responsive

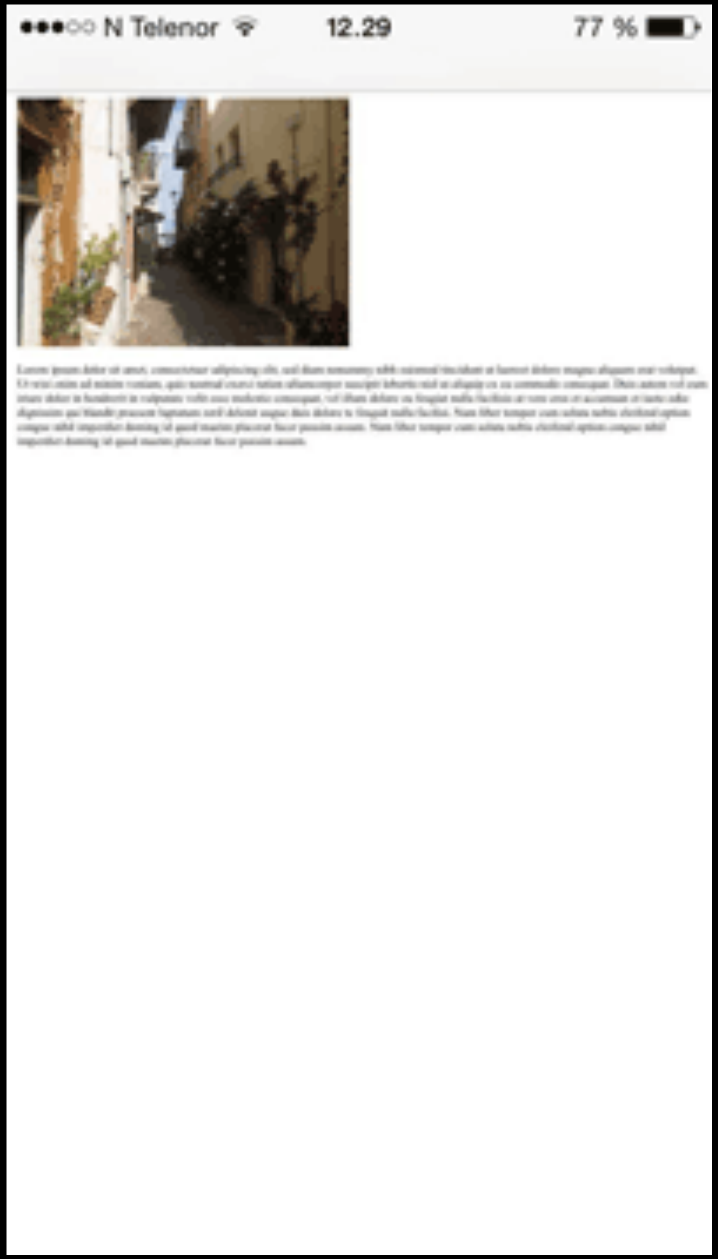
“A pixel is not a pixel”

— Peter Paul Koch

“If the pixel density of the output device is very different from that of a typical computer display, the user agent should rescale pixel values. It is recommended that the pixel unit refer to the whole number of device pixels that best approximates the reference pixel. It is recommended that the reference pixel be the visual angle of one pixel on a device with a pixel density of 96dpi and a distance from the reader of an arm's length.” — w3 consortium

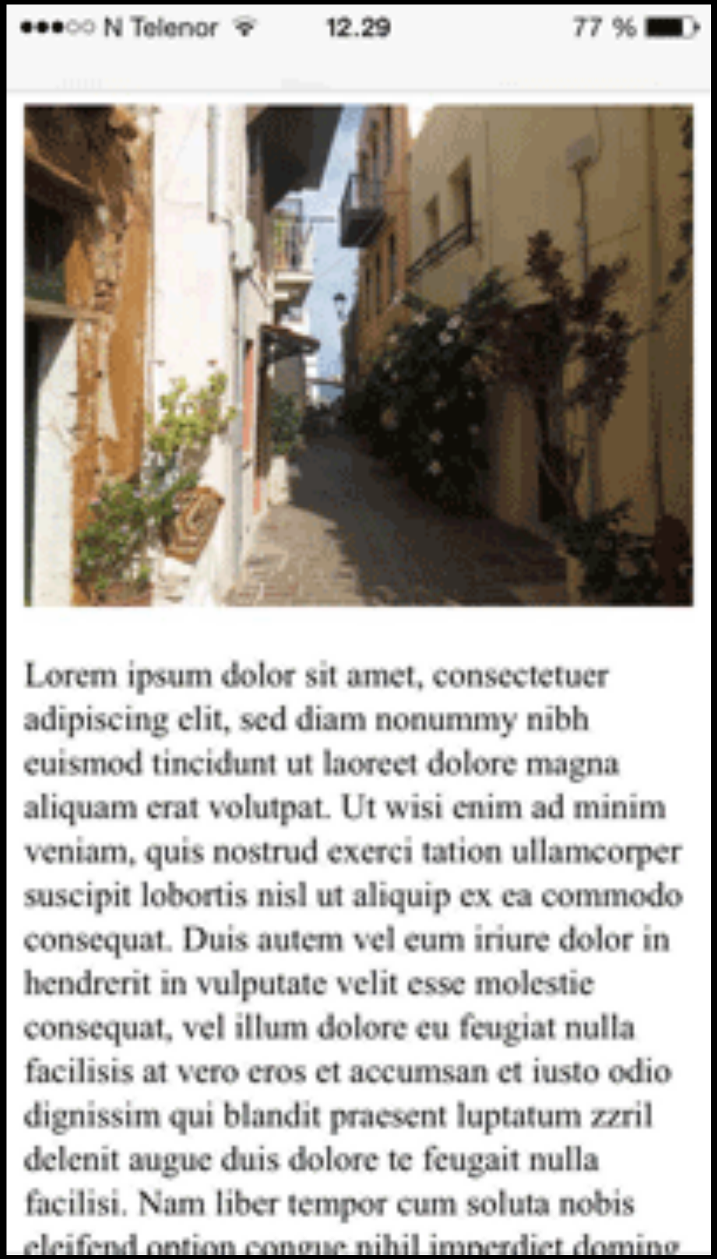
```
<meta name="viewport" content="width=device-width, initial-scale=1.0”>
 <!
 —  - Tells the browser to match the device's width for the viewport
    - Sets an initial zoom value -->
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0”>
```



[illegible]

without



Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming

with

Metadata: `viewport`

The user's visible area of a web page

HTML5 introduced a method to let web designers take control over the viewport, through the **<meta>** tag.

Let's breakdown the `content` value:

- **Values are comma separated, letting you specify a list of values for `content`**
- **The `width` value is set to `device-width`. This will cause the browser to render the page at the same width of the device's screen size.**
- **`initial-scale` set to `1` indicates the "zoom" value if your web page when it is first loaded. `1` means "no zoom."**

There are other values you can specify for the `content` list -

```
<meta name="viewport" content="width=device-width, initial-scale=1.0”>
```

Metadata: `viewport`

There are other values you can specify for the `content` attribute -

```
<meta name="viewport" content="width=device-width, initial-scale=1.0”>
```

500px

minimum-scale
maximum-scale
user-scalable

vw + vh

You can define height and width in terms of the viewport

- Use units **vh** and **vw** to set height and width to the percentage of the viewport's height and width, respectively

- 1vh = 1/100th of the viewport height

- 1vw = 1/100th of the viewport width

```
div {
  width:10vw;
  height: 10vh;
 }
```

responsive text

The text size can be set with a "vw" unit, which means the "viewport width".

That way the text size will follow the size of the browser window.

```
div {
 font-size:10vw;
 }
```

Media Queries

the @media rule tells the browser to include a block of CSS properties only if a certain condition is true.

So this:

```
@media only screen and (max-width: 500px)  {
    body {
      background-color: light blue;
    }
}
```

Translates to:

```
if (the maximum width of the web page is 500 pixels) {
            then do this stuff
}
```

Media Queries

Breakpoint

```
/* For mobile phones: */
[class*="col-"] {
    width: 100%;
}
@media only screen and (min-width: 768px) {
    /* For desktop: */
    .col-1 {width: 8.33%;}
    .col-2 {width: 16.66%;}
    .col-3 {width: 25%;}
    .col-4 {width: 33.33%;}
    .col-5 {width: 41.66%;}
    .col-6 {width: 50%;}
    .col-7 {width: 58.33%;}
    .col-8 {width: 66.66%;}
    .col-9 {width: 75%;}
    .col-10 {width: 83.33%;}
    .col-11 {width: 91.66%;}
    .col-12 {width: 100%;
}
```

add a **breakpoint** where certain parts of the design will behave differently on each side of the breakpoint

many examples: https://www.w3schools.com/Css/css_rwd_mediaqueries.asp

Mobile-first! (Images)

Dimanche

27 NOVEMBRE

THEATRE DU VIDE

UN HOMME DANS L'ESPACE !

```
/* For width smaller than 400px: */
body {
    background-image: url(‘void_newspaper.jpg’);
}

/* For width 400px and larger: */
@media only screen and (min-width: 400px) {
    body {
        background-image: url('void.jpg');
    }
}
```

Responsive layout patterns

The manner in which a site transitions from a small-screen layout to a wide-screen layout must make sense for that particular site, but there are a few patterns (common and repeated approaches) that have emerged over the years. We can thank Luke Wroblewski (known for his “Mobile First” approach to web design, which has become the standard) for doing a survey of how responsive sites handle layout. Following are the top patterns Luke named in his article:

