CSS Layouts

The contents and slides of this topic are used with permission from:

- Jennifer Robbins, Learning Web Design, O'Reilly, 5th edition, May 2018, ISBN 978-1-491-96020-2.
- Paul S. Wang, Dynamic Web programming and HTML5, Routledge, 1 edition, 2012, ISBN 1439871825.

Flexbox Layout

- Flexbox terminology
- Flexbox containers
- Flow: Flow direction and text wrap
- Alignment on main and cross axes
- Specifying how items in a flexbox "flex"
- Changing the order of flex items

About Flexbox

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

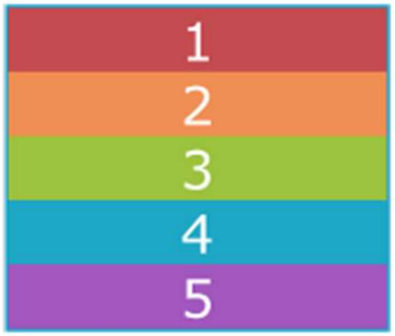
Flexbox Container

```
display: flex
```

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

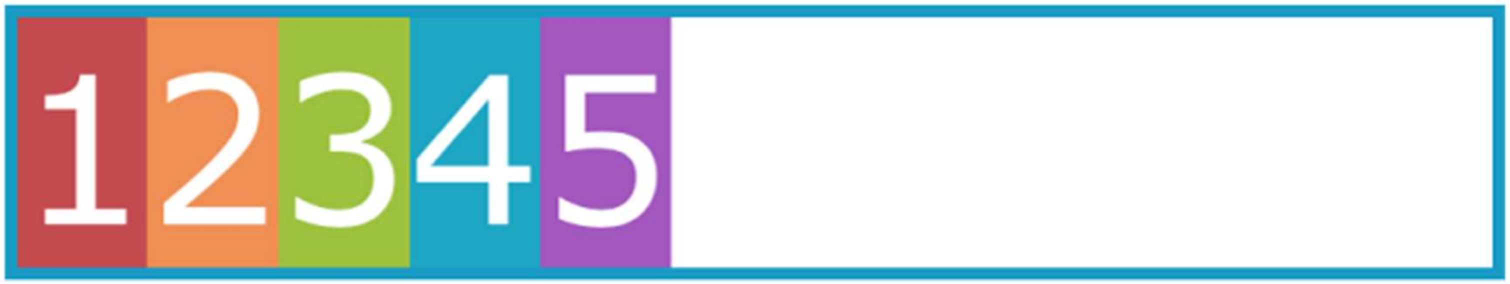
Flexbox Container (cont'd)

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

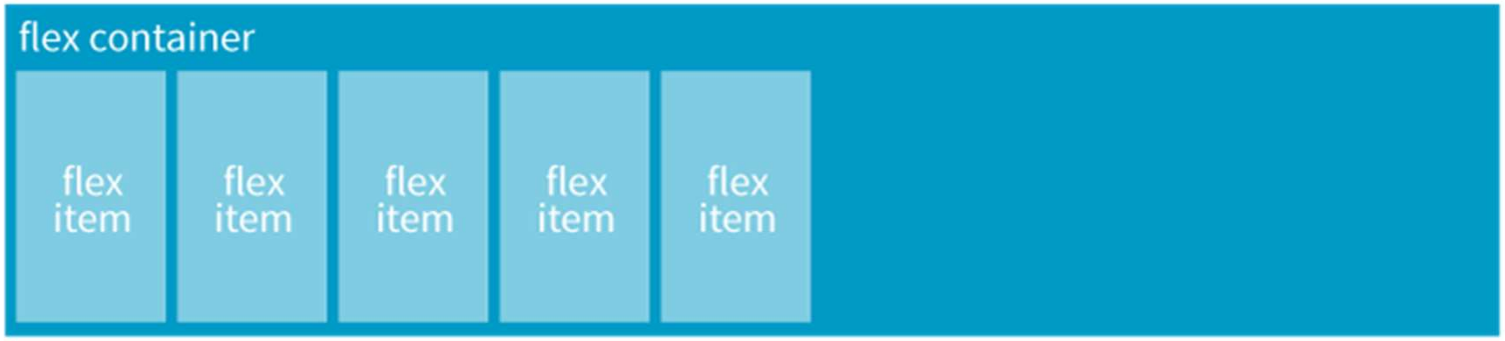


block layout mode

`display: flex;`



flexbox layout mode

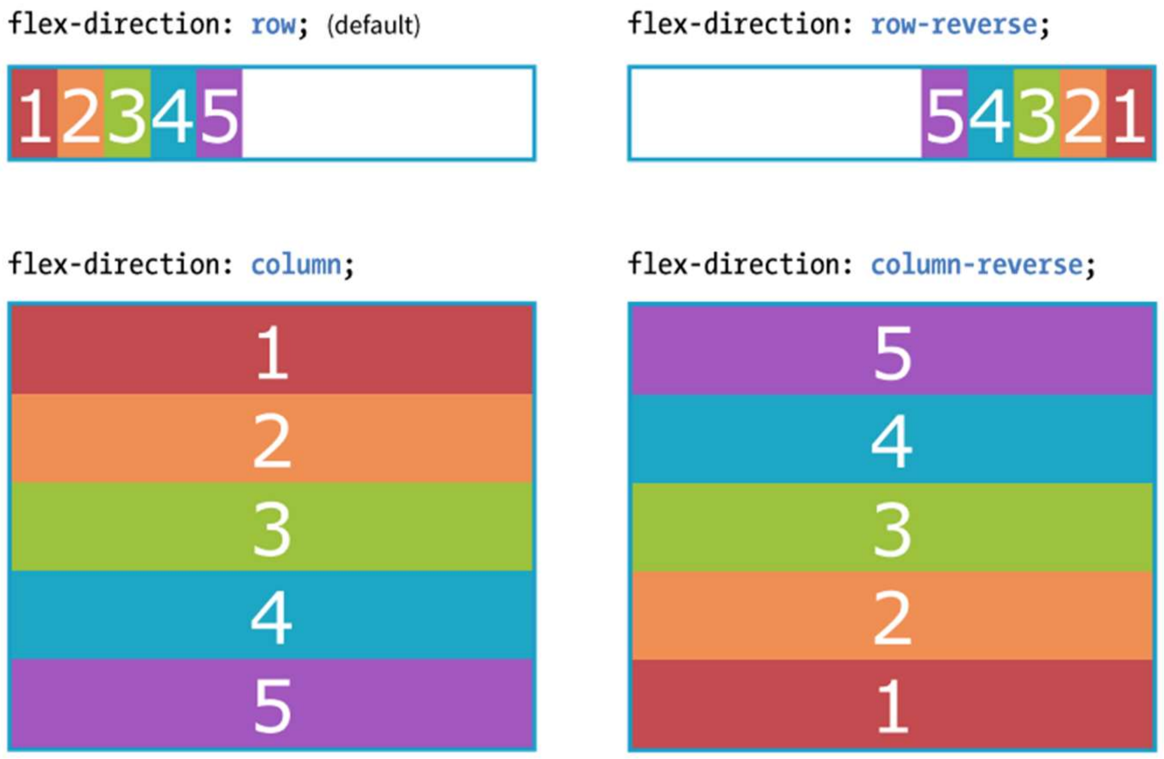


Rows and Columns (Direction)

`flex-direction`

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

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


flex-direction: row; (default)
12345
flex-direction: row-reverse;
54321
flex-direction: column;
1
2
3
4
5
flex-direction: column-reverse;
5
4
3
2
1

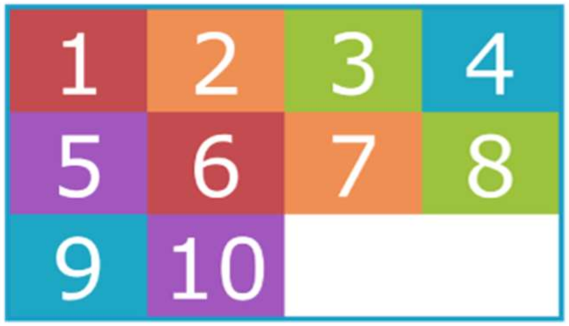
Wrapping Flex Lines

`flex-wrap`

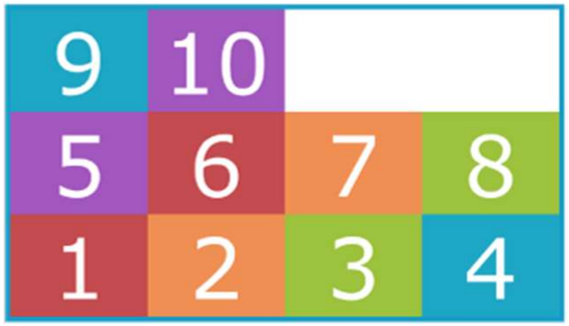
Values: `wrap`, `nowrap`, `wrap-reverse`

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

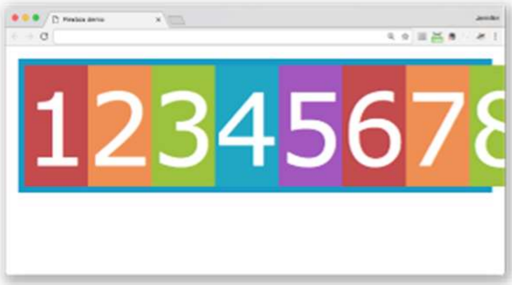
`flex-wrap: wrap;`



`flex-wrap: wrap-reverse;`



`flex-wrap: nowrap;` (default)



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

Flex Flow (Direction + Wrap)

