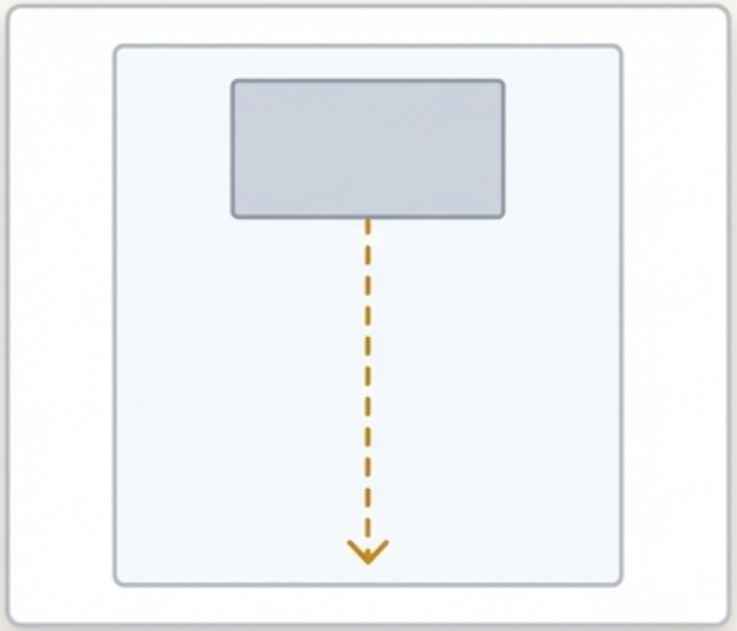
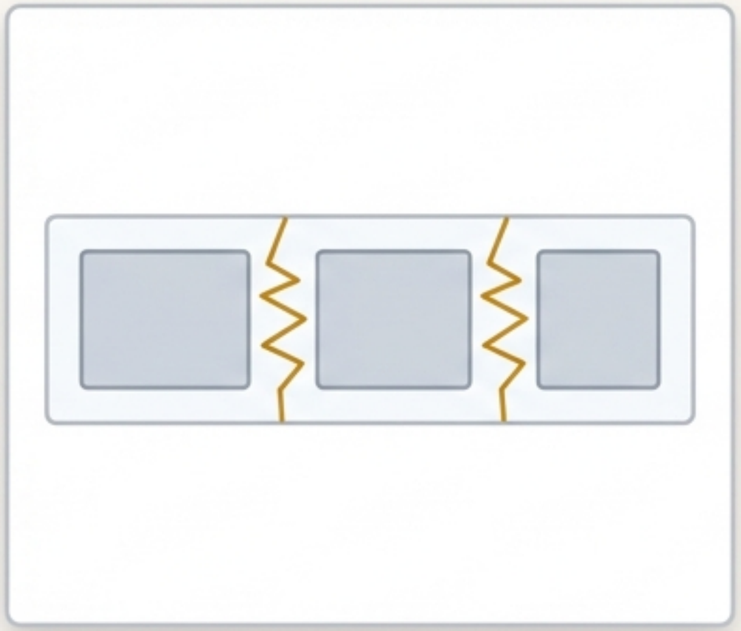
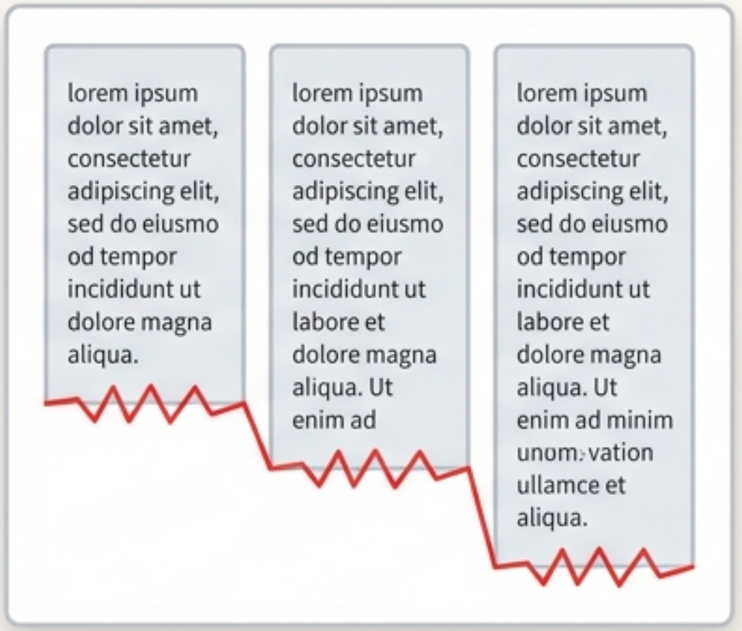
The Old Layout Challenges We All Faced



Vertically centering content.



