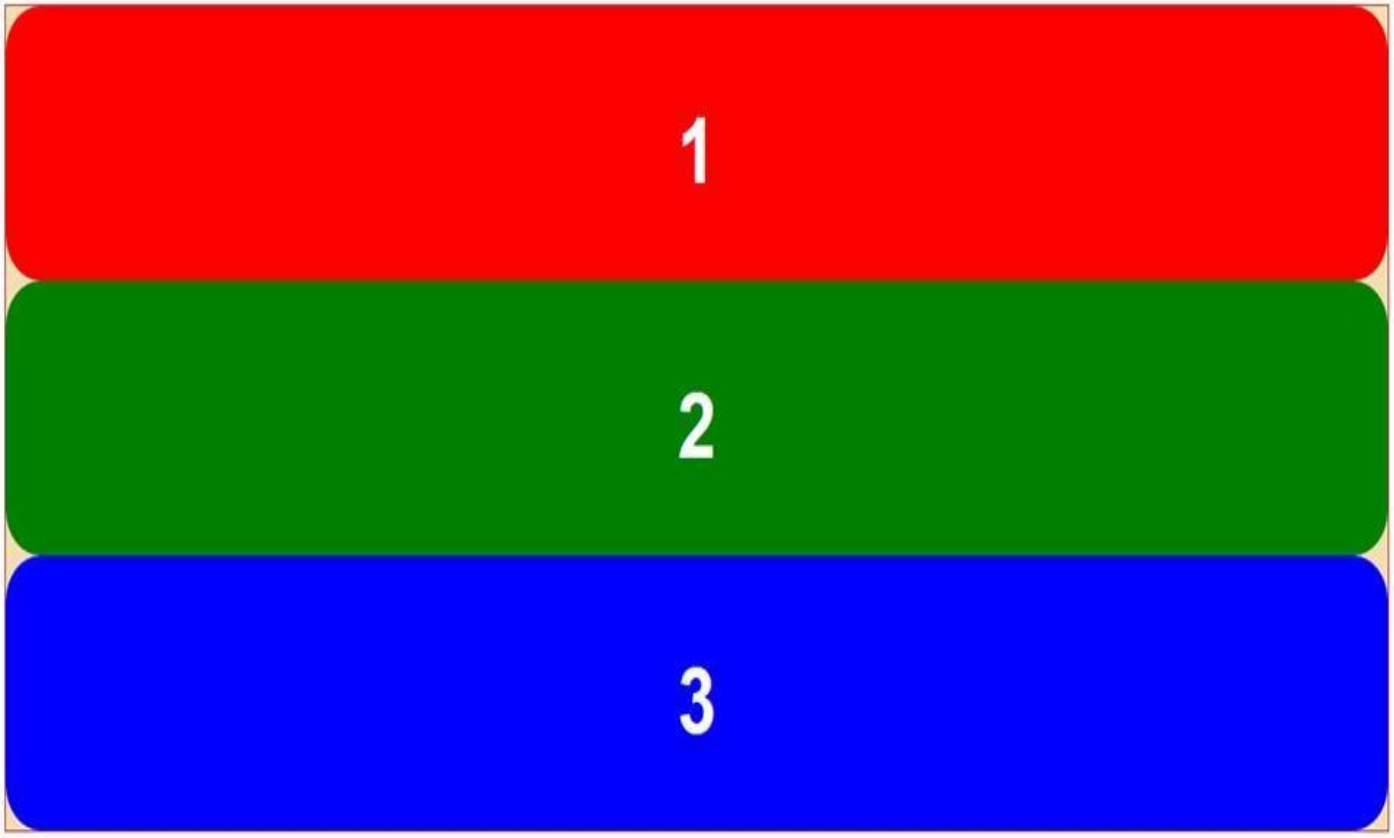


The CSS declaration “flex-direction: row” causes flex items to start at the left edge of the flex container and move right in a row.

If we change that declaration to “flex-direction: row-reverse”, flex items start at the right edge and move left in a row, as shown below:



The CSS declaration “flex-direction: column” causes flex items to start at the top edge of the flex container and move down in a column, like so:



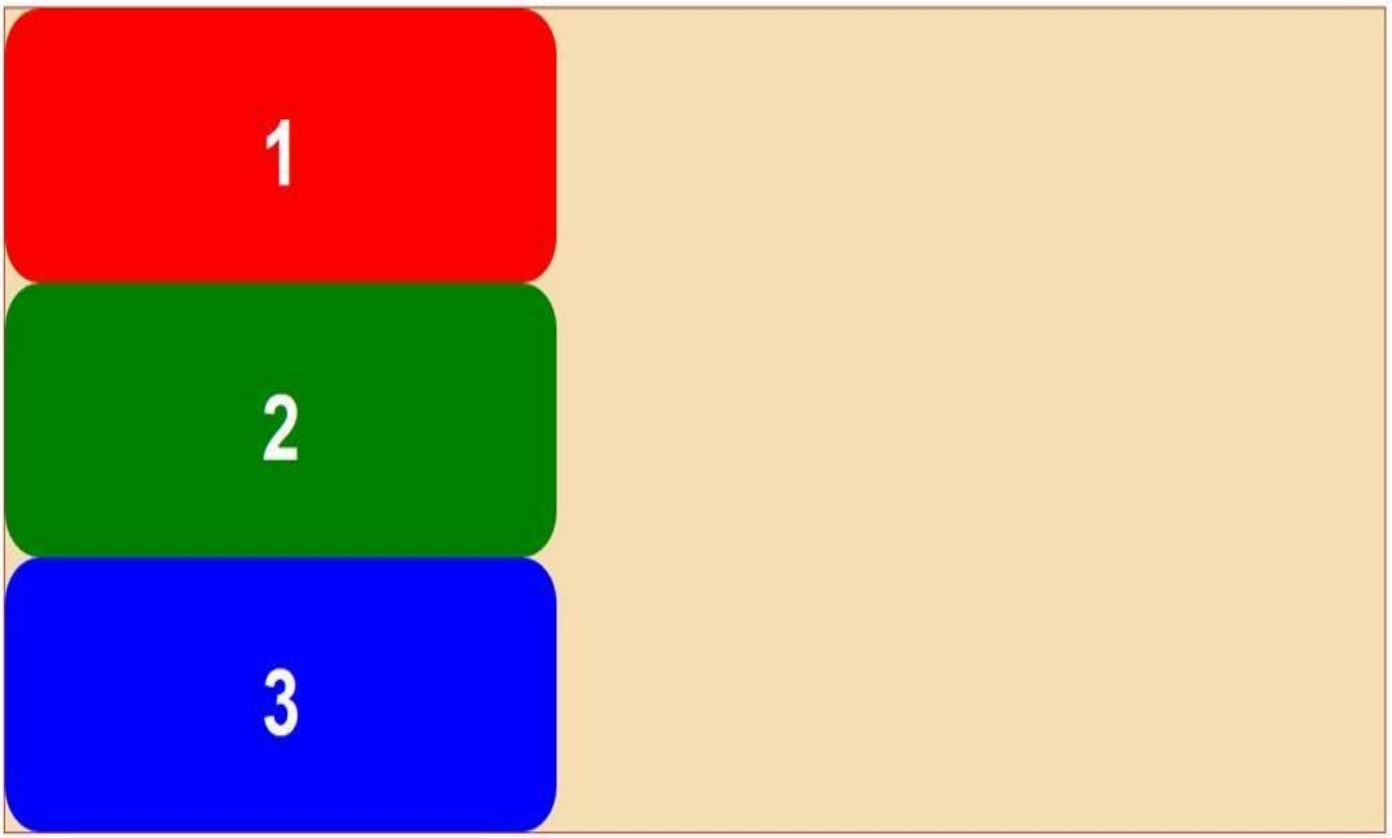
When the width of the flex item is not constrained, each flex item expands to fill the full width of the flex container, as shown above. We could override that behavior, if we wanted. For example, we could set the width of the flex items to

40% of their parent.

CSS

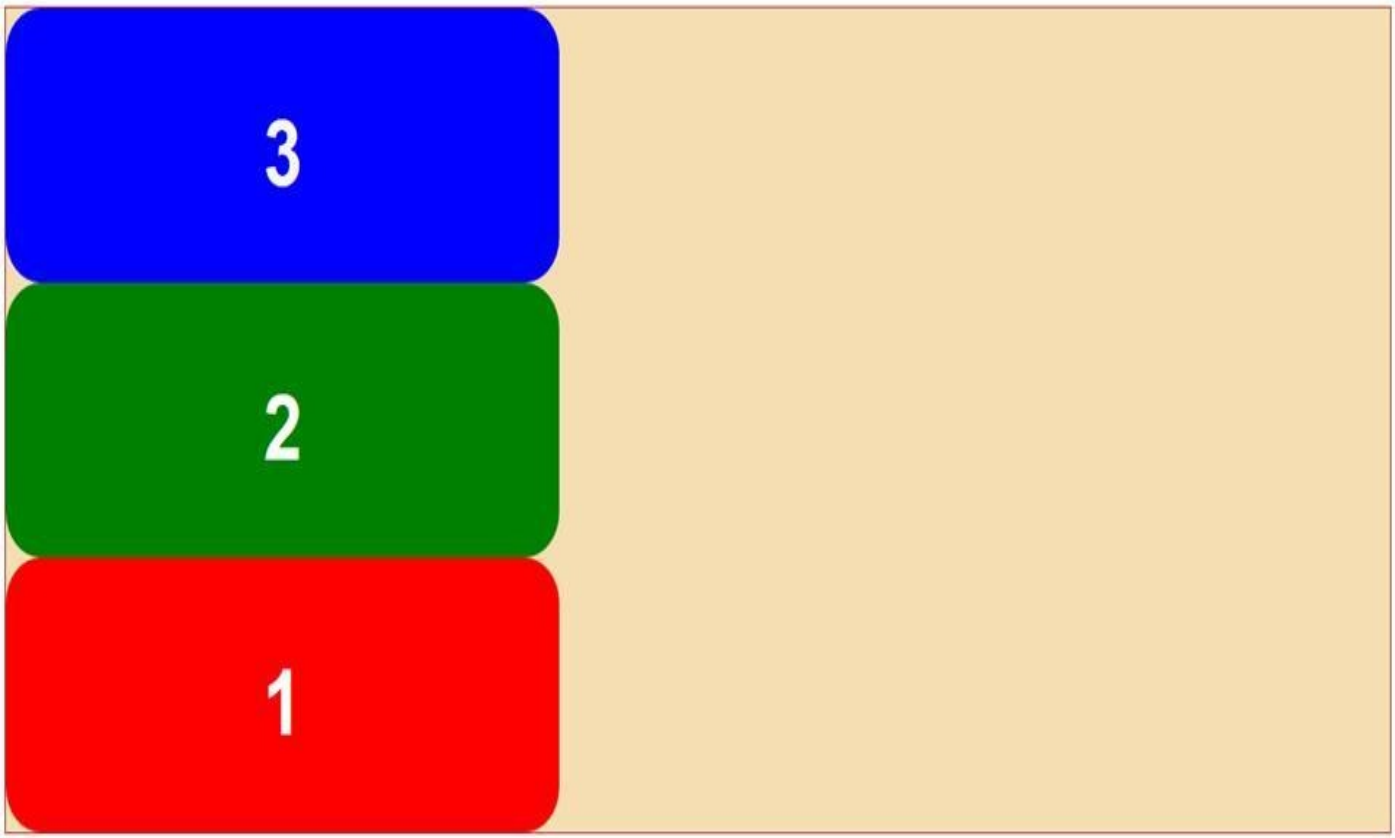
.item { padding: 10px; color: white; **width: 40%;** }

This would produce the following output.



Note that the flex items still move down the left side of the flex container in a column, but each item only occupies 40% of the space on its row. There is plenty of room for items on the right side of the container, but that excess space is not filled.

The CSS declaration “flex-direction: column-reverse” does just what you would expect.



Flex items are still displayed in a column, but they are in reverse order – going bottom to top rather than top to bottom.

Warning: Internet Explorer 10 and earlier versions do not support the flex-direction property. We'll deal with browser support issues in Chapters 7 and 8.

> ***NOTE***
>
> ***If you are working with a different writing system (like Hebrew where text runs from right to left or Chinese where text is laid out vertically), the values for flex-direction have different effects.***
>
> ***For further explanation, see Appendix A.***

Flex-Wrap

The flex-wrap property determines whether a flex container is single-line or multi-line. This property can take three values:

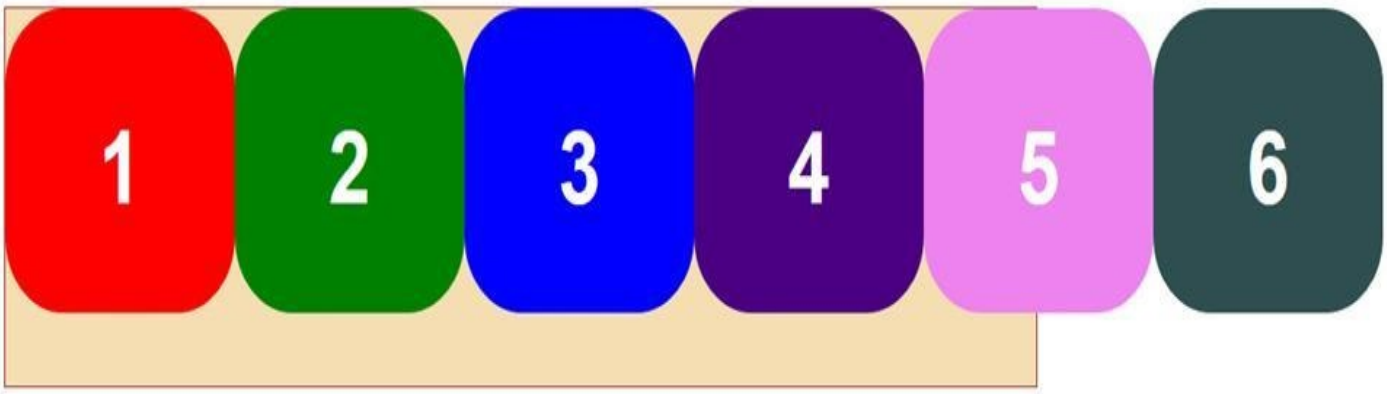
- **nowrap (default)**: Declares a single-line flex container. All flex items are displayed on a single line.
- **wrap:** Declares a multi-line flex container.
- **wrap-reverse:** Like wrap, declares a multi-line flex container.

When flex-wrap is assigned a value of “wrap”, additional flex lines are added along the cross axis moving in the direction of the current writing mode. Huh, what does that mean? Ok, here’s the translation. If you normally write one line after another with each new line beneath the line that preceded it, that is how additional lines will be added when flex-wrap is assigned a value of “wrap”.

When flex-wrap is assigned a value of “wrap-reverse”, additional flex lines are added in the opposite direction of the current writing mode. Thus, with “wrap-reverse”, each new line is added above the line that preceded it.

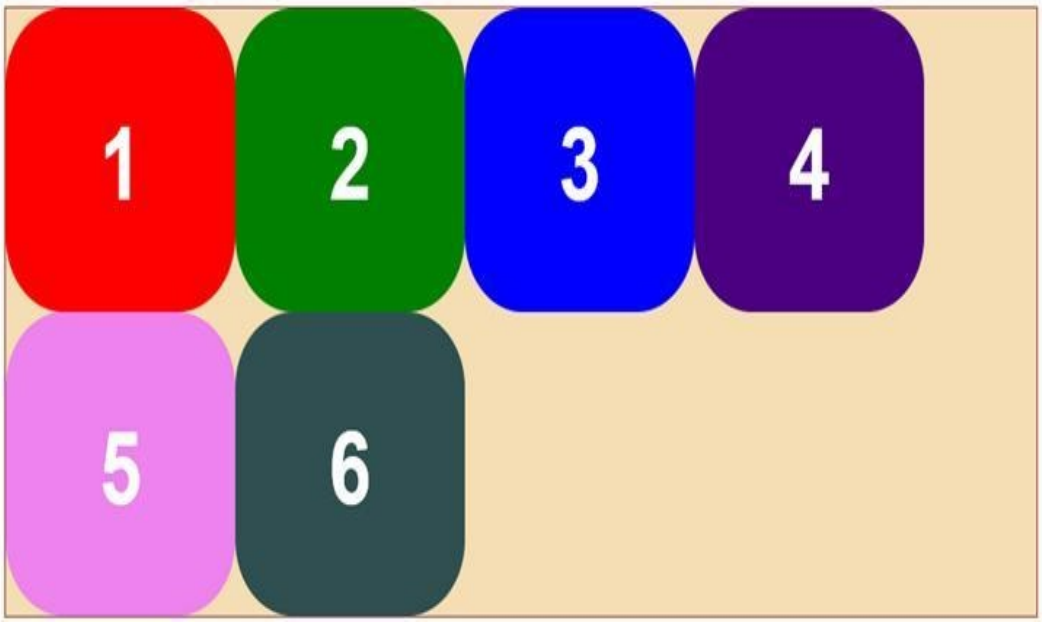
Let’s look at some examples. In each example the container has a width of 900 pixels, and the small boxes each have a minimum width of 200 pixels. There is not enough room to fit all six boxes on a single line in the container. What happens when we assign different values to the flex-wrap property?

The first example uses the following CSS declaration: “flex-wrap: nowrap” and here is the result.



Here, the “no-wrap” keyword creates a single-line flex container. The six flex items are displayed on a single line, breaking out of the flex container on the right side.

Suppose we use the CSS declaration “flex-wrap: wrap”, instead. Here’s what happens.



The “wrap” keyword creates a multi-line container; so the fifth and sixth boxes, which previously broke out of the container, are now displayed within the container on a second line below the first line. Notice that the flex container expands just enough in the vertical direction to hold two lines of flex items.

And finally, let’s see how things change when we use the CSS declaration “flex-wrap: wrap-reverse”.



This is interesting. Like the “wrap” keyword, the “wrap-reverse” keyword creates a multi-line container that displays the fifth and sixth boxes within the container on a second line. This time, though, the second line is displayed above the first line. See how “wrap-reverse” causes lines to be displayed within the flex container in reverse order of the current writing mode. We normally write down the page, but “wrap-reverse” causes lines to go up the page.

Warning: Internet Explorer 10 and earlier versions do not support the flex-wrap property. We’ll deal with browser support issues in Chapters 7 and 8.

Flex-Flow

Flex-flow is a shorthand way to set the “flex-direction” and “flex-wrap” properties. The syntax is:

flex-flow: [flex-direction] [flex-wrap]

For example, consider the CSS code below, which uses flex-flow;

CSS

```
.container-flex {
    display: flex;
    flex-flow: row wrap; }
```

The above code has exactly the same effect as:

CSS

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

In this book, we won't use flex-flow. Instead, we'll write the flex-direction and flex-wrap declarations longhand. And since we've already covered "flex direction" and "flex wrap" with examples, we won't work through additional flex-flow examples here.

Warning: Internet Explorer 10 and earlier versions do not support the flex-flow property. We'll deal with browser support issues in Chapters 7 and 8.

Order

The "order" property takes an integer value, which defines an ordinal group. A flex container lays out flex items in ascending order, according to their ordinal group. Thus, items in group 0 would be displayed before items in group 1; items in group 1 would be displayed before items in group 2; and so on.

If two or more flex items are assigned to the same ordinal group, they are displayed in the order that they appear in the HTML source document. That is, items that appear earlier in the document are displayed before items that appear later.

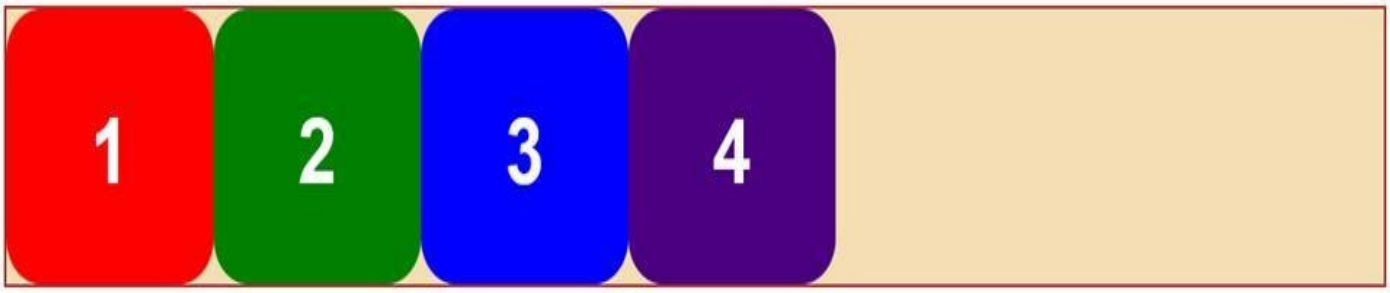
By changing the value assigned to the "order" property of an individual flex item, you can control whether that item appears before or after other items in its flex container. By default, flex items have an initial value of 0.

Consider the following example – a flex container containing four flex items.

HTML

```
<div class="container container-flex">
    <span class="item color1">1</span >
    <span class="item color2">2</span >
    <span class="item color3">3</span >
    <span class="item color4">4</span >
</div>
```

This markup produces the following output:



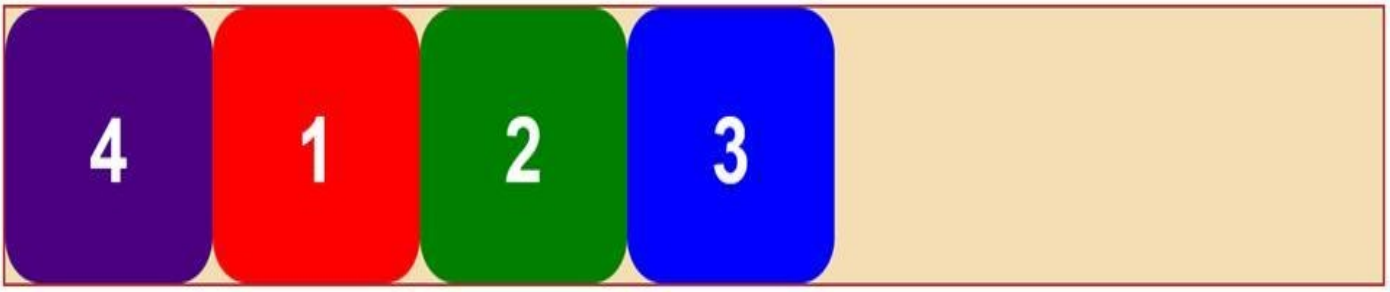
No surprises here. Each numbered box has a default order value of 0, so each box is in the same ordinal group. And all of the boxes within the same ordinal group are displayed in the order in which they appear in the HTML source document.

Suppose we changed the HTML for Box 4 just a little bit by assigning Box 4 an order value of -1, as shown below.

HTML

```
<div class="container container-flex">
    <span class="item color1">1</span >
    <span class="item color2">2</span >
    <span class="item color3">3</span >
    <span class="item color4" style="order: -1;">4</span >
</div>
```

Now, Box 4 is in a different ordinal group than all of the other boxes. All of the other boxes still have their default order value of 0, while Box 4 has been assigned a new order value of -1. Because it is in a group with a lower order value, Box 4 is displayed before all of the other boxes, as shown below.



For those who care about search engine optimization (SEO), the order property lets you position page elements on the page, independent of where they appear in source code. You could, for example, position the main content first in the HTML source code, even if it is displayed last on the web page. In this way, search engine robots can see the main content first, regardless of where it is displayed on the web page.

Warning: Internet Explorer 10 and earlier versions do not support the order property. We'll deal with browser support issues in Chapters 7 and 8.

Chapter 5. Item Alignment

In this chapter, we'll cover flexbox properties that are used to align flex items. We'll show how to align flex items with respect to container edges, and with respect to other items within a flex container.

Justify-Content

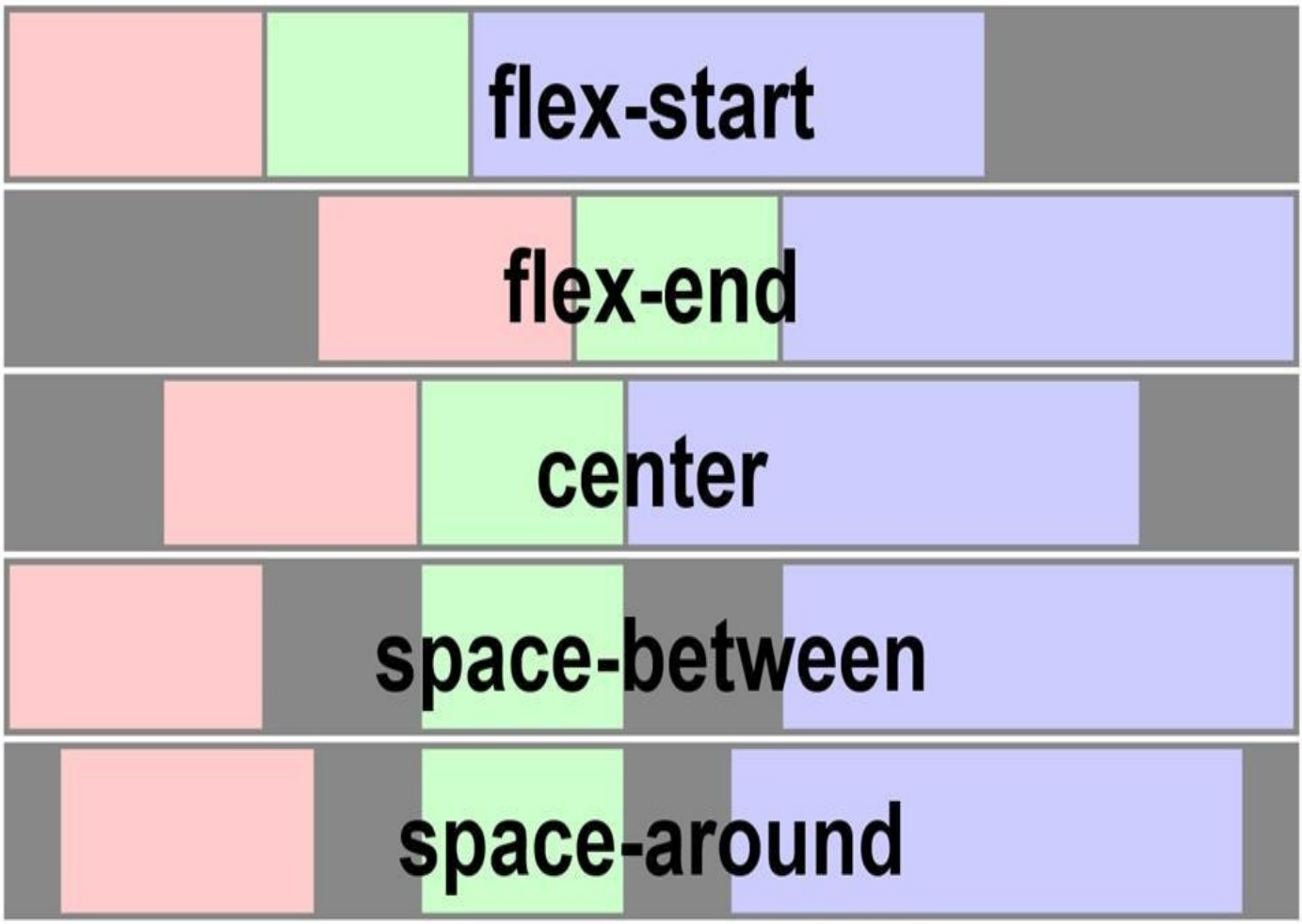
The justify-content property aligns flex items along the main axis of the current line of the flex container. It comes into play when:

- Flex items do not use all of the available space on the main-axis.
- Flex items require more space than is available and overflow the flex container.

Justify-content has five possible values:

- **flex-start (default):**The first flex item is placed at the start of the current line. Each subsequent item is placed flush with the item before it.
- **flex-end:** The last flex item is placed flush with the main end side of the current line. Then, each preceding flex item is added flush with the subsequent item.
- **center:** Flex items are packed on the current line with no space between them. Extra space is distributed equally on the right and left sides of the packed items. If the flex container is too small to hold all the items, flex items overflow equally in both directions.
- **space-between:** If (a) there is only one flex item on the line or (b) the packed flex items overflow the flex container, this value is identical to "flex-start". Otherwise, the first flex item is flush with the start of the current line; the last item is flush with the end of the current line; and free space is distributed equally between the remaining items.
- **space-around:** If (a) there is only one flex item on the line or (b) the packed flex items overflow the flex container, this value is identical to "center". Otherwise, flex items are distributed such that (a) there is equal space between adjacent items and (b) space between the end items and flex container edge is half the space between adjacent items.

That might sound complicated, but it's really not. This chart, from the Worldwide Web Consortium (W3C) at https://drafts.csswg.org/css-flexbox/#justify-content-property, shows how things work when flex items do not overflow the flex container.



Before we leave this topic, let’s work through two examples to show how justify-content works when flex items overflow the flex container. This would occur with the following HTML and CSS.

HTML

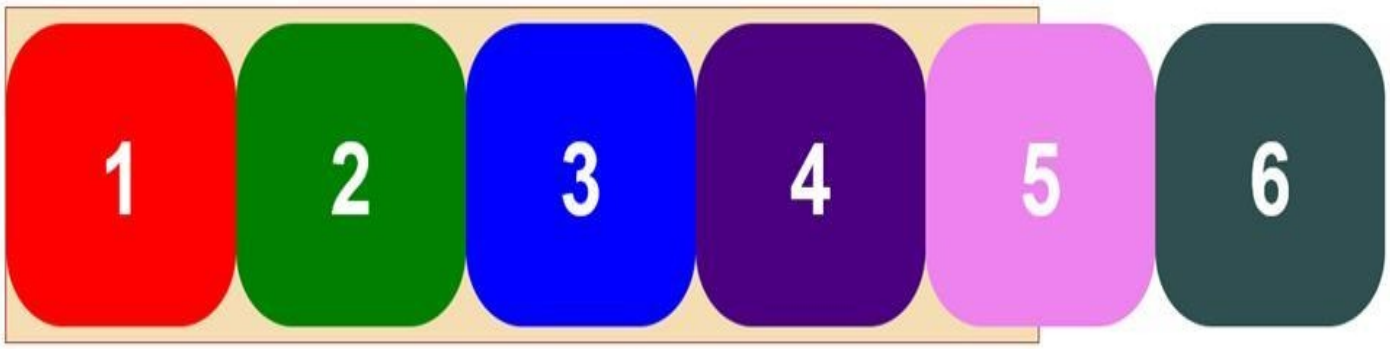
```
<div class=“container container-flex”>
    <span class=“item color1”>1</span >
    <span class=“item color2”>2</span >
    <span class=“item color3”>3</span >
    <span class=“item color4”>4</span >
    <span class=“item color5”>5</span >
    <span class=“item color6”>6</span >
</div>
```

Here is what happens when we use the CSS declaration “justify-content: space-between”.

CSS

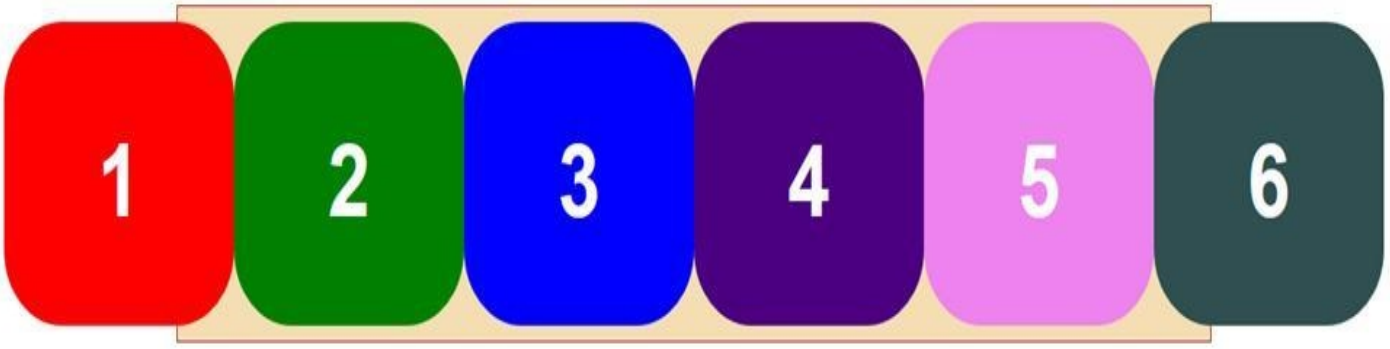
```
.container { width: 900px; padding: 10px 0px; }
.item { min-width: 200px; }
```

```
.container-flex {
    display: flex;
    justify-content: space-between;     /* Identical to flex-start */
}
```

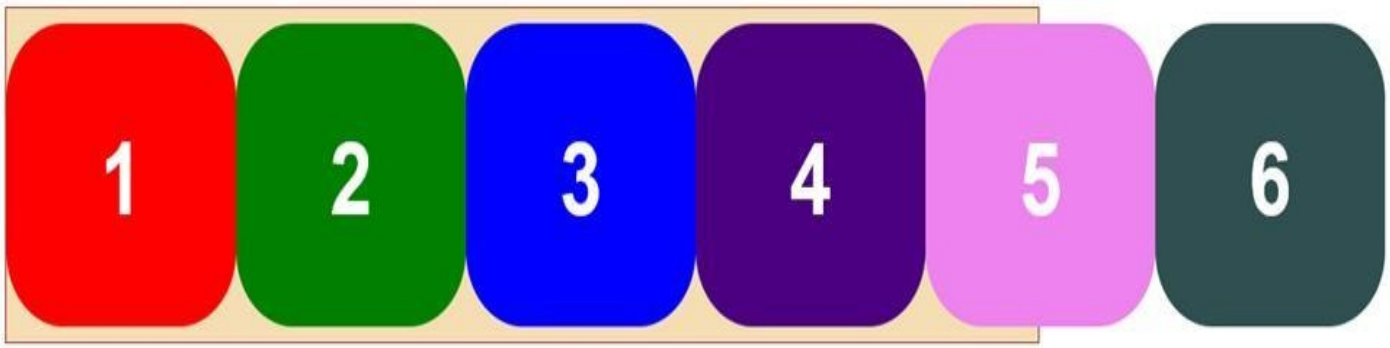


The flex container is too small to hold all of the flex-items, so items are packed at the start of the current line and break out of the container on the right. We would have seen the same result if we had used the CSS declaration “justify-content: flex-start”.

If we use the CSS declaration “justify-content: space-around”, we get an interesting result. According to the Worldwide Web Consortium, we should see the flex items packed, centered, and overflowing the too-small container on both sides. And this is what occurs with most modern browsers, as shown below.



But it doesn’t work this way with Internet Explorer 11 – at least not on my computer. With Internet Explorer 11, we see this result.



This is the same result that we would get with “justify-content: flex-start” or with “justify-content: space-between” – not what we would expect with “justify-content:

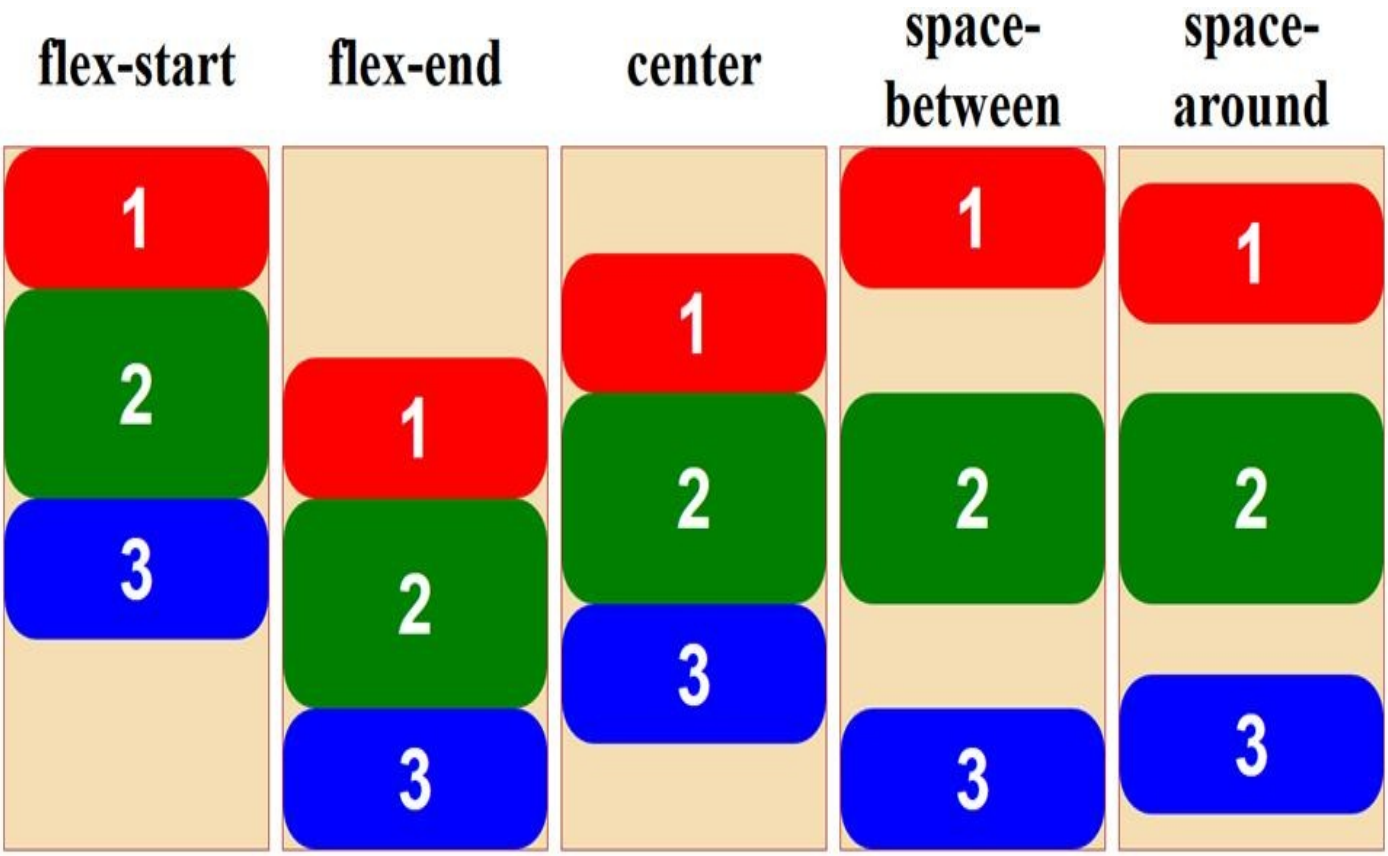
space-around", based on the most recent Worldwide Web Consortium specs (see https://drafts.csswg.org/css-flexbox/#justify-content-property). It looks like Internet Explorer 11 may have a bug.

All of the examples, so far, have examined the justify-content property with flex-direction set to "row". Suppose flex-direction is set to "column", as shown below.

CSS

```
.container-flex {
    display: flex;
    flex-direction: column;      /* Identical to flex-start */
}
```

The figure below shows how each keyword for "justify content" affects the layout of flex items when flex-direction is set to "column".



Notice that each keyword works the same way in a column as it works in a row.

- **flex-start (default):** Flex items are positioned at the start of the flex container (i.e., at the top of the column).
- **flex-end:** Items are positioned at the end of the container (i.e., at the bottom of the column).

- **center:** Items are positioned in the middle of the container (i.e., in the middle of the column).

- **space-between:** There is equal space between items, with the first item up against the top of the column and the last item against the bottom.

- **space-around:** There is space before, between, and after the items (equal space between items and half that space between items and the column top and bottom).

In short, when flex-direction is set to “row” or “row-reverse”, the main axis is a horizontal axis; and justify-content positions flex items along the horizontal axis. But when flex-direction is set to “column” or “column-reverse”, the main axis is a vertical axis; and justify-content positions flex items along the vertical axis.

Warning: Internet Explorer 10 and earlier versions do not support the justify-content property. We’ll deal with browser support issues in Chapters 7 and 8.

Align-Items

The align-items property sets the way that flex items are aligned in the cross axis of the current line of a flex container. This property can take five values:

- **stretch (default):** Flex items stretch to fill the flex container, while respecting constraints imposed by min-width, max-width, min-height, and max-height.

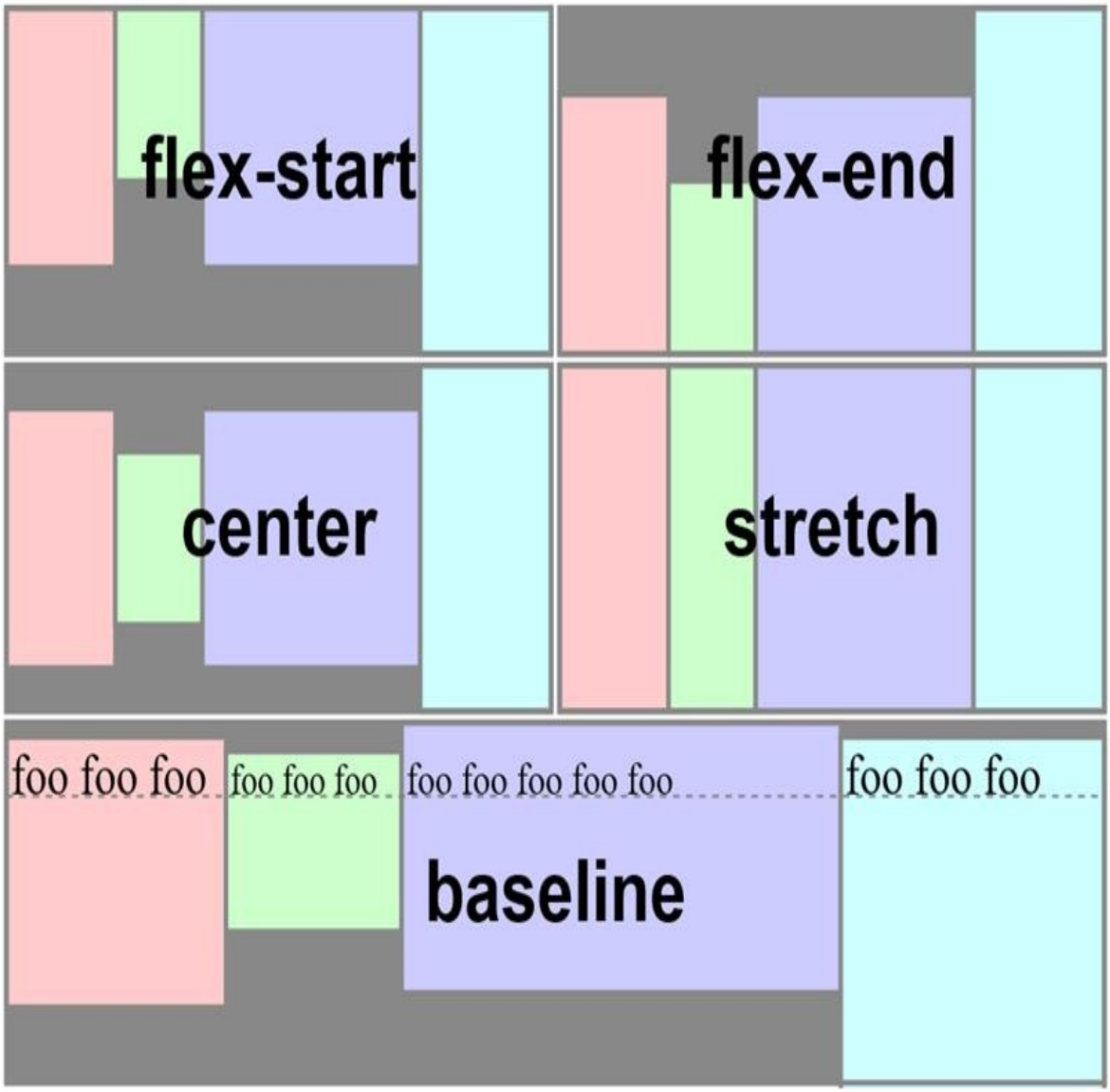
- **flex-start:** Items are placed flush with the top of the flex container.

- **flex-end:** Items are placed flush with the bottom of the flex container.

- **center:** Items are centered along the cross axis of the container.

- **baseline:** Items are positioned such that their baselines align; and the item with the largest distance between its baseline and its cross start is placed flush against the cross start of the line.

At their site (https://drafts.csswg.org/css-flexbox/#align-items-property), the Worldwide Web Consortium prepared a figure to show how each “align-items” keyword affects the vertical alignment of flex items in a row flex container. That figure is reproduced below:

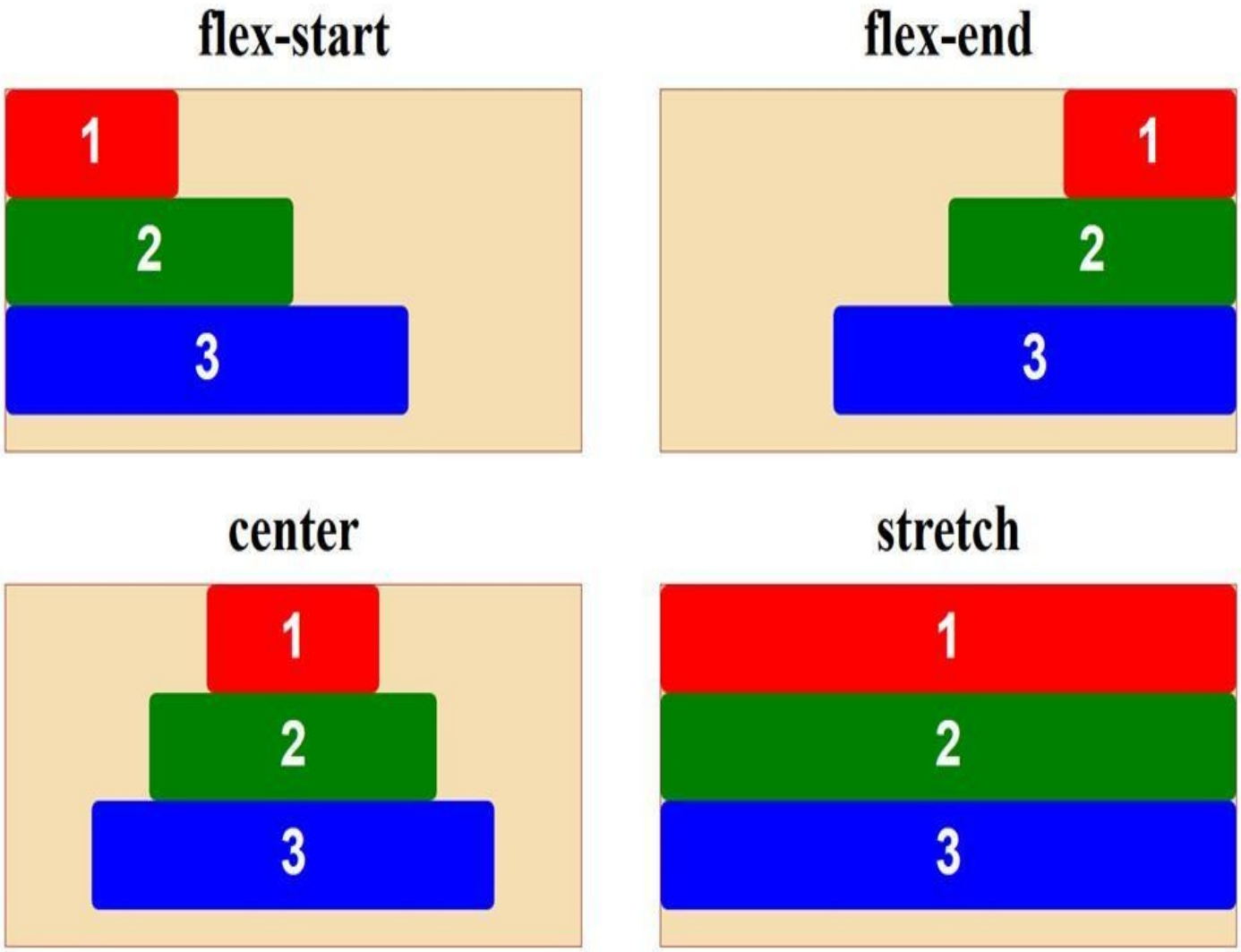


You might ask how the “align-items” property works in a column flex container, as would be the case with the CSS markup shown below.

CSS

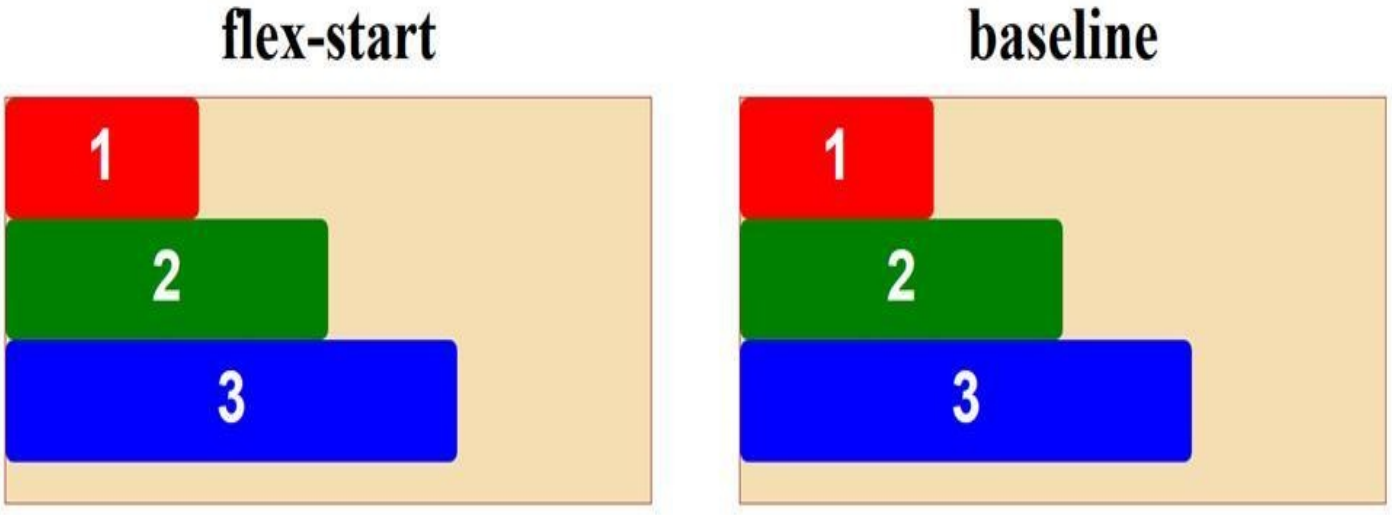
```
.container-flex {
    display: flex;
    flex-wrap: nowrap;          /* Create single-line flex container */
    flex-direction: column;     /* Display flex items in column */
}
```

The next figure shows some effects of the “align-items” property when flex items are laid out in a column.