`flex-flow`

Values: *Flex-direction flex-wrap*

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

Example

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

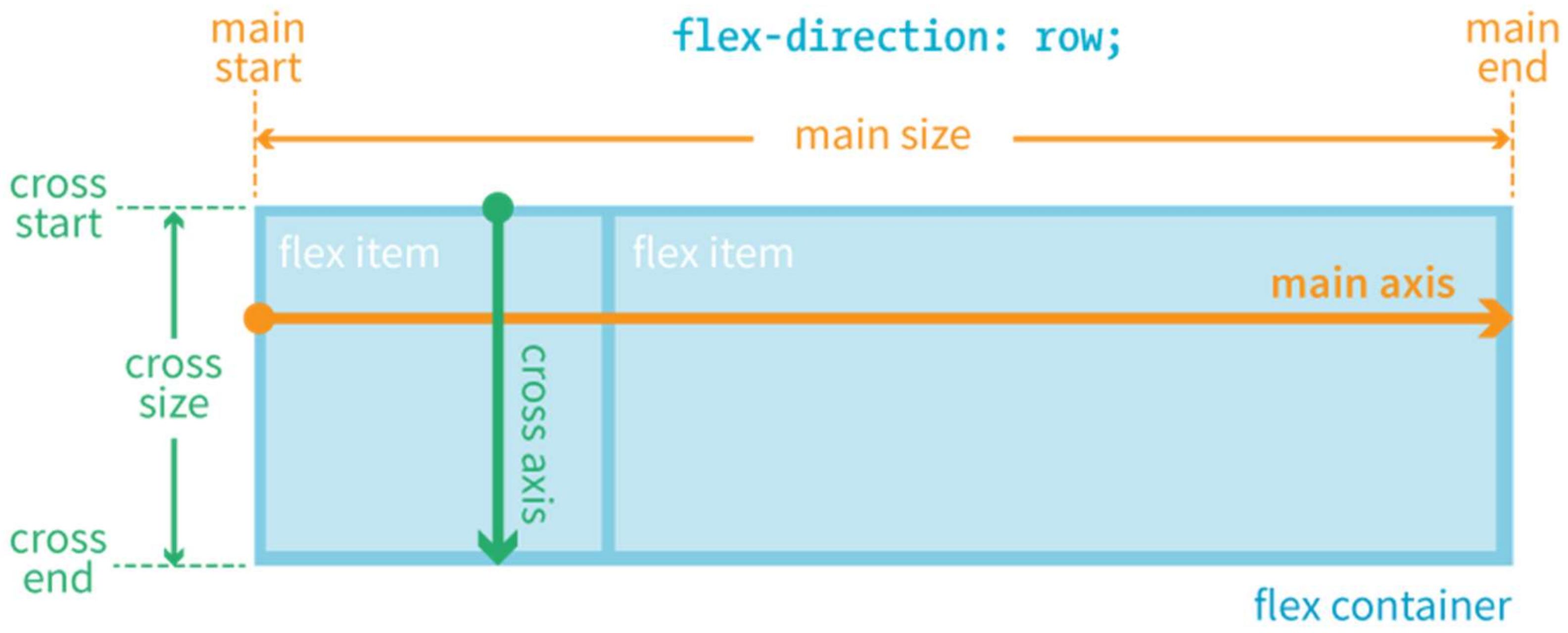
Flexbox Alignment Terminology

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

ROW: Main and Cross Axes

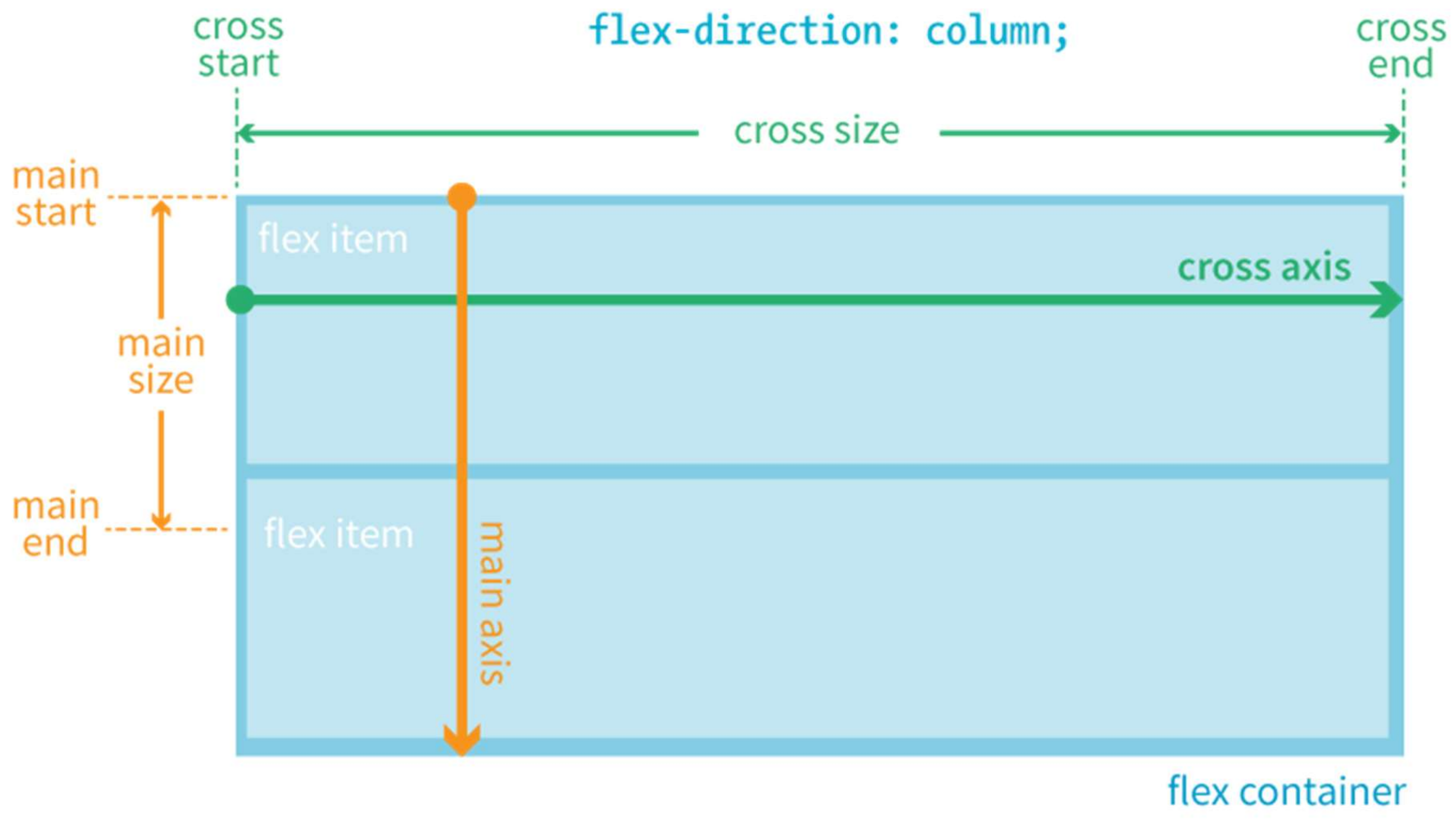
FOR LANGUAGES THAT READ HORIZONTALLY FROM LEFT TO RIGHT:

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



COLUMN: Main and Cross Axes

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



Aligning on the Main Axis

`justify-content`

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

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

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

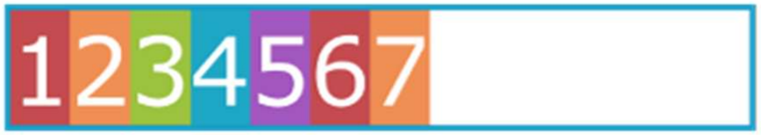
Example:

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

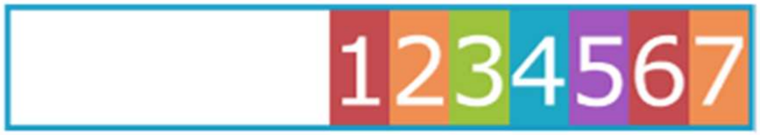
Aligning on the Main Axis (cont'd)

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

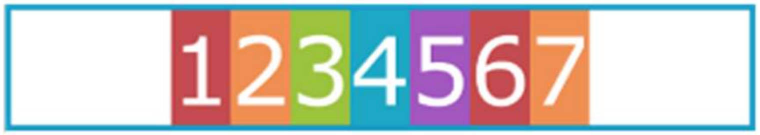
`justify-content: flex-start;` (default)



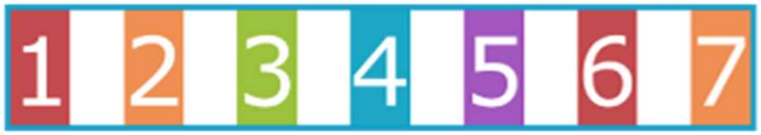
`justify-content: flex-end;`



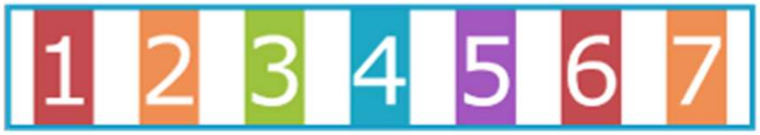
`justify-content: center;`



`justify-content: space-between;`



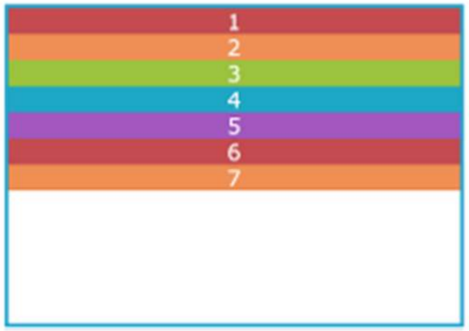
`justify-content: space-around;`



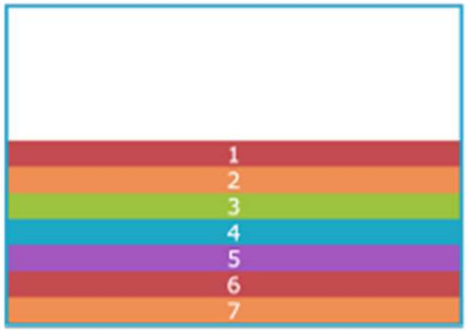
Aligning on the Main Axis (cont’d.)