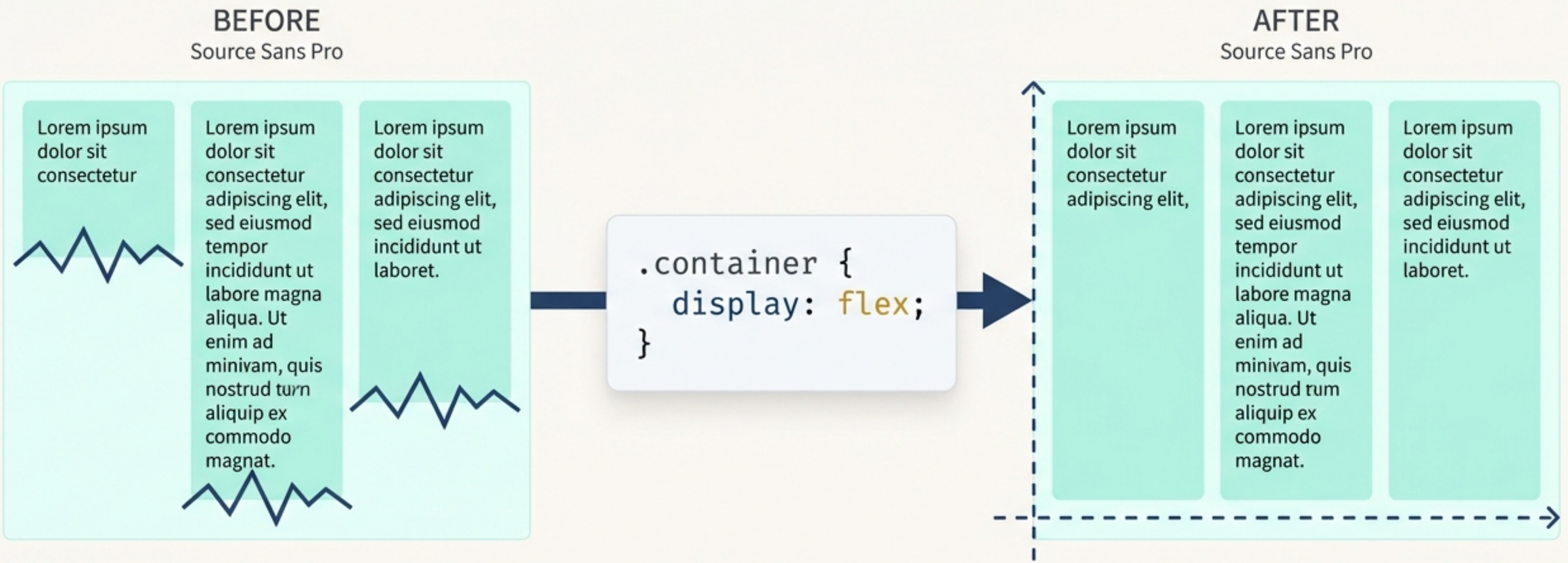
Making items share space equally.



Ensuring equal-height columns.

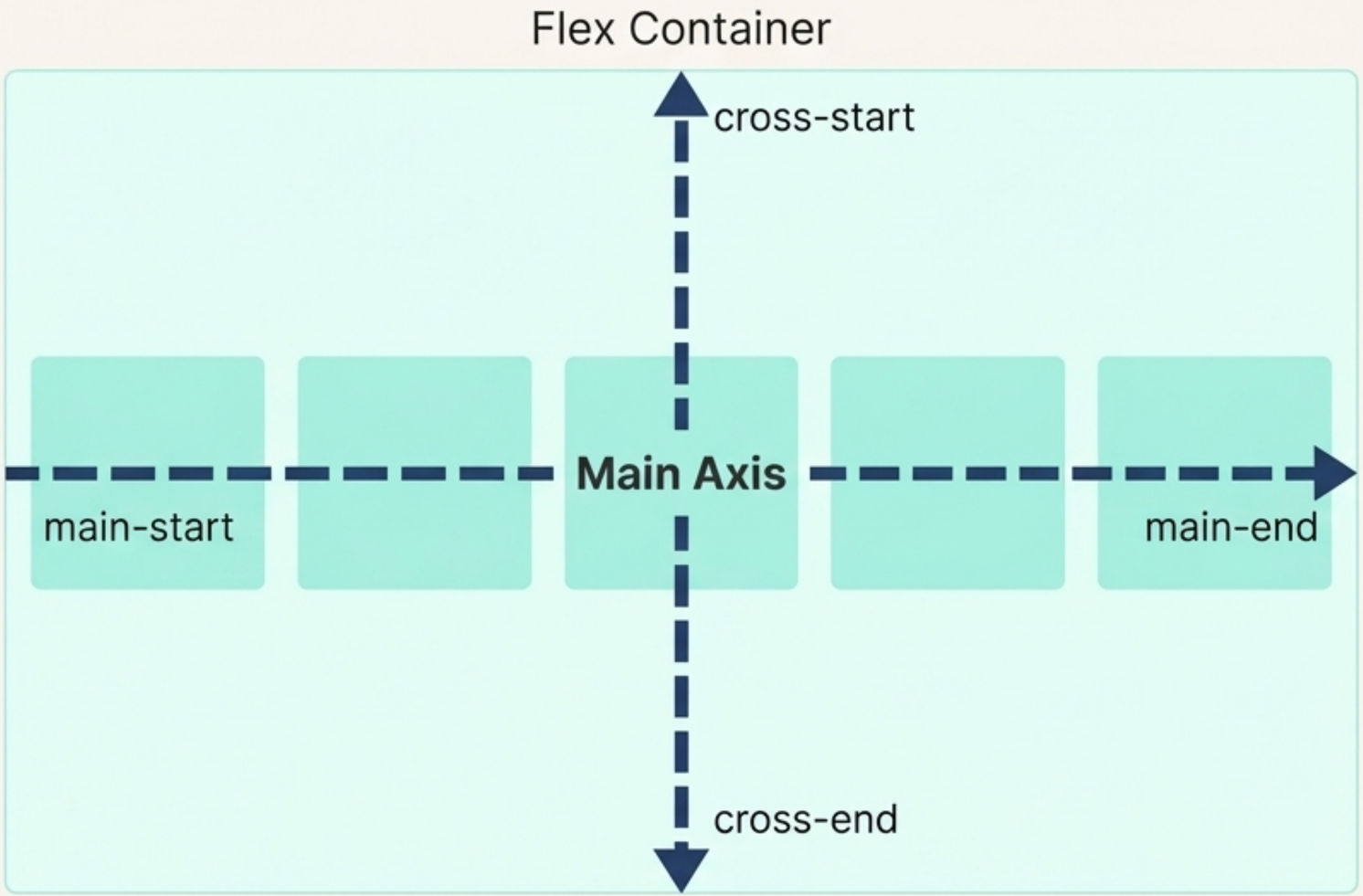
For years, achieving these seemingly simple layouts required complex hacks with floats, positioning, and table displays. They were often brittle and frustrating.