Each of the keywords shown in the above figure has a unique effect. Flex-start displays items on the left edge of the column. Flex-end displays items on the right edge. Center displays items centered along the horizontal axis. And stretch stretches each item from the left to right edge of the column.

The only “align-items” keyword not shown above is “baseline”. When flex items are laid out in a column, “baseline” has the exact same effect as “flex-start”, as shown below:



In a column flex container, both CSS declarations - “align-items: baseline” and “align-items: flex-start” - position flex items flush against the left edge of the column.

Warning: Internet Explorer 10 and earlier versions do not support the align-items property. We’ll deal with browser support issues in Chapters 7 and 8.

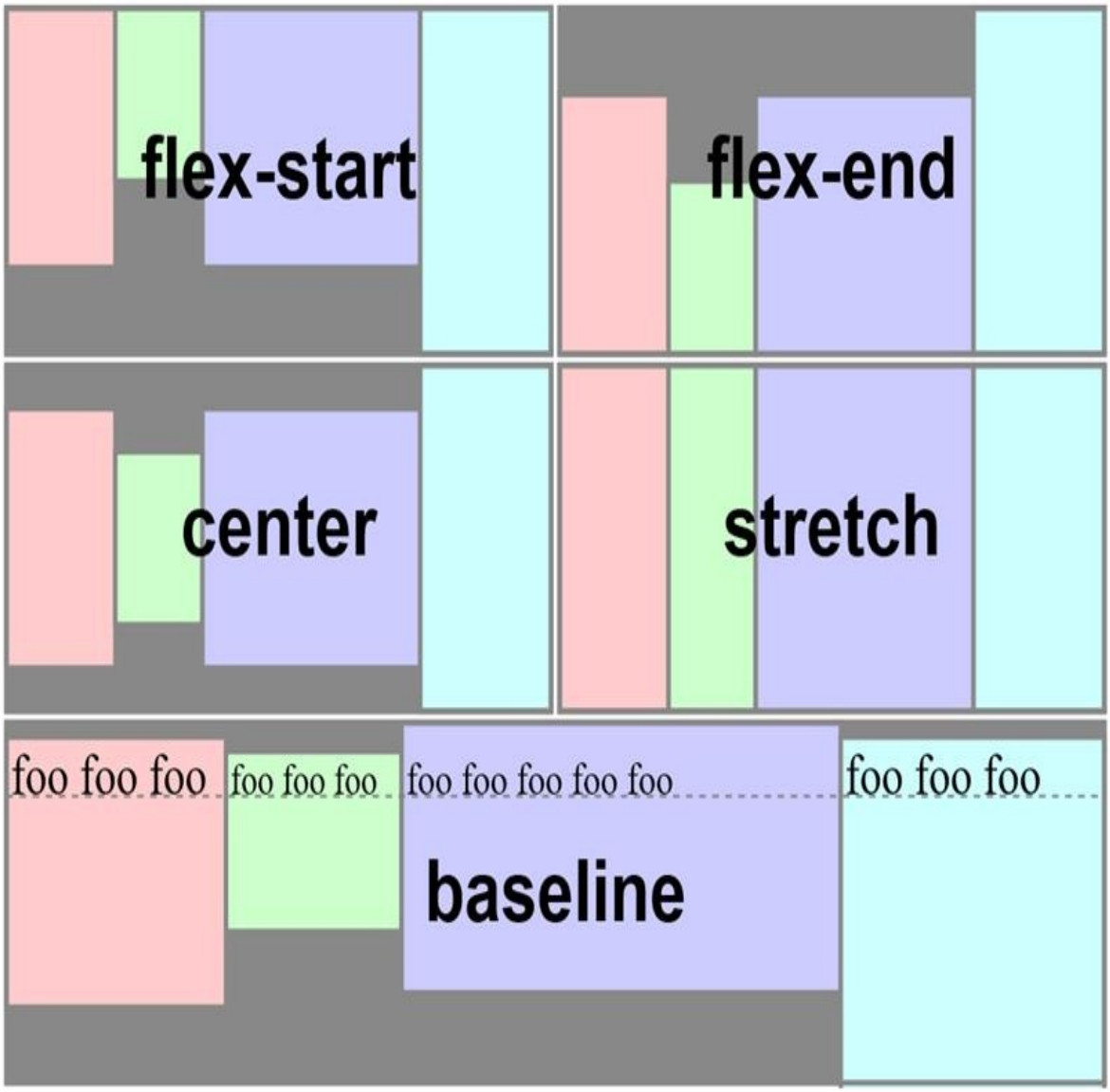
Align-Self

As we just learned, the “align-items” property sets the default cross axis alignment for *all* flex items in a container. The “align-self” property overrides the “align-items” setting for an *individual* flex item.

This “align-self” property can take the same five value as the “align-items” property, plus a sixth value – “auto”:

- **auto (default):** The flex item inherits the value of its parent’s “align-items” property. If the item has no parent, “auto” gets the value “stretch”.
- **stretch:** Flex items stretch to fill the flex container, while respecting constraints imposed by min-width, max-width, min-height, and max-height.
- **flex-start:** Items are placed flush with the top of the flex container.
- **flex-end:** Items are placed flush with the bottom of the flex container.
- **center:** Items are centered along the cross axis of the container.
- **baseline:** Items are positioned such that their baselines align; and the item with the largest distance between its baseline and its cross start is placed flush against the cross start of the line.

A few pages ago, you saw a figure prepared by the Worldwide Web Consortium (see https://drafts.csswg.org/css-flexbox/#align-items-property.) The figure shows how each “align-items” keyword affects the vertical alignment of flex items in a row flex container. The “align-self” keywords work the same way, so the figure is shown again below.

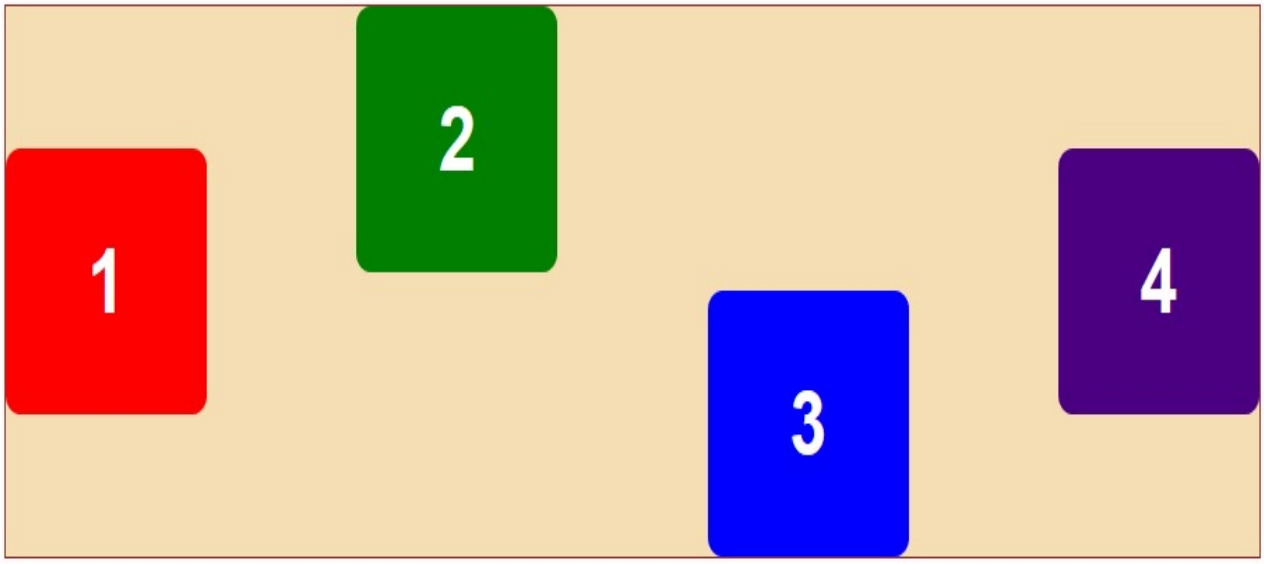


Let’s work through an example to show how the “align-self” property works. We’ll use this HTML to produce four small boxes in a larger box:

HTML

```
<div class=“container-flex”>
    <span class=“box1”>1</span >
    <span class=“box2”>2</span >
    <span class=“box3”>3</span >
    <span class=“box4”>4</span >
</div>
```

Suppose we wanted to lay out the small boxes within the large box as shown below:



Here is the CSS to make that happen:

CSS

```
.container-flex { display: flex; justify-content: space-between; }
.box1 { align-self: center; }
.box3 { align-self: flex-end; }
.box4{ align-self: center; }
```

And here is what is going on. The “display:flex” declaration turns the big box into a flex container and the small boxes, into flex items. The “justify-content” property controls spacing along the main axis (in this case, the horizontal axis). The “justify-content: space-between” declaration sets the first flex item flush with the start of the flex line and the last item flush with the end of the flex line. Free space along the main axis is distributed equally between the remaining items. Without any additional instructions, all of the boxes would be positioned flush against the top edge of the container, like Box 2.

However, the “align-self” property overrides the default vertical positioning for Boxes 1, 3, and 4. The “align-self: center” declaration centers Boxes 1 and 4 vertically. The “align-self: flex-end” declaration positions Box 3 flush with the bottom of the flex container.

Warning: Internet Explorer 10 and earlier versions do not support the “align-self” property. We’ll deal with browser support issues in Chapters 7 and 8.

Auto Margins

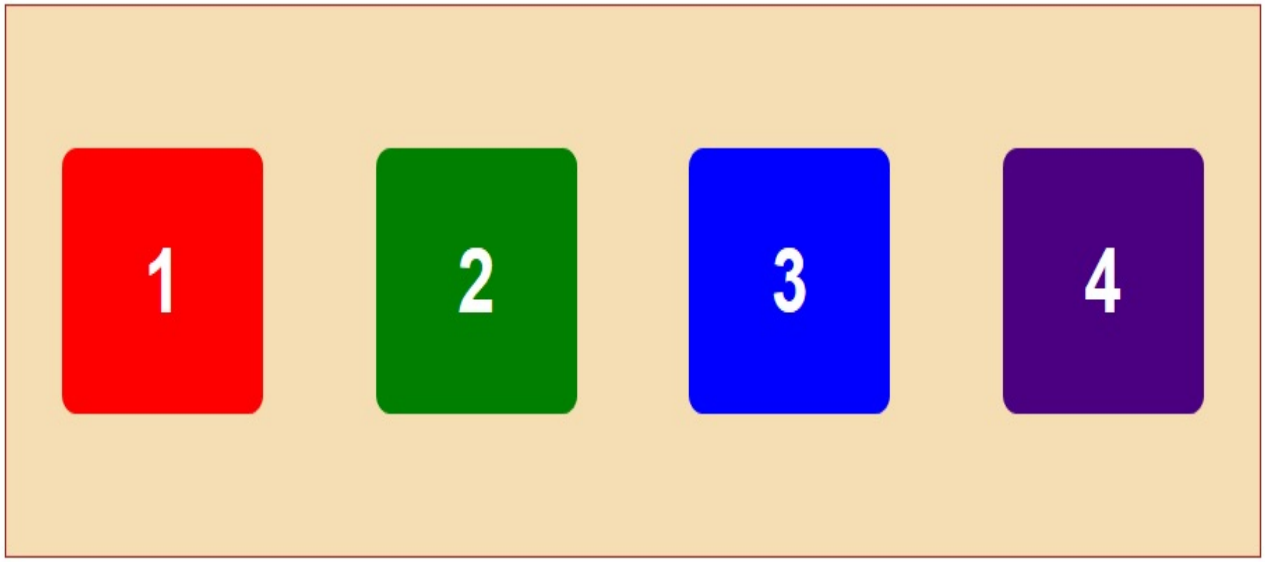
In addition to the properties we’ve covered so far – “justify-content”, “align-items”, and “align-self” – you can also align flex items with auto margins. When you set the “margin” property equal to “auto” in any direction, the browser allocates all of the free space to that direction.

You can produce many useful effects when you assign auto margins to flex items. To illustrate some of the possibilities, let's play around with the following HTML

HTML

```
<div class="container-flex">
    <span class="item box1">1</span >
    <span class="item box2">2</span >
    <span class="item box3">3</span >
    <span class="item box4">4</span >
</div>
```

Suppose we wanted to center the small boxes horizontally and vertically within the larger container. That is, suppose we wanted to produce a layout like this:



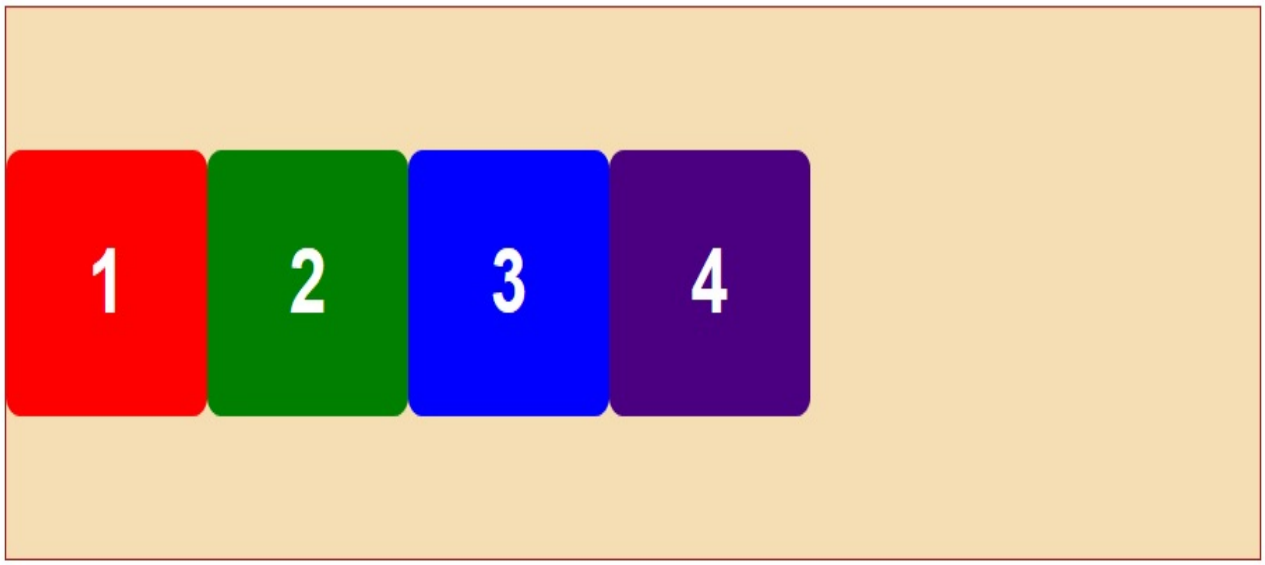
We could use this CSS markup to produce the layout shown above:

CSS

```
.container-flex { display: flex; }
.item { margin:auto; }
```

The vertical free space is allocated equally above and below the small boxes, leaving them vertically centered. The horizontal free space is allocated such that (a) there is equal space between adjacent items and (b) space between the end items and flex container edge is half the space between adjacent items.

Suppose we wanted to center the small boxes on the vertical axis and slide them left on the horizontal axis, as shown below:

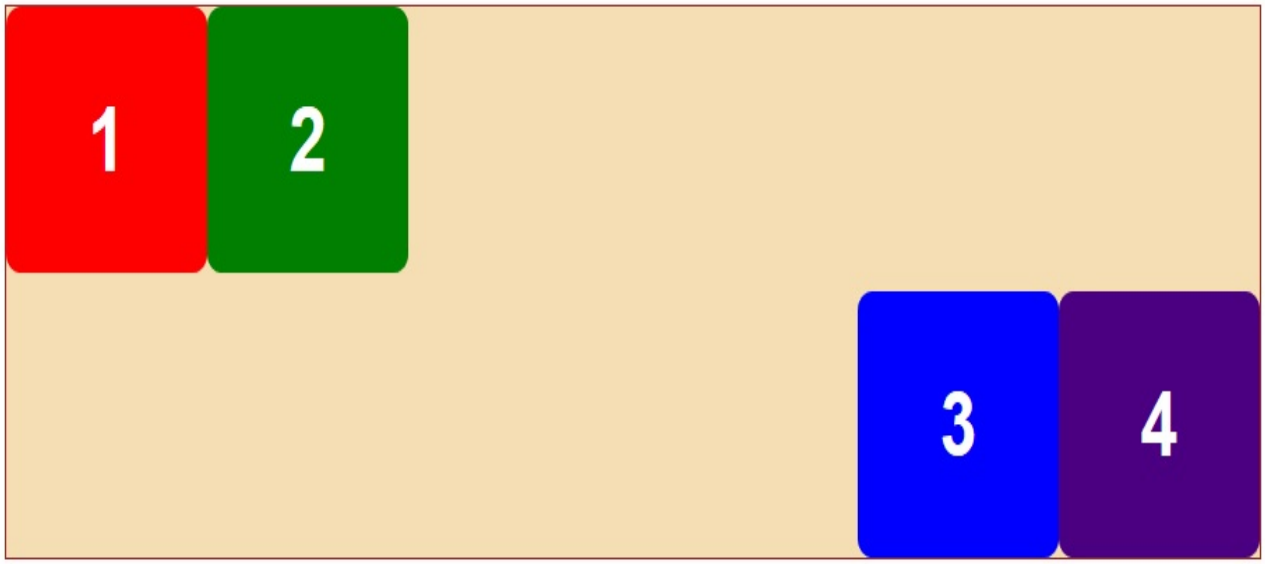


How did we produce this result? Just a small tweak to value assigned to the “margin” property:

CSS

```
.container-flex { display: flex; }
.item { margin: auto 0; }
```

And finally, here’s one last example. Suppose we wanted the first two boxes positioned in the upper left corner and the last two boxes positioned in the lower right corner, like so:



We could use this CSS markup to produce the layout shown above:

CSS

```
.container-flex { display: flex; }
.box1 { margin:0; }
.box2 { margin:0; }
```

```
.box3 { margin:auto 0 0 auto; }
.box4 { margin:auto 0 0 0; }
```

With no margins, Boxes 1 and 2 are positioned in their default location – starting in the upper left corner and moving right. Boxes 3 and 4 each have their “margin-top” values set to auto. This causes the browser to allocate all of the free vertical space above each box, which pushes Boxes 3 and 4 to the bottom edge. And Box 3 has its “margin-left” value set to “auto”. This causes the browser to allocate all of the free horizontal space to the left of Box 3, which pushes Boxes 3 and 4 to the far right. When all is said and done, Boxes 1 and 2 wind up in the upper left corner; and Boxes 3 and 4 wind up in the lower right corner.

Note: When a flex item margin is set to “auto” on the horizontal axis, the alignment properties – “justify-content, “align-items”, and “align-self” –have no effect in that direction. Similarly, when a flex item’s margin is set to “auto” on the vertical axis, the alignment properties have no effect in that direction. That is, any free space left over after flexing is assigned to the margins. If the alignment properties are not behaving, make sure they are not being overridden by a margin auto setting.

Align-Content

So far, we have been talking about aligning flex items within flex containers. Sometimes, though, you might want to align flex lines.

The “align-content” property aligns flex lines within a flex container. It can take six possible values:

- **stretch (default):** Lines stretch to take up the remaining space. Free space is split equally between all lines. If there is no free space, this value has the same effect as “flex-start”.

- **flex-start:** Lines are packed toward the cross-start edge of the flex container, starting with the first flex line.

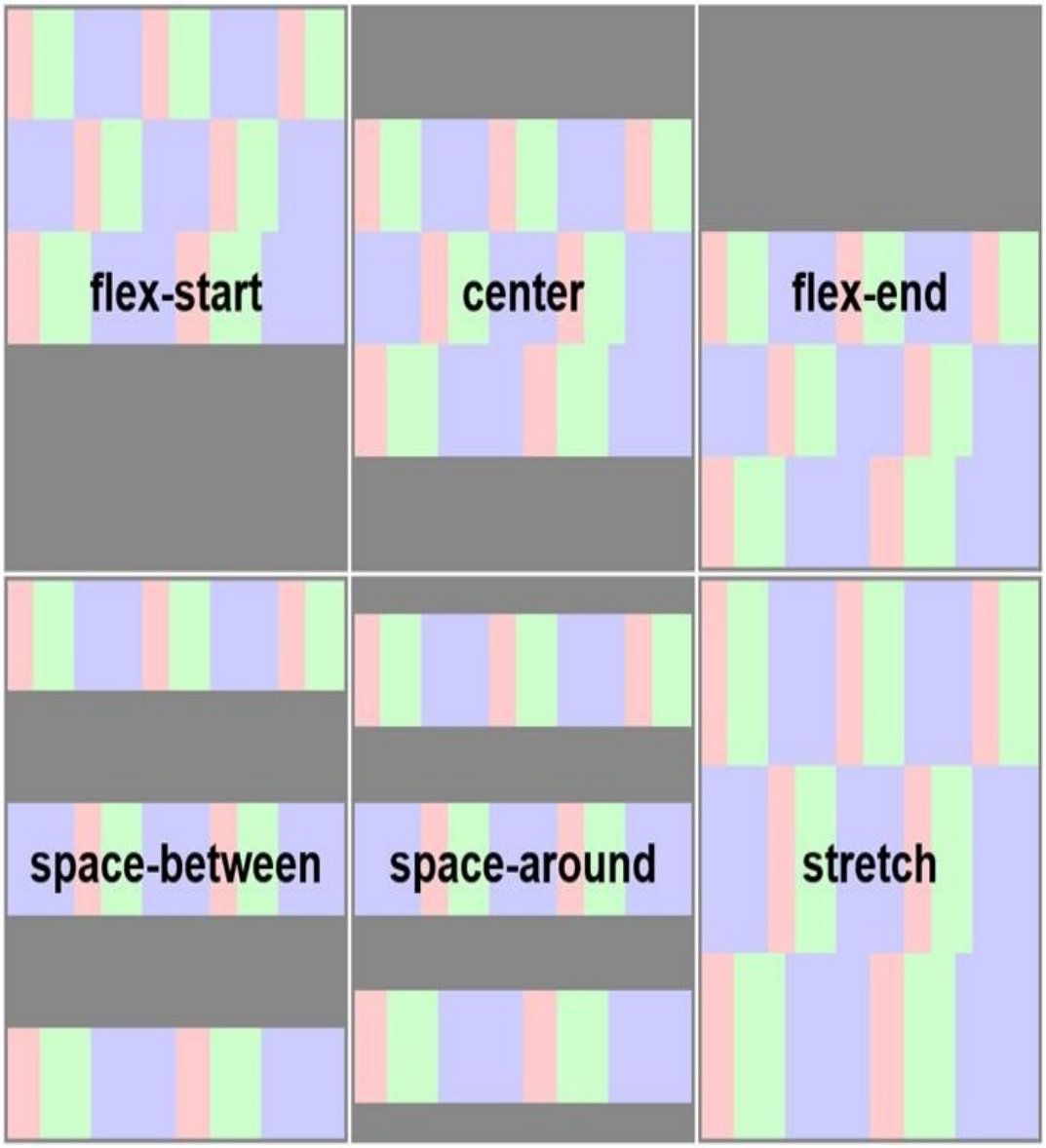
- **flex-end:** Lines are packed toward the cross-end edge of the flex container, starting with the last flex line.

- **center:** Lines are packed toward the center of the flex container. If the flex container is too small to hold all of the flex lines, the line overflow equally on both ends.

- **space-between:** Lines are evenly distributed in the flex container; that is, free space is distributed equally between flex lines. If there is no free space between lines, this value has the same effect as “flex-start”.

- **space-around:** Lines are evenly distributed in the flex container, with equal space between lines and half-size spaces on either end. If there is no free space between lines, this value has the same effect as “center”.

The Worldwide Web Consortium prepared a figure to show how each “align-content” keyword affects the vertical alignment of flex lines in a row flex container. This figure, found at https://drafts.csswg.org/css-flexbox/#align-content-property, is reproduced below.



One final point: The “align-content” property only has an effect when the flex container has at least two flex lines.

Warning: Internet Explorer 10 and earlier versions do not support the “align-content” property. We’ll deal with browser support issues in Chapters 7 and 8.

Chapter 6. Item Flexibility

In this chapter, we'll cover flexbox properties that control the ability of individual flex items to expand and contract within a flex container.

Flexibility

The flexbox model is all about flexibility. As a flex container grows or shrinks, flex items can grow or shrink to fit space available. Or they can be held to a fixed size, regardless of changes in container size.

Three properties are available to control the flexibility of flex items:

- **flex-basis:** This property specifies the initial main size of a flex item.
- **flex-shrink:** This property specifies how much a flex item will shrink relative to other flex items in a flex container.
- **flex-grow:** This property specifies how much a flex item will grow relative to other flex items in a flex container.

A flex item is totally inflexible if its "flex-shrink" value is 0 and its "flex-grow" value is 0. Otherwise, it is flexible.

The Worldwide Web Consortium warns against using these properties directly in your markup (see https://drafts.csswg.org/css-flexbox/#flex-components). Instead, they recommend using a shorthand property called "flex" to assign values indirectly to "flex-basis", "flex-shrink", and "flex-grow". The "flex" property includes useful defaults that are not set by "flex-basis", "flex-shrink", and "flex-grow".

Even if you don't use "flex-basis", "flex-shrink", and "flex-grow" in your markup, you need to understand them; because understanding them will make it easier to use the shorthand "flex" property correctly. So let's begin our discussion of flexibility with a consideration of three properties that we may never use directly in our markup - "flex-basis", "flex-shrink", and "flex-grow".

Flex-Basis

A flex container can grow or shrink along its main axis. And a flex item can grow or shrink in response to changes in the size of its flex container. The flex-basis property specifies the initial main size of a flex item, before the item is re-sized to accommodate changes in the size of its flex container.

Flex-basis can take three possible values:

- **auto:** The auto keyword retrieves the value of the main size property of the flex item. If the main axis is horizontal, the main size property would be the width of the flex item; if the main axis is vertical, the main size property would

be the height of the flex item.

- **content:** Content indicates automatic sizing, based on the flex item's content. Avoid using this keyword, because it is not yet well-supported.

- **<'width'>:** For all other values, flex basis is resolved using the same units as the CSS width property (pixels, percentages, ems, etc.).

Note: The "content" keyword is a new addition to flexbox. As this is being written, it is not supported by all browsers. For the time being, you should not use the "content" keyword. You can produce the same result by using 'auto' together with a main size ('width' or 'height') of 'auto'. To see this approach in action, check out Box 2 in the following example.

Consider the HTML below.

HTML

```
<div class="container container-flex ">
    <span class="item color1" style="width: 200px; flex-basis: auto;">1</span >
    <span class="item color2" style="width: auto; flex-basis: auto;">2</span >
    <span class="item color3" style="flex-basis: 600px; width: 200px;">3</span >
</div>
```

This markup produces the following output:



In this example, the "flex-basis" property sets the width of each box. Here's an explanation of what's going on.

- **Box 1:** Here, "flex-basis" was set to "auto". This instructed the browser to base the size of Box 1 on its width property (in this case, 200 pixels).

- **Box 2:** Again, "flex-basis" was set to "auto", but the width property was also set to "auto". This caused the browser to base the width of Box 2 on its content, so Box 2 was displayed just wide enough to show its content - the number "2".

- **Box 3:** The "flex-basis" for Box 3 was 600 pixels, so Box 3 was displayed 600

pixels wide.

Note that the “flex-basis” property overruled the “width” property for Box 3. The “width” property called for a width of 200 pixels, and the “flex-basis” property called for a width of 600 pixels. Ultimately, Box 3 was displayed at 600 pixels. When there is a conflict between “width” and “flex-basis”, “flex-basis” wins.

Warning: Internet Explorer 10 and earlier versions do not support the flex-basis property. We’ll deal with browser support issues in Chapters 7 and 8.

Flex-Shrink

When there is not enough space for all of the flex items in a row to fit within a flex container, the flex container is said to have **negative free space**, and some or all of the flex items may shrink to fit within the container.

When a flex container has negative free space, the “flex-shrink” property is one of two factors that determine how much a flex item will shrink relative to other items in the flex container. The amount of shrinkage depends on:

- **flex-shrink factor:** A non-negative number assigned to the “flex-shrink” property. The default value of the flex-shrink factor is 1.

- **flex base size:** If the “flex-basis” property is defined by an assigned value, rather than by content, then the flex base size is the size specified by the “flex-basis” property. (If flex-basis is defined by content, then a different method is used to find flex base size. In this book, we aren’t using content to define flex-basis; so we won’t worry about other methods.)

The flex shrink factor is multiplied by the flex base size to compute something that we’ll call a “shrinkage factor”. When a flex container has negative free space, the amount that a flex item will shrink depends on the size of its shrinkage factor, relative to shrinkage factors of other flex items. For example, suppose Item A has a shrinkage factor of 3 and Item B has a shrinkage factor of 1. When a flex container has negative free space, Item A will lose three times as much space to shrinkage as Item B.

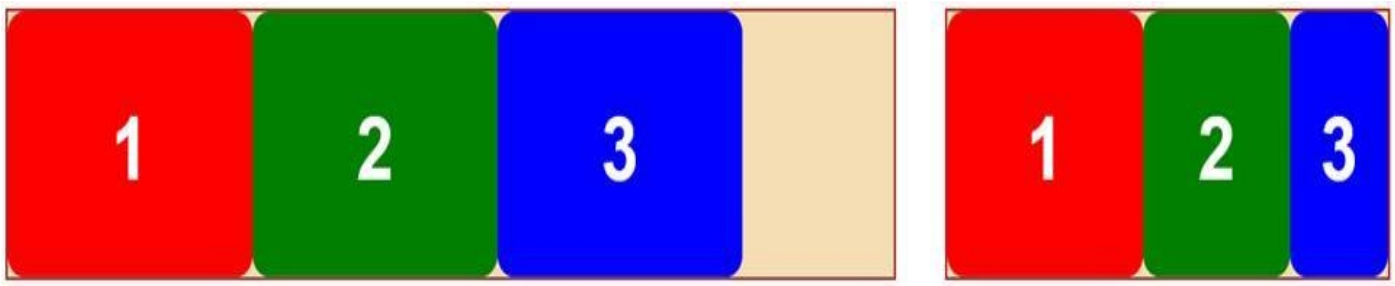
Let’s look at an example, based on the following HTML.

HTML

```
<div class=“container container-flex” style=“width: 800px;”>
    <span class=“item color1” style=“flex-basis: 200px; flex-shrink: 1;”>1</span
    >
    <span class=“item color2” style=“flex-basis: 200px; flex-shrink: 2;”>2</span
    >
    <span class=“item color3” style=“flex-basis: 200px; flex-shrink: 3;”>3</span
```

```
    >
  </div>
  <div class="container container-flex" style="width: 400px;">
    <span class="item color1" style="flex-basis: 200px; flex-shrink: 1;">1</span
    >
    <span class="item color2" style="flex-basis: 200px; flex-shrink: 2;">2</span
    >
    <span class="item color3" style="flex-basis: 200px; flex-shrink: 3;">3</span
    >
  </div>
```

This markup produces the output shown below.



The flex container on the left is 800 pixels wide, and each box in the container is 200 pixels wide. The container is big enough to hold all three boxes with room to spare. The "flex-shrink" property does not come into play.

The flex container on the right is a different story. The container is only 400 pixels wide – not big enough to hold all three boxes at their initial size. To fit the boxes within the container, we need to shrink one or more boxes. To determine how much space to remove from each box, the browser computes a shrinkage factor (SF) for each box.

Box 1 SF = FBS * FSF = 200 * 1 = 200

Box 2 SF = FBS * FSF = 200 * 2 = 400

Box 3 SF = FBS * FSF = 200 * 3 = 600

where

SF = shrinkage factor

FBS = flex base size

FSF = flex shrink factor

The shrinkage factor for Box 2 is twice as big as the shrinkage factor for Box 1, so the browser removes twice as much space from Box 2 as from Box 1. Similarly, the shrinkage factor for Box 3 is three times as big as the shrinkage factor for Box 1, so the browser removes three times as much space from Box 3 as from Box 1.

Thus, in the container on the right, Box 1 ends up being bigger than Box 2; and Box 2 ends up being bigger than Box 3.

The main takeaway here is that the “flex-shrink” property gives a web designer the ability to control how much an individual flex item will shrink, whenever shrinkage is called for; i.e., whenever the flex container is too small to hold all of its flex items at their base size.

Warning: Internet Explorer 10 and earlier versions do not support the flex-shrink property. We’ll deal with browser support issues in Chapters 7 and 8.

Flex-Grow

When a flex container expands, extra space is created within the container. The “flex-grow” property determines how this extra space within an expanded flex container is used. For example, extra space might be used to increase the amount of free space within the container. Or it might be used to increase the size of one or more flex items within the container

The “flex-grow” property accepts a non-negative number that serves as a proportion. This proportion defines how much each flex item will grow relative to other flex items, as the flex container increases in size. The default value of the flex-grow property is zero.

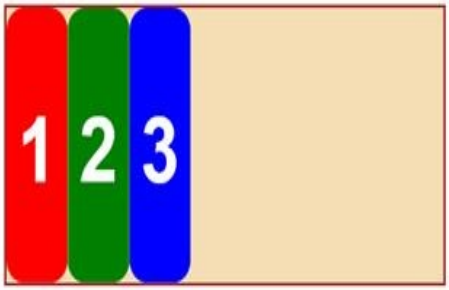
When the flex-grow value of a flex item is zero, that flex items does not grow any larger when the flex container increases in size.

Consider the HTML below.

HTML

```
<div class=“container container-flex “>
    <span class=“item color1” >1</span >
    <span class=“item color2” >2</span >
    <span class=“item color3” >3</span >
</div>
```

The flex-grow property is not specified in the markup; so each flex item is assigned its default flex grow value, which is zero. As a result, we would expect flex items to stay the same size when the flex container expands. And that is exactly what happens, as shown below:





Above, the figure on the left shows the flex container in its initial state. The figure on the right shows the flex container expanded to twice its initial width. The flex container on the right gets bigger, but each of the flex items inside stay the same size. All of the extra space that is created when the container expands is added to the free space within the container.

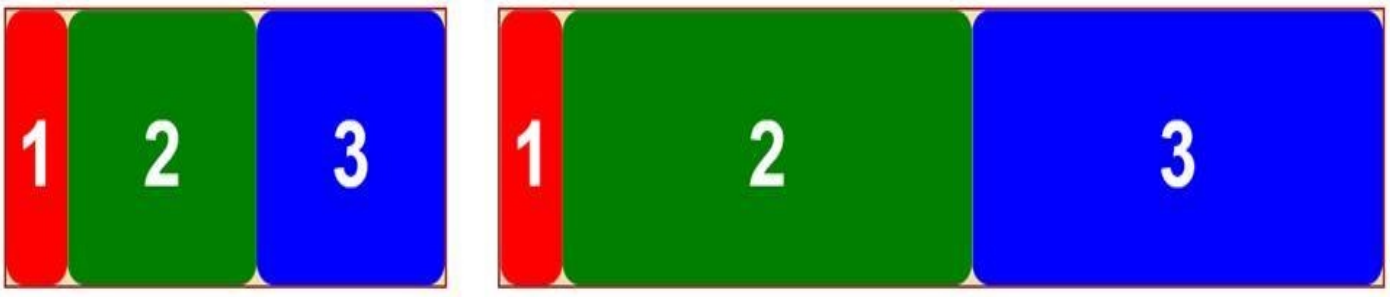
Because each of the flex items has a “flex-grow” value of zero, there is a lot of unused space in each of the above containers. We can use the “flex-grow” property to allocate that unused space to one or more of the boxes within the container. And we can specify what proportion of unused space each box should get.

For example, suppose we wanted Box 1 to stay the same size as its container expanded; that is, we wanted Box 1 to get none of the unused space. And we wanted Boxes 2 and 3 to grow, sharing all of the “extra” space within the container equally. We could produce this result by adding the following “flex-grow” declarations to our markup:

HTML

```
<div class=“container container-flex “>
    <span class=“item color1” style=“flex-grow: 0;”>1</span >
    <span class=“item color2” style=“flex-grow: 1;”>2</span >
    <span class=“item color3” style=“flex-grow: 1;”>3</span >
</div>
```

Setting the “flex-grow” property of Box 1 to zero tells the browser to hold the width of Box 1 constant as its container expands. And assigning equal “flex-grow” values to Boxes 2 and 3 tells the browser to assign any “extra” space within the container equally to Boxes 2 and 3. And here is the result:



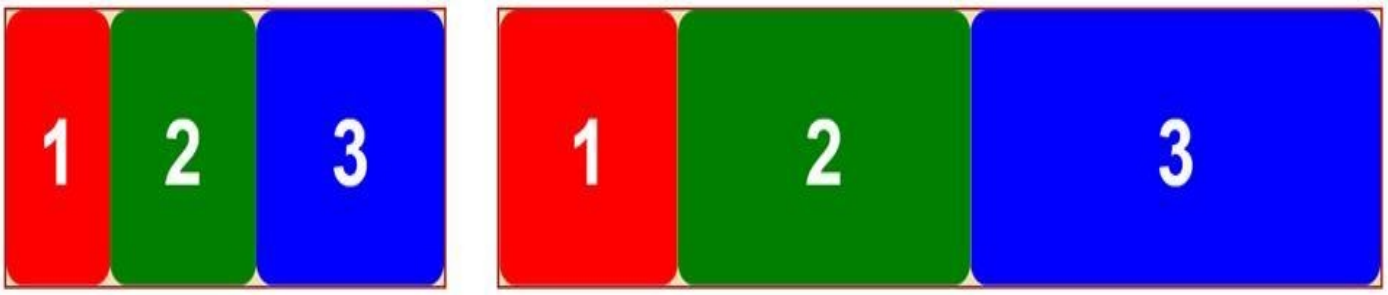
Again, the figure on the left shows the flex container in its initial state, and the figure on the right shows the container expanded to twice its initial width. Box 1 is the same size in both containers because its “flex-grow” property is set to zero. Boxes 2 and 3 share the extra space in each container equally, because they have equal “flex-grow” values.

Let’s look at one more example. As the flex container expands, suppose we want Box 2 to get twice as much unused space as Box 1; and we want Box 3 to get three times as much unused space as Box 1. Here’s the markup we could use

HTML

```
<div class="container container-flex ">
    <span class="item color1" style="flex-grow: 1;">1</span >
    <span class="item color2" style="flex-grow: 2;">2</span >
    <span class="item color3" style="flex-grow: 3;">3</span >
</div>
```

The "flex-grow" values define the proportion of unused space that each box should get as the flex container expands. In this example, Box 1 will get one-sixth of the unused space; Box 2 will get one-third of the unused space; and Box 3 will get one-half of the unused space. Here's what it looks like.



As you see, the proportions defined by the "flex-grow" property are maintained as the flex container expands. Box 1 gets one-sixth of the unused space, Box 2 gets one-third, and Box 3 gets one-half.

Warning: Internet Explorer 10 and earlier versions do not support the flex-grow property. We'll deal with browser support issues in Chapters 7 and 8.

Flex

The "flex" property is a shorthand way to assign values to the "flex-grow", "flex-shrink", and "flex-basis" properties. The syntax for the "flex" property is:

flex: <flex-grow> <flex-shrink> <flex-basis>

where

- **<flex-grow>** is the value of the "flex-grow" property.
- **<flex-shrink>** is the value of the "flex-shrink" property.
- **<flex-basis>** is the value of the "flex-basis" property.

The second and third parameters ("flex-shrink" and "flex-basis") are optional. The default value for the "flex" property is flex: 0 1 auto.

NOTE

The Worldwide Web Consortium advises designers not to use the

longhand “flex-grow”, “flex-shrink”, or “flex-basis” properties directly. They recommend using the shorthand “flex” property instead (see https://drafts.csswg.org/css-flexbox/#flex-property). The shorthand “flex” property sets some defaults that may not be handled well by the longhand properties.

In this book, we will use the shorthand option exclusively.

Three keywords can also be used to specify the “flex” property:

- **Flex: none** is equivalent to flex: 0 0 auto. The flex item does not shrink or expand to fit its container.
- **Flex: auto** is equivalent to flex: 1 1 auto. The flex item adjusts its size proportional to all of the other flex items in the container.
- **Flex: initial** is equivalent to flex: 0 1 auto. The flex item does not grow as its flex container expands. It shrinks proportional to other flex items as its flex container shrinks.

With all three of the above keywords, the “flex-basis” value is “auto”. This means the initial size of the flex item is determined by its width or height property, whichever runs along its main axis.

Consider the HTML below.

HTML

```
<div class=“container container-flex” style=“width: 800px;”>
    <span class=“item color1” style=“width: 200px; flex: 0 0 auto;”>1</span >
    <span class=“item color2” style=“width: 200px; flex: 1 1 auto;”>2</span >
    <span class=“item color3” style=“width: 200px; flex: 0 2 auto;”>3</span >
    <span class=“item color4” style=“width: 200px; flex: 2 0 auto;”>4</span >
</div>
```

This markup produces the container shown at the top of the following chart. The container is 800 pixels wide, and each box within is 200 pixels wide. There is exactly enough room to hold all of the boxes within the container. None of the boxes need to be expanded or shrunk.



The middle container shows what happens when the container width is reduced to 600 pixels. There is not enough room to hold all of the boxes, so one or more boxes will need to be shrunk.

Boxes 1 and 4 have a “flex-shrink” value of zero, so they do not shrink at all. The amount of shrinkage experienced by the other two boxes will be determined by their “flex-shrink” value and their flex base size. Boxes 2 and 3 have an identical flex base size (200 pixels), so the relative shrinkage that they experience can be determined from their “flex-shrink” values alone. Box 2 has a “flex-shrink” value of 1 and Box 3 has a “flex-shrink” value of 2. Since the “flex-shrink” value for Box 3 is twice as big as the value for Box 2, Box 3 experiences twice the shrinkage as Box 2.

The bottom container shows what happens when the container width is stretched to 1200 pixels. There is extra room within the container, so one or more boxes can stretch to fill the extra space. The “flex-grow” value (i.e., the first parameter in the “flex” property) determines which boxes grow and how much they grow. Box 1 and Box 3 have a “flex-grow” value of zero, so they do not grow at all. Box 2 has a “flex-grow” value of 1, and Box 4 has a “flex-grow” value of 2. Space is added to Boxes 2 and 4 in proportion to their “flex-grow” values. Since the “flex-grow” value for Box 4 is twice as big as the value for Box 2, Box 4 gets twice as much extra space as Box 2.

Here’s the main takeaway from this example: The “flex” property allows you to control the flex behavior of individual flex items. You can allow them to grow or shrink when their container changes size. Or you can hold them to a constant size. This capability will come in handy in coming chapters, when we begin to

build responsive web pages.

Warning: Internet Explorer 9 and earlier versions do not support the flex property. We'll deal with browser support issues in Chapters 7 and 8.

Part 2. Browser Support Strategies

The main challenge to widespread use of flexbox is browser compatibility. Although flexbox is recognized by all modern browsers, some people still use older browsers (like Internet Explorer 9) that do not fully support flexbox. Rats!

This part of the book describes two strategies that allow us to build web pages that are fully functional on browsers that don't support the most recent version of flexbox:

- Chapter 7. Vendor Prefixes
- Chapter 8. Progressive Enhancement

The vendor prefix strategy is easy to implement, but it only delivers a small improvement in browser support. The progressive enhancement strategy requires more effort, but it produces web pages that work on all browsers – even browsers that don't support flexbox.

> ***NOTE***
>
> ***As this is being written, 81% of browsers worldwide fully support flexbox. For a current assessment of browser support for flexbox, visit http://caniuse.com/#search=flex.***

Chapter 7. Vendor Prefixes

To counter the lack of support for flexbox in some older browsers, you can add vendor prefixes to your stylesheets. This won't solve the problem for every browser, but it will solve the problem for some browsers.

Note: Feel free to skip this chapter if you are already familiar with vendor prefixes and routinely add vendor prefixes to your CSS stylesheets.

What are Vendor Prefixes?

Before a new feature is added to CSS, the feature goes through a test phase. Vendors – browser makers – sometimes implement the new feature while it is being tested. Between the time when testing begins and when the new feature is actually added to CSS, things change. For example, the syntax used during testing may differ from the syntax that is ultimately implemented in CSS.

This is what happened with flexbox. During testing, several browsers implemented a preliminary version of flexbox. They did not implement the modern version of flexbox, because the modern version did not yet exist. As a result, today we have some fairly recent browsers (e.g., Internet Explorer 10, Safari 8, Safari 6.1) that do not support the current version of flexbox, but do support an early version of flexbox.

Each browser used a "vendor prefix" to specify its version of each flexbox property that it supported. For example,

- **-ms-** (vendor prefix for Internet Explorer)
- **-webkit-** (vendor prefix for Safari)

We can use these vendor prefixes to access the early version of flexbox. In this way, we can provide flexbox support in some older browsers. To see how we might use vendor prefixes, consider this example:

```
.container-flex { display: inline-flex; }
```

That's how the "display: inline-flex" declaration would look, using syntax understood by modern browsers. But if we wanted this declaration to be understood by Internet Explorer 10, Safari 8, and Safari 6.1 as well, this is the markup we would need to use:

```
.container-flex {
    display: -webkit-inline-box;      /* Syntax for Safari */
    display: -webkit-inline-flex;     /* Syntax for Safari */
    display: -ms-inline-flexbox;      /* Syntax for IE 10 */
    display: inline-flex;     /* Syntax for modern browsers */
```

```
}
```

Notice, in the markup above, that the vendor-prefixed versions of the display property appear in the code before the non-prefixed version (i.e., the version used by modern browsers). Here's how it works:

- If the browser does not understand a flexbox property (modern version or prefixed version), it ignores the property.
- If the browser understands one of the prefixed versions, it renders the prefixed version of the property.
- If the browser understands the modern version (which appears last in the list), it renders the modern version of the property.

The bottom line: When using vendor prefixes, you should always write the prefixed versions before you write the non-prefixed version. That way, the browser will render the most current version of flexbox that it understands.

Autoprefixer

Adding vendor prefixes to stylesheets by hand is not trivial. Not only do you need to know the precise syntax used by each vendor, you also need to type all of the required vendor-prefixed statements correctly into your stylesheet. This takes some effort, and it's easy to make mistakes.

Luckily, there is a better way – Autoprefixer (see https://github.com/postcss/autoprefixer).

Autoprefixer is a post-processor for handling vendor prefixes in CSS. Autoprefixer reads standard CSS styles, like this:

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

Then, autoprefixer adds the required vendor prefixes automatically, producing a new style like this:

```
.container-flex {
    display: -webkit-box;
    display: -webkit-flex;
    display: -ms-flexbox;
    display: flex;
    -webkit-box-orient: vertical;
```

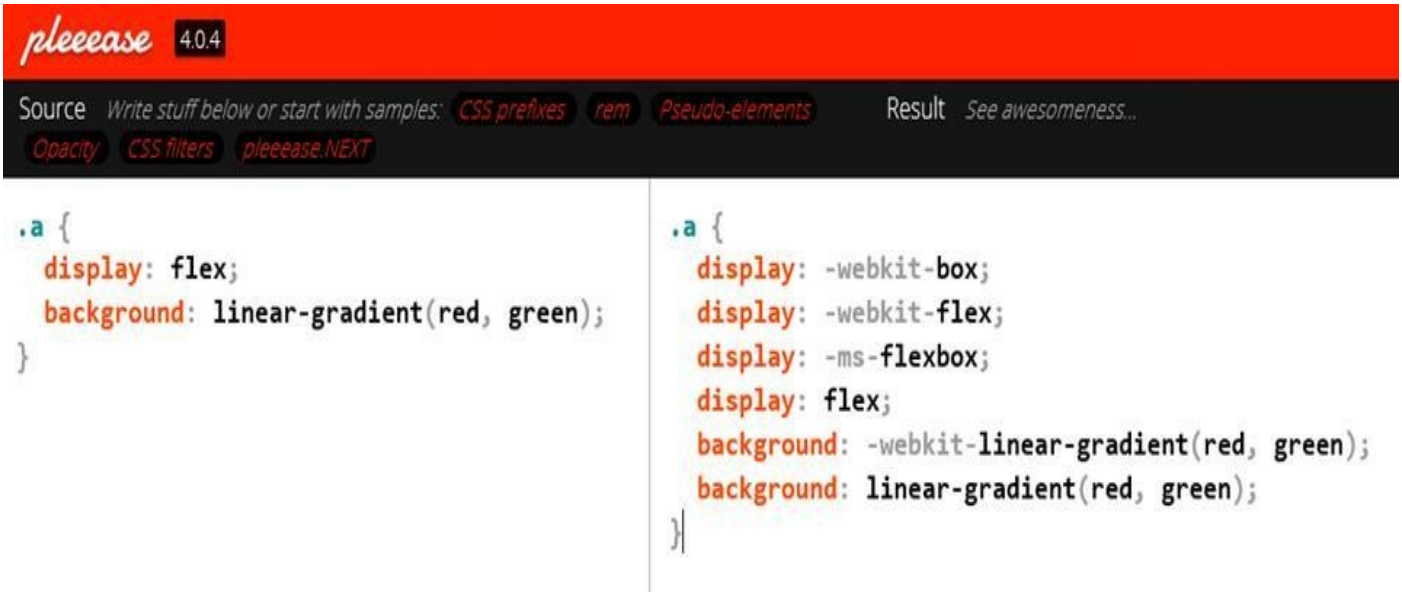
```
    -webkit-box-direction: normal;
    -webkit-flex-direction: column;
    -ms-flex-direction: column;
     flex-direction: column;     }
```

Probably, the easiest way to add vendor prefixes to your stylesheets is via an online Autoprefixer tool. Here are two sites that offer a free Autoprefixer tool:

- **Pleeease** at http://pleeease.io/play/.
- **Autoprefixer CSS online** at https://autoprefixer.github.io/.

Both sites work in the same way. You paste your CSS into a text block on the left side of the screen. Then, the site uses Autoprefixer to output your CSS with prefixed styles to a text block on the right side of the screen. You then copy the CSS markup from the right side of the screen to your stylesheets.

A screen shot of the Pleeease tool appears below:



See how CSS input on the left side of the screen is reproduced on the right, along with the desired vendor-prefixed declarations. Notice also that the vendor-prefixed version of each declaration appears in front of its non-prefixed counterpart – just the way you want it.

If neither of these options (Pleeease or Autoprefixer CSS online) appeals to you, additional Autoprefixer solutions are listed in Appendix B.

Are Vendor Prefixes Worth the Effort?

You may be thinking, “Gosh, adding vendor prefixes sounds like a lot of extra work. Is it really worth the effort?”

