CS 50 Web Design

APCSP Module 2: Internet

Unit 2: CSS (Chapter 15-19)

LECTURE 6A: RESPONSIVE WEB DESIGN

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Flex containers and items
- Flow direction and wrapping
- Flex item alignment
- Controlling item “flex”
- Grid containers and items
- Setting up a grid template
- Placing items in the grid
- Implicit grid features
- Grid item alignment

Objectives

- What RWD is and why it’s important
- Fluid layouts
- Media queries
- Design strategies and patterns
- Testing options

Introduction

SECTION 1

Introduction

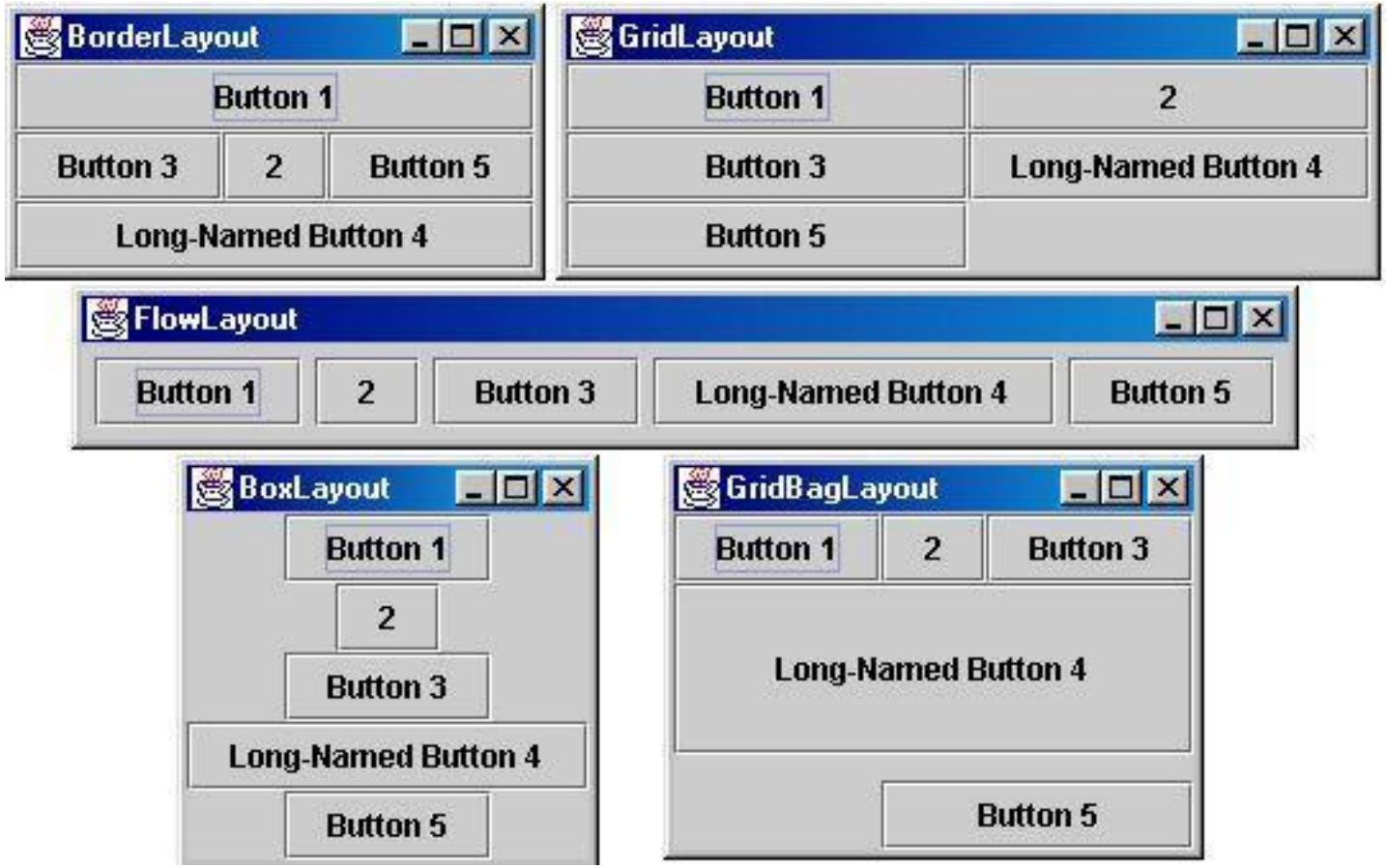
In it, you will learn about two important CSS page layout tools:

- **Flexbox** for greater control over arranging items along one axis
- **Grid** for honest-to-goodness grid-based layouts, like those print designers have used for decades

Introduction

- Each tool has its special purpose, but you can use them together to achieve layouts we've only dreamed of until now. For example, you could create the overall page structure with a grid and use a flexbox to tame the header and navigation elements. Use each technique for what it's best suited for—you don't have to choose just one

Layout managers for Java Swing



Flexible Boxes with Flexbox

SECTION 2

Flexible Boxes with CSS Flexbox

- The CSS Flexible Box Layout Module (also known as simply Flexbox) gives designers and developers a **handy tool** for laying out components of web pages such as menu bars, product listings, galleries, and much more.

- According to the spec,

  The defining aspect of flex layout is the ability to make the flex items "flex," altering their width/height to fill the available space in the main dimension.

Flexible Boxes with CSS Flexbox

- That means it allows items to stretch or shrink inside their containers, preventing wasted space and overflow—a real plus for making layouts fit a variety of viewport sizes.

- Other advantages include the following:
    - The ability to make all neighboring items the same height
    - Easy horizontal and vertical centering (curiously elusive with old CSS methods)
    - The ability to change the order in which items display, independent of the source

In this book, Philadelphia Ice Creams, comprising the first group, are very palatable, but expensive. In many parts of the country it is quite difficult to get good cream. For that reason, I have given a group of creams, using part milk and part cream, but it must be remembered that it takes smart "juggling" to make ice cream from milk. By far better use condensed milk, with enough water or milk to rinse out the cans.

Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream.

In places where neither cream nor condensed milk can be purchased, a fair ice cream is made by adding two tablespoonfuls of olive oil to each quart of milk. The cream for Philadelphia Ice

If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is *very* cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.

Use of Fruits

Use fresh fruits in the summer and the best canned unsweetened fruits in the winter. If sweetened fruits must be used, cut down the given quantity of sugar. Where acid fruits are used, they should be added to the cream after it is partly frozen.

water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

The average time for freezing two quarts of cream should be ten minutes; it takes but a minute or two longer for larger quantities.

Pound the ice in a large bag with a mallet, or use an ordinary ice shaver. The finer the ice, the less time it takes to freeze the cream. A four quart freezer will require ten pounds of ice, and a quart and a pint of coarse rock salt. You may pack the freezer with a layer of ice three inches thick, then a layer of salt one inch thick, or mix the ice and salt in the tub and shovel it around the freezer.

FIGURE 16-1. An example of text formatted with the multicolumn properties.

Multicolumn Layout

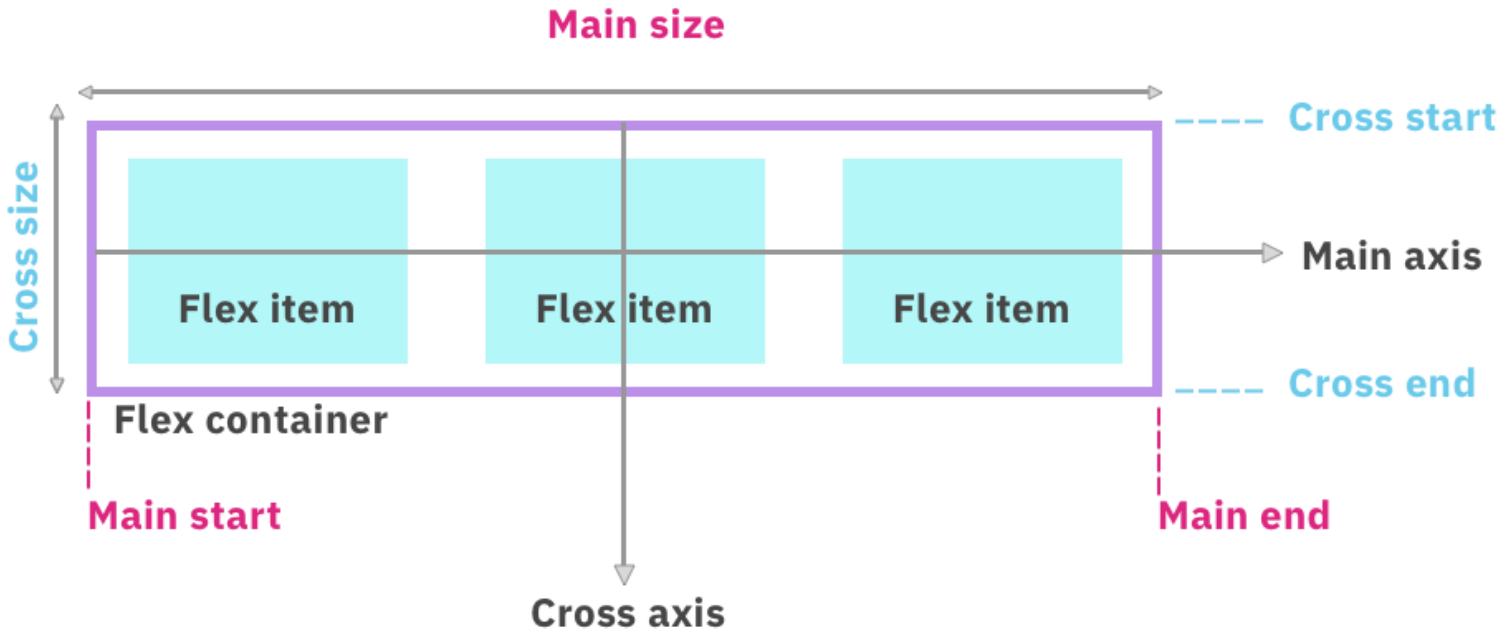
Multicolumn Layout

- A third CSS3 layout tool you may want to try is multicolumn layout.
- The Multi-column Layout Module (*w3.org/TR/css-multicol-1*) provides tools for pouring text content into a number of columns, as you might see in a newspaper (FIGURE 16-1). It is designed to be flexible, allowing the widths and number of columns to automatically fit the available space.
- This chapter is already big enough, so I've put this lesson in an article, "Multicolumn Layout" (PDF), available at *learningwebdesign.com/articles/*.

Multicolumn Layout

Flexbox Containers

Basic Terminology



Flexbox Containers

Basic Terminology

- **main axis** – The main axis of a flex container is the primary axis along which flex items are laid out. Beware, it is not necessarily horizontal; it depends on the flex-direction property (see below).
- **main-start | main-end** – The flex items are placed within the container starting from main-start and going to main-end.
- **main size** – A flex item's width or height, whichever is in the main dimension, is the item's main size. The flex item's main size property is either the 'width' or 'height' property, whichever is in the main dimension.
- **cross axis** – The axis perpendicular to the main axis is called the cross axis. Its direction depends on the main axis direction.
- **cross-start | cross-end** – Flex lines are filled with items and placed into the container starting on the cross-start side of the flex container and going toward the cross-end side.
- **cross size** – The width or height of a flex item, whichever is in the cross dimension, is the item's cross size. The cross size property is whichever of 'width' or 'height' that is in the cross dimension.

Flexbox Containers

THE MARKUP

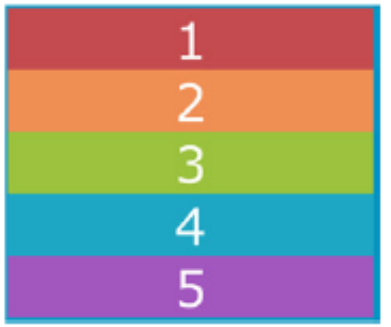
```
<div id="container">
    <div class="box box1">1</div>
    <div class="box box2">2</div>
    <div class="box box3">3</div>
    <div class="box box4">4</div>
    <div class="box box5">5</div>
</div>
```

THE STYLES

```
#container {
    display: flex;
}
```

Flexbox Containers

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



block layout mode

```
display: flex;
```

flexbox layout mode

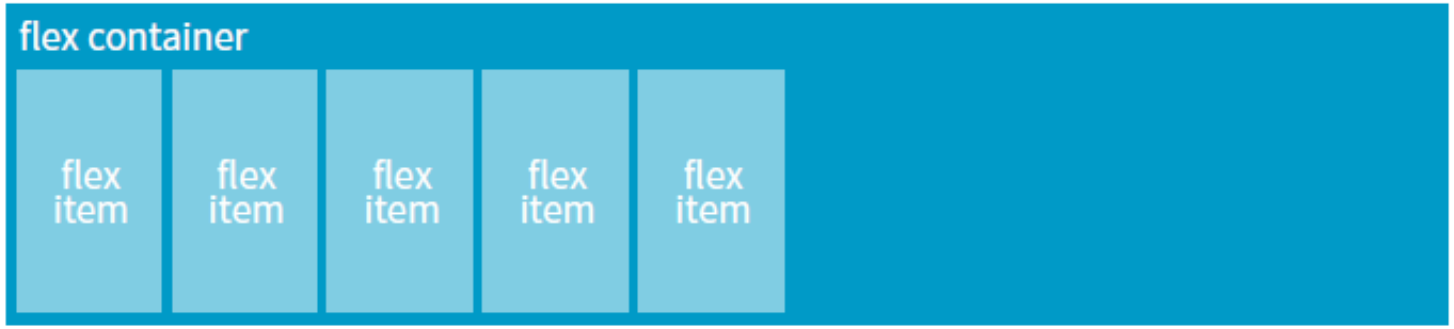


FIGURE 16-2. Applying the flex display mode turns the child elements into flex items that line up along one axis. You don't need to do anything to the child elements themselves.

Controlling the "Flow" Within the Container

Specifying flow direction

You may be happy with items lining up in a row as shown in FIGURE 16-2, but there are a few other options that are controllled with the **flex-direction** property.

flex-direction

Values:	row | column | row-reverse | column-reverse
Default:	row
Applies to:	flex containers
Inherits:	no

Controlling the “Flow” Within the Container

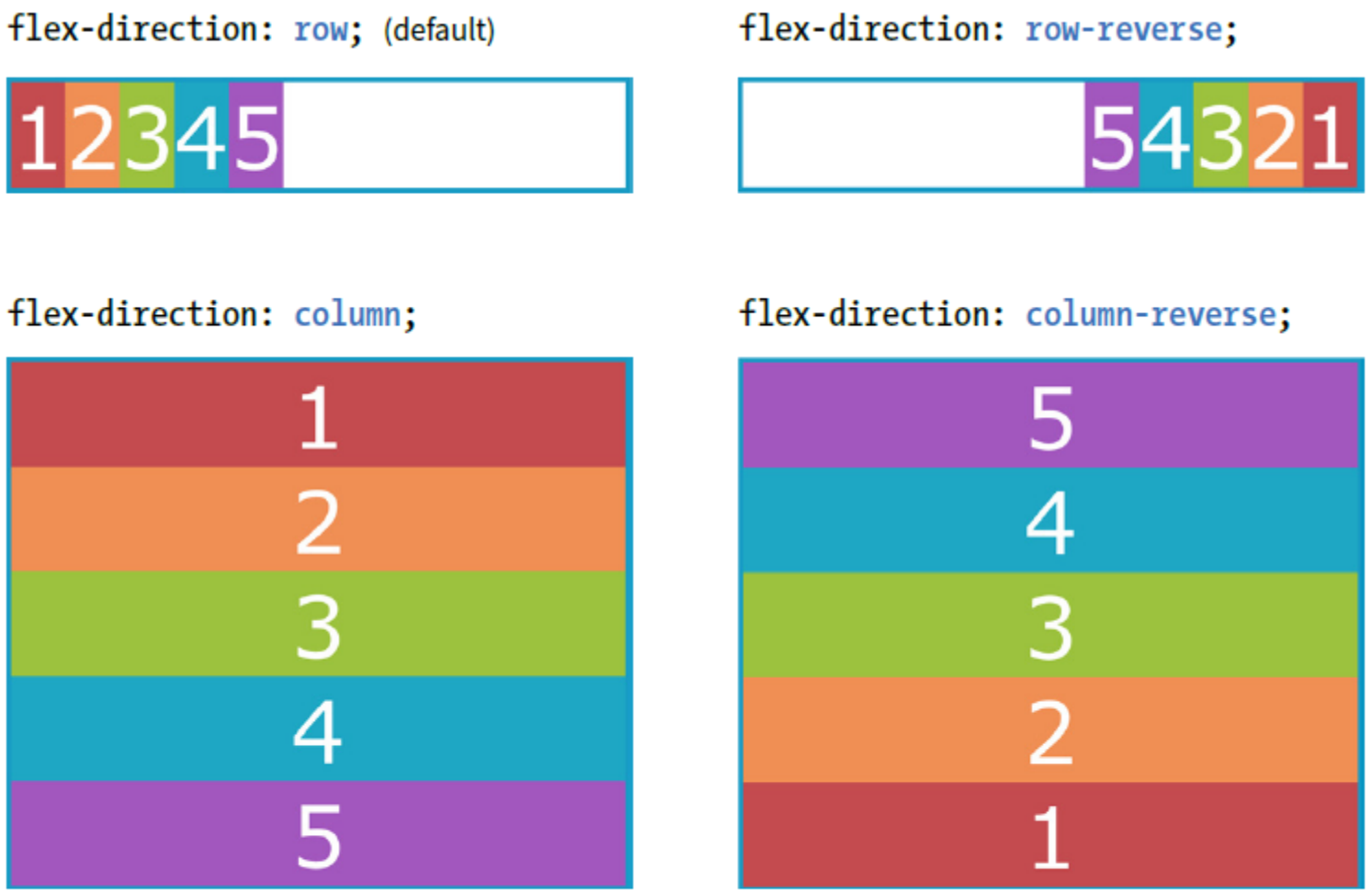
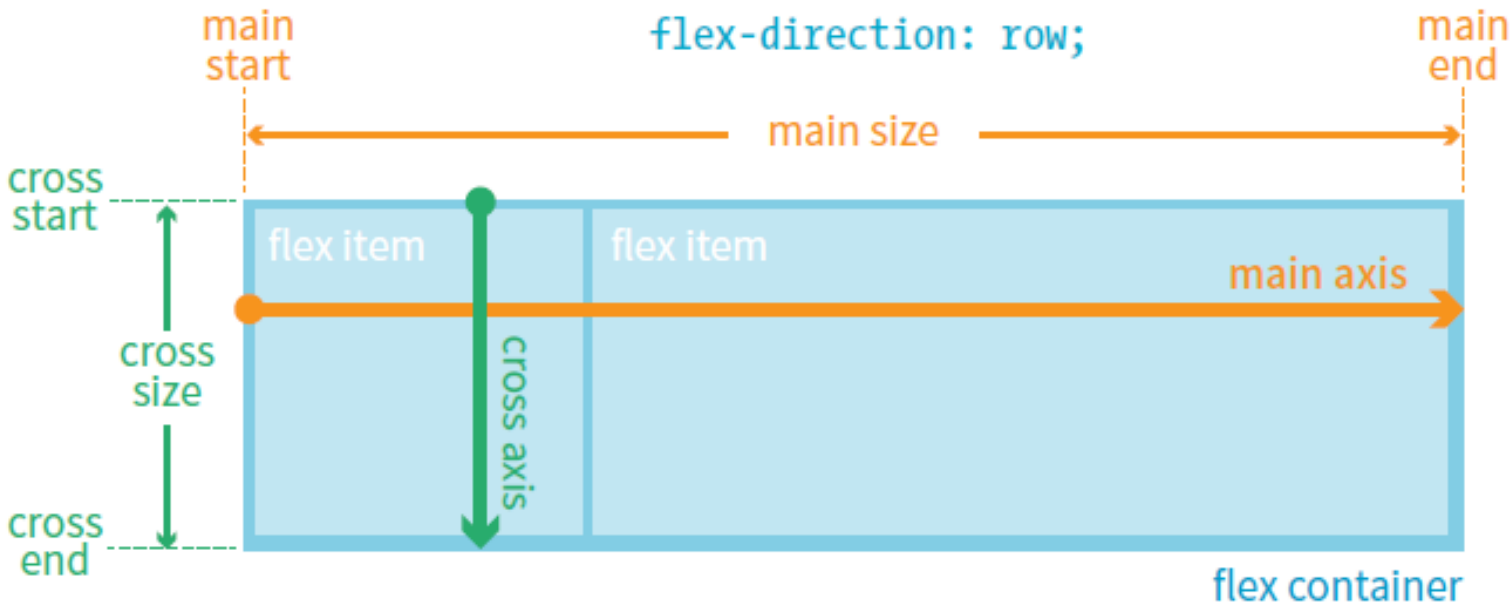


FIGURE 16-3. Examples of **flex-direction** values **row**, **row-reverse**, **column**, and **column-reverse**.

Controlling the “Flow” Within the Container

FOR LANGUAGES THAT READ HORIZONTALLY FROM LEFT TO RIGHT:

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



Controlling the “Flow” Within the Container

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

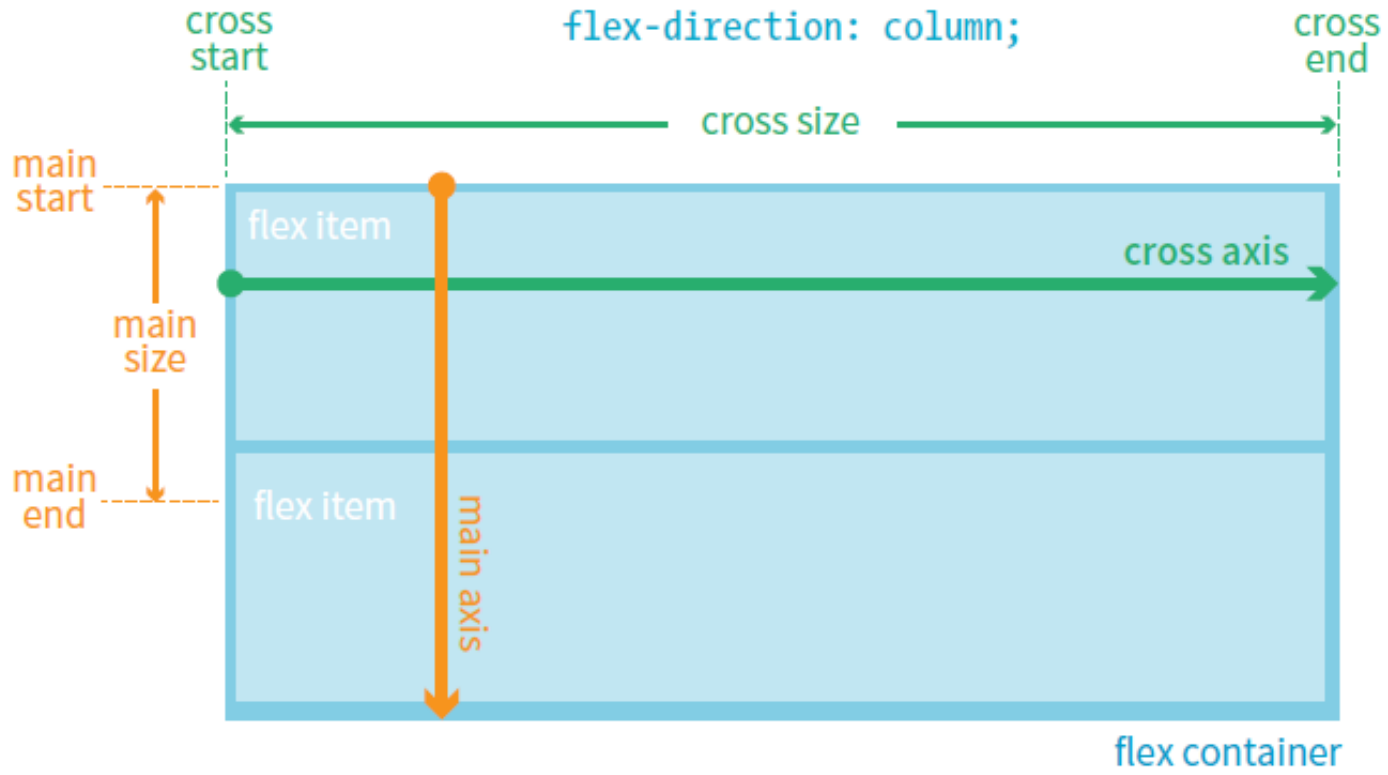


FIGURE 16-4. The parts of a flex container.

Wrapping onto multiple lines

- If you have a large or unknown number of flex items in a container and don’t want them to get all squished into the available space, you can allow them to break onto additional lines with the **flex-wrap** property.

flex-wrap

Values: nowrap | wrap | wrap-reverse
Default: nowrap
Applies to: flex containers
Inherits: no

Wrapping onto multiple lines

THE MARKUP

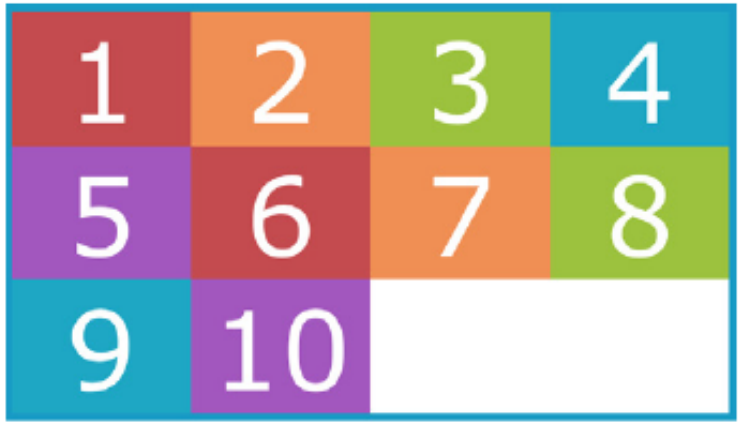
```
<div id="container">
   <div class="box box1">1</div>
   <!-- more boxes here -->
   <div class="box box10">10</div>
</div>
```

THE STYLES

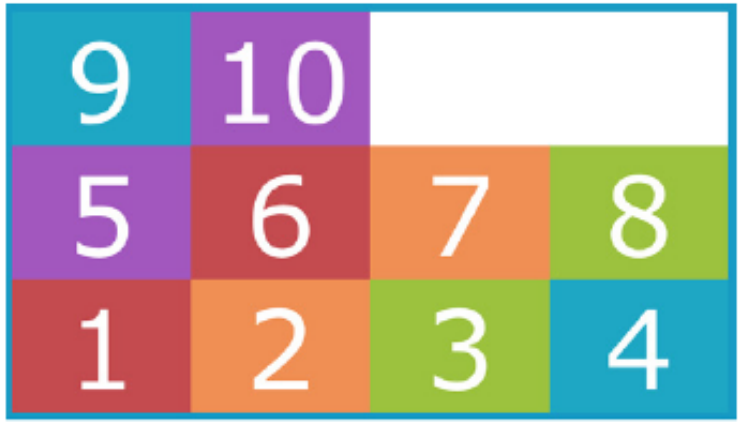
```
#container {
   display: flex;
   flex-direction: row;
   flex-wrap: wrap;
}
.box {
   width: 25%;
}
```

Wrapping onto multiple lines

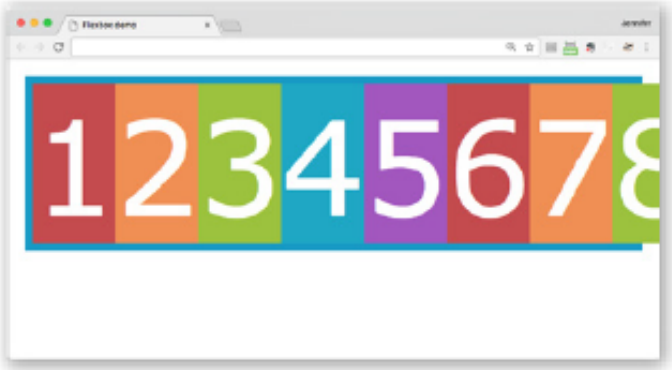
`flex-wrap: wrap;`



`flex-wrap: wrap-reverse;`



`flex-wrap: nowrap;` (default)



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

FIGURE 16-5. Comparing the effects of **nowrap**, **wrap**, and **wrap-reverse** keywords for **flex-wrap**.