One Line of Code Changes Everything



By declaring display: flex on a parent container, you unlock a powerful new layout context for its children, solving these classic problems by default.

Understanding the Core Model: The Two Axes



Flex Container

The parent element (`display: flex`).

Flex Items

The direct children of the container.

Main Axis

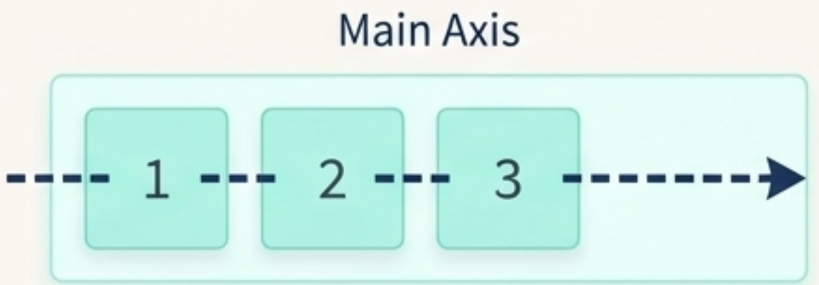
The primary axis items are laid out along. It's horizontal by default, but not always.

Cross Axis

The axis perpendicular to the main axis.

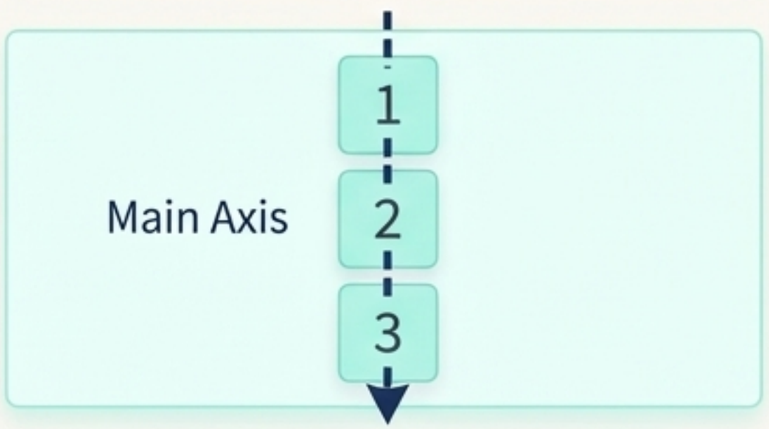
Controlling the Flow: `flex-direction`

This property establishes the main axis, defining the direction flex items are placed in the container.

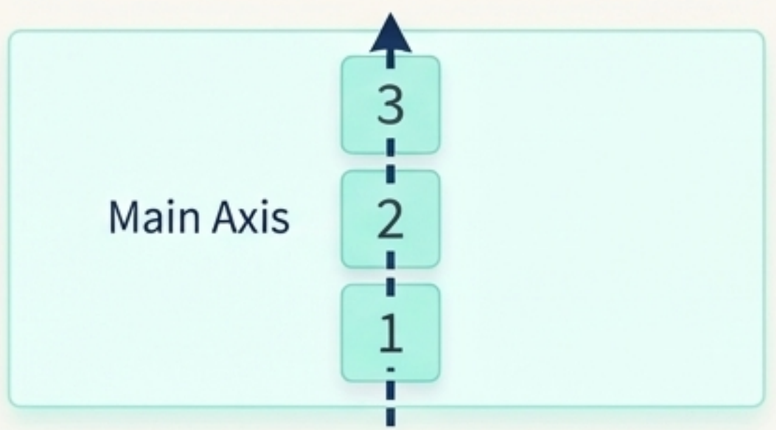


`flex-direction: row; (default)`

`flex-direction: row-reverse;`



`flex-direction: column;`

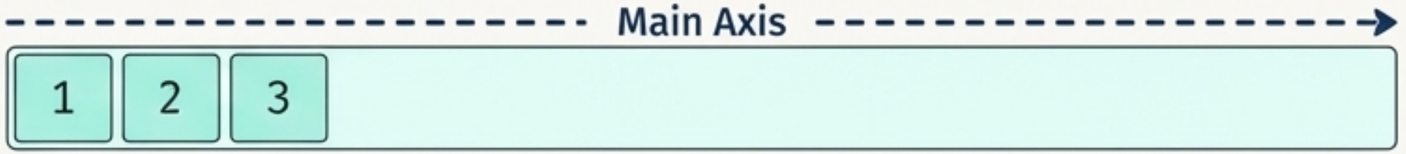


`flex-direction: column-reverse;`

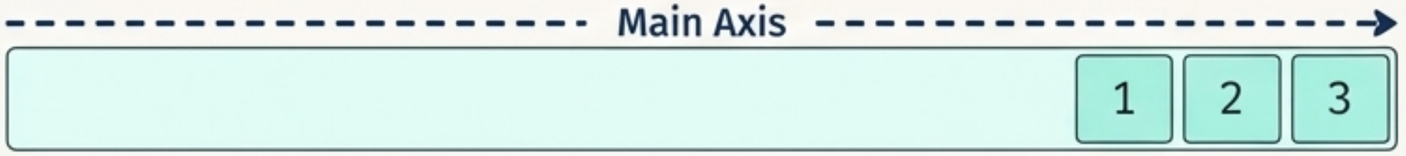
Distributing Space Along the Main Axis: `justify-content`

Defines alignment along the main axis and distributes extra space.