The website “Can I Use” tracks browser support for front-end web technologies,

like flexbox (see http://caniuse.com/#search=flex). As this sentence is being written, here is what they are saying about flexbox:

- **Without vendor prefixes.** About 81% of browsers worldwide fully support the flexible box layout model.

- **With vendor prefixes.** About 83% of browsers worldwide fully support the flexible box layout model.

When you add vendor prefixes to your stylesheets, full browser support for flexbox increases by about 2%. That's not much, but it is something. The 2% of site visitors who benefit from vendor prefixes may appreciate the effort.

Of course, Autoprefixer is not limited to flexbox. It also adds vendor prefixes for other new CSS features, and that may be helpful.

Bottom line: Adding vendor prefixes is easy with Autoprefixer, but it only provides a small improvement in vendor support for flexbox. Whether it is worth the effort is a personal choice.

Vendor Prefixes in This Book

I am not going to show vendor prefixes in the examples presented in this book.

Here's why. This book focuses only on the code that you need to type into your stylesheets. If you use the modern flexbox syntax described in this book, you never need to manually type vendor prefixes into your code. Autoprefixer will add vendor prefixes automatically. And, adding vendor prefixes to the examples in the book just clutters things up and makes it tougher to understand what is going on.

Chapter 8. Progressive Enhancement

In the last chapter, you probably noticed that even with vendor prefixes, only 83% of browsers fully support flexbox. And you may have wondered, “What about the 17% of browsers that don’t understand flexbox?”

In this chapter, we answer that question. We will use progressive enhancement to build web pages that work on all browsers, even browsers that don’t support flexbox.

What is Progressive Enhancement?

Building a web page that works on any browser or any device is a challenge. Progressive enhancement is an approach to web design that tries to meet that challenge by doing three things:

- Building a fully-functioning version of each web page, using well-structured HTML and well-supported CSS.
- Testing browser capabilities before adding advanced features, like flexbox.
- Adding advanced features only to browsers that support those features.

With progressive enhancement, web designers build two versions of each web page: (a) a basic version that uses standard HTML and well-supported CSS and (b) an enhanced version that uses advanced features like flexbox. Only site visitors with capable browsers see the enhanced version; everyone else sees the basic version.

Note: When building the basic version of a web page, some designers use only HTML – no CSS. In this book, we’ll use CSS2 in the basic version. Almost all browsers understand CSS2. We’ll use advanced features from CSS3, like flexbox, in the enhanced version of the web page.

Building the Basic Version

The idea of building a separate, working version of a website using just well-structured HTML and simple CSS may seem like a lot of extra work. But it’s really not.

Building the basic version of the web page should be the easiest part of the development process. There is no advanced CSS and no javascript. There is less code to type and debug. You don’t need to worry too much about aesthetics; since most site visitors will see the enhanced version.

For the basic version, you only need to worry about the layout of the page. The most important information should appear early and prominently on the page. The navigation should be easy to find and easy to use. As much as possible, the basic

version of the site should be fully functional, even without advanced CSS and javascript. (In this book, of course, we're not using javascript. But if you did use javascript, it would show up in the enhanced version of the page.)

The basic version of the page will be the foundation for the enhanced version, which you will build later with flexbox and other advanced CSS. If you use clean, well-structured semantic markup to build the basic version, it makes things easier down the road when you build the enhanced version.

Testing Browser Capabilities

To implement a progressive enhancement strategy, we need to test the client browser to determine whether it supports advanced features, like flexbox. One way to conduct this test is to use a tool called Modernizr.

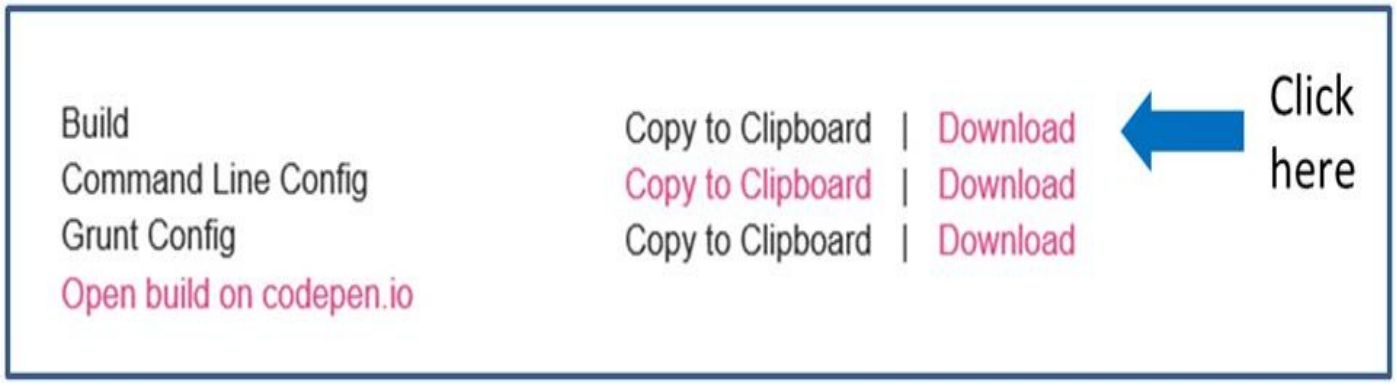
Modernizr is a small javascript file that automatically assesses browser capabilities. You don't have to know javascript to use Modernizr. You just have to download the file and add it to your website.

I'm making the assumption that you've never used Modernizr before, so this section walks you step-by-step through the set-up process.

Step 1. Download Modernizr

Modernizr can be used to assess many different browser capabilities; but in this book, we are only interested in flexbox. To download a lightweight, customized version of Modernizr that just works with flexbox, do the following:

- Go to https://modernizr.com/download#setclasses.
- Click on the radio buttons to select "Flexbox" and "Flexbox Line Wrapping".
- Click the red "Build" button, which appears on the top right side of the page.
- In the dialog box that appears (see below), click the "Download" link on the first line opposite "Build".



- A dialog box will appear asking whether you want to run or save the file. Save

the file to your website.

Congratulations. You have downloaded a working version of Modernizr!

As an alternative, you could also download Modernizr from this book's website at http://flexbox-rwd.com/download.aspx. The file, modernizr-custom.min.js, can be found in the "modernizr" folder in the source code download.

Step 2. Link to Modernizr

Link to the Modernizr file from the <head> section of your web page, as shown in the snippet below:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="modernizr-custom.min.js"></script>
    </head>
```

Now, the Modernizr file will run automatically each time your web page is loaded.

Step 3. Add Classes

Now, when a browser loads the web page, Modernizr will tell you how well the browser supports flexbox. Here's how it works:

- If the "flex-wrap" property is supported on the browser, Modernizr adds a "flexwrap" class to the <html> element. Otherwise, it adds a "no-flexwrap" class.

- If the other flexbox properties are supported on the browser, Modernizr adds a "flexbox" class to the <html> element. Otherwise, it adds a "no-flexbox" class.

To assess browser capabilities, just check the class names in the <html> element at runtime. We'll show how to do this in the next section.

> ***NOTE***
>
> ***If you use "View Source" to look for the classes added by Modernizr, you won't find them. "View Source" shows the original markup before classes are added.***
>
> ***To see the classes, you need to use a tool that shows the markup during runtime. For example, the DOM Explorer in Internet Explorer's Developer Tools will show the classes added by Modernizr.***

Building the Enhanced Version

The basic version of the website is the foundation for the enhanced version. The basic version is built first, using standard HTML. Then, the enhanced version, with advanced features like flexbox, is layered on. Browsers that understand the advanced features see the enhanced version of the website; less capable browsers see the basic version.

To illustrate the process, let’s work through a simple example. Suppose we want to display a small box within a larger box. Here is the HTML:

HTML

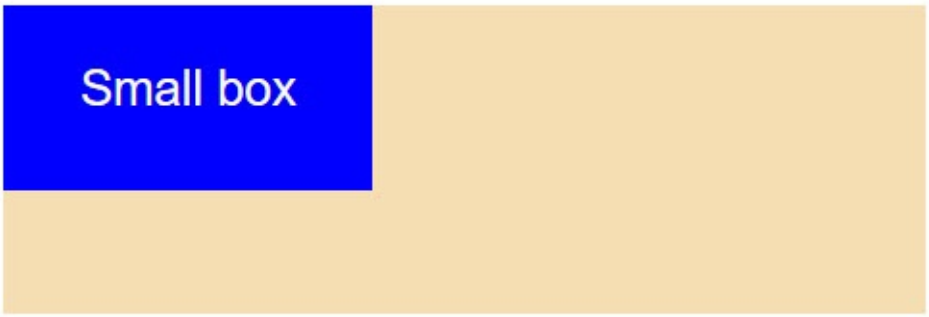
```
<div class=“large-box container-flex”>
    <div class=“small-box”>Small box</div>
</div>
```

And here is the CSS.

CSS

```
.large-box { background-color: wheat; width: 600px; height: 200px; font-size: 2rem;}

.small-box { background-color: blue; color: white; width: 40%; height: 60%; text-align: center; padding-top: 35px;}
```

This would be the basic version of our web page. This version consists of simple HTML and standard CSS. We would expect it to work on any browser or on any device. It produces a small box, positioned in the upper left-hand corner of a larger box.



Suppose that we wanted to create an enhanced version of the web page, in which the small box would be centered vertically and horizontally within the large box, even when screen size changed. We can do this easily with flexbox, but it will work only on browsers that understand flexbox.

To center the small box within the large box, we add this code to the markup for the basic version of the web page:

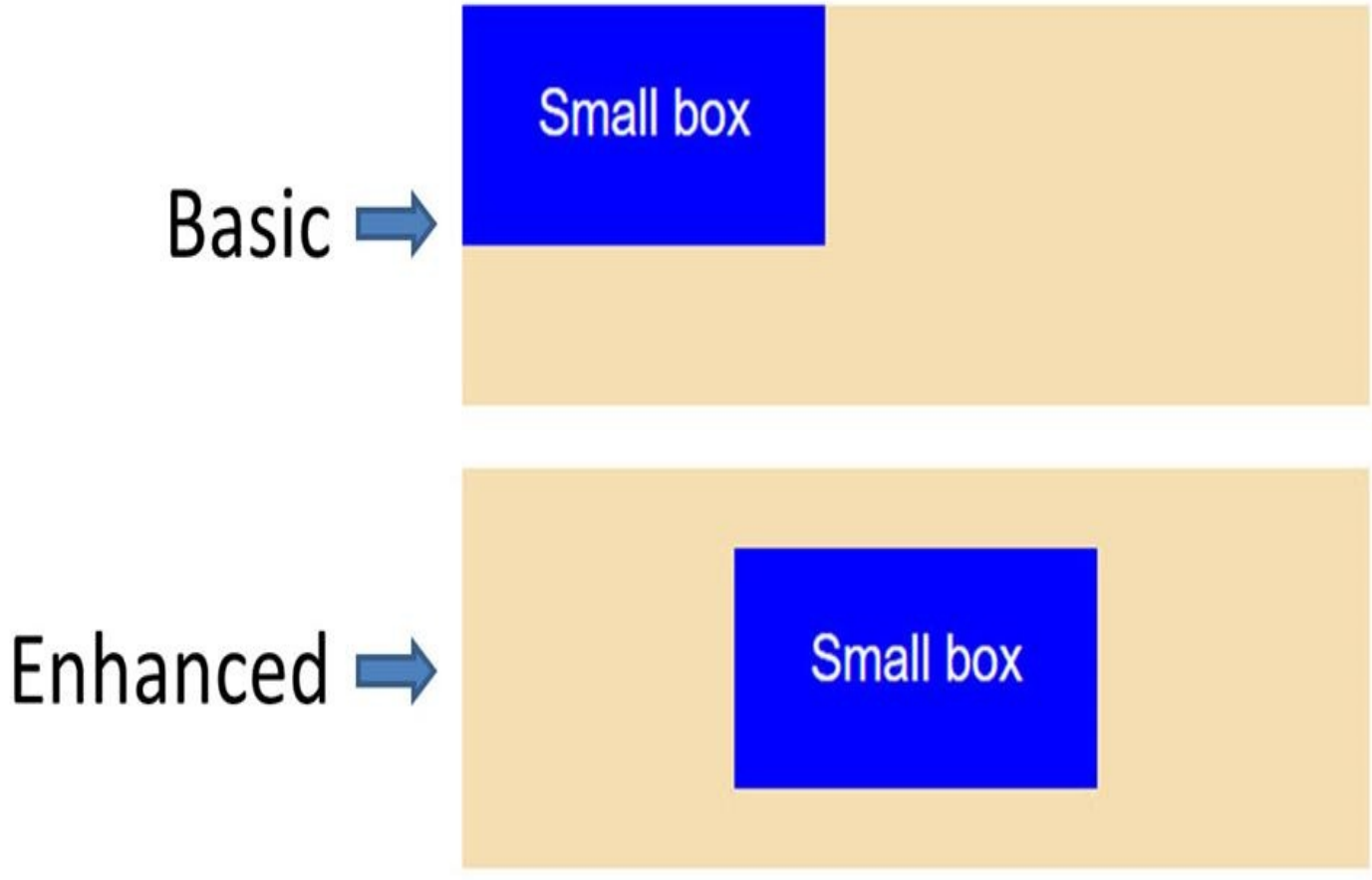
CSS

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

Browsers that understand flexbox will display an enhanced version of the web page that looks like this.



To summarize, this CSS serves up the basic version of the page to browsers that don't understand flexbox; and the enhanced version, to browsers that do understand flexbox. In the basic version, the small box is displayed in the upper left-hand corner of the big box. In the enhanced version, the small box is centered vertically and horizontally. Here is what would be displayed with each version.



The logic for determining which version gets displayed relies on CSS descendant selectors. Here's how it works:

- If the "flex-wrap" property is supported on the browser, Modernizr adds a "flexwrap" class to the <html> element.
- If the other flexbox properties are supported on the browser, Modernizr adds a "flexbox" class to the <html> element.
- Flexbox properties other than "flex-wrap" can be declared in a ".flexbox .container-flex" declaration block.
- All flexbox properties including "flex-wrap" can be declared in a ".flexbox.flexwrap. .container-flex" declaration block.
- The ".flexbox .container-flex" declaration block is processed by browsers that understand flexbox properties other than "flex-wrap", because the "flexbox" class exists on those browsers.
- The ".flexbox.flexwrap.container-flex" declaration block is processed only by browsers that understand all flexbox properties including "flex-wrap", because the "flexbox" class and the "flexwrap" classes exist only on browsers that understand all flexbox properties.
- The ".flexbox .container-flex" declaration block and the ".flexbox.flexwrap.container-flex" declaration block are ignored by browsers that don't understand flexbox, because neither the "flexbox" class nor the "flexwrap" class is added to browsers that don't understand flexbox.

In short, Modernizr identifies browsers that can work with flexbox. And, using descendant selectors, CSS flexbox properties are declared in such a way that they are only processed by browsers that understand flexbox. As a result, browsers that understand flexbox display the enhanced version of the web page, and browsers that don't understand flexbox display the basic version of the web page.

If you're not familiar with descendant selectors, you can find a clear explanation at https://css-tricks.com/almanac/selectors/d/descendant/.

Part 3. Responsive Web Design

Probably, the most consequential application for flexbox is responsive web design.

When you build a responsive website, however, flexbox is not your only useful resource. Other tools, techniques, and considerations are also important. In this section, we cover four additional topics that you should know about if you are going to build a responsive website.

- Chapter 9. Media Queries
- Chapter 10. Responsive Text
- Chapter 11. Responsive Images
- Chapter 12. Setting the Viewport

The notion of responsive web design was introduced by Ethan Marcotte in 2010. You can read his original article at http://alistapart.com/article/responsive-web-design/.

Chapter 9. Media Queries

Responsive web design is all about tailoring the website to the user's device. Often, you can meet this goal with flexible web pages – pages that grow and shrink smoothly in response to changes in the size of a browser window.

Sometimes, though, the smooth, fluid change built into a flexible web page is not enough. There comes a point as the page grows or shrinks that a dramatic change in style or content is needed. For example, on a large screen we might display a full navigation menu; but on a small screen, we might only show a navigation button.

To make dramatic changes in the style or content of a responsive web page, we use media queries.

What is a Media Query?

A media query is a logical expression that evaluates as true or false. The expression consists of one or two elements – a "media type" and/or a "media feature".

There are four media types:

- **screen.** Used for computer screens, tablets, smart phones, TV monitors, etc.
- **print.** Used for printers.
- **speech.** Used for screen reading devices that "read" the web page out loud.
- **all (default).** Used for all media types – screen, print, and speech.

There are many media features. The two that we will use most often for responsive web design focus on the viewport, the display area of a browser window. Specifically, they focus on the width of the viewport:

- **min-width.** The minimum width of the display area.
- **max-width.** The maximum width of the display area.

With media queries, we can change styles or content based on the width of the user's viewport. For example, we might use a large font when the viewport is big; and a small font when it is small.

Media Query Syntax

We can implement media queries in a CSS stylesheet, using the @media rule. The CSS syntax to implement a media query with the @media rule is:

```
@media media-type and (media-feature) {
    CSS-style rules …
}
```

Consider this example. Suppose we wanted a gray background on devices with viewports wider than 480 pixels and a white background on other devices. Here is the CSS we might use:

CSS

```
body  {
    @media screen and (max-width: 480px) {
        background-color: white; }
    @media screen and (min-width: 481px) {
        background-color: gray; }
    }
```

In the above markup, the media queries are shown in bold. Each media query is associated with a block of code that is executed when the media query evaluates as true. In this example, each code block consists of one CSS declaration, which sets the background color of the body element.

Both media queries target screens; so they only work when the website is displayed on a device with a screen, like a computer, tablet, or smart phone. They would not work on devices without screens, like printers. In addition, the first media query targets viewports with a width of at most 480 pixels; and the second, with a width of at least 481 pixels. In this example, viewports less than 481 pixels wide would have a white background; and viewports more than 480 pixels wide would have a gray background.

The Logical “and” Operator

You can use the logical “and” operator to combine two or more media features in a single media query. When you do this, all of the media features must be true in order for the media query to be true.

For example, suppose you wanted to implement a block of CSS code only on mid-size viewports – displays wider than 500 pixels but not as wide as 801 pixels. Here is the CSS you might use:

CSS

```
.midSize {
    @media (min-width: 501px) and (max-width: 800px) {
        CSS style rules …
```

```
}
```

The media query, shown in bold, uses the logical “and” operator to combine two media features – a minimum display width and a maximum display width. If the display width of the user’s device falls within the range defined by the media query, a block of CSS code is executed. If the display width falls outside the range, the block of code is not executed.

Notice that the media query in this example did not define a media type explicitly. When the media type is not defined explicitly, the default value “all” is assumed. This means the query in this example would return “true” for all media types – screen, print, and speech – with displays wider than 500 pixels but not as wide as 801 pixels.

If you wanted to restrict this media query to just devices with screens, you could use the following code:

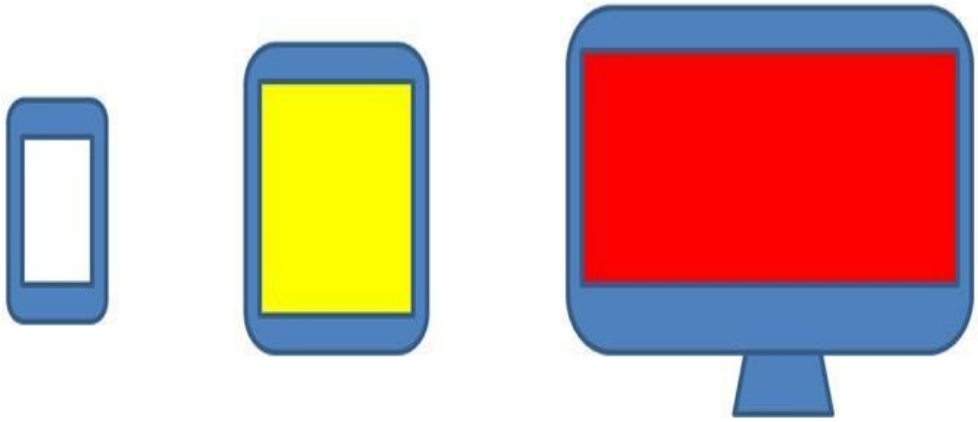
CSS

```
.midSize {
    @media screen and (min-width: 501px) and (max-width: 800px) {
        CSS style rules …
    }
```

Here, the media type “screen” is added explicitly to the media query, which is shown in bold. In this example, the “and” operator is used to chain together the media type and two media features. The block of code that follows the query is only executed when each component of the query (screen, min-width, and max-width) is true.

One Last Example

Let’s walk through one last example that brings together everything that we covered in this chapter. Suppose we wanted our website to use a white background for smart phones, a yellow background for tablets, and a red background for computers, as shown below:



To achieve this result, we could use media queries. We might classify anything smaller than 481 pixels as a smart phone, anything larger than 780 pixels as a computer, and everything else as a tablet. Here is the markup we would use.

CSS

```
body {
    @media screen and (max-width: 480px) {
        background-color: white; }    /* phone background */
    @media screen and (min-width: 481px) and (max-width: 780px) {
        background-color: yellow; }    /* tablet background */}
    @media screen and (min-width: 781px) {
        background-color: red; }   /* computer background */
    }
```

This is a very simple example, but it demonstrates what you can do with media queries. Here, we've tailored the display background to the width of the user's browser window. The same process works equally well for other purposes. You can use media queries to tailor text, images, and layouts to browser width.

NOTE

As this is being written, 97% of browsers worldwide support media queries to assess min/max-resolution. For an up-to-date assessment of browser support for media queries, visit http://caniuse.com/#search=CSS3%20Media%20Queries.

More on Media Queries

In this chapter, to keep things simple, we've focused only on a few of the things you can do with media queries – hopefully the things that are most useful for responsive web design.

We used the CSS @media rule to implement media queries; but there are other ways to implement media queries: as stylesheets in a link element of HTML, in XML, in CSS stylesheets using the @import rule. And we used media queries to find the width of the user's viewport; but you can also use media queries to get other information, like viewport height and browser orientation.

If you're curious about media queries and want to learn more, read the most recent specifications from the Worldwide Web Consortium at https://drafts.csswg.org/mediaqueries-4/#media-query.

Chapter 10. Responsive Text

With so many different devices in use, readability can be a challenge. For example, a font size that is easy to read on the tiny screen of a smart phone may be too small for the large screen of a desktop monitor; and vice versa. The solution, of course, is responsive text – text that changes size to fit the viewport on which it is displayed.

This chapter shows how to use relative sizing units – ems and rems – to make text responsive and readable.

Responsive Text and Readability

Efforts to make text responsive should focus on three factors that affect readability:

- **Font size.** Fonts should be large enough to be easily read, but not so large as to dominate the page. The reader should not have to pinch and zoom to make text readable.

- **Line length.** Rules of thumb for line length range between 45 and 100 characters per line. The line is too long if the reader has a hard time moving from one line to the next; too short if the reader's eye has to travel back and forth too often.

- **Spacing around text.** Lines of text should not be squeezed against one another or against other content on the page. Line height, for example, should be around 150% of font size.

You can control these readability factors through values you assign to CSS properties. Font size is determined by values assigned to the "font-size" property. Line length is determined by values assigned to the "width" of the text container. And spacing is determined by values assigned to "line-height", "padding", and "margin" properties.

Here's the readability challenge: The optimal values for these CSS properties will vary, depending on the size of the user's viewport. You can meet this challenge with media queries and relative sizing units – ems and rems.

What Are EMs and REMs?

Ems and rems are flexible units of measurement that can be used to size text, margins, padding, and just about anything else on a web page. The units are flexible, because they grow or shrink relative to a base defined by the CSS font-size property.

The base tells you how many pixels are represented by 1em or 1rem. For

example, if 1em represents 10 pixels, you know that 2 ems represent 20 pixels; 3 ems represent 30 pixels, and so on. Rems work the same way.

As you can see, ems and rems are similar in the sense that both measure size relative to a base. The main difference between ems and rems has to do with how the base is determined.

- **Ems.** The base is the font size of the element they are used on.
- **Rems.** The base is the font size of the <html> element.

For example, suppose the width of a <p> element and a element are both measured in ems. For the <p>, 1em is equal to the font size of the <p> element; and for the , 1em is equal to the font size of the element. Thus, 1em could have a different value for the <p> element than for the element.

Suppose, on the other hand, the width of a <p> element and a element were both measured in rems. Then, the value of 1rem for both elements would be equal to the font size of the <html> element. In fact, 1rem would have the same value – the font size of the <html> element - for all elements on the web page.

How to Convert EMs to Pixels

In CSS, an em is a unit of measurement equal to the current font size of the element it is used on. Once you know how many pixels are represented by 1em, you can easily convert ems to pixels. Consider the following code:

CSS

```
div {
    font-size: 16px;    /* 1em = 16 px */
    width: 2.5em; ;    /* width = 40 px */
}
```

Here, the element has a font size of 16 pixels; so for that , 1em is equal to 16 pixels. The width of the element is 2.5em. In this example, 2.5em is equal to 2.5 times 16 pixels or 40 pixels.

Here’s a pop test. In the code below, what is the font size of the <p> element in pixels?

HTML

```
<html>
    <div>
        <p>What’s my font size?</p>
    </div>
```

```
</html?
```

CSS

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

div { font-size: 16px;}

p { font-size: 0.5em;}    /* font size is 8px */
```

Initially, the <p> element inherits the font size of the element. The element has a font size of 16 pixels, so the <p> element has an initial font size of 16 pixels. With respect to ems, the base for the <p> element is initially 16 pixels. Then, the font size for the <p> element is set at 0.5em. Thus, the new font size for the <p> element is 0.5 times 16 pixels, which is 8 pixels. And the new em base for the <p> element becomes 8 pixels.

The main difficulty in working with ems is the way that the base size gets redefined through inheritance from a parent element. Because different elements can have different parents, the meaning of an em – the number of pixels it represents – can vary from one element to another. Things can get confusing.

How to Convert REMs to Pixels

A rem is a unit of measurement equal to the current font size of the root <html> element.

Let's look again at the previous example, but this time we'll use rems instead of ems to set font size for the <p> element. In this example, what is the font size in pixels for the <p> element?

HTML

```
<html>
    <div>
        <p>What's my font size?</p>
    </div>
</html?
```

CSS

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

div { font-size: 16px;}

p { font-size: 0.5rem;}    /* font size is 10px */
```

Initially, the font size for the <html> element is set at 20 pixels. As a result, 20 pixels becomes the value of 1rem throughout the stylesheet. That is, 20 pixels is the base for all rem calculations. So the font size of 0.5rem for the <p> element is equal to 0.5 times 20 pixels or 10 pixels.

Notice from this example that the value of a rem is not affected by inheritance. It is constant throughout the web page, always equal to the font size assigned to the root <html> element. Unlike ems, the value of a rem does not vary from one element to another.

How Browsers Affect EMs and REMs

If you don’t specify a font-size for the <html> element, a web browser will use its default setting. For most browsers, the default setting is 16 pixels; but it could be something else, since users can adjust the default to suit their needs (e.g., making text bigger for folks with poor eyesight).

When the font size for the <html> element is not set with CSS, the default setting in the browser determines the value of a rem directly; and it potentially affects the value of an em indirectly through inheritance.

When a designer specifies a font-size for the <html> element with CSS, he/she overrules the default setting in the browser. Overruling the default browser setting has two potential effects – one good and one not so good.

- Overruling default settings gives design control to the designer. For the designer, this is good.
- Overruling default settings removes control from the user, who may have adjusted the default to his or her personal tastes. For the user, this may be bad.

Recommendation: Specify a font size of 100% for the <html> element, as shown below:

CSS

```
html { font-size: 100%; }
```

This accepts the default browser setting for font size, which will usually be 16 pixels. However, when the user adjusts the default setting to something different, the user preference will be honored.

Why Use EMs or REMs?

For years, many of us have used pixels to size elements on web pages. Why switch to ems or rems?

The reason that ems and rems are so useful is that they make it easy to size page elements on the fly – something we do a lot when we build responsive websites. Let’s look at an example to illustrate what I’m talking about.

First, we’ll define font sizes and padding for a small screen. Then, using a media query, we’ll update the styles for a larger screen (800 pixels and up). For the

larger screen, we'll make everything 20 percent bigger. The markup is shown below, using pixels on the left and rems on the right.

With pixels

```
html { font-size: 15px; }
h1 { font-size: 30px; }
h2 { font-size: 25px; }
.container { padding: 10px; }
.item { width: 300px; }
@media screen and (min-width: 800px) {
  html { font-size: 18px; }
  h1 { font-size: 36px; }
  h2 { font-size: 30px}
  .container {padding: 12px; }
  .item { width: 360px; }
  }
```

With rems

```
html { font-size: 15px; }
h1 { font-size: 2rem; }
h2 { font-size: 1.67rem; }
.container { padding: 0.67rem; }
.item { width: 20rem; }
@media screen and (min-width: 800px) {
  html { font-size: 18px; }
  }
```

Notice the media query block (shown in bold), where we defined styles for the large screen.

- **With pixels.** We had to (1) calculate a new pixel value (up by 20%) for each property and (2) set the new value for each property – five calculations and five lines of code in the media block.

- **With rems.** We calculated a new pixel value (up by 20%) for one property and added one line of code – one calculation and one line of code in the media block.

With rems, the <html> font size declaration in the media block set a new base value for 1 rem, which flowed automatically to the other rem-defined properties. For those properties, there was no need to compute new sizing values or to write new CSS declarations in the media query block.

To summarize, both sets of markup produce exactly the same output. But the version that uses rems is much easier to implement. It requires fewer

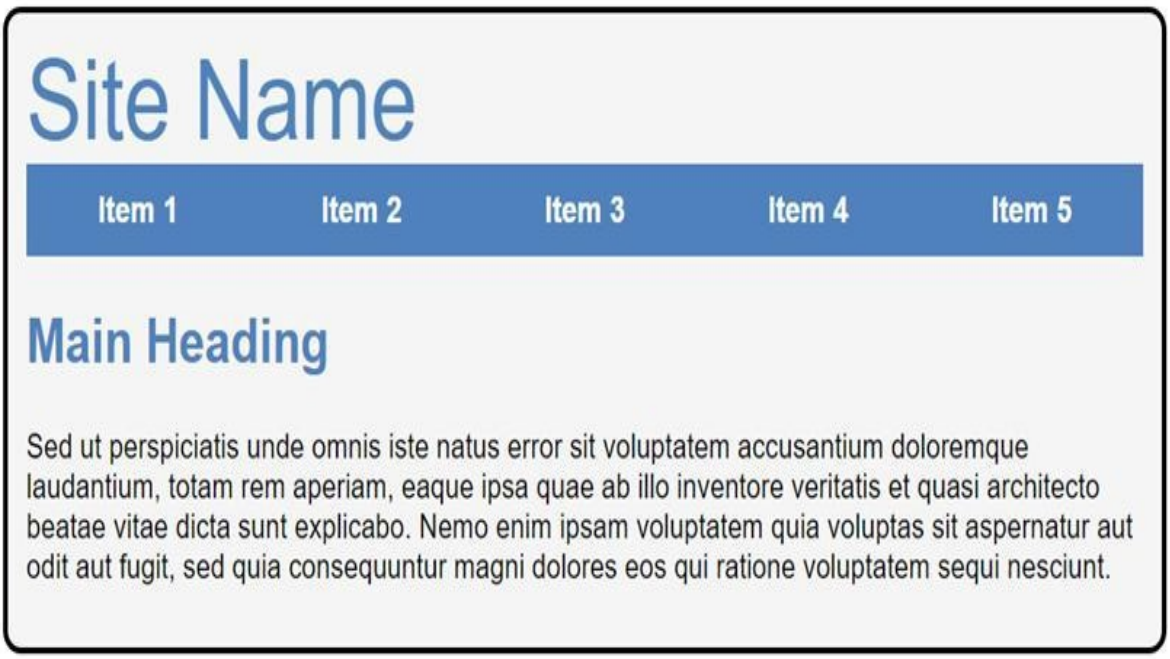
computations and generates fewer lines of code. This makes it easier to maintain. And font sizes and spacing can be easily scaled up or down simply by changing the base – in this case, by changing the <html> font size.

This is why designers like ems and rems.

Responsive Text With REMs

Consider the following example: a responsive, one-column website in which content expands or contracts to fill the space available. In a one-column layout, we would expect page components and font size to shrink or grow in the same direction. That is, as a page component increased in size, we would expect font size within the component to do the same; and vice versa. This type of page behavior is a good application for rem sizing.

For our example, we'll examine a simple one-column web page consisting of four components: a site name, a navigation bar, a main heading, and a block of text.



Above, we see the web page displayed on a viewport about 1000 pixels wide. Everything looks ok. The site name is a good size - bigger than other text, but not so large as to dominate the page. The five menu items are spaced nicely on one line. The body text contains about 80 characters per line, a good line length for readability.

Here is the HTML and CSS used to generate our web page.

HTML

```
<header>Site Name</header>
<nav>
    <span>Item 1</span>
```

```
    <span>Item 2</span>
    <span>Item 3</span>
    <span>Item 4</span>
    <span>Item 5</span>
</nav>

<section>
    <h1>Main Heading</h1>
    <p>Sed ut perspiciatis  … </p>
</section>
```

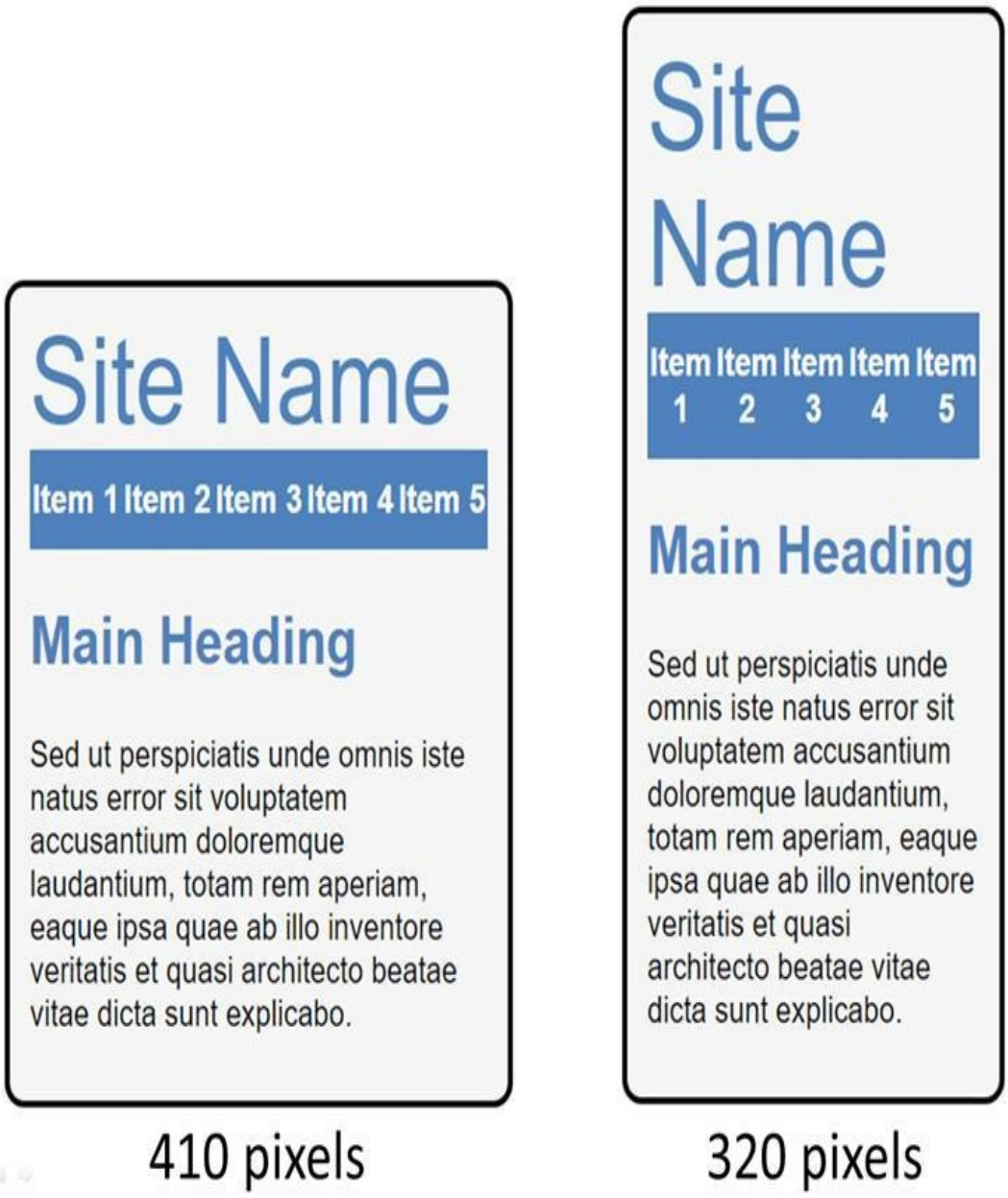
CSS

```
html { font-size: 100%; }
header { font-size:  4.5rem }
nav { font-size: 1.5rem; padding: 1rem; display: flex; justify-content: space-around; }
section { 1.4rem; }
section h1 { font-size: 2.5rem; }
```

Notice the CSS code above. There are two points of interest:

- **REM base size.** By setting the <html> font size equal to 100%, we set the value of 1rem to be equal to the browser's base text size. This would be the browser default (usually 16 pixels) or some other size defined by the user.
- **REM sizing.** Except for the <html> element, all of the sizing for fonts and space around fonts were defined using rem units. As you'll see shortly, this will make life much easier when we write media queries.

Now, let's see what happens when this web page is displayed on smaller viewports.



At 410 pixels, things begin to go awry - menu items are squeezed together, making them hard to read. As the viewport gets smaller, things get worse. At 320 pixels (the size of some Apple iPhones), the site name is too big to fit on one line, menu items no longer fit on one line, and body text holds about 24 characters per line – too few characters for comfortable readability. The page is a mess, but we can fix it easily by adding a media query to our CSS.

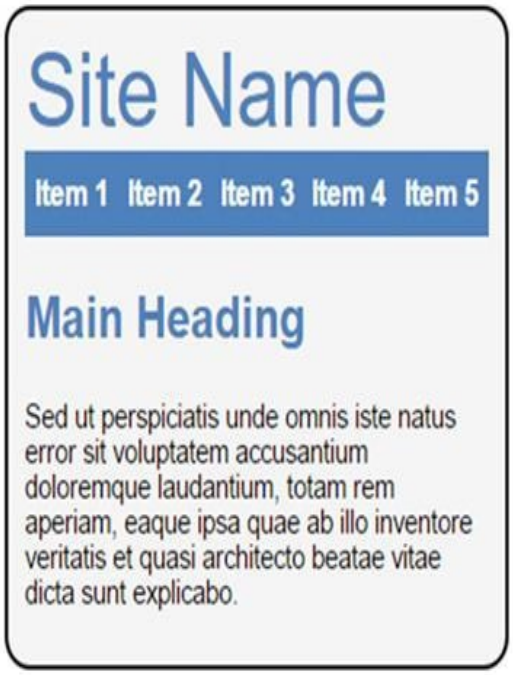
Where should we set a breakpoint for our media query? Since things start to go bad at 410 pixels, we might want to set a breakpoint a little before that – maybe around 480 pixels. Here is the media query, with a breakpoint at 480 pixels.

CSS

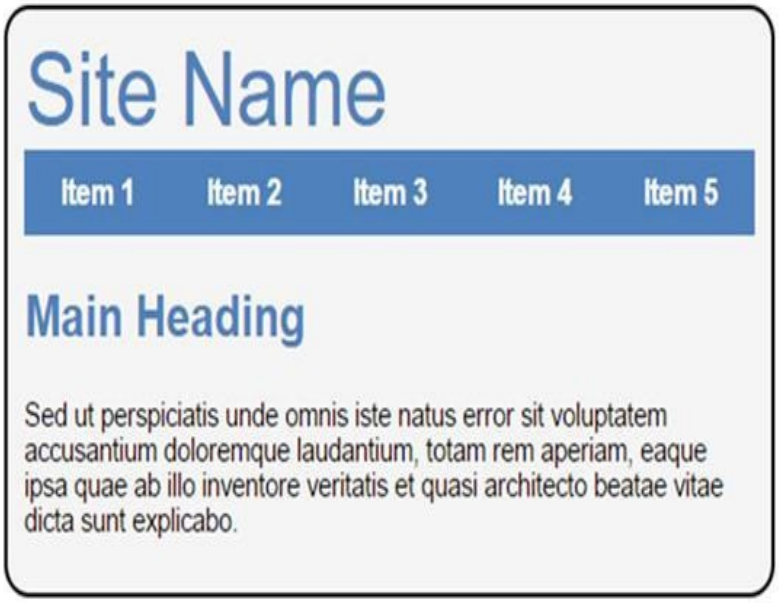
```
@media (max-width: 480px) {
    html { font-size: .65rem; }
}
```

In the code block of the query, we reduced the font size of the <html> element to

.65rem. This reduced the base size for all rem units on the page, so all of the elements sized in rems got smaller. With the media query added to our CSS, here's what the page looked like at 320 pixels and at 480 pixels.



320 pixels



480 pixels

As you can see in the figure above, all of the text fits the space available. The site name fits comfortably on one line, and menu items fit on one line. Line length varies from about 42 characters per line in the 320-pixel viewport to about 72 characters in the 480-pixel viewport – well within the sweet spot for readability.

This simple solution was possible because we had the foresight to size page elements with rem units. When the original design fell apart on small viewports, we were able to restore readability with one <html> font-size declaration in one media query. Very cool!

Responsive Text With EMs

Anything you can do with rems can also be done with ems. To illustrate, let's reverse-engineer the web page we just worked on and recreate it using ems.

To start, let's just switch ems for rems in the CSS markup and see what happens. We're not making any changes to numbers; we're just changing sizing units – ems for rems - in the markup:

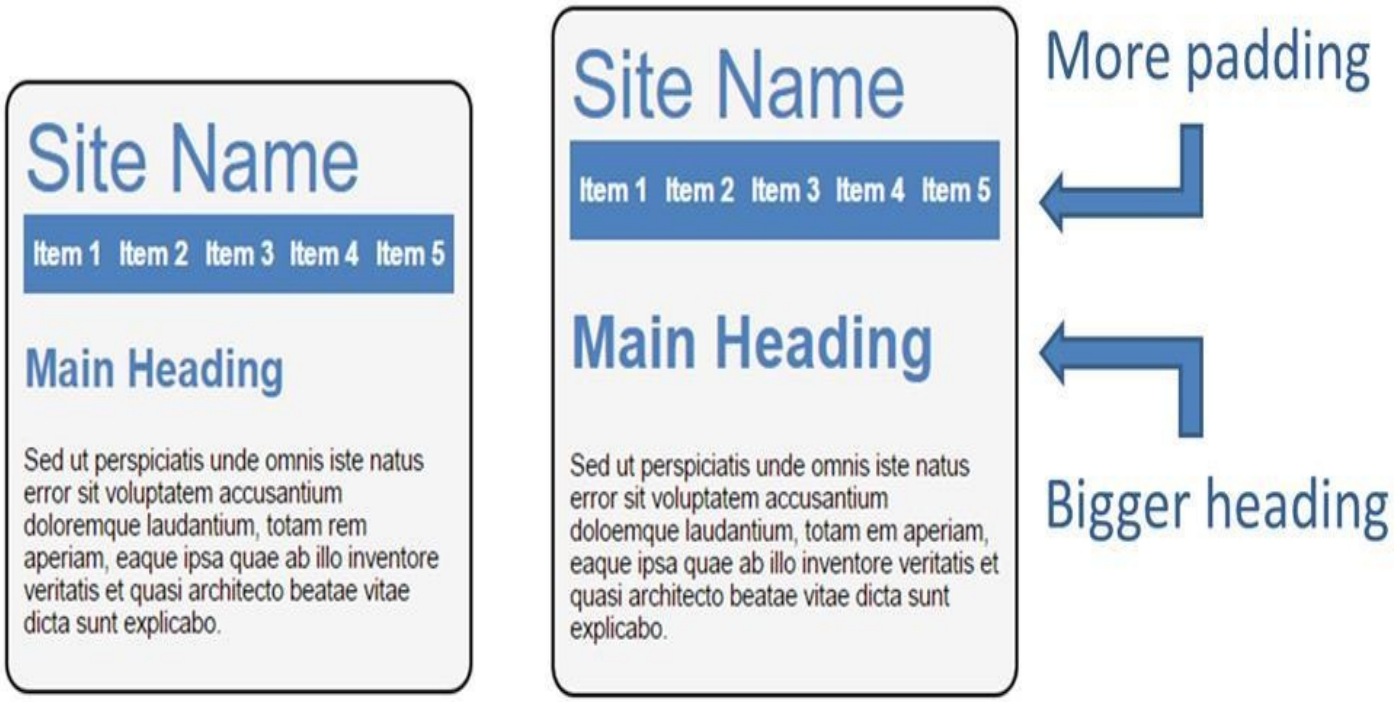
REMs

```
html { font-size: 100%; }
heading { font-size:  4.5rem }
nav { font-size: 1.5rem;
      padding: 1rem; }
section { font-size: 1.4rem; }
section h1 { font-size: 2.5rem; }
```

EMs