Wrapping onto multiple lines

THE MARKUP

```
<div id="container">
   <div class="box box1">1</div>
   <!-- more boxes here -->
   <div class="box box10">10</div>
</div>
```

THE STYLES

```
#container {
   display: flex;
   height: 350px;
   flex-direction: column;
   flex-wrap: wrap;
}
.box {
   width: 25%;
}
```

Wrapping onto multiple lines

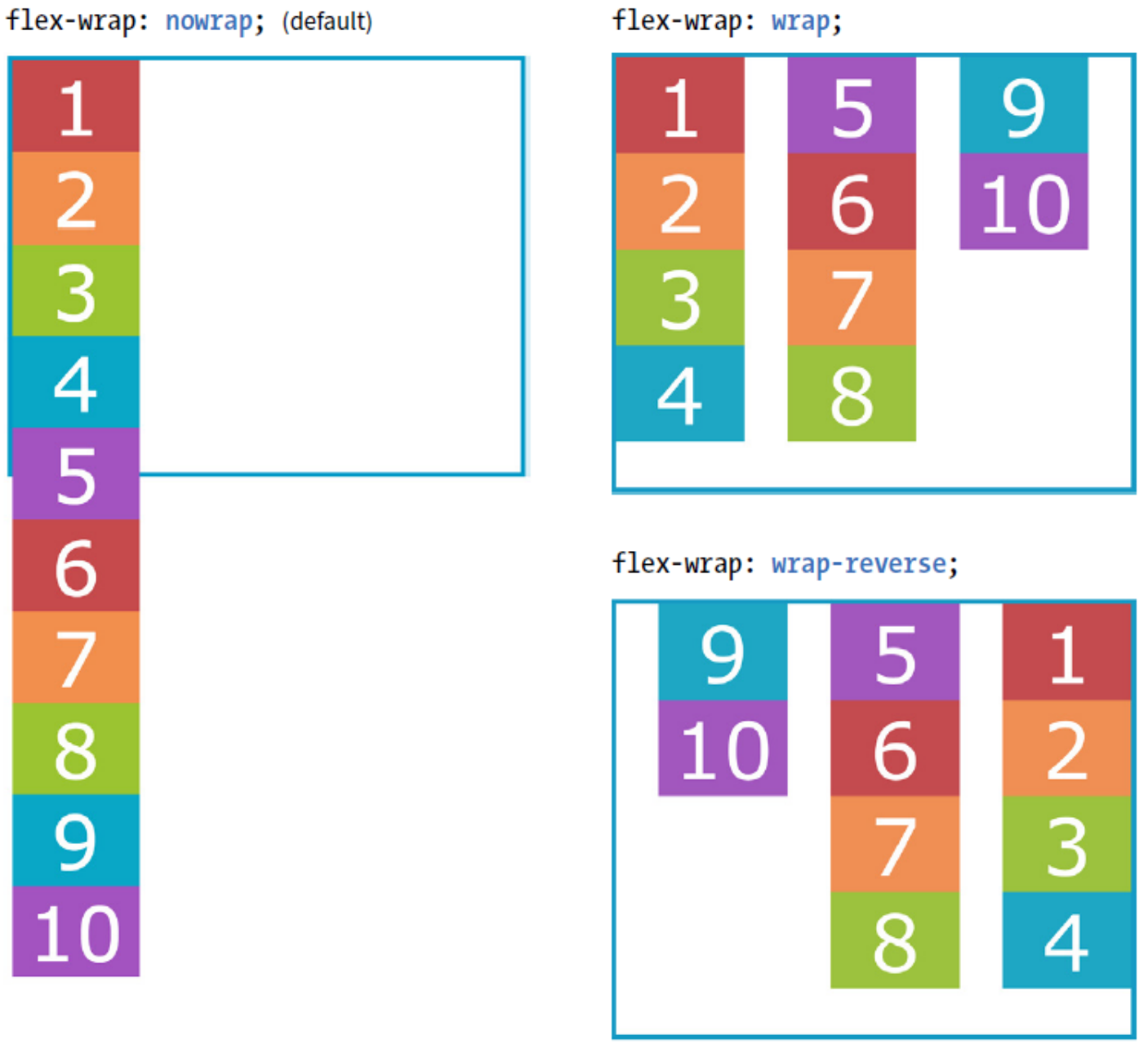


FIGURE 16-6. Comparing **nowrap**, **wrap**, and **wrap-reverse** when the items are in a column.

Putting it together with flex-flow

- The shorthand property flex-flow makes specifying flex-direction and flex-wrap short and sweet. Omitting one value results in the default value for its respective property, which means you can use flex-flow for either or both direction and wrap.

flex-flow

Values: *flex-direction flex-wrap*
Default: row nowrap
Applies to: flex containers
Inherits: no

- Using **flex-flow**, I could shorten the previous example (FIGURE 16-6) like so:

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

Alignment/Sizing/Order for Flexbox

SECTION 3

Aligning on the main axis

- The **justify-content** property defines how extra space should be distributed around or between items that are inflexible or have reached their maximum size (see Note).

justify-content

Values: flex-start | flex-end | center | space-between | space-around
Default: flex-start
Applies to: flex containers
Inherits: no

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

Aligning on the main axis

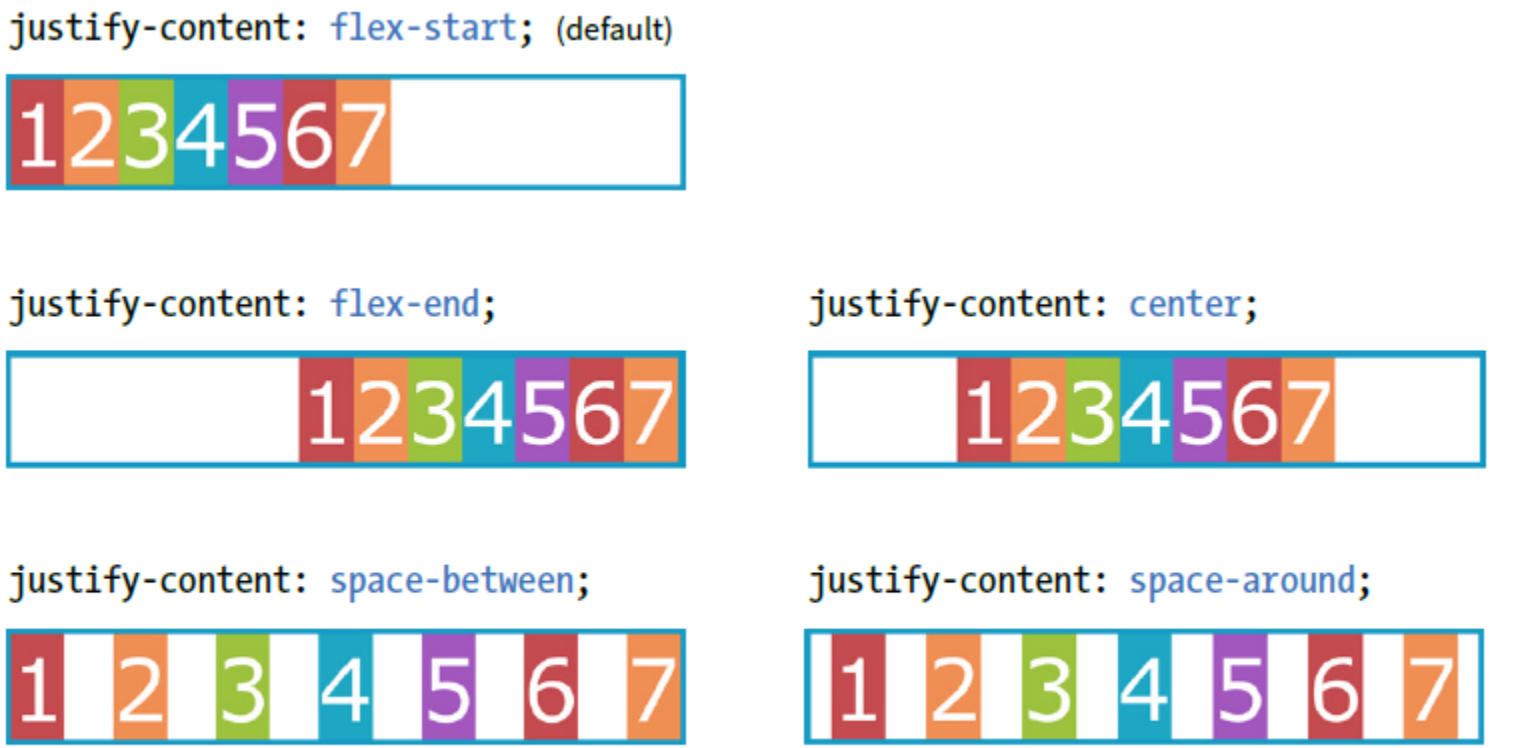
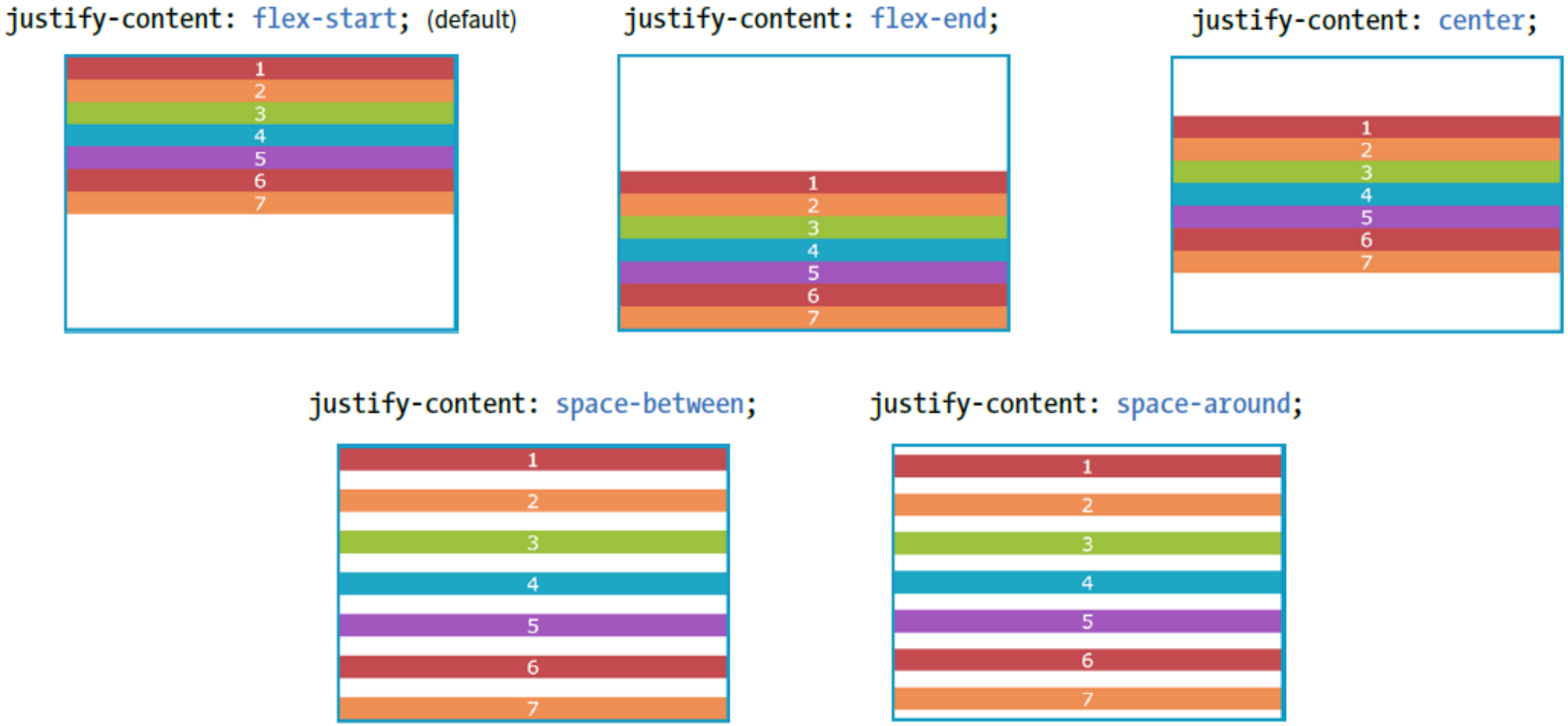


FIGURE 16-8. Options for aligning items along the main axis with **justify-content**.



Aligning on the main axis

FIGURE 16-9. Options for aligning items along a vertical main axis (**flex-direction** set to **column**) with **justify-content**.

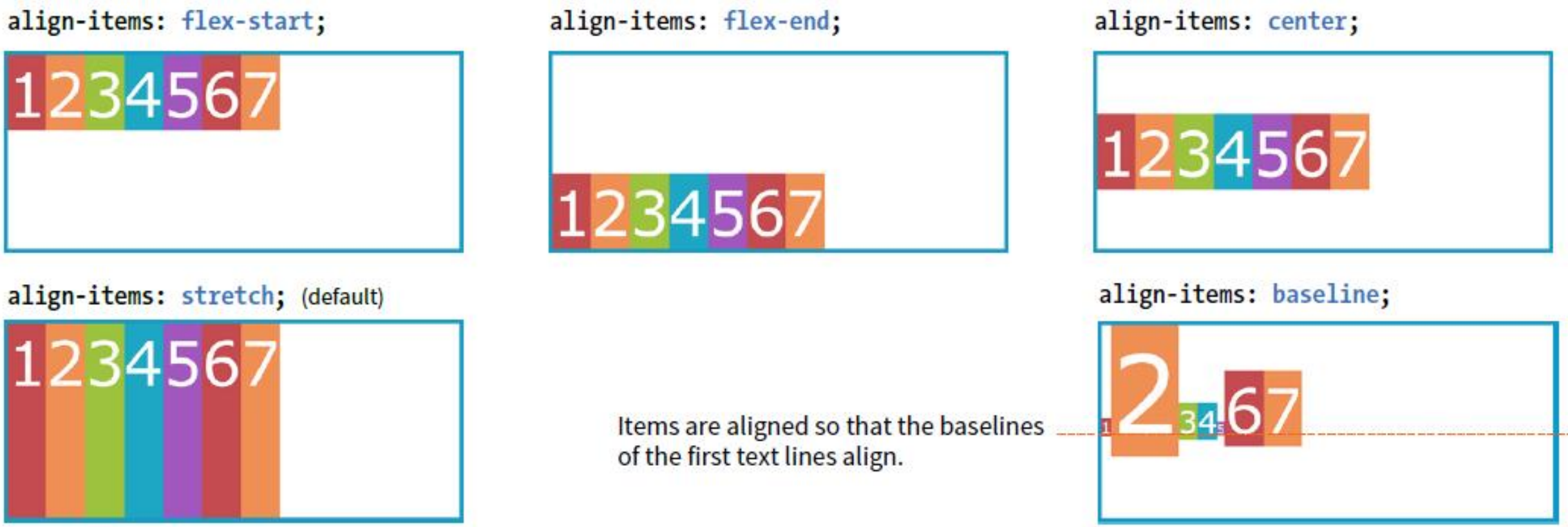
Aligning on the cross axis

- That takes care of arranging things on the main axis, but you may also want to play around with alignment on the cross axis (up and down when the direction is **row**, left and right if the direction is **column**). Cross-axis alignment and stretching is the job of the **align-items** property.

align-items

Values: flex-start | flex-end | center | baseline | stretch
Default: stretch
Applies to: flex containers
Inherits: no

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



Aligning on the cross axis

FIGURE 16-10. Aligning along the cross axis with **align-items**.

Self-Aligning for an Individual Element

- If you'd like one or more items to override the cross-axis setting, use the **align-self** property on the individual item element(s). This is the first property we've seen that applies to an *item*, not the container itself. **align-self** uses the same values as **align-items**; it just works on one item at a time.

align-self

Values: flex-start | flex-end | center | baseline | stretch
Default: stretch
Applies to: flex items
Inherits: no

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

Self-Aligning for an Individual Element

```
align-self: flex-end;
```

FIGURE 16-11. Use **align-self** to make one item override the cross-axis alignment of its container.

Aligning multiple lines

- The final alignment option, **align-content**, affects how multiple flex lines are spread out across the cross axis. This property applies only when **flex-wrap** is set to **wrap** or **wrap-reverse** and there are multiple lines to align. If the items are on a single line, it does nothing.

align-content

Values:	flex-start | flex-end | center | space-around | space-between | stretch
Default:	stretch
Applies to:	multi-line flex containers
Inherits:	no

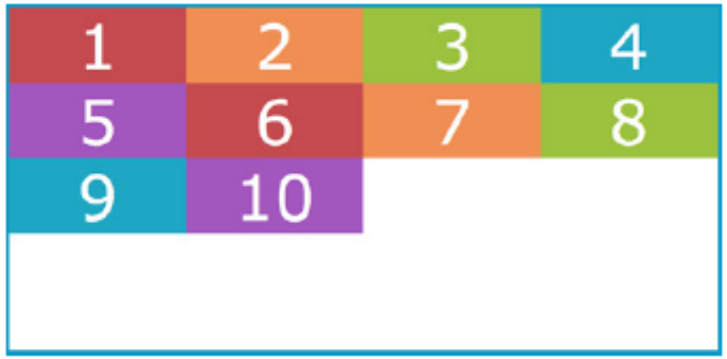
Aligning multiple lines

- Again, the **align-content** property applies to the flex container element. A height is required for the container as well, because without it the container would be just tall enough to accommodate the content and there would be no space left over.

```
#container {
   display: flex;
   flex-direction: row;
   flex-wrap: wrap;
   height: 350px;
   align-items: flex-start;
}

box {
   width: 25%;
}
```

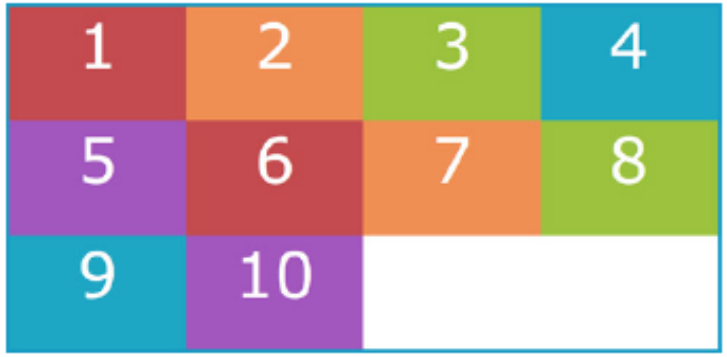
align-content: flex-start;



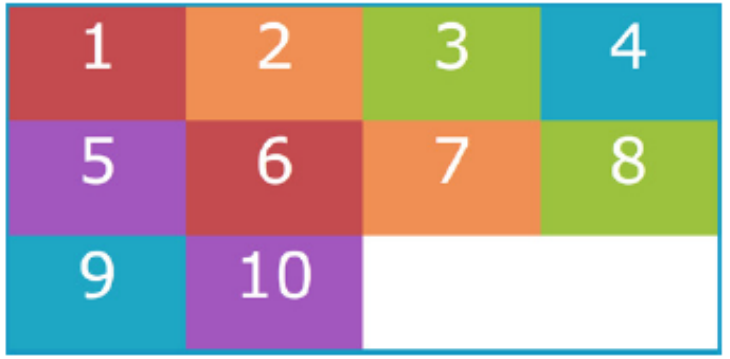
align-content: flex-end;

align-content: center;

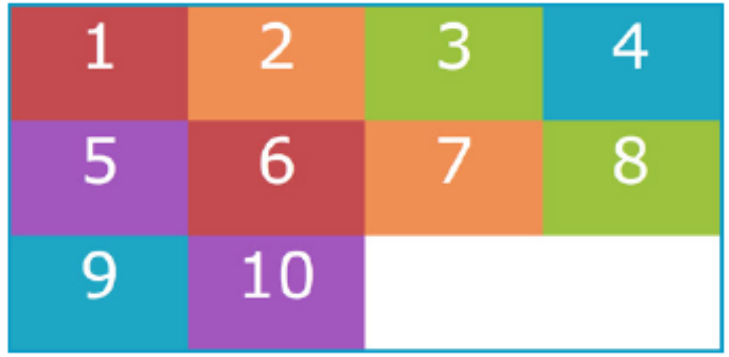
align-content: space-between;



align-content: space-around;



align-content: stretch; (default)



Aligning multiple lines

FIGURE 16-12. The **align-content** property distributes space around multiple flex lines. It has no effect when flex items are in a single line.

Aligning items with margins

THE MARKUP

```
<ul>
  <li class="logo"><img src="logo.png" alt="LoGoCo"></li>
  <li>About</li>
  <li>Blog</li>
  <li>Shop</li>
  <li>Contact</li>
</ul>
```

THE STYLES

```
ul {
  display: flex;
  align-items: center;
  background-color: #00af8f;
  list-style: none; /* removes bullets */
  padding: .5em;
  margin: 0;
}
li {
  margin: 0 1em;
}
li.logo {
  margin-right: auto;
}
```

FIGURE 16-13. Using a margin to adjust the space around flex items. In this example, the right margin of the logo item pushes the remaining items to the right.

Determining How Items “Flex” in the Container

- Flex is controlled with the **flex** property, which specifies how much an item can grow and shrink, and identifies its starting size. The full story is that **flex** is a shorthand property for **flex-grow**, **flex-shrink**, and **flex-basis**, but the spec strongly recommends that authors use the **flex** shorthand instead of individual properties in order to avoid conflicting default values and to ensure that authors consider all three aspects of **flex** for every instance.

flex

Values:	none | '*flex-grow flex-shrink flex-basis*'
Default:	0 1 auto
Applies to:	flex items
Inherits:	no

flex: *flex-grow flex-shrink flex-basis;*

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

Expanding items (flex-grow)

- The first value in the **flex** property specifies whether (and in what proportion) an item may stretch larger—in other words, its **flex-grow** value (see Note). By default it is set to 0, which means an item is not permitted to grow wider than the size of its content or its specified width. Because items do not expand by default, the alignment properties have the opportunity to go into effect. If the extra space was taken up inside items, alignment wouldn't work.

flex-grow

Values:	*number*
Default:	0
Applies to:	flex items
Inherits:	no

Expanding items (flex-grow)

THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
  <div class="box box5">5</div>
</div>
```

THE STYLES

```
.box {
  …
  flex: 1 1 auto;
}
```

`flex: 0 1 auto;` (prevents expansion)

`flex: 1 1 auto;` (allows expansion)

FIGURE 16-18. When **flex-grow** is set to 1, the extra space in the line is distributed into the items in equal portions, and they expand to fill the space at the same rate.

Expanding items (flex-grow)

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

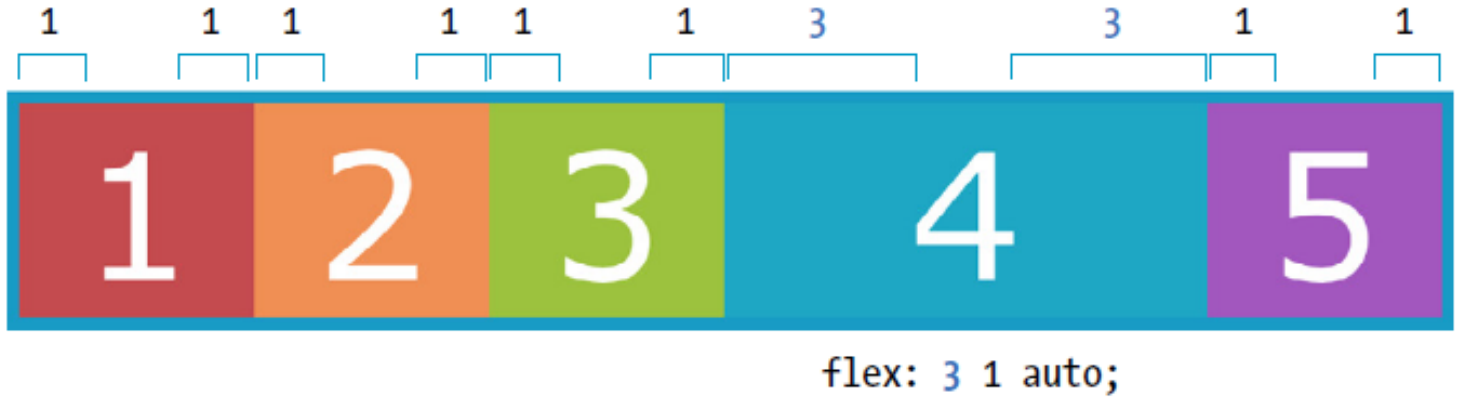


FIGURE 16-19. Assigning a different amount **flex-grow** to an individual item. Here "box4" was set to expand at three times the rate of the other items.

Squishing items (flex-shrink)

- The second **flex** property value, **flex-shrink**, kicks in when the container is not wide enough to contain the items, resulting in a space deficit. It essentially takes away some space from within the items, shrinking them to fit, according to a specified ratio.

flex-shrink

Values: *number*
Default: 1
Applies to: flex items
Inherits: no

Squishing items (flex-shrink)

- By default, the **flex-shrink** value is set to 1, which means if you do nothing, items shrink to fit at the same rate. When **flex-shrink** is 0, items are not permitted to shrink, and they may hang out of their container and out of view of the viewport. Finally, as in **flex-grow**, a higher integer works as a ratio. An item with a **flex-shrink** of 2 will shrink twice as fast as if it were set to 1.

- You will not generally need to specify a shrink ratio value. Just turning shrinking on (1) or off (0) should suffice.

Providing an initial size (flex-basis)

- The third **flex** value defines the starting size of the item before any wrapping, growing, or shrinking occurs (**flex-basis**). It may be used instead of the **width** property (or **height** property for columns) for flex items.

flex-basis

Values:	*length* | *percentage* | content | auto
Default:	auto
Applies to:	flex items
Inherits:	no

- In this example, the **flex-basis** of the boxes is set to 100 pixels (FIGURE 16-20). The items are allowed to shrink smaller to fit in the available space (**flex-shrink: 1**), but they are not allowed to grow any wider (**flex-grow: 0**) than 100 pixels, leaving extra space in the container.