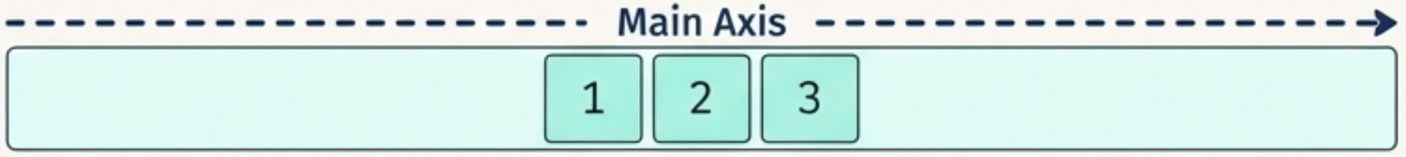
```
justify-content: flex-start; (default)
```



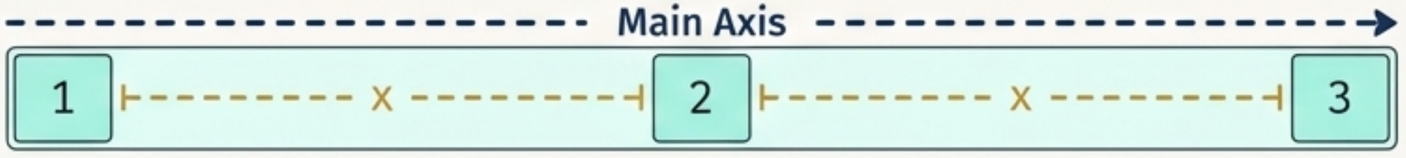

```
justify-content: flex-end;
```



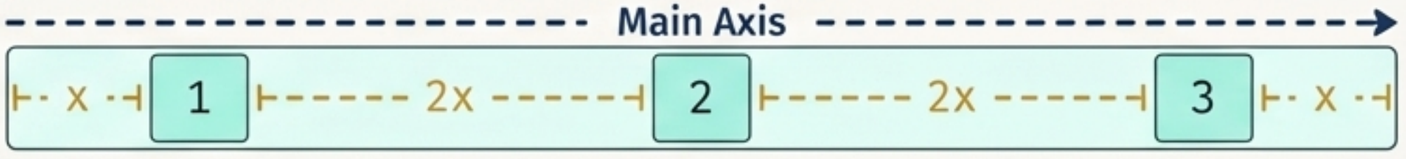

```
justify-content: center;
```



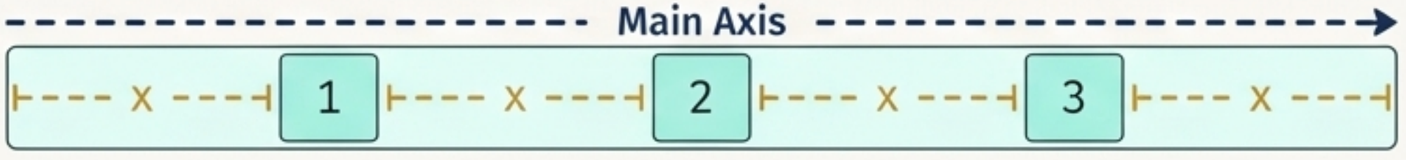

```
justify-content: space-between;
```




```
justify-content: space-around;
```




```
justify-content: space-evenly;
```



Aligning Items on the Cross Axis: `align-items`

Defines how flex items are laid out along the cross axis on the current line.

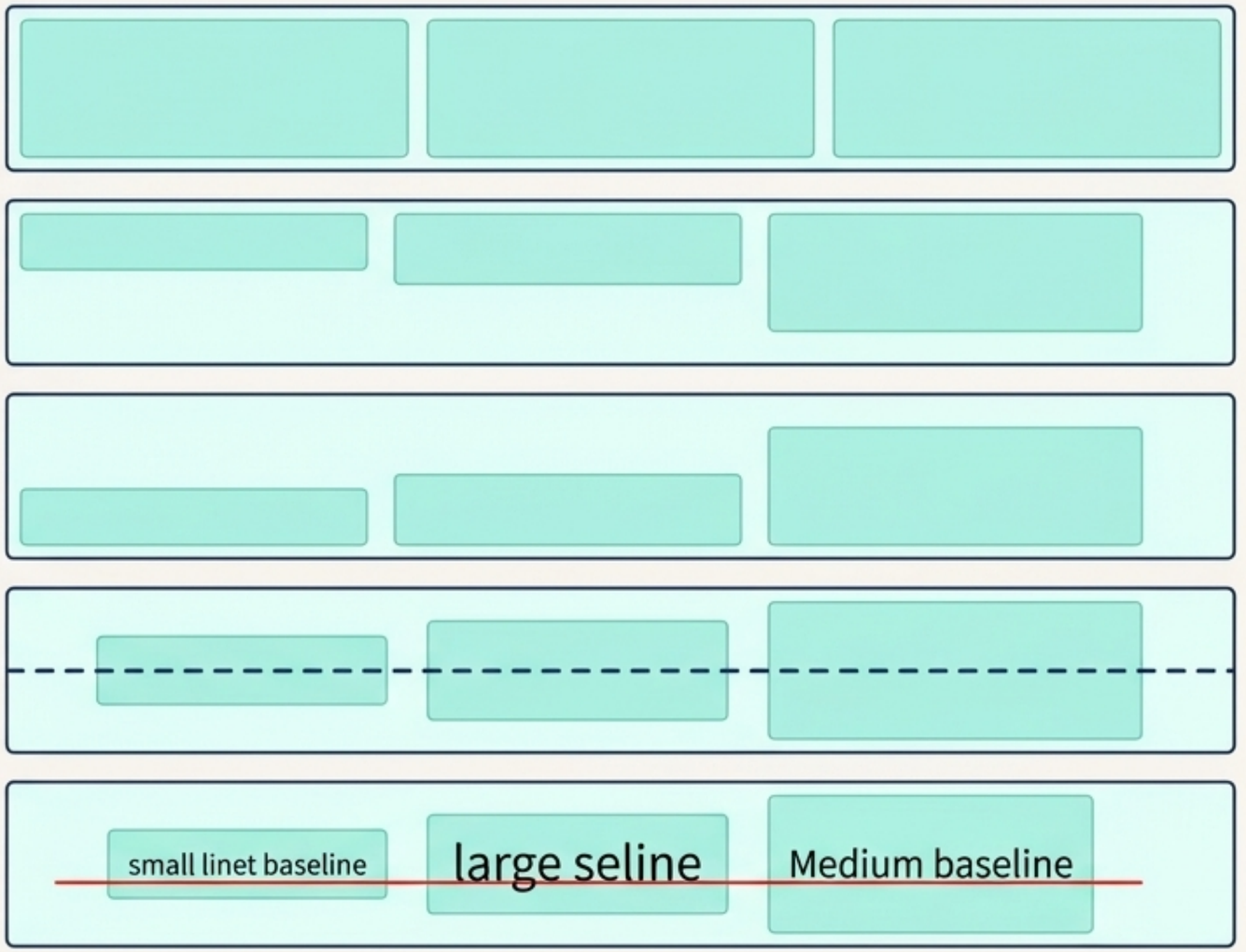
```
align-items: stretch; (default)
```

```
align-items: flex-start;
```

```
align-items: flex-end;
```