```
html { font-size: 100%; }
heading { font-size:  4.5em }
nav { font-size: 1.5em;
      padding: 1em; }
section { font-size: 1.4em; }
section h1 { font-size: 2.5em; }
```

The figure below shows page output on a 320-pixel viewport. On the left, we see the page with rems; and on the right, with ems.

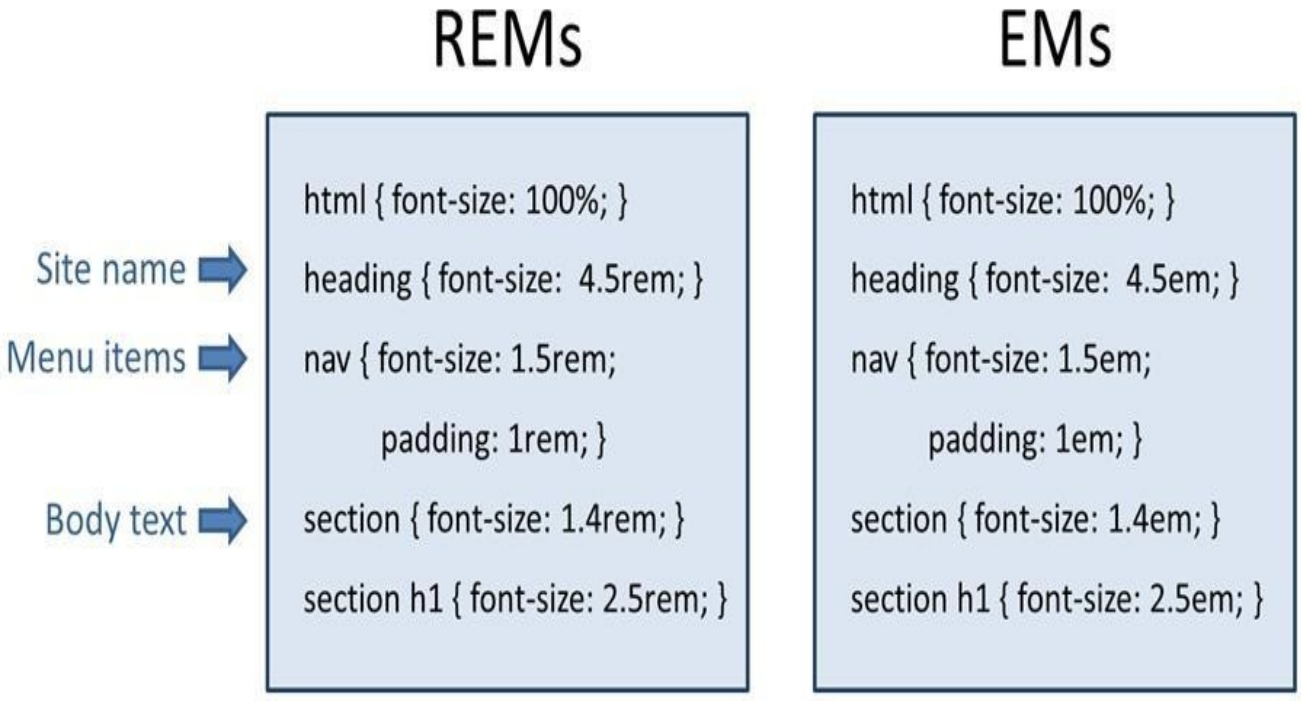


REMs EMs

When we size the text with ems, the site name, menu items, and body text are the same size as they were when we sized the text with rems. But the main heading is larger when it is sized with ems – almost as large as the site name. And the menu bar has more padding. To understand what is going on, we need to answer three questions.

- Why are the site name, menu items, and body text the same size, whether they are sized with rems or ems?
- Why is the main heading bigger when it is sized with ems?
- Why is there more padding in the navigation bar when it is sized with ems?

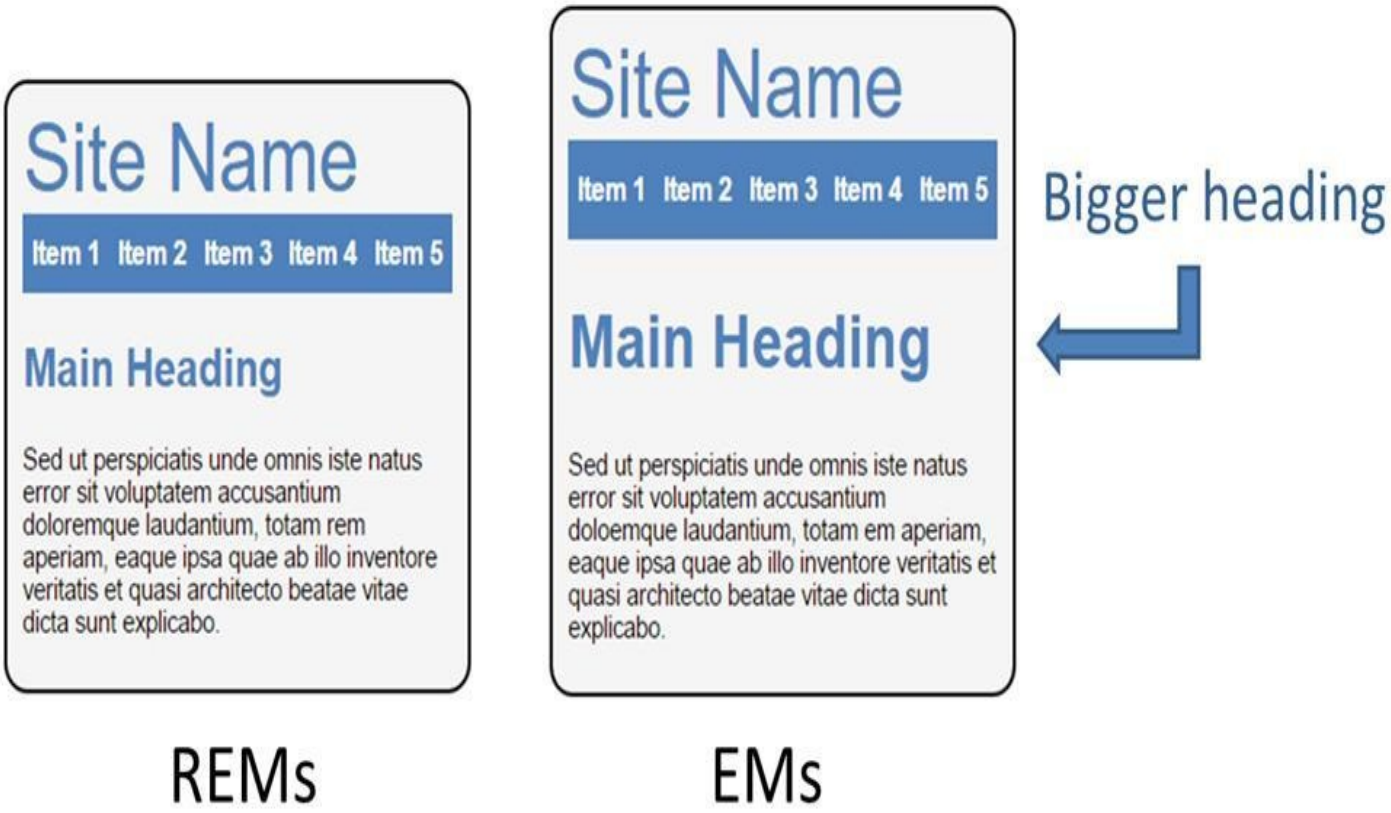
We'll answer these questions one by one, starting with the first question: Why are the site name, menu items, and body text the same size? To answer this question, we need to look at the CSS markup; so here, once again, is the markup.



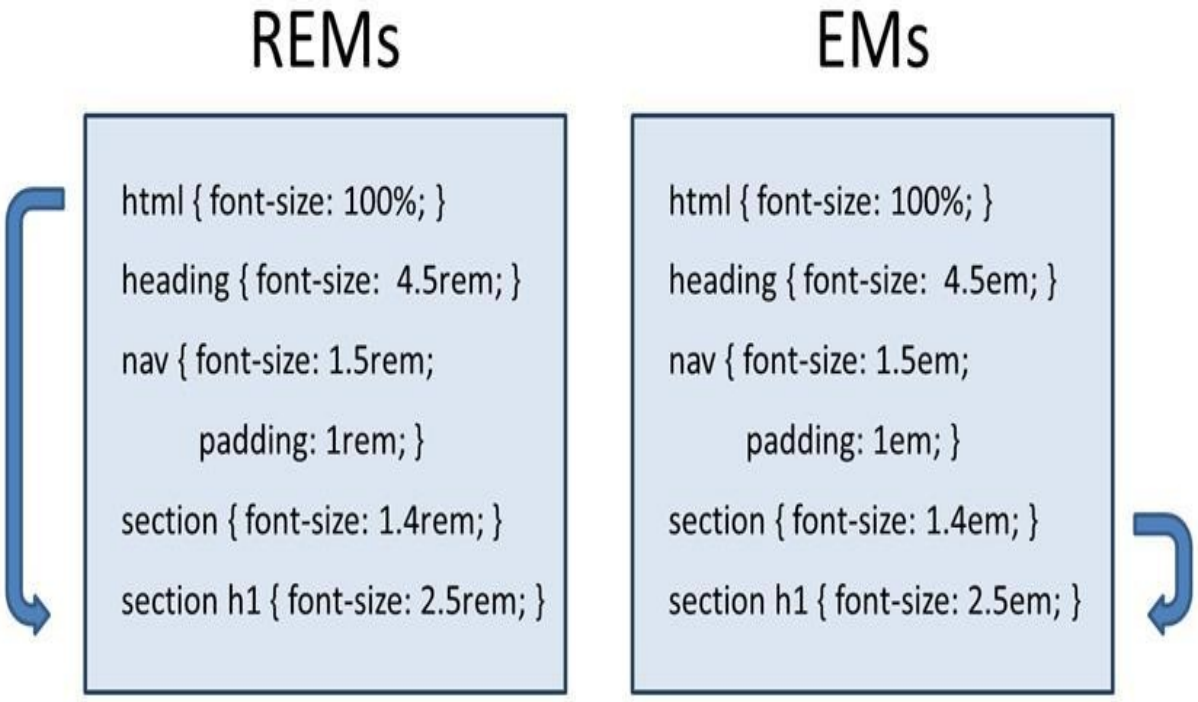
Note the font size declarations for the site name (the <heading> element), menu items (the <nav> element), and body text (the <section> element).

For each of these font size declarations, the size of 1rem is equal to the font size of the <html> element. Through inheritance, the size of 1em is also equal to the font size of the <html> element. For these declarations, 1rem is equal to 1em. Therefore, the site name has the same font size, whether it is set using ems or rems. The same is true for menu items and for body text.

Now, we understand why the site name, menu items, and body text are the same size on both pages. But we still haven't explained why the main heading is bigger when it is sized with ems (see figure below).



To understand why the main heading is bigger when it is sized with ems, it helps to look once again at the CSS markup. This time, though, focus on the <h1> element, since the <h1> element determines the font-size for the main heading.



Compare the value of 1rem to the value of 1em for the <h1> element.

- One rem is equal to the font size of the <html> element.
- One em is equal to the font size of the current selector. Here, the current selector is the <h1> element. The <h1> element is a child of the <section>

element; so through inheritance, the 1em is equal to the font-size of the <section> element. And through inheritance, the font size of the <section> element is equal to 1.4 times the font size of the <html> element.

There was a lot of confusing inheritance there – the main disadvantage of working with ems – but here's the bottom line. For the <h1> element, 1 rem is equal to the font size of the <html> element; and 1em is equal to 1.4 times the font size of the <html> element. This explains why the main heading is bigger when it is sized with ems. For the <h1> element, 1em is equal to 1.4rem.

This suggests a strategy for changing the <h1> font size declaration in ems to reproduce the output generated when everything was sized in rems. We know from the CSS markup that the font-size of the <h1> element is 2.5rem. And we know that 1rem is equal to 1.4em. With this information, we can convert rems to ems.

$$2.5\text{rem} = (2.5\text{rem}) * (1\text{em}/1.4\text{rem}) = 1.79\text{em}$$

Let's make this small adjustment to the CSS markup and see what happens. Here's the new code.

REMs

```
html { font-size: 100%; }
heading { font-size:  4.5rem }
nav { font-size: 1.5rem;
      padding: 1rem; }
section { font-size: 1.4rem; }
section h1 { font-size: 2.5rem; }
```

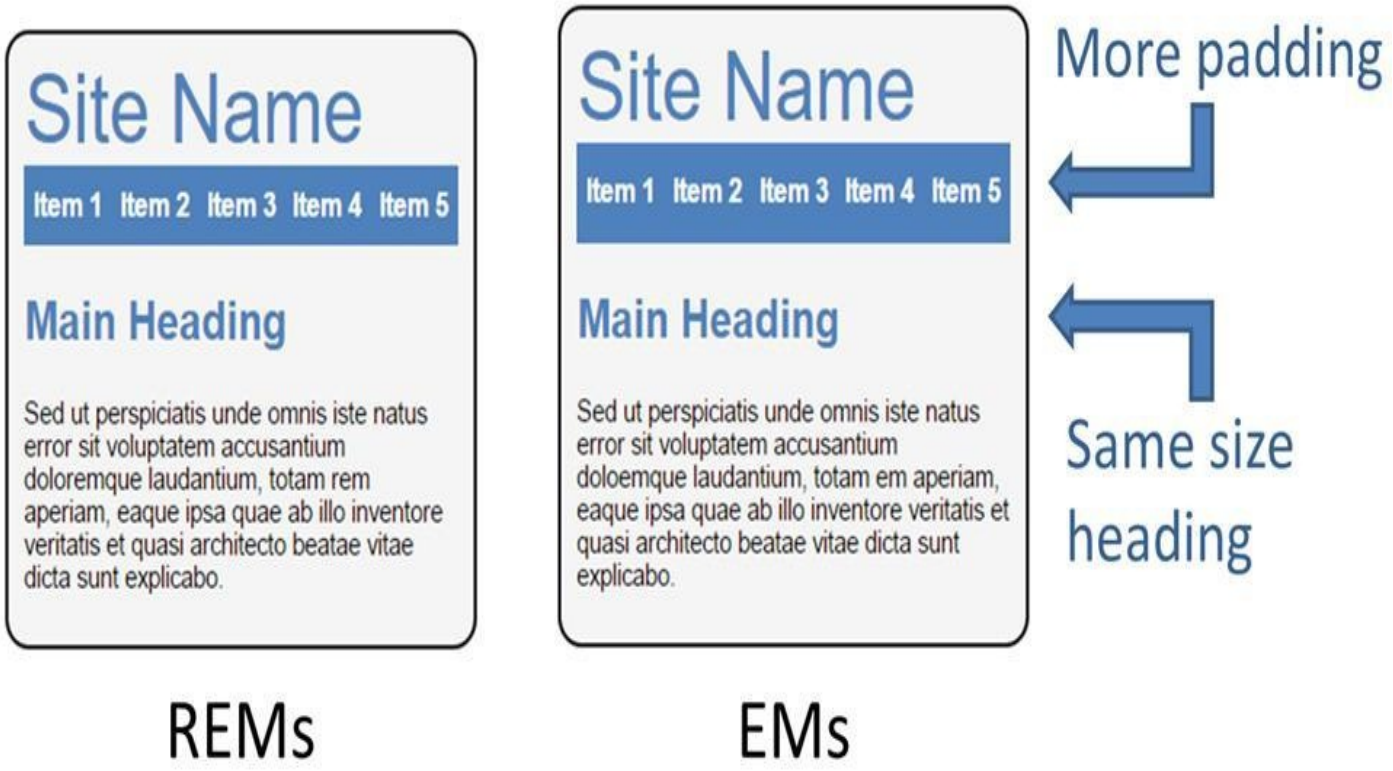
EMs

```
html { font-size: 100%; }
heading { font-size:  4.5em }
nav { font-size: 1.5em;
      padding: 1em; }
section { font-size: 1.4em; }
section h1 { font-size: 1.79em; }
```

New font size



And here's the output it produces.



Hooray! The main heading is the same size on both pages.

Now, the only difference between the two pages is the padding in the navigation bar – more padding in the navigation bar sized with ems. Let's figure out why this difference in padding exists, and let's develop a strategy to eliminate the difference.

Compare the value of 1rem to the value of 1em for the "padding" property.

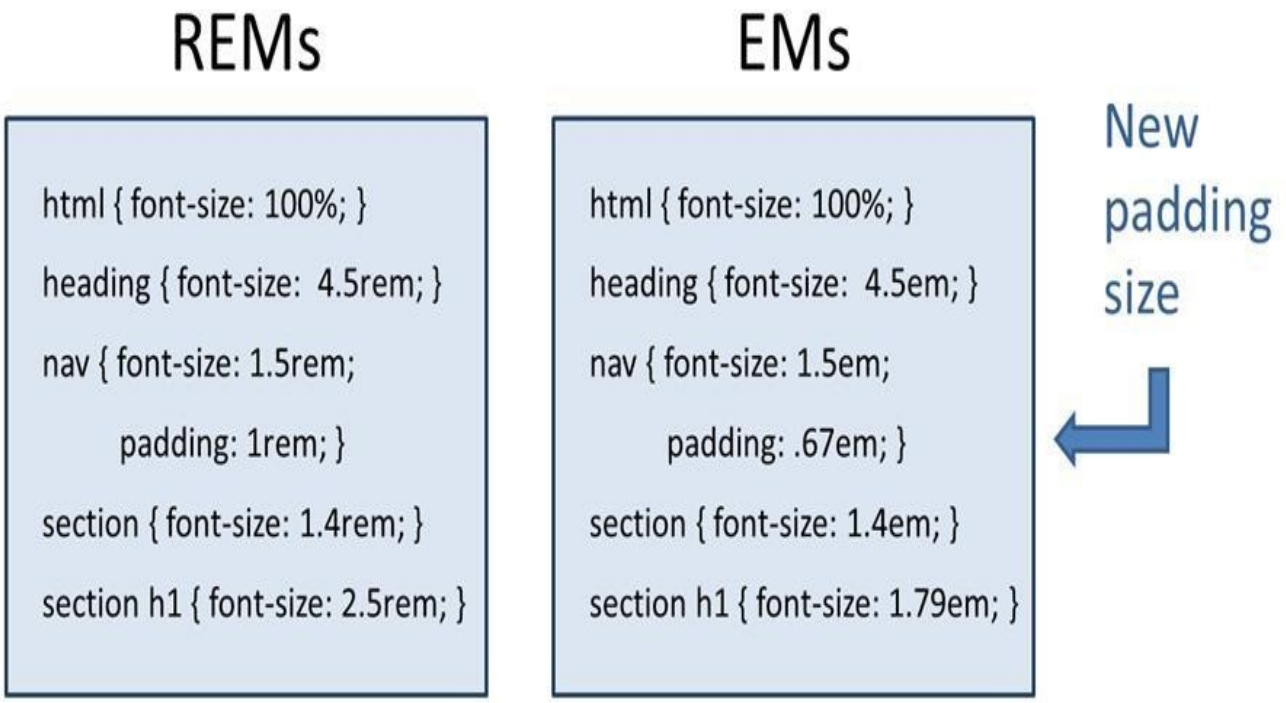
- One rem is equal to the font size of the <html> element.
- One em is equal to the font size of the current selector. Here, the current selector is the <nav> element. The <nav> element has a font size of 1.5em. Through inheritance, the value of 1em for the <nav> element is equal to the font-size of the <html> element. Therefore, the value of 1em for the "padding" property is equal to 1.5 times the font size of the <html> element.

So here's the bottom line for the "padding" property: 1 rem is equal to the font size of the <html> element; and 1em is equal to 1.5 times the font size of the <html> element. This explains why we see more padding in the navigation bar when it is sized with ems. For the "padding" property, 1em is equal to 1.5rem.

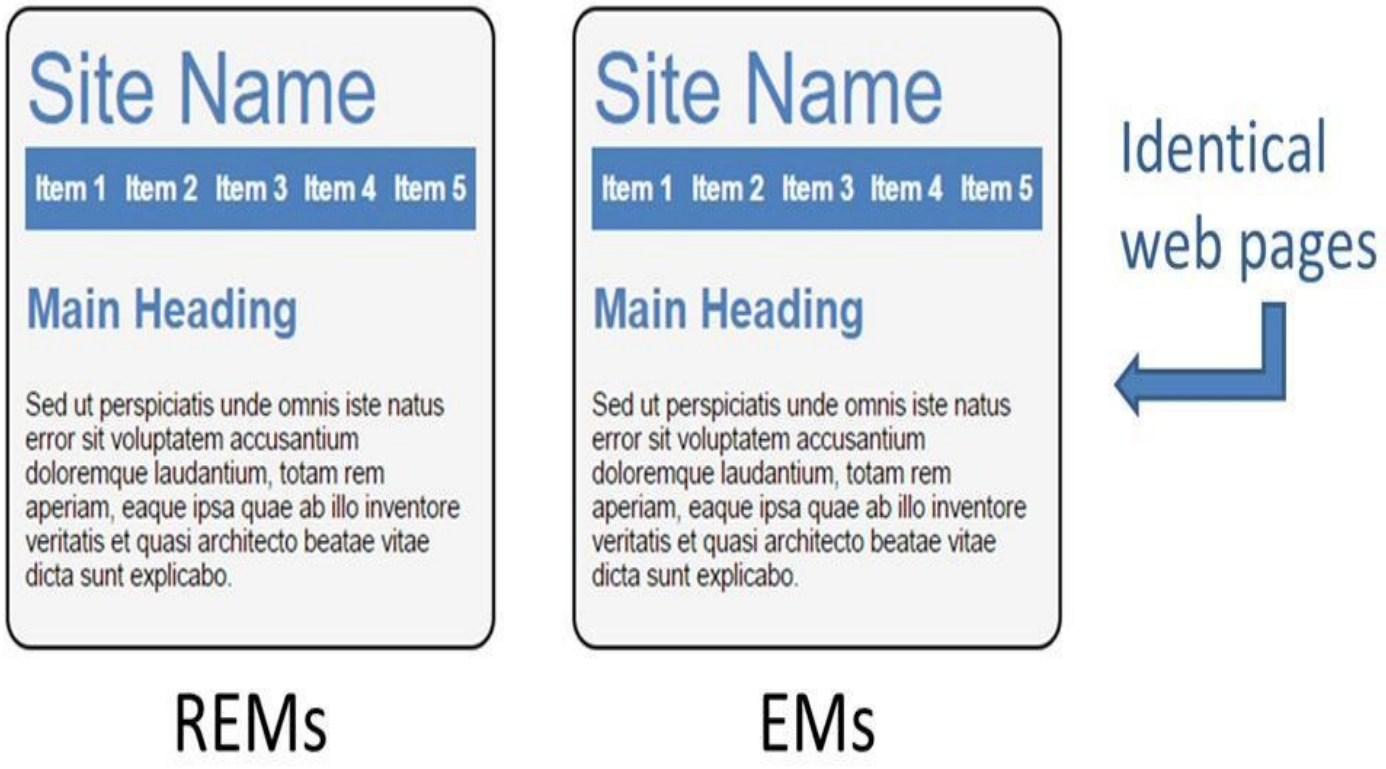
Again, this suggests a strategy for changing the padding declaration in ems to reproduce the output generated when everything was sized in rems. We know from the CSS markup that the padding of the <nav> element is 1rem. And we know that 1em is equal to 1.5rem. With this information, we can convert rems to ems.

$$1\text{rem} = (\ 1\text{rem}\ ) * (\ 1\text{em}/1.5\text{rem}\ ) = 0.67\text{em}$$

Let’s make this small adjustment to the CSS markup and see what happens. Here’s the new code.



And here is the output it produces.



Hooray! The web page built with rems is identical to the page built with ems.

Which Is Better – EMs or REMs?

Now that we know about ems and rems, you might ask: Which is better?

There isn’t a definitive answer to that question. Ems are better in some ways, and

rems are better in other ways. For example, there is general agreement about the following:

- Rems are easier to understand and easier to implement correctly.
- Browser support is slightly better for ems than for rems. Ems are fully supported in all browsers; rems, in 97% of browsers (see http://caniuse.com/#search=rem).

Except for this chapter, I used rems exclusively in this book. That allowed us to avoid messy inheritance issues associated with ems, which was easier on me and on you. However, in a real-world application, I might consider ems because they have the edge in browser support.

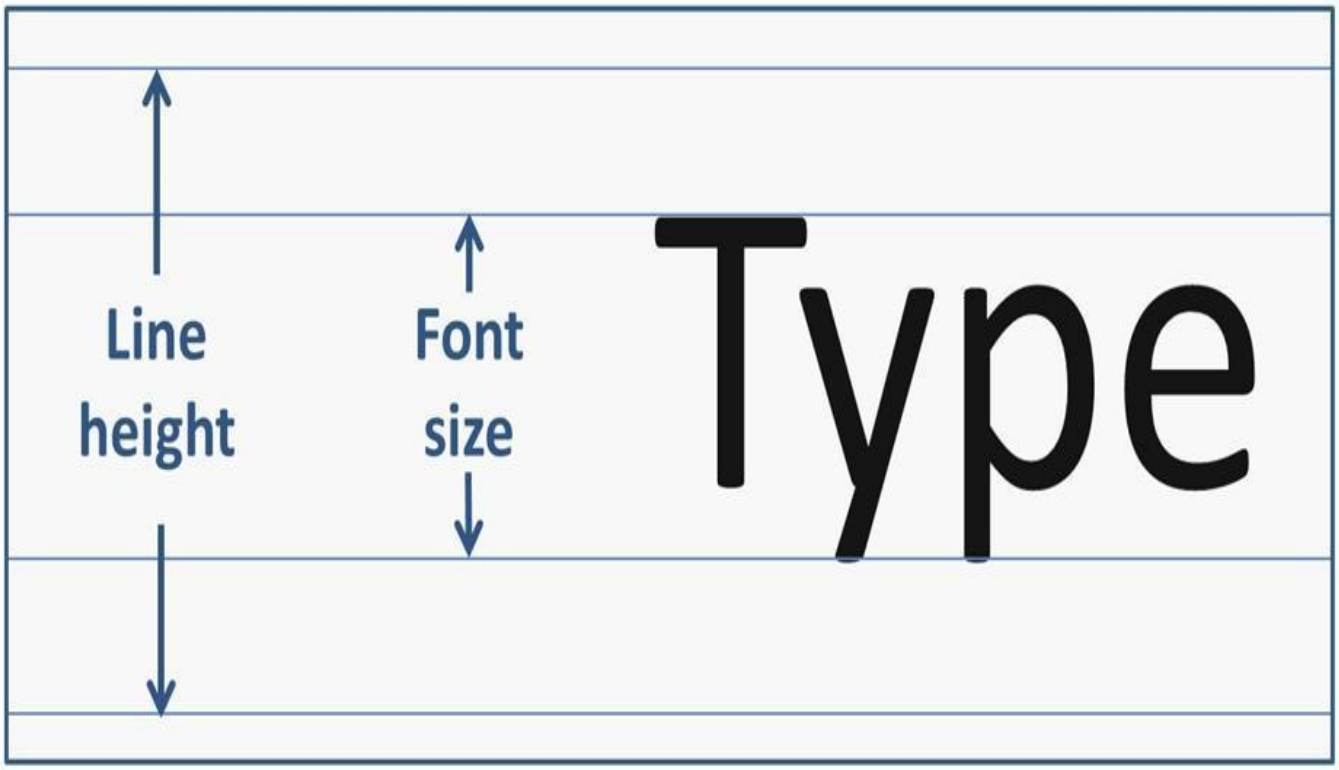
Line Height and Readability

Because it is harder to read lines of text that are squeezed together or spaced too far apart, website designers pay attention to the vertical space between lines of text. In this section, we're going to cover two topics:

- How to control space between lines.
- How much space to leave between lines.

Website designers use the CSS "line-height" property to control spacing between lines of text. So, you might ask, what is line height, and how does the "line-height" property control the amount of space above and below a line of text?

Actually, line height is proportional to font size. Font size is measured from the top of a capital letter to the bottom of a descending letter. In the figure below, font-size stretches from the top of the letter "T" to the bottom of the letter "y". Line height stretches from a point above the capital letter to a point equidistant below the descending letter.



Here is the syntax for line height:

CSS

line-height: “unit-less” number;

In the declaration above, a “unit-less” number is multiplied by the current font size to compute line height. Here’s an example to illustrate what’s going on:

CSS

div { font-size: 10px; line-height: 1.5; }

What is the line height in pixels for the element above? It is 1.5 times the current font size. Here, the font size of the element is 10 pixels; so the line height is 1.5 times 10 pixels or 15 pixels.

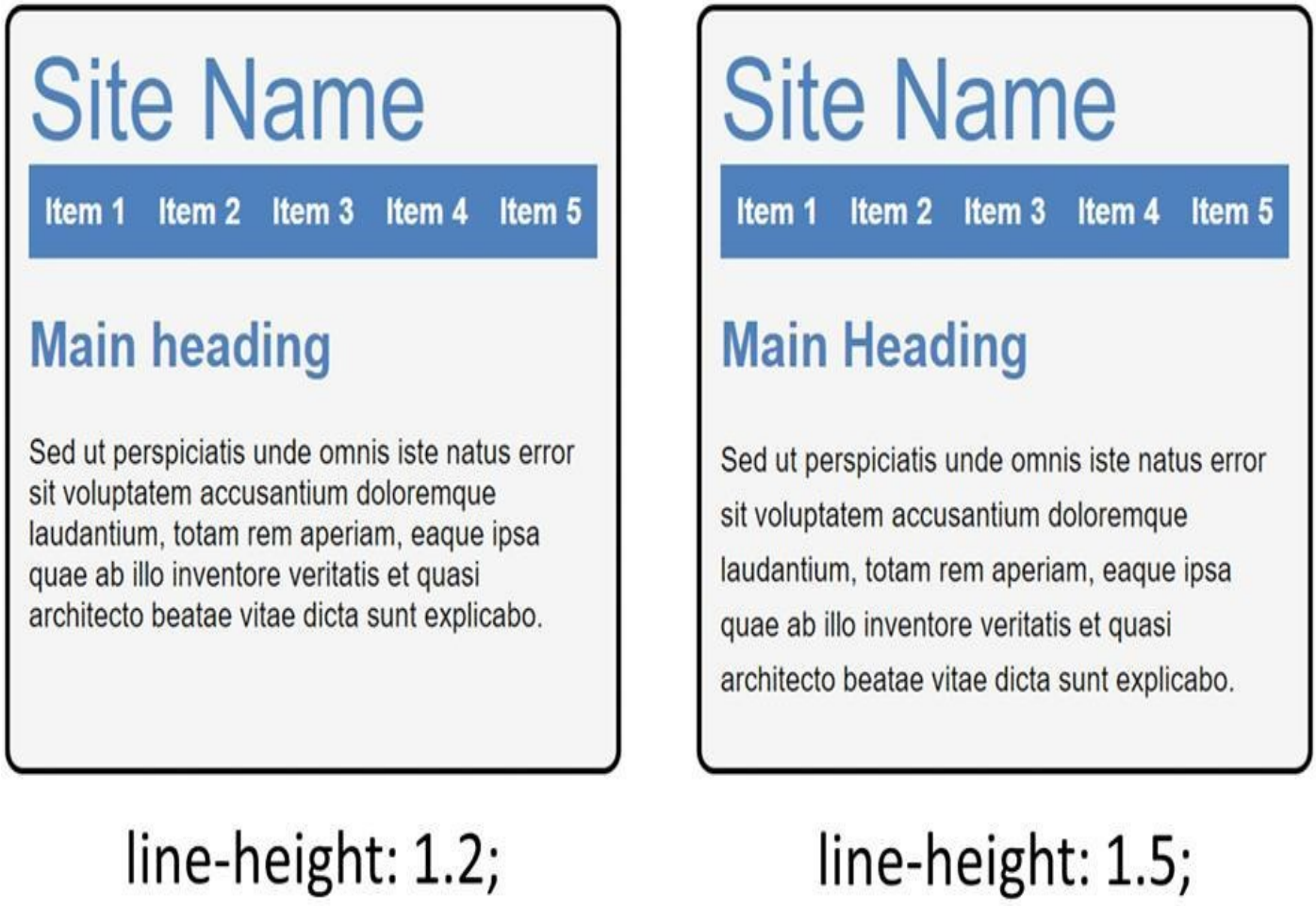
Note: In addition to a “unit-less” number, there are other ways to set line height. It is possible to specify a length with units (like pixels or points) or to use a percentage. Avoid these options. Numbers with units and percentages can have unexpected results. Whenever possible, use a “unit-less” number to set line height.

Now that you know how to set line height, you’re probably asking: What line height should I use?

- A common rule of thumb says to use a line height of 1.5.
- Some folks recommend increasing line height when line length is long (e.g., over 70 characters per line).

I think it makes sense to start out with a line height of 1.5, and adjust as needed.

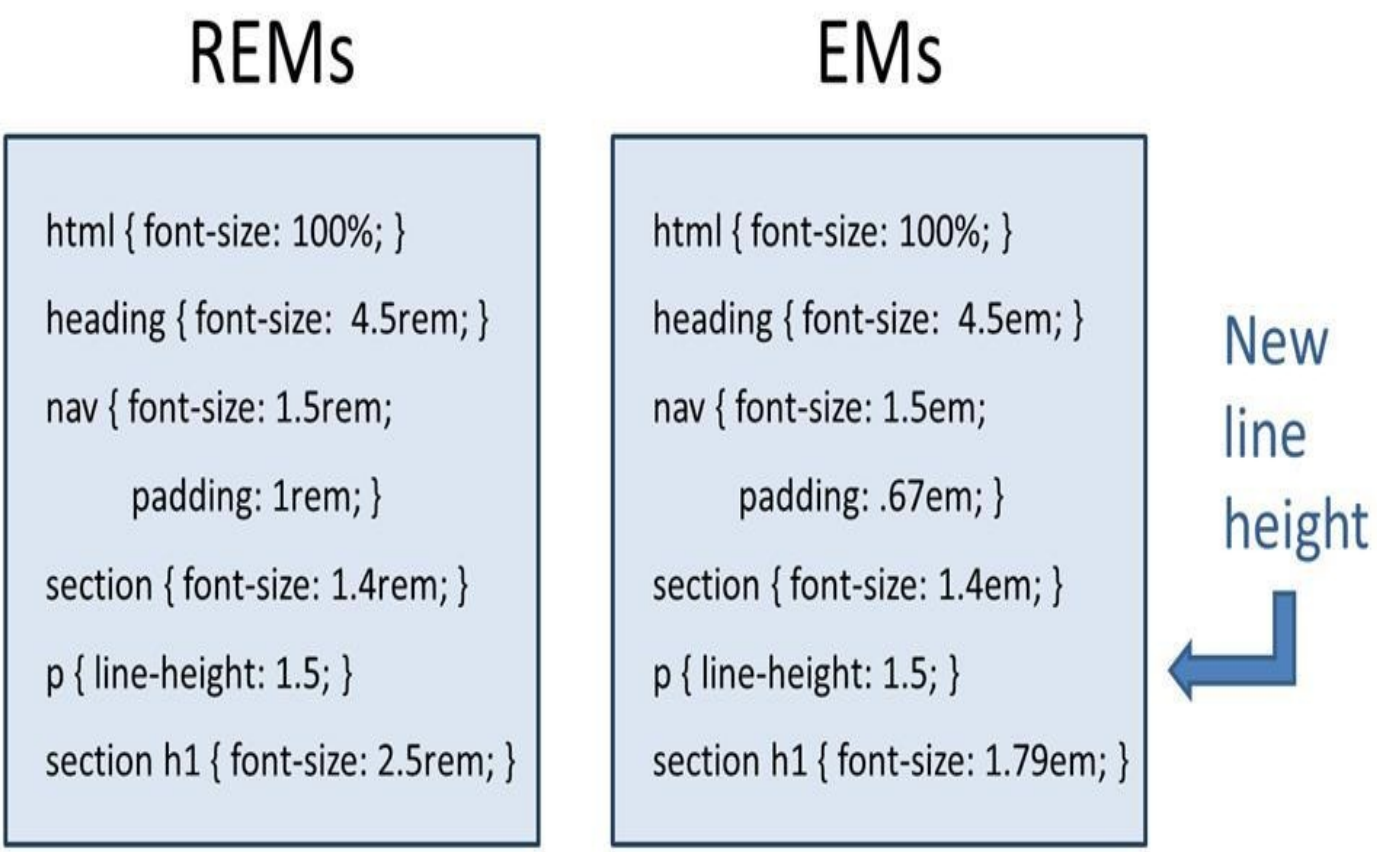
Let’s work through an example and apply what we’ve learned. In the figure below, we see two versions of the same web page that we’ve been working on throughout this chapter.



line-height: 1.2; line-height: 1.5;

On the left, we see the original version of the web page. Focus on the paragraph of text beneath the main heading. Originally, we didn’t set the line height for the paragraph explicitly; so the original version of the web page used the browser’s default setting for line height, which was 1.2. On the right, we see the same web page with a line height setting of 1.5. Notice the difference. The paragraph on the left is squeezed vertically; the paragraph on the right is not.

To be absolutely clear about what we have done, here is the CSS markup with the new line height declaration.



Notice that the same line-height declaration works equally well, whether the web page is sized with rems or with ems.

And finally, suppose you wanted slightly more line height when the line length was longer. For example, you might want a line height of 1.5 when viewport was under 780 pixels; and a line height of 1.6 when the viewport was at least 780 pixels. How would you adjust the CSS code to make this happen?

That's right! All you need to do is add a media query to the CSS markup, as shown below:

CSS

```
@media (min-width: 780px) {
    p { line-height: 1.6; }
}
```

Now, the paragraph of text would have a line height of 1.5 on small viewports; and a line height of 1.6 on large viewports.

Chapter 11. Responsive Images

Responsive images adapt to the device on which they are displayed – large images for big screens and small images for small screens. This chapter covers three topics:

- How to create responsive images.
- Performance issues associated with responsive images.
- How to overcome performance issues.

By the end of the chapter, you will be able to create responsive images that load quickly on your website.

How to Create a Responsive Image

The first requirement for a responsive image is that it fit the space available. There are two ways to make this happen:

- **Flexible images.** Use a single image that grows or shrinks to fit the viewport.
- **Art direction.** Use different images to fit different-sized viewports.

Both of these strategies can be easily implemented in any browser on any device.

Flexible Images

The easiest way to create a responsive image is to make the image flexible. This can be accomplished by setting the “max-width” property of the image element to 100%, as shown below:

CSS

```
img { max-width: 100%; }
```

With the “max-width” property set at 100%, the image will scale down if necessary; but it will never scale up to be larger than its original size. Here’s an example of this CSS markup in action:



On the left, the image of the lemur shrinks to fit the small screen of a mobile device; and on the right, it expands to fit the larger screen of a desktop monitor.

Because we set the maximum width of the image to 100%, it will never expand beyond its natural full width – in this case, 960 pixels. If we wanted the picture to expand beyond its natural width of 960 pixels, we could set the “width” property instead of the “max-width” property, as shown below:

CSS

```
img { width: 100%; }
```

When you use the “width” property, there is a risk that the image will be blurry or pixelated when it expands beyond its original width. Here is what happens when we use the “width” property instead of the “max-width” property, and display the page on a very large monitor:



Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

The image of the lemur is blurry, not sharp. To prevent images from becoming blurry when the page expands, use the “max-width” property rather than the “width” property; and set the “max-width” property of the image to 100%.

Note that we defined image width explicitly in the CSS code, but we did not define image height. As a result, the browser needs to do some extra work to figure out how much space the image will require on the page. This results in a small performance hit. You can help the browser and minimize the performance hit by defining image height explicitly, as shown below:

CSS

```
img { max-width: 100%; height: auto; }
```

Setting image height to “auto” tells the browser to calculate height. The end result is that image height grows or shrinks in proportion to image width. And the browser does less work figuring out how much space to allocate for the image, which improves performance a little bit.

Art Direction

An alternative to creating flexible images is to use different images for different situations – for example, one image for small screens and a different image for large screens. Changing images based on device attributes like screen width is called “art direction”.

Why would you choose art direction over flexible images? Sometimes, an image that looks great on a large screen can lose detail or look bad when it is scaled down for a small screen. Consider the following photo of a lioness at rest:



Displayed on a large screen, the wide shot paints a powerful picture. It shows the lioness blending into her environment, surrounded in the background by trees and distant mountains. But scaled down to fit the small screen of a smart phone (320 pixels), the photo loses impact.



In the small shot, the lioness is barely visible. Rather than simply shrinking the original photo, it might be better to use a new photo entirely. For example, we could crop to original photo to focus more on the lion and less on the background, as shown below:



In this example, the original photo is the better choice for a large screen; and the cropped photo is the better choice for a small screen. This situation is tailor-made for art direction. How do you think we might implement an art direction strategy?

That’s right! We can use media queries to serve up a small photo for small screens and a large photo for large screens. Consider the following markup:

HTML

```
<img class=“small-image” src=“small-photo.jpg” />
<img class=“big-image” src=“big-photo.jpg” />
```

CSS

```
img.small-image { display: none; }
img.big-image { display: inline; }
@media (max-width: 600px) {
    img.big-image { display: none; }
    img.small-image { display: inline; }
}
```

When the viewport is large (601 pixels or bigger), the big image is displayed and the small image is hidden. When the viewport is small (600 pixels or less), the media query reverses the visibility of each image; that is, the big image is hidden and the small image is displayed.

Performance Issues

Other things being equal, pages with many image files load slower than pages with fewer image files. And pages with big image files load slower than pages with small image files. With this in mind, designers try to:

- Use the smallest image files that they can.
- Load only image files that are absolutely necessary.

In theory, this sounds simple; but in practice, with responsive images, it can be challenging.

Flexible Image Performance

First, consider how performance is affected when we work with a flexible image – an image that shrinks and expands to fit the viewport on which it is displayed. Specifically, consider the image below:



Suppose we turned this into a flexible image, using the “max-width: 100%” CSS declaration that we described earlier.

CSS

```
img { max-width: 100%; height: auto; }
```

This would cause the image to be displayed full-size on large monitors and reduced-size on smaller monitors. On the large screen of a desktop monitor, the image would be 1280 pixels wide and 640 pixels high. On the small screen of a smart phone, it could be as small as 320 pixels wide and 160 pixels high.

With a 1-Mbps connection, it would take about 2 seconds to display the full-size version of this flexible image on a large monitor. How long do you think it would take to display a smaller version of this flexible image on a smart phone?

If you said that it would take 2 seconds or longer to display the smaller version of the flexible image, congratulations. You are right! It takes as long or longer to load the small version of the flexible image as it takes to load the full-size version.

Here is what is happening. Before it can shrink the image to fit the smaller screen of a smart phone, the browser must first download the full-size image. It then shrinks the full-size image to fit on the smaller screen. So even when a small image is displayed, a big image is downloaded. This can really degrade performance.

Art Direction Performance

You might think that we could get around this problem by employing an “art direction” strategy.

Instead of making the image flexible, you’d think we could use media queries to display a big image on big screens and a small image on small screens. That way, the browser on a smart phone would never need to download a big image. It could download the small image, which would be much faster. Unfortunately, this strategy does not work in modern browsers.

The problem is that modern browsers pre-load images. That is, modern browsers retrieve images in the background, before they act on CSS code. Consider the code that we used earlier to implement an art direction strategy:

HTML

```
<img class=“small-image” src=“small-photo.jpg” />
<img class=“big-image” src=“big-photo.jpg” />
```

CSS

```
img.small-image { display: none; }
img.big-image { display: inline; }
@media (max-width: 600px) {
    img.big-image { display: none; }
    img.small-image { display: inline; }
}
```

This code displays a big image when the viewport is wider than 600 pixels; and a small image, otherwise. But the browser pre-loads both images, regardless of viewport size!

By the time the browser understands that only one image will be displayed, it is too late. Both images have already been downloaded. Again, we are downloading more pixels than we need, which degrades performance.

How to Overcome Performance Issues

In this chapter, we've described two strategies for creating responsive images: (1) a flexible image strategy and (2) an art direction strategy. Each approach works, but not always as efficiently as users would like.

- **Flexible image strategy.** The browser may download a bigger image than it needs.
- **Art direction strategy.** The browser downloads more images than it needs.

For users on slow connections, this slows everything down. And, for users with bandwidth caps, it may cost more. In this section, we'll describe two ways to get around these performance issues.

The Srcset Attribute

When you want to display different versions of the same image based on viewport size, you can use the "srcset" attribute of the <img> element in combination with the "w" descriptor. Here's a code example:

HTML

```
<img srcset="small.jpg 400w, medium.jpg 700w, big.jpg 1000w" />
```

The "w" descriptor in the "srcset" attribute tells the browser that the width of the small image is 400 pixels, the width of the medium-sized image is 700 pixels, and the width of the big image is 1000 pixels. Given this information, the browser chooses the image that it thinks will work best. The browser might display the small image on the small screen of a smart phone, the medium-sized image on a tablet, and the big image on a large desktop monitor.

Note that the image that actually gets displayed is chosen by the browser, not by the web designer. The "srcset" attribute provides a list of images from which to choose. The "w" descriptor describes the width of each image. The browser uses this image width information plus its knowledge of the user's device (viewport width, connection speed, etc.) to choose an image for display.

The "srcset" attribute is a fairly recent addition to CSS. It is supported by about 71% of current browsers worldwide (see http://caniuse.com/#search=srcset). To handle browsers that do not understand "srcset" (like Internet Explorer 11), you should include a "src" attribute in the <img> element statement. A code snippet with the "src" attribute in bold is shown below:

HTML

```
<img src="small.jpg"
    srcset="small.jpg 400w, medium.jpg 700w, big.jpg 1000w" />
```

Browsers that don’t understand “srcset” display the image from the “src” attribute. Browsers that understand “srcset” display one of the images from the “srcset” list.

Let’s walk through an example to show how to use the “’srcset” attribute. Suppose you want to display one image on a web page. You want the image to be flexible – to shrink or expand to fill the viewport. You have three photos to choose from – one for a small screen, one for a medium-sized screen, and one for a large screen. Here’s the code you might use:

HTML

```
<img src=“cat.jpg”
    srcset=“cat.jpg 400w, dog.jpg 600w, elephant.jpg 800w” />
```

CSS

```
img { width: 100%; }
```

The CSS style rule sets width equal to 100%, which makes the images flexible. Browsers that don’t understand “srcset” will display the image identified by the “src” attribute - the small image, cat.jpg. Browsers that do understand “srcset” will choose from the three images listed - a small image of a cat, a medium-sized image of a dog, and a large image of an elephant. The “w” descriptor describes the width of each image, and browsers that understand “srcset” will use that width information to make their choice.

Here’s how things worked with a Firefox browser:



400 px



401 px - 600 px



601 px

When the initial size of the viewport was 400 pixels or less, Firefox displayed the cat image. When the initial size of the viewport was between 401 and 600 pixels, Firefox displayed the dog image. And when the initial size of the viewport was 601 pixels or more, Firefox displayed the elephant image. Chrome, MS Edge, and

Opera did the same thing. Internet Explorer does not understand “srcset”, so it displayed the image identified by the “src” attribute (in this example, the small, cat image), regardless of viewport size.

After the image was initially displayed, the browsers reacted differently when the user zoomed in or zoomed out.

- **Firefox.** Displayed the small image when the viewport shrank to 400 pixels or less, the mid-sized image between 401 and 600 pixels, and the big image above 600 pixels.

- **Chrome, Edge, and Opera.** Displayed larger images as the user zoomed out, but did not display smaller images as the user zoomed in.

- **Internet Explorer.** Displayed the image identified by the “src” attribute (i.e., the cat image), regardless of zoom behavior.

These are the results that I saw on my computer using the particular browsers that I tested. You might see different results on a different device, a different browser, or a different version of the same browsers that I used. These results show how the browser – not the designer – chooses the image that actually gets displayed.

If you can tolerate not knowing exactly which image the user will see, the “srcset” attribute offers a reasonable solution to the performance problems we described earlier. With “srcset”, the browser downloads just one image so the issue of downloading more images than needed is minimized. And, at least in some cases, the browser can choose a smaller image and avoid downloading a bigger image than needed.

The Picture Element

If you can’t tolerate not knowing exactly which image the browser will display, you might want to consider using the <picture> element. The <picture> element is a fairly recent addition to HTML, currently supported in about 70 percent of browsers worldwide (see http://caniuse.com/#search=picture).

Think of the <picture> element as a container for images. It holds one <img> element and one or more <source> elements:

- **<img>.** The <img> element provides backward compatibility for browsers that don’t understand the <picture> element.

- **<source>.** Each <source> element includes a required “srcset” attribute and one or more optional attributes, such as the “media” attribute.

To illustrate how things work, let’s select once again from images of a cat, dog, and elephant. This time, though, we’ll use the <picture> element to choose the

image.

HTML