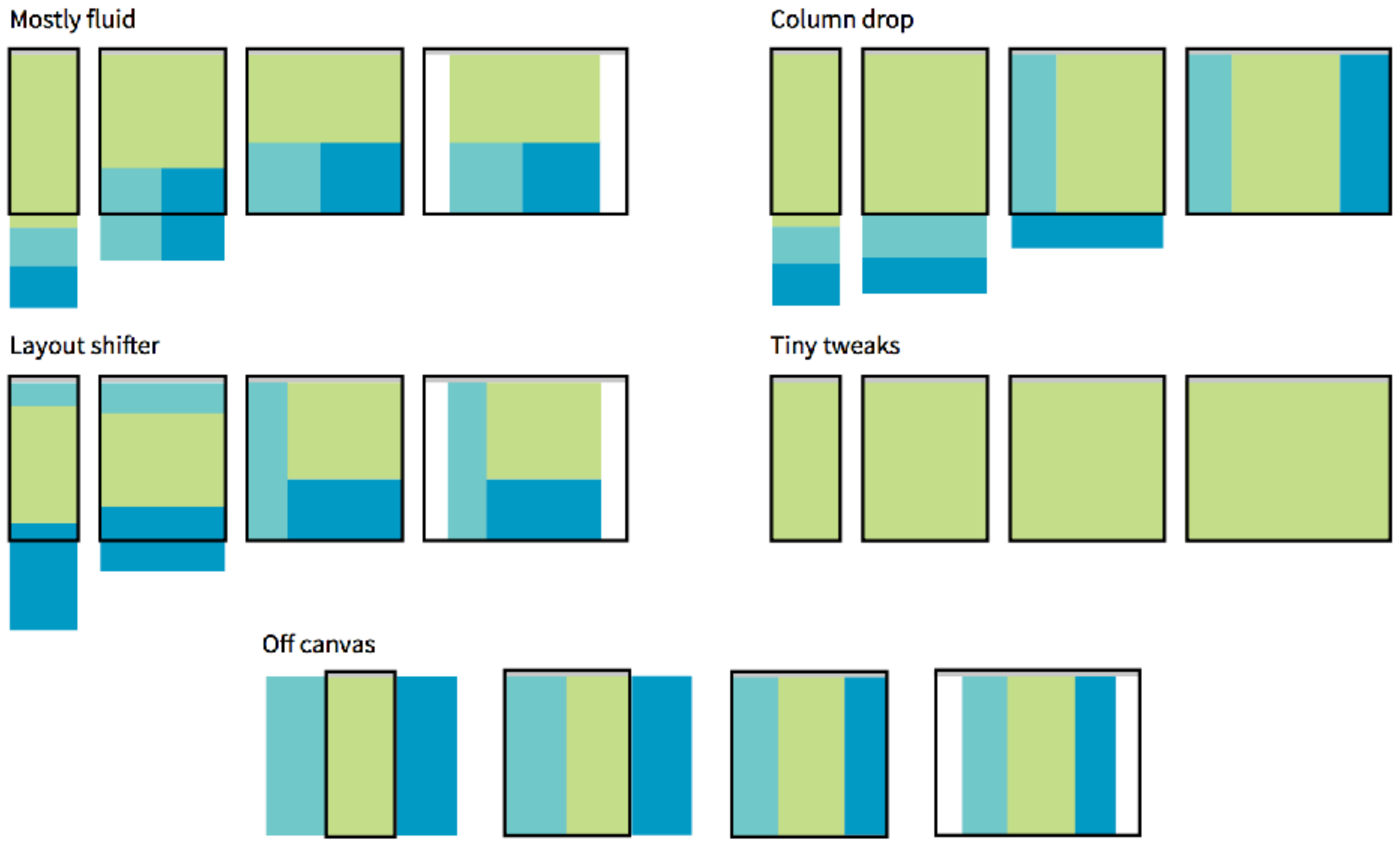


FIGURE 17-9. Examples of the responsive layout patterns identified by Luke Wroblewski.

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 verti- cally 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.

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.

Display Property

```
display: none;

display: inline;
display: block;

display: flex;
display: grid;
```

Overriding Default Display

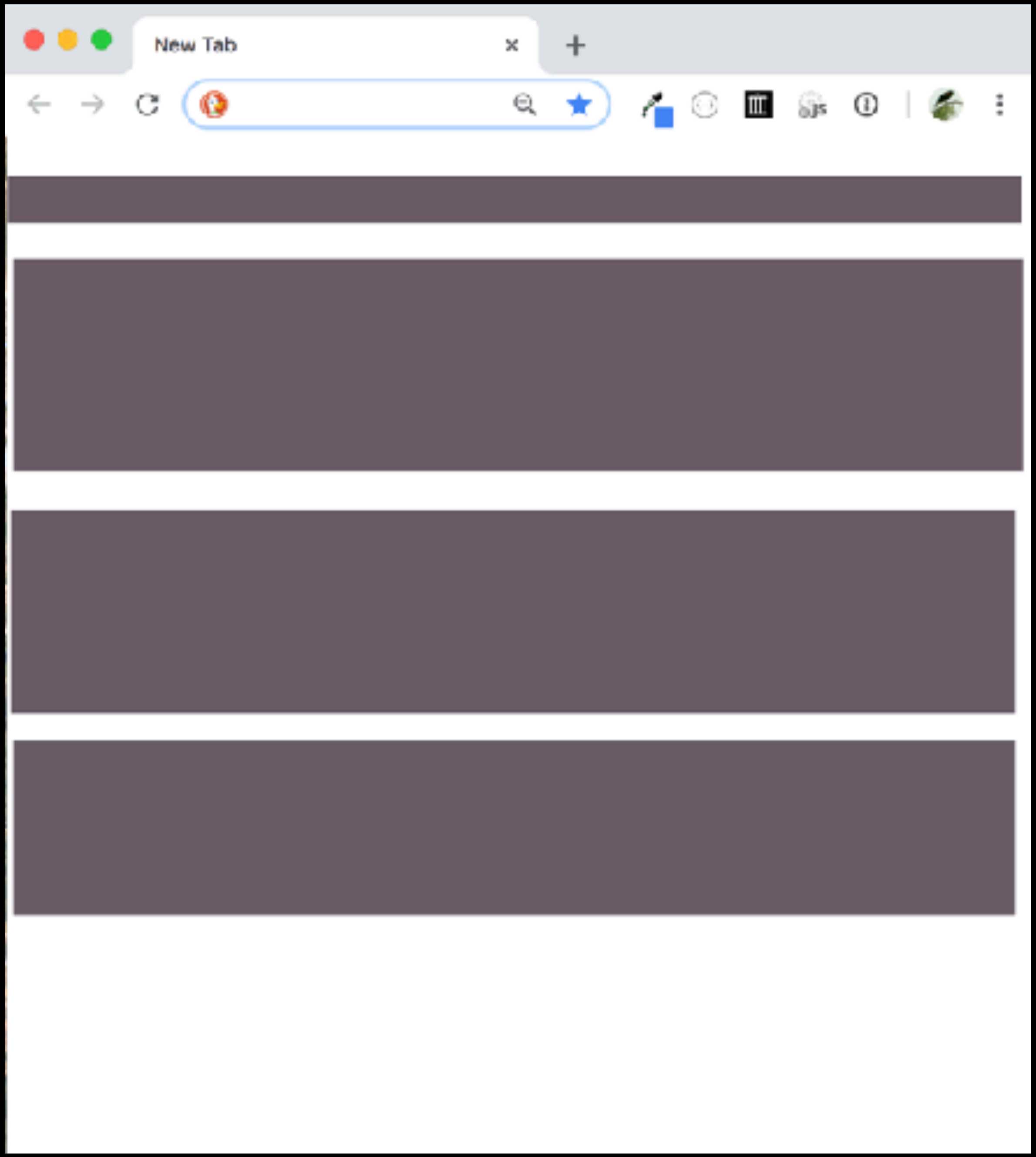
Changing an inline element to a block element, or vice versa, can be useful for making the page look a specific way, and still follow the web standards.

```css
li {
    display: inline;
}

span {
    display: block;
}
```

Note: Setting the display property of an element only changes how the element is displayed, NOT what kind of element it is. So, an inline element with display: block; is not allowed to have other block elements inside it.