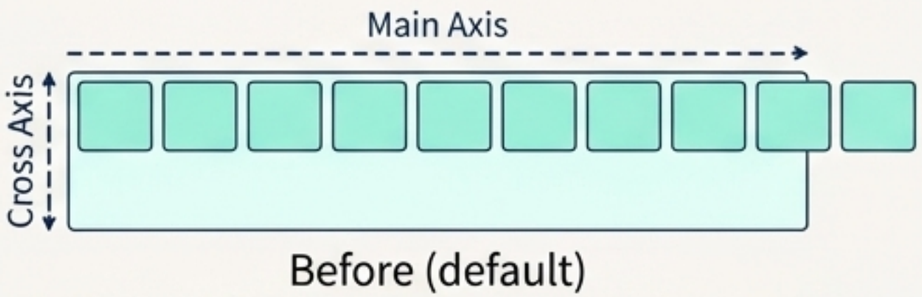
```
align-items: center;
```

```
align-items: baseline;
```



Handling Overflow with `flex-wrap` and `align-content`

Allowing Items to Wrap

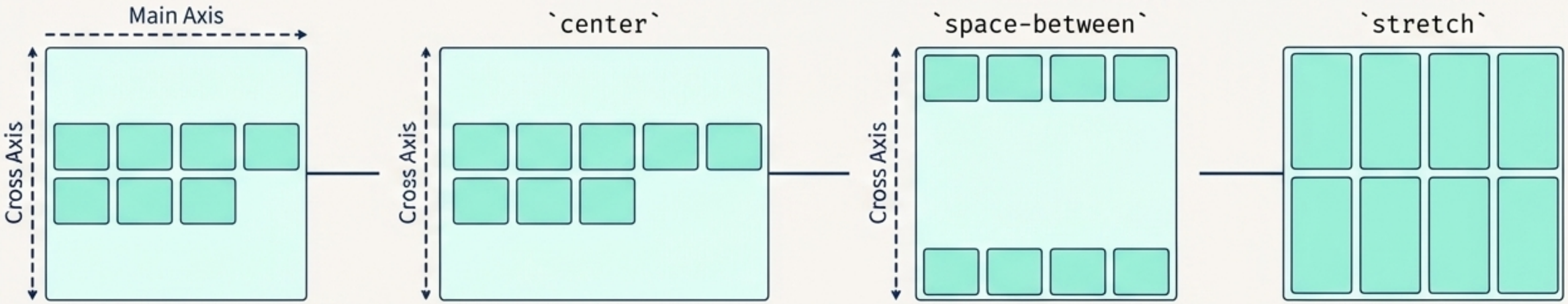


Before (default)

After

```
flex-wrap: wrap;
```

Aligning the Wrapped Lines



Important: `align-content` only takes effect on multi-line containers (when `flex-wrap` is `wrap` or `wrap-reverse`).

Controlling Individual Items

Change the visual order without touching the HTML.