```
<picture>
    <source media="(min-width: 400px) and (max-width: 599px)" srcset="/images/dog.jpg" />
    <source media="(min-width: 600px)" srcset="/images/elephant.jpg" />
    <img src="/images/cat.jpg" />
</picture>
```

In the code snippet above, we have two source elements and one image element. Each source element includes a “srcset” attribute and a “media” attribute. Notice that the value of the media attribute is a media query. The browser reads from top to bottom and displays the first image with a media query that evaluates as true. If none of the media queries evaluate as true, the image from the <img> element is used.

Here are the results produced by that code. Internet Explorer 11 does not understand the <picture> element and “srcset”, so it displayed the default image (the cat) at all viewport sizes. All of the other browsers (Firefox, Chrome, Opera, MS Edge) followed the <source> element instructions perfectly. They displayed the elephant when the viewport was at least 600 pixels wide; the dog, between 400 and 599 pixels; and the cat when the viewport was less than 400 pixels.

Summary

We’ve covered a lot of ground in this chapter. We’ve described two strategies for creating responsive images. We’ve described four methods for implementing one or both of those strategies. And we’ve noted advantages and disadvantages of each strategy/method combination.

At this point, you may be wondering: Which method should I use to create a responsive image? Use the table below to sort through your options.

Method	Strategy	Pros and Cons
Set width	Flexible image	• Works in all browsers • Performance issues
Media queries	Art direction	• Works in all browsers • Performance issues
Srcset	Flexible image	• Browser support issues • Improved performance • Browser chooses image
Picture	Flexible image Art direction	• Browser support issues • Improved performance • Designer chooses image

The best method for creating a responsive image will depend on at least four factors.

- Extent of browser support.
- Speed with which images are processed.
- Responsive image strategy (flexible image and/or art direction).
- Whether images are chosen by the browser or by the designer.

The first two methods that we described – using the width property to create flexible images and using media queries for art direction – work in all browsers. And they allow the designer to choose the image that gets displayed. But they load more pixels than the browser actually needs, which hurts performance.

The “srcset” attribute is a good choice for displaying flexible images; and it only loads one image, which improves performance. It is not supported by older browsers, but it should degrade gracefully to display a default image. It may not be a good choice to implement an “art direction” strategy, since the browser – not the designer – chooses the image that gets displayed.

The <picture> element can be used to implement a flexible image strategy or an art direction strategy. Like the “srcset” attribute, it only loads one image, which improves performance. The biggest drawback is browser support; <picture> is only supported in about 70% of browsers worldwide. Although a default image is displayed in browsers that don’t support <picture>, it may not be the right image for an art direction strategy.

To summarize, the choice of a method to create a responsive image depends on many factors, and it involves tradeoffs. Hopefully, you now have the information you need to make the right choice for your situation.

Chapter 12. Setting the Viewport

The viewport is the visible area of the browser on the user's device. Think of it as the space available to display a web page.

The viewport is not the same as screen size. The viewport of a device can change, but screen size is fixed. Screen size tends to be larger on computers, smaller on smart phones, and somewhere in between on tablets.

Viewport often follows the same pattern – large on computers, smaller on tablets, and smallest on smart phones. However, the viewport can also be very small on a computer, because users can typically resize their browser window on a computer.

The point here is that the viewport refers to the visible size of the browser, independent of the size of the device screen. When you build a responsive website, pages reflow and resize to fit the viewport – not the screen size.

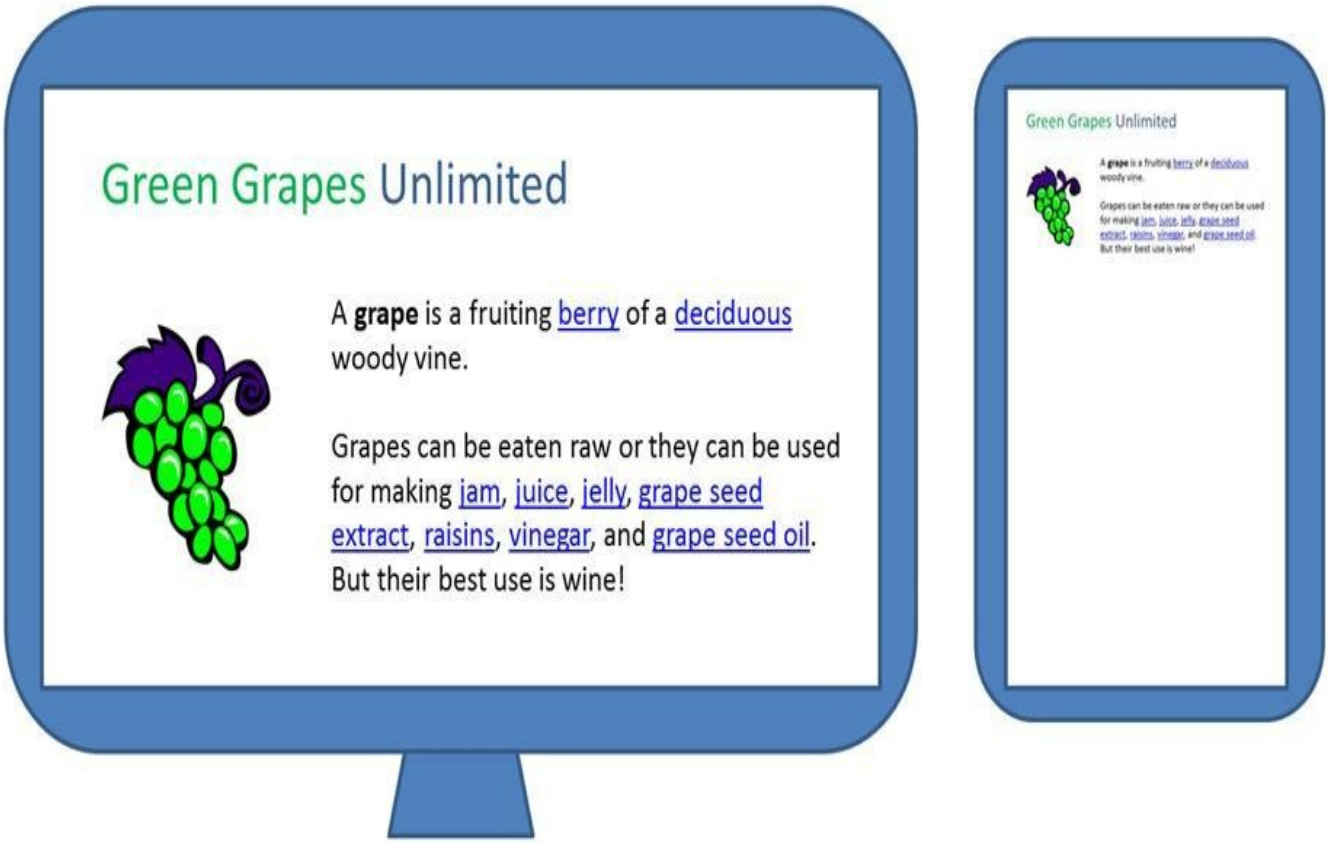
The Viewport Challenge

Until recently, websites were designed mainly for display on computer monitors – screens with potentially large viewports. Viewports as wide as 980 pixels were common. Then, as tablets and smart phones become popular, websites began to appear on smaller screens with smaller viewports. Some mobile phones had screens as small as 320 pixels.

To render the websites on smaller screens with smaller viewports, browsers took two actions:

- They shrunk the web pages to fit on the smaller screen.
- They changed font sizes to try to make text easier to read.

The result was often a big page shrunk too small to be easily viewed. Site visitors needed to zoom in to read page content. Not user-friendly.



In the figure above, the Green Grapes Unlimited website is readable on the desktop monitor. But when it is shrunk to fit the screen of a mobile device, everything is too small; and the site can't be read without zooming in.

For a site that is not responsive, this is probably the best we can do. Let the browser shrink the site, and make the user zoom and pan.

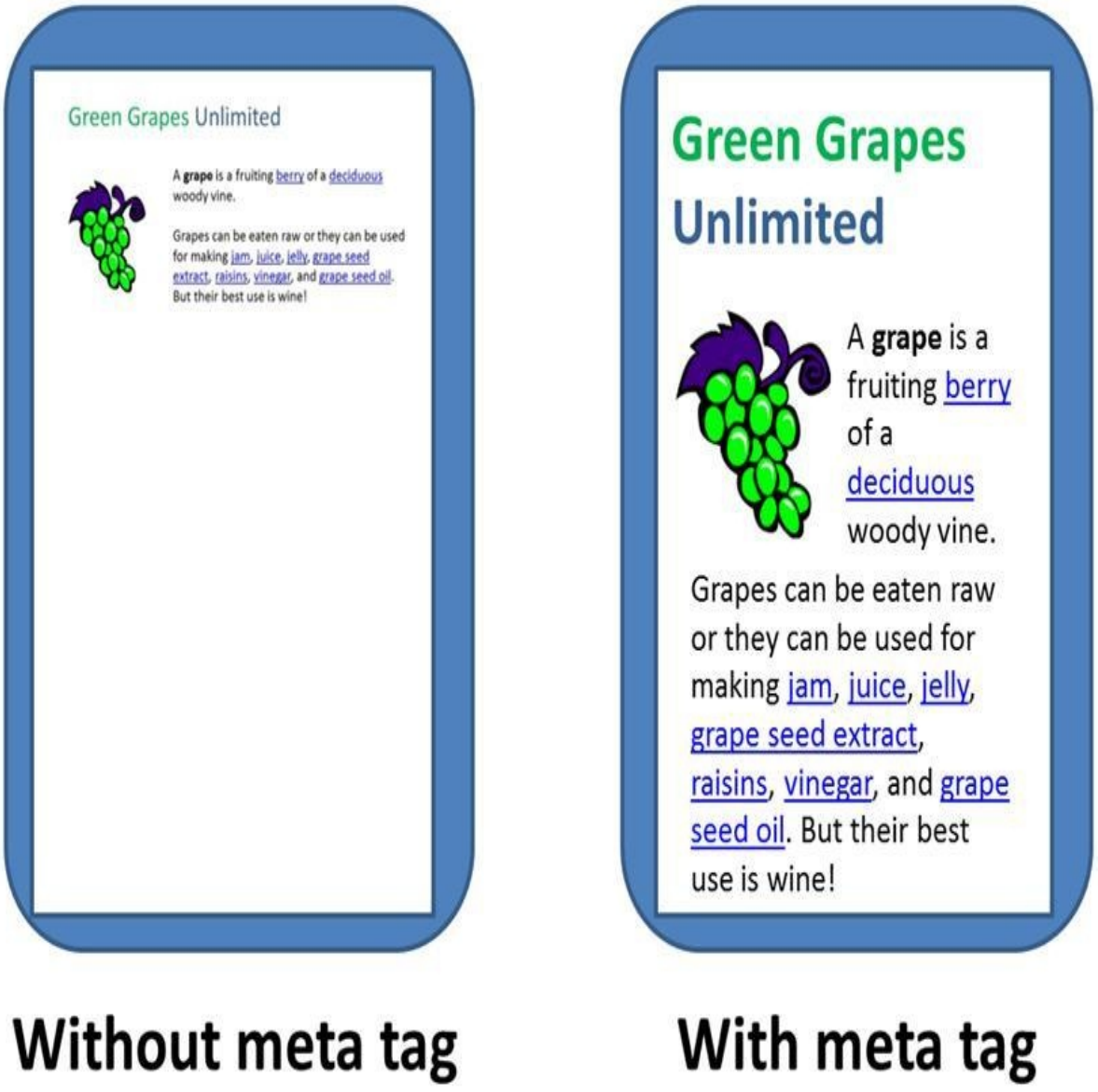
The Viewport Meta Tag

For a responsive website, we can do better. With a responsive website, each web page reconfigures itself to fit the device on which it is displayed; so there should be no need for the browser to shrink any web pages. With responsive sites, we need a way to tell the browser, "Don't shrink this page!"

The meta viewport tag serves this purpose. So when you create a responsive website, you need to add the following meta tag to the <head> section of each web page:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

When the browser sees this instruction, it will display content to fit the viewport of the user's device. The website will look good and be readable, whether it is displayed on a small smart phone or a large computer monitor. Assume that Green Grapes Unlimited is a responsive website. Note the effect with and without a viewport meta tag when the site is displayed on a smart phone.



Without a viewport meta tag, the page is shrunk, squeezed on to the screen, and text is hard to read. The user needs to zoom in. With the tag, content is sized to the viewport, text is readable, and zooming is not necessary.

Search Engine Optimization

Setting the viewport has implications for search engine optimization (SEO).

In May of 2015, Google added “mobile-friendliness” as a ranking factor in its search results. Other things being equal, sites that are mobile-friendly rank higher than sites that are not mobile-friendly.

How can you judge whether your site is mobile-friendly? Google built a helpful tool that you can access at https://www.google.com/webmasters/tools/mobile-friendly/. Use this tool to test any web page. Pages that lack a viewport meta tag are flagged with the warning: “Mobile viewport not set”. It looks like sites without a viewport meta tag are judged by Google to be less mobile-friendly than sites with the tag.

Conclusion

When you build a responsive website, you should always set the viewport. It provides guidance for the browser that makes the site easier for visitors to read. And it may help the site rank higher on Google’s search engine results page.

Part 4. Responsive Web Pages

With responsive web design, the look of a web page changes, based on browser features and capabilities. The web page reconfigures itself to fit the device and the browser on which it is displayed. The goal is for the web page to look good and load quickly on any device and any browser.

In this section, we'll walk step-by-step through real-world examples that show how to build responsive web components and responsive web pages with flexbox, using just HTML and CSS – no javascript.

- Chapter 13. Responsive Navigation
- Chapter 14. Responsive Footer
- Chapter 15. Responsive Layout
- Chapter 16. Responsive Page Template

The book culminates in Chapter 16, where we build a fully-functional template for a responsive web page. To see an actual website that that was built with this template, visit http://flexbox-rwd.com.

Chapter 13. Responsive Navigation

Responsive websites need responsive navigation. In this chapter, we'll build a navigation menu for a responsive website, working within the following constraints:

- The menu should be responsive.
- The code should work on all browsers.
- Performance should be lightning-quick.

To build this navigation menu, we'll design with progressive enhancement, as described in Chapter 8.

Basic vs Enhanced Menus

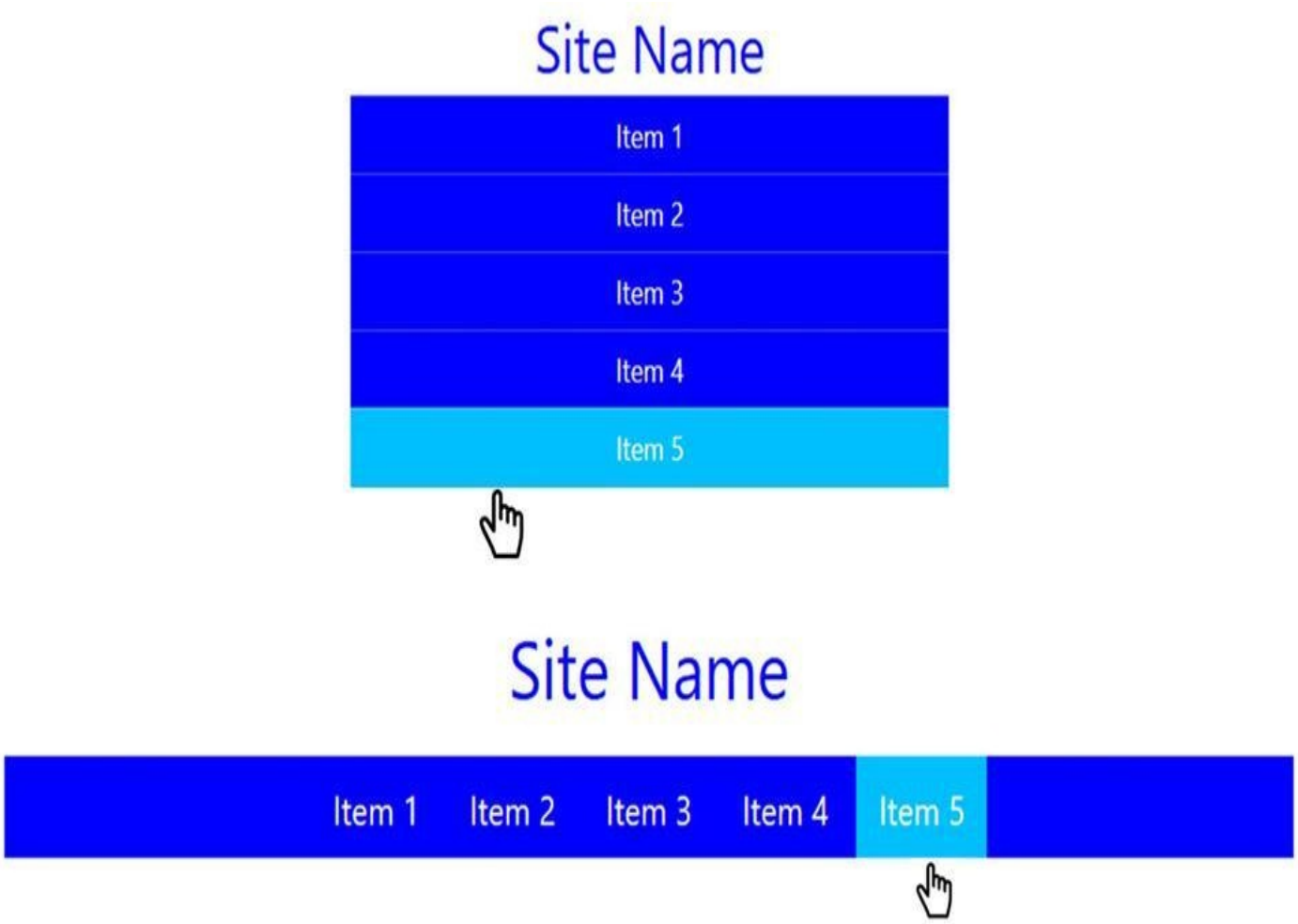
Progressive enhancement calls for a basic and an enhanced version of the navigation menu. For the basic version of the menu, we'll use only standard HTML and well-accepted CSS – no javascript and no advanced CSS3. This will result in a responsive navigation menu that works quickly in all browsers on all devices.

For the enhanced version of the menu, we'll continue to avoid javascript. This will result in code that is lean and quick. To save space on small screens, we'll hide the navigation, until it is requested by the user. On large screens, where space is not an issue, we'll display the navigation by default. And we'll add style flourishes as needed, with flexbox.

Basic Navigation

Let's begin with the basic version of our navigation menu. This is the version designed to work on all browsers and all devices. This is the version that will be displayed by older browsers that can't work with advanced features, like flexbox.

The figure below shows what we want this version to look like.



On the top, we see the vertical navigation menu that will be displayed on small viewports. And on the bottom, we see the horizontal menu that will be displayed on larger viewports.

Navigation Markup

The vertical and horizontal menus shown above use the same HTML:

HTML

```
<div class="header-nav">
    <!— Header —>
    <header class="nav-header">Site Name</header>
    <!— Navigation —>
    <nav class="header-nav">
        <ul class="header-nav">>
            <li><a class="header-nav" href="#"> Item 1</a></li>
            <li><a class="header-nav" href="#"> Item 2</a></li>
            <li><a class="header-nav" href="#"> Item 3</a></li>
            <li><a class="header-nav" href="#"> Item 4</a></li>
```

```
            <li><a class="header-nav" href="#"> Item 5</a></li>
        </ul>
    </nav>
</div>
```

The menus also share much of the same CSS. Here is the CSS markup that is common to both the vertical and horizontal menus:

CSS

```
html { box-sizing: border-box; font-size: 100%; padding: 0 1rem; min-width: 320px; font-family: Arial; line-height: 1.5; }

*, *:before, *:after { box-sizing: inherit; }

a.nav-header { text-decoration: none; color: blue; }

a.nav-header:hover { color: darkblue;}

nav.header-nav { background-color: blue; width: 100%; text-align: center;}

ul.header-nav { list-style-type: none; padding: 0; margin: 0 auto; }

a.header-nav { text-decoration: none; display: block; color: white; }
```

The difference between the horizontal and vertical menus amounts to just a few lines of code. Consistent with mobile-first design, we'll build the vertical menu used on small screens before we build the horizontal menu for large screens. We'll use media queries to set styles for different-sized viewports.

Note: Media queries are often used to style responsive web pages, and we will use media queries quite a bit in the examples that follow. If you need a quick media query refresher, see Chapter 9.

Here is the markup used to produce a vertical menu on viewports up to 850 pixels wide:

CSS

```
@media (max-width: 850px) {
    header.nav-header { text-align: center; }
    a.nav-header { font-size: 2.3rem; }
    label#menu { display: none; }
    nav.header-nav { display: block; }
    ul.header-nav { background-color: blue; }
    a.header-nav { font-size: 1.1rem; line-height: 2.333333; border-top: 1px solid rgba(255, 255, 255, 0.3); border-bottom: 1px solid rgba(0, 0, 0, 0.1);}
    a.header-nav:hover { background-color: deepbluesky; }
```

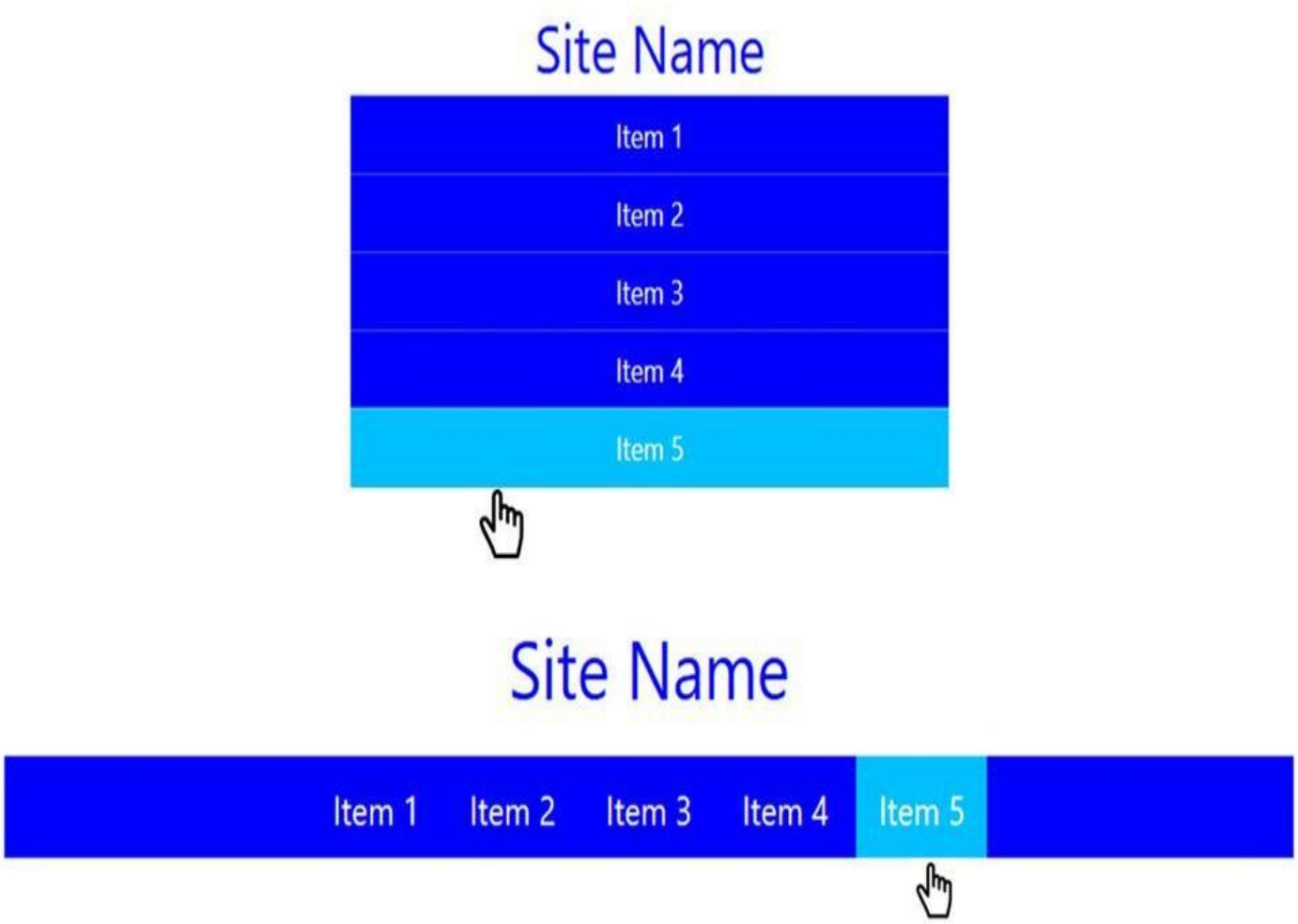
```
}
```

And here is the CSS markup used to produce a horizontal menu when the viewport is wider than 850 pixels:

CSS

```
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%;
    background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    nav.header-nav { display: block; height: 3.5rem; }
    ul.header-nav { background-color: blue; max-width: 1300px; }
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5;
    padding: 0 1rem; }
    a.header-nav:hover { background-color: deepbluesky; }
}
```

The choice of breakpoint depends on the horizontal space required to display all of the menu items on a single line. In this example, a breakpoint of 850 pixels does the trick. Once again, here is how the basic navigation menu looks on small and large screens.



This basic navigation menu is simple, responsive, reasonably attractive, and completely functional in all browsers. The main disadvantage is the space it takes up on small screens.

Advanced Checkbox Hack

One way to address the problem of limited space on small screens is to hide content until it is needed. To hide navigation on small screens, we'll use a technique called the advanced checkbox hack. Appendix C describes the advanced checkbox hack in gory detail, but here's the CliffsNotes explanation.

> ***NOTE***
>
> ***The checkbox hack was first described by Chris Coyier. You can read his original article at https://css-tricks.com/the-checkbox-hack/.***

The checkbox hack is built from three elements: a checkbox <input>, an associated <label>, and a third element whose style you want to control. For our responsive navigation menu, we want to control the visibility of menu items, so the <nav> element will be our third element. The HTML will look like this:

> **HTML**

```
<input type="checkbox" id="nav">
<label for="nav" > … </label>
<nav class="nav"> … </nav>
```

We use a "for" attribute to associate the label with the checkbox. Then, we use CSS to hide the checkbox. When a user clicks the label, the hidden checkbox is toggled on and off. Then, we use a general sibling selector (~) to control visibility of menu items, based on whether the checkbox is checked or unchecked.

NOTE

For a good description of the general siblings selectors (~), visit https://developer.mozilla.org/en-US/docs/Web/CSS/General_sibling_selectors.

Here's the CSS:

CSS

```
input { display: none; } /* Hide checkbox */
nav { display: none; } /* Hide menu */
input:checked ~ nav { display: block; } /* Show menu */
```

If you thought this looks too easy, you're right. There's a catch. The checkbox hack doesn't work in some Android and iOS devices. Luckily, there are simple solutions for problems with each device. Added to the original checkbox hack, these solutions are called the advanced checkbox hack.

NOTE

The advanced checkbox hack was developed by Tim Pietrusky. You can read his original article at http://timpietrusky.com/advanced-checkbox-hack.

The solution to problems with Android devices is to add the following CSS to the <body> element:

CSS

```
body { webkit-animation: bugfix infinite 1s; }
@-webkit-keyframes bugfix {
    from {padding:0;}
    to {padding:0;}
}
```

The solution for iOS devices is even simpler. Just add an empty “onclick” to the <label> element, as shown below in bold.

HTML

<label for=“nav” **onclick**>Menu</label>

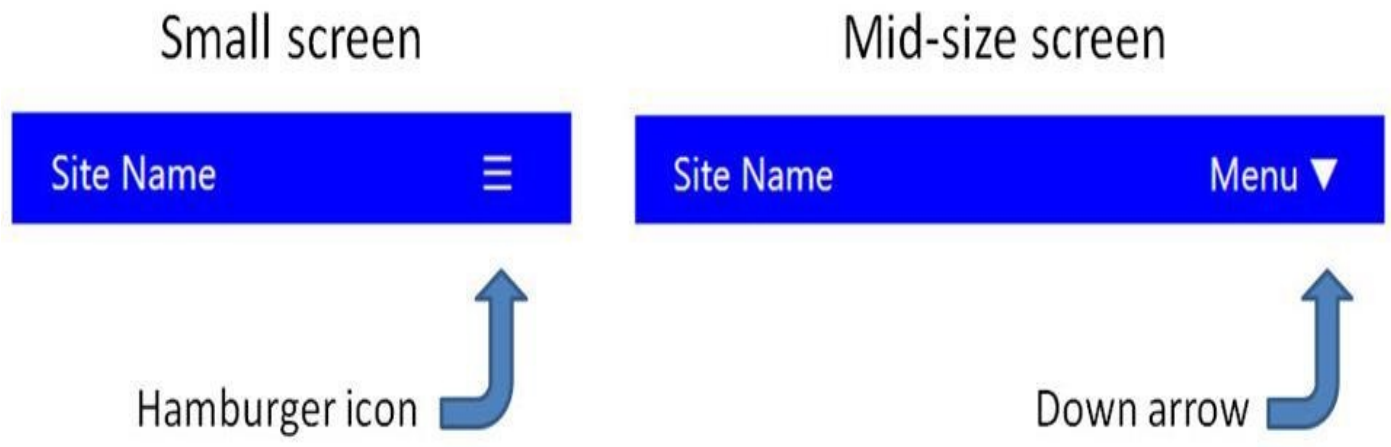
With this explanation behind us, we can use the advanced checkbox hack to hide and show the enhanced navigation menu on small screens.

Enhanced Navigation

In this section, we’ll build an enhanced navigation menu for use on modern browsers that understand flexbox. For small and mid-size viewports, we’ll build vertical menus that use the advanced checkbox hack to hide menu items until they are needed. And we’ll use flexbox to position <header>, <label> and <nav> elements on the screen. For large viewports, we’ll build a horizontal menu that uses flexbox to provide increased control over the positioning of menu items.

Vertical Navigation

The figure below shows how we want the enhanced version of vertical navigation to work on small and mid-size screens. When the page is first loaded, a <label> element is displayed next to the site name, navigation is hidden, and precious space is conserved on the screen. On small screens the <label> shows a “hamburger” icon; and on mid-size screens, it shows the word “Menu” with a down arrow.



After the <label> is clicked, a vertical navigation menu is displayed. On small screens, the “hamburger” icon becomes an “X” icon. On mid-size screens, the arrow points up. If the <label> is clicked again the navigation menu reverts to its hidden state.



This enhanced navigation menu uses the same HTML markup as we used to produce the basic navigation menu, plus two new lines of code (shown below in bold):

HTML

```
<div class="header-nav">
    <header>Site Name</header>
    <input type="checkbox" id="nav">
    <label for="nav" id="menu" onclick> </label>
    <nav class="header-nav">
        <ul class="header-nav">>
            <li><a class="header-nav" href="#"> Item 1</a></li>
            <li><a class="header-nav" href="#"> Item 2</a></li>
            <li><a class="header-nav" href="#"> Item 3</a></li>
            <li><a class="header-nav" href="#"> Item 4</a></li>
            <li><a class="header-nav" href="#"> Item 5</a></li>
        </ul>
    </nav>
</div>
```

The two new lines of code add an <input> checkbox and a <label> element to the

navigation menu. We will use these new elements to implement the advanced checkbox hack. When the label is clicked, it will toggle the checkbox to show or hide the navigation menu. But for that to happen, we need to add some code to the CSS markup.

First, we need to style the <input> checkbox and the <label> element. We want those to be hidden on large screens. Later, we will make the <label> visible on small and mid-size screens.

CSS

```
input#nav { display: none; }

label#menu { display: none; }
```

Next, we need to define styles when the viewport is small or mid-size. In this example, we'll define "small" as being no bigger than 500 pixels; and we'll define mid-size as being between 501 and 850 pixels. We'll use a media query to set ordinary styles (font size, colors, spacing, etc.) for small and mid-size viewports.

CSS

```
/* Small and mid-size viewports */

@media (max-width: 850px) {

    /* Ordinary style rules */

    .flexbox.flexwrap a.nav-header{ color: white; font-size: 1.4rem; line-height: 2.3333333; }

    .flexbox.flexwrap header.nav-header { width: auto; background-color: blue; color: white; line-height: 2.333333; text-align:left; padding-left: 1.5rem; }

    .flexbox.flexwrap label#menu { display: block; width: auto; background-color: blue; color: white; font-size: 1.4rem; font-weight: bold; line-height: 2.333333; text-align: center; cursor: pointer; }

    .flexbox.flexwrap label#menu:hover { background-color: darkblue; }

    .flexbox.flexwrap ul.header-nav { background-color: deepskyblue; }

    .flexbox.flexwrap a.header-nav:hover { background-color: darkblue; }

    .flexbox.flexwrap input#nav:checked ~ label#menu:hover { background-color: darkblue; }

}
```

Because the ".flexbox.flexwrap" selector only exists on browsers that support flexbox (see Building the Enhanced Version in Chapter 8), the code block shown above is only executed by browsers that support flexbox.

With vertical navigation, our focus is on several HTML elements. We have a parent element with three children - a <header>, <label>, and <nav> element. On

small and mid-sized screens, we would like the <header> and <label> elements to sit next to each other on the same row with no space between and no space on the edges. We would like the <label> to maintain a fixed width, but we would like the <header> to grow and shrink, as the viewport expands and contracts. And we would like the <nav> element to sit below the <header> and <label> on a second row.

This sounds like a job for flexbox. We need to turn the parent into a multi-line flex container with three flex items. Here's the CSS markup to do that.

CSS

```
@media (max-width: 850px) {
    /* Flexbox properties */
    .flexbox.flexwrap div { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header { flex: 1 1 auto; width: auto; }
    .flexbox.flexwrap label { display: block; flex: 0 0 140px; }

    /* Checkbox hack */
    .flexbox.flexwrap nav.header-nav.header-nav { display: none; }
    .flexbox.flexwrap input#nav:checked ~ nav.header-nav { display: block; }

}
```

When it is executed, this code block has the following effects:

- **div.** The "display: flex" declaration turns the parent into a flex container and makes all of its children flex items. The "flex-wrap: wrap" declaration makes the container a multi-line flex container.

- **header.** We set the <header> "flex" and "width" properties with the following declarations: "flex: 1 1 auto" and "width: auto". This causes the initial width of the <header> element to be defined by its content, but it allows the <header> to grow or shrink as the viewport expands and contracts.

- **label.** Recall that the <label> element was initially hidden on large screens. The "display: block" declaration makes it visible on small screens. And the "flex: 0 0 140px" declaration gives it a fixed width of 140 pixels.

- **nav.** The <nav> element holds menu items. The "display: none" declaration causes it to be hidden initially. However, when the input box is checked, the "display" value is set to "block", and the <nav> element becomes visible. Previously, the width of the <nav> element was set at 100%, so it fills a row

all by itself. As a result, the <nav> element wraps to the second row of the multi-line flex container when it becomes visible.

Recall that we wanted to display different content in the <label> element, depending to whether viewport was small or mid-sized. On small viewports, we wanted to display a “hamburger” icon or an “X” icon. And on mid-sized viewports, we wanted to display “Menu” text plus an up or down arrow. Here’s the code to make that happen.

CSS

```
@media (max-width: 500px) {
    /* Generated label content*/
    label#menu:before { content: “\2630”; } /* hamburger icon */
    input#nav:checked ~ label#menu:before { content: “X”; } /* x icon */
}
@media (min-width: 501px) and (max-width: 850px) {
    /* Generated label content*/
    label#menu:before { content: “Menu \25BC”; } /* unicode down arrow */
    input#nav:checked ~ label#menu:before { content: “Menu \25B2”; } /*
    unicode up arrow */
}
```

Here, we’re using the “:before” pseudo-element to add content to our <label> element. On small screens, the <label> element will show a “hamburger” icon when the menu items are hidden; and, a big “X” when they are displayed. On mid-sized screens, it will show “Menu” text, plus a down arrow when menu items are hidden and an up arrow when menu items are displayed.

With the code we’ve written to this point, the <label> element is displayed 140 pixels wide. That is a good size to show the “Menu” text and the up and down arrows used on mid-sized screens; but it is bigger than necessary to show the “hamburger” or “X” icons used on small screens. We only need 90 pixels to display the “hamburger” or “X” icon on small viewports. We can make that adjustment in a media query.

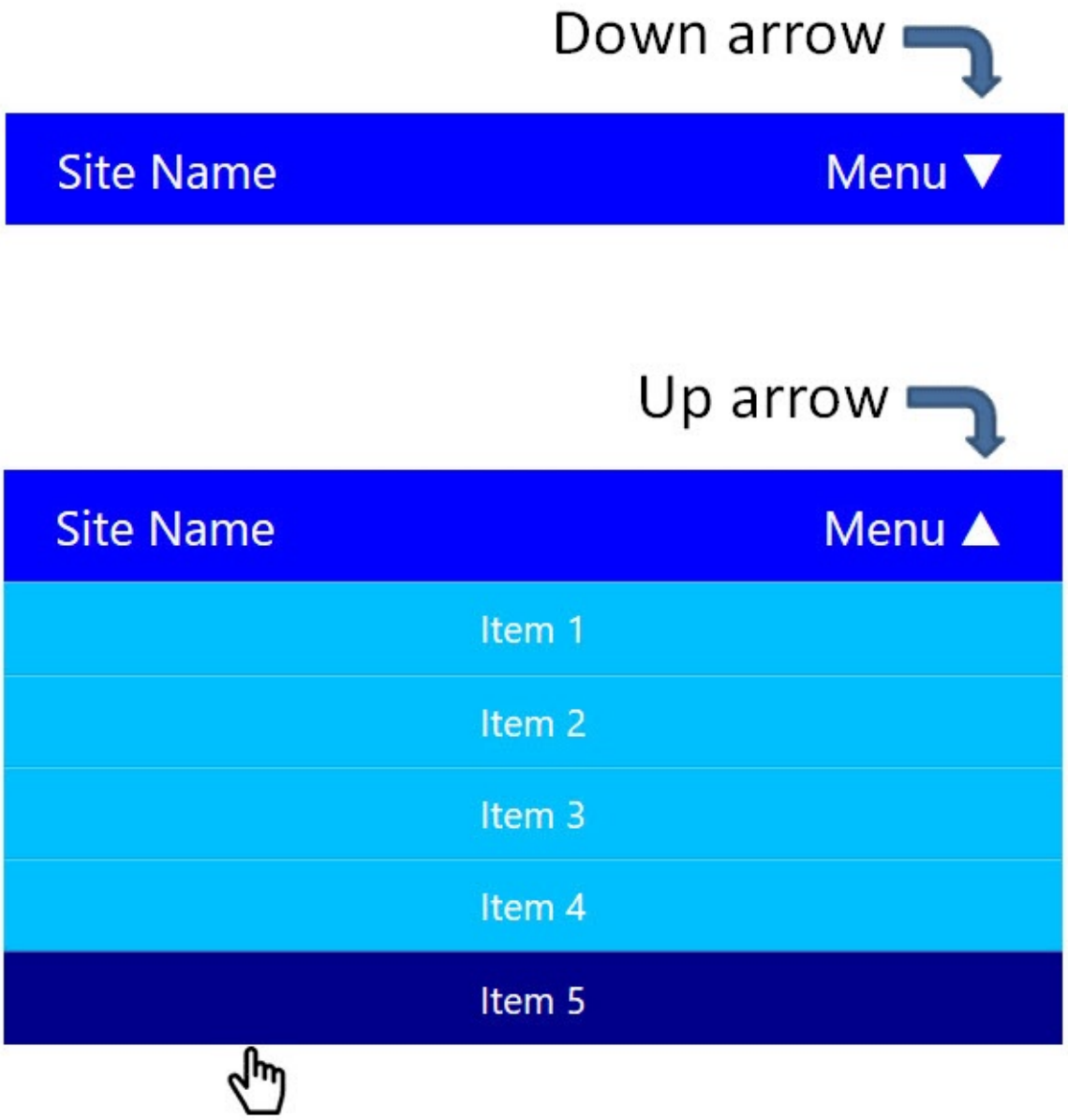
CSS

```
@media (max-width: 500px) {
    /* Set label width to 90 pixels */
    .flexbox.flexwrap label { display: block; flex: 0 0 90px; }
}
```

This small change leaves more space for the <header> element, which may come

in handy on small screens.

And finally, let's look once again at the vertical navigation menu that we are building for mid-sized viewports:



In each view, notice the menu label. There is a down arrow when navigation is hidden and an up arrow when navigation is displayed. Here is the code to make that happen:

CSS

```
@media (min-width: 501px) and (max-width: 850px) {
    /* unicode down arrow */
    .flexbox.flexwrap label:after { content: "\25BC"; padding-left: 5px; }
    /* unicode up arrow */
    .flexbox.flexwrap input:checked ~ label:after { content: "\25B2"; }
}
```

That’s it for vertical navigation. If you’re dazzled by all the details, don’t worry. At the end of the chapter, we’ll pull everything together. Before that, though, let’s look at some enhancements to the horizontal navigation menu.

Horizontal Navigation

Before we create an enhanced version of a horizontal navigation menu, let’s remember what the basic version looks like and how it works. Here’s what the basic version of our horizontal menu looks like.



And here’s how it works. On browsers that don’t understand flexbox, the basic version of the horizontal menu is displayed when the viewport is at least 851 pixels wide. Menu items are centered beneath the site name in a single row, with equal space between items. And when the user hovers over a menu item, it is highlighted.

The HTML used to produce the basic horizontal navigation menu is identical to the HTML used to produce the vertical navigation menu.

HTML

```
<div>
    <header class=”nav-header”>Site Name</header>
    <input type=“checkbox” id=“nav”>
    <label for=“nav” id=“menu” onclick>Menu</label>
    <nav class=”header-nav”>
        <ul class=”header-nav”>>
            <li><a class=”header-nav” href=”#”>Item 1</a></li>
            <li><a class=”header-nav” href=”#”> Item 2</a></li>
            <li><a class=”header-nav” href=”#”> Item 3</a></li>
            <li><a class=”header-nav” href=”#”> Item 4</a></li>
            <li><a class=”header-nav” href=”#”> Item 5</a></li>
        </ul>
    </nav>
```

```
</div>
```

And here is the CSS used to style the basic horizontal navigation menu.

CSS

```
/* Large screen: Basic */
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%;
    background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    input#nav { display: none; }
    label#menu { display:none; }
    nav.header-nav { display: block; height: 3.5rem; }
    ul.header-nav { background-color: blue; max-width: 1300px; }
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5;
    padding: 0 1rem; }
    a.header-nav:hover { background-color: deepskyblue; }
}
```

Notice that the <input> and <label> elements are hidden with a “display: none” declaration (shown in bold). The <input> and <label> elements play no role in the horizontal navigation menu.

The basic version of the horizontal navigation menu is user-friendly, looks fine, and runs quickly. A web designer could use this navigation menu without embarrassment. At the same time, a web designer might see opportunities for improvement with flexbox. Let’s look at a few examples.

Consider this situation. On very wide screens, the basic version of the horizontal navigation menu will tend to squeeze menu items in the center of the navigation bar, as shown below:



This is not terrible. But a designer might prefer spacing between menu items to vary, based on viewport size. Ideally, there would be more spacing between items

on large viewports, and less spacing between items on small viewports. This enhancement is very easy to accomplish with flexbox – just one line of CSS added to the media query code block:

CSS

```
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%; background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    input#nav { display: none; }
    label#menu { display:none; }
    nav.header-nav { display: block; height: 3.5rem; }
    ul.header-nav { background-color: blue; max-width: 1300px; }
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5; padding: 0 1rem; }
    a.header-nav:hover { background-color: deepskyblue; }
    .flexbox.flexwrap ul { display:flex; justify-content:space-around; }
}
```

The new line of code is shown in bold. The “display: flex” declaration turns the unordered list into a flex container. And it turns the list items into flex items. The “justify-content”: space-around” declaration distributes flex items such that (a) there is equal space between adjacent items and (b) space between the end items and flex container edge is half the space between adjacent items.

Here is what the enhanced horizontal navigation looks like at various screen widths:



Note: You can produce a similar effect without flexbox by setting the width of each list item to 20% and floating items to the left. This works with the five menu items used in this example. However, if you used only four menu items, you would need to change the width of each list item to 25%. Or if you used six menu items, you would need to change the width to 16.66%. And you need to use the clearfix hack to guard against layout problems. With flexbox, you don't need to calculate or set widths. And you don't need to deal with floats or clears. Everything is handled automatically.

Another situation where flexbox comes in handy occurs when you want to position one set of menu items on the left side of the page, and something else on the right. You see this when a navigation bar shows menu items on the left and a site search widget on the right. Or you might see menu items on the left and social media icons on the right.

In the past, designers have used floats and clearfixes to produce this layout. But they are not needed with flexbox. For example, suppose we wanted to position four menu items on the left side of the page and one item by itself on the right. Here is the layout we want to produce:



To produce this layout, we only need to add two lines of code to the horizontal navigation markup:

CSS

```
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%;
    background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    input#nav { display: none; }
    label#menu { display:none; }
    nav.header-nav { display: block; height: 3.5rem; }
    ul.header-nav { background-color: blue; max-width: 1300px; }
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5;
    padding: 0 1rem; }
    a.header-nav:hover { background-color: deepskyblue; }
    .flexbox.flexwrap ul { display:flex; }
    .flexbox.flexwrap ul li:last-of-type { margin-left: auto; }
}
```

The new lines of code are shown in bold. We used the “display: flex” declaration to turn the unordered list into a flex container. This turned each of the list items into flex items. And we used the “margin-left: auto” declaration to push the last menu item to the right side of the page. If you’ve forgotten how auto margins work with flex items, review the explanation in Chapter 5.

All the Navigation Code

We’ve covered a lot of ground in this chapter. We talked about basic and enhanced navigation menus. We produced vertical menus and horizontal menus. We introduced the advanced checkbox hack. We showed how to change the

direction of the menu arrow dynamically. We covered each topic separately, so you may be a little bit baffled about how everything fits together.

In this section, we put it all together. We show all the code – everything in one place. Here's the HTML:

HTML

```
<!— Container for header and navigation —>
<div class=”header-nav”>
    <!— Header —>
    <header class=”nav-header”>Site Name</header>
    <input type=“checkbox” id=“nav”>
    <label for=“nav” id=“menu” onclick>Menu</label>
    <!— Navigation —>
    <nav class=”header-nav”>
        <ul class=”header-nav”>>
            <li><a class=”header-nav” href=”#”> Item 1</a></li>
            <li><a class=”header-nav” href=”#”> Item 2</a></li>
            <li><a class=”header-nav” href=”#”> Item 3</a></li>
            <li><a class=”header-nav” href=”#”> Item 4</a></li>
            <li><a class=”header-nav” href=”#”> Item 5</a></li>
        </ul>
    </nav>
</div>
```

Here's the CSS used to overcome some problems with the advanced checkbox hack:

CSS

```
body { webkit-animation: bugfix infinite 1s; }
@-webkit-keyframes bugfix {
    from {padding:0;}
    to {padding:0;}
}
```

Some of the CSS markup is common to all versions of the navigation menu.

CSS

```
/* All screens: Common styles */
a.nav-header { text-decoration: none; }
a.nav-header:hover { color: darkblue;}
input#nav { display: none; }
nav.header-nav { background-color: blue; width: 100%; text-align: center;}
ul.header-nav { list-style-type: none; padding: 0; margin: 0 auto; }
a.header-nav { text-decoration: none; display: block; color: white; }
```

Here's the CSS for the basic version of the vertical navigation menu:

CSS

```
/* Small and medium screens */
@media (max-width: 850px) {
    /* Basic */
    header.nav-header { text-align: center; }
    a.nav-header { font-size: 2.3rem; }
    label#menu { display: none; }
    nav.header-nav { display: block; }
    ul.header-nav { background-color: blue; }
    a.header-nav { font-size: 1.1rem; line-height: 2.333333; border-top: 1px solid rgba(255, 255, 255, 0.3); border-bottom: 1px solid rgba(0, 0, 0, 0.1);}
    a.header-nav:hover { background-color: deepskyblue; }
}
```

Here's the CSS for the basic version of the horizontal navigation menu:

CSS

```
/* Large screen: Basic */
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%; background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    label#menu { display:none; }
    nav.header-nav { display: block; height: 3.5rem;  }
    ul.header-nav { background-color: blue; max-width: 1300px; }
```

```
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5; padding: 0 1rem; }
    a.header-nav:hover { background-color: deepskyblue; }
}
```

Here are CSS style rules for the enhanced version of the vertical navigation menu. These rules apply to both small and mid-size screens:

CSS

```
/* Small and mid-size screens */
@media (max-width: 850px) {
    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}

    /* Checkbox hack */
    .flexbox.flexwrap nav.header-nav { display: none; }
    .flexbox.flexwrap input#nav:checked ~ nav.header-nav  { display: block; }

    /* Ordinary style rules */
    .flexbox.flexwrap a.nav-header{ color: white; font-size: 1.4rem; line-height: 2.3333333; }
    .flexbox.flexwrap header.nav-header { width: auto; background-color: blue; color: white; line-height: 2.333333; text-align:left; padding-left: 1.5rem; }
    .flexbox.flexwrap label#menu { display: block; width: auto; background-color: blue; color: white; font-size: 1.4rem; font-weight: bold; line-height: 2.333333; text-align: center; cursor: pointer; }
    .flexbox.flexwrap label#menu:hover { background-color: darkblue; }
    .flexbox.flexwrap ul.header-nav { background-color: deepskyblue; }
    .flexbox.flexwrap a.header-nav:hover { background-color: darkblue; }
    .flexbox.flexwrap input#nav:checked ~ label#menu:hover { background-color: darkblue; }
}
```

Here are additional CSS rules for the enhanced version of the vertical navigation menu. These are additional rules for mid-size screens:

CSS

```
/* Mid-size screen: */
@media (min-width: 501px) and (max-width: 850px) {
    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}
    .flexbox.flexwrap label#menu { flex: 0 0 140px; font-weight: normal; }

    /* Generated label content*/
    .flexbox.flexwrap label#menu:before { content: “Menu \25BC”; } /* unicode down arrow */
    .flexbox.flexwrap input#nav:checked ~ label#menu:before { content: “Menu \25B2”; } /* unicode up arrow */
    }
}
```

Here are additional CSS rules for the enhanced version of the vertical navigation menu. These are additional rules for small screens:

CSS

```
/* Small screen: Enhanced */
@media (max-width: 500px) {

    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}
    .flexbox.flexwrap label#menu { flex: 0 0 90px; }

    /* Generated label content*/
    .flexbox.flexwrap label#menu:before { content: “\2630”; } /* hamburger icon */
    .flexbox.flexwrap input#nav:checked ~ label#menu:before { content: “X”; } /* x icon */
    }
}
```

And finally, here's the CSS for an enhanced version of the horizontal navigation menu:

CSS

```
/* Large screen: Enhanced */
@media (min-width: 851px) {
    /* Menu items equally spaced */
    .flexbox.flexwrap ul.header-nav { display:flex; justify-content: space-around; }
}
```

That's a lot of code to copy. You can download this source code for a responsive navigation menu from http://flexbox-rwd.com/download.aspx. Look for header-nav.html in the Chapter 13 directory.

Chapter 14. Responsive Footer

If you survived the previous chapter on navigation, building a responsive footer will be a piece of cake. We will build the footer in three steps.

- **Basic version.** The basic version of the footer will be built from standard HTML and CSS.
- **Enhanced version.** The enhanced version will require only two additional flexbox declarations.
- **Optional content.** As an option, we may generate additional content for display on large screens.

Here is what our footer will look like on a large screen.



This footer will be flexible. It will shrink and expand automatically to fit the viewport on which it is displayed.

HTML Markup

We will use the same HTML for all conditions: for big screens as well as small screens, and for the basic version of the website as well as the enhanced version. Here's the HTML for our footer:

HTML

```
<!— Footer —>
<footer>
    <!—Navigation bar —>
    <nav class="footer-nav">
        <ul class="footer-nav">
            <li><a class="footer-nav " href="#">Item 1</a></li>
            <li><a class="footer-nav " href="#">Item 2</a></li>
            <li><a class="footer-nav " href="#">Item 3</a></li>
            <li><a class="footer-nav " href="#">Item 4</a></li>
        </ul>
```

```
    </nav>
    <!—Footer details —>
    <div class=“footer-details”>
        <p class=“footer-details”>Copyright &copy; 2016</p>
        <p class=“footer-details”>Powered by <a href=“http://flexbox-rwd.com/”>flexbox-rwd</a></p>
    </div>
</footer>
```

The main part of this footer is the horizontal navigation bar. There are four menu items in this version of the navigation bar. You can change that by adding or subtracting list items in the unordered list.

CSS Markup

As usual, CSS is where the action is. We will create separate style rules to control the look of the footer on small screens, medium-sized screens, and large screens. We will write one set of style rules for older browsers that don’t understand flexbox; this produces the basic version of the footer. We will write a few additional rules for modern browsers to produce the enhanced version of the footer. And finally, we’ll use CSS to generate additional content for large viewports.

Basic Version

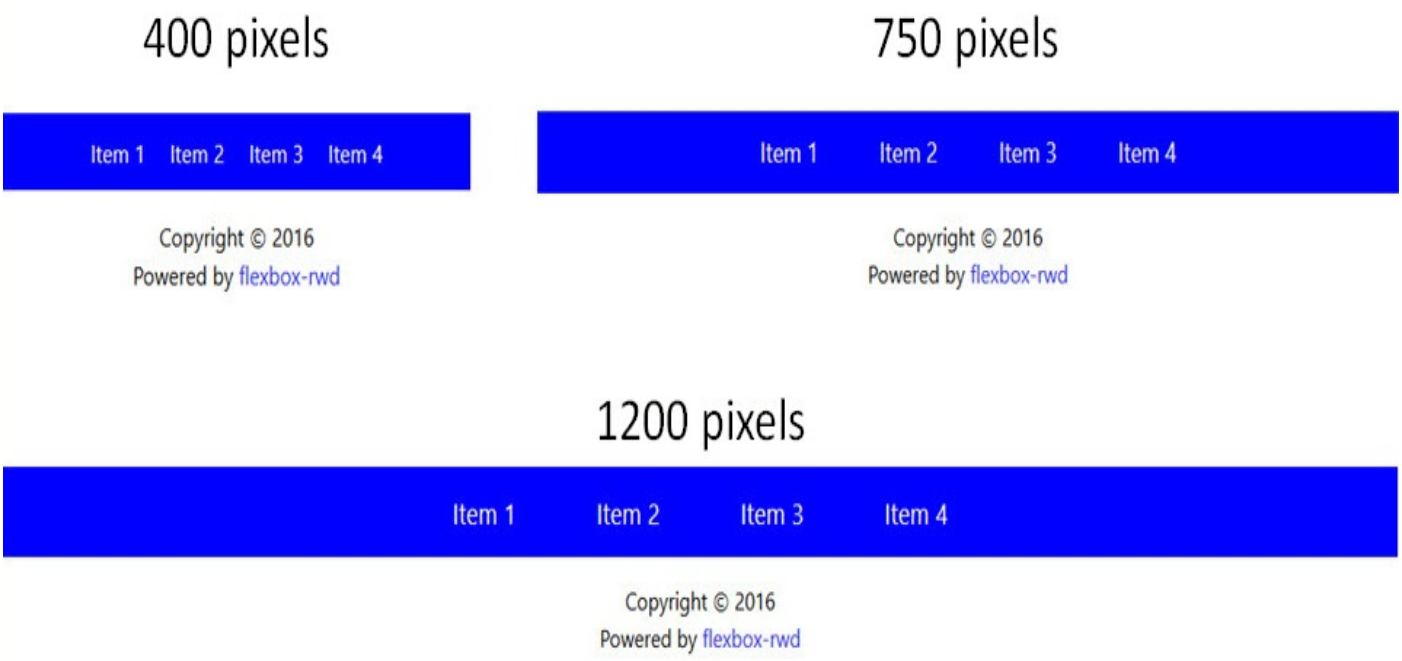
First, let’s build the basic version of the footer. This is built entirely from standard HTML and standard CSS – no advanced features – so it should work on all devices and all browsers.

CSS

```
/* Common basic styles */
footer { background-color: white; }
nav.footer-nav { background-color: blue; width: 100%; text-align: center;}
ul.footer-nav { list-style: none; background-color: blue; margin:0 auto; padding: 0; max-width: 1200px; }
.footer-nav li { display: inline-block; }
a.footer-nav { display: inline-block; text-decoration: none; color: white; line-height: 3; }
a.footer-nav:hover { background-color: deepskyblue; }
div.footer-details { max-width: 1200px; margin: 1rem auto; text-align:center; }
```

```
/* Small screens */
@media (max-width: 500px) {
    a.footer-nav { font-size: 1rem; padding: 0 .5rem;}
}

/* Medium screens */
@media (min-width: 501px) and (max-width: 850px) {
    a.footer-nav { font-size: 1.1rem; padding: 0 1.5rem;}
}