```
box {
    flex: 0 1 100px;
}
```

Providing an initial size (flex-basis)

```
flex: 0 1 100px;
```

When the container is wide, the items will not grow wider than their `flex-basis` of 100 pixels because `flex-grow` is set to 0.

When the container is narrow, the items are allowed to shrink to fit (`flex-shrink: 1`).

FIGURE 16-20. Using **`flex-basis`** to set the starting width for items.

Absolute Versus Relative Flex

```
.box {
  /* applied to all boxes */
  flex: 1 0 0%;
}
.box1 {
  flex: 2; /* shortcut value for flex: 2 1 0px */
}
.box4 {
  flex: 3; /* shortcut value for flex: 3 1 0px */
}
```

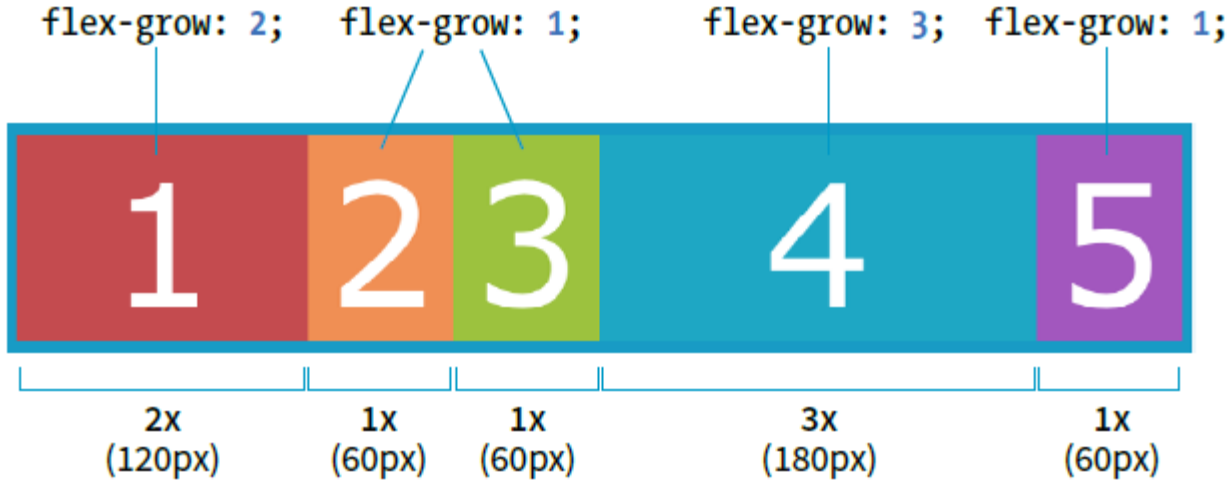


FIGURE 16-21. In absolute flex, boxes are sized according to the flex value ratios.

Changing the Order of Flex Items

- One of the killer features of Flexbox is the ability to display items in an order that differs from their order in the source (see the “When to Reorder (and When Not To)” sidebar). That means you can change the layout order of elements by using CSS alone. This is a powerful tool for responsive design, allowing content from later in a document to be moved up on smaller screens.
- To change the order of items, apply the **order** property to the particular item(s) you wish to move.

order

Values:	*integer*
Default:	0
Applies to:	flex items and absolutely positioned children of flex containers
Inherits:	no

Changing the Order of Flex Items

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

FIGURE 16-22. Changing the order of items with the **order** property. Setting **box3** to **order: 1** makes it display after the rest.

Changing the Order of Flex Items

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

FIGURE 16-23. Setting **box2** to **order: 1** as well makes it display after the items with the default order of 0.

Changing the Order of Flex Items

```
.box5 {
  order: -1
}
```

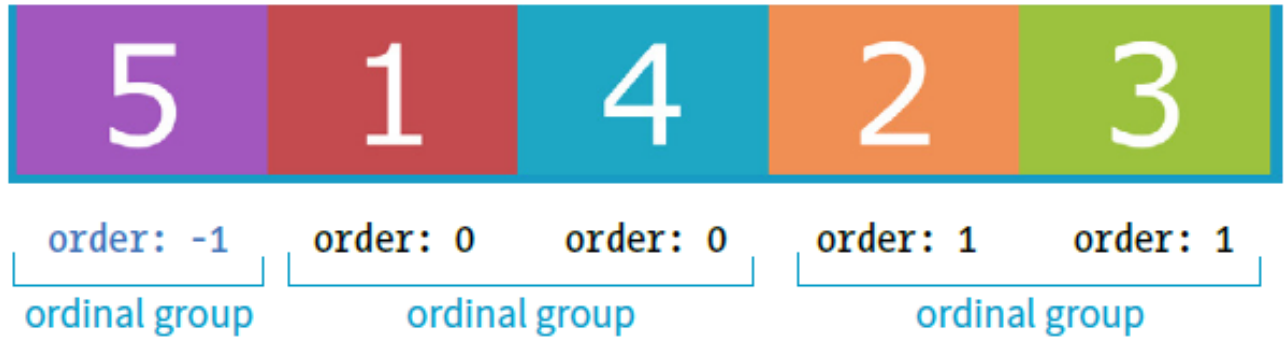


FIGURE 16-24. Negative values display before items with the default order of 0.

Changing the Order of Flex Items

- Now let's take a look at how we can use **order** for something more useful than moving little boxes around in a line. Here is a simple document with a header, a main section consisting of an article and two aside elements, and a footer:

```
<header>…</header>
<main>
    <article><h2>Where It's At</h2></article>
    <aside id="news"><h2>News></h2></aside>
    <aside id="contact"><h2>Contact</h2><aside>
</main>
<footer>…<footer>
```

Changing the Order of Flex Items

```
main {
    display: flex;
}
article {
    flex: 1 1 50%;
    order: 2;
}
#news {
    flex: 1 1 25%;
    order: 3;
}
#contact {
    flex: 1 1 25%;
    order: 1;
}
```

Changing the Order of Flex Items

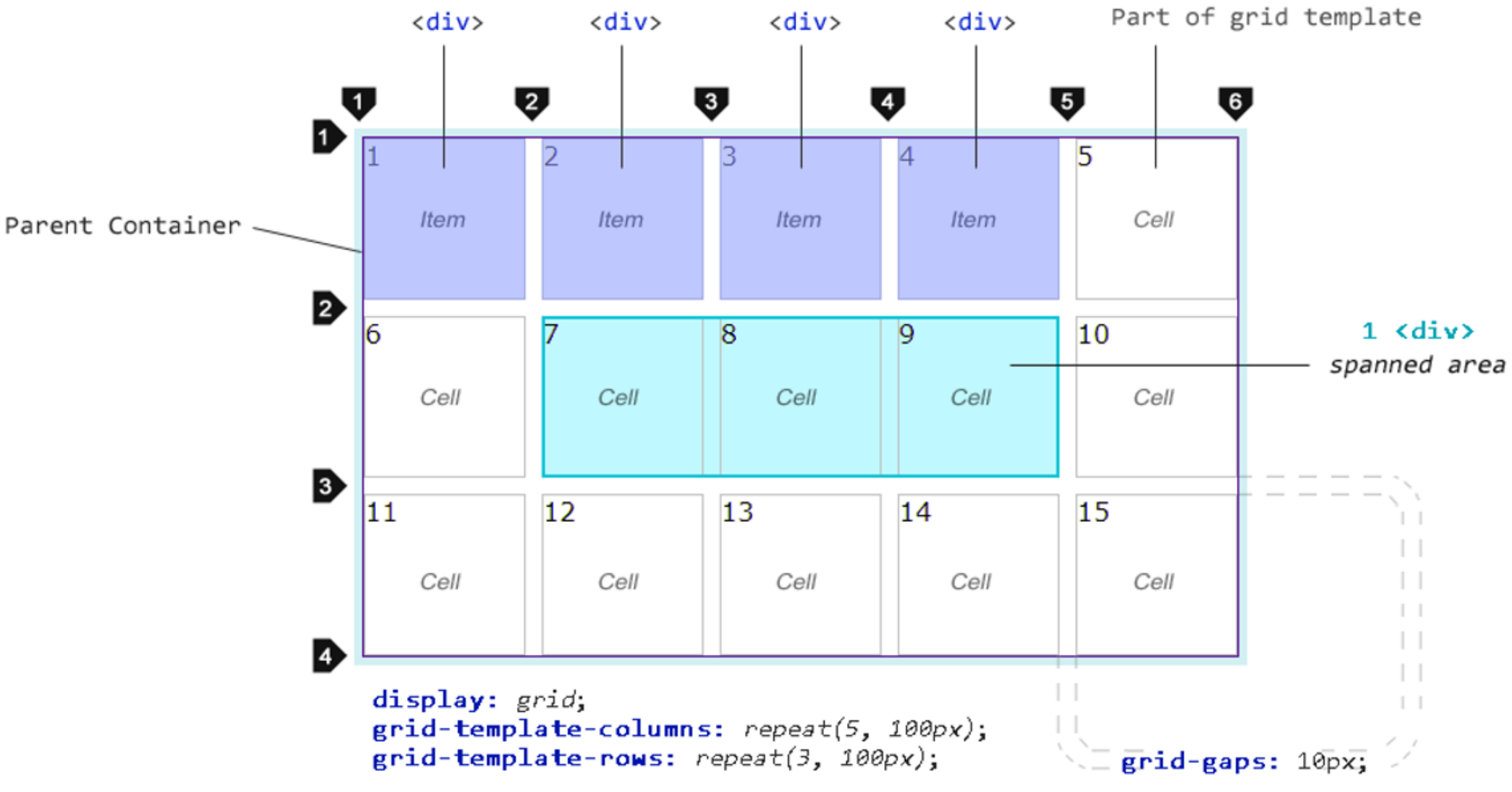
FIGURE 16-25. A columned layout using Flexbox.

Changing the Order of Flex Items

FIGURE 16-27. The Autoprefixer site converts standard Flexbox styles into all the styles needed for full browser support.

Grid Layout

SECTION 4



Part of grid template
1
2
3
4
5
6
Parent Container
Item
Item
Item
Item
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
Cell
1
spanned area
display: grid;
grid-template-columns: repeat(5, 100px);
grid-template-rows: repeat(3, 100px);
grid-gaps: 10px;

Re-creation of print jazz poster using grid

Re-creation of Die Neue Typography lecture invitation (1927) using grid

Overlap experiment with photos by Dorthea Lange

FIGURE 16-29. Examples of grid-based designs from Jen Simmons's "Experimental Layout Lab" page (*labs.jensimmons.com*).

How Grid Layout Works

The process for using the CSS Grid Layout Module is fundamentally simple:

1. **Use the display property to turn an element into a grid container.** The element's children automatically become grid items.
2. **Set up the columns and rows for the grid.** You can set them up explicitly and/or provide directions for how rows and columns should get created on the fly.
3. **Assign each grid item to an area on the grid.** If you don't assign them explicitly, they flow into the cells sequentially.

Grid Terminology

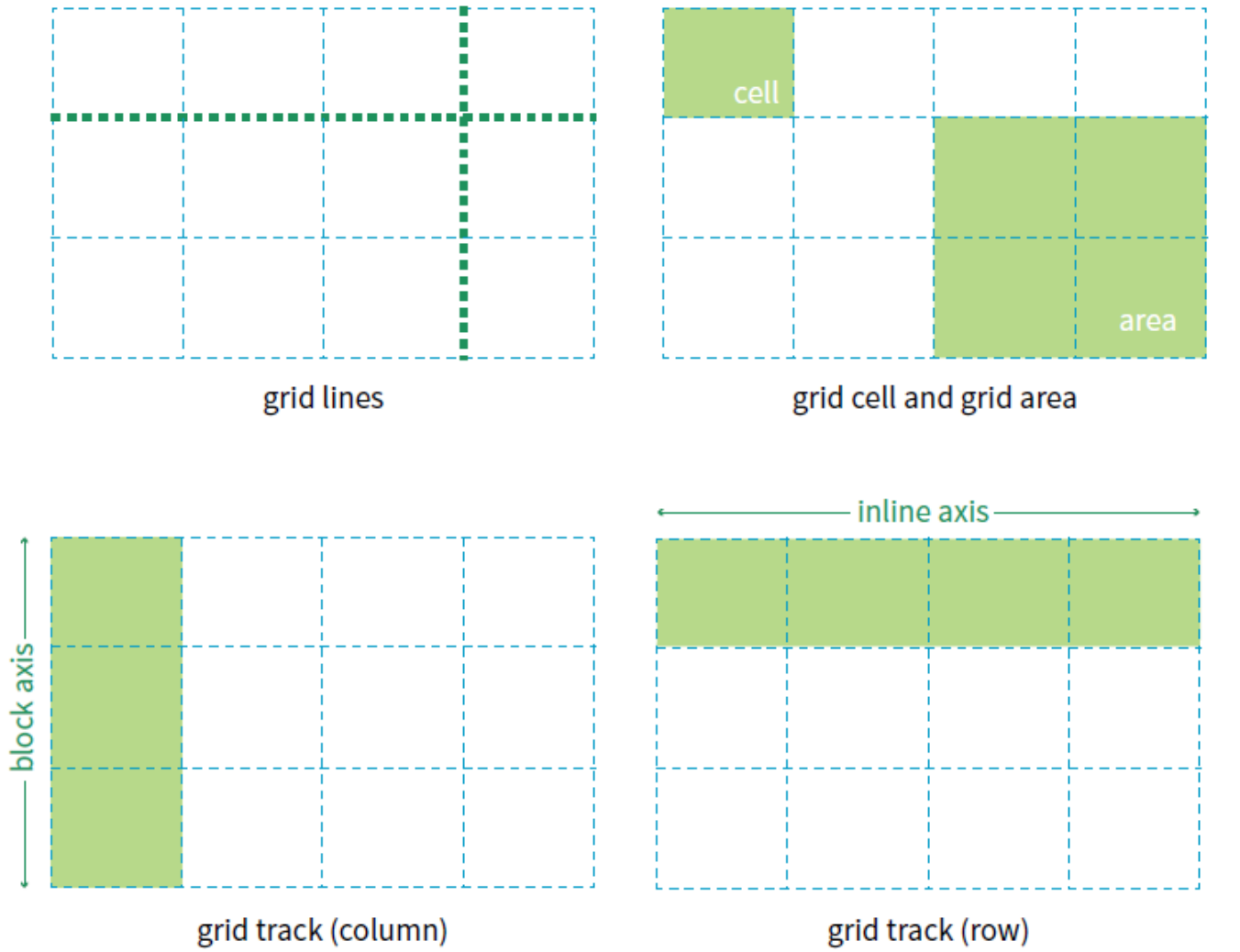


FIGURE 16-30. The parts of a CSS grid.

Grid Terminology

Grid line

The horizontal and vertical dividing lines of the grid are called grid lines.

Grid cell

The smallest unit of a grid is a grid cell, which is bordered by four adjacent grid lines with no grid lines running through it.

Grid area

A grid area is a rectangular area made up of one or more adjacent grid cells.

Grid track

The space between two adjacent grid lines is a grid track, which is a generic name for a grid column or a grid row. Grid columns are said to go along the block axis, which is vertical (as block elements are stacked) for languages written horizontally. Grid rows follow the inline (horizontal) axis.

Declaring Grid Display

THE MARKUP

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

THE STYLES

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

Setting Up the Grid

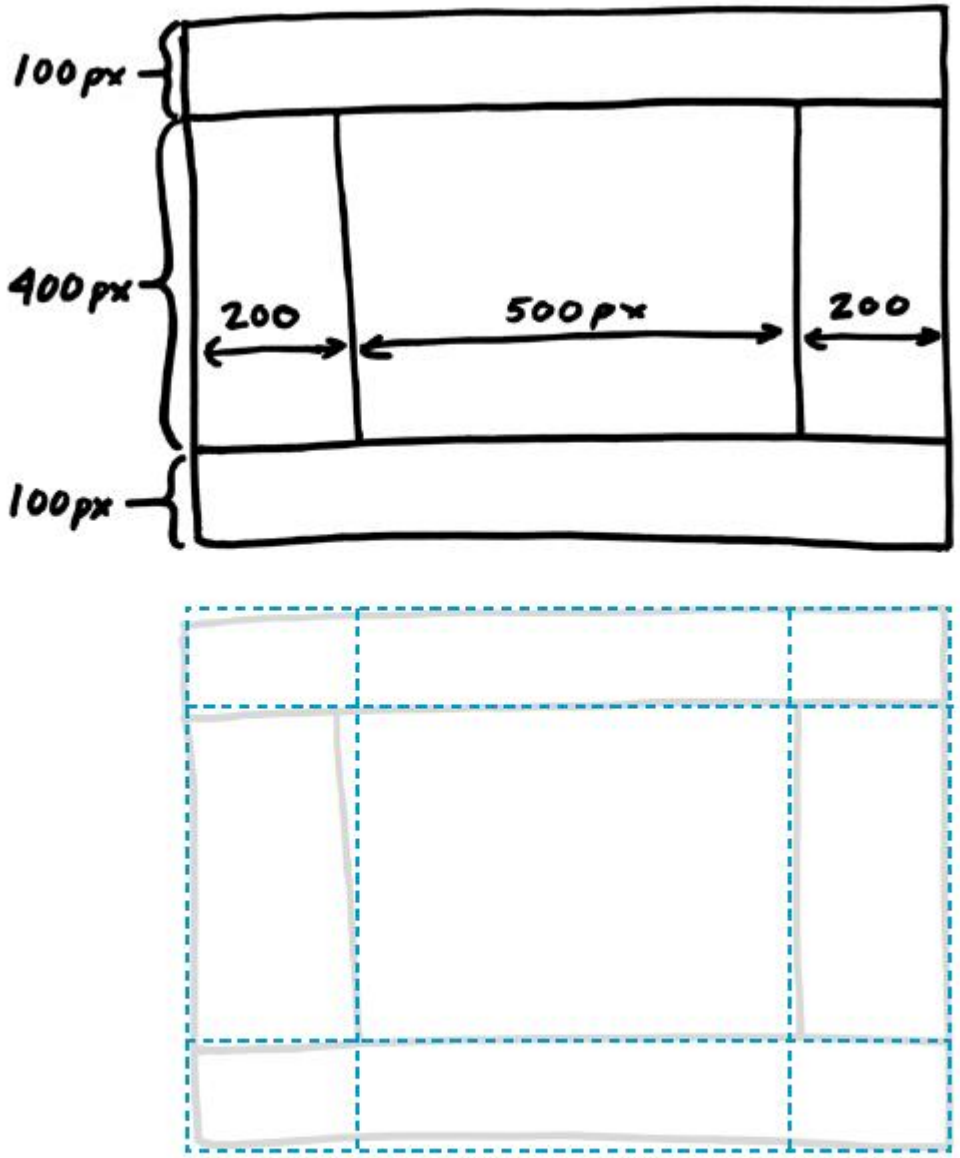


FIGURE 16-31. A rough sketch for my grid-based page layout. The dotted lines in the bottom image show how many rows and columns the grid requires to create the layout structure.

Defining grid tracks

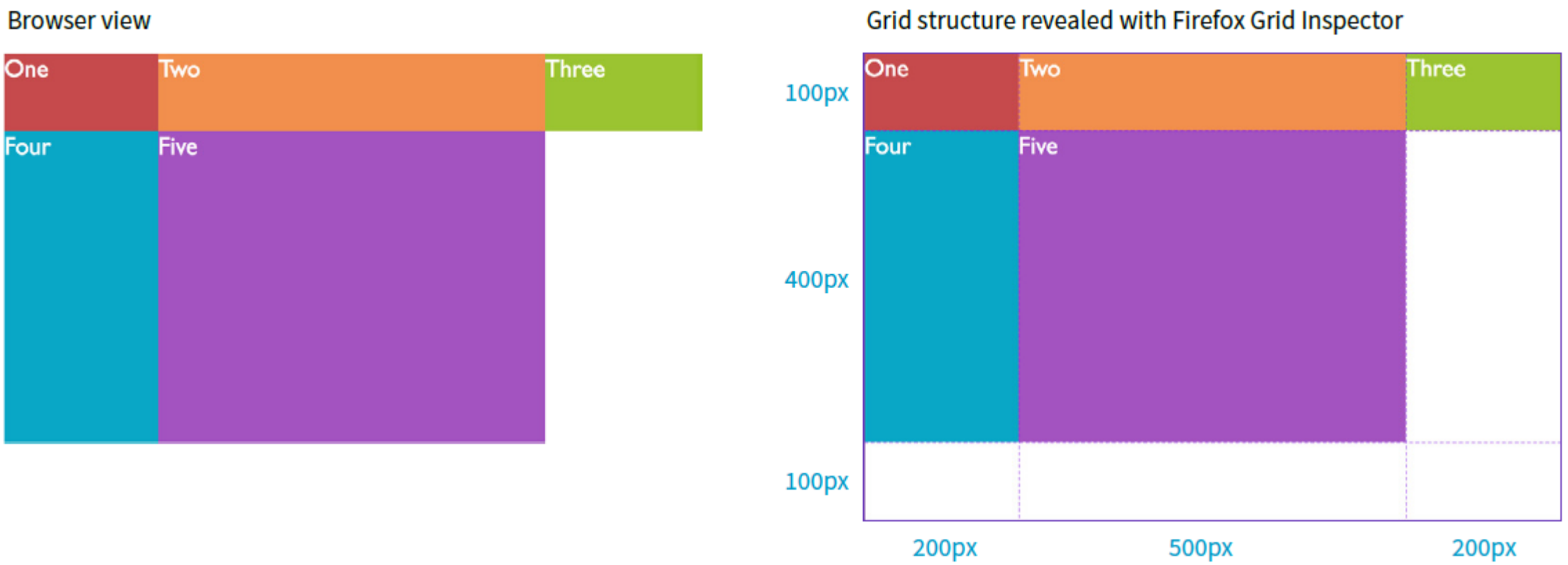
- To set up a grid in CSS, specify the height of each row and the width of each column (see Note) with the template properties, **grid-template-rows** and **grid-template-columns**, which get applied to the container element.

grid-template-rows
grid-template-columns

Values:	none | *list of track sizes and optional line names*
Default:	none
Applies to:	grid containers
Inherits:	no

Grid track sizes

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



Grid track sizes

FIGURE 16-32. By default, grid items flow into the grid cells by rows.

Grid line numbers and names

- When the browser creates a grid, it also automatically assigns each grid line a number that you can reference when positioning items. The grid line at the start of the grid track is 1, and lines are numbered sequentially from there. FIGURE 16-33 shows how the grid lines are numbered for our sample grid.
- The lines are numbered from the end of tracks as well, starting with –1, and numbers count back from there (–2, –3, etc.), as shown by the gray numbers in FIGURE 16-33. Being able to target the end of a row or column without counting lines (or even knowing how many rows or columns there are) is a handy feature. You'll come to love that –1.

Grid line numbers and names

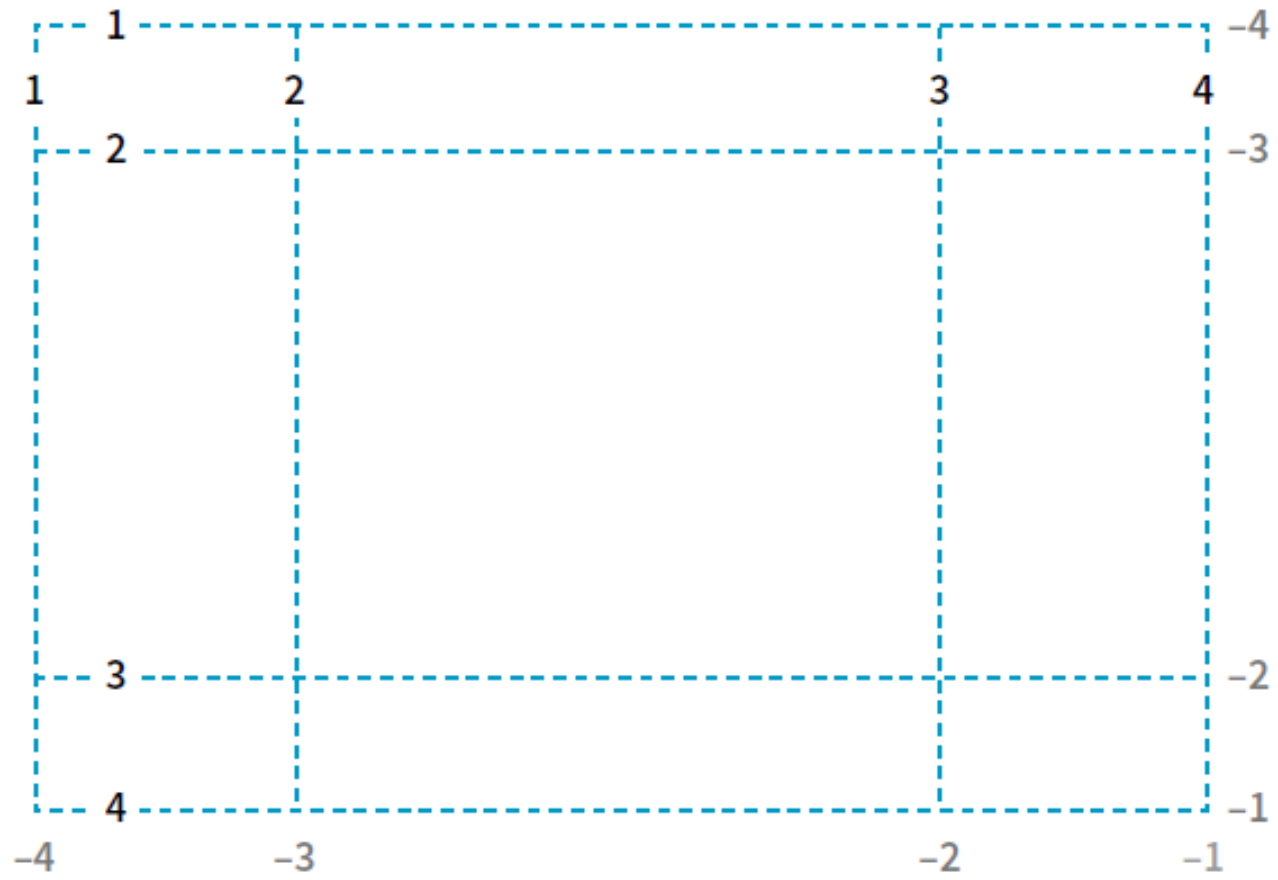


FIGURE 16-33. Grid lines are assigned numbers automatically.

Specifying track size values

- I provided all of the track sizes in my example in specific pixel lengths to make them easy to visualize, but fixed sizes are one of many options. They also don't offer the kind of flexibility required in our multi-device world.
- The **Grid Layout Module** provides a whole bunch of ways to specify track sizes, including old standbys like lengths (e.g., pixels or ems) and percentage values, but also some newer and Grid-specific values. I'm going to give you quick introductions to some useful Grid-specific values: the fr unit, the minmax() function, auto, and the content-based values min-content/max-content.
- We'll also look at functions that allow you to set up a repeating pattern of track widths: the repeat() function with optional auto-fill and auto-fit values.

Specifying track size values

Fractional units (flex factor)

- The Grid-specific fractional unit (**fr**) allows developers to create track widths that expand and contract depending on available space. To go back to the example, if I change the middle column from **500px** to **1fr**, the browser assigns all leftover space (after the 200-pixel column tracks are accommodated) to that column track (FIGURE 16-34).

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

Track Size Values

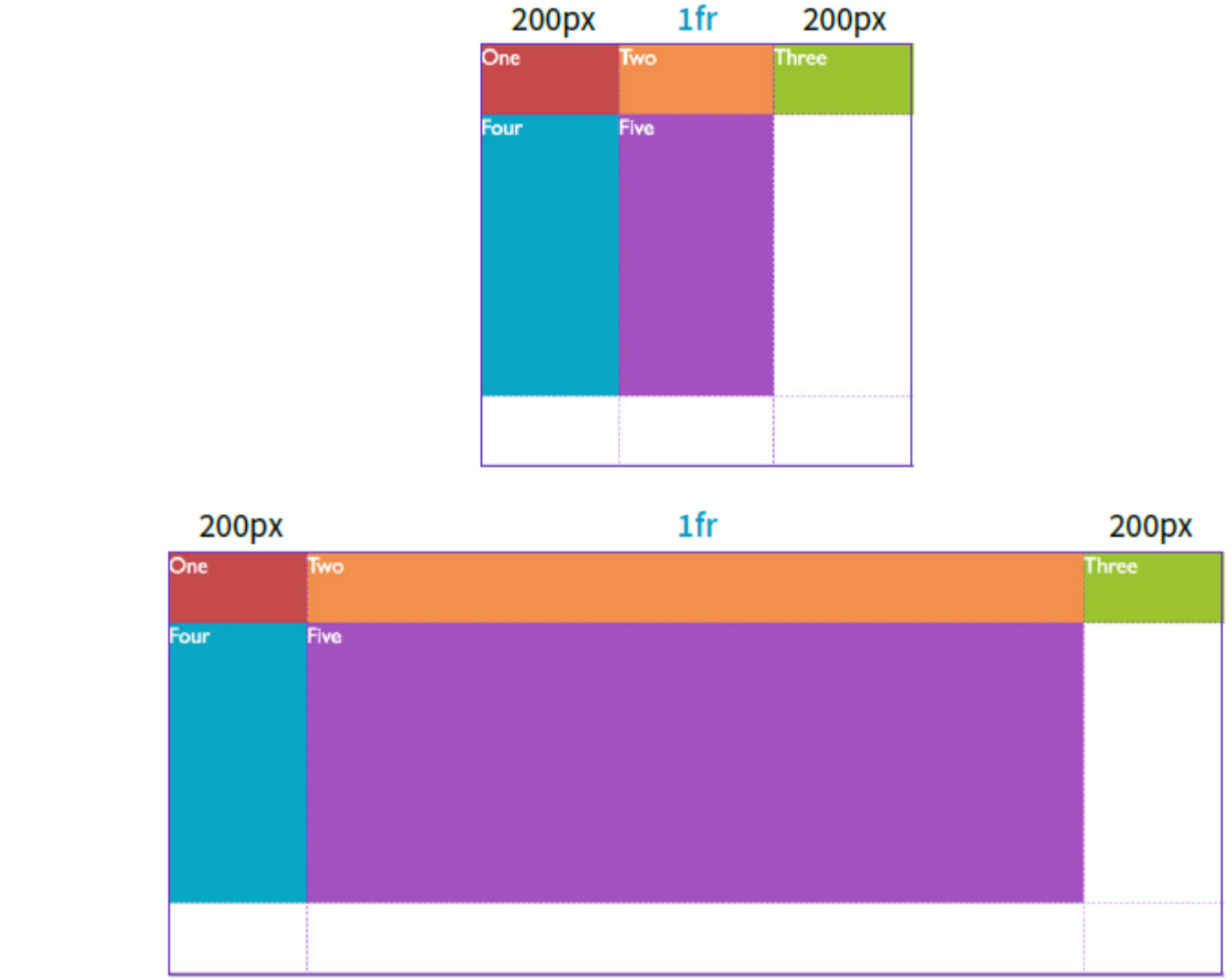


FIGURE 16-34. When the middle column has a track size of 1fr, it takes up the remaining space in the browser window and flexes to adapt to the browser width.

Track Size Values

The Grid specification provides the following values for the **grid-template-*** properties:

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

Track Size Values

Fractional units (flex factor)

- The **fr** unit is great for combining fixed and flexible track widths, but I could also use all **fr** units to give all the columns proportional widths. In this example, all of the column widths flex according to the available browser width, but the middle column will always be twice the width of the side columns (see Note).

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

Track Size Values

Fractional units (flex factor)

Minimum and maximum size range

You can constrict the size range of a track by setting its minimum and maximumwidths using the **minmax()** function in place of a specific track size.

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

This rule sets the middle column to a width that is at least 15em but never wider than 45em. This method allows for flexibility but allows the author to set limits.

Track Size Values

Fractional units (flex factor)

Content-based sizing

The **min-content**, **max-content**, and **auto** values size the track based on the size of the content within it (FIGURE 16-35). The **min-content** value is the *smallest* that track can get without overflowing (by default, unless overridden by an explicit **min-width**). It is equivalent to the "largest unbreakable bit of content"—in other words, the width of the longest word or widest image. It may not be useful for items that contain normal paragraphs, but it may be useful in some cases when you don't want the track larger than it needs to be. This example establishes three columns, with the right column sized just wide enough to hold the longest word or image:

```
grid-template-columns: 50px 1fr
                       min-content;
```

Repeating track sizes

- Say you have a grid that has 10 columns with alternating column widths, like so:

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

- That's kind of a bummer to have to type out (I know, I just did it), so the fine folks at the W3C have provided a nice shortcut in the form of the **repeat()** function. In the previous example, the pattern "20px 1fr" repeats five times, which can be written as follows:

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

Repeating track sizes

- Much better, isn't it? The first number indicates the number of repetitions, and the track sizes after the comma provide the pattern. You can use the repeat() notation in a longer sequence of track sizes—for example, if those 10 columns are sandwiched between two 200-pixel-wide columns at the start and end:

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

- You can also provide grid line names before and/or after each track size, and those names will be repeated in the pattern:

```
grid-template-rows:
  repeat(4, [date] 5em [event] 1fr);
```

Repeating track sizes

auto-fill and auto-fit

- In the previous repeat() examples, we told the browser how many times to repeat the provided pattern. You can also let the browser figure it out itself based on the available space by using the auto-fill and auto-fit values instead of an integer in repeat().

- For example, if I specify

```
grid-template-rows:
    repeat(auto-fill, 10em);
```

- and the grid container is 35em tall, then the browser creates a row every 10 ems until it runs out of room, resulting in three rows. Even if there is only enough content to fill the first row, all three rows are created and the space is held in the layout.

Repeating track sizes

auto-fill and auto-fit

- The **auto-fit** value works similarly, except any tracks that do not have content get dropped from the layout. If there is leftover space, it is distributed according to the vertical (**align-content**) an horizontal (**justify-content**) alignment values provided (we'll discuss alignment later in this section).

Grid Area

SECTION 5

Defining grid areas

- To assign names to grid areas, use the **grid-template-areas** property.

grid-template-areas

Values: none | *series of area names*
Default: none
Applies to: grid containers
Inherits: no

Defining grid areas

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

Defining grid areas

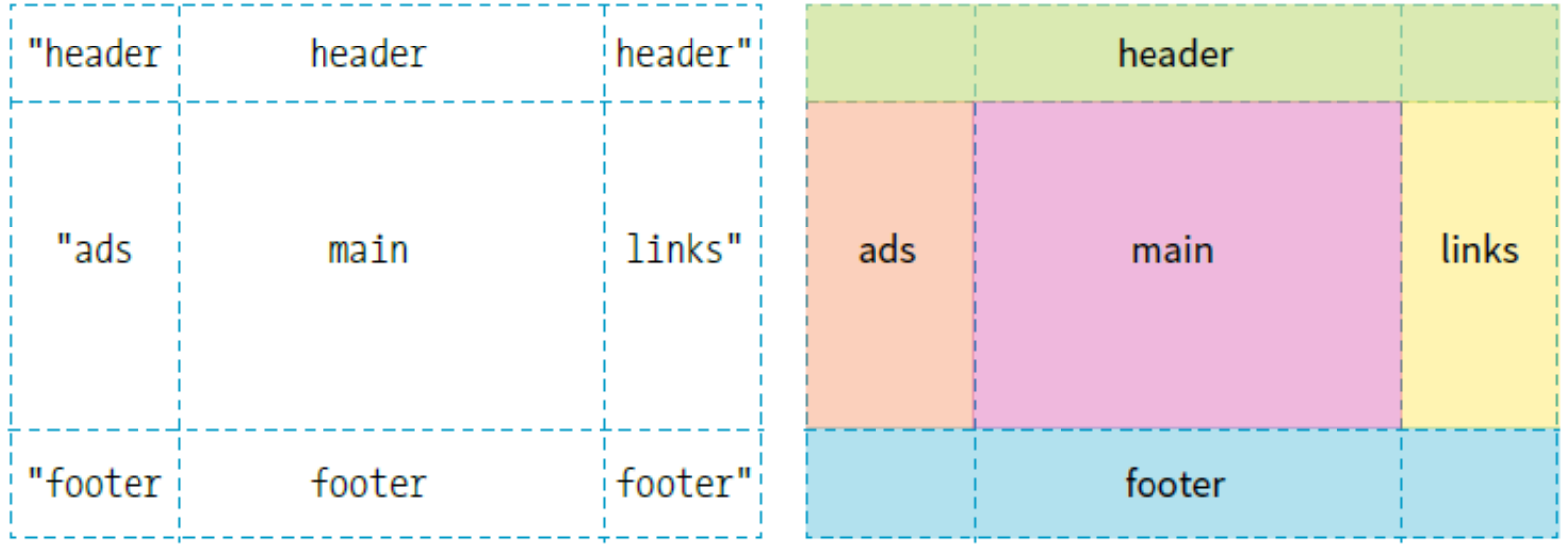


FIGURE 16-36. When neighboring cells have the same name, they form a named area that can be referenced later.

Placing Grid Items

Positioning using lines

- One method for describing a grid item's location on the grid is to specify the four lines bordering the target grid area with four properties that specify the start and end row lines and the start and end column lines. Apply these properties to the individual grid item element you are positioning.

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

Values:	auto | *grid line* | span *number* | span *'line name'* | *number 'line name'*
Default:	auto
Applies to:	grid items
Inherits:	no

Placing Grid Items

Positioning using lines

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

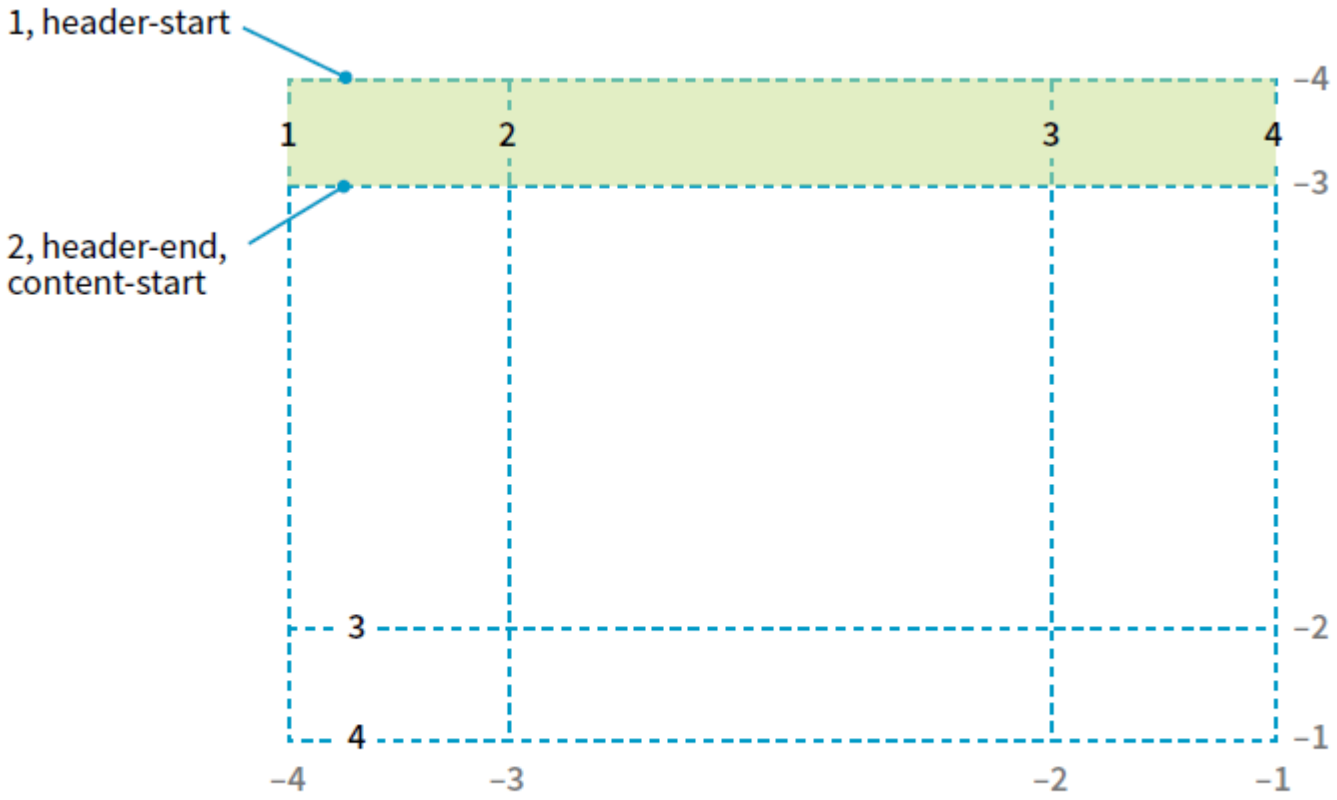


FIGURE 16-39. Positioning a grid item across the top row track in our sample grid.

Placing Grid Items

Positioning using lines

grid-row
grid-column

Values:	*start line / end line*
Default:	see individual properties
Applies to:	grid items
Inherits:	no

These properties combine the ***-start** and ***-end** properties into a single declaration. The start and end line values are separated by a slash (/). With the shorthand, I can shorten my example to the following two declarations. Any of the methods for referring to lines work in the shorthand values.

```
#one {
   grid-row: 1 / 2;
   grid-column: 1 / span 3;
}
```

Positioning by area

- The other way to position an item on a grid is to tell it go into one of the named areas by using the **grid-area** property.

grid-area

Values: *area name* | *1 to 4 line identifiers*
Default: see individual properties
Applies to: grid items
Inherits: no

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

Positioning by area

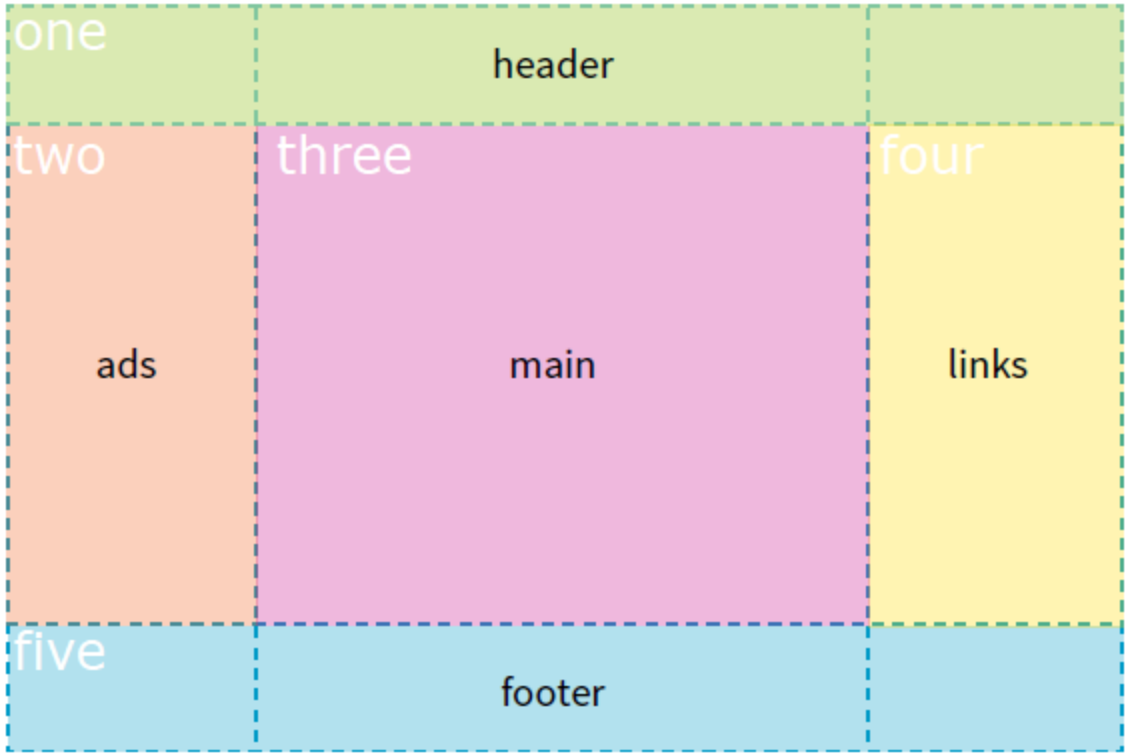


FIGURE 16-40. Assigning grid items by area names.

Implicit Grid Behavior

- By default, any row or column automatically added to a grid will have the size **auto**, sized just large enough to accommodate the height or width of the contents. If you want to give implicit rows and columns specific dimensions, such as to match a rhythm established elsewhere in the grid, use the **gridauto-*** properties.

grid-auto-rows
grid-auto-columns

Values:	*list of track sizes*
Default:	auto
Applies to:	grid containers
Inherits:	no

Implicit Grid Behavior

THE MARKUP

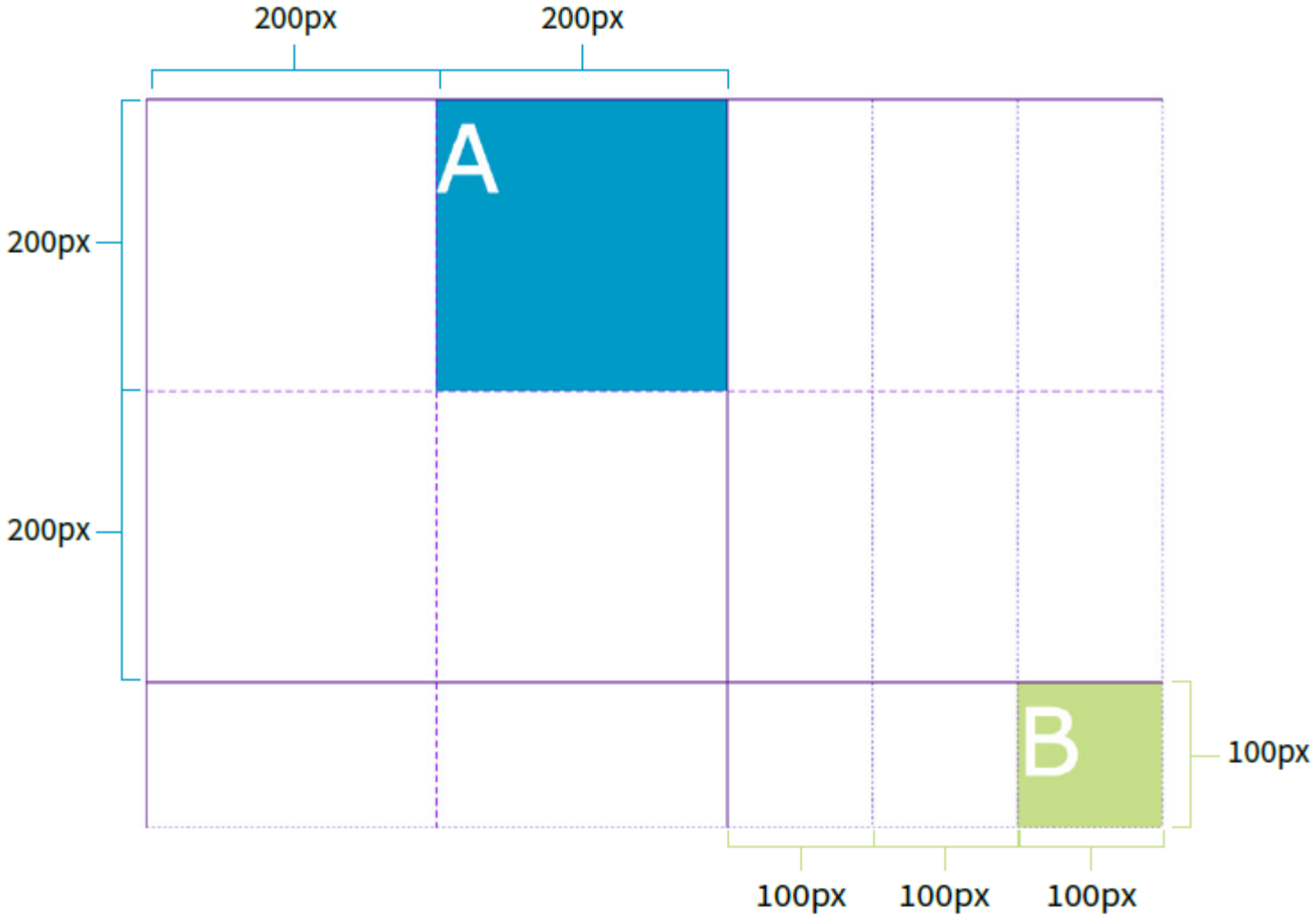
```
<div id="littlegrid">
    <div id="A">A</div>
    <div id="B">B</div>
</div>
```

THE STYLES