Source Order

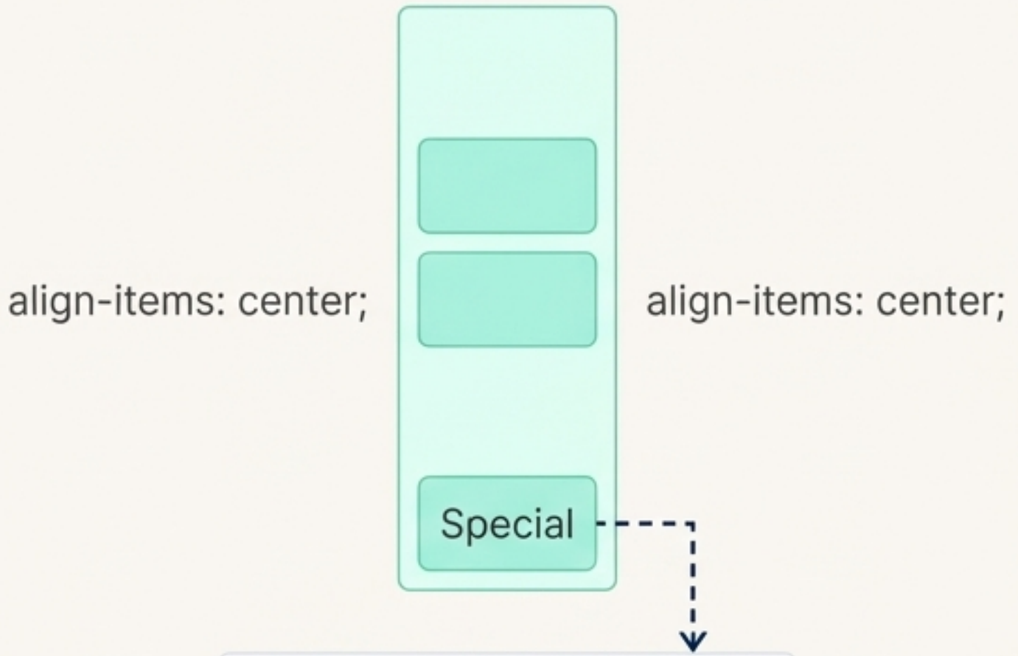
After

```
.item-1 { order: 2; } /* Default is 0 */
```

```
.item-2 { order: 3; }
```

```
.item-3 { order: 1; }
```

Override the container's `align-items` for a single item.




```
.special-item {
  align-self: flex-end;
}
```

The Essence of Flexibility: `flex-grow`, `flex-shrink`, `flex-basis`

Conceptual Definitions

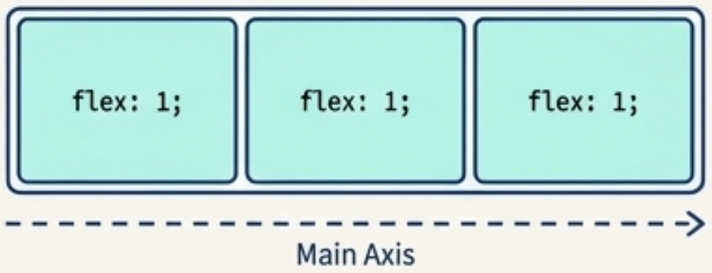
- `flex-grow`: A proportion dictating how much of the *remaining* space an item should take up.
- `flex-shrink`: A proportion dictating how much an item should shrink when space is scarce.
- `flex-basis`: The ideal starting size of an item before space is distributed.

Best Practice: Use the `flex` shorthand property. It intelligently sets all three values.

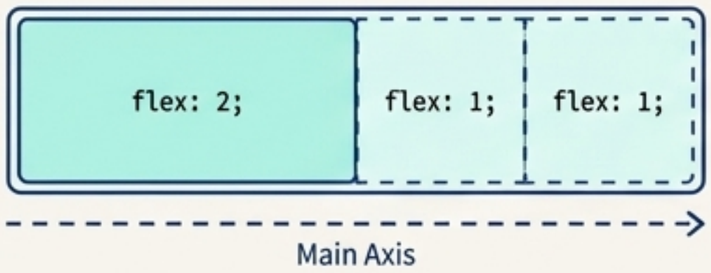
```
flex: <flex-grow> <flex-shrink> <flex-basis>;
```

Common Patterns with Visuals

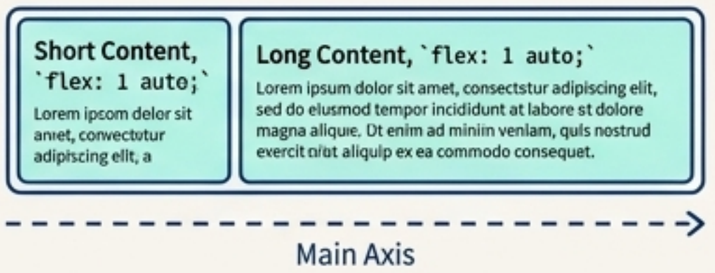
Pattern 1: `flex: 1;`
(shorthand for `1 1 0%`): All items grow and shrink equally from a basis of 0. Perfect for equal distribution.



Pattern 2: `flex: 2;` **vs** `flex: 1;`
Show one item taking up twice as much available space as its siblings.

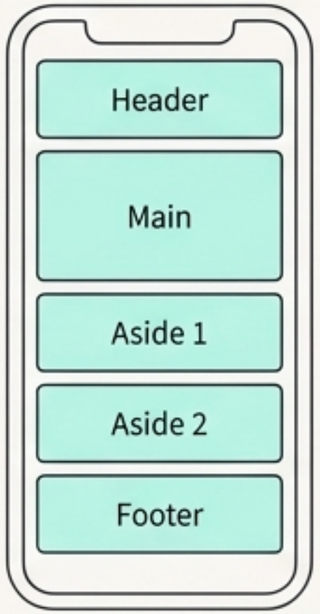


Pattern 3: `flex: 1 auto;`
An item can grow/shrink but its initial size is based on its width/height property.