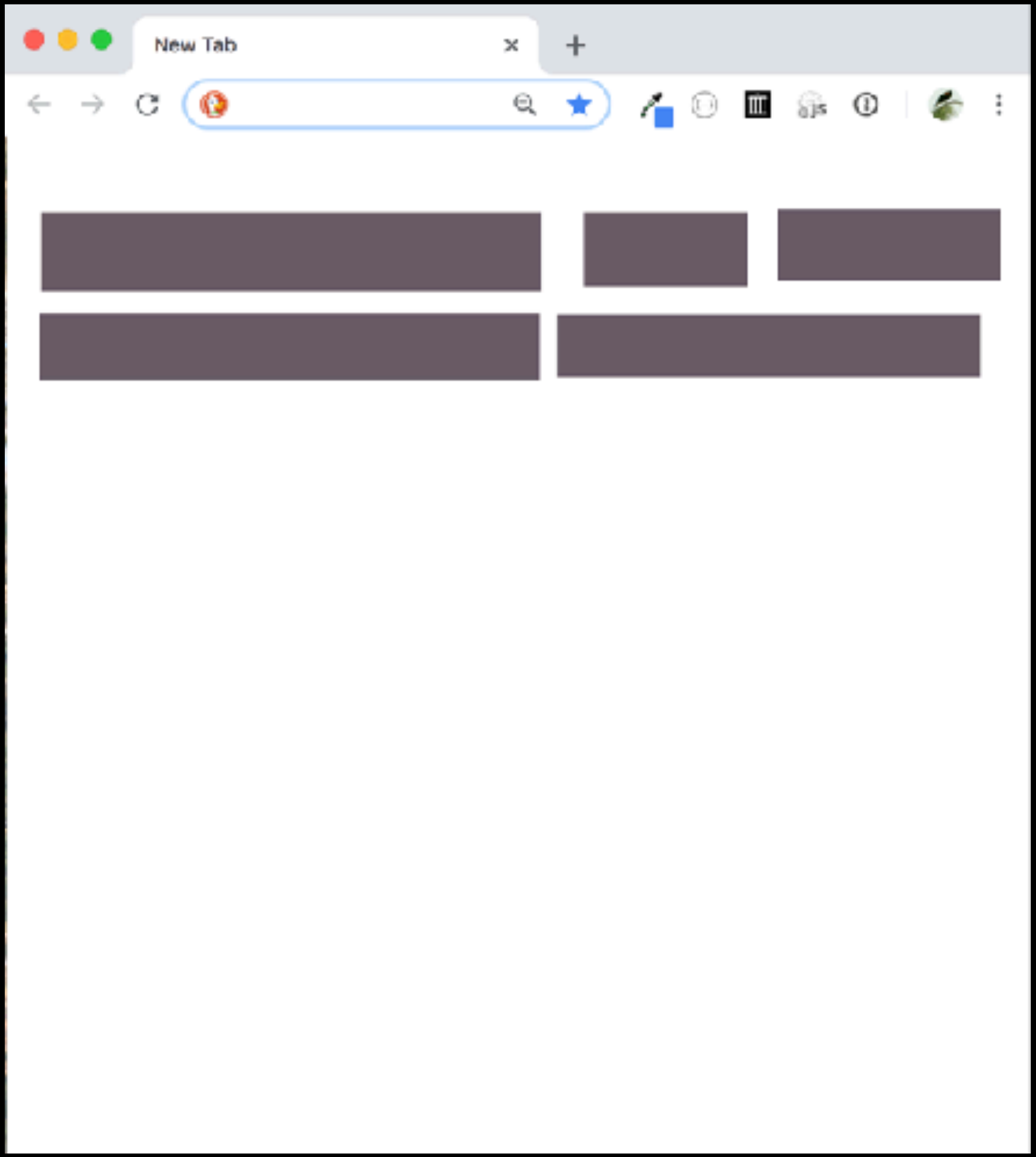
```
display: flex;
display: grid;
```

flex display



Block layout

Laying out large sections of a page



Inline layout

Laying out txt + other inline content within a section

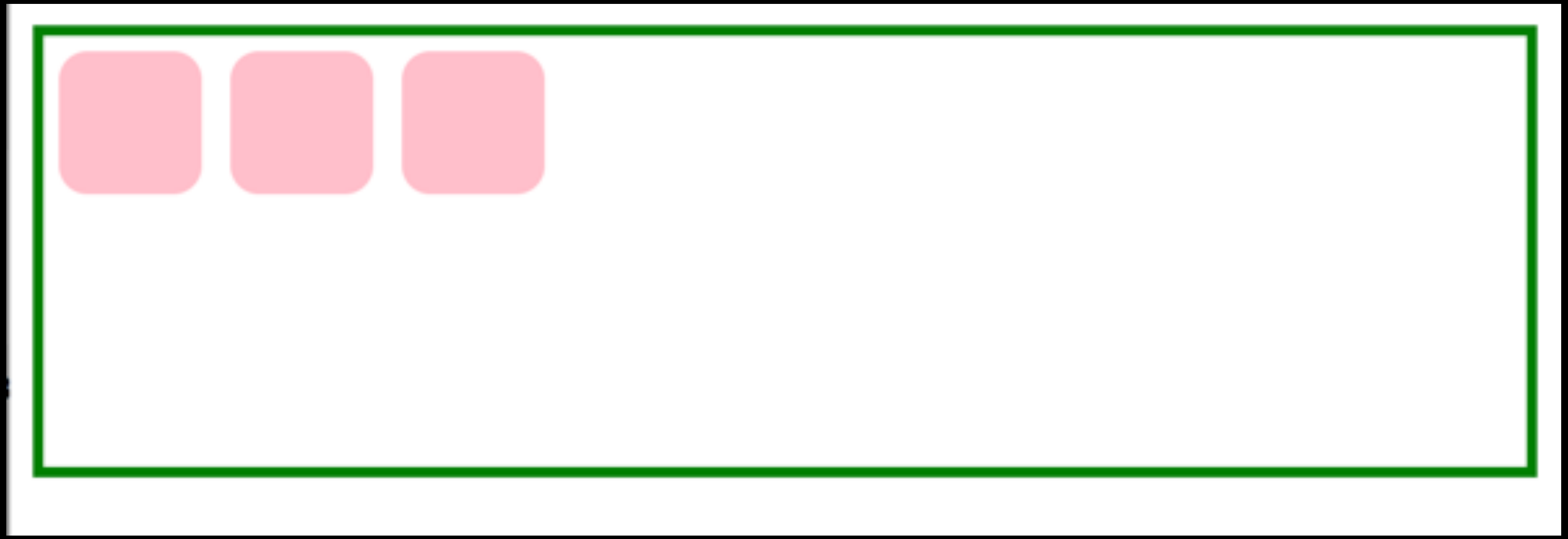
Flex - different rendering model

When you set a container to **display: flex**, the direct children in that container are **flex items** + follow a new set of rules.

Flex items are not block or inline; they have different rules for their height, width + layout.

- The **contents** of a flex item follow the usual block/inline rules, relative to the flex item's boundary.

Flex Basics



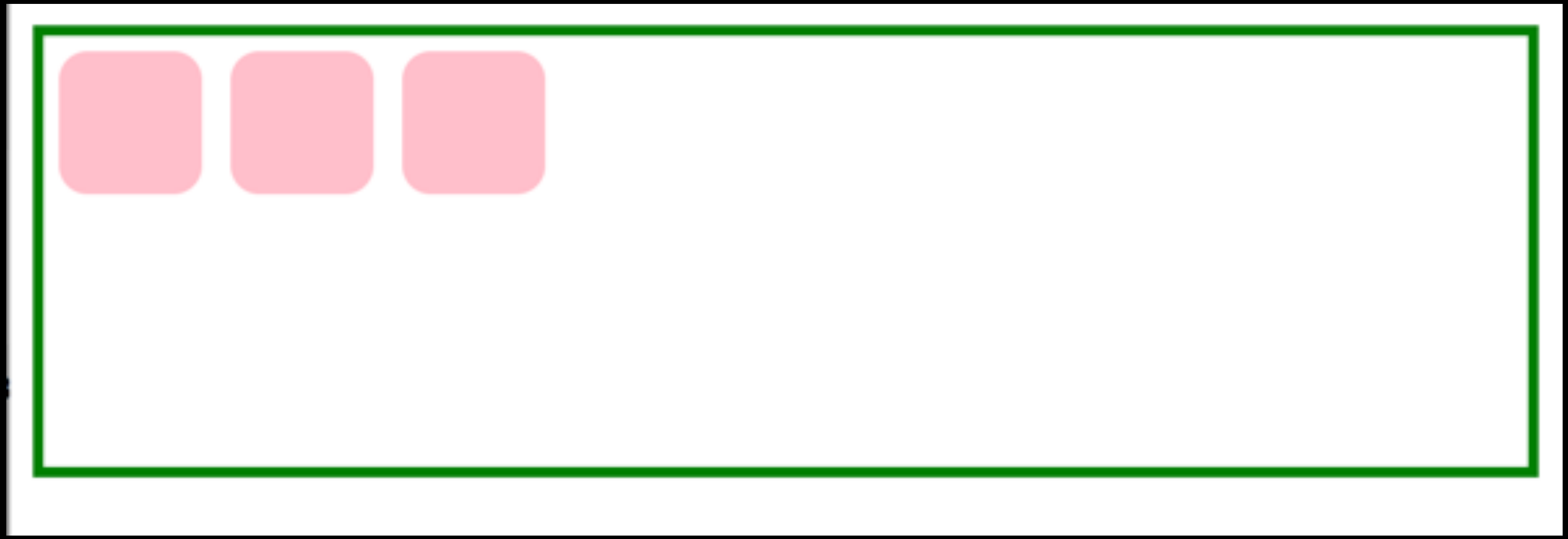
Flex layouts are composed of:

a **Flex container**, which contains one or more:

Flex item(s)

You can then apply CSS properties on the **Flex container** to dictate how the **Flex item(s)** are displayed