```
#littlegrid {
    display: grid;
    grid-template-columns: 200px 200px;
    grid-template-rows: 200px 200px;
    grid-auto-columns: 100px;
    grid-auto-rows: 100px;
}
#A {
    grid-row: 1 / 2;
    grid-column: 2 / 3;
}
#B {
    grid-row: 3 / 4;
    grid-column: 5 / 6;
}
```

Implicit Grid Behavior

The grid has two explicitly defined rows and columns at 200 pixels wide each.



Rows and column tracks are added automatically as needed. They are sized as specified by `grid-auto-rows` and `grid-auto-columns` (100 pixels).

FIGURE 16-42. Browsers generate rows and columns automatically to place grid items that don't fit the defined grid.

Flow direction and density

grid-auto-flow

Values:	row or column | dense *(optional)*
Default:	row
Applies to:	grid containers
Inherits:	no

- Use **grid-auto-flow** to specify whether you'd like items to flow in by row or column. The default flow follows the writing direction of the document (leftto-right and top-to-bottom for English and other left-to-right languages).

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

Flow direction and density

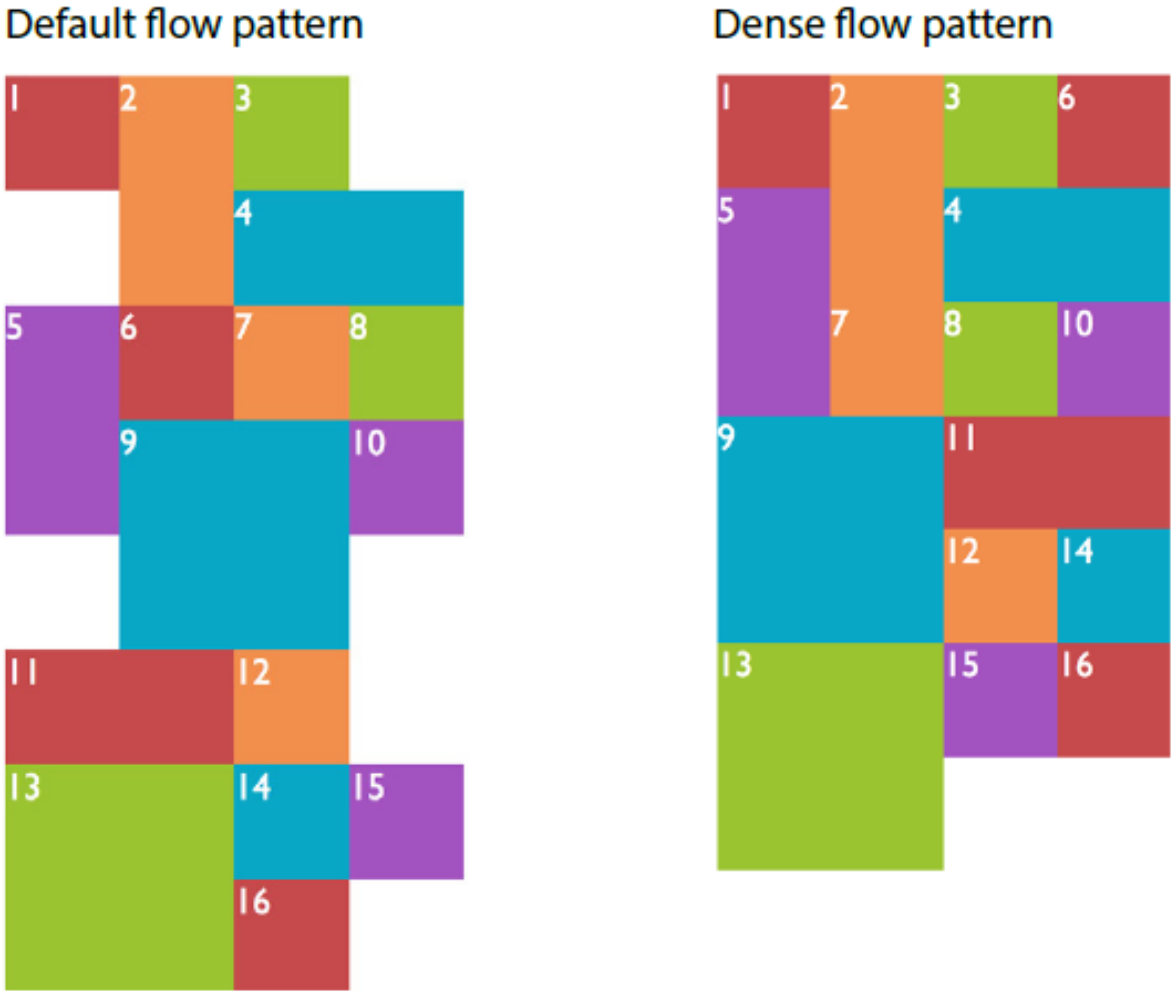


FIGURE 16-43. Comparison of default and dense auto-flow modes.

Grid Alignment

SECTION 6

Spacing and Alignment

Spacing between tracks (gutters)

grid-row-gap
grid-column-gap

Values: *length (must not be negative)*
Default: 0
Applies to: grid containers
Inherits: no

grid-gap

Values: *grid-row-gap grid-column-gap*
Default: 0 0
Applies to: grid containers
Inherits: no

Grid and item alignment


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

FIGURE 16-44. Grid gaps add gutter spaces between tracks.

Aligning individual items

justify-self

Values: start | end | center | left | right | self-start | self-end | stretch | normal | auto
Default: auto (looks at the value for justify-items, which defaults to normal)
Applies to: grid items
Inherits: no

align-self

Values: start | end | center | left | right | self-start | self-end | stretch | normal | auto
Default: auto (looks at the value for align-items)
Applies to: grid items
Inherits: no

Aligning individual items

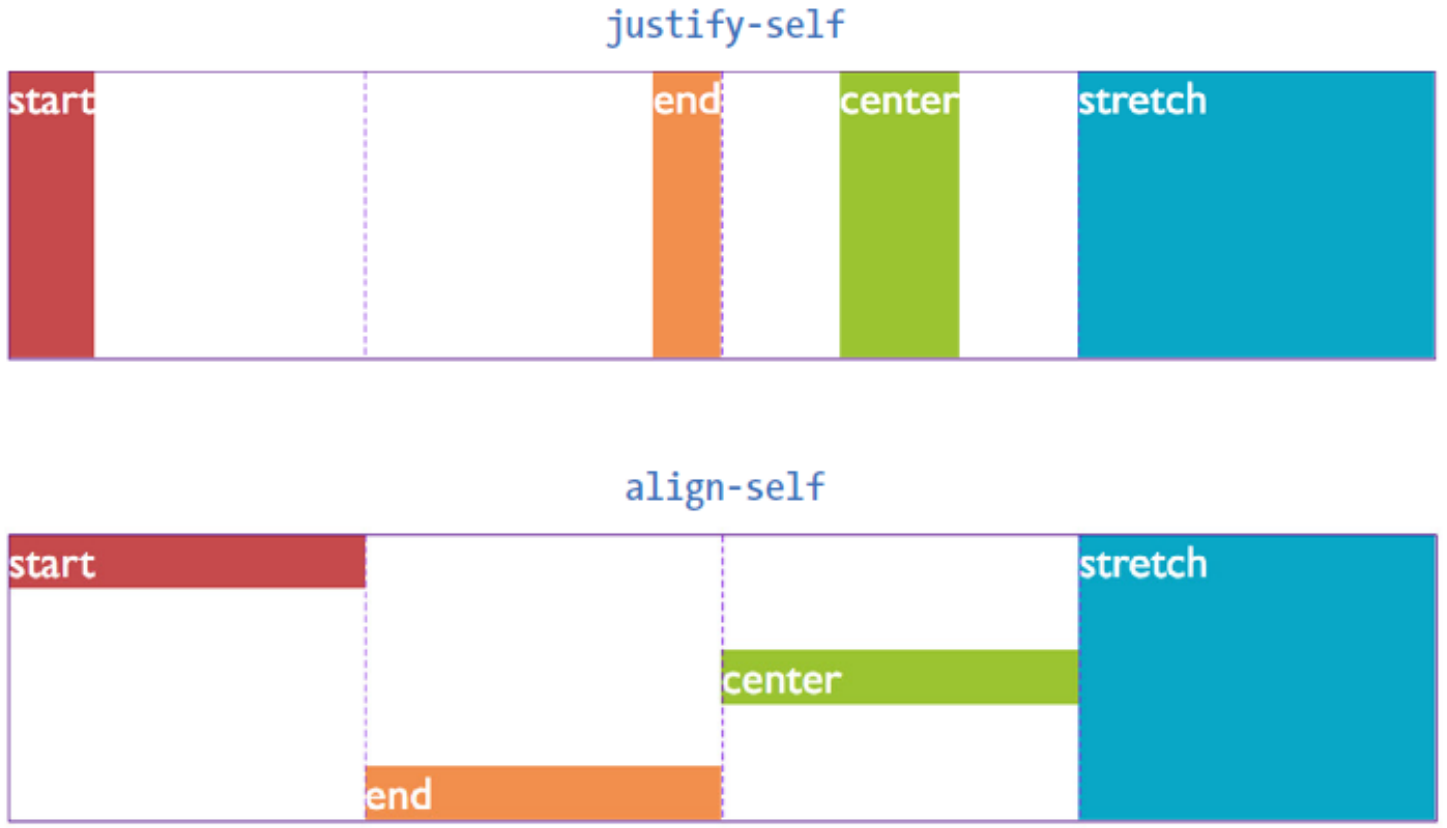


FIGURE 16-45. Values for **justify-self** and **align-self** for aligning a grid item within its respective grid area. These values have the same use in the **justify-items** and **align-items** properties that are used to align all the items in the grid.

Aligning all the items in a grid

justify-items

Values:	start | end | center | left | right | self-start | self-end | stretch |normal
Default:	normal (stretch for non-replaced elements; start for replaced elements)
Applies to:	grid containers
Inherits:	no

align-items

Values:	start | end | center | left | right | self-start | self-end | stretch | normal
Default:	normal (stretch for non-replaced elements; start for replaced elements)
Applies to:	grid containers
Inherits:	no

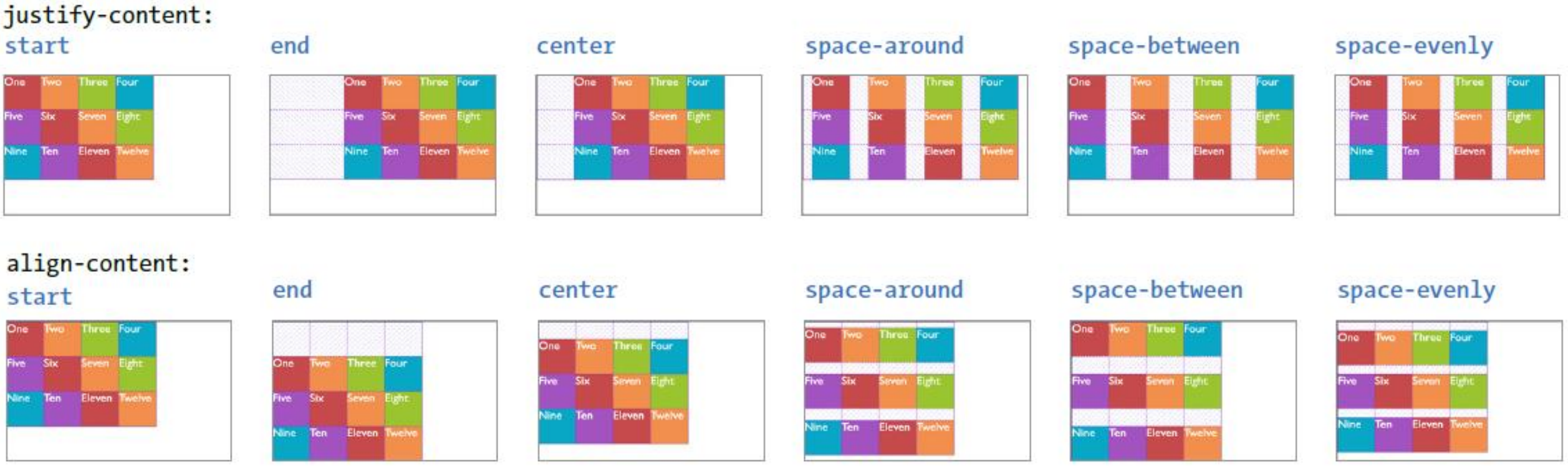
Aligning tracks in the grid container

justify-content

Values: start | end | left | right | center | stretch | space-around | space-between | space-evenly
Default: start
Applies to: grid containers
Inherits: no

align-content

Values: start | end | left | right | center | stretch | space-around | space-between | space-evenly
Default: start
Applies to: grid containers
Inherits: no



Aligning tracks in the grid container

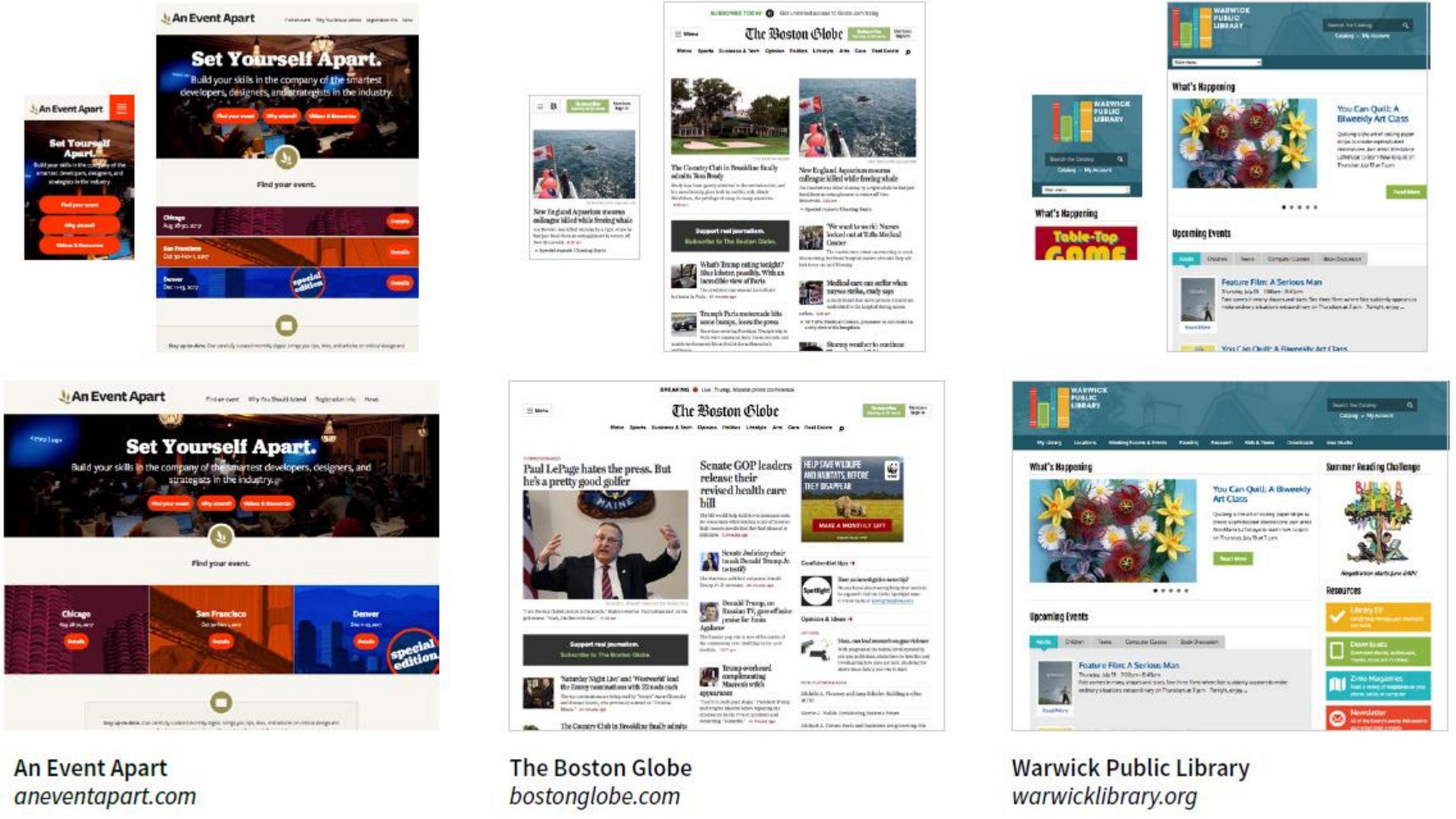
FIGURE 16-46. The **justify-content** and **align-content** properties distribute extra space in the container.

Responsive Web Design

SECTION 7

Responsive Web Design

- Responsive Web Design (or RWD) is a design and production approach that allows a website to be comfortably viewed and used on all manner of devices. The core principle is that all devices get the same HTML source, located at the same URL, but different styles are applied based on the viewport size to rearrange components and optimize usability. FIGURE 17-1 shows examples of responsive sites as they might appear on a smartphone, tablet, and desktop, but it is important to keep in mind that these sites are designed to work well on the continuum of every screen width in between.



An Event Apart
aneventapart.com

The Boston Globe
bostonglobe.com

Warwick Public Library
warwicklibrary.org

FIGURE 17-1. Examples of responsive sites that adapt to fit small, medium, and large screens and all sizes in between.

The Responsive Recipe

The technique has three core components:

A flexible grid

Rather than remaining at a static width, responsive sites use methods that allow them to squeeze and flow into the available browser space.

Flexible images

Images and other embedded media need to be able to scale to fit their containing elements.

CSS media queries

Media queries give us a way to deliver sets of rules only to devices that meet certain criteria, such as width and orientation.

Setting the Viewport

- To fit standard websites onto small screens, mobile browsers render the page on a canvas called the viewport and then shrink that viewport down to fit the width of the screen (device width). For example, on iPhones, mobile Safari sets the viewport width to 980 points (see Note), so a web page is rendered as though it were on a desktop browser window set to 980 pixels wide.
- That rendering gets shrunk down to the width of the screen (ranging from 320 to 414 points, depending on the iPhone model), cramming a lot of information into a tiny space.

Setting the Viewport

- Mobile Safari introduced the viewport **meta** element, which allows us to define the size of that initial viewport. Soon, the other mobile browsers followed suit. The following **meta** element, which goes in the **head** of the HTML document, tells the browser to set the width of the viewport equal to the width of the device screen (width=device-width), whatever that happens to be (FIGURE 17-2). The initial-scale value sets the zoom level to 1 (100%).

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

Setting the Viewport

- With the viewport meta element in place, if the device's screen is 320 pixels wide, the rendering viewport on that device will also be 320 pixels across (not 980) and will appear on the screen at full size. That is the width we test for with media queries, so setting the viewport is a crucial first step.

Setting the Viewport

By default, the viewport shrinks to the size of the screen.

With the viewport meta tag, the viewport is created at the same size as the screen.

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

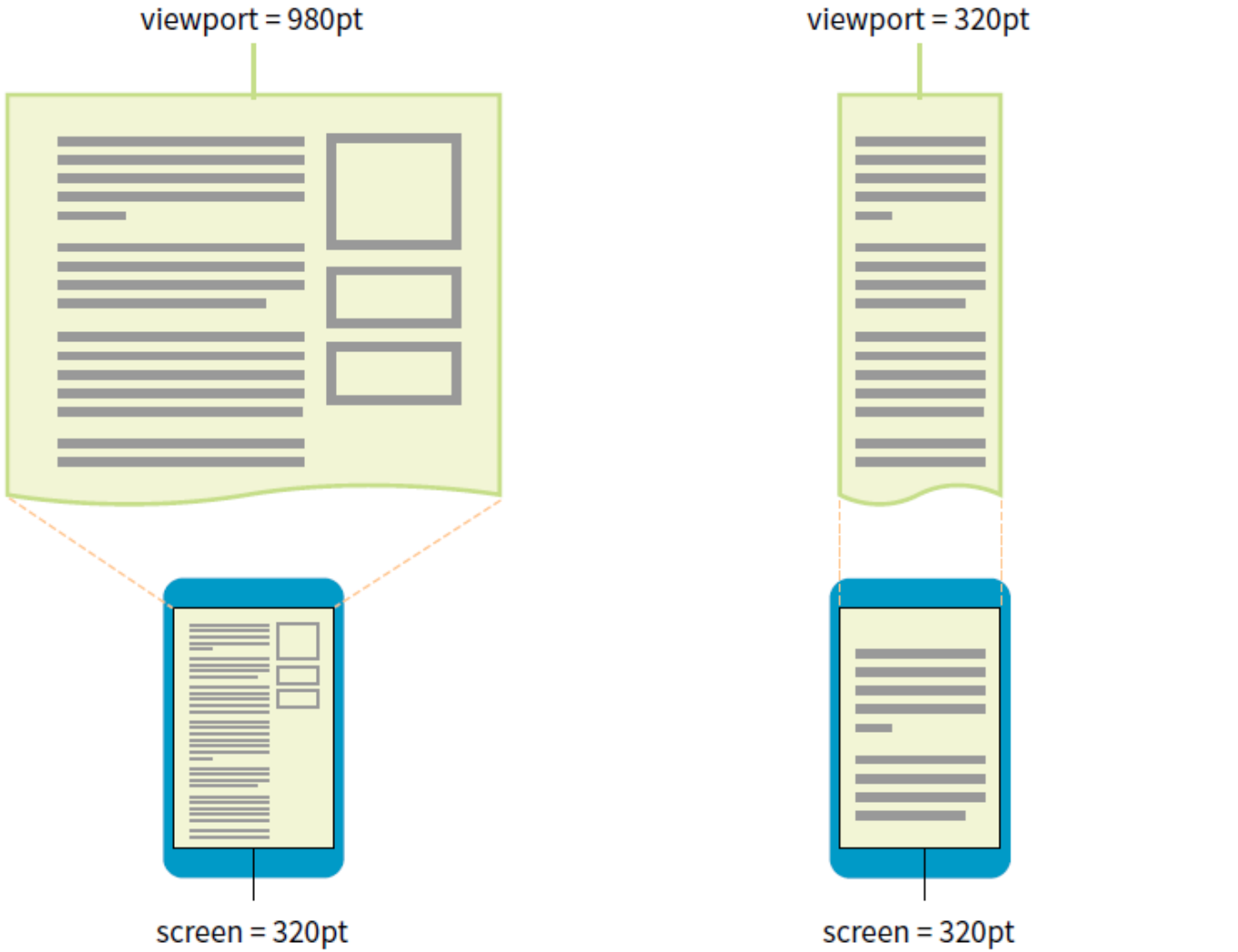


FIGURE 17-2. The viewport **meta** element matches the resolution of the device's browser viewport to the resolution of its screen.

Flexible Grids (Fluid Layouts)

In the **Flexbox** and **Grid** discussions in the previous chapter, we saw examples of items expanding and contracting to fill the available space of their containers.

That fluidity is exactly the sort of behavior you need to make content neatly fit a wide range of viewport sizes. Fluid layouts (or “flexible grids,” as Ethan Marcotte calls them in his article and book) are the foundation of responsive design.

Page Designs

SECTION 7

Fixed

Fluid

Adaptive

Responsive

Fluid layouts fill the viewport proportionally.

w3c.org

Fixed layouts stay the same size and may get cut off or leave extra space.

kexp.org

FIGURE 17-3. Fluid and fixed layout examples.

Making Images Flexible

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

FIGURE 17-4. Setting the **max-width** of inline images allows them to shrink to fit available space but not grow larger than actual size.

Responsive images

- If you think back to our discussion of responsive images in Chapter 7, Adding Images, you'll remember that there is some elbow grease required to avoid serving unnecessarily large images to small devices as well as making sure large, high-density monitors get high-resolution images that shine.
- **Choosing the best image size for performance is part of the responsive design process**, but we won't be concentrating on that in this chapter.

Other embedded media

- Videos and other embedded media (using object or embed elements) also need to scale down in a responsive environment. Unfortunately, videos do not retain their intrinsic ratios when the width is scaled down, so there are a few more hoops to jump through to get good results. Thierry Koblentz documents one strategy nicely in his article “Creating Intrinsic Ratios for Video” at *www.alistapart.com/articles/creating-intrinsic-ratios-for-video*.
- There is also a JavaScript plug-in called FitVids.js (created by Chris Coyier and the folks at Paravel) that automates Koblentz’s technique for fluid-width videos. It is available at *fitvidsjs.com*.

Media Query Magic

SECTION 7

Media Query Magic

- The query itself includes a media type followed by a particular feature and a value for which to test. The criteria are followed by a set of curly brackets that contain styles to apply if the test is passed. The structure of a media query as used within a style sheet goes like this:

```
@media type and (feature:
value) {
  /* styles for browsers that
  meet this criteria */
}
```

```
@media screen and (orientation: landscape){
    body {
        background: skyblue;
    }
}
@media screen and (orientation: portrait){
    body {
        background: coral;
    }
}
```

Media Query Magic

When the viewport is in portrait mode, the background color is “coral.”

When the viewport is in landscape mode, the background color is “skyblue.”

FIGURE 17-5. Changing the background color based on the orientation of the viewport with media queries.

Media Query Magic

Media types

- Media types, as included in the first part of a query, were introduced in CSS2 as a way to target styles to particular media. For example, this **@media** rule delivers a set of styles only when the document is printed (it does not test for any specific features or values):

```
@media print {
   /* print-specific styles here */
}
```

Media types

- The most current defined media types are **all**, **print**, **screen**, and **speech** (see Note). If you are designing for screen, the media type is optional, so you can omit it as shown in the syntax example shown here, but including it doesn't hurt. I'll be including the screen **media** type for the sake of clarity in my examples.

```
@media (feature: value) {
}
```

Media feature queries

- Here is a simple example that displays headline fonts in a fancy cursive font only when the viewport is 40em or wider—that is, when there is enough space for the font to be legible. Viewports that do not match the query (because they are narrower than 40em) use a simple serif face.

```
h1 {
    font-family: Georgia, serif;
}

@media screen and (min-width: 40em) {
h1 {
    font-family: 'Lobster', cursive;
    }
}
```

TABLE 17-1. Media features you can evaluate with media queries

Feature	Description
`width`	The width of the display area (viewport). Also **`min-width`** and **`max-width`**.
`height`	The height of the display area (viewport). Also **`min-height`** and **`max-height`**.
`orientation`	Whether the device is in **`portrait`** or **`landscape`** orientation.
`aspect-ratio`	Ratio of the viewport's width divided by height (width/height). Example: **`aspect-ratio: 16/9`**.
`color`	The bit depth of the display; for example, **`color: 8`** tests for whether the device has at least 8-bit color.
`color-index`	The number of colors in the color lookup table.
`monochrome`	The number of bits per pixel in a monochrome device.
`resolution`	The density of pixels in the device. This is increasingly relevant for detecting high-resolution displays.
`scan`	Whether a **`tv`** media type uses progressive or interlace scanning. (Does not accept **`min-`**/**`max-`** prefixes.)
`grid`	Whether the device uses a grid-based display, such as a terminal window. (Does not accept **`min-`**/**`max-`** prefixes.)

Deprecated features

The following features have been deprecated in Media Queries Level 4 Working Draft and are discouraged from use.

`device-width`	The width of the device's rendering surface (the whole screen). (Deprecated in favor of **`width`**.)
`device-height`	The height of the device's rendering surface (the whole screen). (Deprecated in favor of **`height`**.)
`device-aspect-ratio`	Ratio of the whole screen's (rendering surface) width to height. (Deprecated in favor of **`aspect-ratio`**.)

New in Media Queries Level 4

These features have been added in the Working Draft of MQ4. Some may gain browser support, and some may be dropped from future drafts. I include them here to show you where the W3C sees media queries going. For details, see *drafts.csswg.org/mediaqueries-4*.

`update-frequency`	How quickly (if at all) the output device modifies the appearance of the content.
`overflow-block`	How the device handles content that overflows the viewport along the block axis.
`overflow-inline`	Whether the content that overflows the viewport along the inline axis can be scrolled.
`color-gamut`	The approximate range of colors that are supported by the user agent and output device.
`pointer`	Whether the primary input mechanism is a pointing device and how accurate it is.
`hover`	Whether the input mechanism allows the user to hover over elements.
`any-pointer`	Whether any available input mechanism is a pointing device, and how accurate it is.
`any-hover`	Whether any available input mechanism allows hovering.

How to use media queries

Within a style sheet

- The most common way to utilize media queries is to use an **@media** (“atmedia”) rule right in the style sheet. The examples in this chapter so far are all **@media** rules.
- When you use media queries within a style sheet, the order of rules is very important. Because rules later in the style sheet override the rules that come before them, your media query needs to come *after* any rules with the same declaration.

How to use media queries

Within a style sheet

- The strategy is to specify the baseline styles that serve as a default, and then override specific rules as needed to optimize for alternate viewing environments.
- In RWD, the best practice is to set up styles for small screens and browsers that don’t support media queries, and then introduce styles for increasingly larger screens later in the style sheet.
- That’s exactly what I did in the headline font-switching example earlier. The h1 sets a baseline experience with a local serif font, and then gets enhanced for larger screens with a media query.

How to use media queries

With external style sheets

For large or complicated sites, developers may choose to put styles for different devices into separate style sheets and call in the whole *.css* file when certain conditions are met. One method is to use the **media** attribute in the **link** element to conditionally load separate *.css* files. In this example, the basic styles for a site are requested first, followed by a style sheet that will be used only if the device is more than 1,024 pixels wide (and if the browser supports media queries):

```
<head>
    <link rel="stylesheet"
    href="styles.css">
    <link rel="stylesheet" href="2column-
    styles.css" media="screen and
    (min-width:1024px)">
</head>
```

How to use media queries

With external style sheets

Some developers find this method helpful for managing modular style sheets, but it comes with the disadvantage of requiring extra HTTP requests for each additional .css file. Be sure to provide only as many links as necessary (perhaps one for each major breakpoint), and rely on @media rules within style sheets to make minor adjustments for sizes in between.