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

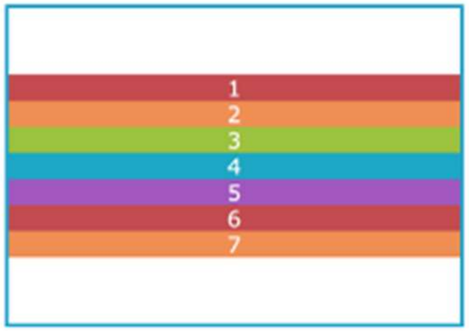
`justify-content: flex-start;` (default)



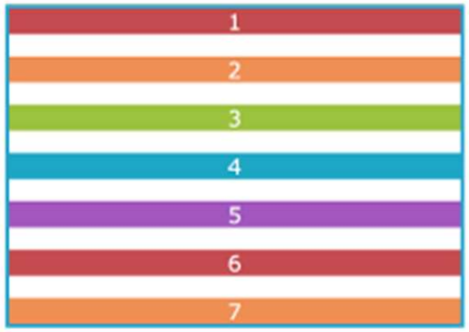
`justify-content: flex-end;`



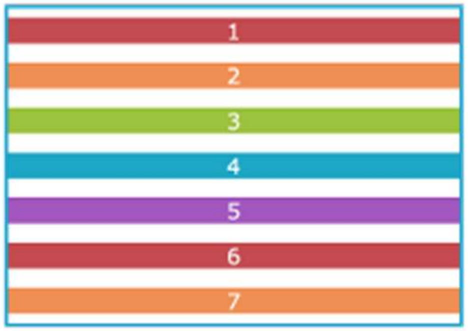
`justify-content: center;`



`justify-content: space-between;`



`justify-content: space-around;`



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

Specifying How Items “Flex”

`flex`

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

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

flex Property Example

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

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

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

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

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

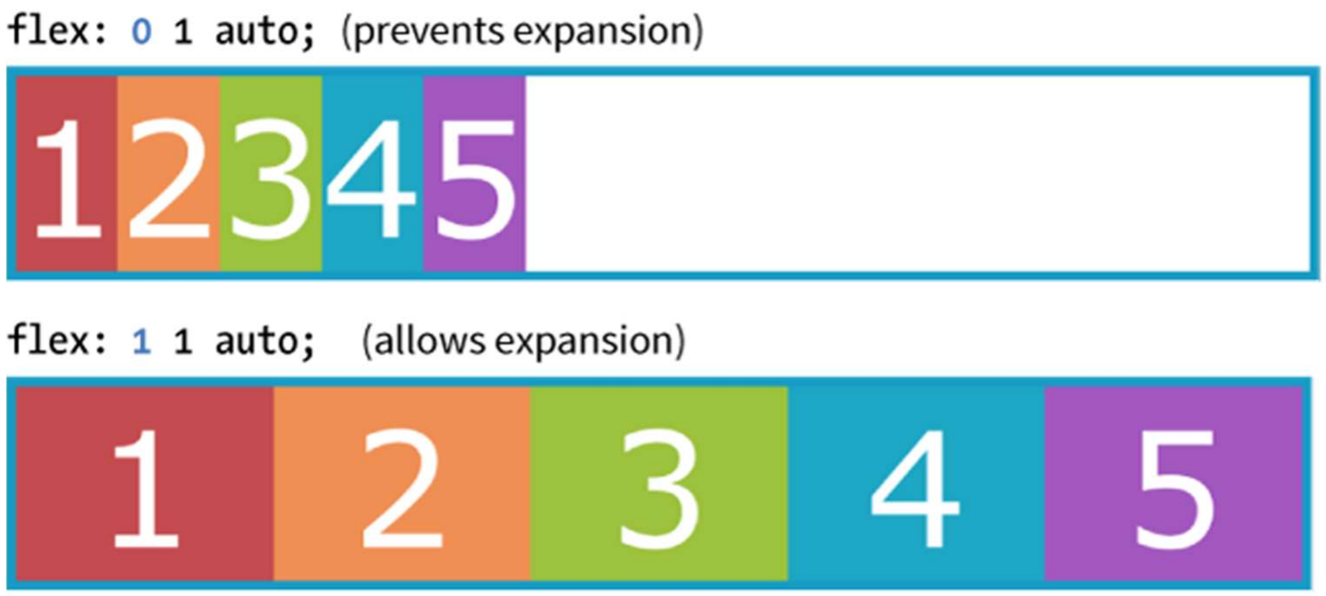
Expanding Items (flex-grow)

`flex-grow`

Values: *Number*

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

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


flex: 0 1 auto; (prevents expansion)
1
2
3
4
5
flex: 1 1 auto; (allows expansion)
1
2
3
4
5

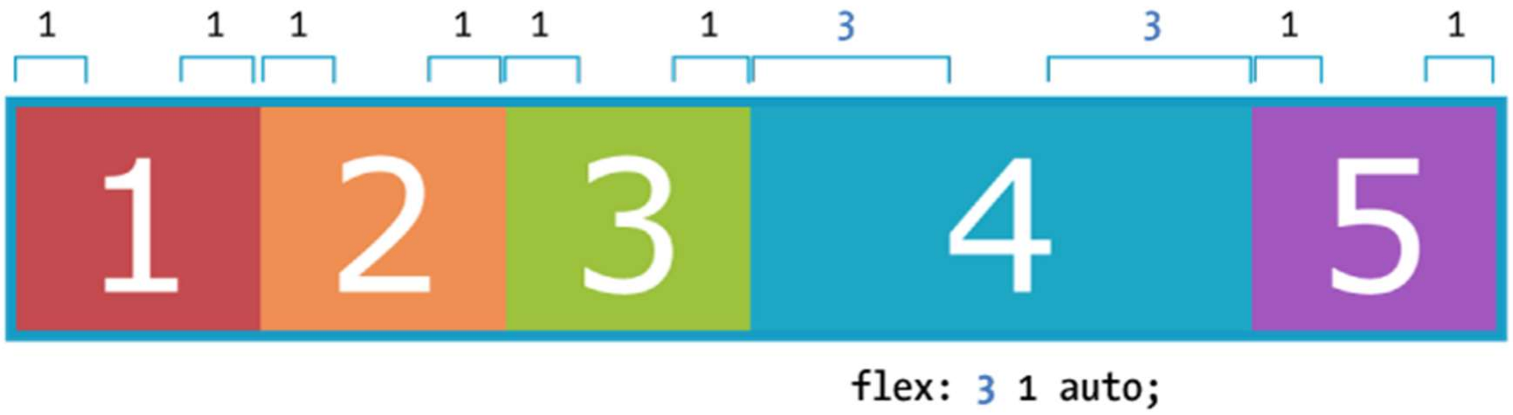
Expanding Items (cont'd)

Relative Flex

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

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

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



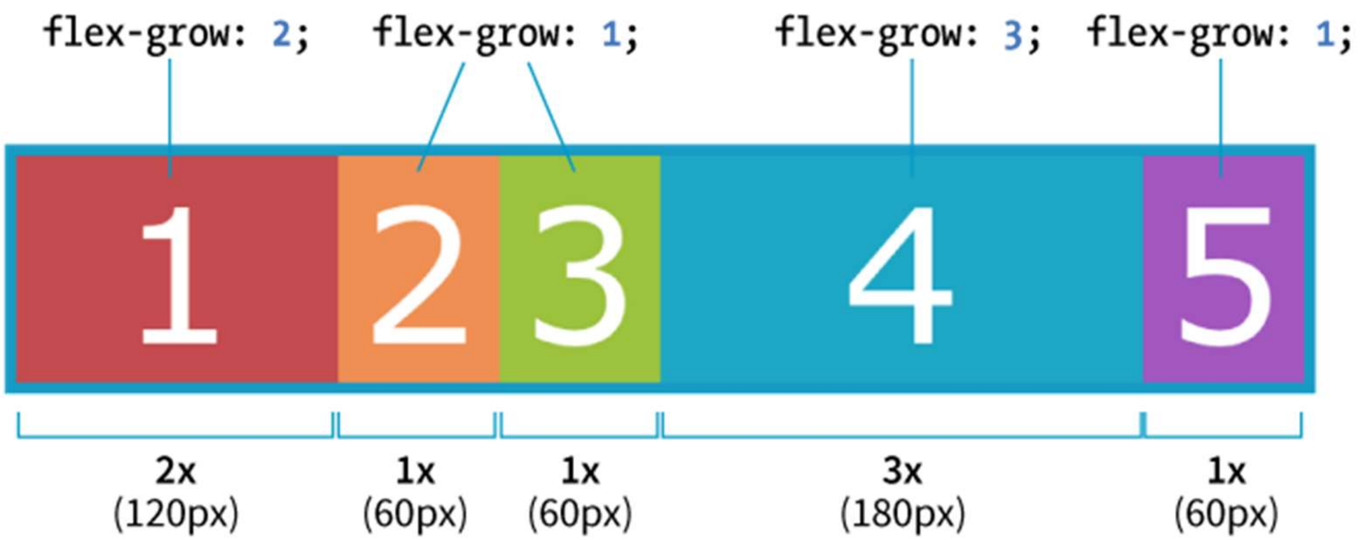
Expanding Items (cont'd.)