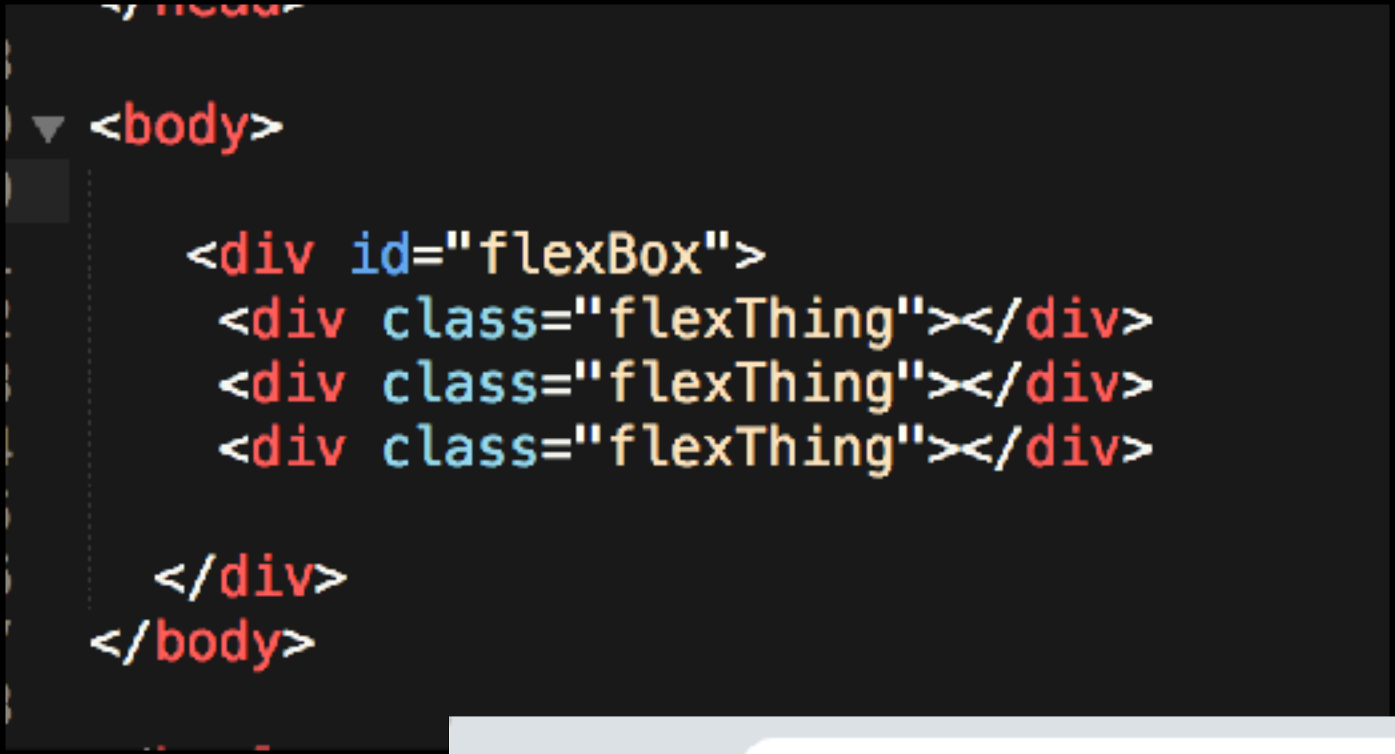
Flex Basics



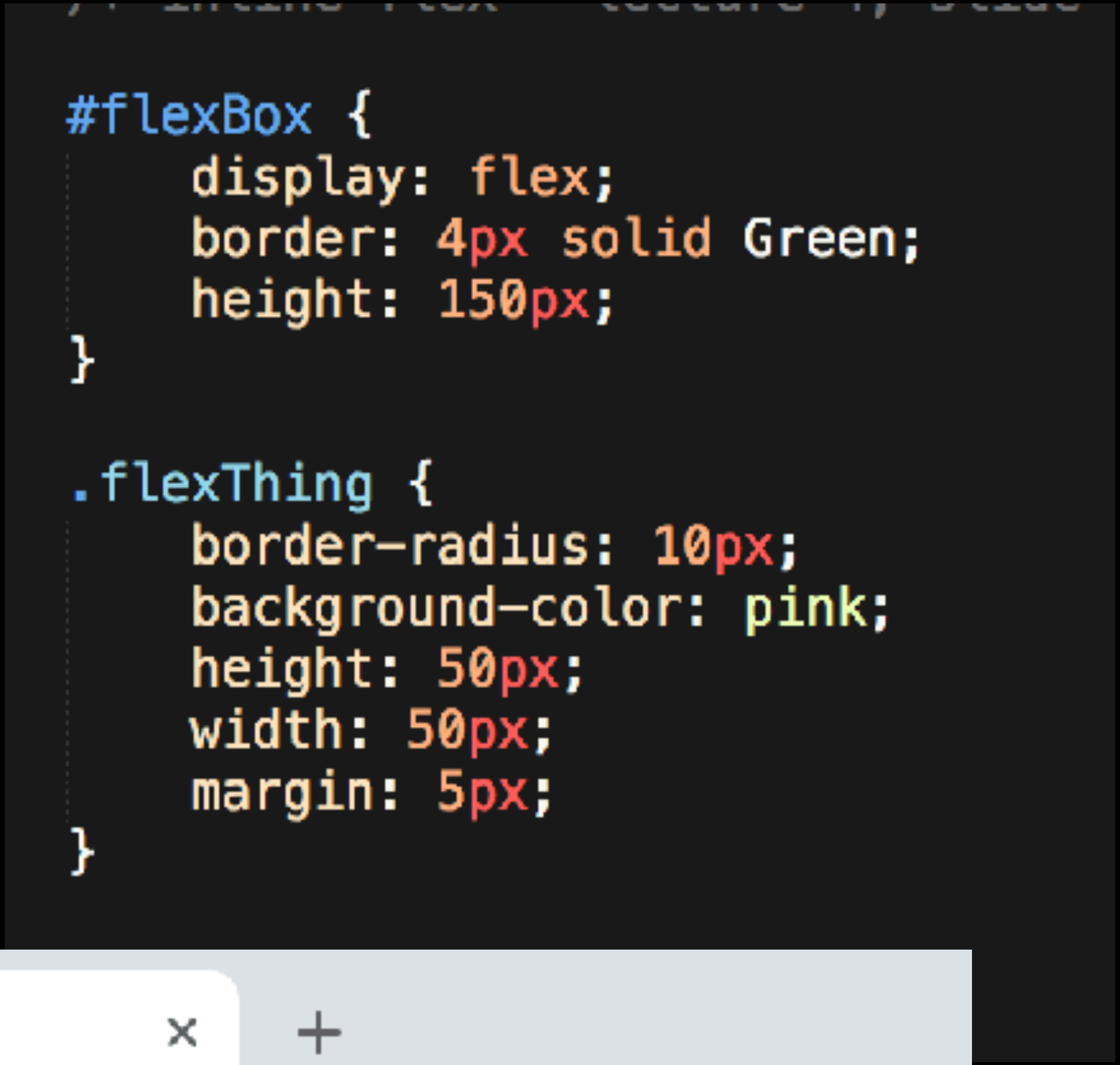
To make an element a flex container, change display:

- Block container: display: flex;
- Inline container: display: inline-flex;

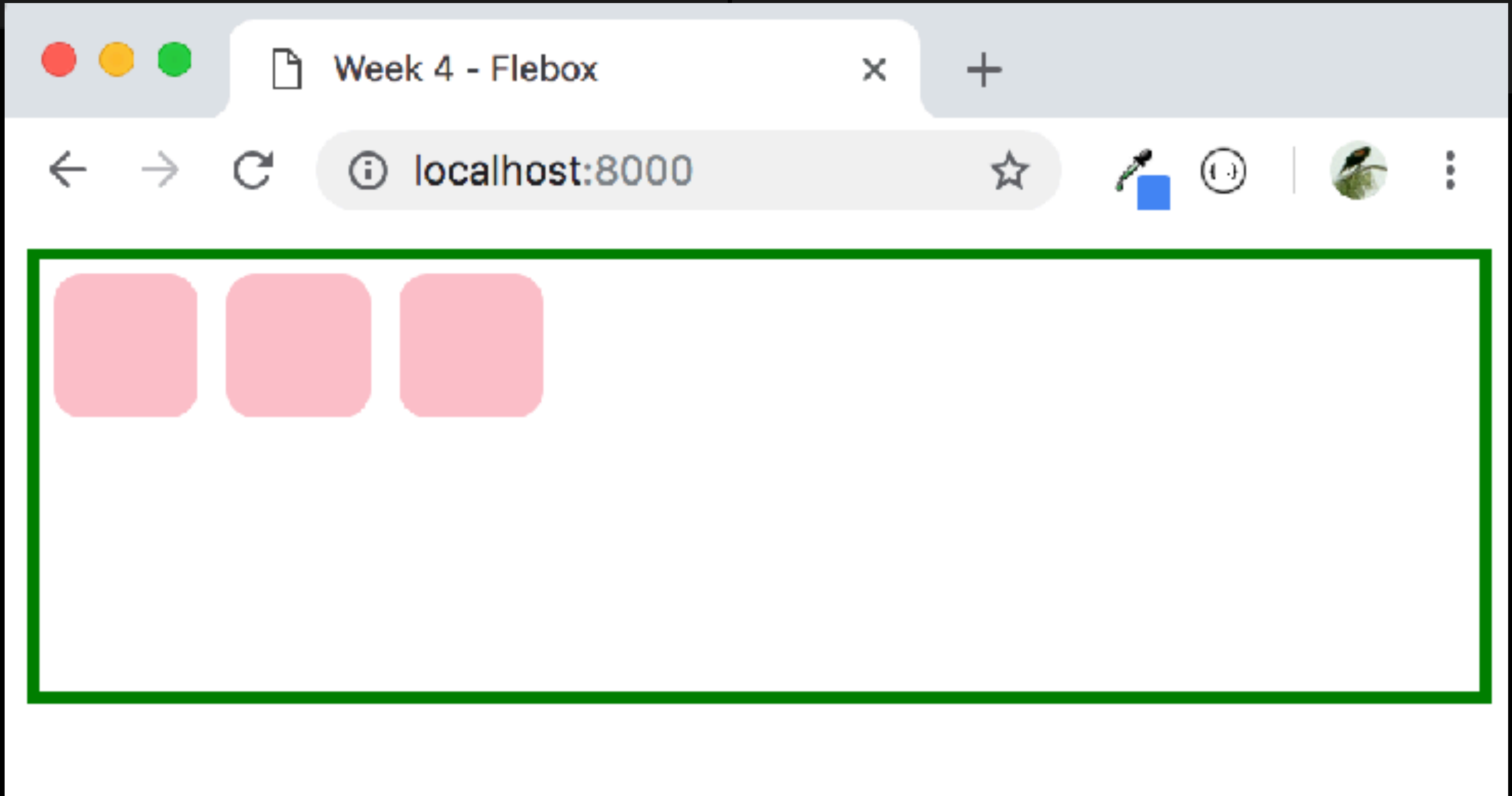
Flex Basics



```
<body>

    <div id="flexBox">
     <div class="flexThing"></div>
     <div class="flexThing"></div>
     <div class="flexThing"></div>

   </div>
</body>
```

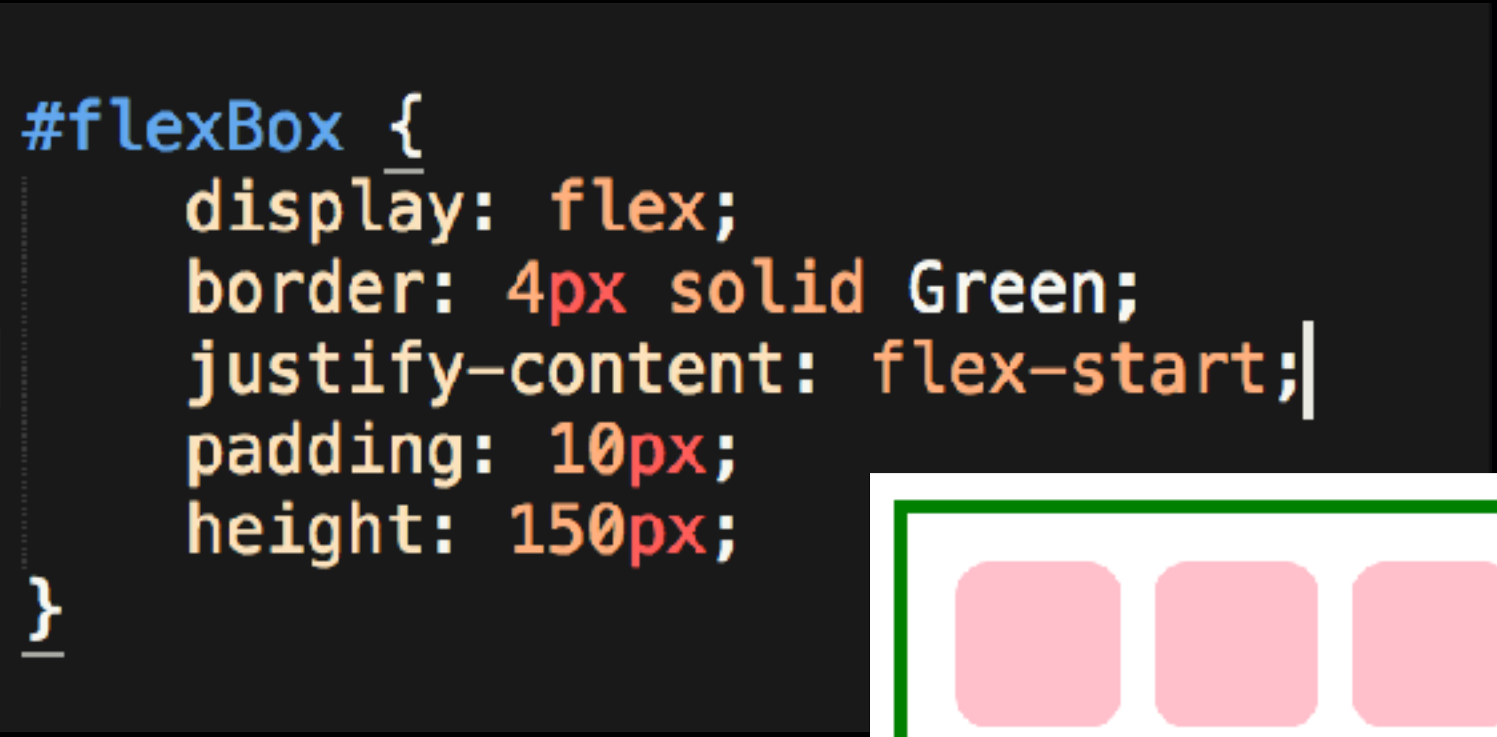


```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    height: 50px;
    width: 50px;
    margin: 5px;
}
```