```
<style>
    @import url("/default-styles.css");
    @import url("/wide-styles.css") screen
    and(min-width: 1024px);
    /* other styles */
</style>
```

Choosing Breakpoints

SECTION 7

Choosing Breakpoints

- A breakpoint is the point at which we use a media query to introduce a style change. When you specify **min-width: 800px** in a media query, you are saying that 800 pixels is the “breakpoint” at which those particular styles should be used. FIGURE 17-6 shows some of the breakpoints at which Etsy.com makes both major layout changes and subtle design tweaks on its home page.
- Choosing breakpoints can be challenging, but there are a few best practices to keep in mind.

Choosing Breakpoints

- A **breakpoint** is a width at which you introduce a style change.
- When RWD was first introduced, there were only a handful of devices to worry about, so we tended to base our breakpoints on the common device sizes (320 pixels for smartphones, 768 pixels for iPads, and so on), and we created a separate design for each breakpoint.
- It didn’t take long until we had to deal with device widths at nearly every point from 240 to 3,000+ pixels. That device-based approach definitely didn’t scale.

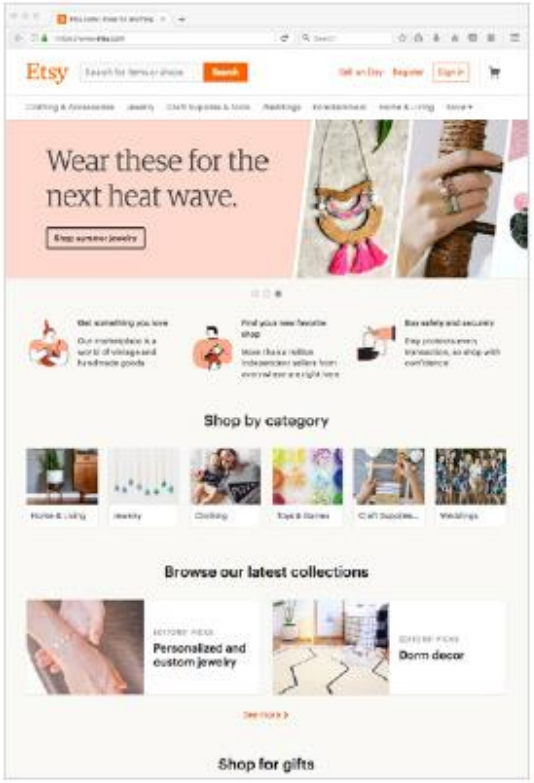
Choosing Breakpoints

At the **480-pixel breakpoint**, the category navigation changes from a list to photos. “Register” is added to the top navigation bar.

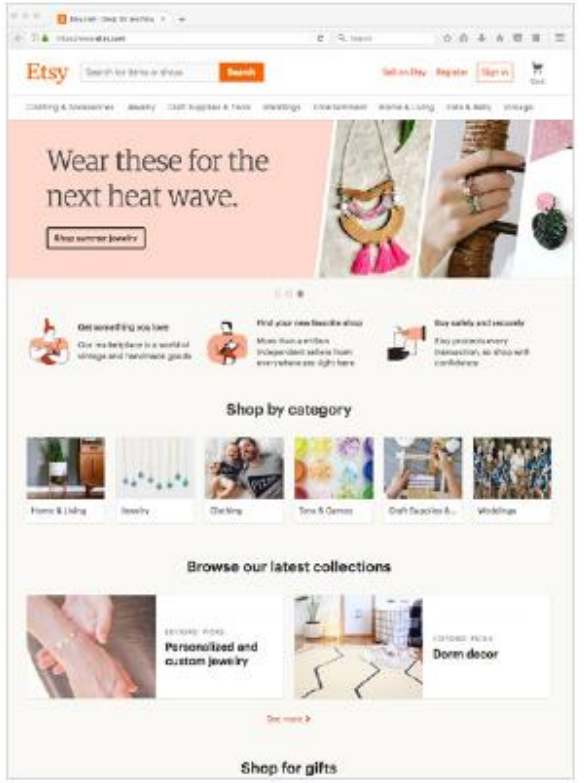
At **501 pixels**, “Sell” becomes “Sell on Etsy” (a very subtle adjustment). You can also see more links in the navigation bar under the search field.

At **640 pixels**, the “How Etsy Works” images and messages move above the categories. In smaller views, they were accessible via the “Learn how Etsy works” link in a yellow bar.

Choosing Breakpoints



At **901 pixels**, the search input form moves into the top header.



At **981 pixels**, the word "Cart" appears under the shopping cart icon. We now see the full list of navigation options in the header (no "More" link). At this point, the layout expands to fill larger windows until it reaches its maximum width of 1400 pixels. Then margins add space equally on the left and right to keep the layout centered.

FIGURE 17-6. A series of breakpoints used by Etsy's responsive site (2017).

Module-Based Breakpoints

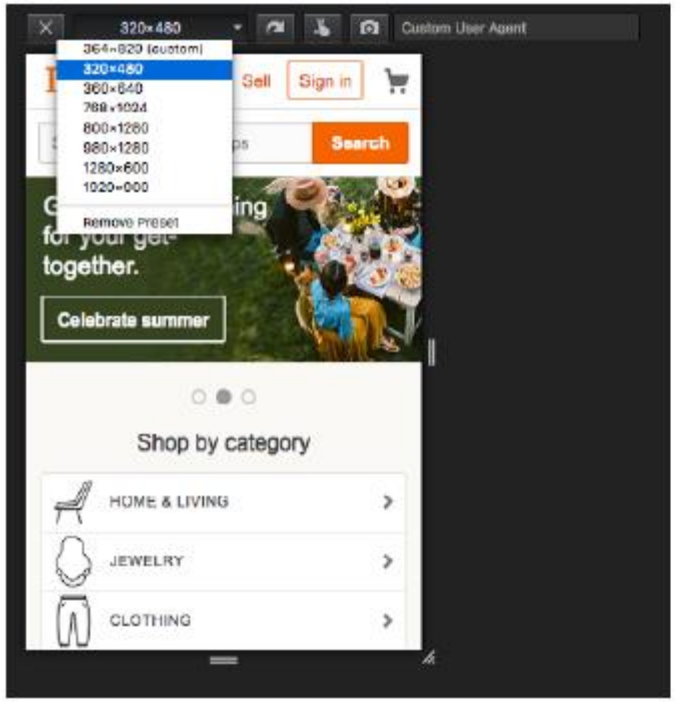
- A better approach is to create breakpoints for the individual parts of a page rather than switching out the entire page at once (although for some pages that may work just fine). A common practice is to create the design for narrow screens first, and then resize the browser wider and pay attention to the point at which each part of the page starts to become unacceptable.
- The navigation might become too awkward and need a breakpoint at 400 pixels wide, but the one-column layout might be OK until it reaches 800 pixels, at which point a two-column design could be introduced.

Module-Based Breakpoints

- If you find that you have a lot of breakpoints within a few pixels or ems of one another, grouping them together may streamline your style sheet and process. And it doesn't hurt to keep the screen sizes of the most popular devices in mind in case nudging your breakpoint down a little helps improve the experience for a whole class of users.
- The site Screen Sizes (*screensiz.es*) lists the dimensions of a wide range of popular devices. A web search will turn up similar resources.

Em-Based Breakpoints

- **em**-based media queries keep the layout proportional to the font size.
- **pixel**-based media queries don’t adapt if the user changes their font size settings, which people do in order to be able to read the page more easily. But **em**-based media queries respond to the size of the text, keeping the layout of the page in proportion.
- Using **em**-based queries, if the query targets browsers wider than 50em, when the base font size is 16 pixels, the switch happens at 800 pixels (as designed).



Responsive Design Mode in Firefox shows you the exact pixel dimensions of its viewport. It has shortcuts to resize it to common device dimensions. Chrome and Safari have similar responsive views.

MQTest.io is a web page that reports on how your browser responds to media queries, including width.

FIGURE 17-7. Checking viewport size in Firefox's Responsive Design Mode and MQTest.io.

How Wide Is the Viewport?

Design Responsively

SECTION 7

Design Responsively

- In the broadest of strokes, the tricky bits to keep optimized over a wide range of viewport sizes include the following:
  - Content hierarchy
  - Layout
  - Typography
  - Navigation
  - Images
  - Special content such as tables, forms, and interactive features

Content Hierarchy

- Content is king on the web, so it is critical that content is carefully considered and organized before any code gets written. These are tasks for **Information Architects** and **Content Strategists** who address the challenges of organizing, labeling, planning, and managing web content.
- Organization and hierarchy across various views of the site are a primary concern, with a particular focus on the small-screen experience. It is best to start with an inventory of potential content and pare it down to what is most useful and important for all browsing experiences. Once you know what the content modules are, you can begin deciding in what order they appear on various screen sizes.

Layout

- Rearranging content into different layouts may be the first thing you think of when you picture responsive design, and with good reason. The layout helps form our first impression of a site's content and usability.
- As mentioned earlier, responsive design is based on fluid layouts that expand and contract to fill the available space in the viewport. One fluid layout is usually not enough, however, to serve all screen sizes. More often, two or three layouts are produced to meet requirements across devices, with small adjustments between layout shifts.

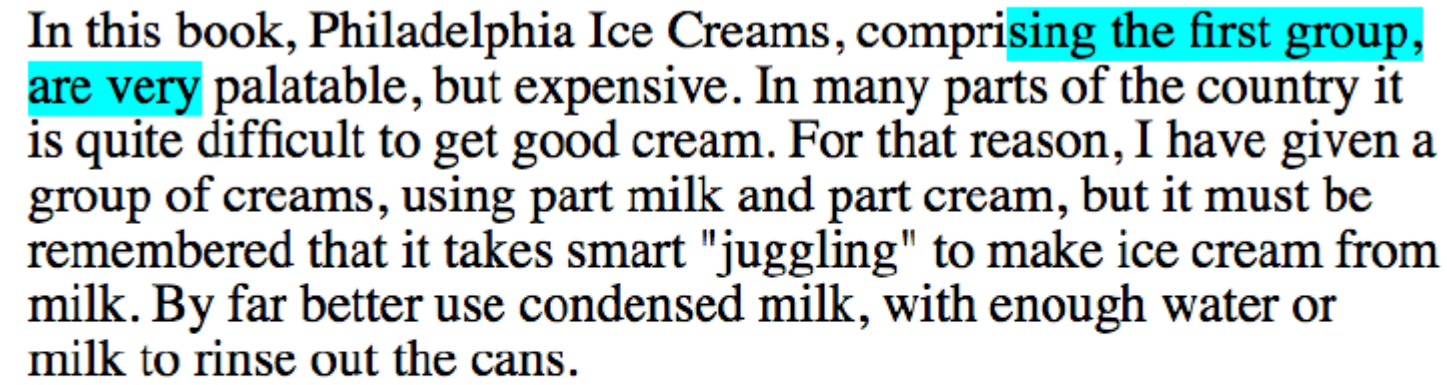


FIGURE 17-8. Highlight the 45th to 75th characters to test for optimal line lengths at a glance.

Layout and line length

Layout and line length

Mostly fluid

This pattern uses a single-column layout for small screens, and another fluid layout that covers medium and large screens, with a maximum width set to prevent it from becoming too wide. It generally requires less work than other solutions.

Column drop

This solution shifts between one-, two-, and three-column layouts based on available space. When there isn't room for extra columns, the sidebar columns drop below the other columns until everything is stacked vertically in the one-column view.

Layout shifter

If you want to get really fancy, you can completely reinvent the layout for a variety of screen sizes. Although expressive and potentially cool, it is not necessary. In general, you can solve the problem of fitting your content to multiple environments without going overboard.

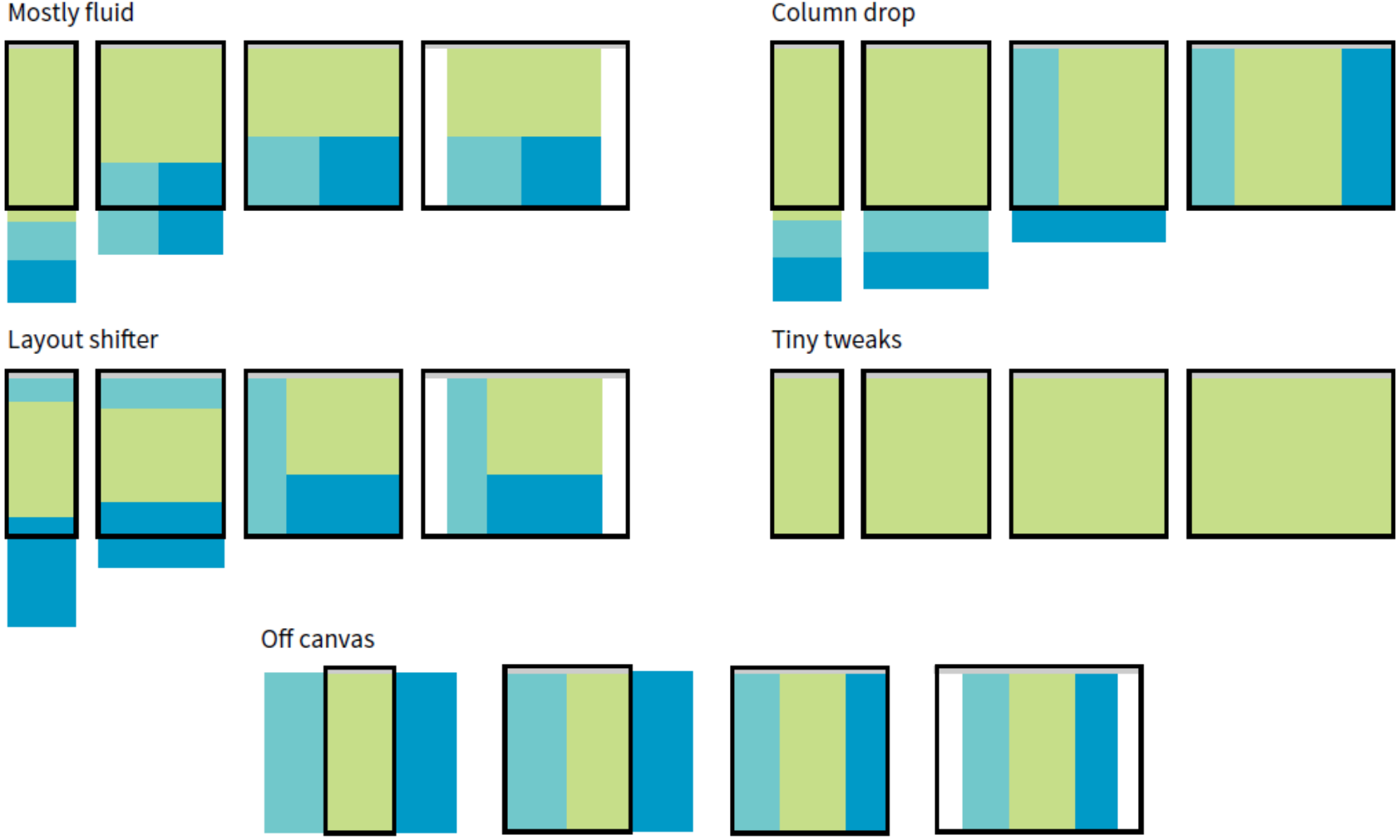
Layout and line length

Tiny tweaks

Some sites use a single-column layout and make tweaks to type, spacing, and images to make it work across a range of device sizes.

Off canvas

As an alternative to stacking content vertically on small screens, you may choose to use an “off-canvas” solution. In this pattern, a page component is located just out of sight on the left or right of the screen and flies into view when requested. A bit of the main content screen remains visible on the edge to orient users as to the relationship of moving parts. This was made popular by Facebook, wherein Favorites and Settings were placed on a panel that slid in from the left when users clicked a menu icon.



Responsively

FIGURE 17-9. Examples of the responsive layout patterns identified by Luke Wroblewski.

Typography

Font face

Be careful about using fancy fonts on small screens and be sure to test for legibility. At small sizes, some fonts become difficult to read because line strokes become too light or extra flourishes become little blobs. Consider also that small screens may be connecting over cellular, so taking advantage of locally available fonts may be better for performance than requiring a web font to download. If a strict brand identity requires font consistency on all devices, be sure to choose a font face that works well at all sizes. If that is not a concern, consider using a web font only on larger screens. We strive to serve the same design to all devices, but as with everything else in web design, flexibility is important.

Font size

Varying viewport widths can wreak havoc on line lengths. You may find that you need to increase the font size of text elements for wider viewports to maintain a line length of between 45 and 75 characters. It also makes it easier to read from the distance users typically sit from their large screens. Conversely, you could use em-based media queries so that the layout stays proportional to the font size. With em-based queries, line lengths stay consistent.

Typography

Line height

Line height is another measurement that you may want to tweak as screens get larger. On average, line height should be about 1.5 (using a number value for the **line-height** property); however, slightly tighter line spacing (1.2 to 1.5) is easier to read with the shorter line lengths on small screens. Large screens, where the type is also likely to be larger, can handle more open line heights (1.4 to 1.6).

Margins

On small screens, make the most of the available space by keeping left and right margins on the main column to a minimum (2–4%). As screens get larger, you will likely need to increase side margins to keep the line lengths under control and just to add some welcome whitespace to the layout. Remember to specify margins above and below text elements inem units so they stay proportional to the type.

Navigation

- Navigation feels a little like the Holy Grail of Responsive Web Design. It is critical to get it right. Because navigation at desktop widths has pretty much been conquered, the real challenges come in re-creating our navigation options on small screens. A number of successful patterns have emerged for small screens, which I will briefly summarize here (FIGURE 17-11):

Navigation

Top navigation

If your site has just a few navigation links, they may fit just fine in one or two rows at the top of the screen.

Priority +

In this pattern, the most important navigation links appear in a line across the top of the screen alongside a More link that exposes additional options. The pros are that the primary links are in plain view, and the number of links shown can increase as the device width increases. The cons include the difficulty of determining which links are worthy of the prime small-screen real estate.

Navigation

Select menu

For a medium list of links, some sites use a **select** input form element. Tapping the menu opens the list of options using the select menu UI of the operating system, such as a scrolling list of links at the bottom of the screen or on an overlay. The advantage is that it is compact, but on the downside, forms aren't typically used for navigation, and the menu may be overlooked.

Link to footer menu

One straightforward approach places a Menu link at the top of the page that links to the full navigation located at the bottom of the page. The risk with this pattern is that it may be disorienting to users who suddenly find themselves at the bottom of the scroll.

Navigation

Accordion sub-navigation

When there are a lot of navigation choices with sub-navigation menus, the small-screen solution becomes more challenging, particularly when you can’t hover to get more options as you can with a mouse. Accordions that expand when you tap a small arrow icon are commonly used to reveal and hide sub-navigation. They may even be nested several levels deep. To avoid nesting navigation in accordion submenus, some sites simply link to separate landing pages that contain a list of the sub-navigation for that section.

Navigation

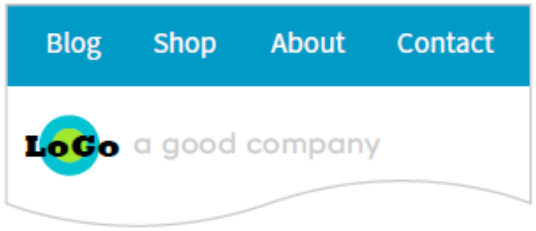
Push and overlay toggles

In toggle navigation, the navigation is hidden but expands downward when the menu link is tapped. It may push the main content down below it (push toggle) or slide down in front of the content (overlay toggle).

Off-canvas/fly-in

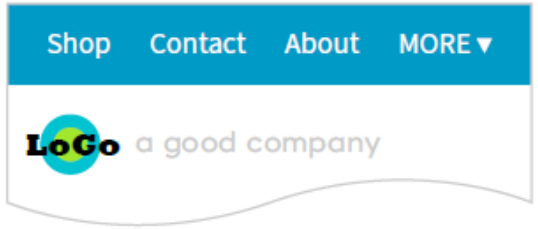
This popular pattern puts the navigation in an off-screen panel to the left or right of the main content that slides into view when you tap the menu icon.

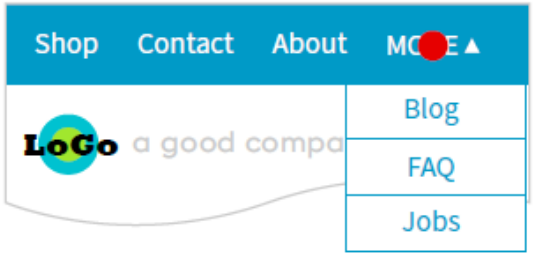
Top navigation