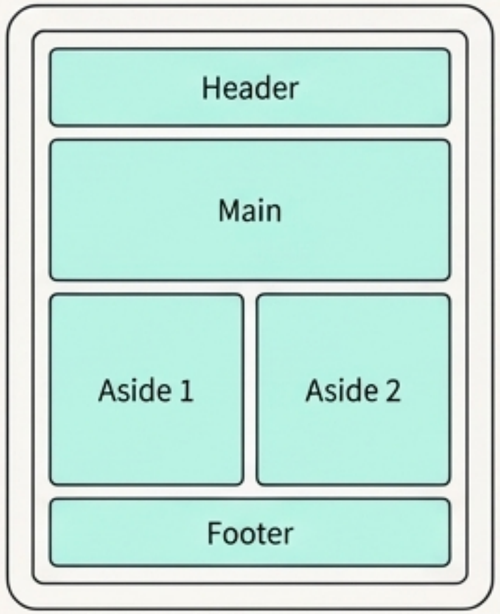
Putting It All Together: A Mobile-First Responsive Layout

Small Screen (<500px)



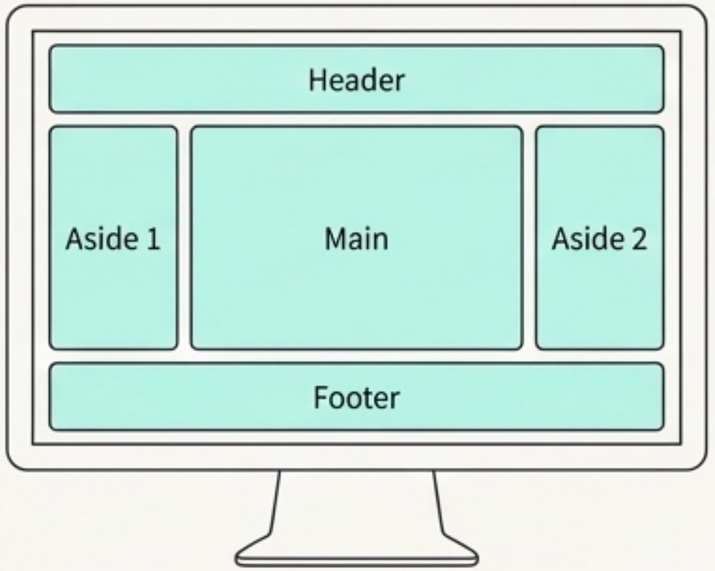

```
/* All items take full width */
.wrapper > * {
  flex: 1 1 100%;
}
```

Medium Screen (>600px)



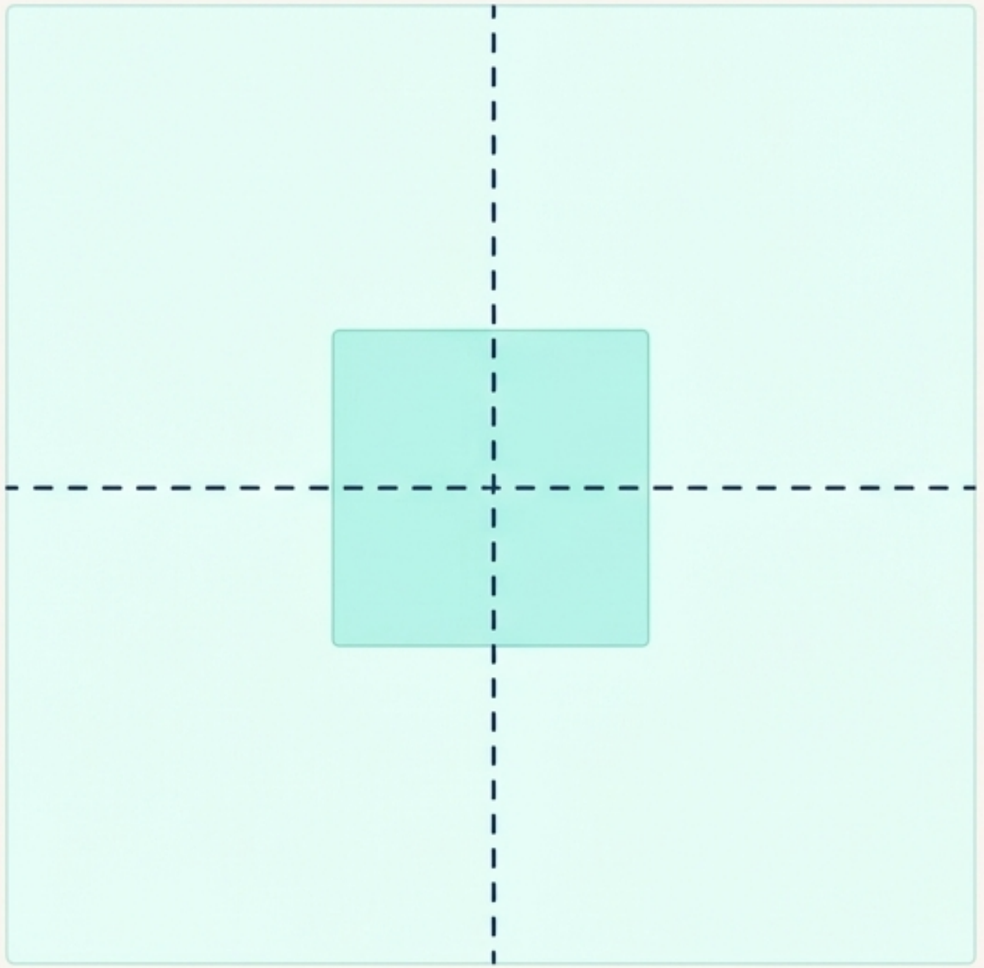

```
@media (min-width: 600px) {
  .aside { flex: 1 auto; }
}
```

Large Screen (>800px)




```
@media (min-width: 800px) {
  .main {
    flex: 3 0px;
    order: 2;
  }
  .aside-1 {
    flex: 1 auto;
    order: 1;
  }
  .aside-2 {
```