Absolute Flex

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

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

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



Shortcut flex Values

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

Browser Support for Flexbox

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

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

Flexbox Property Review

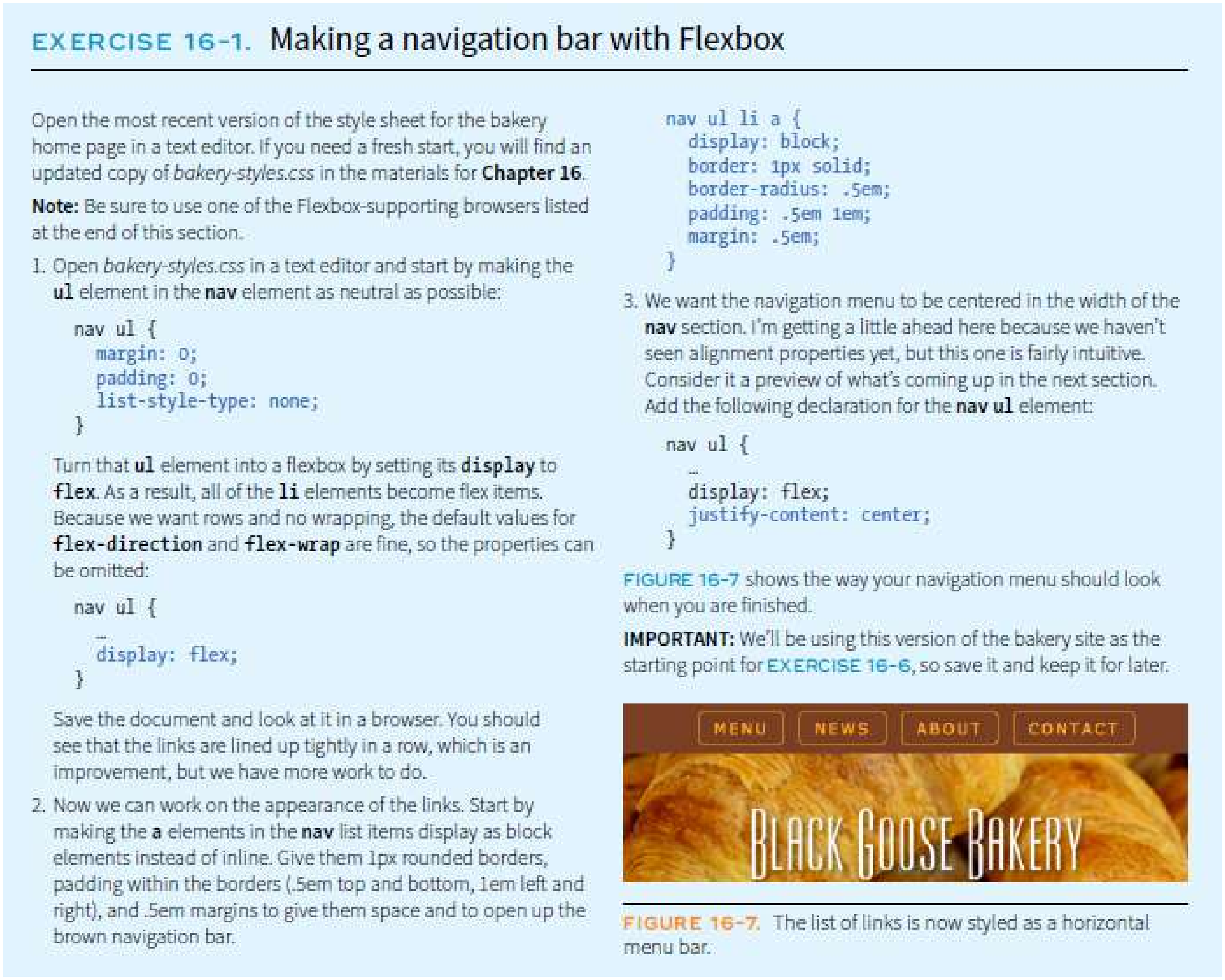
Flex container properties

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

Flex item properties

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

Example



EXERCISE 16-1. Making a navigation bar with Flexbox

Open the most recent version of the style sheet for the bakery home page in a text editor. If you need a fresh start, you will find an updated copy of *bakery-styles.css* in the materials for **Chapter 16**.

Note: Be sure to use one of the Flexbox-supporting browsers listed at the end of this section.

1. Open *bakery-styles.css* in a text editor and start by making the **ul** element in the **nav** element as neutral as possible:

```
nav ul {
  margin: 0;
  padding: 0;
  list-style-type: none;
}
```

Turn that **ul** element into a flexbox by setting its **display** to **flex**. As a result, all of the **li** elements become flex items. Because we want rows and no wrapping, the default values for **flex-direction** and **flex-wrap** are fine, so the properties can be omitted:

```
nav ul {
  …
  display: flex;
}
```

Save the document and look at it in a browser. You should see that the links are lined up tightly in a row, which is an improvement, but we have more work to do.

2. Now we can work on the appearance of the links. Start by making the **a** elements in the **nav** list items display as block elements instead of inline. Give them 1px rounded borders, padding within the borders (.5em top and bottom, 1em left and right), and .5em margins to give them space and to open up the brown navigation bar.

```
nav ul li a {
  display: block;
  border: 1px solid;
  border-radius: .5em;
  padding: .5em 1em;
  margin: .5em;
}
```

3. We want the navigation menu to be centered in the width of the **nav** section. I'm getting a little ahead here because we haven't seen alignment properties yet, but this one is fairly intuitive. Consider it a preview of what's coming up in the next section. Add the following declaration for the **nav ul** element:

```
nav ul {
  …
  display: flex;
  justify-content: center;
}
```

FIGURE 16-7 shows the way your navigation menu should look when you are finished.

IMPORTANT: We'll be using this version of the bakery site as the starting point for EXERCISE 16-6, so save it and keep it for later.

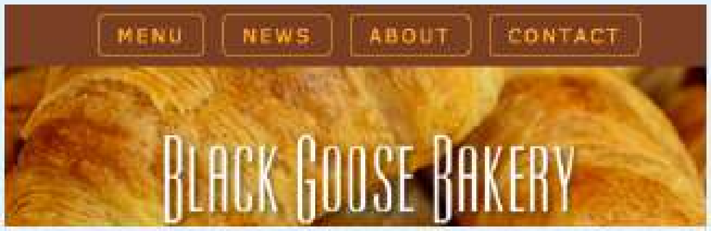


FIGURE 16-7. The list of links is now styled as a horizontal menu bar.

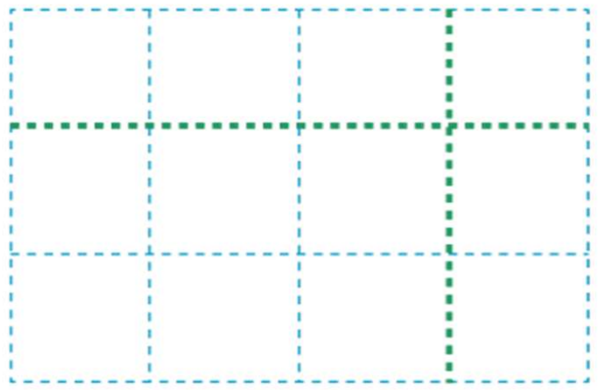
Grid Layout

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

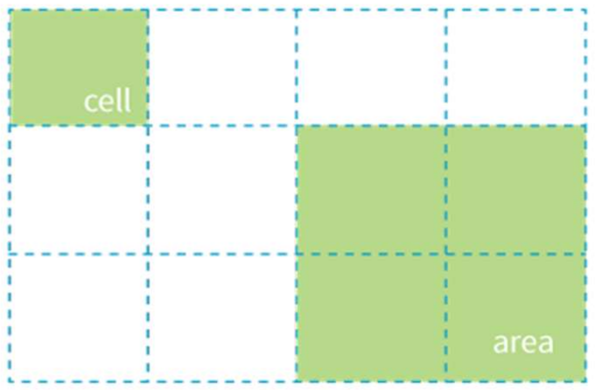
How CSS Grids Work

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

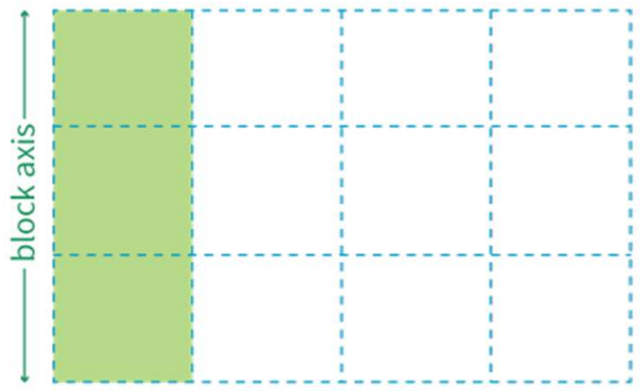
Grid Terminology



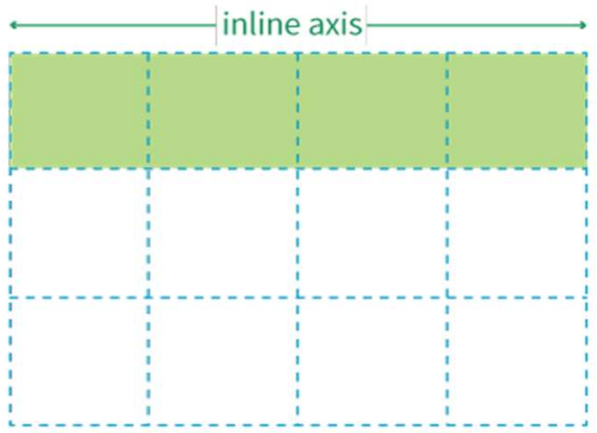
grid lines



grid cell and grid area



grid track (column)



grid track (row)