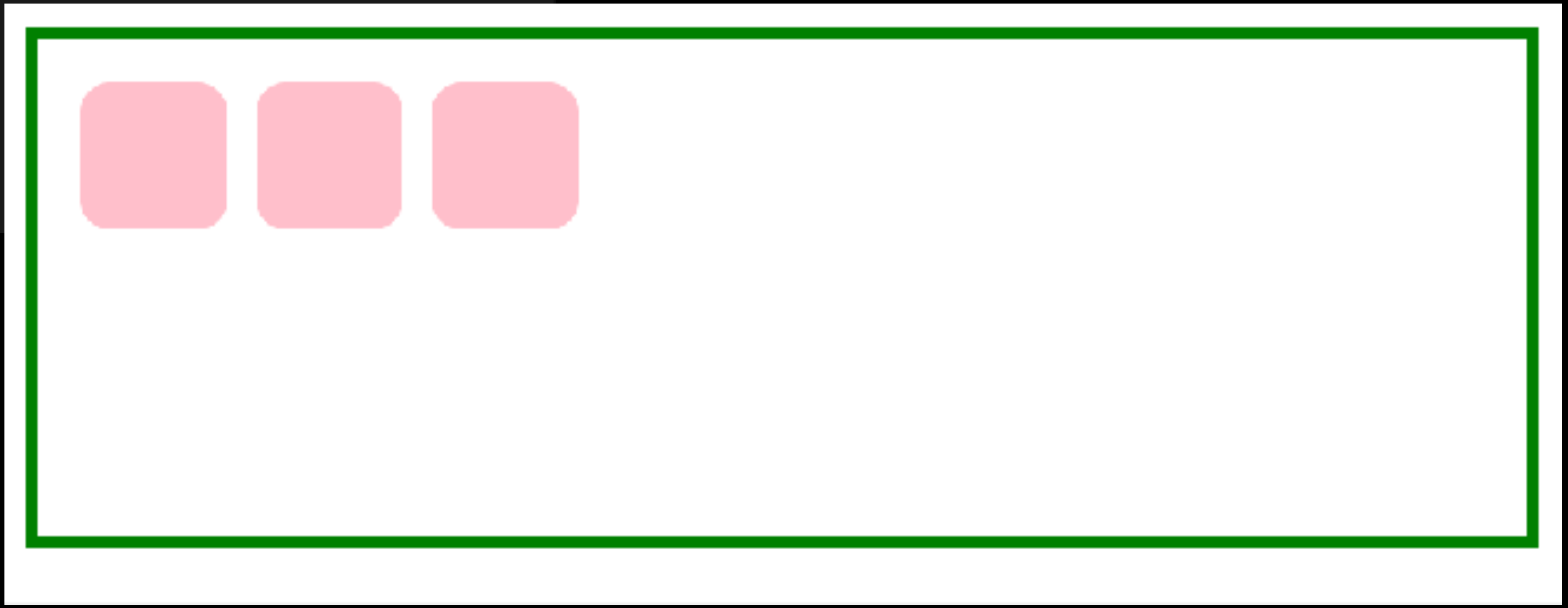


Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

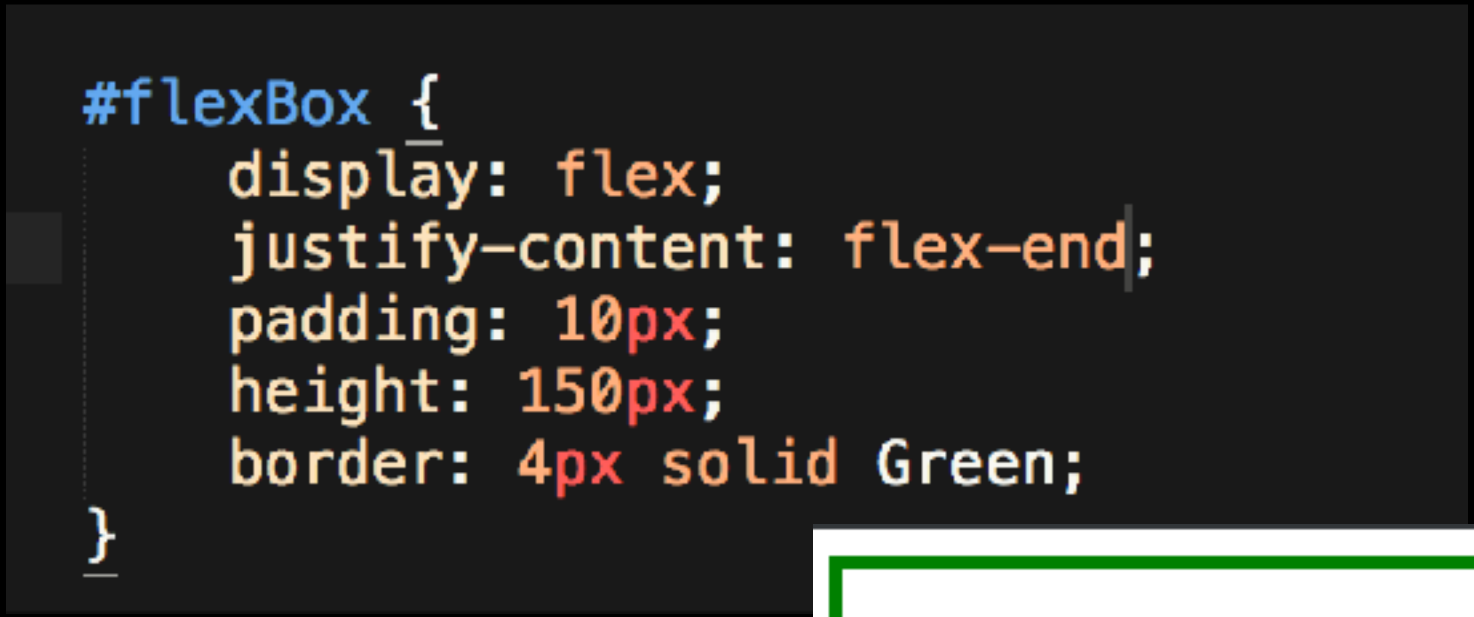


```
#flexBox {
    display: flex;
    border: 4px solid Green;
    justify-content: flex-start;
    padding: 10px;
    height: 150px;
}
```