The Famous “Perfect Centering” Trick



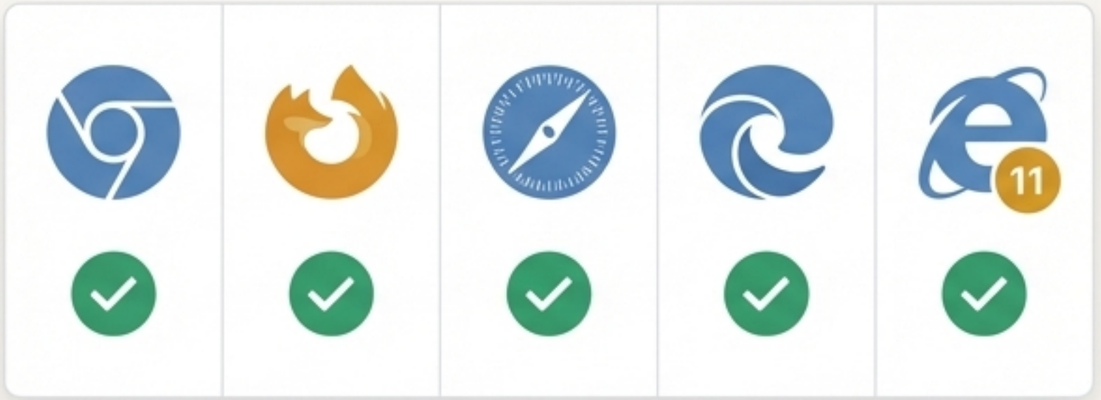
```css
.parent {
  display: flex;
  /* Add height to see vertical centering */
  height: 300px;
}

.child {
  /* The magic! */
  margin: auto;
}
```

When an item in a flex container has `margin: auto`, it automatically absorbs all available free space on the axes it's applied to. This makes it the simplest and most robust way to center an element.

Professional Practice: Browser Support & Tooling

Browser Support

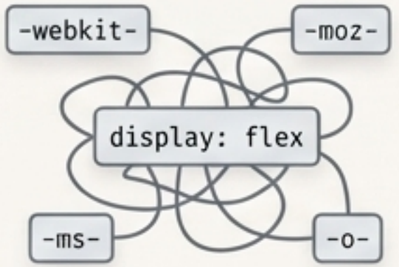


Modern Flexbox is fully supported in all major browsers. Internet Explorer 11 has partial support with some bugs and prefixes.

Vendor Prefixes

The Challenge

The Flexbox spec evolved over time, resulting in “old” (`-moz-box`), “tweener” (`-ms-flexbox`), and modern syntaxes. Handling this manually is error-prone.



The Solution

Don’t prefix by hand. **Use a post-processor like Autoprefixer.** Write clean, modern CSS, and let the tool automatically add the necessary prefixes and fallbacks during your build process.

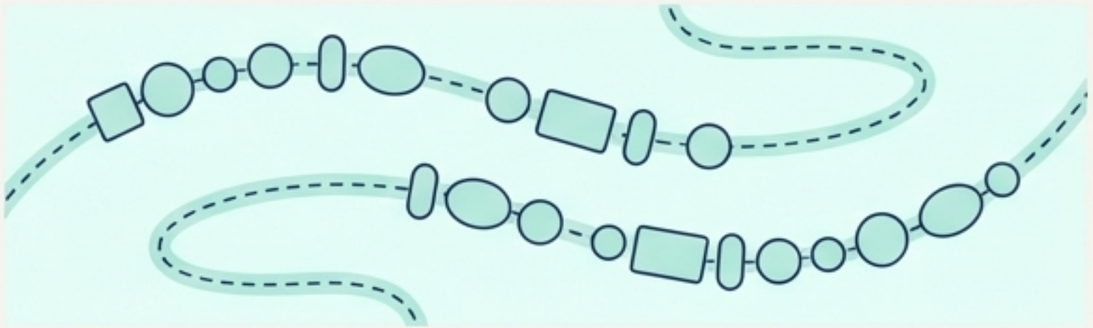
```
.container {
  display: flex;
}
```

→ Autoprefixer →

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

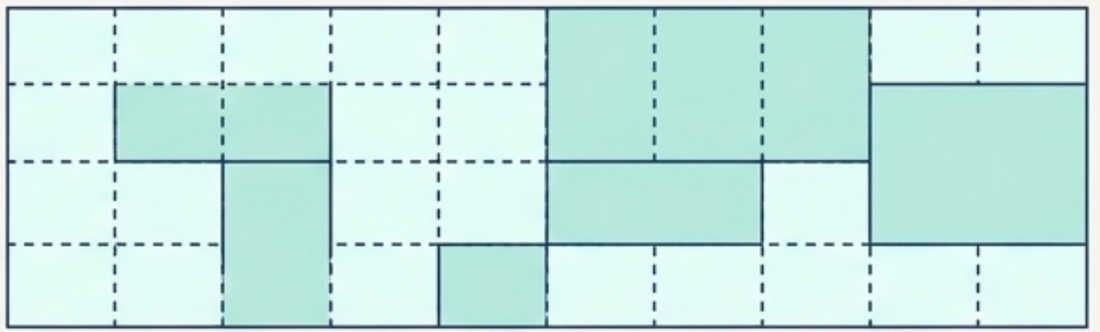
Flexbox for Components, Grid for Layouts

Flexbox



- A **one-dimensional** layout system for rows OR columns.
- Excels at distributing space and aligning items in a line.
- Perfect for UI components: navigation bars, form elements, card collections.

CSS Grid



When to Use CSS Grid

- For **two-dimensional** layouts managing both rows AND columns simultaneously.
- Ideal for overall page structure, creating a macroscopic grid that your flexbox components can live inside.

The Modern Approach: Don't think of it as Flexbox vs. Grid. Think of them as complementary tools. Use Grid for the page layout and Flexbox for the components within it.