Creating a Grid Container

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

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

The markup

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

The styles

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

Defining Row and Column Tracks

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

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

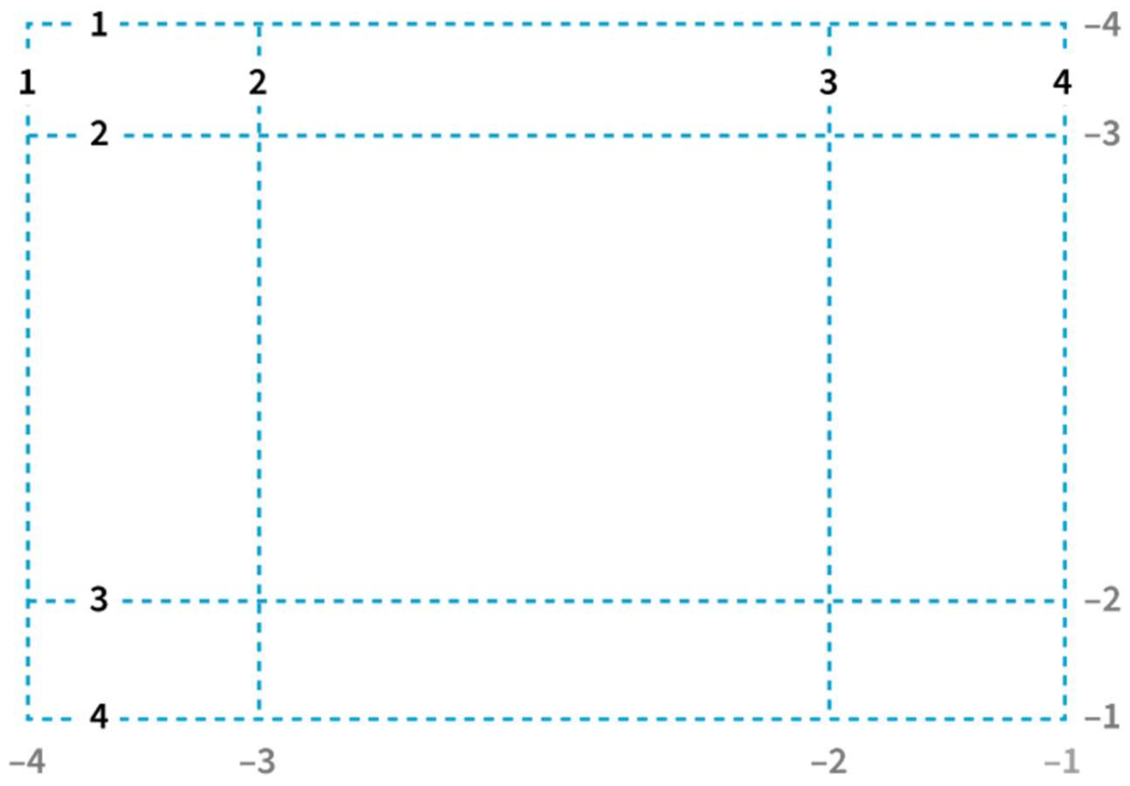
- The value of **`grid-tempate-rows`** is a list of the *heights* of each row track in the grid.
- The value of **`grid-template-columns`** is a list of the *widths* of each column track in the grid.

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

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

Grid Line Numbers

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



Grid Line Names

You can also assign names to lines to make them more intuitive to reference later.

Grid line names are added in square brackets in the position they appear relative to the tracks.

To give a line more than one name, include all the names in brackets, separated by spaces.

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

Track Size Values

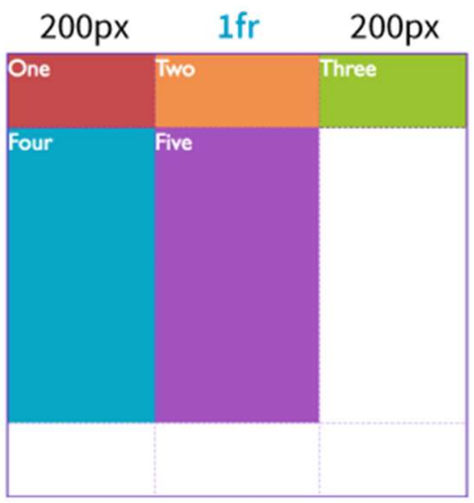
The CSS Grid spec provides a *lot* of ways to specify the width and height of a track. Some of these ways allow tracks to adapt to available space and/or to the content they contain:

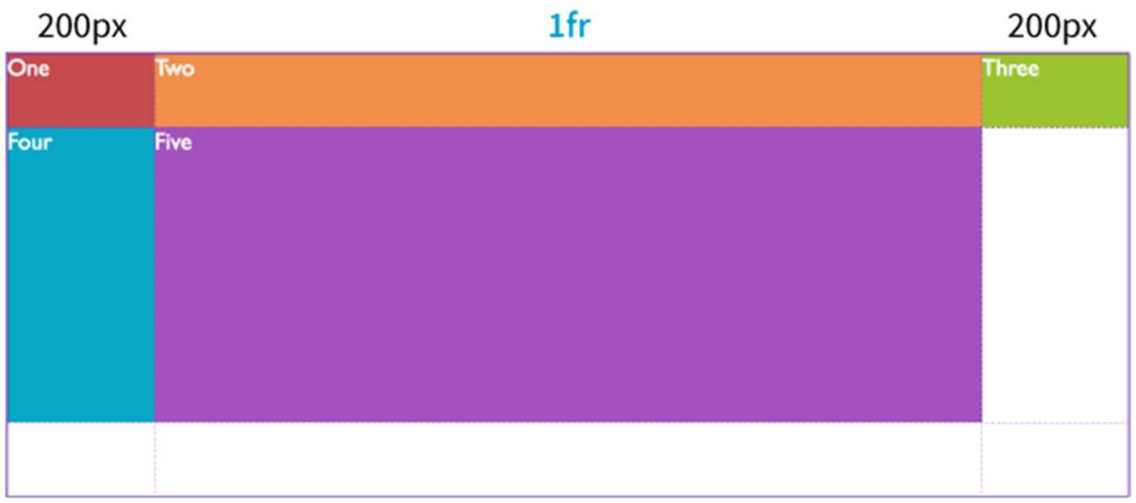
- Lengths (such as `px` or `em`)
- Percentage values (%)
- Fractional units (`fr`)
- `minmax()`
- `min-content, max-content`
- `auto`
- `fit-content()`

Fractional Units (fr)

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

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





Size Range with minmax()

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

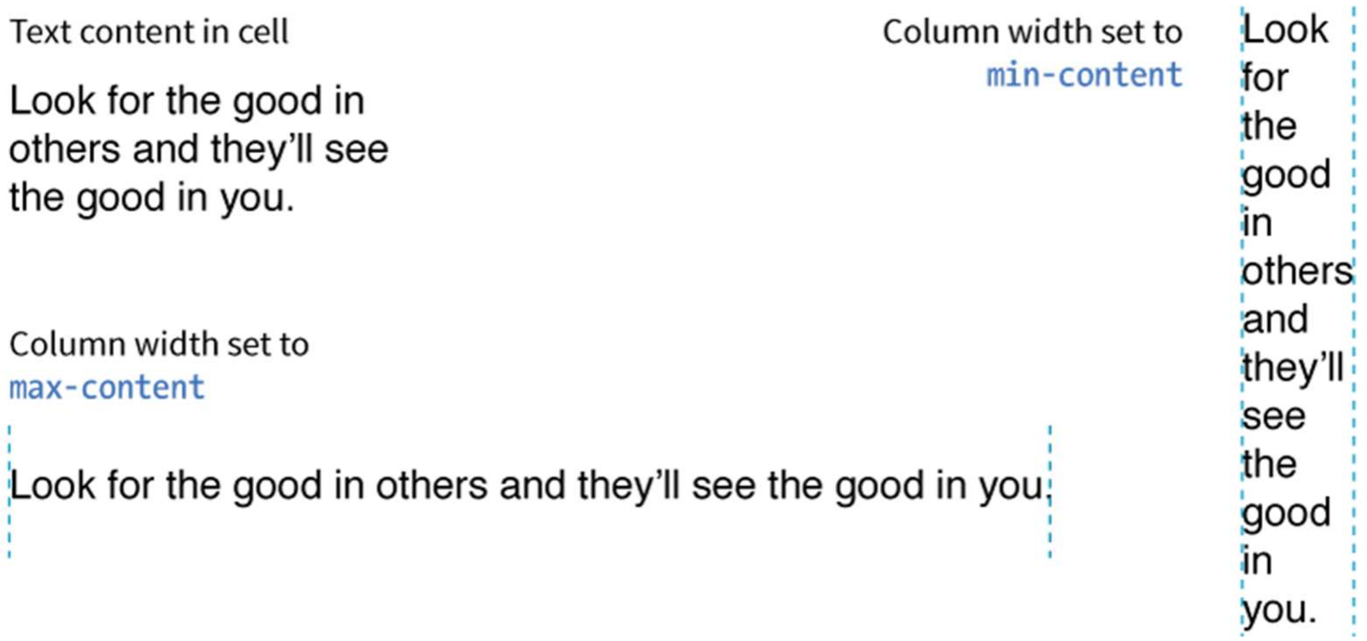
```
grid-template-columns: 200px minmax(15em, 45em) 200px;
```

min-content and max-content

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

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

`auto` lets the browser take care of it. Start with **`auto`** for content-based sizing.



Text content in cell

Look for the good in others and they'll see the good in you.

Column width set to `max-content`

Look for the good in others and they'll see the good in you.

Column width set to `min-content`

Look
for
the
good
in
others
and
they'll
see
the
good
in
you.