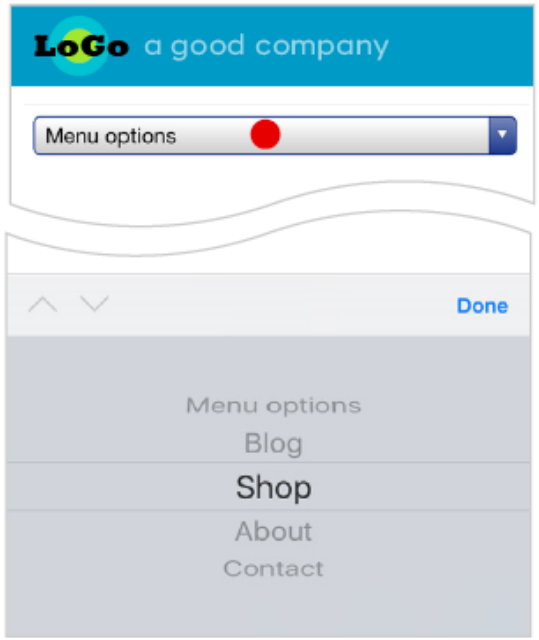
KEY

Priority +

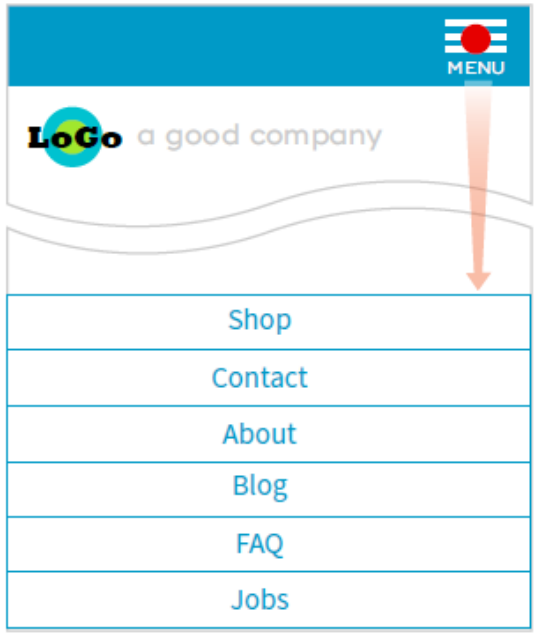




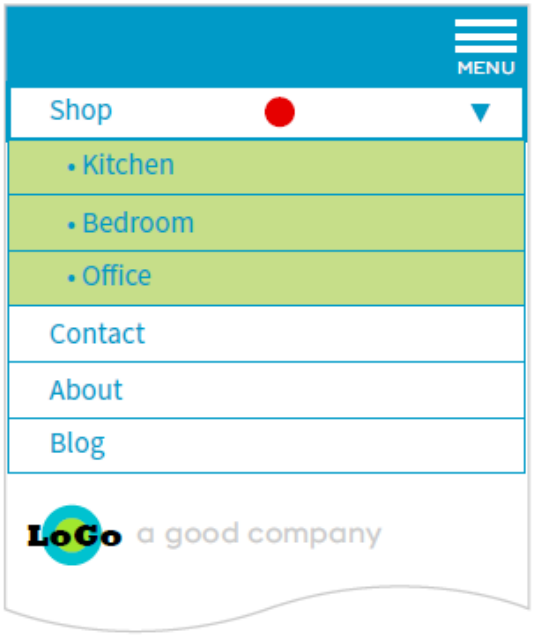
Select menu



Link to footer menu



Accordion sub-navigation



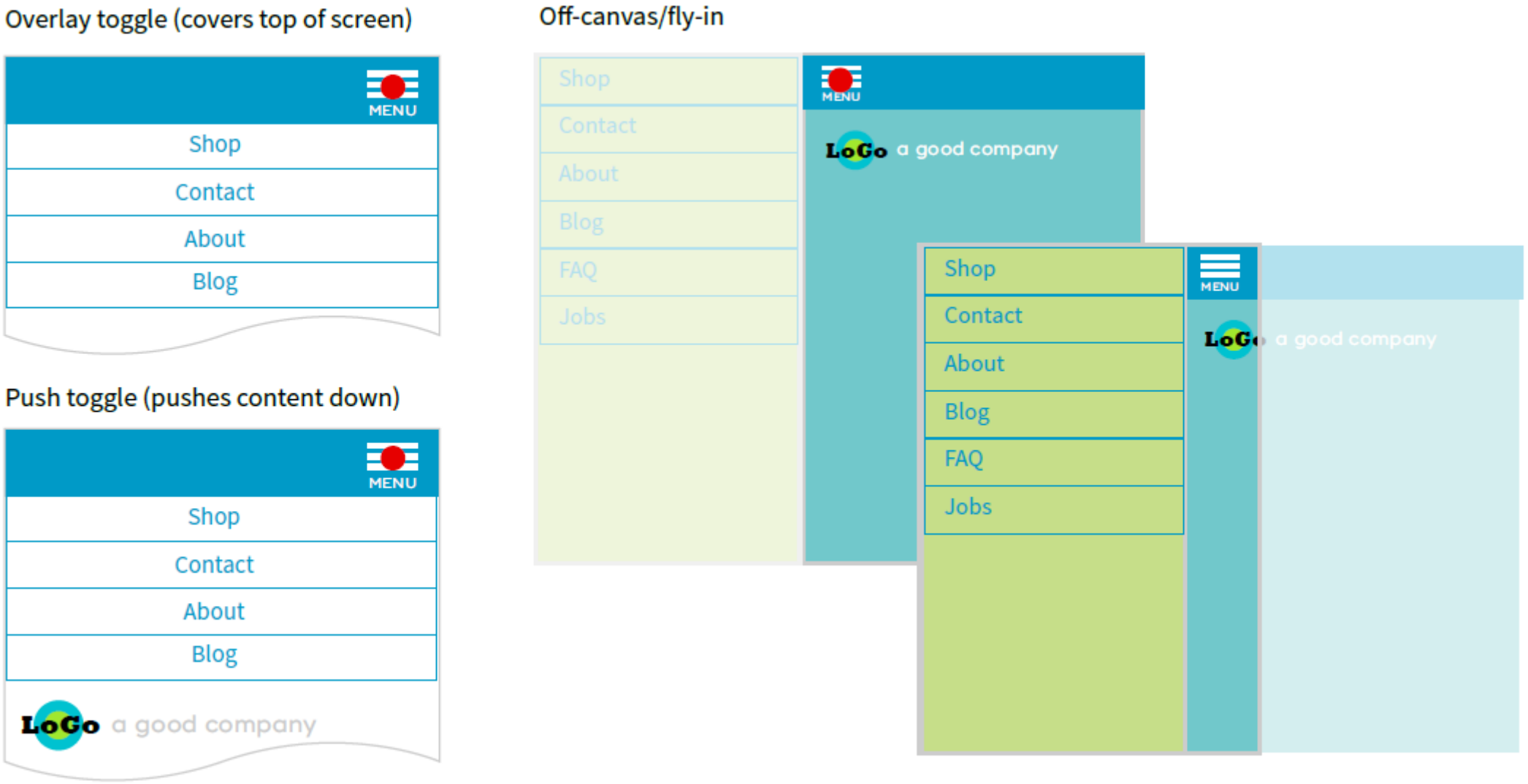


FIGURE 17-11. Responsive navigation patterns.

Images

Images require special attention in responsive designs. Here is a quick rundown of some of the key issues, most of which should sound familiar:

- Use responsive image markup techniques (covered in Chapter 7) to provide multiple versions of key images for various sizes and resolutions.
- Serve the smallest version as the default to prevent unnecessary data downloads.
- Be sure that important image detail is not lost at smaller sizes. Consider substituting a cropped version of the image for small screens.
- Avoid putting text in graphics, but if it is necessary, provide alternate versions with larger text for small screens.

Images

Special Content

Without the luxury of wide-open, desktop viewports, some of our common page elements pose challenges when it comes to fitting on smaller screens:

Forms

Forms often take a little finagling to fit the available space appropriately. Flexbox is a great tool for adding flexibility and conditional wrapping to form fields and their labels. A web search will turn up some fine tutorials.
Also make sure that your form is as efficient as possible, with no unnecessary fields, which is good advice for any screen size. Finally, consider that form inputs will be used with fingertips, not mouse pointers, so increase the target size by adding ample padding or margins and by making labels tappable to select an input.

Images

Special Content

Tables

One of the greatest challenges in small-screen design is how to deal with large data tables. Not surprisingly, because there are many types of tables, there are also many solutions. See the “The Trouble with Tables” sidebar for more information and resources.

Interactive elements

A big embedded map may be great on a desktop view of a site, but it is less useful when it is the size of a postage stamp. Consider whether some interactive features should be substituted for other methods for performing the same task. In the case of the map, adding a link to a map can trigger
the device’s native mapping app to open, which is designed to provide a better small-screen experience. Other interactive components, such as carousels, can be adapted for smaller viewports.

A Few Words about Testing

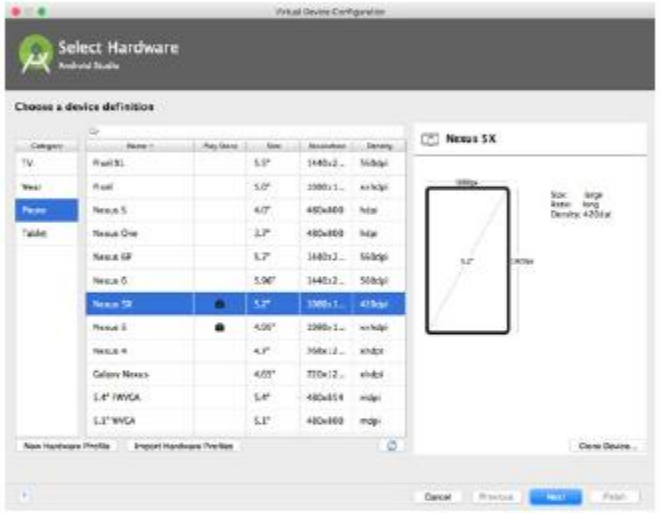
SECTION 7

Real Devices

- Web development companies may have a device lab comprising iPhones and iPads of various sizes, Android smartphones and tablets of various sizes, and Macs and PCs with recent operating systems (Windows and Linux) that can be used by designers and developers for testing sites (FIGURE 17-17). The size of the device lab depends on the size of the budget, of course (electronic devices aren't cheap!).

FIGURE 17-17. The device lab at Filament Group in Boston, Massachusetts.

Emulators



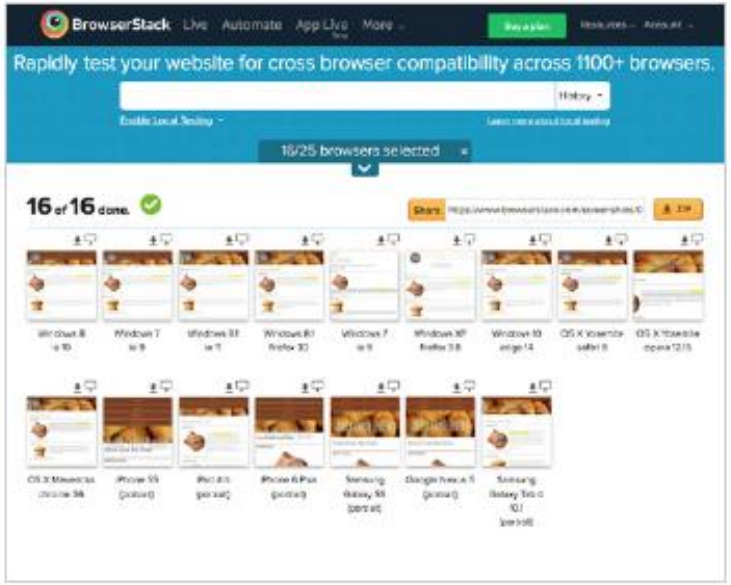
The Android Emulator lets you set up a wide variety of phones, televisions, wearables, and tablets for testing. I chose a Nexus 5X.

The Nexus 5X emulator displays an image of the device at actual size. All of the buttons work as they would on the phone.

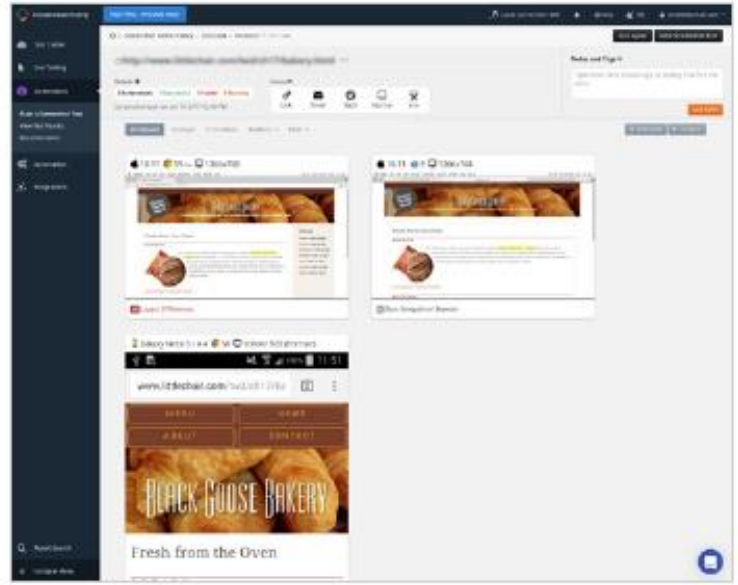
The Bakery page viewed on the Nexus 5X emulator.

FIGURE 17-18. Examples of the Android Emulator (download at *developer.android.com/studio/index.html*).

Third-Party Services



BrowserStack.com



CrossBrowserTesting.com

FIGURE 17-19. Screenshots generated by BrowserStack and CrossBrowserTesting (using free trial tools). Notice the variation in how the bakery page displays. This is why we test!

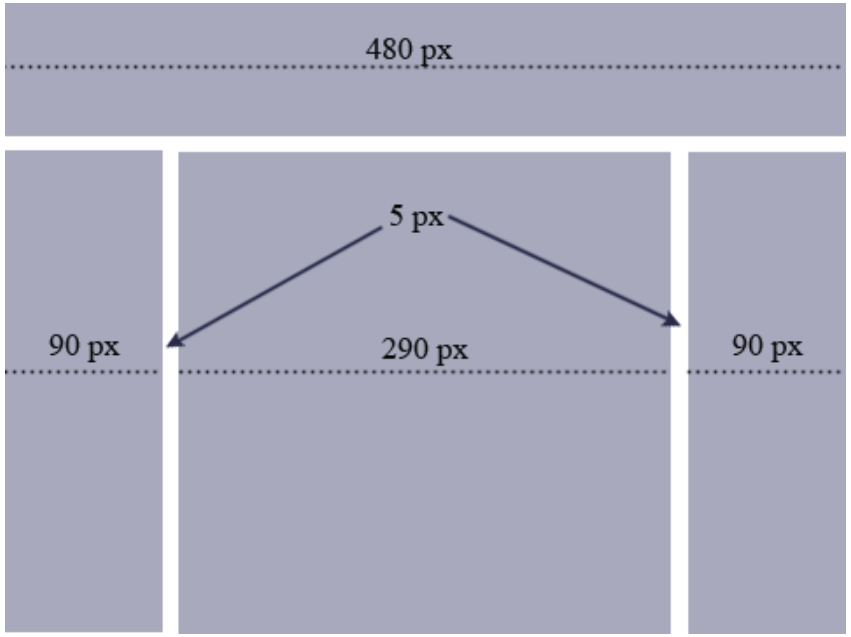
Design Styles

[Flow Layout Design]

SECTION 7

Page Layout Strategies

- **Fixed Layouts:** stay put at a specific pixel width regardless of the size of the browser window or text size.
- **Fluid (or liquid) Layouts:** resize proportionally when the browser window resizes.
- **Elastic Layouts:** resize proportionally based on the size of the text.
- **Hybrid Layouts:** combine fixed and scalable areas.



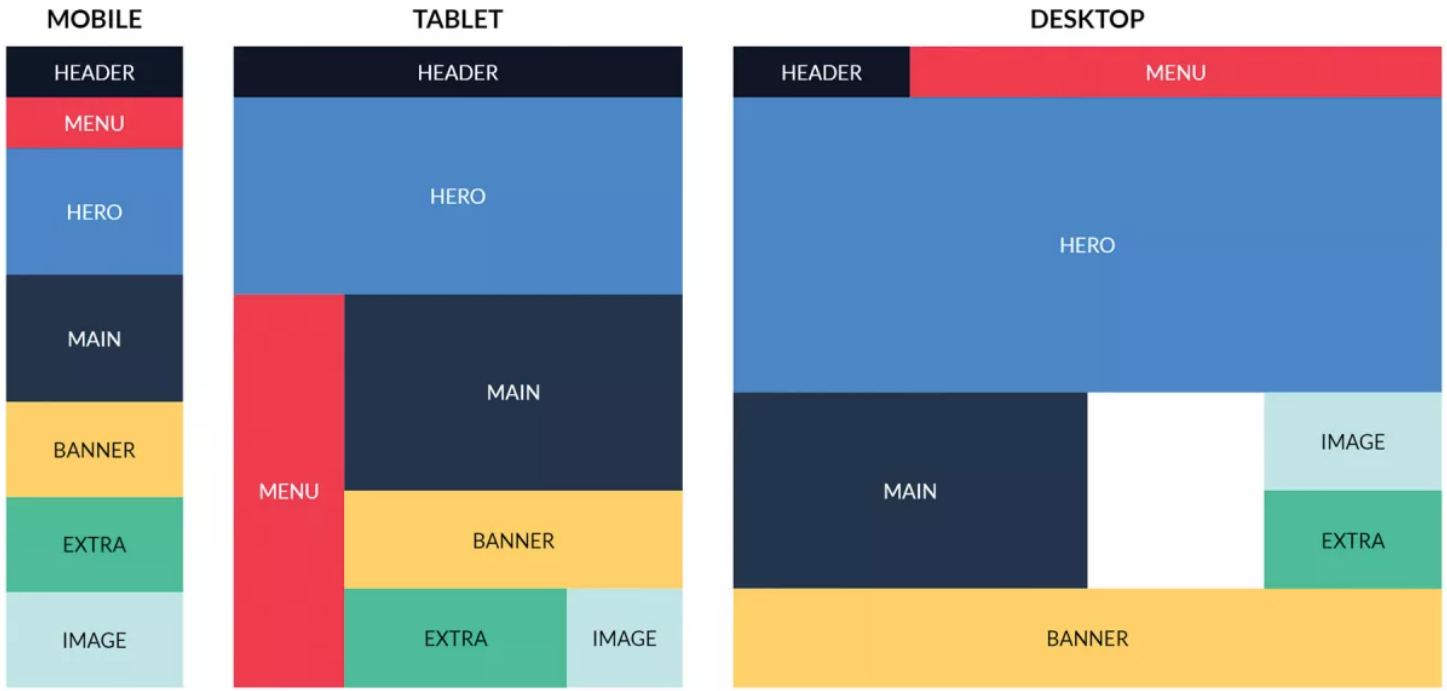
Advantages and Disadvantages of Fixed

Advantages:

- The layout is predictable and offers better control over line length.
- It is easier to design and produce.
- It behaves the way the majority of web pages behave as of this writing, but that may change as users visit the web primarily on the devices other than the desktop.

Disadvantages:

- Content on the right edge will be hidden if the browser window is smaller than the page .
- There may be an awkward amount of leftover space on large screens.
- Line lengths may grow awkwardly short at very large text sizes.
- Takes control away from the user.



An easier way to do fixed layout is to use CSS Grid Framework

Grid framework is to design web pages on grid (larger granularity).

A grid is an invisible foundation that provides the page into equal units that can be used to determine where columns, headlines, images, and so on should fall .

www.960.gs

www.blueprintcss.org

http://www.bluetrip.org

http://developer.yahoo.com/yui/grids/

http://www.cssgrid.net

http://www.getskeleton.com

http://twitter.github.com/bootstrap

http://grids.subtraction.com

Fluid Design

[Flow Layout Design]

SECTION 8

Fluid Page Design (Liquid Design)

- In fluid page layouts (also called liquid layouts), the page area and columns within the page get wider or narrower to fill the available space in the browser window. In other words, they follow the default behavior of the normal flow.
- There is no attempt to control the width of the content or line breaks, the text is permitted to reflow as required and as in natural to the medium.
- Fluid layouts are a cornerstone of the **responsive web design** technique.

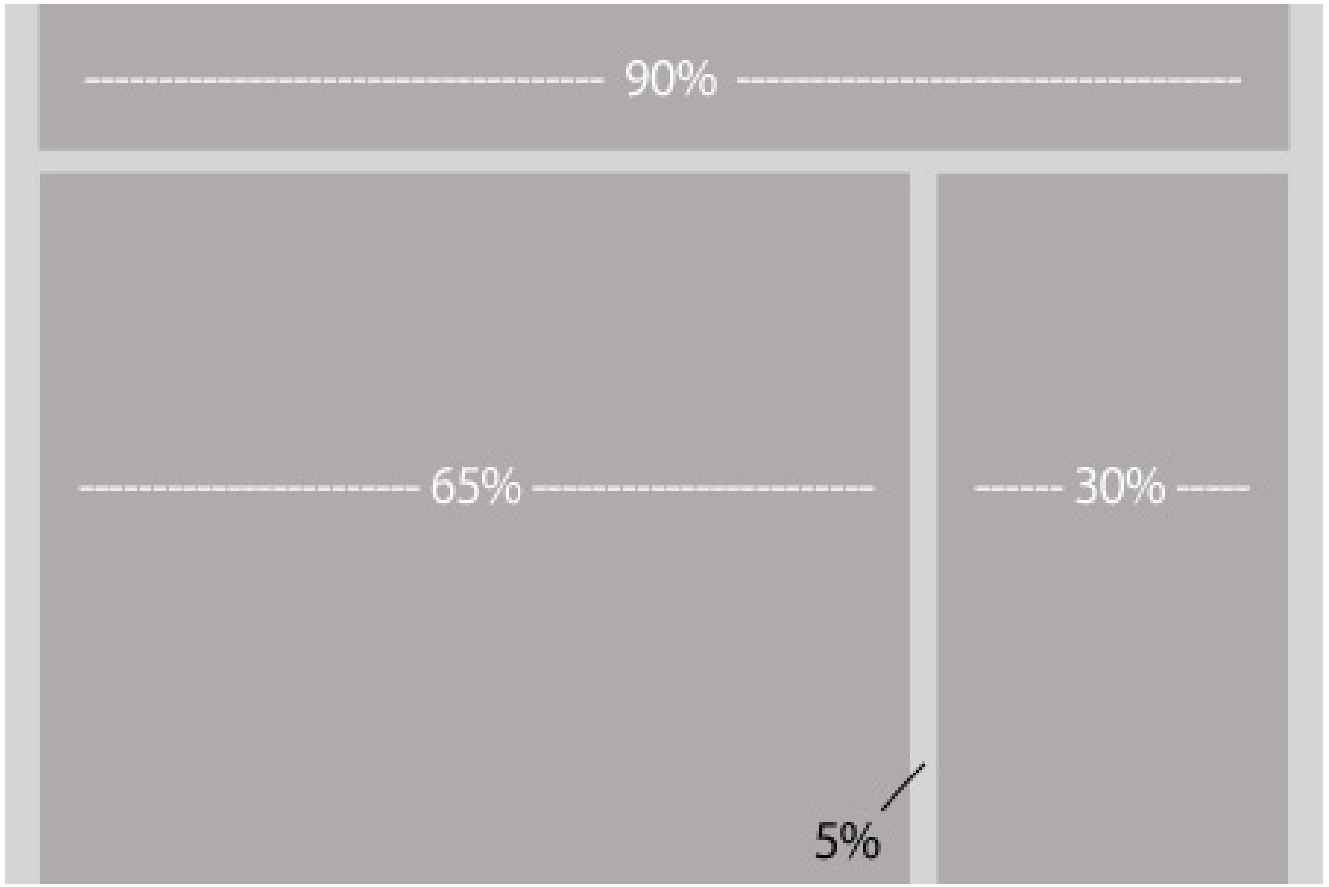
Fluid Page Design (Liquid Design)

Advantages:

- Fluid layouts keep with the spirit and nature of the medium.
- They avoid potentially awkward empty space because the text fills the window.
- On the desktop browsers, users can control the width of the window and content.
- No horizontal scrollbars.

Disadvantages:

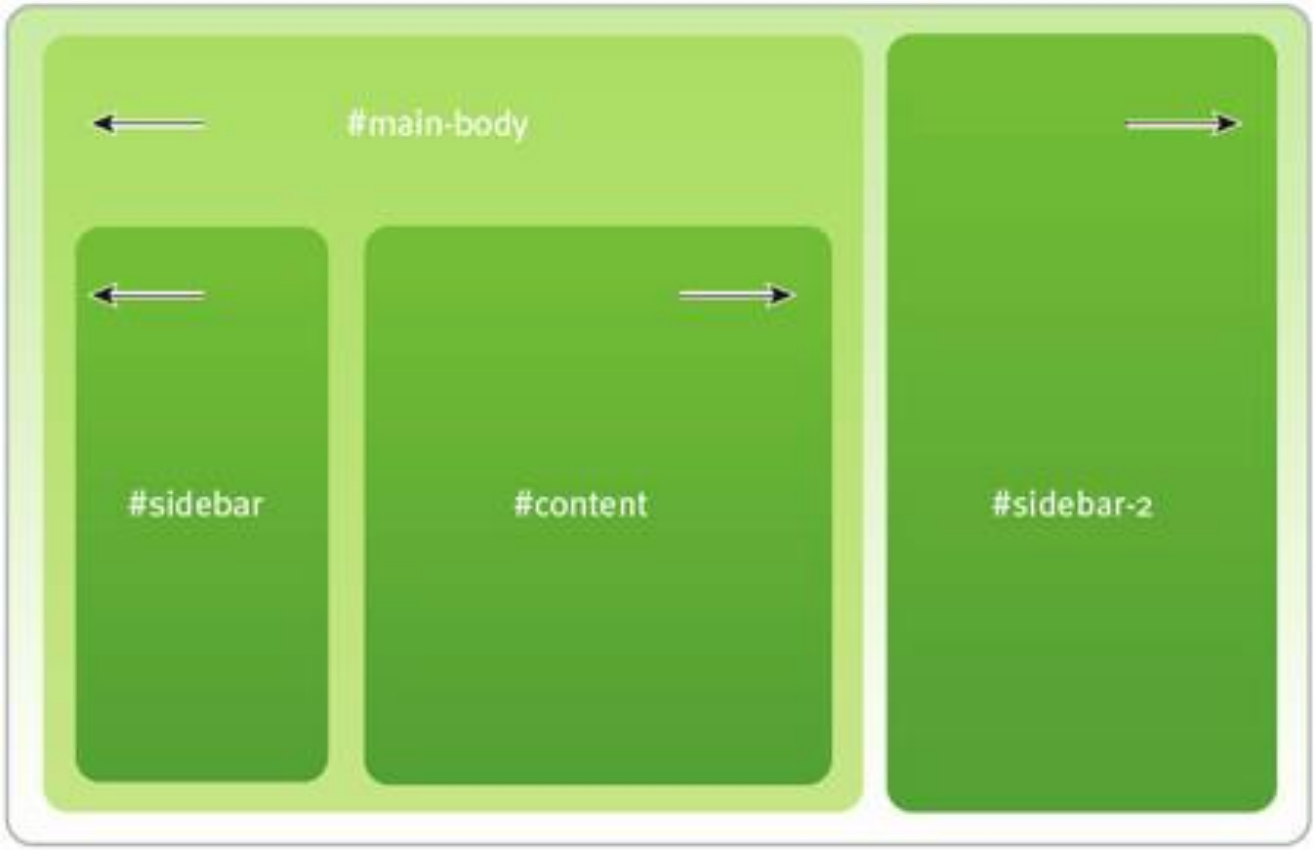
- On large monitors, line lengths can get very long and uncomfortable to read.
- They are less predictable. Elements may be too spread out or too cramped at extreme browser dimensions.
- It may be more difficult to achieve whitespace.
- There is more math involved in calculating measurements



Fluid Page

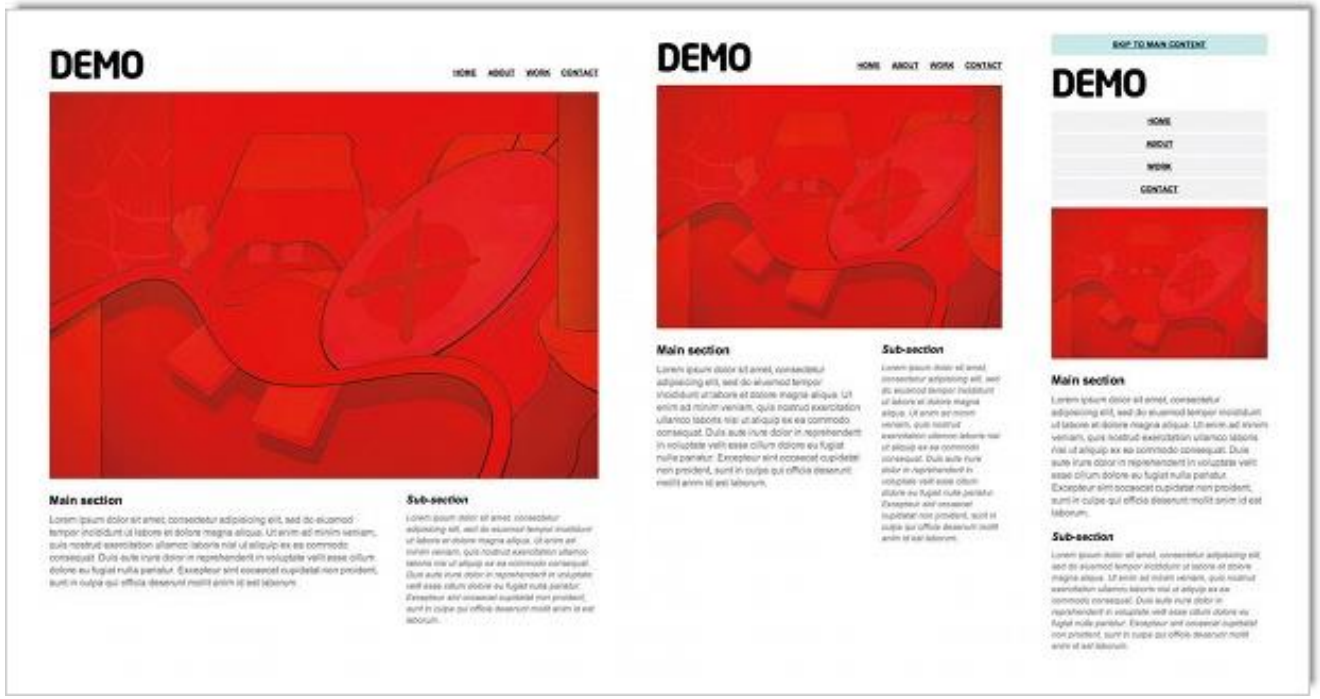
(All in Percentage. Text Font size fixed. Text will flow down.)

MIN-WIDTH AND **MAX-WIDTH** KEEP THE FLUID PAGE FROM BECOMING UNREASONABLY LARGE OR SMALL.



Fluid Page

(All in Percentage. Text Font size fixed. Text will flow down.)



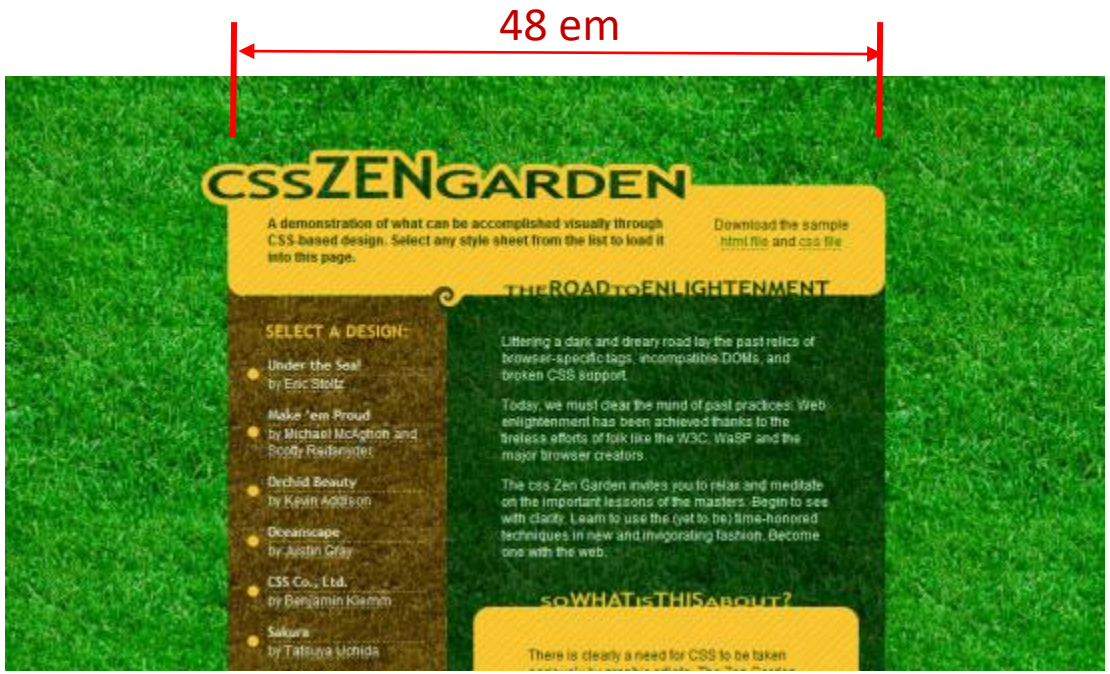
Fluid Page example

Elastic Layout

[Flow Layout Design]

SECTION 9

Elastic Layout (width scaled by em as unit)



Elastic Page

(Width scaled elastically.)

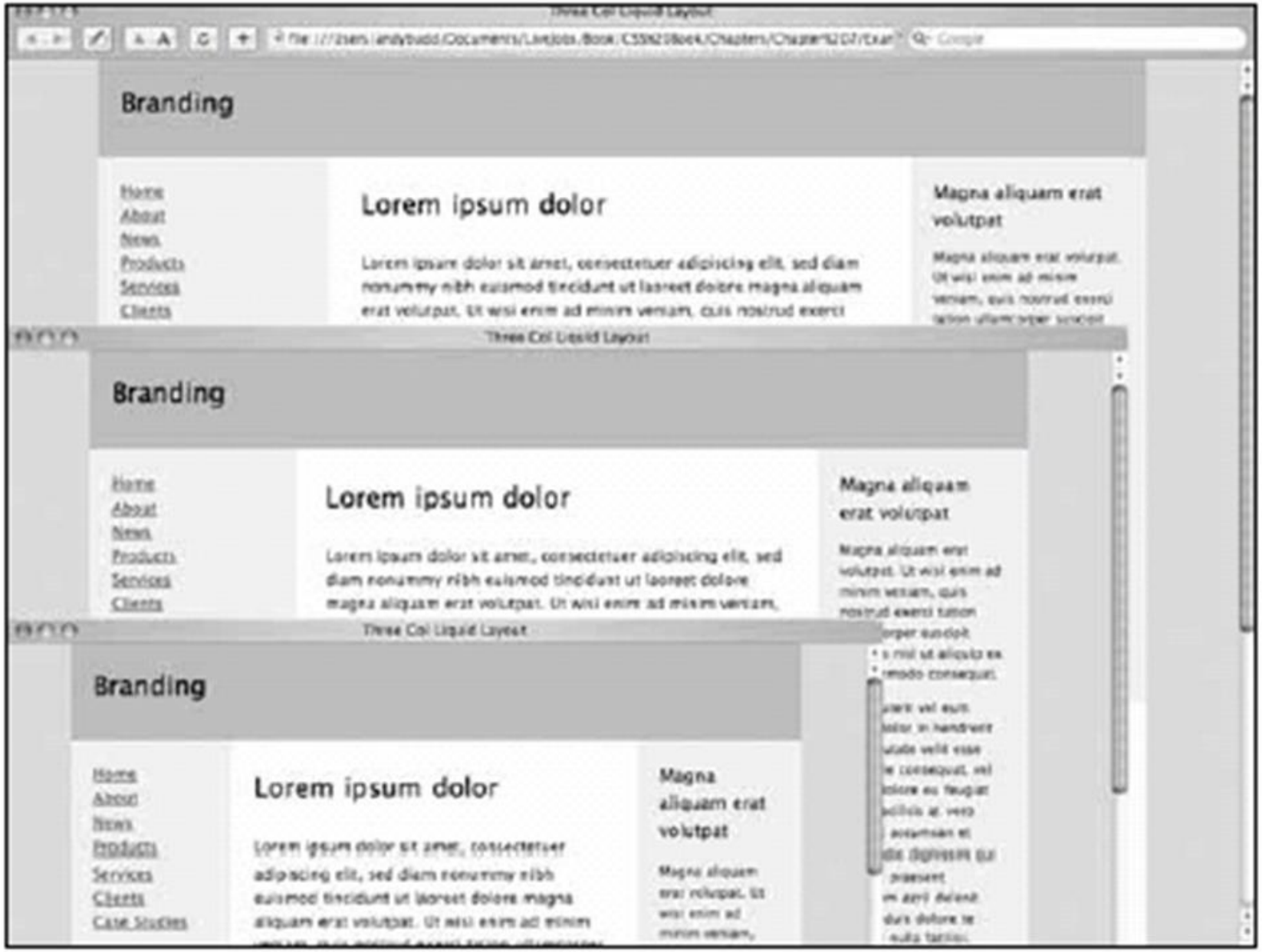
Most browser support **FULL-PAGE ZOOM**.
min-width and max-width can limit the extreme cases.

Advantage:

- Provide a consistent layout experience while allowing flexibility in text size.
- Tighter control over line lengths than liquid and fixed layouts.

Disadvantage:

- Images and videos don’t lend themselves to automatic rescaling along with the text and the rest of the layout (but there are methods to achieve this.)
- The width of the layout might exceed the width of the browser window at largest text sizes.
- Not as useful for addressing device and browser size variety.
- More complicated to create than fixed width layouts.


Branding
Home
About
News
Products
Services
Clients
Lorem ipsum dolor
Magna aliquam erat volutpat
Three Col Liquid Layout
Branding
Home
About
News
Products
Services
Clients
Lorem ipsum dolor
Magna aliquam erat volutpat
Three Col Liquid Layout
Branding
Home
About
News
Products
Services
Clients
Case Studies
Lorem ipsum dolor
Magna aliquam erat volutpat

Difference Between Fluid Design and Elastic Design

Hello World ! Hello World !
Happy World ! Happy World !
Good Job ! Merry Goo Job !
Wonderful ! Wonderful !

Hello World !
Hello World !
Happy World !
Happy World !
Good Job !
Merry Goo Job
!
Wonderful !
Wonderful !

Fluid (Flow down like liquid)

Difference Between Fluid Design and Elastic Design

Elastic (Flexible like Rubber Band)

How to create Elastic Page

- The key to elastic layouts is the **em**s, the unit of measurement that is based on the size of the text.
- It is common to specify **font-size** in ems.
- In elastic layouts, the dimensions of containing elements are specified in **em**s as well. That is how the widths can respond to the text size.

Hybrid Design

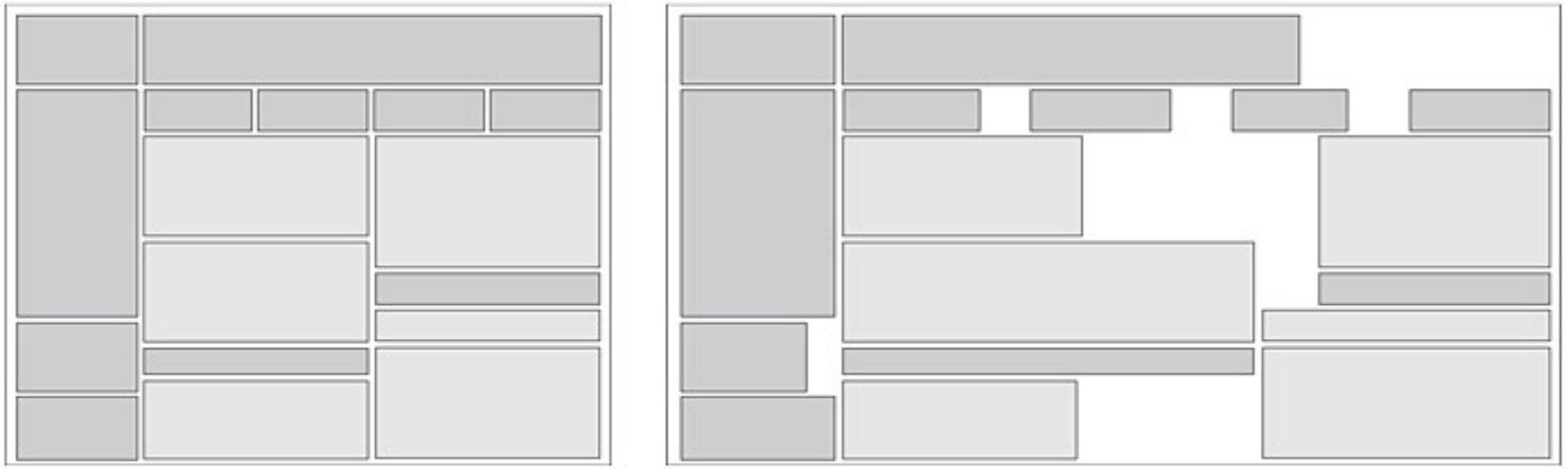
[Flow Layout Design]

SECTION 10

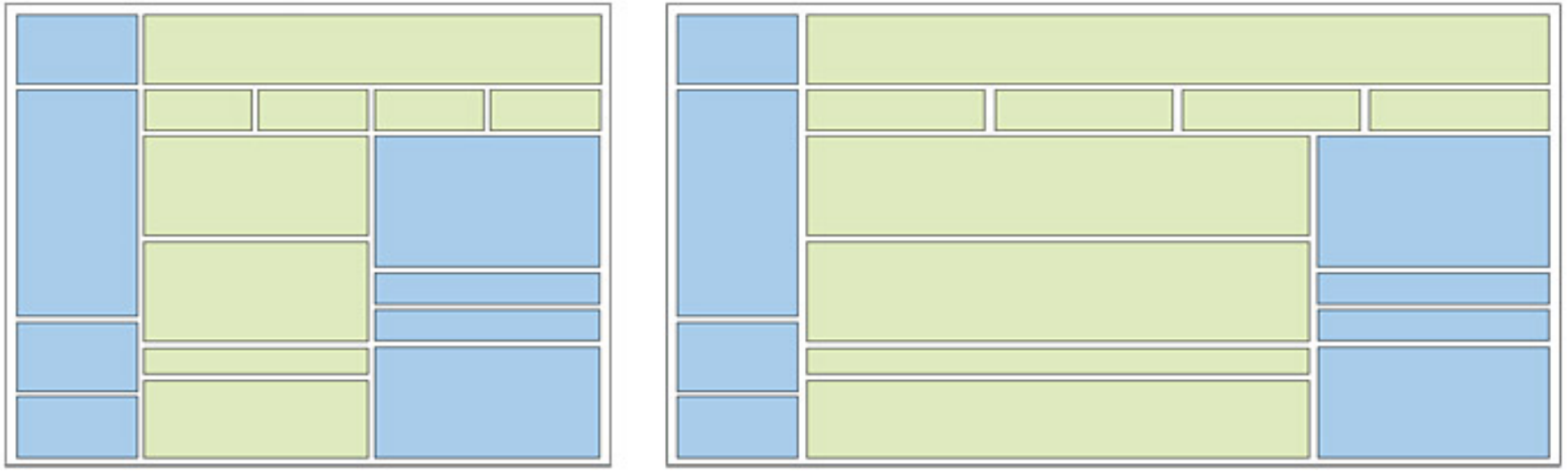
Hybrid Layouts

- Layouts that use a combination of **pixel (fixed), percentage (fluid),** and **em (elastic)** measurements are sometimes called Hybrid Layouts.
- In many scenarios, it makes sense to mix fixed and scalable content areas. For example, you might have a side-bar that contains a stack of ad banners that must stay a particular size. You could specify that sidebar at a particular pixel width and allow the column next to it to resize to fill the remaining space.

Pages that look reasonable in a fixed layout (below, left) can fall apart when converted to a stretchy "liquid" layout (right)

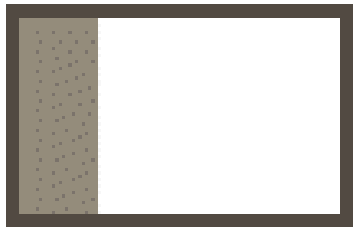


Careful planning and a hybrid approach that mixes liquid (green) and fixed-width layout elements (blue) can give you the best of both worlds: pages that adapt to a wide variety of screen sizes and media, with some areas of predictable widths

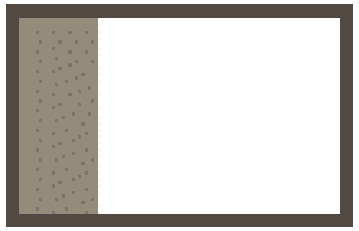


Mixed Layouts

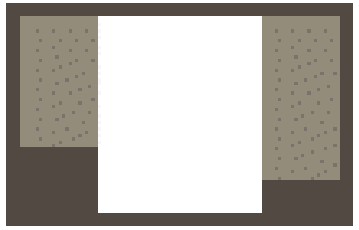
Menu and content dynamic



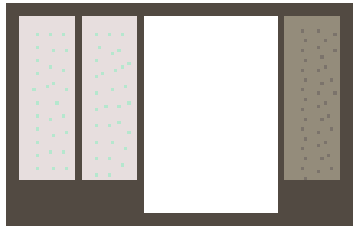
Menu fixed, Content dynamic



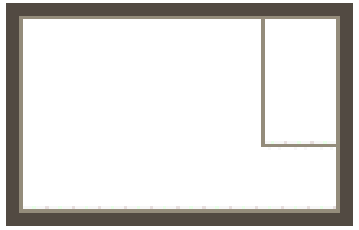
Menu and content dynamic



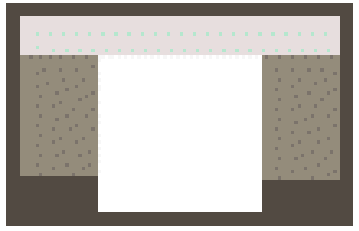
3 columns, all dynamic



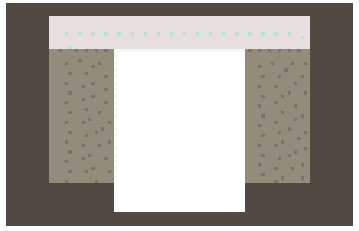
4 columns, all dynamic



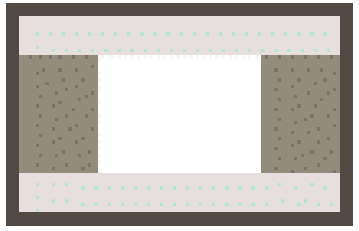
Menu floating



Menu fixed, content & header dynamic



3 columns fixed centered



dynamic with header and footer

Page Layout Techniques

- CSS Templates: two column and three column layout examples using CSS. Templates can provide general purpose layout for many applications.
- The next section provides templates and techniques for the following:
  - Two- and three-column layouts using floats
  - A source-independent layout using floats and negative margins
  - A multicolumn layout using positioning

The future of CSS Layout

- Column (www.w3.org/TR/css3-multicol): use the property **column-count** to specify a number of columns or a specific column width that will repeat until it runs out of room.
- Flexbox (http://www.the-haystack.com/2012/04/learn-you-a-flexbox): The CSS Flexible Box Layout Model provides a much simpler way to arrange element boxes in relation to one another. For example, you can line children elements up within a parent, select where extra space appears, center things horizontally or vertically, and even change the order of appearance – all without resorting to floats and margin offsets and the tricky calculation s that come with them.
- Grid Layout System (http://dev.w3.org/csswg/css3-grid-layout , http://msdn.Microsoft.com/library/ie/hh673536.aspx#-CSSGrid): Initiated by Microsoft
- Regions and Exclusions (http://dev.w3.org/csswg/css3-regions, http://dev.w3.org/csswg/css3-exclusions): Initiated by Adobe. (http://html.adobe.com)

Multiple Column Design

[Flow Layout Design]

SECTION 11

Multi-column Layouts Using Floats

- Floats are the primary tool for creating columns on web pages. As a tool, it is flawed, but it’s the best that we’ve got as of the book.
- The **advantages** that floats have over absolute positioning for layout are that they prevent content from overlapping other content, and they make it easier to keep footer content at the bottom of the page.
- The **drawback** is that they are dependent on the order in which the elements appear in the source, although there is a workaround using negative margins, aw we’ll see later in this section.

Two column, fluid layout

The strategy

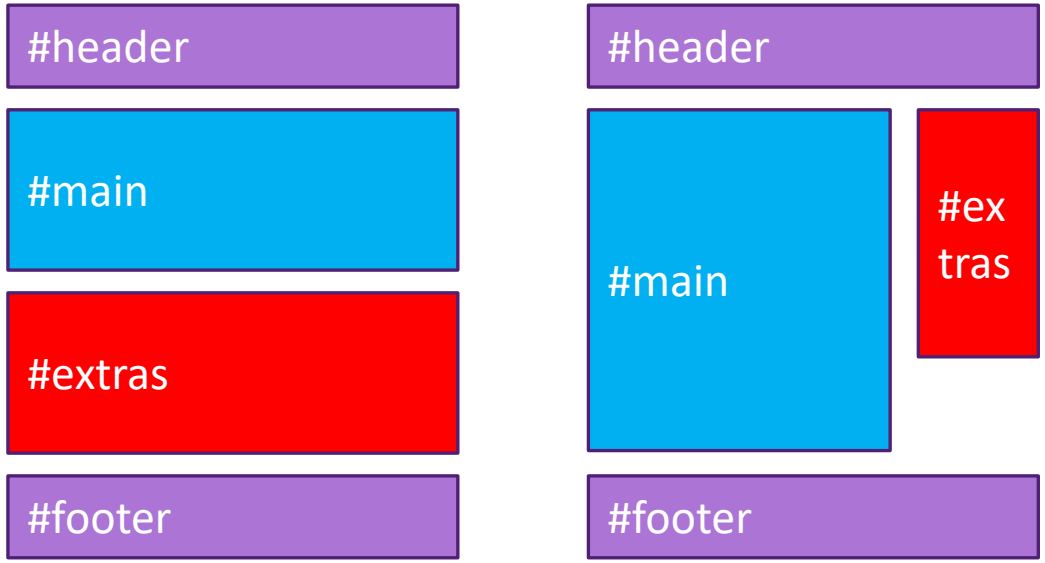
Set widths on both column elements and float them to the left. Clear the footer to keep it at the bottom of the page.

The markup