/* Large screens */
@media (min-width: 851px) {
    a.footer-nav { font-size: 1.2rem; padding: 0 2rem;}
}
```

For the most part, the same code is used on small screens, medium-sized screens, and large screens. The only thing that changes is font size and padding of footer anchor elements. We use media queries to assign bigger fonts and more padding as screen size increases. Here's the result at various screen sizes.



This looks ok. But some might prefer menu items to be spaced more widely along the navigation bar, especially on wide screens. We can make that happen easily with flexbox.

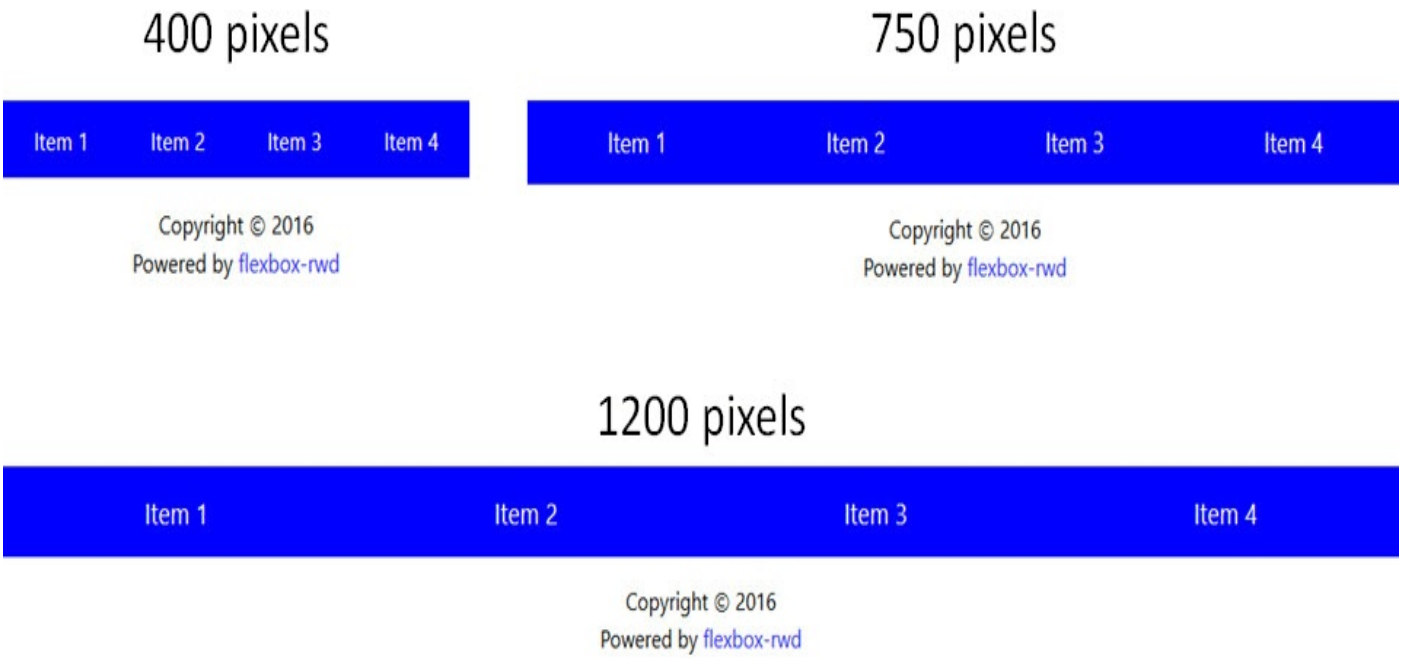
Enhanced Version

With a few simple CSS declarations, we can change the spacing of the links within the navigation bar. Here's the code:

CSS

```
/* Enhanced */

.flexbox.flexwrap ul.footer-nav { display: flex; justify-content: space-around; }
```

Here's what's happening. The "display: flex" declaration turns the <ul> element into a flex container, and it turns children of the <ul> element into flex items. Thus, the <li> elements become flex items. The "justify-content: space-around" declaration controls how list items are positioned along the navigation bar. Specifically, it places equal space between adjacent list items, and half that space between the end items and the edge of the flex container. Here's the result:



This is better. The menu items no longer huddle in the middle of the navigation bar; instead, they spread out naturally to fill the available space.

And speaking of available space, notice that there is a lot of unused space when the footer is displayed on a wide screen (at 1200 pixels in the figure above). We can take advantage of that space to use more descriptive labels within the navigation bar.

Suppose, for example, that that we use "real-world" menu labels, like "About", "Contact", "Privacy", and "Terms" instead of "Item 1", "Item 2", "Item 3", and "Item 4". This only requires a small edit (shown below in bold) to our existing HTML.

HTML

```
<!—Navigation bar —>
<nav class="footer-nav">
    <ul class="footer-nav">
```

```
        <li><a class="footer-nav " href="#">About</a></li>
        <li><a class="footer-nav " href="#">Contact</a></li>
        <li><a class="footer-nav " href="#">Privacy</a></li>
        <li><a class="footer-nav " href="#">Terms</a></li>
    </ul>
</nav>
```

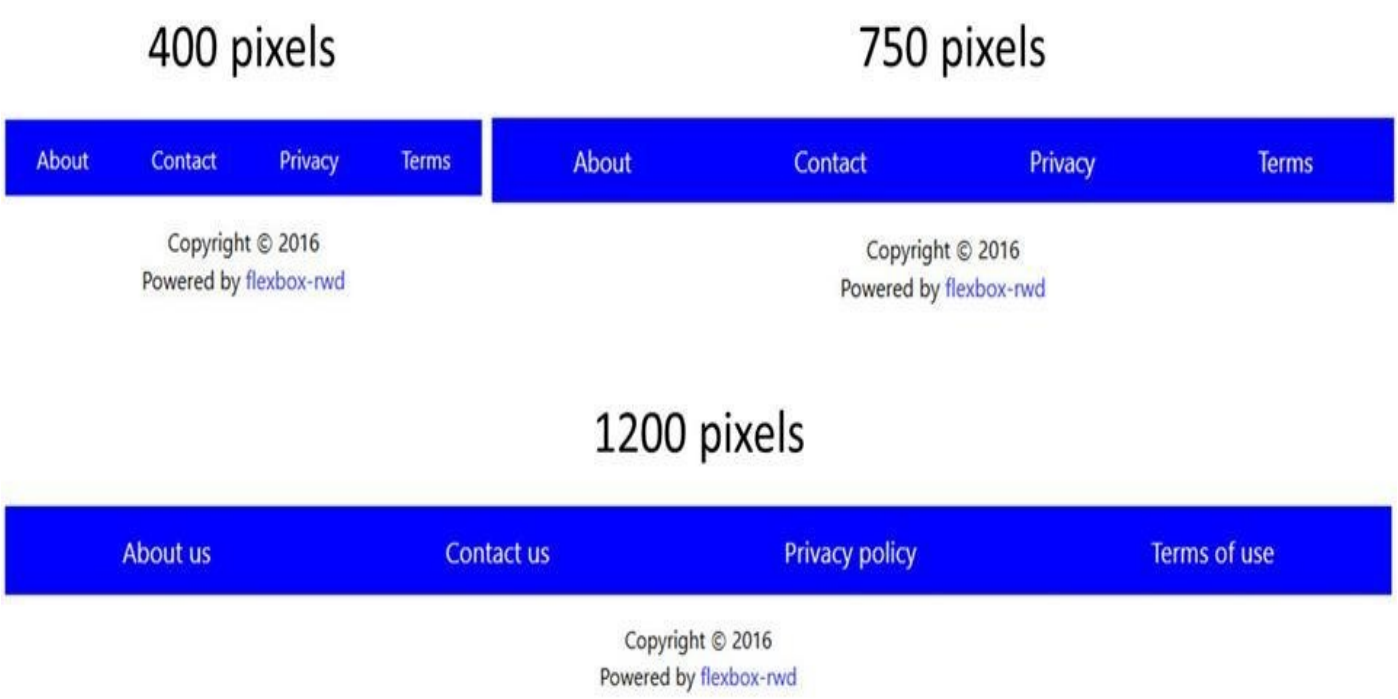
On larger screens, we might like to use “About us” instead of “About”; “Contact us” instead of “Contact”; “Privacy policy” instead of “Privacy”; and “Terms of use” instead of “Terms”.

We can implement this improvement with CSS, via an “:after” pseudo-element. Here’s the markup:

CSS

```
/* Large screens: Optional */
@media (min-width: 851px) {
    /* Additional content */
    ul.footer-nav li:nth-child(1) a:after { content: " us";}
    ul.footer-nav li:nth-child(2) a:after { content: " us";}
    ul.footer-nav li:nth-child(3) a:after { content: " policy";}
    ul.footer-nav li:nth-child(4) a:after { content: " of use";}
}
```

With this code, we are generating new content programmatically whenever the viewport is bigger than 850 pixels. Here is what the “real-world” footer looks like at various screen sizes, now:



Small screens display abbreviated menu labels; large screens display more descriptive labels. The use of descriptive labels on large screens is optional – not essential, but maybe a little bit more user-friendly. Notice that we didn't adjust the horizontal spacing to handle the extra text added on large screens. Flexbox positioned everything for us, automatically.

All the Footer Code

In this chapter, we created a basic and enhanced version of a simple, responsive footer. Here's the HTML markup for our footer:

HTML

```
<!— Footer —>
<footer>
    <!—Navigation bar —>
    <nav class="footer-nav">
        <ul class="footer-nav">
            <li><a class="footer-nav" href="#">Item 1</a></li>
            <li><a class="footer-nav" href="#">Item 2</a></li>
            <li><a class="footer-nav" href="#">Item 3</a></li>
            <li><a class="footer-nav" href="#">Item 4</a></li>
        </ul>
    </nav>
    <!—Footer details —>
```

```
    <div class="footer-details">
        <p class="footer-details">Copyright &copy; 2016</p>
        <p class="footer-details">Powered by <a href="http://flexbox-rwd.com/">flexbox-rwd</a></p>
    </div>
</footer>
```

And here is the CSS markup:

CSS

```
/* Common basic styles */
footer { background-color: white; }
nav.footer-nav { background-color: blue; width: 100%; text-align: center;}
ul.footer-nav { list-style: none; background-color: blue; margin:0 auto; padding: 0; max-width: 1200px; }
.footer-nav li { display: inline-block; }
a.footer-nav { display: inline-block; text-decoration: none; color: white; line-height: 3; }
a.footer-nav:hover { background-color: deepskyblue; }
div.footer-details { max-width: 1200px; margin: 1rem auto; text-align:center; }

/* Small screens */
@media (max-width: 500px) {
    a.footer-nav { font-size: 1rem; padding: 0 .5rem;}
}

/* Medium screens */
@media (min-width: 501px) and (max-width: 850px) {
    a.footer-nav { font-size: 1.1rem; padding: 0 1.5rem;}
}

/* Large screens */
@media (min-width: 851px) {
    a.footer-nav { font-size: 1.2rem; padding: 0 2rem;}
```

```
}
/* Enhanced */
.flexbox.flexwrap ul.footer-nav { display: flex; justify-content: space-around; }
```

Nobody wants to copy all this code by hand. You can download the source code for this responsive footer from http://flexbox-rwd.com/download.aspx. Look for footer.html in the Chapter 14 directory.

Chapter 15. Responsive Layout

In 2012, Luke Wroblewski identified five popular layout patterns used by responsive websites. He gave each pattern a name: Tiny Tweaks, Column Drop, Mostly Fluid, Layout Shifter, and Off Canvas (see http://www.lukew.com/ff/entry.asp?1514).

Pete LePage, a developer advocate for Google, showed how to create each of Wrobleski's layouts using flexbox and javascript for some of the more complicated designs (see https://developers.google.com/web/fundamentals/design-and-ui/responsive/patterns/?hl=en).

This chapter builds on the excellent work of Wroblewski and LePage. In this chapter, we modify LePage's markup a little bit – using flexbox but no javascript – to produce the five layout patterns identified by Wroblewski.

Page Structure

Web pages are made up of one or more components. For example, a web page might include a header, navigation, main content, footer, ads, etc.

In this chapter, we'll build layouts for web pages that have four components - a header, a navigation bar, main content, and an ad. This will allow us to create layout examples that are simple to understand; yet, still reflect real-world layout challenges.

For each example, we'll use roughly the same HTML.

HTML

```
<div class="container">
    <div class="header">Header</div>
    <div class="nav">Nav</div>
    <div class="main">Main</div>
    <div class="ad">Ad</div>
</div>
```

From this basic HTML, we'll use different CSS rules to generate different page layouts.

Progressive Enhancement

In Chapter 8, we described how to use progressive enhancement to build web pages that work on all browsers and all devices. You will recall that progressive enhancement involves three steps:

- Build a basic version of each web page, using only HTML and widely-

supported CSS2 – no javascript and no advanced CSS, like flexbox.

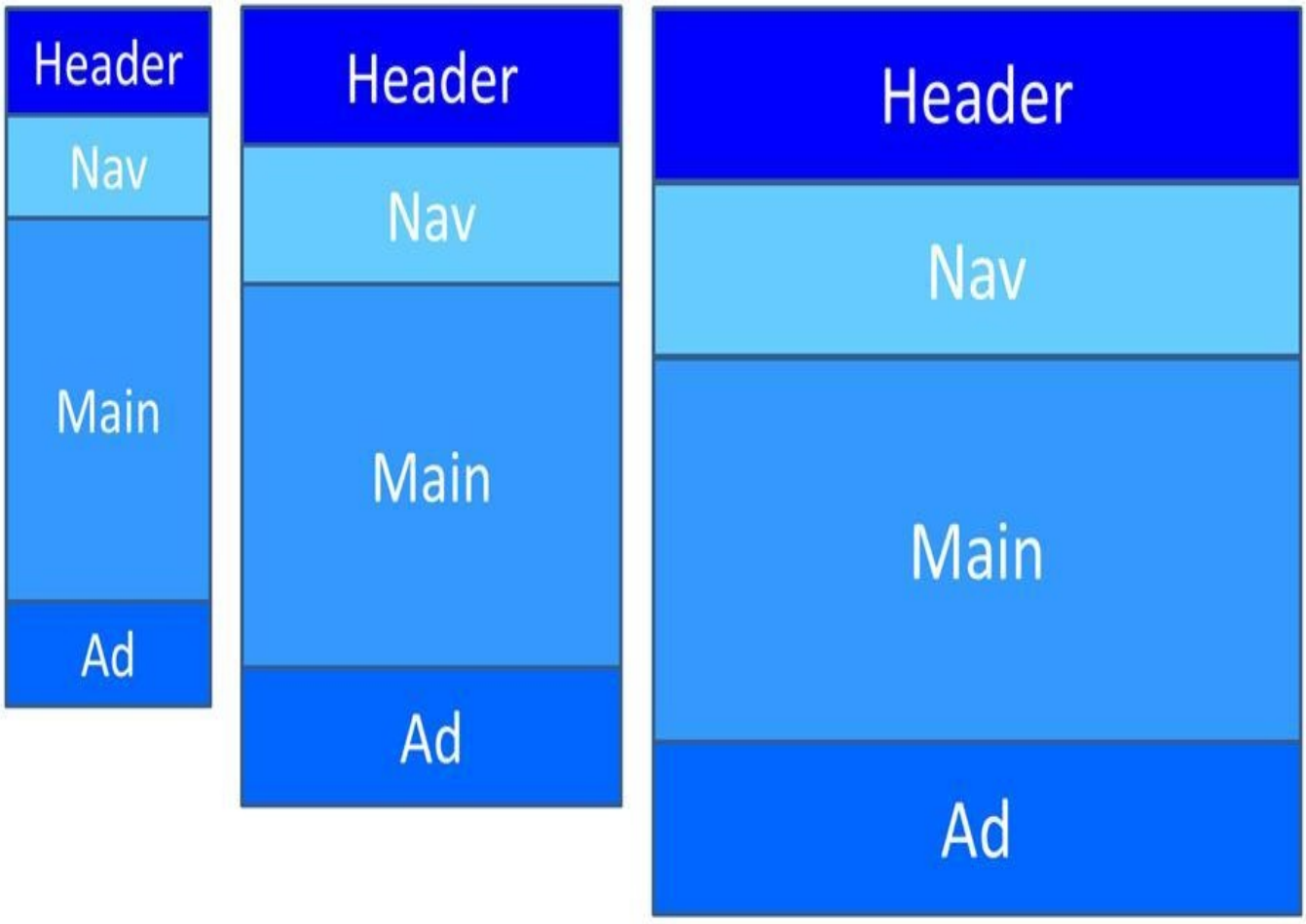
- Test browser support for advanced features, like flexbox.
- Build an enhanced version of each web page for browsers that support advanced features.

Browsers that don't support advanced features see the basic version of the web page. Browsers that do support advanced features see the enhanced version. But all browsers see a fully-functional, responsive web page.

In the examples that follow, we'll produce five page layouts - Tiny Tweaks, Column Drop, Mostly Fluid, Layout Shifter, and Off Canvas. The Tiny Tweaks layout uses no advanced features, so it will serve as the basic version of our web page. Each of the other layouts uses flexbox, so they each represent an enhanced version.

Tiny Tweaks

Tiny Tweaks is the simplest layout, consisting of a single column that expands or contracts to fit the user's viewport. The figure below shows a Tiny Tweaks design on different viewports.



With the Tiny Tweaks design, page components sit on top of each other in a single column. Column width grows or grows to fit the viewport on which the page is displayed. Making a web page fully responsive requires only a few tweaks to

font size and to images. Guidelines for producing responsive text are described in Chapter 10, and guidelines for producing responsive images are described in Chapter 11.

Tiny Tweaks Markup

Here's CSS code for the simple Tiny Tweaks layout shown in the figure above:

CSS

```
.header, .nav, .main, .ad { padding: 40px; width: 100%; }
.header {background-color:#007;}
.nav {background-color:#070;}
.main {background-color:#700;}
.ad {background-color: highlight;}

@media (min-width: 400px) {
    .header, .nav, .main, .ad { font-size: 2em; }
}
@media (min-width: 700px) {
    .header, .nav, .main, .ad { font-size: 3em; }
}
```

Because the Tiny Tweaks design does not require javascript or advanced CSS3, it is a good choice for the basic version of a website when you design with progressive enhancement. Here's the output from this Tiny Tweaks web page on a viewport 800 pixels wide:



The header, navigation, main content, and ad are displayed in a column, one on top of another. The page is responsive, because the width of the column expands and contracts to fit the viewport on which the page is displayed. And font size is bigger on large screens.

Tiny Tweaks Examples

Here are some examples of sites that use the Tiny Tweaks layout:

Lycos at http://mediaqueri.es/lyc/.

Future Friendly at http://futurefriendlyweb.com/.

Neovade at http://mediaqueri.es/neo/.

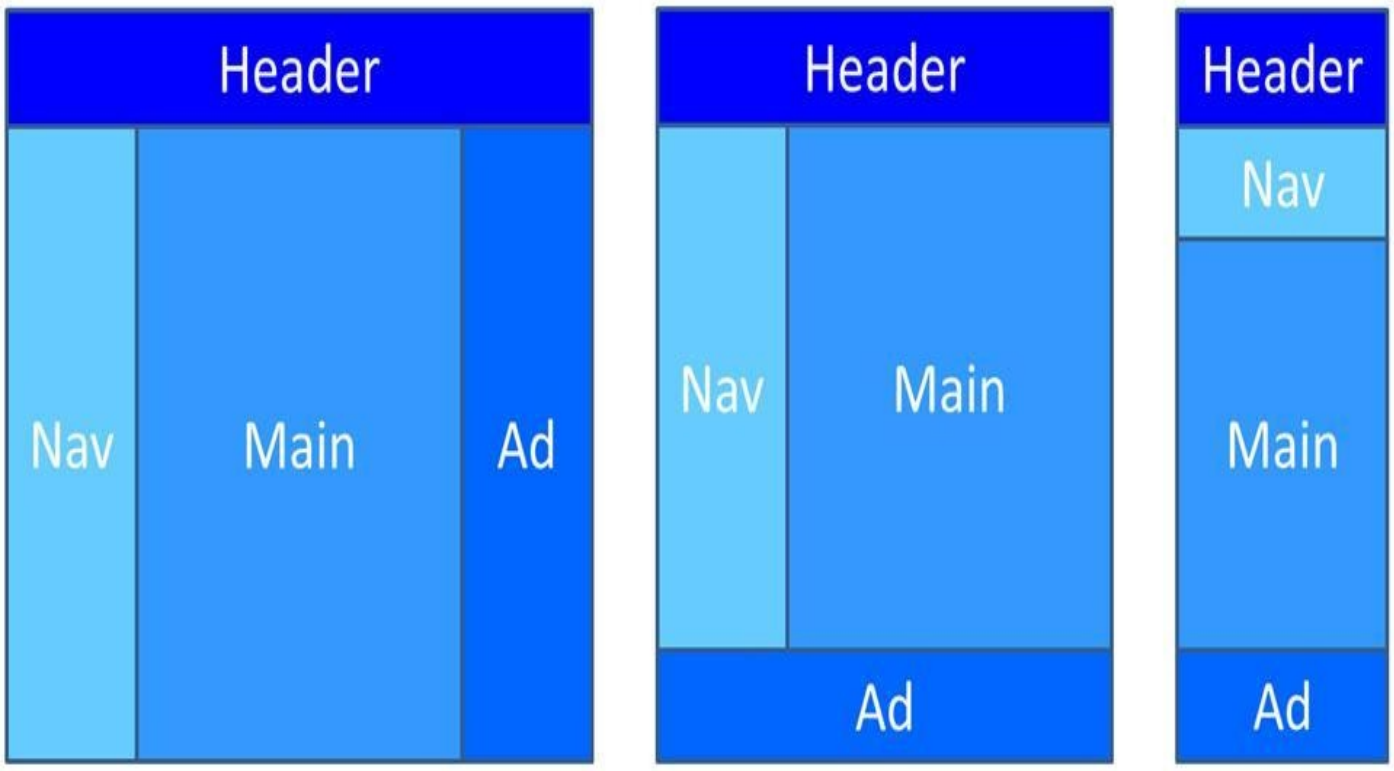
You can download source code for a Tiny Tweaks layout from http://flexbox-rwd.com/download.aspx. Look for tiny-tweaks.html in the Chapter15 directory.

Column Drop

On wide screens, the Column Drop design starts out as a multiple-column website with major web components displayed in their own columns on a single row. As the screen shrinks, the viewport becomes too narrow to hold all of the page components in a single row; so, one by one, components drop to a new row.

Ultimately, the website ends up as a single column with each page component displayed on its own row.

The figure below illustrates a typical Column Drop layout.



To implement a Column Drop design, we will use flexbox. To ensure that our website works even on browsers that don't support flexbox, we will use a progressive enhancement strategy.

To be sure that the steps required to implement a progressive enhancement strategy are clear, we'll describe those steps in some detail for this layout.

Step 1: Build a Basic Version

The first step in designing with progressive enhancement is to build a basic version of each web page. This is a version built from standard HTML and widely-accepted CSS – nothing more advanced than CSS2. This version of the web page should work on virtually any browser. It is the version we will use with browsers that don't support advanced features, like flexbox.

Luckily, we've already built a basic version of our web page - the Tiny Tweaks layout that we built in the last section. Here's the CSS code for Tiny Tweaks:

CSS

```
.header, .nav, .main, .ad { padding: 40px; width: 100%; }


.header {background-color:#007;}
.nav {background-color:#070;}
```

```
.main {background-color:#700;}
.ad {background-color: highlight;}

@media (min-width: 400px) {
    .header, .nav, .main, .ad { font-size: 2em; }
}
@media (min-width: 700px) {
    .header, .nav, .main, .ad { font-size: 3em; }
}
```

As we've seen, this code will produce a one-column website. It is the layout that will be displayed by browsers that don't support flexbox. The Tiny Tweaks layout may not do everything that the Column Drop layout does; but it can be attractive, and it will be 100% functional on any browser and any device.

Step 2: Test Browser Support

The second step in designing with progressive enhancement is to test browser support for flexbox. To conduct this test, we use Modernizr. Modernizr is a small javascript file that automatically assesses browser capabilities.

You don't have to know javascript to use Modernizr. You just have to download the file and add it to your web page. We described how to download Modernizr in Chapter 8.

Once the file is downloaded, you can link to the Modernizr file from the <head> section of your web page, as shown in the snippet below:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="modernizr-custom.min.js"></script>
    </head>
```

Now, the Modernizr file will run automatically each time your web page is loaded. If the browser supports the "flex-wrap" property, Modernizr will add a "flexwrap" class to the <html> element. If the browser supports other flexbox properties, Modernizr adds a "flexbox" class to the <html> element.

Step 3: Build an Enhanced Version

The third step in designing with progressive enhancement is to build the enhanced version of the web page. The enhanced version is the version that uses

advanced features, like flexbox. Browsers that understand flexbox display the enhanced version of the web page; less capable browsers display the basic version.

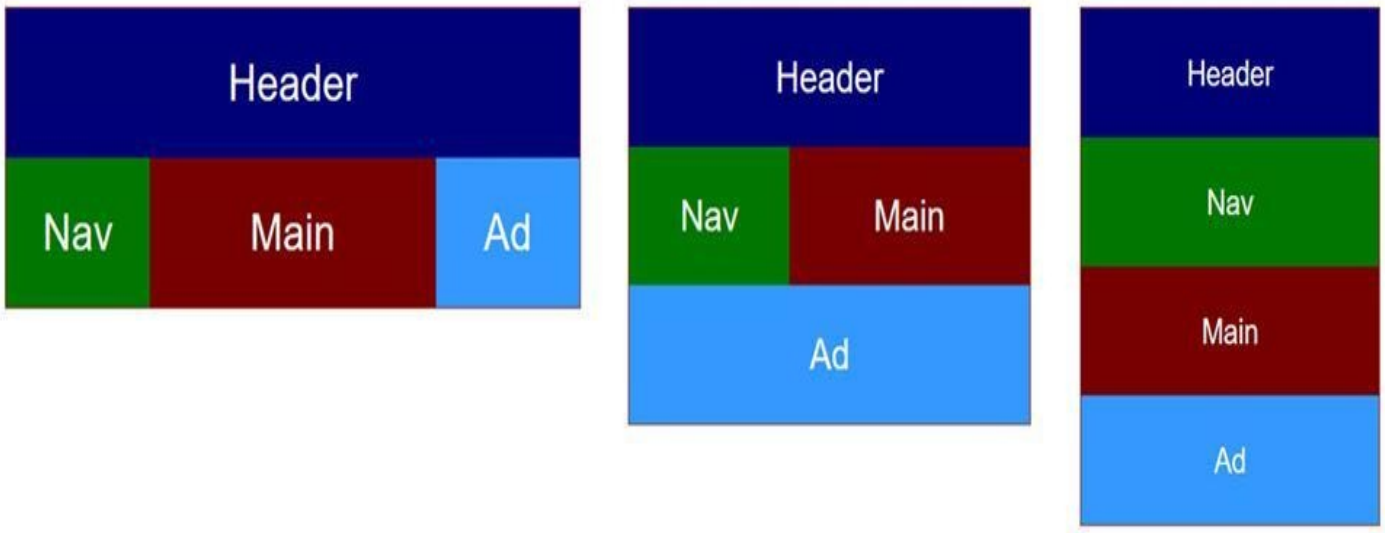
To create an enhanced version of the web page, we add this code to the Tiny Tweaks markup:

CSS

```
.flexbox.flexwrap .container {
    display: flex;
    flex-wrap: wrap; }
@media (min-width: 400px) {
    .flexbox.flexwrap .nav { width:40%;}
    .flexbox.flexwrap .main {width: 60%;}
    .flexbox.flexwrap .ad {width: 100%;}
    }
@media (min-width: 700px) {
    .flexbox.flexwrap .nav {width:25%;}
    .flexbox.flexwrap .main {width:50%;}
    .flexbox.flexwrap .ad {width: 25%;}
    }
```

In the code above, “.flexbox.flexwrap” is a selector that only exists on browsers that support flexbox; because the “flexbox” and “flexwrap” classes are added to the <html> element by Modernizr only when the browser supports flexbox. Therefore, style rules that include “.flexbox.flexwrap” are only executed by browsers that support flexbox.

In combination, the markup for Tiny Tweaks and Column Drop will deliver a working web page to all browsers and all devices. Browsers that don’t support flexbox will see the Tiny Tweaks layout, and browsers that do support flexbox will see the Column Drop layout. Here’s the Column Drop layout in action.



On large viewports (left), the navigation, main content, and ad occupy three columns in a single row under the header. On mid-sized viewports (center), there is not enough room for all three components on a single row, so the ad component drops to second row. On small viewports (right), there is only room for one component per row.

Column Drop Examples

Here are examples of sites that use the Column Drop layout:

Modernizr at http://mediaqueri.es/mod/.

Owltastic at http://mediaqueri.es/owl/.

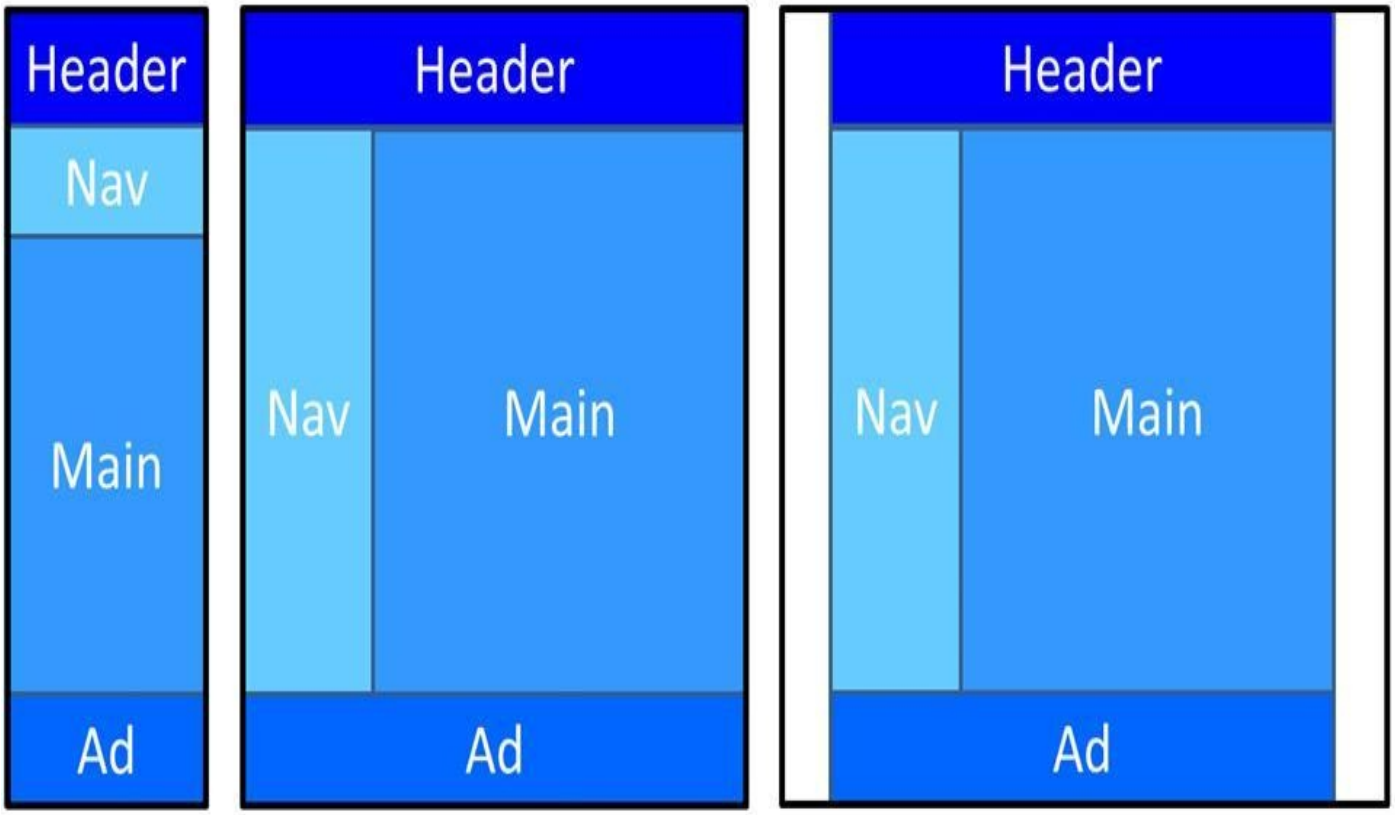
Wee Nudge at http://mediaqueri.es/wee/.

You can download source code for a Column Drop layout from http://flexbox-rwd.com/download.aspx. Look for column-drop.html in the Chapter15 directory.

Mostly Fluid

The Mostly Fluid design is another popular, easy-to-produce layout.

On small screens, page elements are stacked vertically in a column, just like the Column Drop design. On large screens, page elements are displayed in multiple columns. At a certain point, content ceases to expand; and extra space is allocated to increasingly larger margins, as shown below in the right-most image.



Mostly Fluid Markup

To implement a Mostly Fluid layout with progressive enhancement, we first create a basic version of the web page. Once again, we will use the Tiny Tweaks markup for the basic version of our Mostly Fluid layout.

The markup for the enhanced version of the Mostly Fluid layout is shown below. To create an enhanced version of the web page, we add this code to the Tiny Tweaks markup:

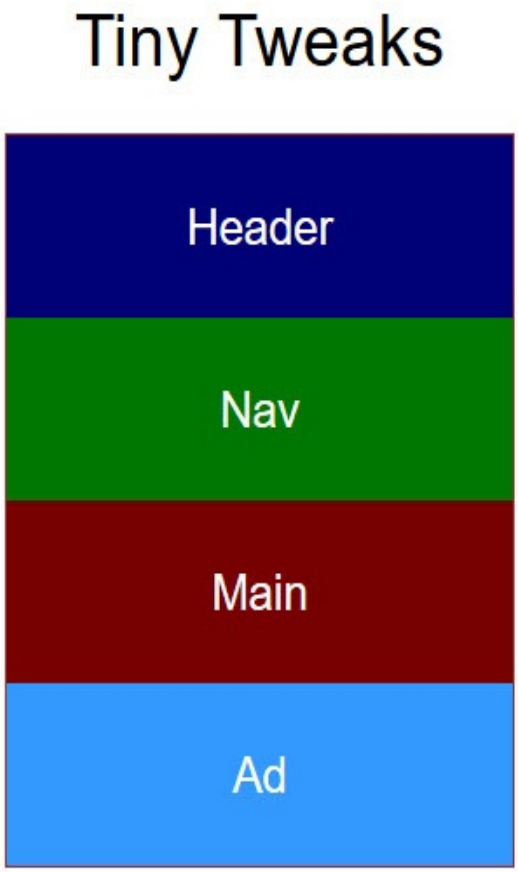
CSS

```
.flexbox.flexwrap .container {
    display: flex;
    flex-wrap: wrap; }
@media (min-width: 400px) {
    .flexbox.flexwrap .nav { width:40%;}
    .flexbox.flexwrap .main {width: 60%;}
    .flexbox.flexwrap .ad {width: 100%;}
}
@media (min-width: 700px) {
    .flexbox.flexwrap .nav {width:25%;}
    .flexbox.flexwrap .main {width:50%;}
```

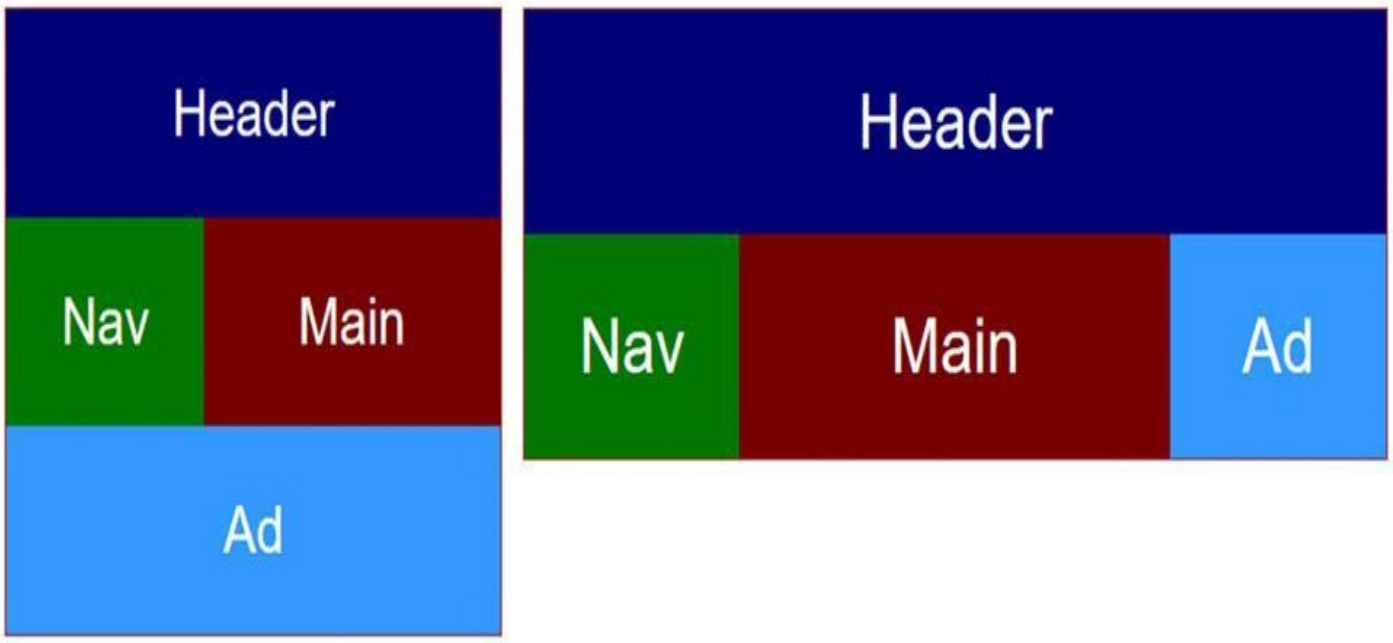
```
    .flexbox.flexwrap .ad {width: 25%;}
    .container { max-width: 900px; margin: 0 auto; }
}
```

This markup is almost identical to the markup used to produce the Column Drop layout. The main difference is a CSS declaration block added to the “container” class for large screens (greater than 700 pixels in the example). This declaration block, shown in bold, centers the content container on screens wider than 900 pixels.

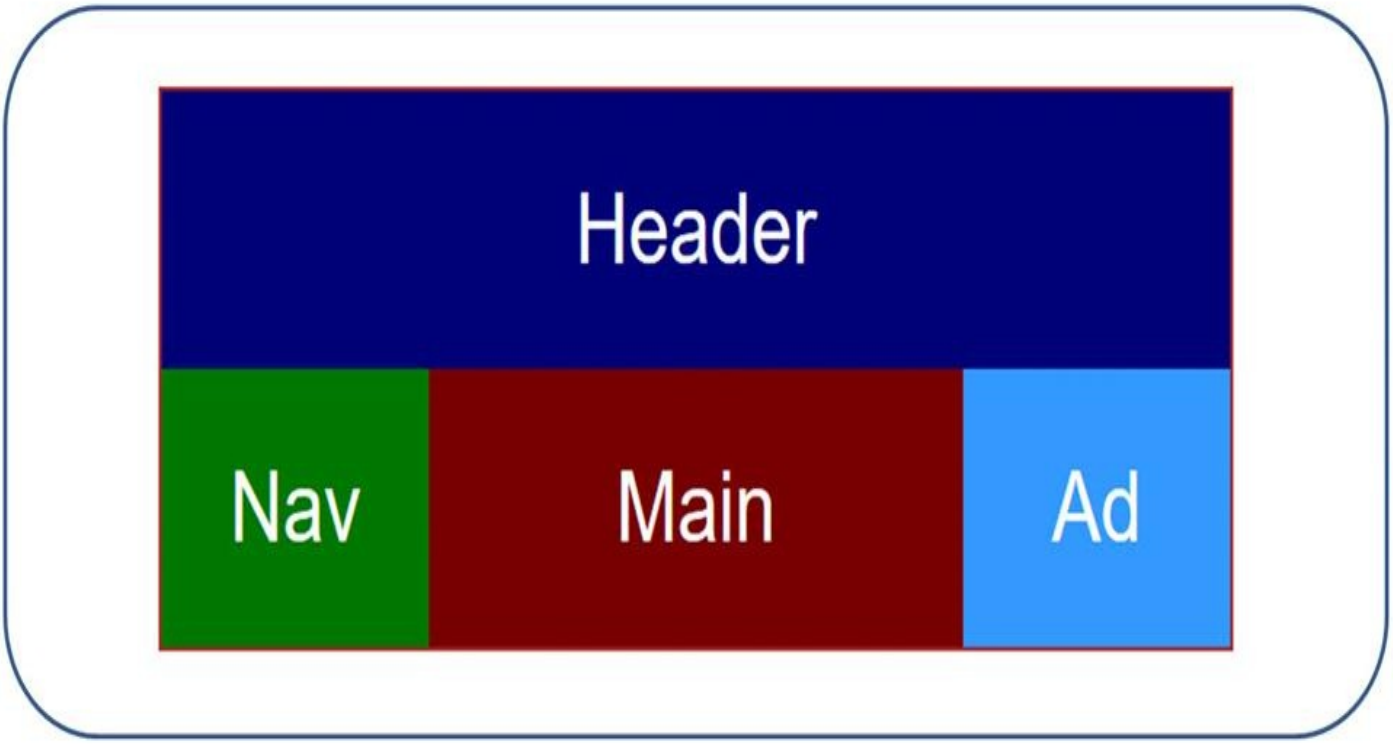
In combination, the markup for Tiny Tweaks and Mostly Fluid will deliver a working web page to all browsers and all devices. Browsers that don’t support flexbox and/or viewports smaller than 400 pixels will see the Tiny Tweaks layout, which displays each page component stacked in a single column .



On larger viewports with browsers that do support flexbox, site visitors will see the Mostly Fluid layout. On mid-sized screens, the navigation and main components will be displayed in two columns on the same row; and the ad components will be displayed in a separate row beneath (below left). On screens that are at least 700 pixels wide, the navigation, main, and advertising components will be displayed side-by-side on the same row (below right).



We designed this web page to fill the width of the viewport up to 700 pixels. Beyond that, the page will be centered in the viewport with extra space divided equally on either side. Below, we see the Mostly Fluid layout on a 900-pixel viewport.



The page is centered on the screen, and extra spaced is allocated equally on both sides.

Mostly Fluid Examples

Here are examples of sites that use the Mostly Fluid layout:

A List Apart at http://mediaqueri.es/ala/.

Sifter at http://mediaqueri.es/sif/.

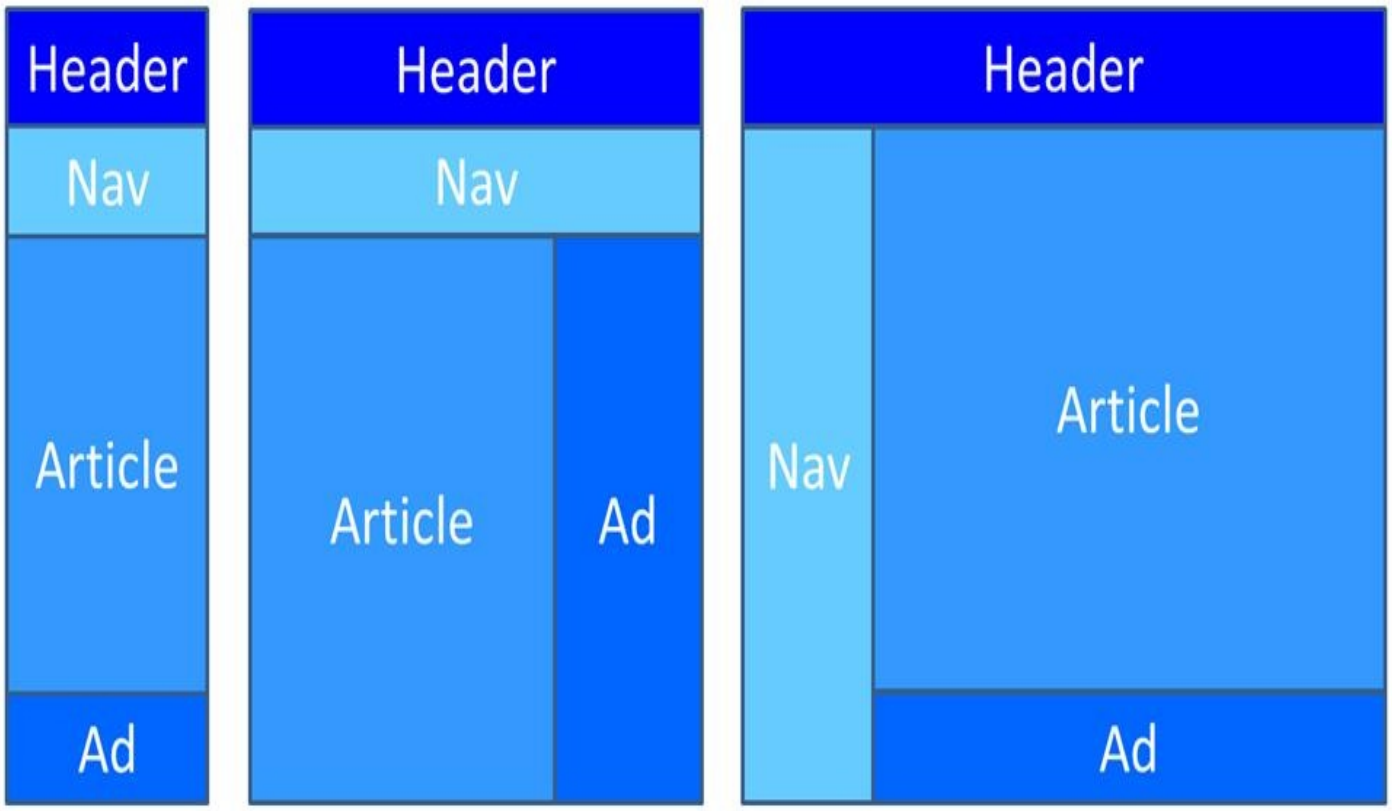
Trent Walton at http://mediaqueri.es/tre/.

You can download source code for a Mostly Fluid layout from http://flexbox-rwd.com/download.aspx. Look for mostly-fluid.html in the Chapter15 directory.

Layout Shifter

The Layout Shifter design is more complicated than the other designs that we've considered. It can involve multiple breakpoints, with very different layouts at each breakpoint.

Because of its complexity, the layout shifter design can be hard to maintain. Here's one example of a Layout Shifter design.



As you can see, the layout is very different at each screen size.

Layout Shifter Markup

To produce the layout shown in the figure above, we will make a small change to the HTML that we have been using. The HTML markup is shown below, with the new code in bold:

HTML

```
<div class="container">
    <div class="header">Header</div>
    <div class="nav">Nav</div>
```

```
    <div class = “small-container”>
        <div class=“main”>Main</div>
        <div class=“ad”>Ad</div>
    </div>
</div>
```

Since we are designing with progressive enhancement, we need a basic version of our web page. Once again, we will use the Tiny Tweaks CSS markup for the basic version. The CSS that we used with Tiny Tweaks will work perfectly with the HTML markup shown above.

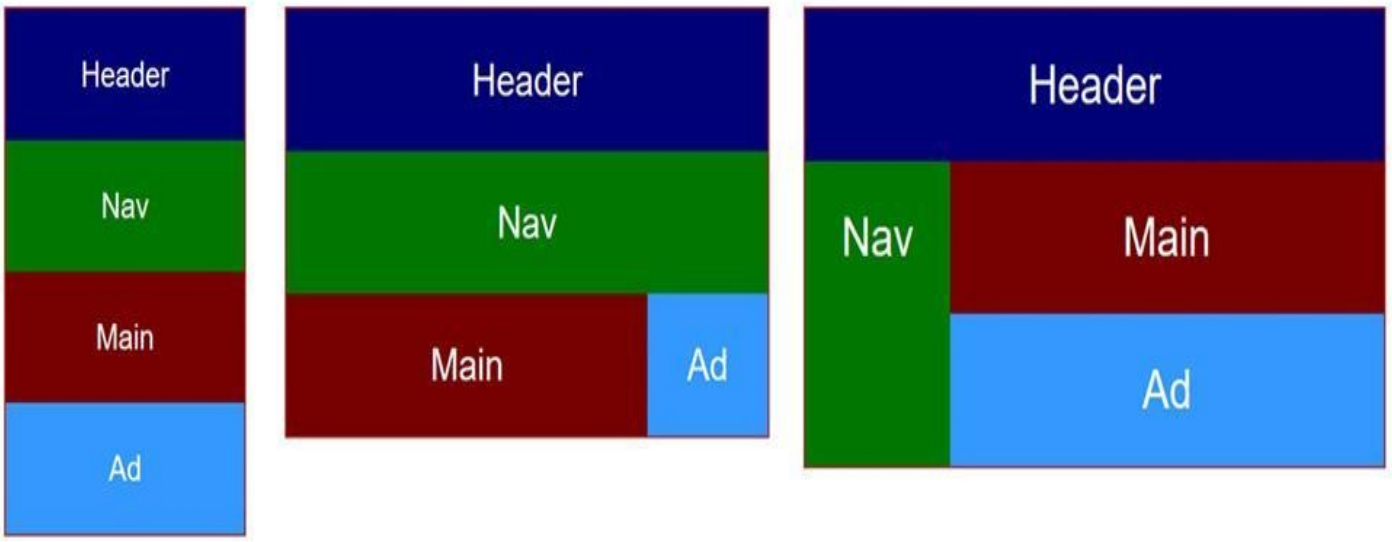
The CSS markup for the enhanced version of the Layout Shifter design is shown below. To create an enhanced version of the web page, we add this code to the Tiny Tweaks markup:

CSS

```
.small-container {width: 100%;}
.flexbox.flexwrap .container {
    display: flex;
    flex-wrap: wrap; }
@media (min-width: 400px) {
    .flexbox.flexwrap .nav { width: 100%; }
    .flexbox.flexwrap .small-container {
        width: 100%; display:flex; }
    .flexbox.flexwrap .main { width: 75%; }
    .flexbox.flexwrap .ad { width: 25%;}
}
@media (min-width: 700px) {
    .flexbox.flexwrap .nav {width:25%;}
    .flexbox.flexwrap .main {width:50%;}
    .flexbox.flexwrap .ad {width: 25%;}
    .container { max-width: 900px; margin: 0 auto; }
}
```

This markup illustrates the power of flexbox for responsive web design. With two media queries and a few lines of code in the enhanced version, we created two completely different layouts to better fit larger screen sizes. It was all very easy – no floats, no javascript, and no headaches. Here’s the result on small, medium,

and large viewports.



As usual, browsers that don't support flexbox will see the basic Tiny Tweaks layout (above left). Browsers that do support flexbox will see one of the enhanced Layout Shifter layouts (center and right).

Layout Shifter Examples

Here are examples of sites that use the Layout Shifter layout:

- Food Sense at http://foodsense.is/.
- Forefathers Group at http://mediaqueri.es/ffg/.
- Baker Street Inquirer at http://alistapart.com/d/responsive-web-design/ex/ex-site-FINAL.html.

You can download source code for a Layout Shifter design from http://flexbox-rwd.com/download.aspx. Look for layout-shifter.html in the Chapter15 directory.