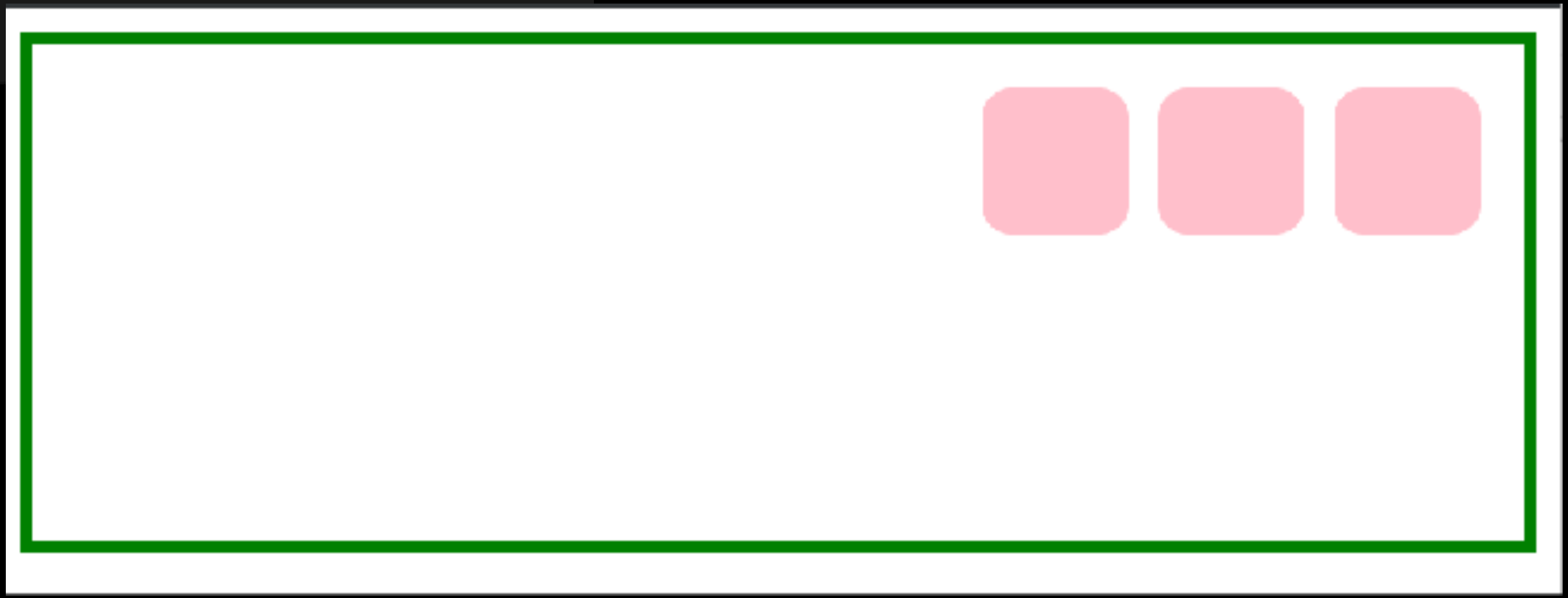


Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

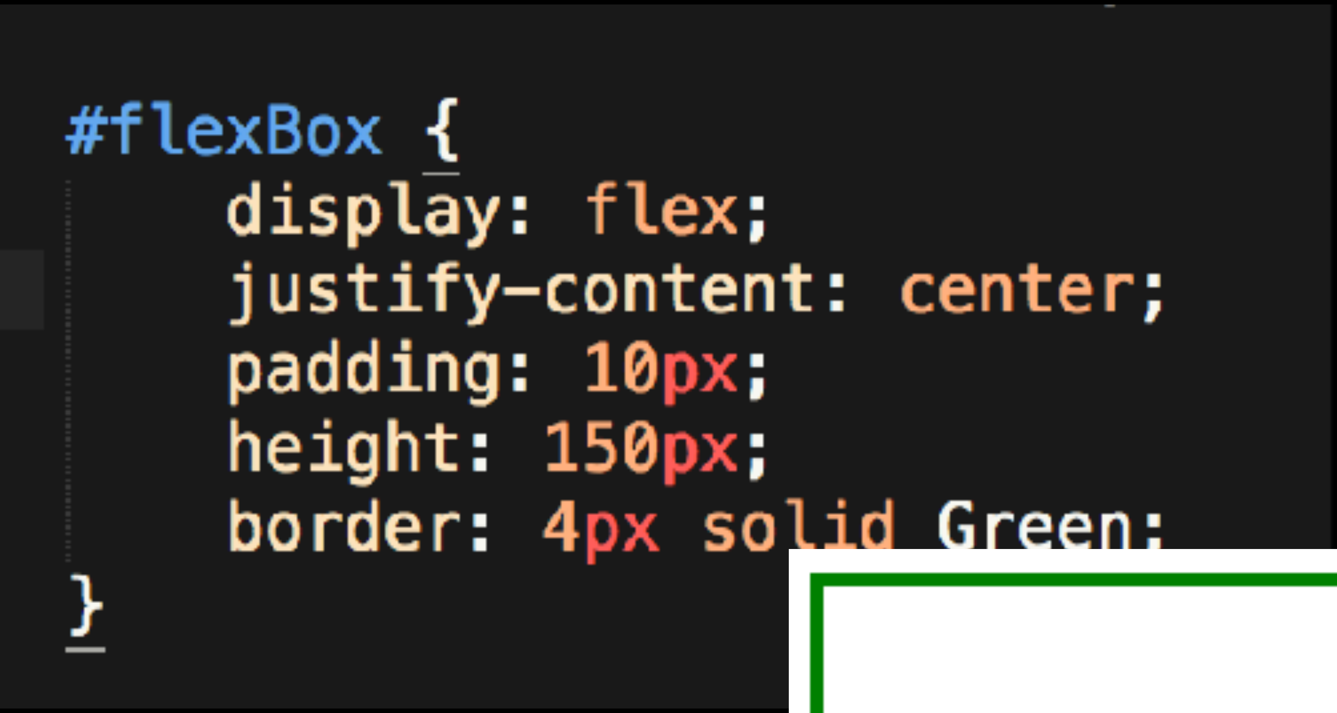


```css
#flexBox {
    display: flex;
    justify-content: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

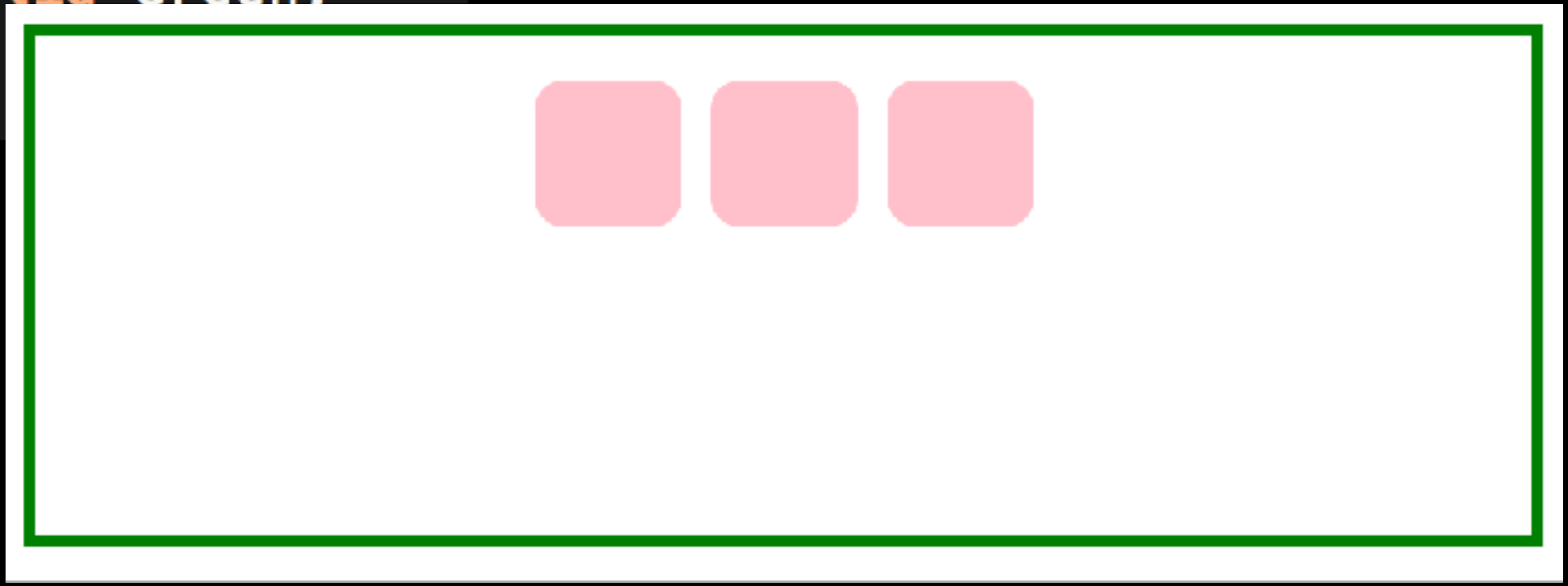


Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.

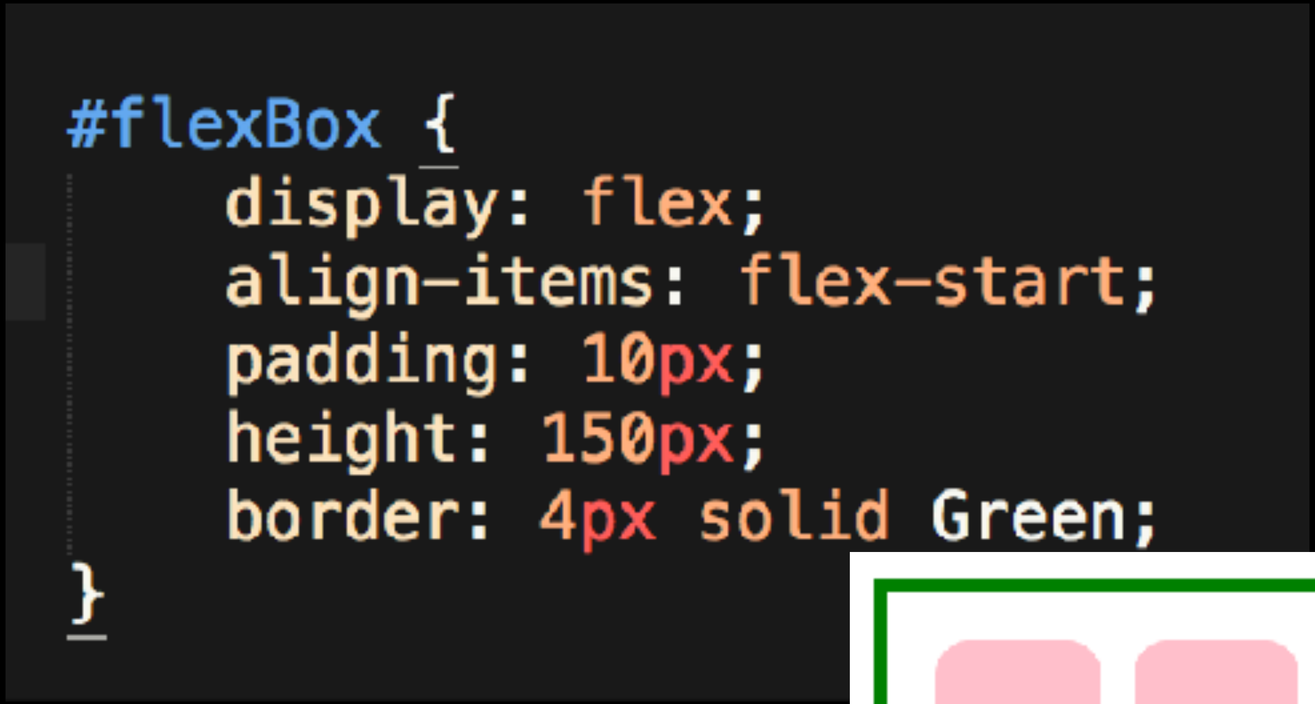


```
#flexBox {
    display: flex;
    justify-content: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