```
<div id="header">Masthead and headline</div>
<div id="main">Main article</div>
<div id="extras">List of links and news</div>
<div id="footer">Copyright information</div>
```

The styles

```
#main {float: left; width:60% margin 0 5%}
#extras {float: left; width:25%; margin: 0 5% 0 0; }
#footer {clear: left; }
```



Two column, fixed width layout

The strategy

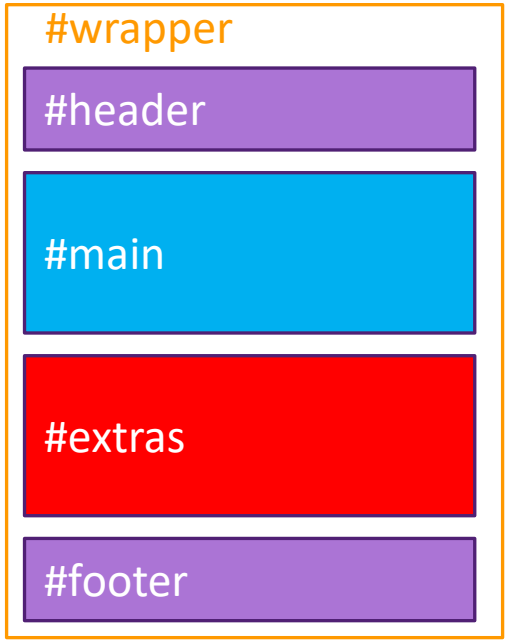
Set widths on both column elements and float them to the left. Clear the footer to keep it at the bottom of the page.

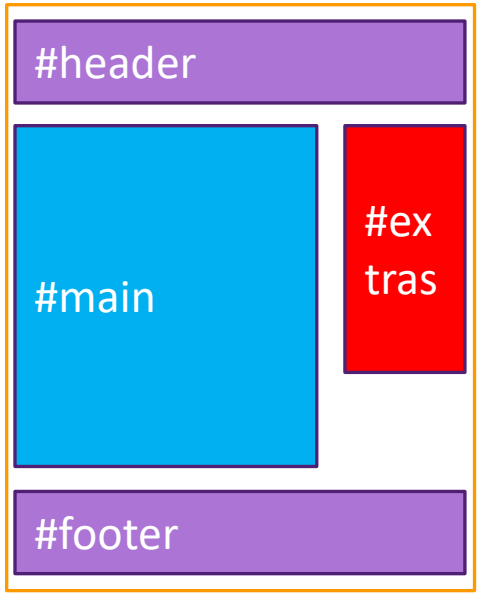
The markup

```
<div id="wrapper">
   <div id="header">Masthead and headline</div>
   <div id="main">Main article</div>
   <div id="extras">List of links and news</div>
   <div id="footer">Copyright information</div>
</div>
```

The styles

```
#wrapper {width:960px}
#main {float: left; width:60%; margin 0 5%}
#extras {float: left; width:25%; margin: 0 5% 0 0; }
#footer {clear: left; }
```





Three columns, positioned, fluid layout

The strategy

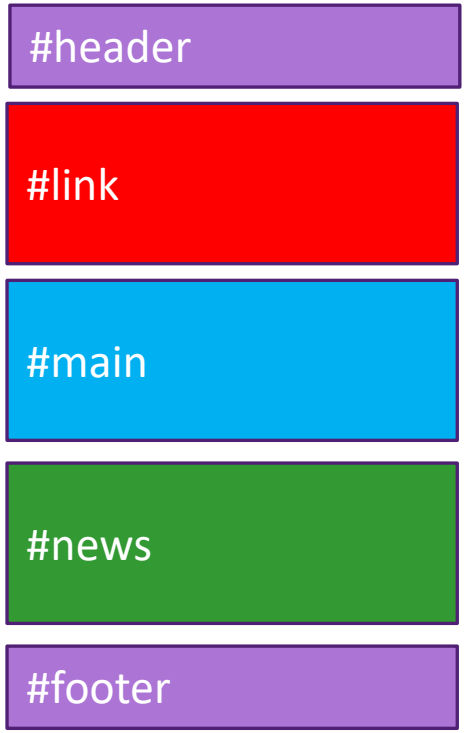
Set width on all three-column elements and float them to the left. Clear the footer to keep it at the bottom of the page.

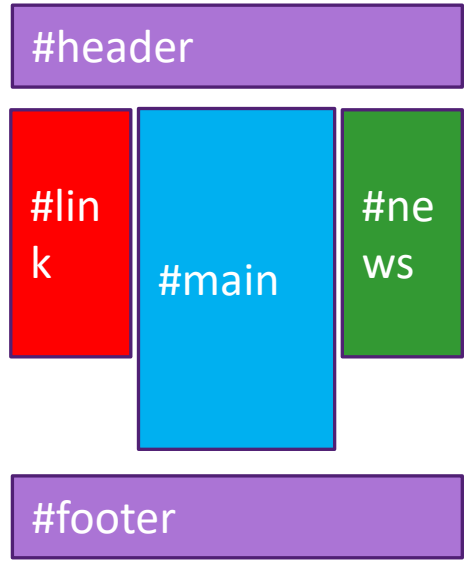
The markup

```
<div id="header">Masthead and headline</div>
<div id="link">List of Links</div>
<div id="main">Main article</div>
<div id="news">News Item</div>
<div id="footer">Copyright information</div>
```

The styles

```
#link {float: left; width: 22.5%; margin: 0 0 0 2.5%;}
#main {float: left; width:45% margin 0 2.5%}
#news {float: left; width:22.5%; margin: 0 2.5% 0 0; }
#footer {clear: left; }
```





Any order columns using negative margins

The strategy

Apply widths and floats to all three column elements, and use a negative margin to “drag” the left column across the page into the left position. Notice that although #main comes first in the source, it is in the second column position. In addition , the #links div (last in the source) is in the first column position on the left. This example is fixed, but you can do the same thing with a fluid layout using percentage values.

The markup

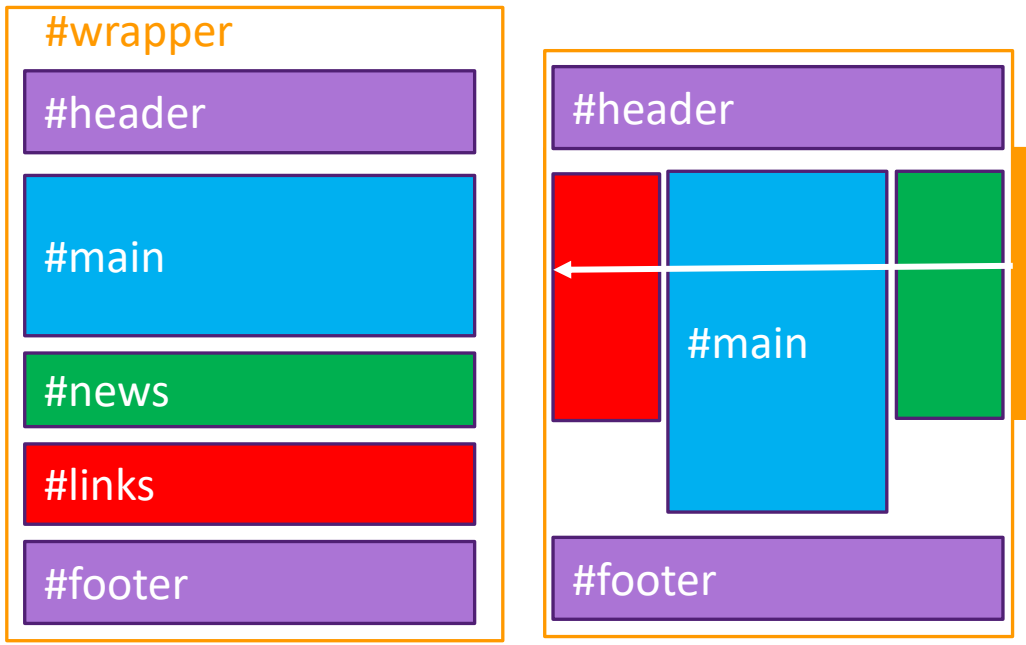
```
<div id=“wrapper”>
   <div id=“header”>Masthead and headline</div>
   <div id=“main”>Main article</div>
   <div id=“news”>News items</div>
   <div id=“links”>List of links</div>
   <div id=“footer”>Copyright information</div>
</div>
```

The styles

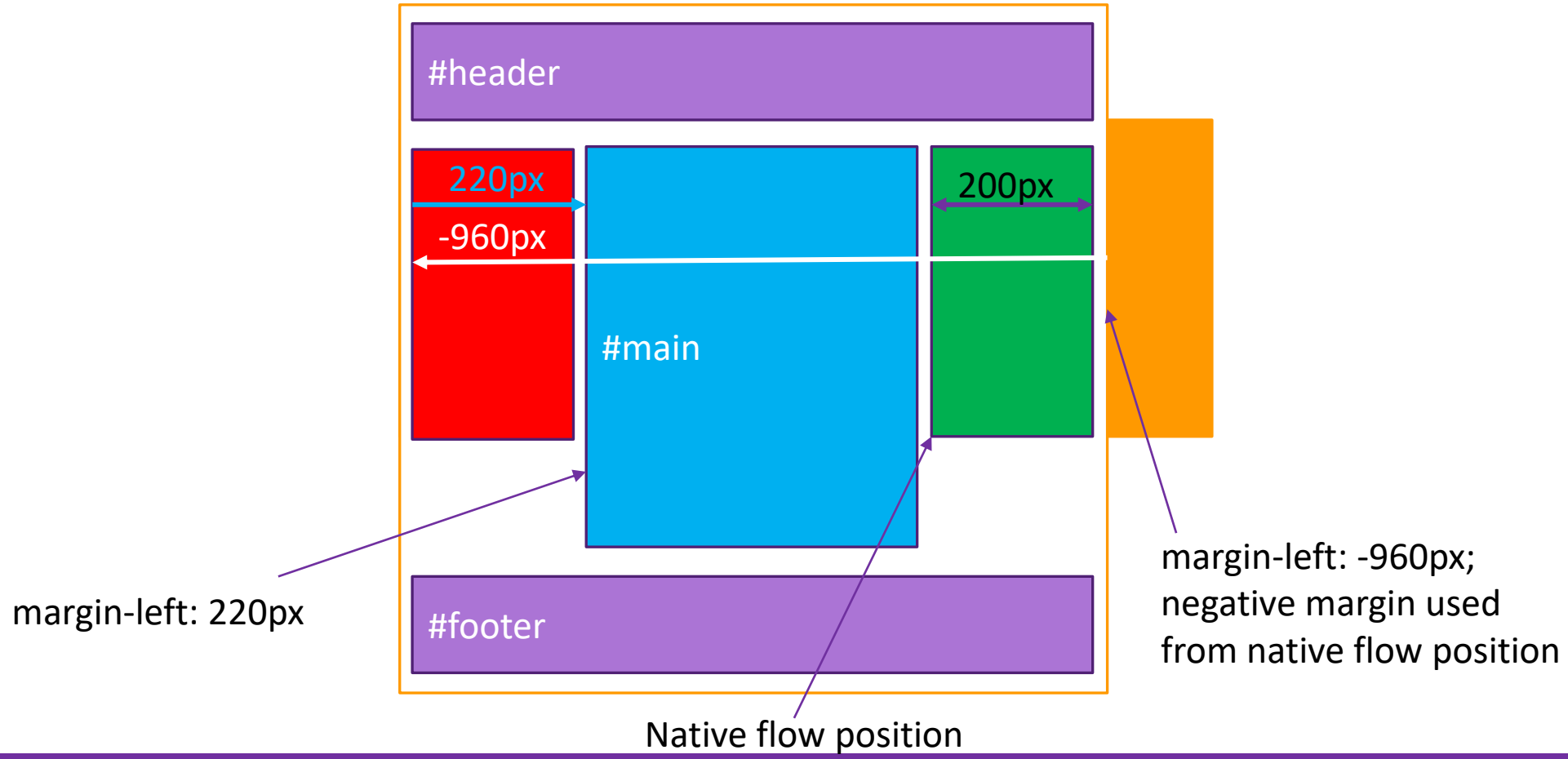
```
   #wrapper {width:960px; margin: 0 auto; }

   #main {float: left; width: 520px; margin-top: 0; margin-left: 200px; margin-left: 20px;}

   #news { float: left; width: 200px; margin: 0; }
   #links {float: left; width 200px; margin-top: 0; margin-left: -960px; }
   #footer {clear: left; }
```



Any order columns using negative margins



Three columns, positioned, fluid layout

The strategy

Wrap the three content divs (#main, #news, #links) in a div (#content) to serve as a containing block for the three positioned columns. Then, give the column elements widths and position them in the containing #content element.

The markup

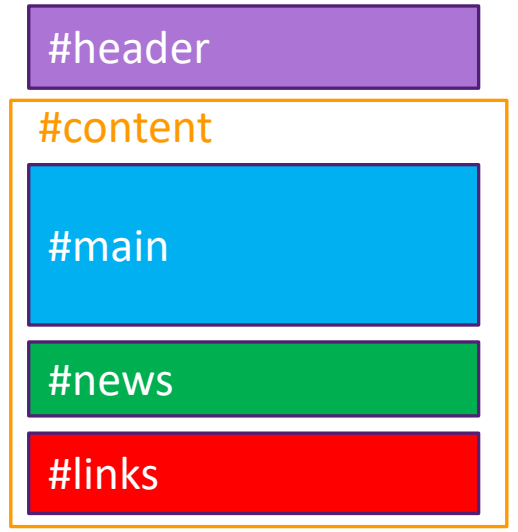
```
<div id="header">Masthead and headline</div>
<div id="content">
   <div id="main">Main article</div>
   <div id="news">News items</div>
   <div id="links">List of links</div>
</div>
```

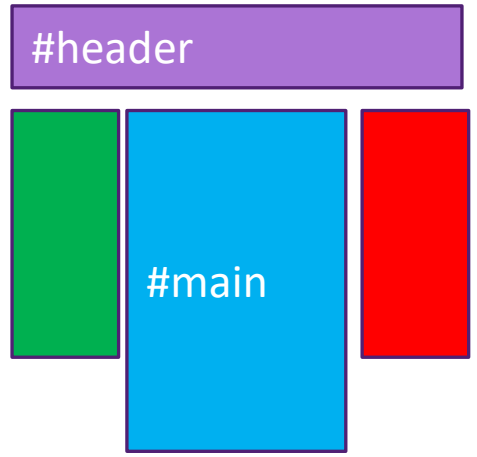
The styles

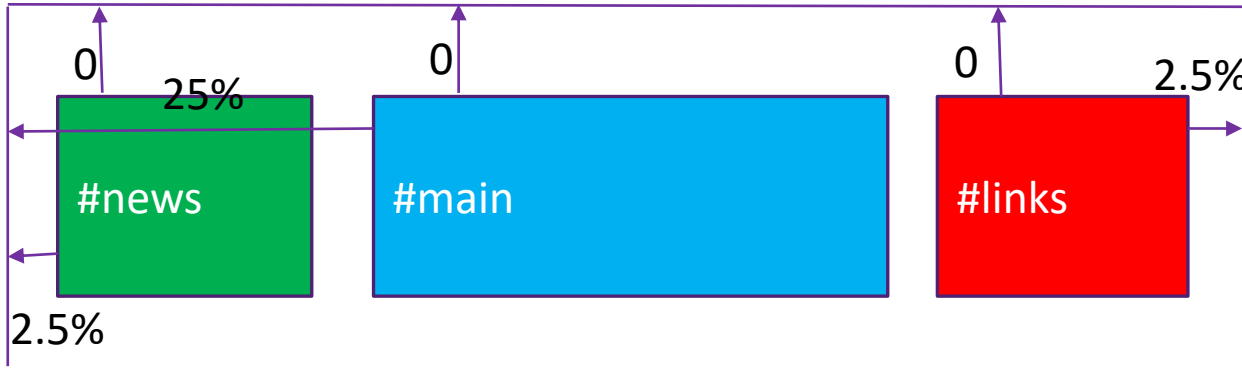
```
    #content {position: relative; margin: 0; }

    #main {width: 50%; position: absolute; top:0; left: 25%; margin: 0;}

    #news { width: 20%; position: absolute; top:0; left: 2.5%; margin: 0;
}
    #links {width:20%; position: absolute; top: 0; left: 2.5%; margin: 0;}
```







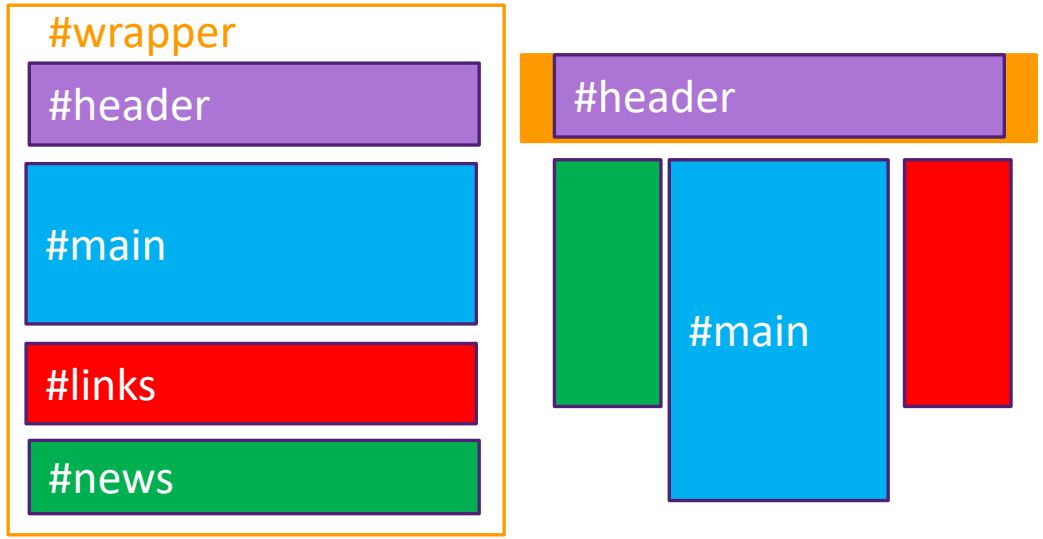
Three columns, positioned, fixed

The markup

```
<div id="wrapper">
<div id="header">Masthead and headline</div>
<div id="content">
   <div id="main">Main article</div>
  <div id="links">List of links</div>
   <div id="news">News items</div>
</div>
</div>
```

The styles

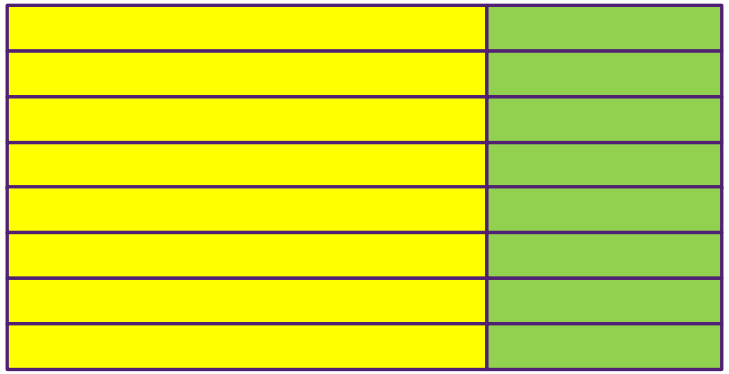
```
#wrapper {width: 960px; margin: 0 auto; }
#content {position: relative; margin: 0; }
#main {width: 520px; position: absolute; top:0; left: 220px; margin: 0;}
#news { width: 200px; position: absolute; top:0; left: 0; margin: 0; }
#links {width:200px; position: absolute; top: 0; right: 0px; margin: 0;}
```



Top-to-Bottom Backgrounds

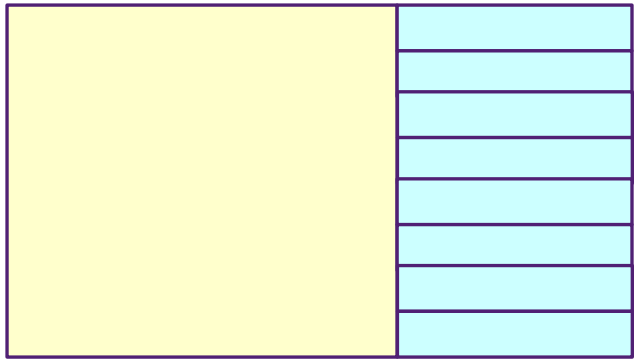
Vertical Repeated Background Image

```
#wrapper {
   width: 960px;
    margin: 0 auto;
    background-image: url(two_columns.png);
    background-repeat: repeat-y;
}
```



Faux columns for fluid layouts

```
body {
    background-image: url(two_cols_3000px.png);
    background-repeat: repeat-y;
    background-position: 76.5%;
}
```



Top-to-Bottom Backgrounds

Three Faux Columns

```
<div id="wrapper">
 <div id="inner">
   <div id="main"></div>
   <div id="links"></div>
   <div id="news"></div>
 </div>
</div>

#links {.....;  width: 26.25%;}
#news {.....; left: 73.25; width: 26.25; }
```