Off Canvas

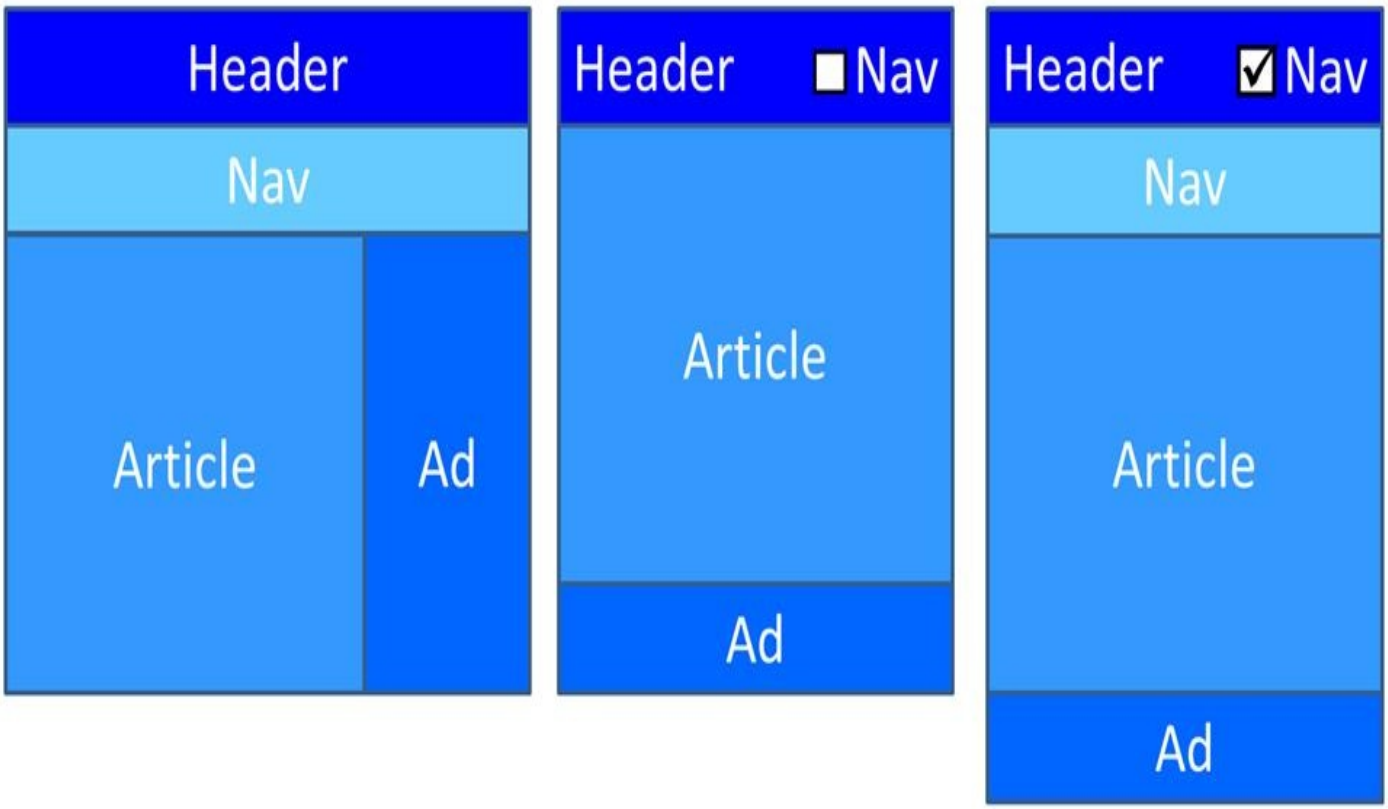
Compared to the other layouts that we discussed in this chapter, the Off Canvas layout is a different beast entirely. It addresses the problem of limited space on small screens by hiding content. The hidden content is revealed when either of the following occurs:

- A user asks to see the hidden content (e.g., by checking a box or clicking a command button).
- The viewport expands enough to accommodate the hidden content.

A common application for an Off Canvas layout is a navigation menu. On small screens, the menu is hidden until the user checks a box or clicks a button. On

large screens, the menu is visible by default. We created a navigation menu like this in Chapter 13. To demonstrate off-canvas layout, we'll build a simple version of the navigation menu from Chapter 13.

Here is a typical Off Canvas layout:



On the left, we see the web page on a wide viewport. There is plenty of room for two columns of content, plus a horizontal navigation bar. In the center, we see what happens when the viewport shrinks. Content is displayed in one column. There is no longer room for the full navigation bar; so it is hidden, and a navigation checkbox appears in the header. On the right, we see how the page changes when a user clicks the checkbox. The navigation bar appears, and other content is shifted out of the way.

Off Canvas Markup

To produce the layout shown in the figure above, we will use the following markup:

HTML

```
<div class="container">
    <div class="header-wrapper">
        <div class="header">Header</div>
        <input type="checkbox" id="toggle">
        <label for="toggle" onclick>Menu</label>
        <div class="nav">Nav</div>
```

```
    </div>
    <div class = “small-container”>
        <div class=“main”>Main</div>
        <div class=“ad”>Ad</div>
    </div>
</div>
```

Note: In the CSS presented below, we will be implementing the checkbox hack, without hiding the checkbox. (We introduced the checkbox hack in Chapter 13. It is fully described in Appendix C.)

As usual, since we are designing with progressive enhancement, we begin with a basic version of our web page. Here is the CSS code for our basic version:

CSS

```
.header-wrapper, .nav, .main, .ad {
    padding: 40px; width: 100%; font-size: 2rem; }

.header-wrapper { background-color: #007; padding: 0; }
.nav { background-color: #070; }
.main { background-color: #700; }
.ad { background-color: #39f; }

.small-container { width: 100% }

@media (max-width: 400px) {
    .header-wrapper { padding: 0; }
    .header, .nav, .main, .ad { font-size: 2rem; }
    .header { margin:1rem 0; margin-right: 3rem; display: inline; line-
    height:3rem;}

    .header-wrapper input { display: inline-block; padding: 40px 0; }
    .header-wrapper label { display: inline-block; padding: 40px 0 }

    /*Collapsible content displaces existing content */
```

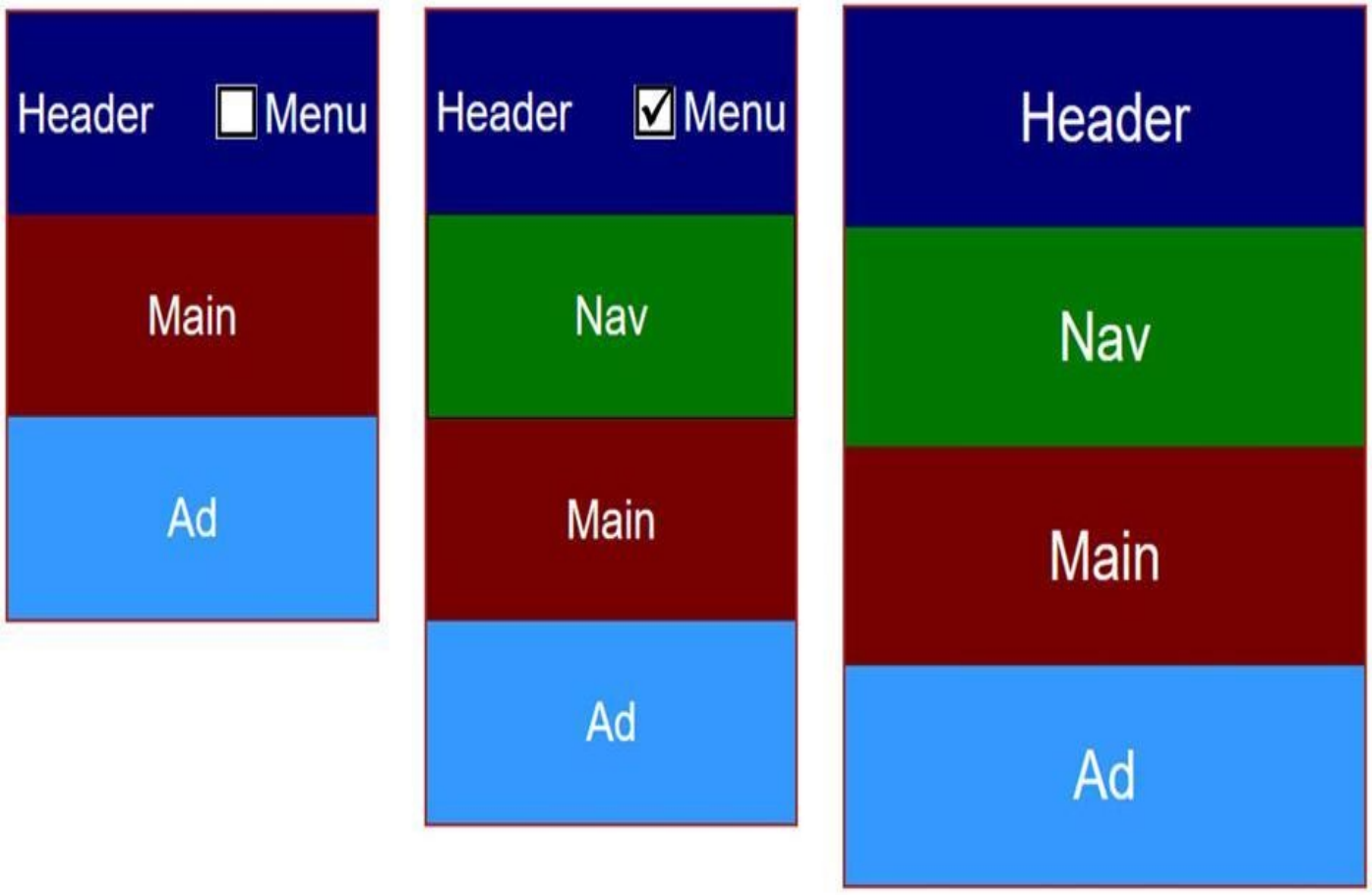
```
    div.nav { display:none; }
    input[type=checkbox]:checked ~ div.nav { display: block; border:1px solid black; }
}
@media (min-width: 401px) {
    .header-wrapper { padding:0;}
    .header, .nav, .main, .ad {
        font-size: 2.5rem;
        display: block;
    }
    .header { display:inline-block; line-height: 3rem; padding: 40px;}
    Input, label { display: none; }
}
```

Note: The lines of code shown in bold control the visibility of the navigation bar. The markup uses the general siblings selector (~) to display the navigation bar only when the checkbox is checked.

NOTE

For a good description of the general siblings selector (~), visit https://developer.mozilla.org/en-US/docs/Web/CSS/General_sibling_selectors.

Here, the basic version of the web page is a one-column layout, with hidden content. On small screens, the checkbox and navigation label are always displayed. The navigation bar is hidden when the checkbox is unchecked (below left); visible, when it is checked (center). On large screens, the checkbox and navigation label are always hidden; and the navigation bar is always displayed (below right).



If you were content with a one-column layout, this would be all you needed to do. This basic version of the web page would suffice. But if you wanted to display the main content and the ad in two columns on larger screens, you might want to create an enhanced version of the page.

We can use flexbox to create an enhanced, two-column layout. Here's the additional code that would be needed:

CSS

```
.flexbox.flexwrap .container {
    display: flex; flex-wrap: wrap; }
@media (min-width: 400px) {
    .flexbox.flexwrap .nav { width: 100%; }
    .flexbox.flexwrap .small-container {
        width: 100%;display: flex; }
    .flexbox.flexwrap .main { width: 75%; }
    .flexbox.flexwrap .ad { width: 25%; }
}
```

Added to the basic version of the web page, this markup still displays a one-column layout with hidden navigation on small screens (screens less than 400

pixels wide). But on larger screens, it displays the main information component and the ad beneath a visible navigation bar in a two-column layout, as shown below:



This coding should work just about everywhere. On small screens, all browsers will see the one-column layout with a hidden navigation menu. Users can click the checkbox to display the navigation. On large screens, the checkbox will be hidden, and the navigation will always be visible. Modern browsers will see the two-column layout. Older, less capable browsers will see the one-column layout.

Off-Canvas Examples

Here are examples of sites that use the Off-Canvas layout. These sites use a command button rather than a checkbox, but the principle is the same. To save space on small viewports, the navigation is hidden.

Kaemingk at http://www.kaemingk.com/en/.

HTML5 Rocks at http://www.html5rocks.com/en/.

Google Nexus at https://www.google.com/nexus/.

You can download source code for an Off Canvas design from http://flexbox-rwd.com/download.aspx. Look for off-canvas.html in the Chapter15 directory.

Chapter 16. Responsive Page Template

In previous chapters, we learned about flexbox commands and syntax (Chapters 1 through 6). We learned strategies for dealing with old browsers that don't support flexbox (Chapter 7 and Chapter 8). And we learned how to:

- Build responsive navigation menus (Chapter 13).
- Build responsive footers (Chapter 14).
- Produce page layouts for responsive websites (Chapter 15).

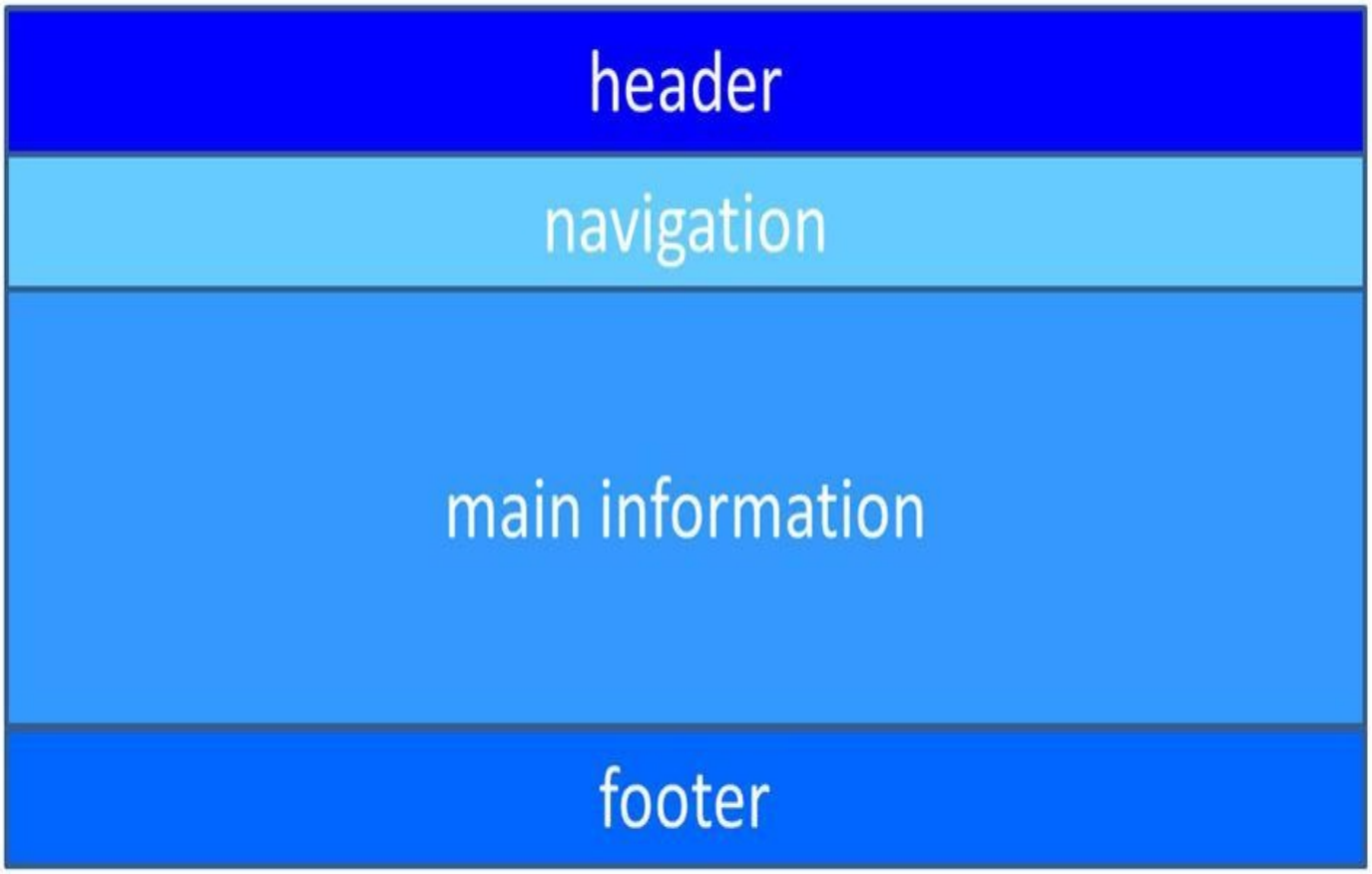
In this chapter, we'll put our new skills to use. We will build a simple template for a responsive web page. We will use only HTML and CSS to build our template; so this template will be something you can easily customize to suit your unique needs.

You can see an example of a working website that used this template at http://flexbox-rwd.com/. Each page on the site was built from the template developed in this chapter.

Note: The template that we build in this chapter won't win any design awards. But it is functional. All of the page elements are responsive. And the page should display correctly on any browser.

Template Design Plan

Our web page template will consist of four components – a header, navigation, main information, and a footer.



We already built a responsive header/navigation combination in Chapter 13, and we built a responsive footer in Chapter 14. We'll use these components without change in our template. To build a web page from this template, all we need to do is add content to the main information component.

Template Markup

In previous chapters, we developed HTML code and CSS rules for the header, navigation, and footer. We'll use exactly the same markup for our template, along with a little bit of new code for the main information component.

For convenience, we show all of the code for the template here, in one place. However, since the bulk of this markup repeats material covered in previous chapters, you may want to skip to the next section. The next section covers new HTML and CSS code for the layout of the main information component.

HTML Markup

In this section, we present all of the HTML code for our template, starting with structural markup for the header and navigation.

HTML

```
<!— Container for header and navigation —>
<div class="header-nav">
    <!— Header —>
```

```
    <header class="nav-header">Site Name</header>
    <input type="checkbox" id="nav">
    <label for="nav" id="menu" onclick>Menu</label>
    <!— Navigation —>
    <nav class="header-nav">
      <ul class="header-nav">>
        <li><a class="header-nav" href="#"> Item 1</a></li>
        <li><a class="header-nav" href="#"> Item 2</a></li>
        <li><a class="header-nav" href="#"> Item 3</a></li>
        <li><a class="header-nav" href="#"> Item 4</a></li>
        <li><a class="header-nav" href="#"> Item 5</a></li>
      </ul>
    </nav>
  </div>
```

Here is the markup for the main information section. In our template, the main information is just a placeholder for unique page content to be added later.

HTML

```
  <!—Main information —>
  <div class="main-info">
    <!—Page content—>
    <div class="row1">Row 1</div>
    <div class="row2">Row 2</div>
    <div class="row3">Row 3</div>
  </div>
```

And finally, here is the markup for the footer.

HTML

```
  <!— Footer —>
  <footer>
    <!—Navigation bar —>
    <nav class="footer-nav">
      <ul class="footer-nav">
        <li><a class="footer-nav about" href="#">Item 1</a></li>
```

```
            <li><a class="footer-nav contact" href="#">Item 2</a></li>
            <li><a class="footer-nav privacy" href="#">Item 3</a></li>
            <li><a class="footer-nav terms" href="#">Items 4</a></li>
        </ul>
    </nav>
    <!—Footer details —>
    <div class="footer-details">
        <p class="footer-details">Copyright &copy; 2016</p>
        <p class="footer-details">Powered by <a href="http://flexbox-rwd.com/">flexbox-rwd</a></p>
    </div>
</footer>
```

In this page template, most of the structural markup is used for the header, navigation, and footer. These are all page components that we developed in Chapter 13 and Chapter 14. The only new component is a container that holds content for the main information section.

CSS Markup

The bulk of the CSS markup for our template is used to style the header, navigation, and footer. This is code that we already discussed in Chapter 13 and Chapter 14, but we repeat it here for convenience. The only thing here that is new is CSS markup for the main information section.

Since the bulk of this markup repeats material covered in previous chapters, you may want to skip to the next section. The next section covers the new CSS code developed for the main information component.

To begin, here are style rules to overcome some problems with the advanced checkbox hack:

CSS

```
body { webkit-animation: bugfix infinite 1s; }
@-webkit-keyframes bugfix {
    from {padding:0;}
    to {padding:0;}
}
```

Some of the CSS markup is common to all versions of the navigation menu. Here are those style rules:

CSS

```
/* All screens: Common styles */
a.nav-header { text-decoration: none; }
a.nav-header:hover { color: darkblue;}
input#nav { display: none; }
nav.header-nav { background-color: blue; width: 100%; text-align: center;}
ul.header-nav { list-style-type: none; padding: 0; margin: 0 auto; }
a.header-nav { text-decoration: none; display: block; color: white; }
```

Here's the CSS for the basic version of the vertical navigation menu:

CSS

```
/* Small and medium screens */
@media (max-width: 850px) {
    /* Basic */
    header.nav-header { text-align: center; }
    a.nav-header { font-size: 2.3rem; }
    label#menu { display: none; }
    nav.header-nav { display: block; }
    ul.header-nav { background-color: blue; }
    a.header-nav { font-size: 1.1rem; line-height: 2.333333; border-top: 1px solid
    rgba(255, 255, 255, 0.3); border-bottom: 1px solid rgba(0, 0, 0, 0.1);}
    a.header-nav:hover { background-color: deepskyblue; }
}
```

Here's the CSS for the basic version of the horizontal navigation menu:

CSS

```
/* Large screen: Basic */
@media (min-width: 851px) {
    div.header-nav { display: block; background-color: white; }
    header.nav-header { display: block; padding: 1rem 0; width: 100%;
    background-color: white; color: blue; text-align: center;}
    a.nav-header { font-size: 2.8rem; color: blue; background-color: white;}
    label#menu { display:none; }
    nav.header-nav { display: block; height: 3.5rem;  }
```

```
    ul.header-nav { background-color: blue; max-width: 1300px; }
    ul.header-nav li { display: inline-block; }
    a.header-nav { font-size: 1.4rem; line-height: 2.9166667; line-height: 2.5; padding: 0 1rem; }
    a.header-nav:hover { background-color: deepskyblue; }
}
```

Here are CSS style rules for the enhanced version of the vertical navigation menu. These rules apply to both small and mid-size screens:

CSS

```
/* Small and mid-size screens */
@media (max-width: 850px) {
    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}

    /* Checkbox hack */
    .flexbox.flexwrap nav.header-nav { display: none; }
    .flexbox.flexwrap input#nav:checked ~ nav.header-nav  { display: block; }

    /* Ordinary style rules */
    .flexbox.flexwrap a.nav-header{ color: white; font-size: 1.4rem; line-height: 2.3333333; }
    .flexbox.flexwrap header.nav-header { width: auto; background-color: blue; color: white; line-height: 2.333333; text-align:left; padding-left: 1.5rem; }
    .flexbox.flexwrap label#menu { display: block; width: auto; background-color: blue; color: white; font-size: 1.4rem; font-weight: bold; line-height: 2.333333; text-align: center; cursor: pointer; }
    .flexbox.flexwrap label#menu:hover { background-color: darkblue; }
    .flexbox.flexwrap ul.header-nav { background-color: deepskyblue; }
    .flexbox.flexwrap a.header-nav:hover { background-color: darkblue; }
    .flexbox.flexwrap input#nav:checked ~ label#menu:hover { background-color: darkblue; }
    }
```

```
}
```

Here are additional CSS rules for the enhanced version of the vertical navigation menu. These are additional rules for mid-size screens:

CSS

```
/* Mid-size screen: */
@media (min-width: 501px) and (max-width: 850px) {
    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}
    .flexbox.flexwrap label#menu { flex: 0 0 140px; font-weight: normal; }

    /* Generated label content*/
    .flexbox.flexwrap label#menu:before { content: “Menu \25BC”; } /* unicode down arrow */
    .flexbox.flexwrap input#nav:checked ~ label#menu:before { content: “Menu \25B2”; } /* unicode up arrow */
    }
}
```

Here are additional CSS rules for the enhanced version of the vertical navigation menu. These are additional rules for small screens:

CSS

```
/* Small screen: Enhanced */
@media (max-width: 500px) {

    /* Flexbox properties */
    .flexbox.flexwrap div.header-nav { display: flex; flex-wrap: wrap; }
    .flexbox.flexwrap header.nav-header { flex: 1 1 auto;}
    .flexbox.flexwrap label#menu { flex: 0 0 90px; }

    /* Generated label content*/
    .flexbox.flexwrap label#menu:before { content: “\2630”; } /* hamburger icon */
    .flexbox.flexwrap input#nav:checked ~ label#menu:before { content: “X”; } /* x icon */
```

```
    }
}
```

And finally, here's the CSS for an enhanced version of the horizontal navigation menu:

CSS

```
/* Large screen: Enhanced */
@media (min-width: 851px) {
    /* Menu items equally spaced */
    .flexbox.flexwrap ul.header-nav { display:flex; justify-content: space-around; }
}
```

Now, let's set some style rules for the main information section

CSS

```
/* Large screen: Enhanced */
div.main-info { width: 100%; }
div.row1, div.row2, div.row3 {
    margin: 1rem auto; width: 100%; max-width: 980px; color: white;
    padding:2rem; text-align: center; font-size:  1.5rem;
}
div.row1 { background-color: red; }
div.row2 { background-color: green; }
div.row3 { background-color: blue; }
```

And finally, let's look at style rules for the footer.

CSS

```
/* Common basic styles */
footer { background-color: white; }
nav.footer-nav { background-color: blue; width: 100%; text-align: center;}
ul.footer-nav { list-style: none; background-color: blue; margin:0 auto; padding:
0; max-width: 1200px; }
.footer-nav li { display: inline-block; }
a.footer-nav { display: inline-block; text-decoration: none; color: white; line-
height: 3; }
a.footer-nav:hover { background-color: deepskyblue; }
```

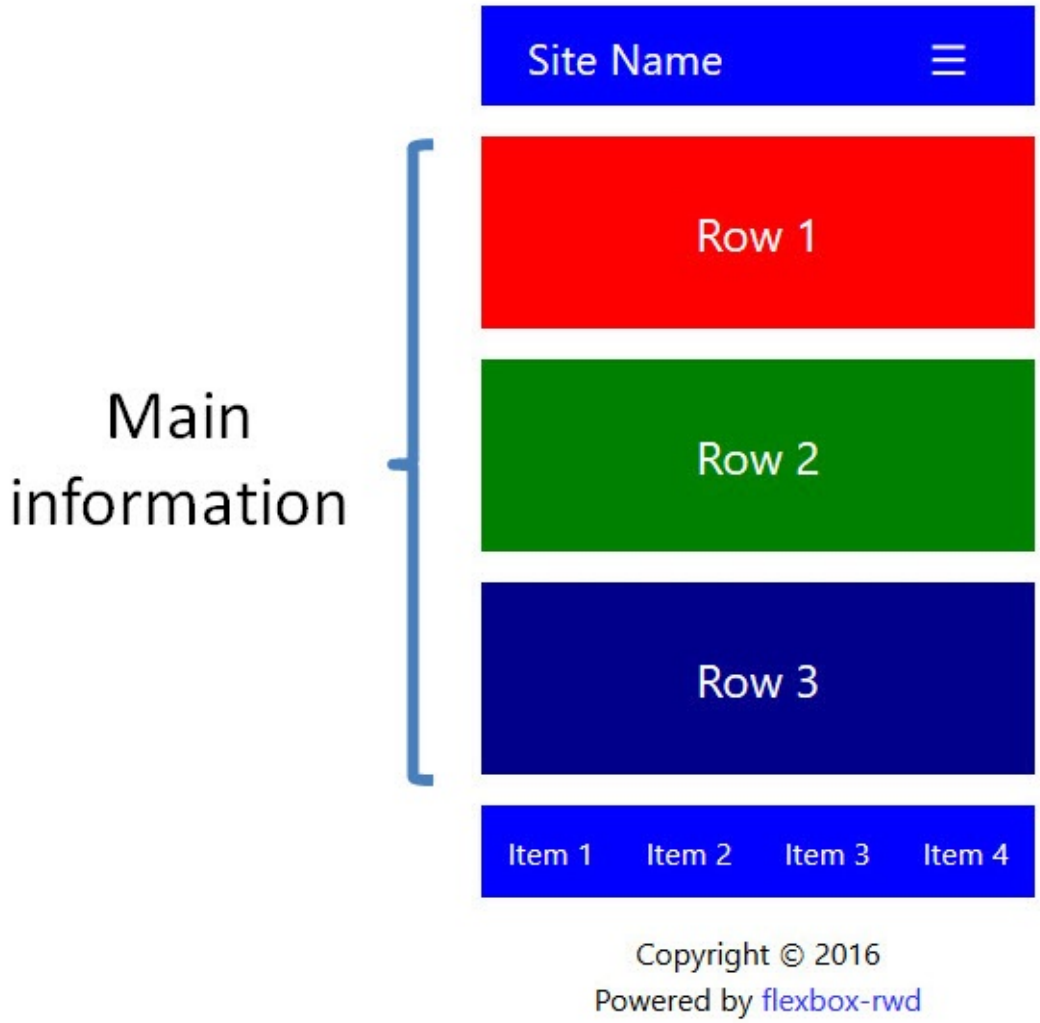
```
div.footer-details { max-width: 1200px; margin: 1rem auto; text-align:center; }

/* Small screens */
@media (max-width: 500px) {
    a.footer-nav { font-size: 1rem; padding: 0 .5rem;}
}

/* Medium screens */
@media (min-width: 501px) and (max-width: 850px) {
    a.footer-nav { font-size: 1.1rem; padding: 0 1.5rem;}
}

/* Large screens */
@media (min-width: 851px) {
    a.footer-nav { font-size: 1.2rem; padding: 0 2rem;}
}
/* Enhanced */
.flexbox.flexwrap ul.footer-nav { display: flex; justify-content: space-around; }
```

Note: You can download all of the source code for examples in this book at http://flexbox-rwd.com/download.aspx. To get source code for the web page template, look for template.html in the Chapter 16 directory.

Template Layout

The template that we have built consists of a header and navigation at the top of the page, and a footer at the bottom. In between, we have the main information section, which consists of three rows of content.

Here’s what the template looks like on a screen 320 pixels wide.



If you display the template on a wider viewport, the page reconfigures itself to fill the space available. Here's the same web page on a screen 1200 pixels wide.



The template is responsive because each of its page components is responsive. In Chapter 13, we built the header and navigation component to be responsive. In Chapter 14, we built the footer to be responsive. In this chapter, we built the main information component from three elements. Here's the HTML.

HTML

```
<!—Main information —>
<div class="main-info">
    <!—Page content—>
    <div class="row1">Row 1</div>
    <div class="row2">Row 2</div>
    <div class="row3">Row 3</div>
</div>
```

The elements used to create Rows 1, 2, and 3 are flexible by default; they shrink and expand to fill their container. Therefore, each of the rows is responsive.

Because this template is responsive, web pages that you create from this template can also be responsive. There is only one requirement. Any new content that you add to the main information section must be responsive. Luckily, we showed how to create responsive layouts in Chapter 15.

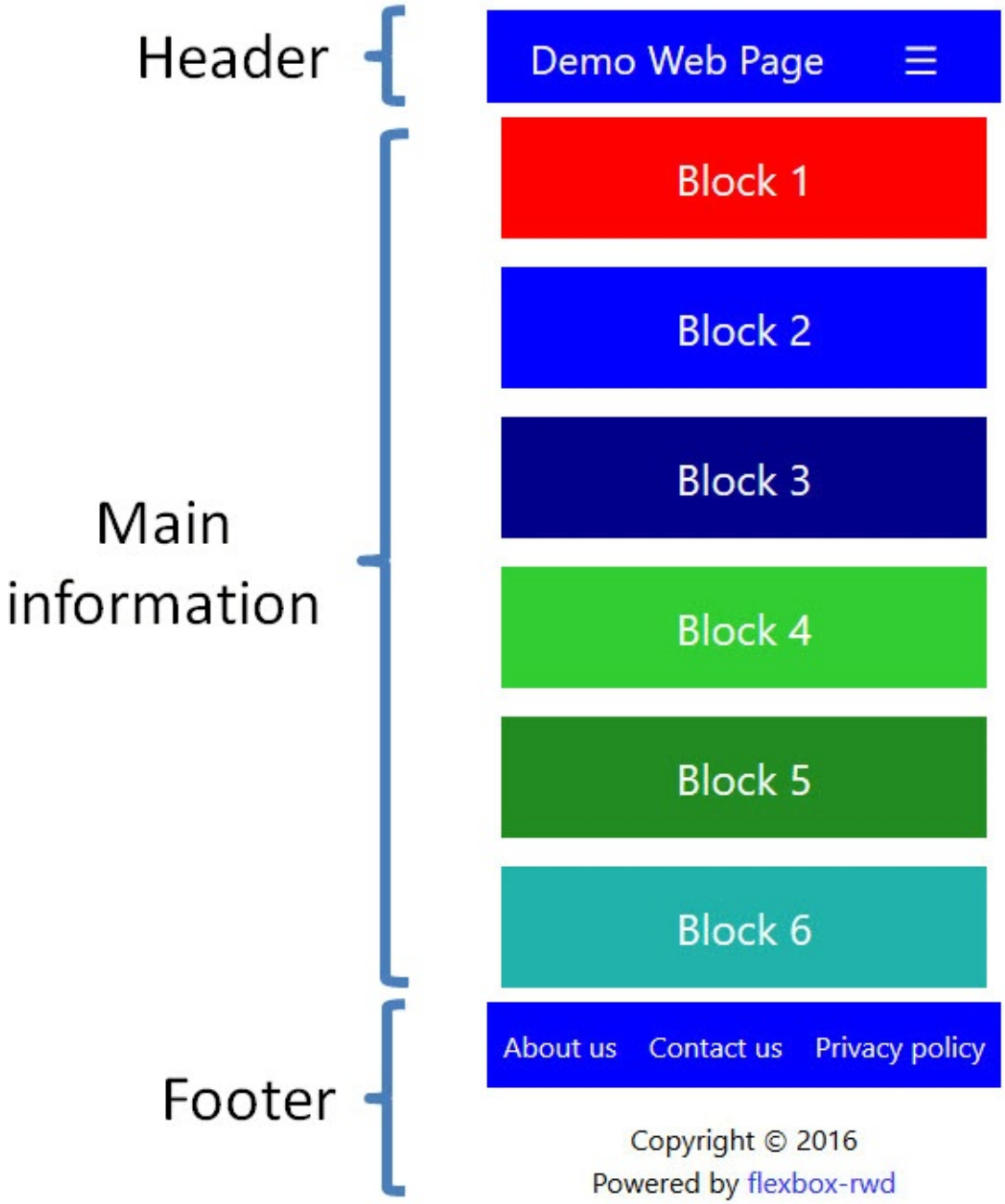
How to Build a Web Page

So, how do you build a new web page, using the template that we created? There are four steps:

- Edit the header.
- Edit the navigation bar.
- Edit the footer.
- Add responsive content to the main information component.

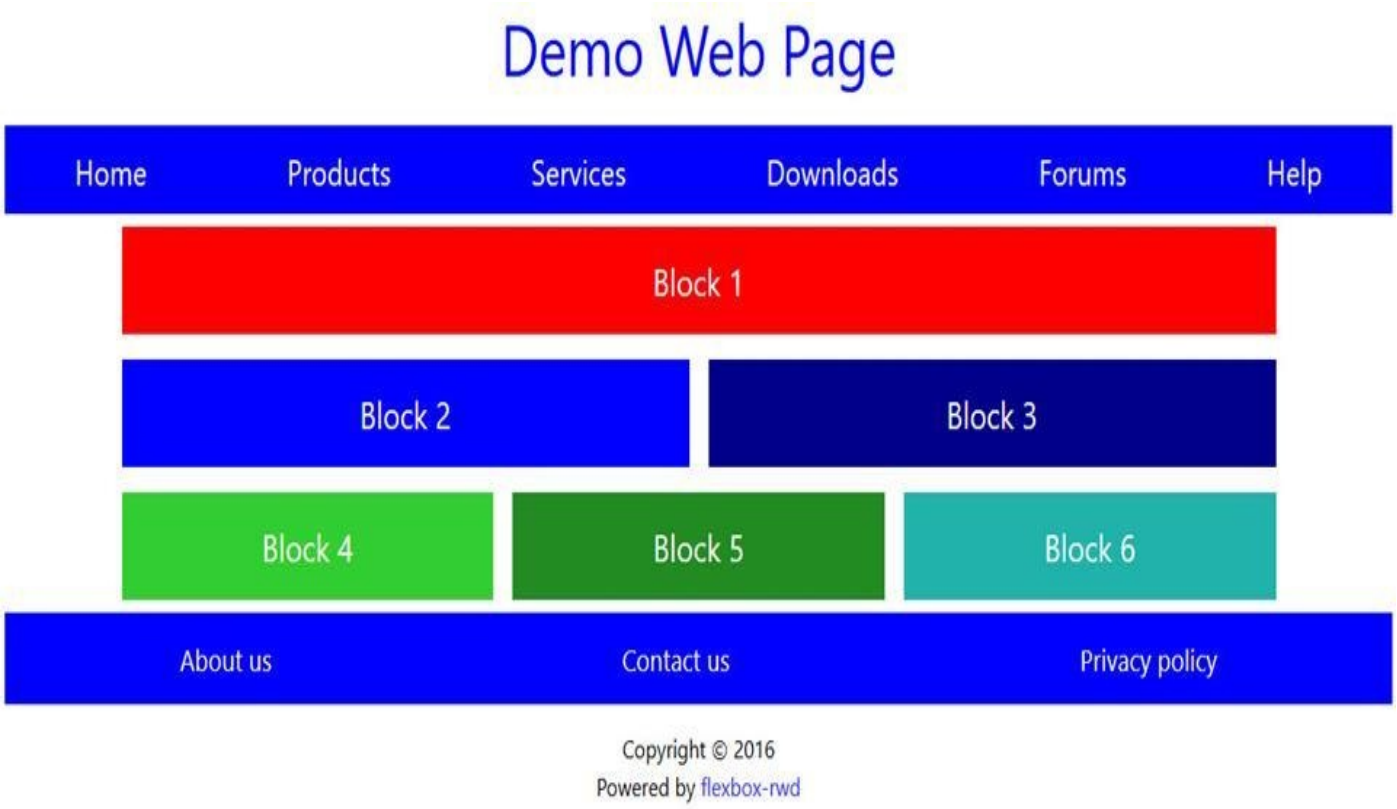
The first three steps are simple edits to the original template. They result in a new template that can be used for pages on the new website. Once you have the new template, you can create new web pages simply by adding content to the main information component. Each web page on the new website will have unique content in the main information component.

Enough talk. Let’s illustrate the process by building a new web page. Here is how the page we will build looks on a small screen.



At the top of the page, we have a header and hidden navigation. At the bottom, we have a footer. In between, we have the main information component, which consists of six blocks of content stacked in a single column.

Since this page is responsive, it should reconfigure itself for display on wider screens. Here is how we want the page to look on a large viewport.



Notice how the page layout on the small screen differs from the page layout on the large screen. On the large screen, the header is centered at the top of the page. The navigation bar is displayed, not hidden. Blocks of content in the main information component are arranged in rows, not stacked in a single column. And each row has a different number of columns. Row 1 has one column; Row 2, two columns; and Row 3, three columns.

Now let's walk through the steps necessary to turn our original template into this new, responsive web page.

Header and Navigation

We begin with the header and navigation. Here is the HTML code for the header and navigation in our template.

HTML

```
<!— Container for header and navigation —>
<div class="header-nav">
  <!— Header —>
  <header class="nav-header">Site Name</header>
  <input type="checkbox" id="nav">
  <label for="nav" id="menu" onclick>Menu</label>
  <!— Navigation —>
  <nav class="header-nav">
```

```
        <ul class="header-nav">>
            <li><a class="header-nav" href="#">Item 1</a></li>
            <li><a class="header-nav" href="#">Item 2</a></li>
            <li><a class="header-nav" href="#">Item 3</a></li>
            <li><a class="header-nav" href="#">Item 4</a></li>
            <li><a class="header-nav" href="#">Item 5</a></li>
        </ul>
    </nav>
</div>
```

To create the header and navigation for our new web page, we only need to make a few changes. First, we need to change the text within the <header> element. In the template, the <header> text is:

```
<header class="nav-header">Site Name</header>
```

We will change that to :

```
<header class="nav-header">Demo Web Page</header>
```

The new text is shown in bold. In addition, we need to edit the unordered list in the navigation section. This list in the template has five menu items. We want six menu items in the new web page. Here's the navigation section with a six menu items. The new item is shown in bold:

```
 <nav class="header-nav">
    <ul class="header-nav">>
        <li><a class="header-nav" href="#"> Item 1</a></li>
        <li><a class="header-nav" href="#">Item 2</a></li>
        <li><a class="header-nav" href="#">Item 3</a></li>
        <li><a class="header-nav" href="#"> tem 4</a></li>
        <li><a class="header-nav" href="#">Item 5</a></li>
        <li><a class="header-nav" href="#">Item 6</a></li>
    </ul>
</nav>
```

In addition, we want to add new text for each menu item. Here's the navigation component with new text for the menu labels shown in bold:

```
 <nav class="header-nav">
    <ul class="header-nav">>
```

```
        <li><a class="header-nav" href="#">Home</a></li>
        <li><a class="header-nav" href="#">Products</a></li>
        <li><a class="header-nav" href="#">Services</a></li>
        <li><a class="header-nav" href="#">Downloads</a></li>
        <li><a class="header-nav" href="#">Forums</a></li>
        <li><a class="header-nav" href="#">Help</a></li>
    </ul>
</nav>
```

That was pretty painless. We started with this on our template.



And after a few quick edits, we ended up with this on our new web page.



Basically, we changed the site name, we added an additional menu item, and we wrote new text for each menu item. We didn't worry about spacing for the new menu item. Flexbox handled spacing for us, behind the scenes.

Footer

To build a new footer for our new web page, our starting point is the navigation element for the template footer. Here's the HTML code from the template:

HTML

```
<nav class="footer-nav">
    <ul class="footer-nav">
        <li><a class="footer-nav" href="#">Item 1</a></li>
        <li><a class="footer-nav" href="#">Item 2</a></li>
        <li><a class="footer-nav" href="#">Item 3</a></li>
        <li><a class="footer-nav" href="#">Items 4</a></li>
```

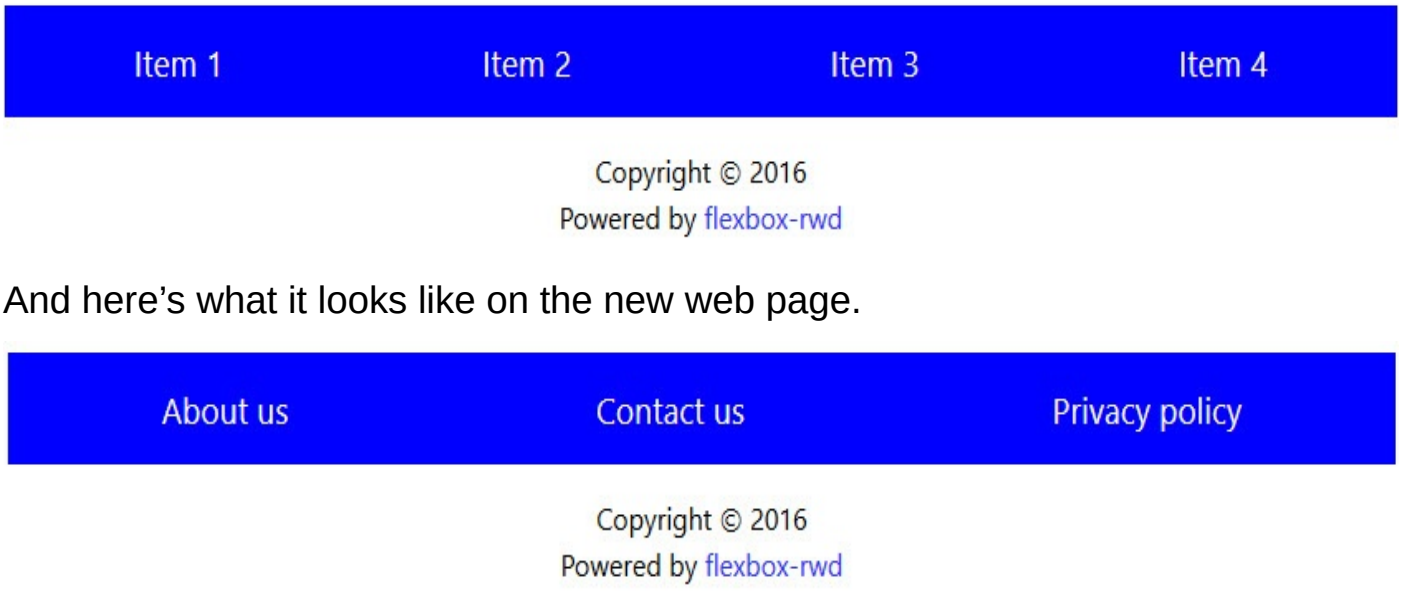
```
    </ul>
  </nav>
```

In the template, the navigation bar for the footer has four menu items. Adding or subtracting menu items is just a matter of pasting or cutting. For our new web page, let's create a navigation bar with three items: "About us", "Contact us", and "Privacy Policy". Here's the new code, with changes shown in bold:

HTML

```
<nav class="footer-nav">
    <ul class="footer-nav">
        <li><a class="footer-nav" href="#">About us</a></li>
        <li><a class="footer-nav" href="#">Contact us</a></li>
        <li><a class="footer-nav" href="#">Privacy policy</a></li>
    </ul>
</nav>
```

Here's what the footer looked like on the template.



And here's what it looks like on the new web page.

Building a new footer from the template footer required just a few edits to the template. We deleted one menu item from the template. And we changed the descriptive text for the three remaining items. Once again, the positioning of menu items was handled automatically by flexbox.

Main Information

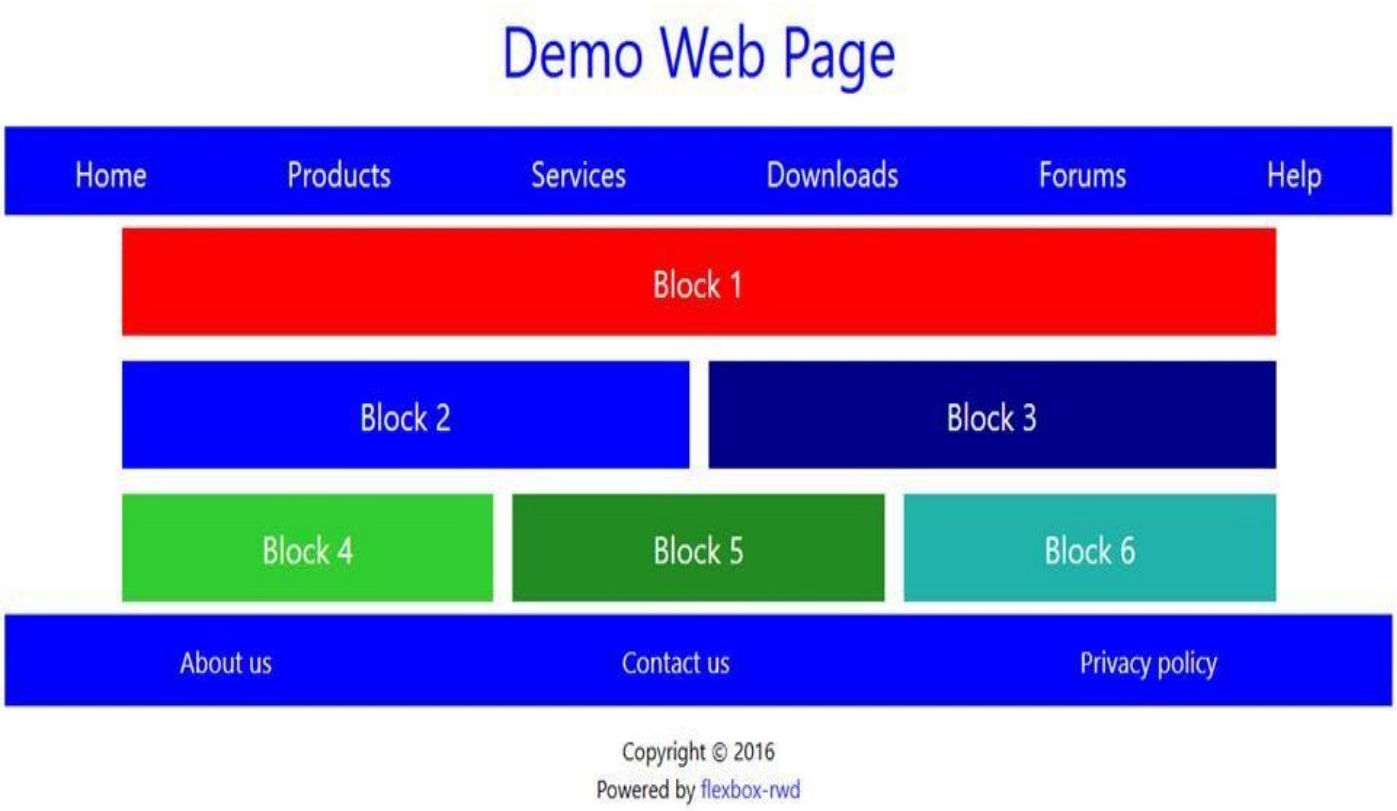
At last, we've come to the main information component. Here is the structural code for the main information component in our template.

HTML

```
<!—Main information —>
<div class=”main-info”>
    <!—Page content—>
    <div class=”row1”>Row 1</div>
    <div class=”row2”>Row 2</div>
    <div class=”row3”>Row 3</div>
</div>
```

In the template, the main information component is a wrapper that holds three virtually empty rows of content. It is up to you to fill those rows with new content. You can delete one or more rows if it is not needed. Or you can add additional rows as required.