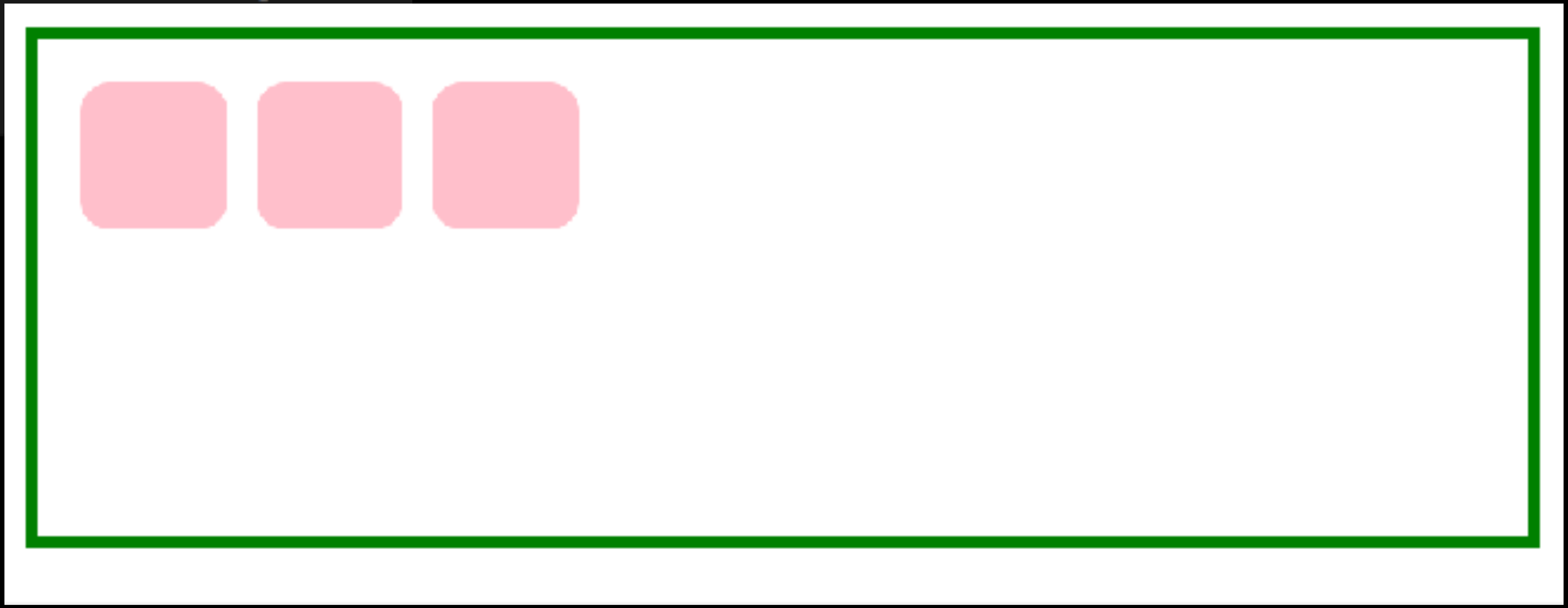


Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

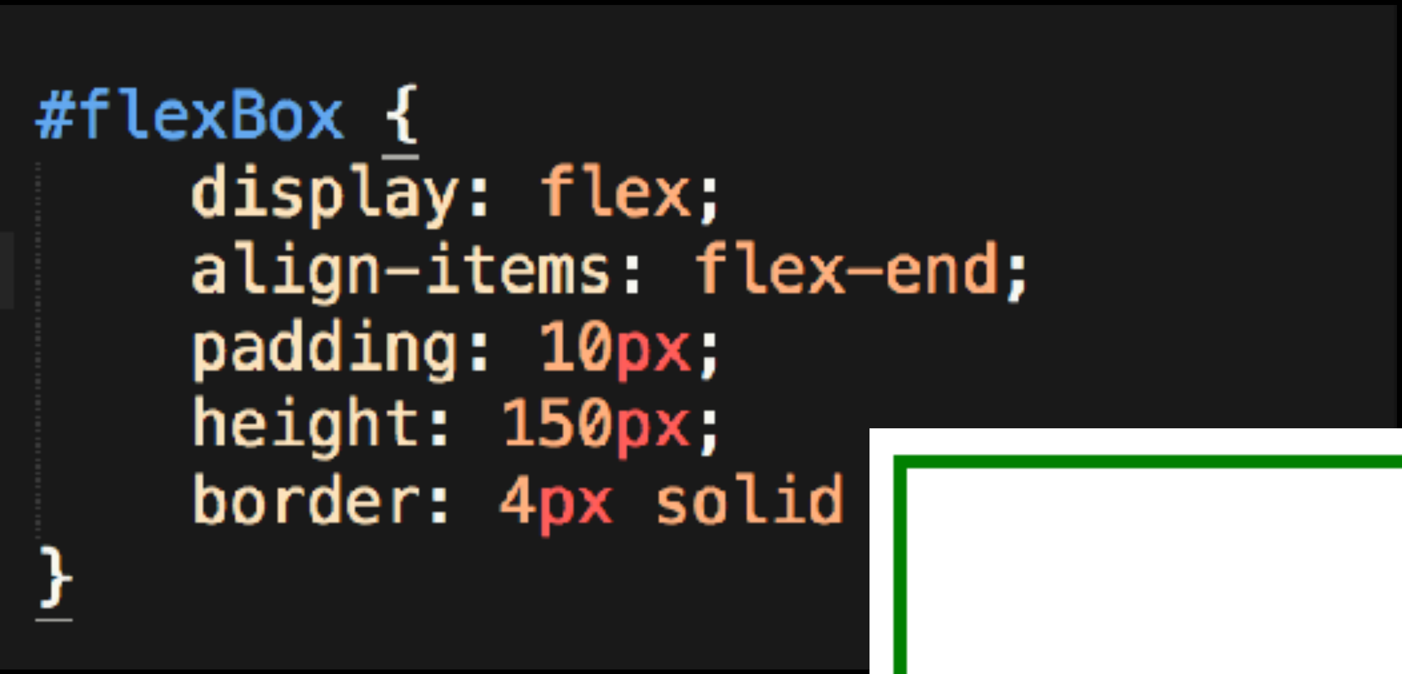


```
#flexBox {
    display: flex;
    align-items: flex-start;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

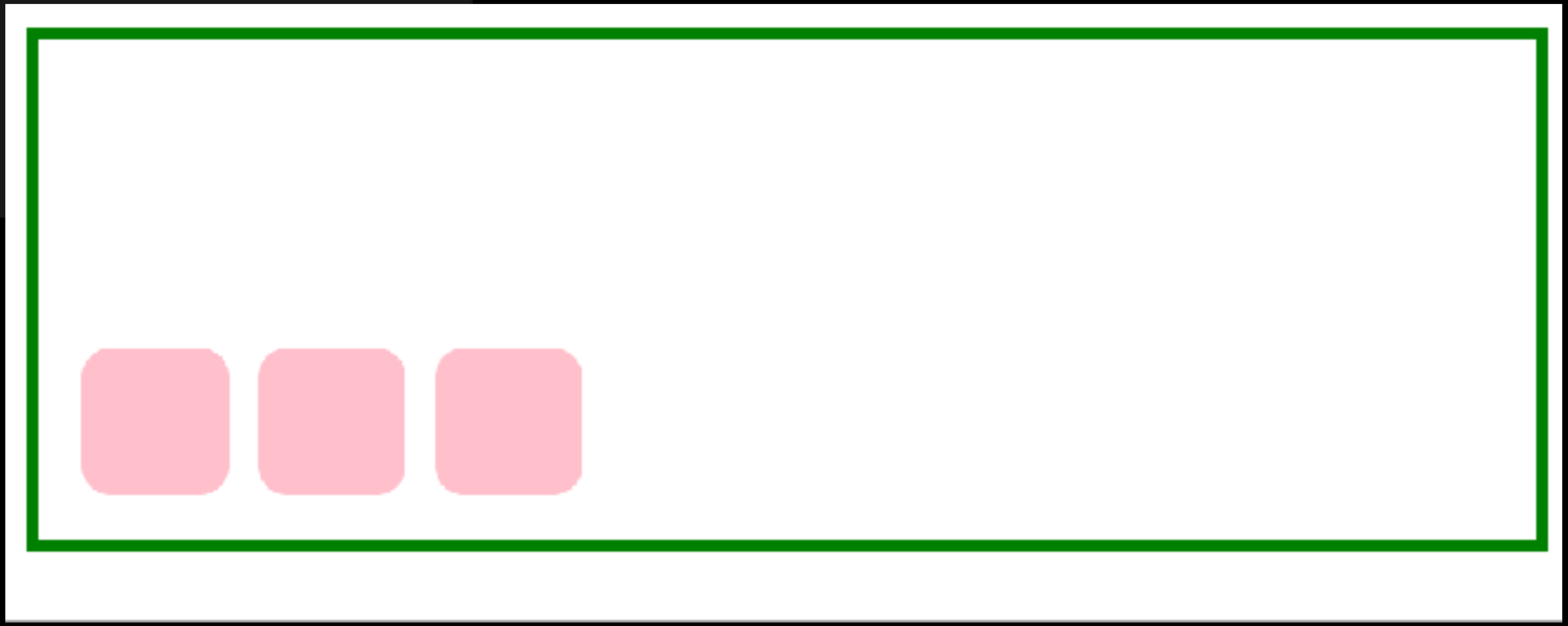


Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

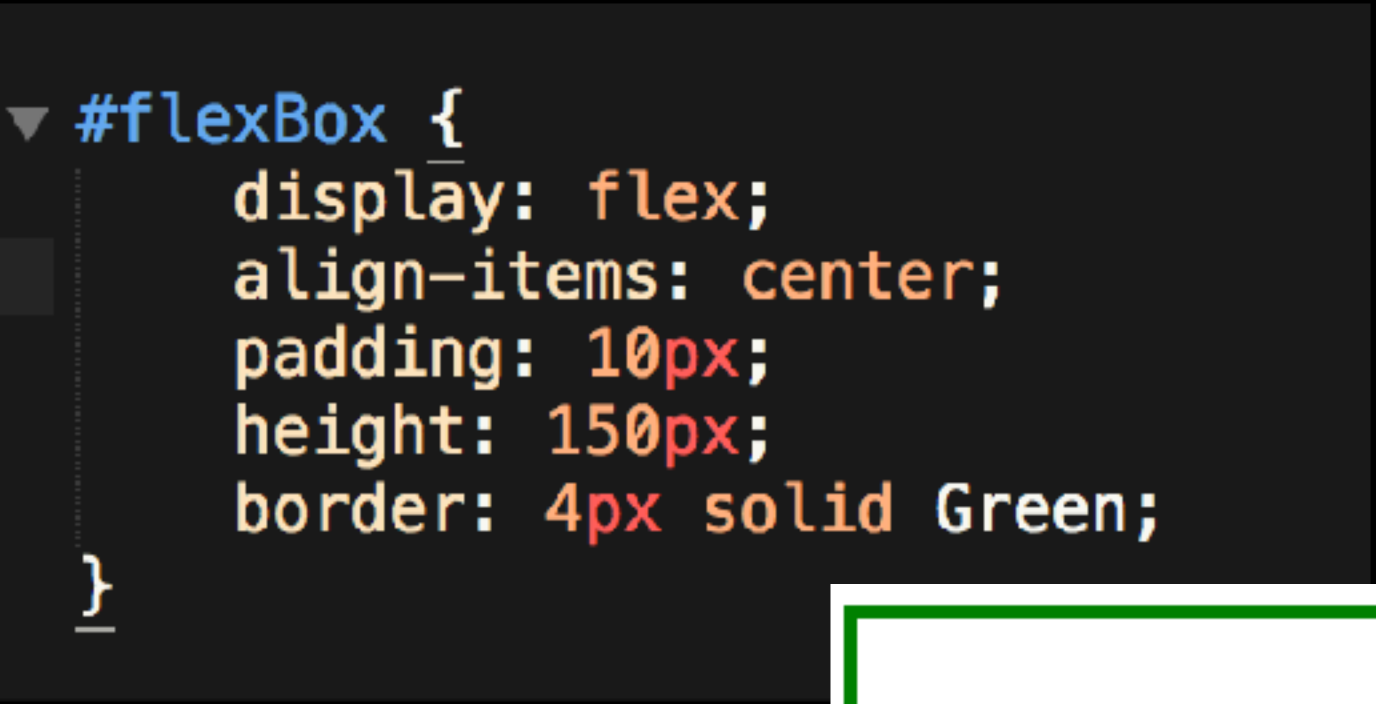