Repeating Track Sizes

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

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

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

BEFORE:

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

AFTER:

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

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

Giving Names to Grid Areas

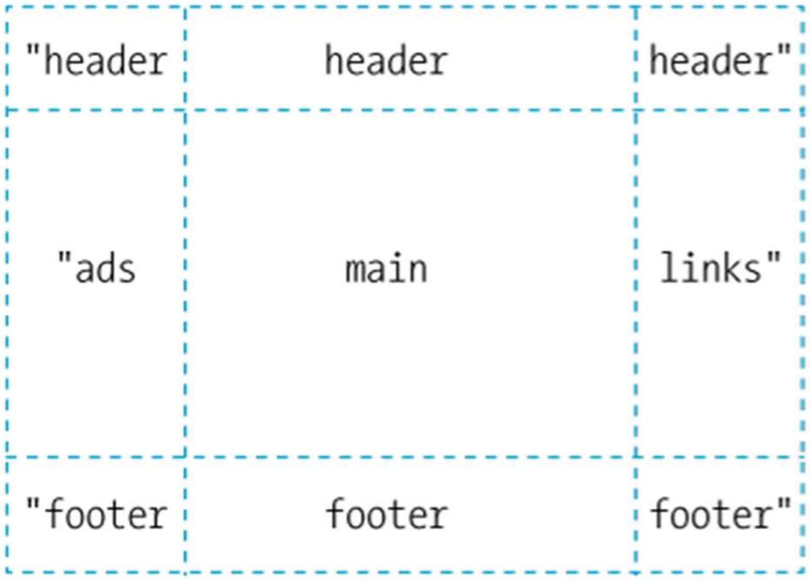
`grid-template-areas`

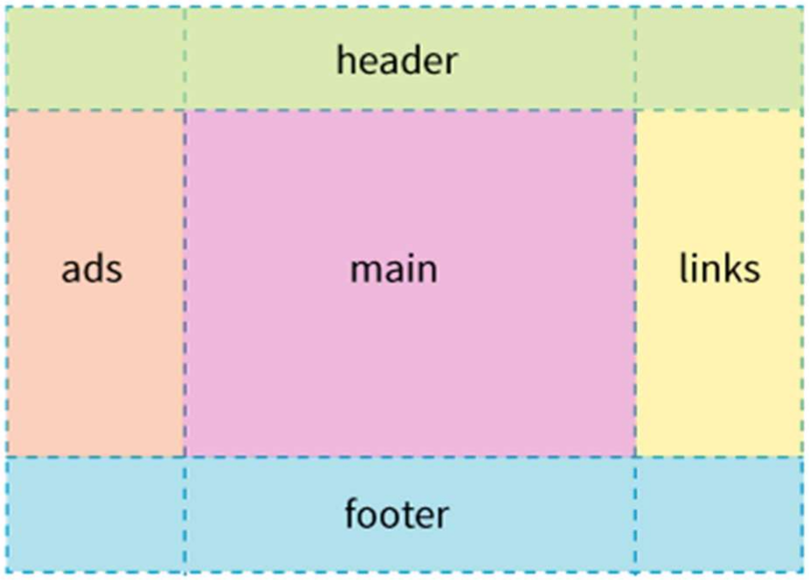
Values: `none`, *series of area names by row*

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

Giving Names to Grid Areas (cont’d)

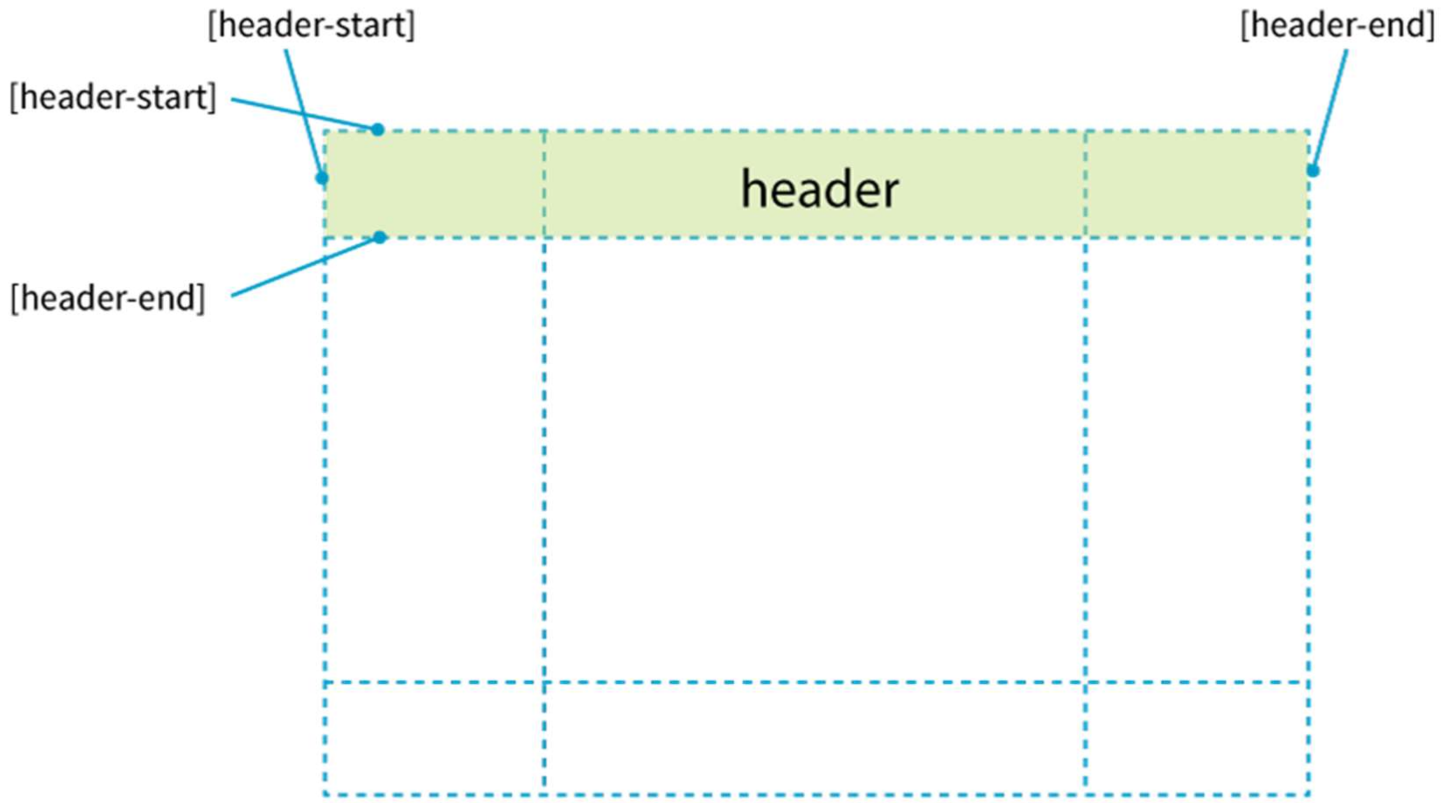
```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px [footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer"
}
```





Giving Names to Grid Areas (cont'd)

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



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

The grid Shorthand Property

`grid`

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

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

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

The grid Shorthand Property (cont’d)

Put the row-related values before the slash (/) and column-related values after:

```
grid: rows / columns
```

Example:

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

The grid Shorthand Property (cont’d)

You can include line names and area names as well, in this order:

```
[start line name] "area names" <track size> [end line name]
```

Example:

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

The names and height for each row are stacked here for clarity. Note that the column track information is still after the slash (/).

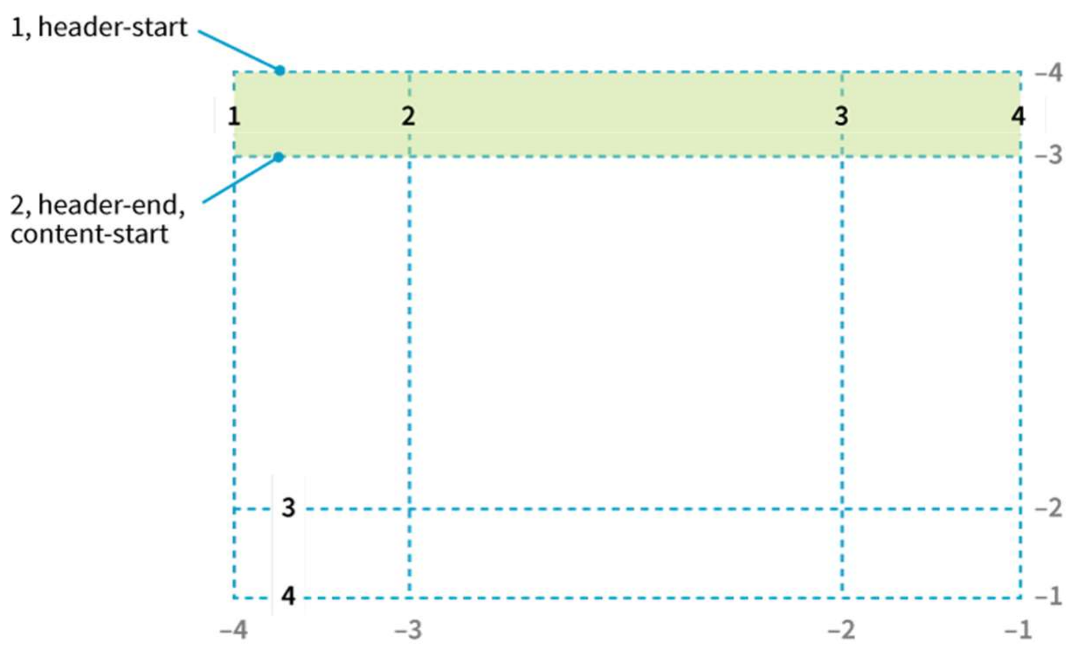
Placing Items Using Grid Lines

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

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

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

Placing Items on the Grid (cont'd)



By line number:

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

Using a span:

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

Starting from the last grid line and spanning backward:

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

By line name:

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

Placing Items on the Grid (cont’d)

`grid-row`
`grid-column`

Values: *"start line” / “end line”*

These shorthand properties combine the `*-start` and `*-end` properties into a single declaration. Values are separated by a slash (/):