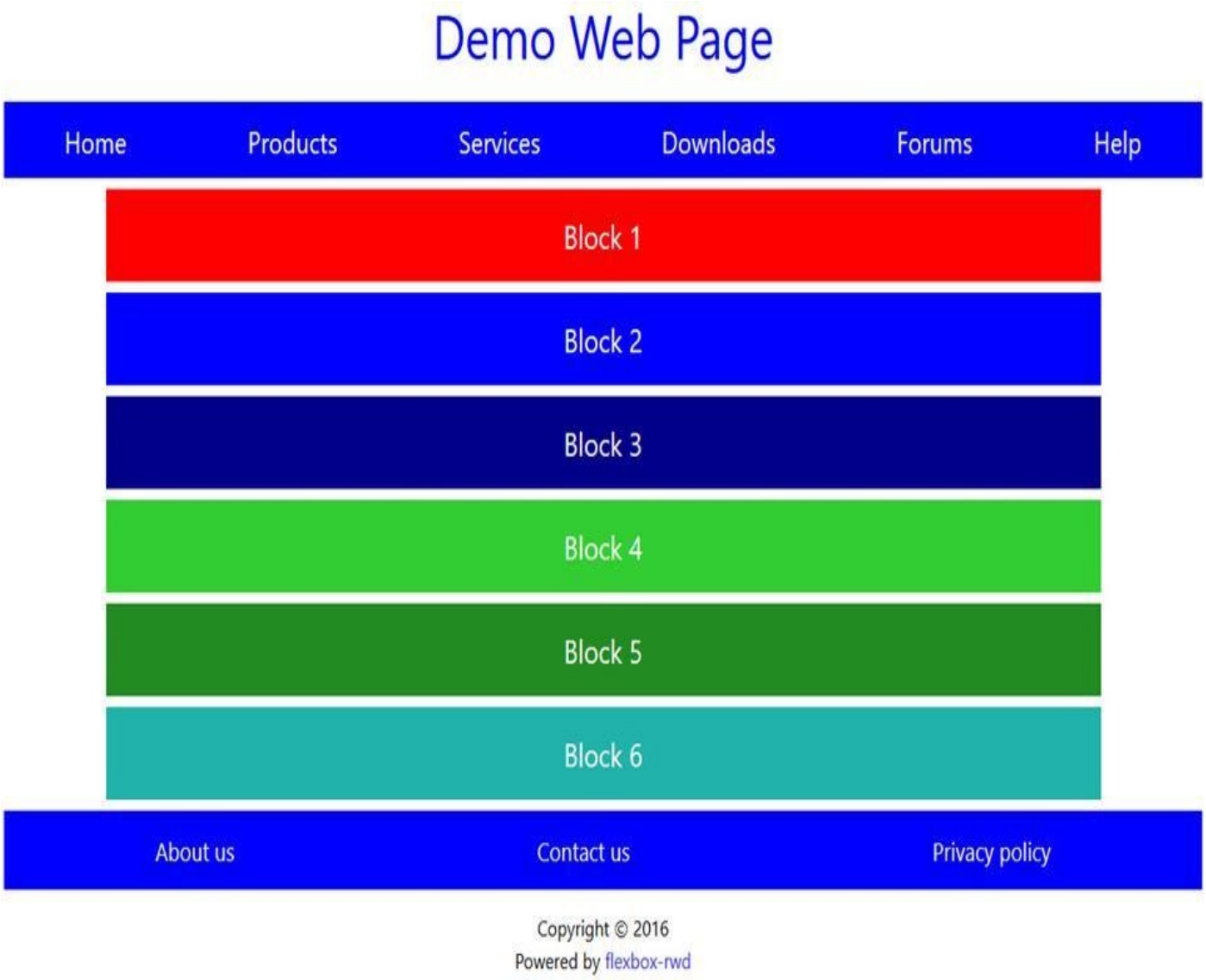
For this example, we’ll use all three rows from the original template – none added and none deleted. This is how we want our new web page to look when it is displayed on a large screen in a browser that understands flexbox.



The main information component is positioned between the site navigation and the footer. The first row contains one block of content; the second row, two blocks of content; and the third row, three blocks of content.

Each block of content has a minimum width of 300 pixels. Except for that constraint, we want each block of content to be flexible – to shrink and expand as the viewport changes size. And finally, if the viewport becomes too small to hold multiple blocks on the same line, we want one block to wrap to a new line.

In browsers that don’t understand flexbox, we want to stack each block of content in a single column, like this:



Now that our objectives are clear, we are ready to build the main information component of our new web page. Here’s the HTML.

HTML

```
<!—Main information —>
<div class=”main-info”>
    <!—Page content—>
    <div class=“row1”>
        <div class=“block1”>Block 1</div>
    </div>
    <div class=“row2”>
        <div class=“block2”>Block 2</div>
        <div class=“block3”>Block 3</div>
```

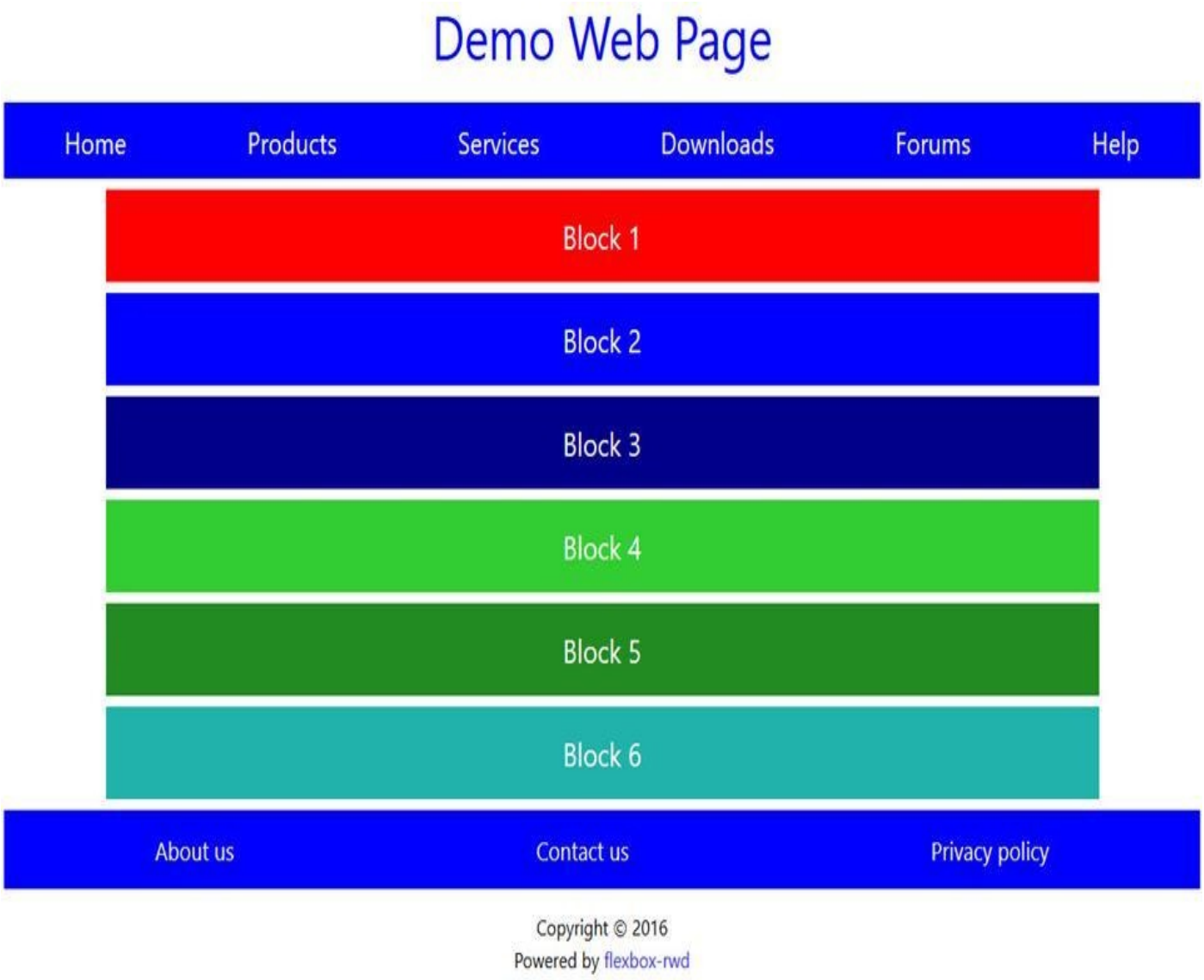
```
    </div>
    <div class="row3">
        <div class="block4">Block 4</div>
        <div class="block5">Block 5</div>
        <div class="block6">Block 6</div>
    </div>
</div>
```

Now, let's write some rules to style the main information section. We'll start with style rules for the basic version of the main information component. This is the version that will be displayed on old browsers that don't understand flexbox.

CSS

```
/* Main info: Basic version */
div.main-info { width: 100%; }

div.row1, div.row2, div.row3 {
    margin: 0rem auto; max-width: 980px; padding:0;
}

div.block1, div.block2, div.block3, div.block4, div.block5, div.block6 {
    margin: .5rem; padding:1rem; text-align: center; font-size: 1.5rem; color: white;
}

div.block1 { background-color: red; }
div.block2 { background-color: blue; }
div.block3 { background-color: darkblue; }
div.block4 { background-color: limegreen; }
div.block5 { background-color: forestgreen; }
div.block6 { background-color: lightseagreen; }
```

This code sets styles for various properties – colors, font size, widths, etc. With browsers that don't support flexbox, it produces a main information component with blocks of content stacked in a single column, like this:



We add the following CSS code to produce an enhanced version of the main information component.

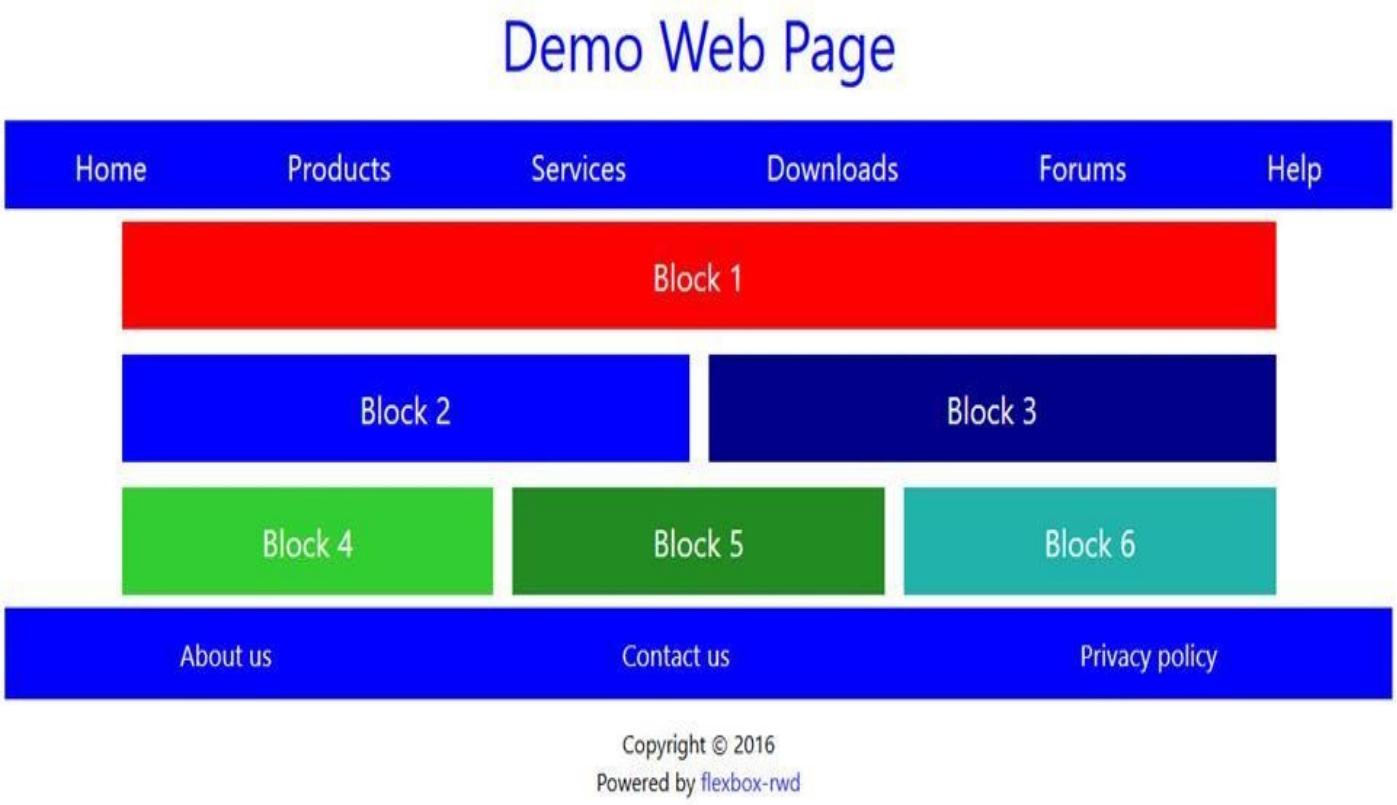
CSS

```
/* Main info: Enhanced version */
.flexbox.flexwrap div.row2,
.flexbox.flexwrap div.row3 {
    display:flex; flex-wrap:wrap;
}
.flexbox.flexwrap div.block2,
.flexbox.flexwrap div.block3,
.flexbox.flexwrap div.block4,
.flexbox.flexwrap div.block5,
.flexbox.flexwrap div.block6 {
    flex:1 0 300px;}
```

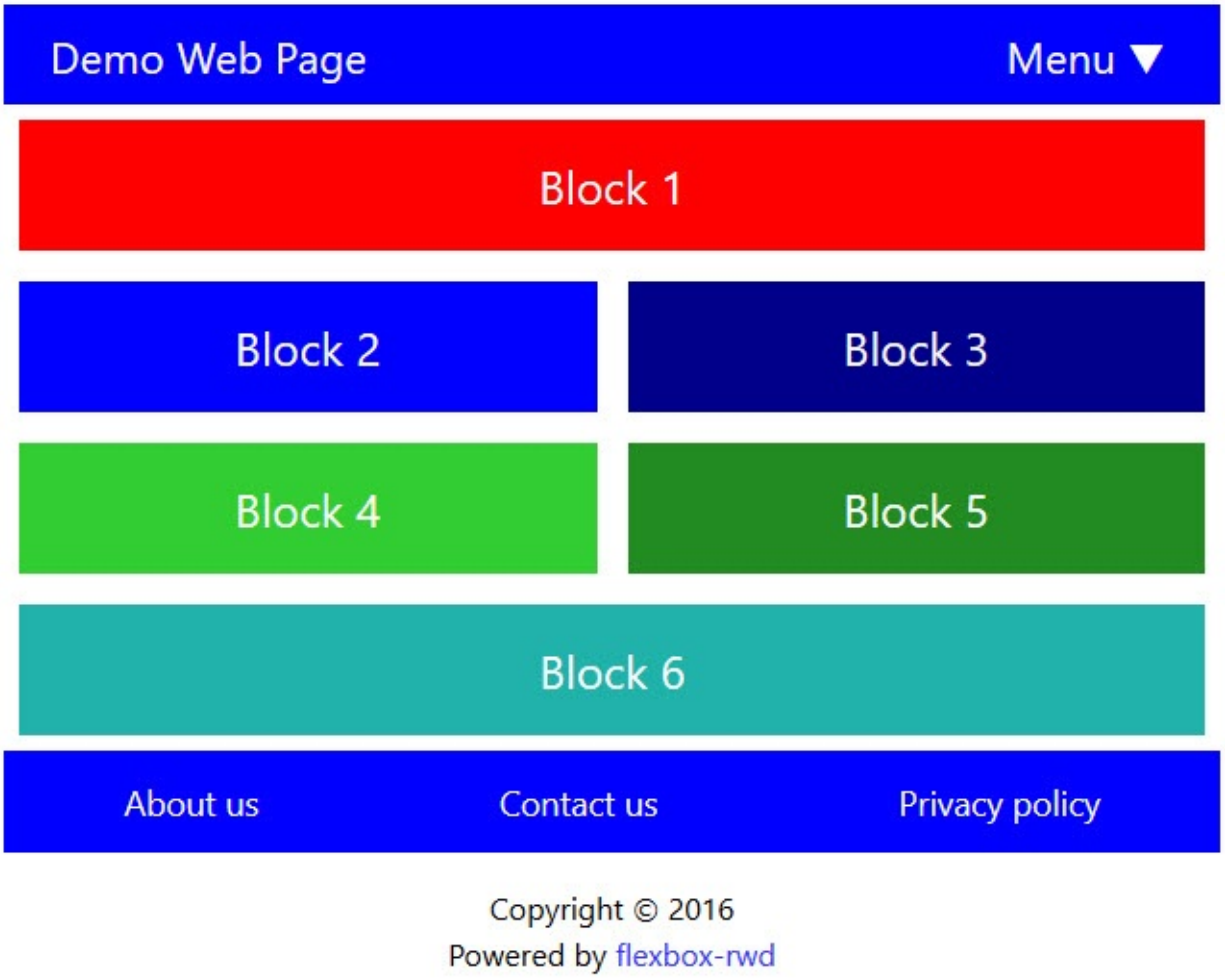
```
}
```

The “display: flex” declaration turns rows 2 and 3 into flex containers. And it turns each block within those rows into flex items. The “flex-wrap: wrap” declaration turns the flex containers into multi-line flex containers. It allows a block of content to wrap to a new line when there is not enough room for all the blocks of content on the current line. The “flex: 1 0 300px” declaration controls the flexibility of Blocks 2 through 6. It sets the initial width of all the blocks in rows 2 and 3 at 300 pixels. It allows them to expand beyond 300 pixels, but prevents them from shrinking below 300 pixels. If extra space is available on the line, all of the blocks expand equally. If the viewport is not wide enough to hold all of the blocks within the a single line, one of the blocks wraps to a new line.

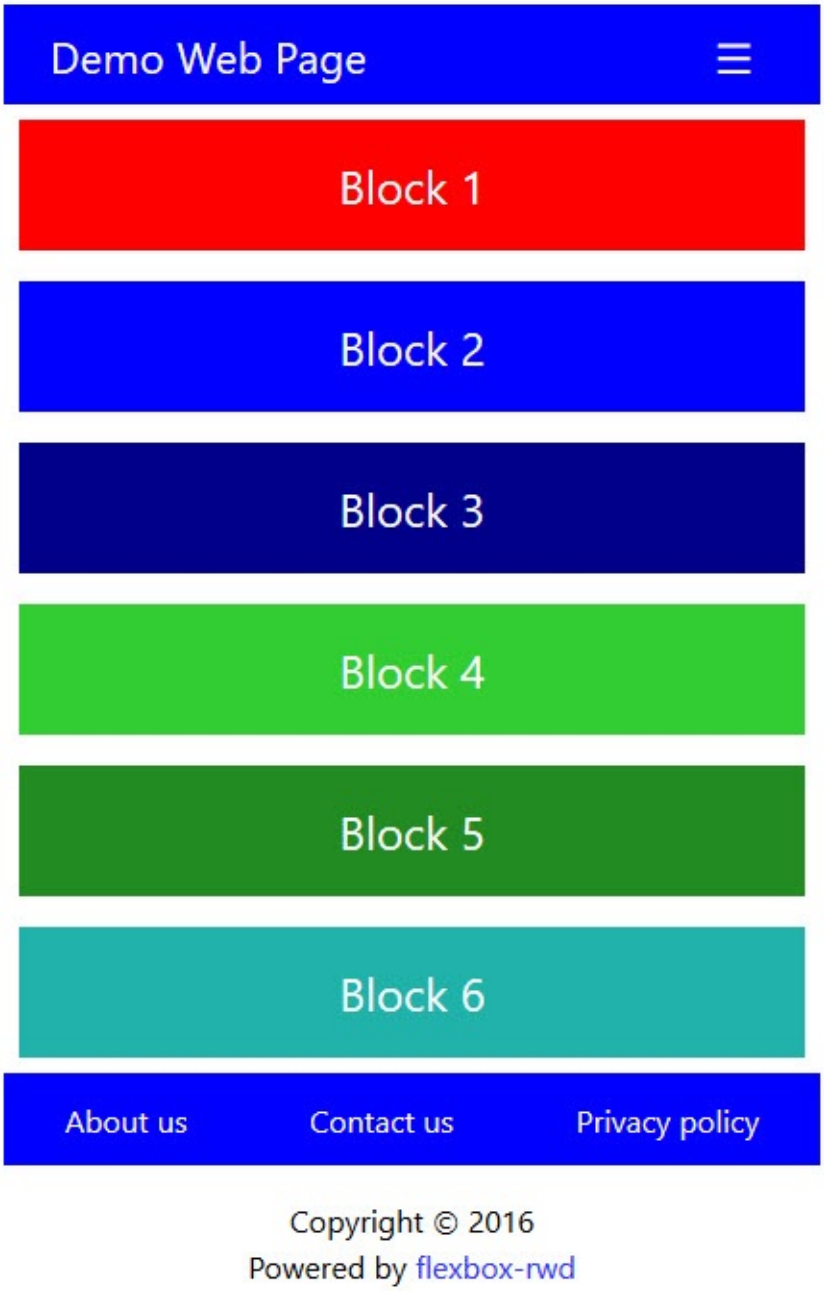
On large viewports with browsers that support flexbox, the enhanced version displays three rows in the main information component. There is one block of content in the first row, two blocks in the second row, and three blocks in the third row.



Recall that each block has a minimum-width constraint of 300 pixels. As the viewport grows smaller, eventually there is not enough room to hold three 300-pixel-wide blocks on a single line in the third row. At that point, Block 6 wraps to a new line, as shown below.



If you continue to shrink the viewport, eventually the screen is not wide enough to hold two 300-pixel-wide blocks on a single line. At that point, each block of content wraps to its own line; and the blocks are stacked one on top of another, just like the basic version of the main information component.



This layout, where content drops to a new line as the viewport shrinks, is an example of the Column Drop layout that we discussed in Chapter 15.

Source Code

Source code for the examples from this chapter can be downloaded at http://flexbox-rwd.com/download.aspx. Look in the Chapter 16 directory. For the template, see the template.html file. For the demo page, see the demo-page.html file.

Appendix A. Understanding Flex-Direction

The flex-direction property identifies the main axis of the flex container and determines how flex items are laid out along that axis. Flex-direction can take four values: row, row-reverse, column, and column-reverse.

The interpretation of these values depends on values assigned to two other CSS properties: writing-mode and direction.

The Western Writing System

In Western countries, the default settings for writing-mode and direction are:

- **writing-mode: horizontal-tb**. This means lines of text are horizontal, starting at the top of the page and moving toward the bottom.
- **direction: ltr**. This means text moves from left to right on the page.

With these settings, the values for flex-direction have the following meanings.

- **row** (default): Flex items are laid out in a row from left to right.
- **row-reverse**: Flex items are laid out in a row from right to left.
- **column**: Flex items are laid out in a column from top to bottom.
- **column-reverse**: Flex items are laid out in a column from bottom to top.

Other Writing Systems

In some situations, the default settings for writing-mode and direction may differ from the default settings for the Western writing system. For example, Hebrew text runs from right to left; and Chinese text is laid out vertically.

In those situations, the values for flex-direction have different meanings.

If you create a web page that is written in English, it is probably safe to assume that most folks who read your web page use a browser that defaults to the Western writing system. However, to be absolutely sure, you can define browser settings for “writing-mode” and “direction” explicitly. Just add the following code to your style sheet:

CSS

```
html {
    writing-mode: horizontal-tb;
    direction: ltr;
}
```

Now, any browser that displays your web page will use the Western writing system, and the values for “flex-direction” will have their common meanings.

Appendix B. Autoprefixer Options

There are many ways to incorporate Autoprefixer into your work flow. In Chapter 7, we identified two online options for implementing Autoprefixer:

- **Pleeease** at http://pleeease.io/play/.
- **Autoprefixer CSS online** at https://autoprefixer.github.io/.

If neither of these online options appeals to you, you can also implement Autoprefixer using build tools and text editors. Here are some other possibilities that you can check out:

- Gulp-Postcss with Autoprefixer at https://github.com/postcss/gulp-postcss.
- Grunt-Postcss with Autoprefixer at https://github.com/nDmitry/grunt-postcss.
- PostCSS for Webpack at https://github.com/postcss/postcss-loader.
- Ruby on Rails at https://github.com/ai/autoprefixer-rails#compass.
- Sublime Text at https://github.com/sindresorhus/sublime-autoprefixer.
- Brackets at https://github.com/mikaeljorhult/brackets-autoprefixer.
- Atom Editor at https://github.com/sindresorhus/atom-autoprefixer.

The last three options (Sublime Text, Brackets, and Atom Editor) are text editors. They work but they have some disadvantages. Build tools like Gulp work much better, but they can be challenging for beginners.

Appendix C. Checkbox Hack

In the world of responsive web design, websites that work on large screens must also work on small screens. This can be challenging. Sometimes, all of the content displayed on a large screen just doesn’t fit nicely on a small screen.

One way to address the problem of limited space on small screens is to hide content. The hidden content is revealed when either of the following occurs:

- A user asks to see the hidden content (e.g., by checking a box or clicking a button).
- The viewport expands enough to accommodate the hidden content.

This appendix describes the checkbox hack - a way to control the visibility of web content using HTML and CSS.

Checkbox Hack Markup

The checkbox hack is a clever bit of code that allows you to respond to click events with just HTML and CSS – no javascript. The technique can be used to hide or show web content, in response to a click event.

> ***NOTE***
>
> ***For a description of stuff you can do with the checkbox hack, read Chris Coyier's excellent article at https://css-tricks.com/the-checkbox-hack/.***

To illustrate how the checkbox hack works, we'll step through an example. We'll build a simple, collapsible content button – a button that hides or shows content when it is clicked. Along the way, we'll touch on some style choices that can enhance the user experience.

HTML for the Checkbox Hack

The checkbox hack consists of only three HTML elements – an <input> checkbox, a <label> element, and a container for the "hidden" content. For this example, we will use a element to contain the content.

> **HTML**
>
> <input type="checkbox" id="toggle">
>
> <label for="toggle">Click me!</label>
>
> Lorem ipsum dolor sit …

There is nothing unusual here – just ordinary HTML. But for the checkbox hack to work correctly, the following conditions must be met:

- The <input> element must precede the <label> element.
- The <label> element must be linked through its "for" attribute to the <input> element.
- The <input>, <label>, and elements must all be siblings.

Because the <label> and checkbox are linked, the checkbox will toggle from checked to unchecked whenever a user clicks the <label> element.

CSS for the Checkbox Hack

The magic happens through CSS. The <label> is visible by default, the checkbox is hidden, and the toggles between hidden and visible when a user clicks

the <label> element.

CSS

```
input [type="checkbox"] { display: none; }

div.hide { display: none; }

input [type=checkbox]:checked ~ div.hide { display: block; }
```

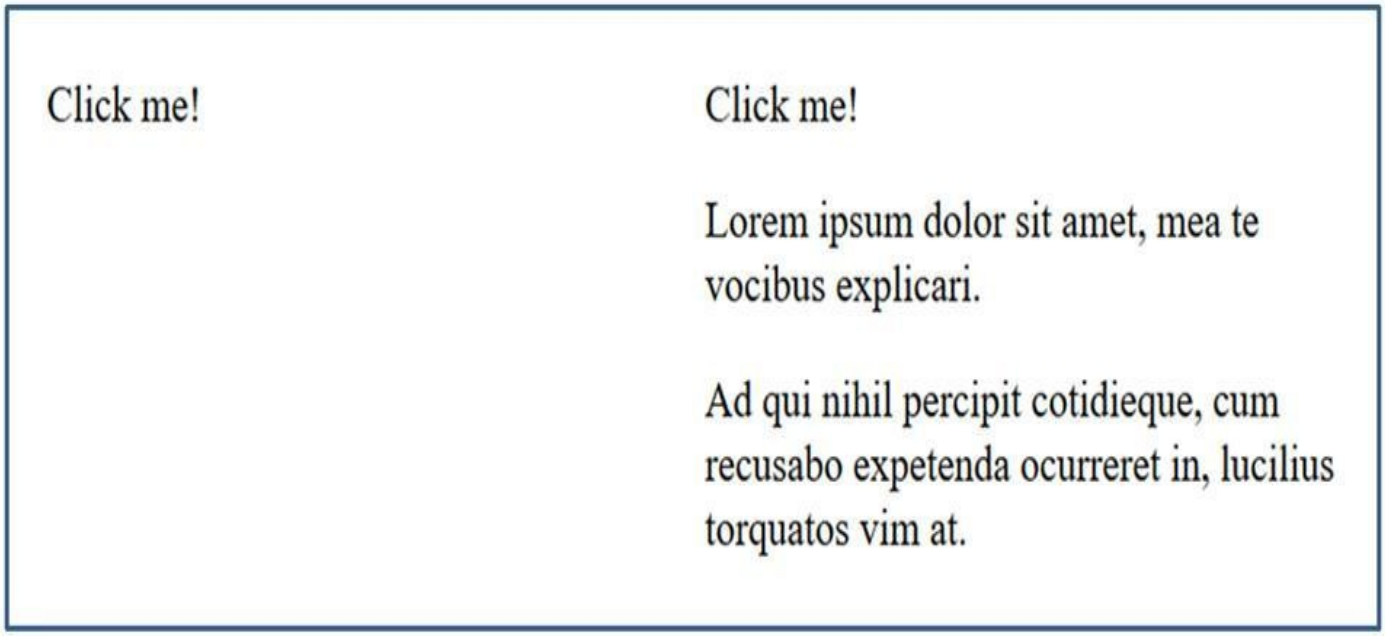
The first line of CSS markup hides the checkbox. The second line hides the content. The third line of code is the heart of the checkbox hack. It uses the ":checked" selector and the general siblings selector (~) to toggle the visibility of the element.

NOTE

For a good description of the general siblings selectors (~), visit https://developer.mozilla.org/en-US/docs/Web/CSS/General_sibling_selectors.

Here's how the checkbox hack works. The checkbox itself is not visible, but it is linked to the <label> element through a "for" attribute. When a user clicks the <label> element, it toggles the hidden checkbox. When the box is checked, the collapsible content (i.e., the element) is displayed; when the box is unchecked, the collapsible content is hidden.

And here is how things look on the page.



On the left, you see the "Click me!" label with the collapsible content hidden. On the right, you see what happens after a user clicks the label. Neither view is very attractive or very user-friendly, yet.

Styling the Label

At this point, the checkbox hack is functional. When the content is hidden, clicking the label makes it appear; and when the content is visible, clicking the label makes it disappear.

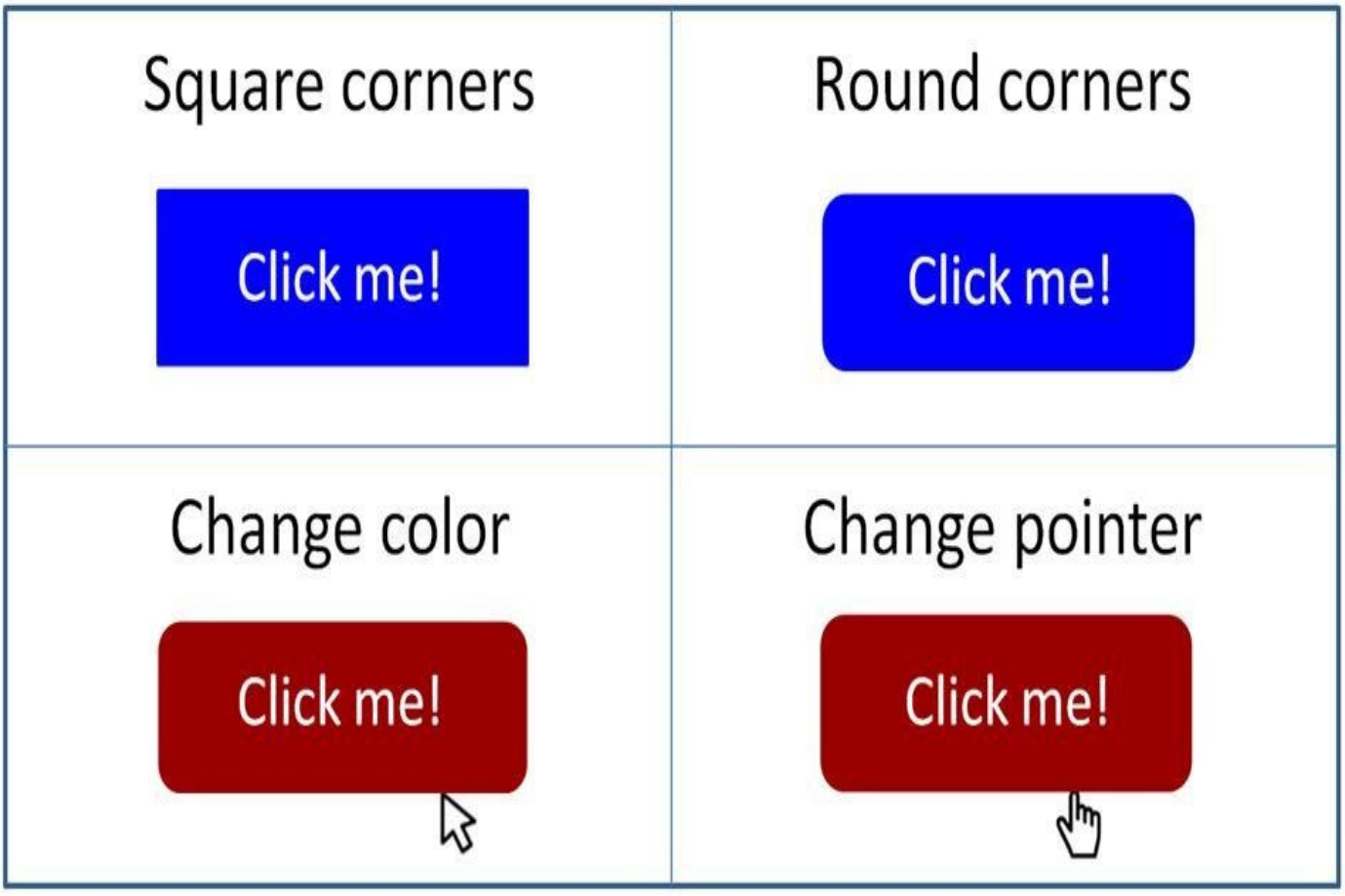
But the user experience is poor. To an unsuspecting user, the “Click me” <label> looks just like other text on the page. There is no hint that this <label> has hidden powers. We can fix that.

In the context of the checkbox hack, a <label> behaves like a command button. When a user clicks it, things happen. So let’s style it to look and to act like a button.

CSS

```
label {
    background-color: blue;
    color: white;
    padding: 1em 2em;
    border-radius: .5em;
    cursor: pointer; }
label:hover { background-color: red; }
```

You can use some or all of the CSS properties shown above to simulate the look and feel of a command button. Here are a few things you can do.



Setting the background-color, color, and padding properties gives the <label> the look of a square-cornered button (upper left). Adding a border-radius gives it round corners (upper right) – a little more buttony. Setting the ":hover" pseudo class changes the background color when a user mouses over the button (lower left). And setting the "cursor" property equal to "pointer" turns the cursor into a hand icon when the mouse pointer moves over the button (lower right).

When you're building a button to display collapsible content, it's helpful to add arrows to the button. You can do that with just a little more CSS.

CSS

```
label:after {
    content: "\25BC";     /* unicode down arrow */
    padding-left: 250px; }
input[type=checkbox]:checked ~ label:after {
    content: "\25B2"; }     /* unicode up arrow */
```

Here are the buttons with arrows.



On the left, when the collapsible content is hidden, the downward-facing arrow tells the user that something will slide down from the button if it is clicked. On the right, the upward-facing arrow tells the user that content will collapse into the button after it is clicked. Each time the button is clicked, content appears or disappears; and the arrow direction changes to indicate the current visibility of content.

The main takeaway here is that you should style the <label> element to reflect its special functionality in the checkbox hack. It looks better and it enhances the user experience.

Styling Hidden Content

Just as it is important to style the label in the checkbox hack, it is also important to style the hidden content. You can affect not only the look of the content, but also the behavior. Depending on how it is styled, hidden content can:

- Overlap existing content or push it out of the way.
- Appear and disappear instantly or slide in and out gradually like a window shade.

Best of all, this functionality can be achieved with only one or two additional lines of code.

Overlapping Content

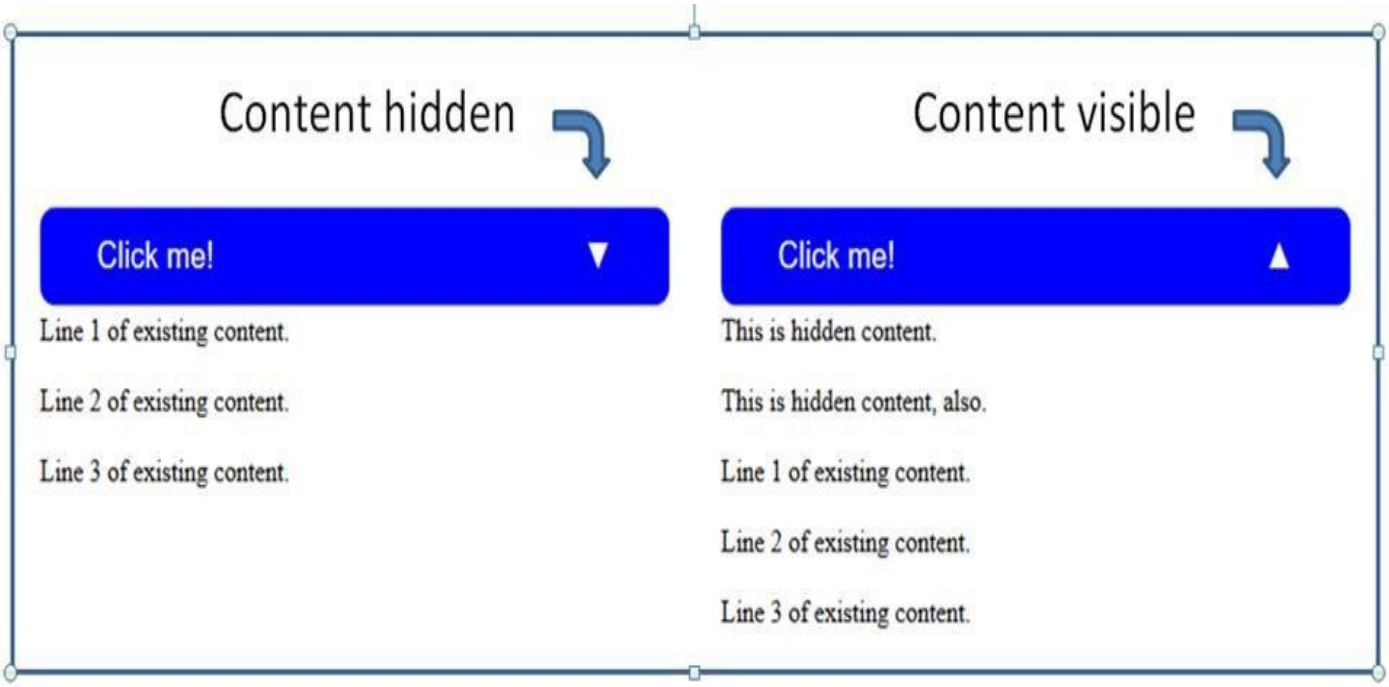
What happens to existing content when hidden content is revealed? If you make no changes to the code we've used so far, hidden content will push existing content out of the way. Sometimes, that is what you want. But sometimes you want the hidden content to sit on top of existing content. This section explains how to make that happen.

Here's the markup that we've used to style the element, so far:

CSS

```
div.hide { display: none; }

input [type=checkbox]:checked ~ div.hide { display: block; }
```

With this markup, the hidden content pushes surrounding content out of the way when it becomes visible as, as shown below.



When it becomes visible (above right), the hidden content displaces existing content – pushes it two lines down the page.

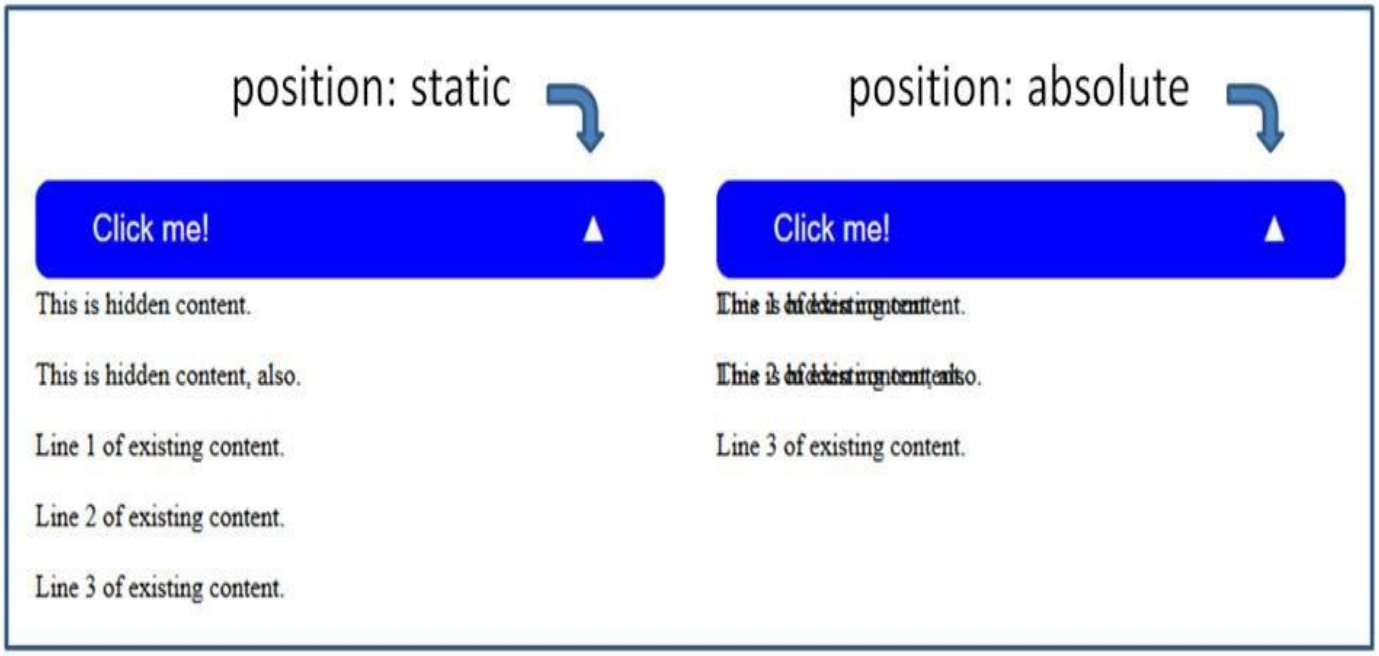
This occurs because the container that holds hidden content has a “position” property equal to “static”. This is the default value of the “position” property. It causes elements to render in order as they appear in the document flow. In our HTML markup, hidden content appears before existing content; so when hidden content is made visible, it pushes existing content out of the way.

To change this behavior, we need to change the value of the “position” property. If we change its value from “static” to “absolute”, the hidden content will sit on top of the existing content, rather than push it away. Here’s the CSS markup, with the “position” declaration shown in bold:

CSS

```
div.hide { display: none; position:absolute; }

input [type=checkbox]:checked ~ div.hide { display: block; }
```

And here is how things look with the position property at its default value of “static” versus a value of “absolute”:



Oops! There is a problem. When we set the “position” property equal to “absolute”, the hidden content covers the existing content, as expected. But the covered text remains visible as well. Neither the hidden content nor the text that gets covered is legible.

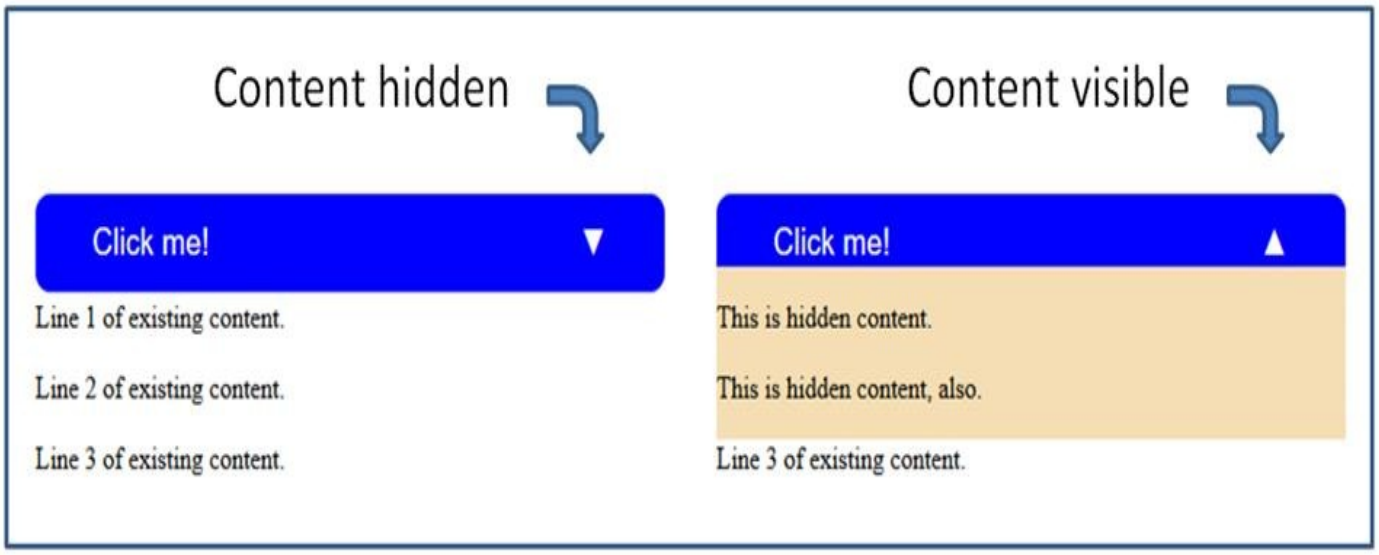
The solution is to declare a background color for the hidden content. Here’s the markup, with the background-color declaration shown in bold:

CSS

```
div.hide { display: none; position:absolute; background-color: wheat;}

input [type=checkbox]:checked ~ div.hide { display: block; }
```

Now, the hidden content will cover the existing content, and the existing content

won’t show through.



Success! When the hidden content container (i.e., the element) has a non-transparent background color, the existing content does not show through; and the hidden content is readable when it becomes visible.

To summarize, when you want hidden content to push existing content out of the way, use these style rules:

CSS

```
div.hide { display: none; }

input [type=checkbox]:checked ~ div.hide { display: block; }
```

When you want hidden content to sit on top of existing content, use these rules:

CSS

```
div.hide { display: none; position:absolute; background-color: wheat;}

input [type=checkbox]:checked ~ div.hide { display: block; }
```

Naturally, the particular choice of background color is arbitrary. The main thing is to use a non-transparent background color when you want hidden content to sit on top of existing content.

Hidden Content Transition

The markup that we have been using causes hidden content to appear instantly when it is displayed and disappear instantly when it is hidden. As an alternative, hidden content can be made to slide in and out of view gradually.

To slide content in and out of view, we need to use the CSS “transition” property. Unfortunately, the “transition” property does not play nicely with the “display” property. And we have used the display property to control the visibility of the hidden content container, as shown below:

CSS

div.hide { display: none; }

input [type=checkbox]:checked ~ div.hide { display: block; }

So to use the “transition” property, we need to control the visibility of the hidden content container without using the “display” property. One way to do this is the use the “max-height” property and the “overflow” property, shown below in bold:

CSS

div.hide { **max-height: 0, overflow: hidden;** }

input [type=checkbox]:checked ~ div.hide { **max-height: 999px;** }

See how this works. Initially, the hidden content container has a maximum height of zero pixels, with “overflow: hidden”, so it is not visible. Then, when the checkbox is checked, the maximum height gets bigger; and the hidden content is visible.

To introduce a transition to the checkbox hack, we need to add one additional CSS declaration. New markup for the complete checkbox hack appears below, with the added CSS transition declaration highlighted in bold:

HTML

<input type=“checkbox” id=“toggle”>

<label for=“toggle” >Collapsible Content</label>

Lorem ipsum dolor sit …

CSS

input [type=“checkbox”] { display: none; }

div.hide { max-height: 0, overflow: hidden; **transition: all .8sec ease-in-out;** }

input [type=checkbox]:checked ~ div.hide { max-height: 999px; }

On the screen, here’s what happens. When a user clicks the <label> element, the hidden content will take 0.8 sec to slide up or down, much like a window shade. To learn more about how to use the CSS transition property, visit https://css-tricks.com/almanac/properties/t/transition/.

Advanced Checkbox Hack

You may have noticed that the checkbox hack uses standard HTML and CSS – nothing fancy. You'd think it would work in all browsers, but it doesn't.

What's wrong with the checkbox hack? The checkbox hack has two problems:

It doesn't work on some Android devices.

It doesn't work on some iOS devices.

Luckily, there are simple solutions for each problem. Added to the original checkbox hack, these solutions are called the advanced checkbox hack.

> ***NOTE***
>
> ***The advanced checkbox hack was developed by Tim Pietrusky. You can read his original article at http://timpietrusky.com/advanced-checkbox-hack.***

Android Solution

The solution to problems with Android devices is to add the following CSS to the <body> element:

CSS

```
body { webkit-animation: bugfix infinite 1s; }
@-webkit-keyframes bugfix {
    from {padding:0;}
    to {padding:0;}
}
```

For some reason, this solves the problem with Android devices.

iOS Solution

The solution for iOS devices is even simpler. Just add an empty "onclick" to the <label> element, as shown below in bold.

HTML

```
<label for="toggle" onclick>Collapsible Content</label>
```

That's all there is to it. Now, the iOS problem is solved.

Complete Markup

When you add the Android solution and the iOS solution to the original checkbox

hack, you have the advanced checkbox hack. Complete markup for the advanced checkbox hack appears below:

HTML

```
<input type="checkbox" id="toggle">
<label for="toggle" onclick>Collapsible Content</label>
<div class="hide">Lorem ipsum dolor sit …</div>
```

CSS

```
body { webkit-animation: bugfix infinite 1s; }
@-webkit-keyframes bugfix {
    from {padding:0;}
    to {padding:0;}
}
input [type="checkbox"] { display: none; }
div.hide { display: none; }
input [type=checkbox]:checked ~ div.hide{ display: block; }
```

The advanced checkbox solves the Android and iOS problems, and only requires a few extra lines of code. There is no reason to use the checkbox hack when the advanced checkbox hack is available.

What About Javascript?

Sadly, using the checkbox hack to hide content is frowned upon by some designers. Why? Because, in a sense, the checkbox hack extends CSS beyond its classic purpose.

In an ideal web design world, HTML is used for structure; CSS, for style; and javascript, for behavior. The checkbox hack uses HTML and CSS to control behavior – that is, to respond to click events. That's javascript's job!

In reality, the ability to respond to click events without javascript is sort of cool, but it has pros and cons. Benefits include lean code, quick response, and a modest learning curve. On the other hand, javascript is very powerful. You can do things with javascript that are difficult or impossible to do with CSS.

Throughout this book, we use the checkbox hack rather than javascript to control the visibility of elements on the page. This allows readers who are familiar with HTML and CSS, but not with javascript, to understand the code. It is in keeping with the goal of the book to build responsive web pages with HTML and CSS – no javascript.

However, if you are already proficient in javascript, you might be more comfortable

using javascript in place of the checkbox hack.