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

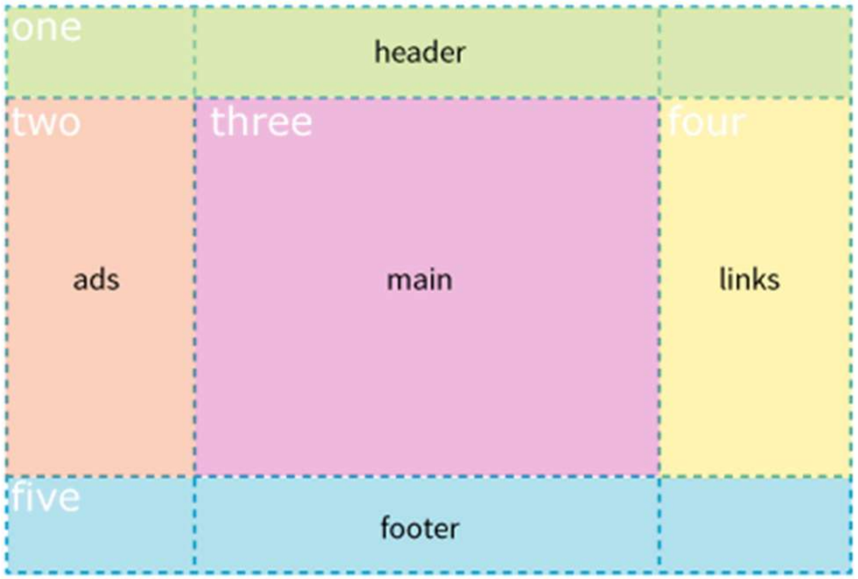
Placing Items on the Grid Using Areas

grid-area

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

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

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



Flow Direction and Density

`grid-auto-flow`

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

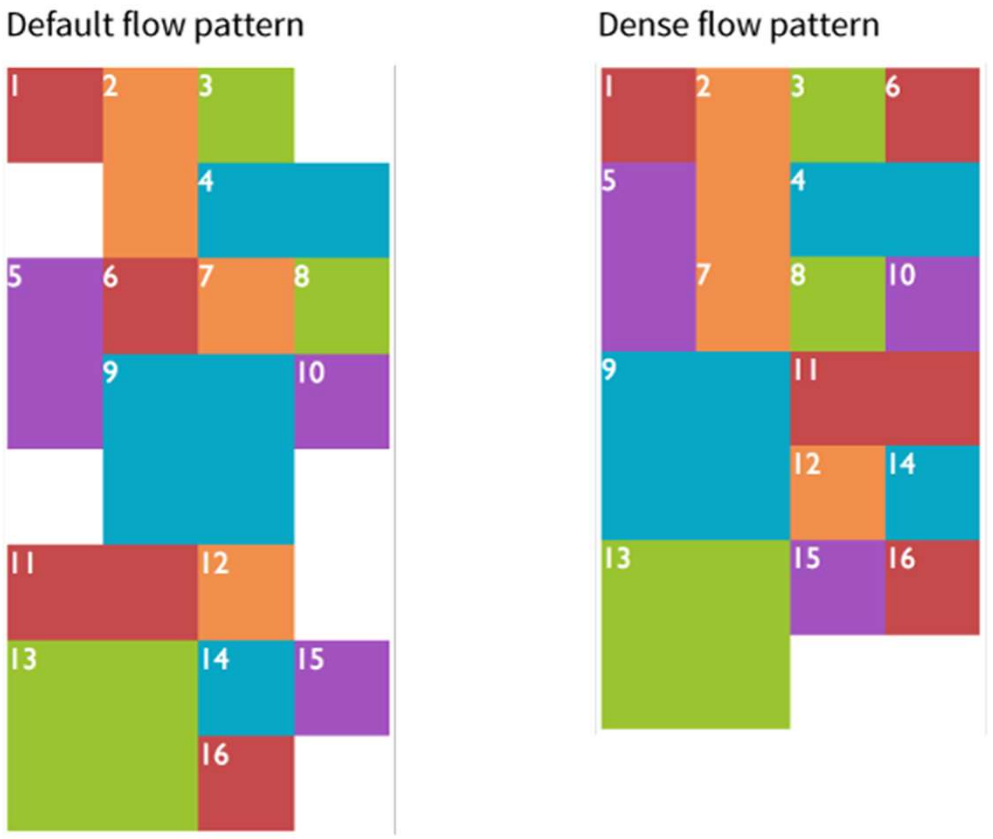
Specifies whether you'd like items to flow in by **`row`** or **`column`**. The default is the writing direction of the document.

Example:

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

Flow Direction and Density (cont'd)

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



Spacing Between Tracks

`grid-row-gap`
`grid-column-gap`

Values: *Length (must not be negative)*

`grid-gap`

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

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

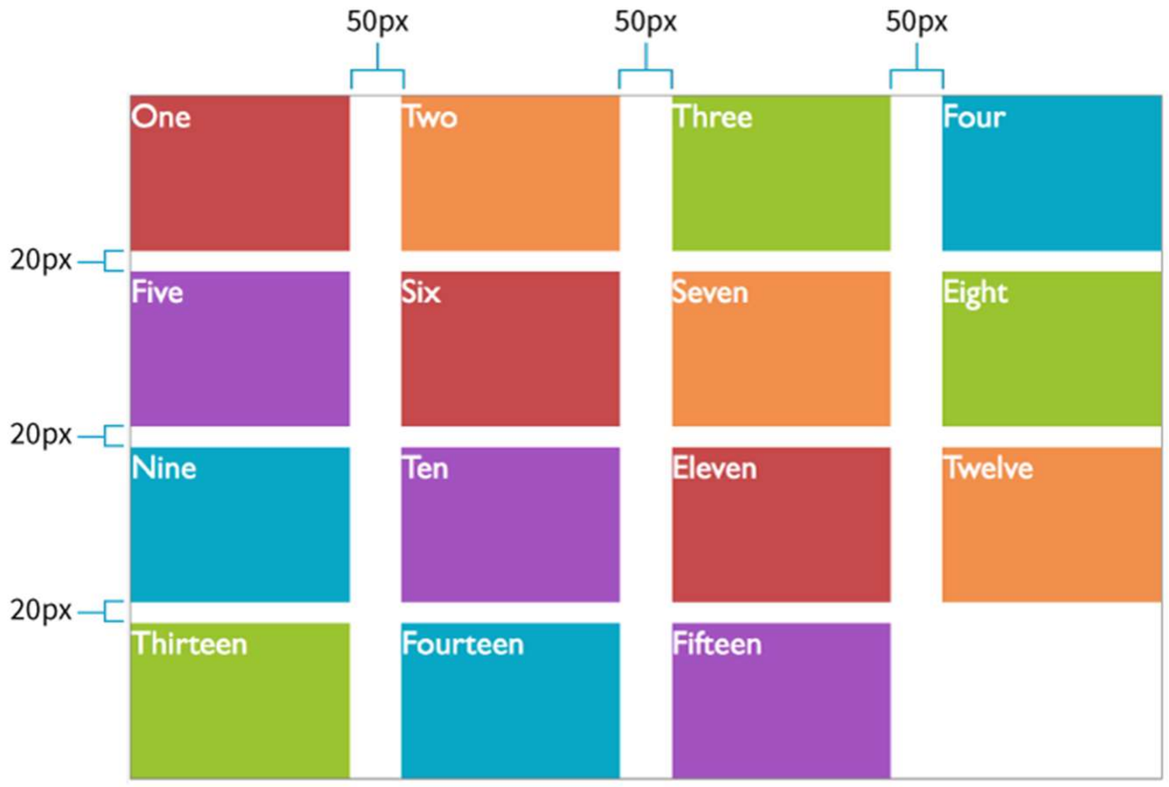
NOTE: These property names will be changing to `row-gap`, `column-gap`, and `gap`, but the new names are not yet supported.

Space Between Tracks (cont’d)

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

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

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



Item Alignment

`justify-self`
`align-self`

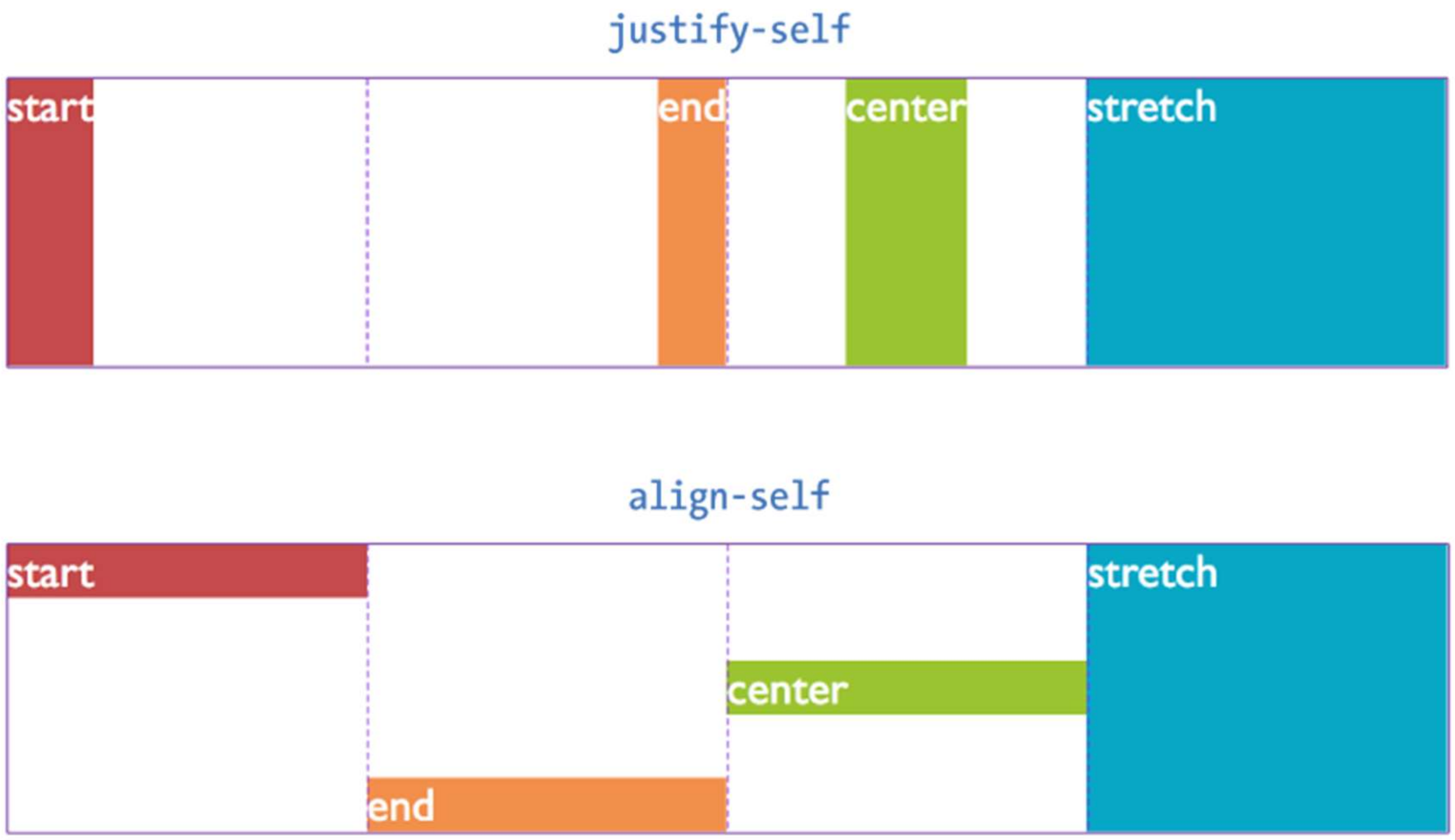
Values: `start`, `end`, `center`, `left`, `right`, `self-start`, `self-end`, `stretch`, `normal`, `auto`

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

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

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

Item Alignment (cont’d)



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

Aligning All the Items

`justify-items`
`align-items`

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

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

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

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

Track Alignment

`justify-content`
`align-content`

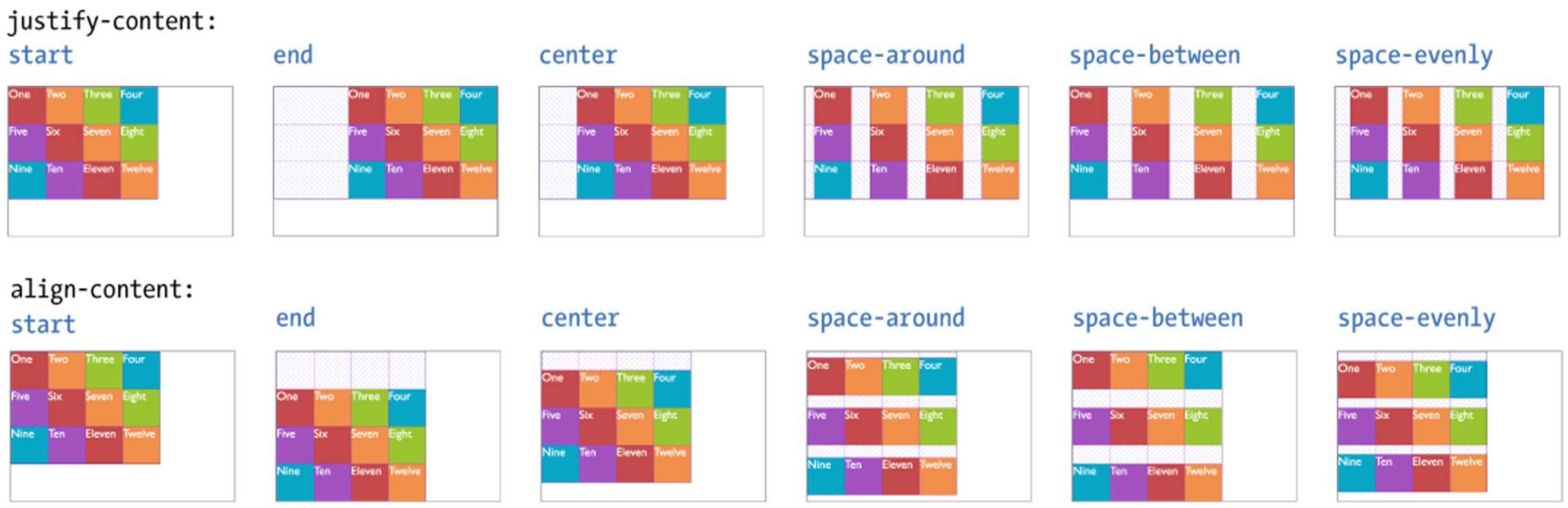
Values: `start`, `end`, `center`, `left`, `right`, `stretch`, `space-around`, `space-between`, `space-evenly`

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

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

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

Track Alignment (cont’d)



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

Grid Property Review

Grid container properties

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

Grid item properties

```
grid-column
   grid-column-start
   grid-column-end
grid-row
   grid-row-start
   grid-row-end
grid-area
justify-self
align-self
```

`order` (not part of Grid Module)

`z-index` (not part of Grid Module)

Example

EXERCISE 16-6. A grid layout for the bakery page

The Black Goose Bakery page has come a long way. You've added padding, borders, and margins. You've floated images, positioned an award graphic, and created a navigation bar by using Flexbox. Now you can use your new grid skills to give it a two-column layout that would be appropriate for tablets and larger screens (FIGURE 16-47).

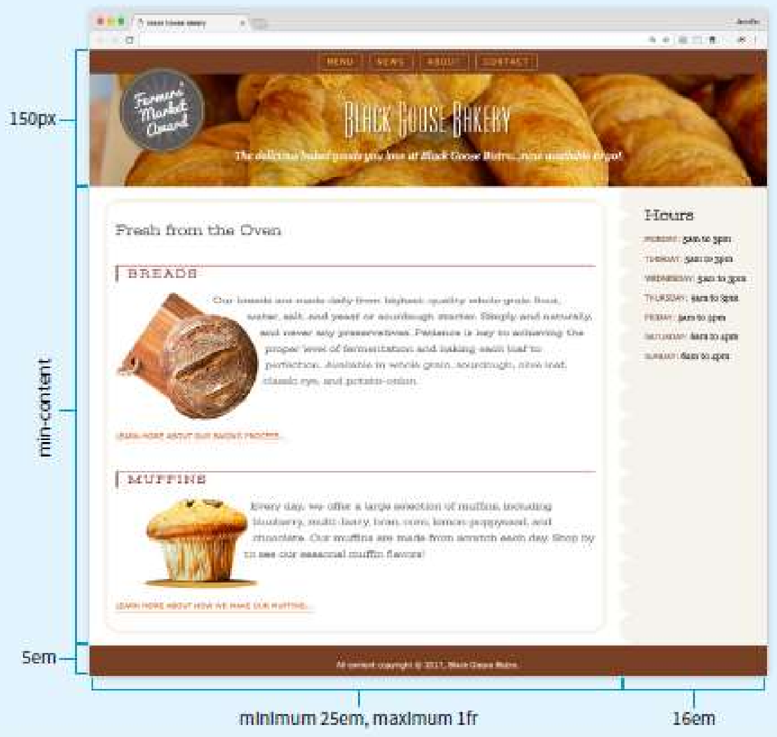


FIGURE 16-47. The Black Goose Bakery page with a two-column grid layout.

Start by opening the bakery file as you left it in EXERCISE 16-1.

1. We need to add a bit of markup that encloses everything in the body of the document in an element that will serve as the grid container. Open the HTML document *bakery.html*, add a **div** around all of the content elements (from **header** to **footer**), and give it the **id** "container". Save the HTML file.

```
<body>
  <div id="container">
    <header>…</header>
    <main>…</main>
    <aside>…</aside>
    <footer>…</footer>
  </div>
</body>
```

2. In the style sheet (*bakery-styles.css*), add a new style to make the new **div** display as a grid:

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

3. First we'll work on the rows. FIGURE 16-47 shows that we need three rows to create the layout. Set the height of the first row track to **auto** so it will observe the height settings on the elements within it and automatically accommodate the content. The second row has a lot of text, so use the **auto** track value again to guarantee the track will expand at least as much as necessary to fit the text. For the third row, a height of 5em should be sufficient to fit the few lines of text with a comfortable amount of space:

```
  #container {
    display: grid;
    grid-template-rows: auto auto 5em;
  }
```

4. Now we can set up the column tracks. It looks like we'll need only two: one for the main content and one for the Hours sidebar. I've used the **minmax()** value so I can ensure the text column never gets narrower than 25em, but it can expand to fill the available space in the browser (**1fr**). The Hours column feels right to me at 16em. Feel free to try other values.

```
  #container {
    display: grid;
    grid-template-rows: auto auto 5em;
    grid-template-columns: minmax(25em, 1fr) 16em;
  }
```

Next, name the areas in the grid so we can place the items in it easily and efficiently. Use the **grid-template-areas** property to name the cells in the grid:

```
  #container {
    display: grid;
    grid-template-rows: auto auto 5em;
    grid-template-columns: minmax(25em, 1fr) 16em;
    grid-template-areas:
      "banner banner"
      "main   hours"
      "footer footer";
  }
```

5. With everything set up, it'll be a breeze to put the content items into their proper places. Create a style rule for each grid item and tell it where to go with **grid-area**:

```
  header {
    grid-area: banner;
  }
  main {
    grid-area: main;
  }
  aside {
    grid-area: hours;
  }
  footer {
    grid-area: footer;
  }
```

Pretty easy, right? Now would be a good time to save the file and take a look at it in the browser (if you haven't already). The 2.5% margins that we had set on the **main** element earlier give it some nice breathing room in its area, so let's leave that alone. However, I'd like to remove the right margin and border radius we had set on the **aside** so it fills the right column. I'm going to just comment them out so that information is still around if I need to use it later:

```
  aside {
    …
    /* border-top-right-radius: 25px; */
    /* margin: 1em 2.5% 0 10%; */
  }
```

That does it! Open the *bakery.html* page in a browser that supports CSS grids, and it should look like the screenshot in FIGURE 16-47.

Now the bakery page has a nice two-column layout using a simple grid. In the real world, this would be just one layout in a set that would address different screen sizes as part of a responsive design strategy. We'll be talking about responsive design in the next chapter. And because grids are not supported by Internet Explorer, Edge, and older browsers, you would also create fallback layouts using Flexbox or floats depending on how universally you need the layout to work. I don't mean to kill your buzz, but I do want you to be aware that although this exercise let you sharpen your skills, it's part of a much broader production picture.

Note: For float- and position-based layout techniques that could be used as fallbacks, get the article **"Page Layout with Floats and Positioning"** (PDF) at learningwebdesign.com/articles/.