```
#flexBox {
    display: flex;
    align-items: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid
}
```

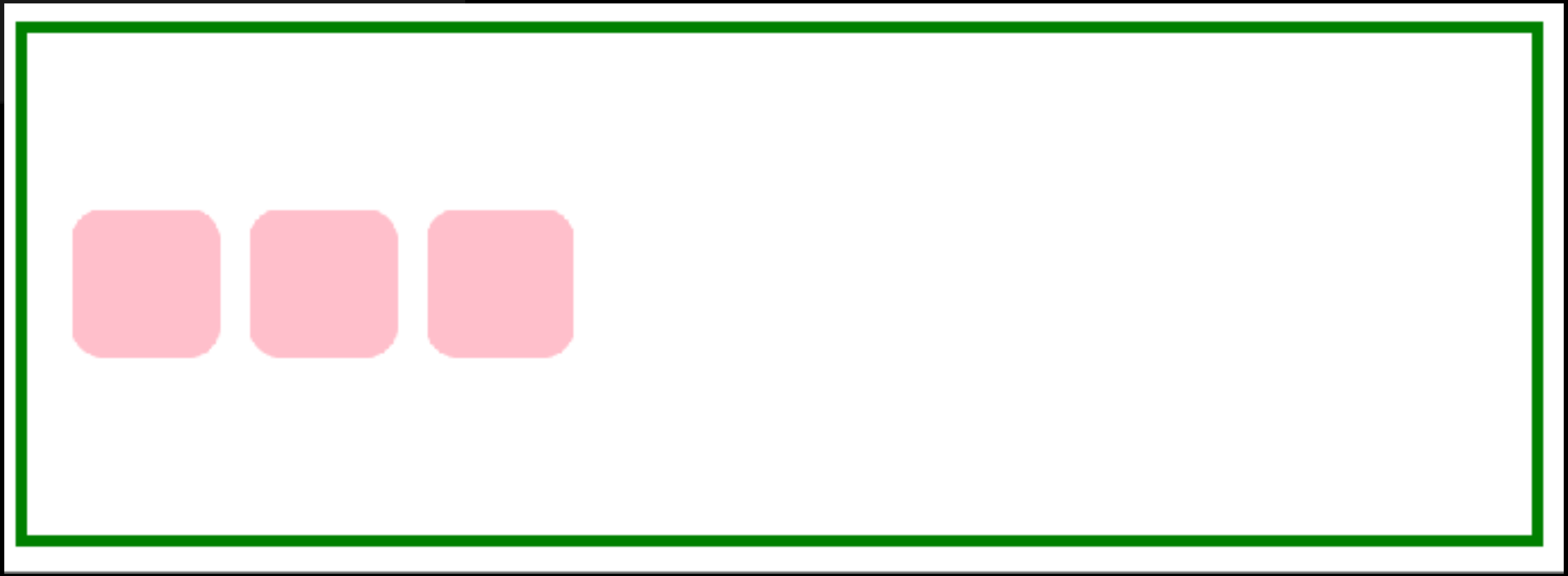


Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.

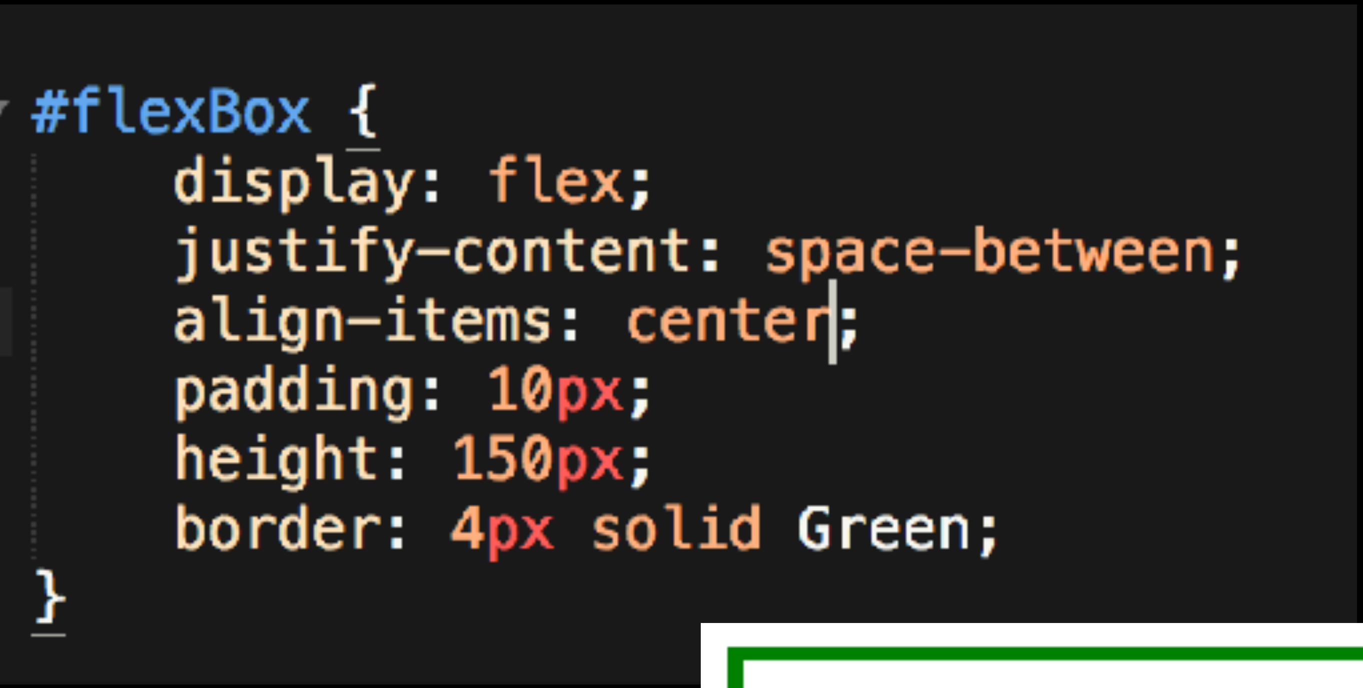


```
#flexBox {
    display: flex;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

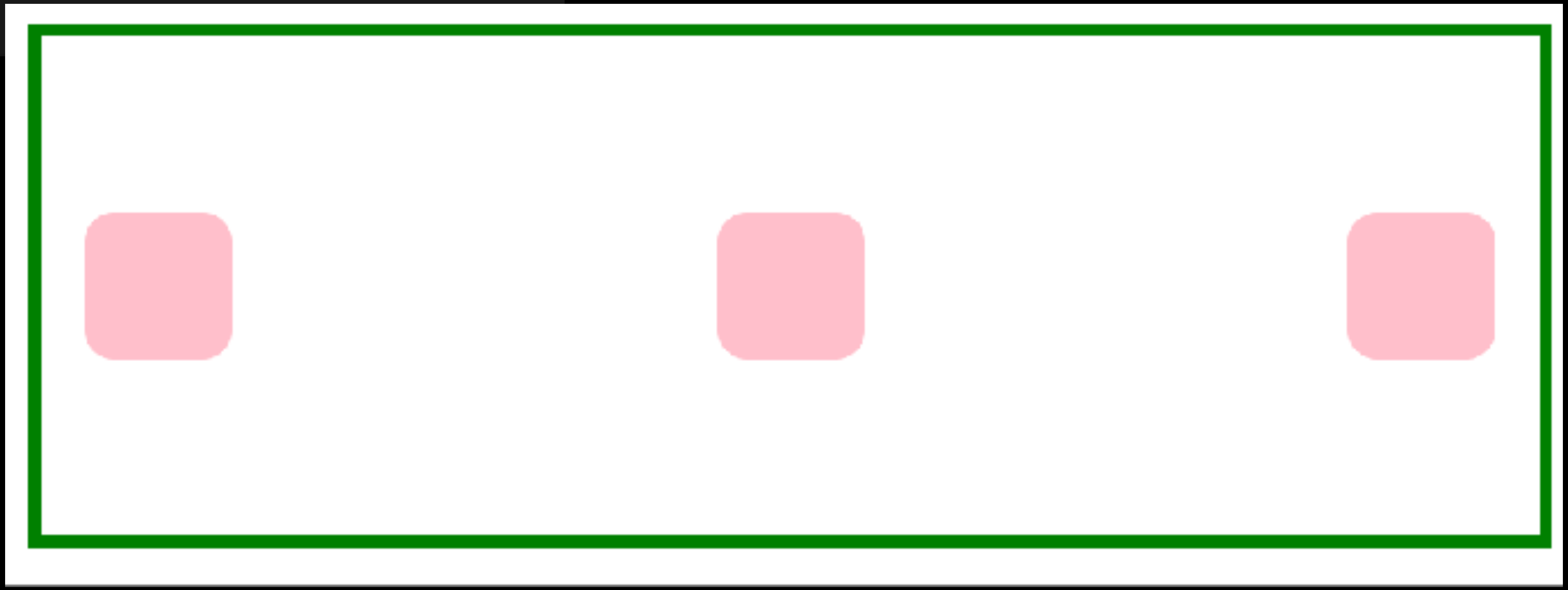


Flex Basics:

space-between + space-around

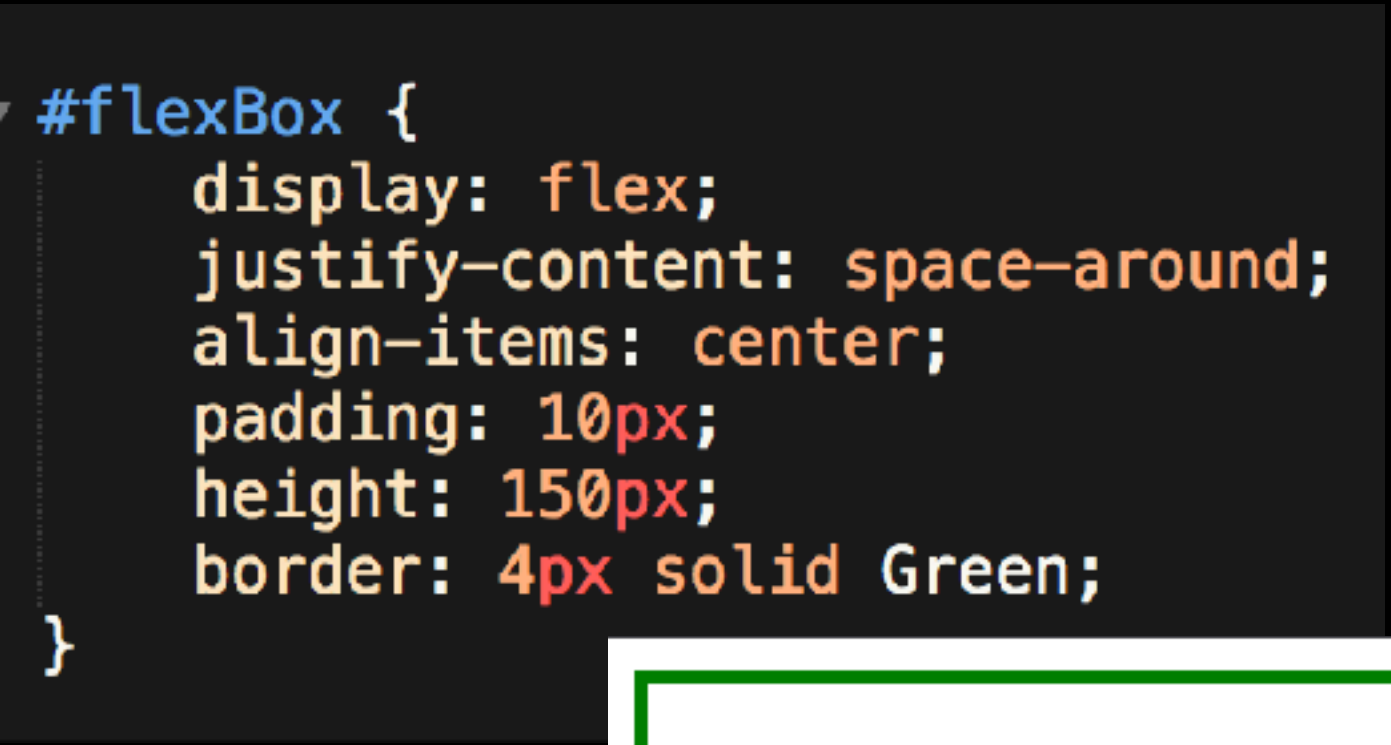


```
#flexBox {
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

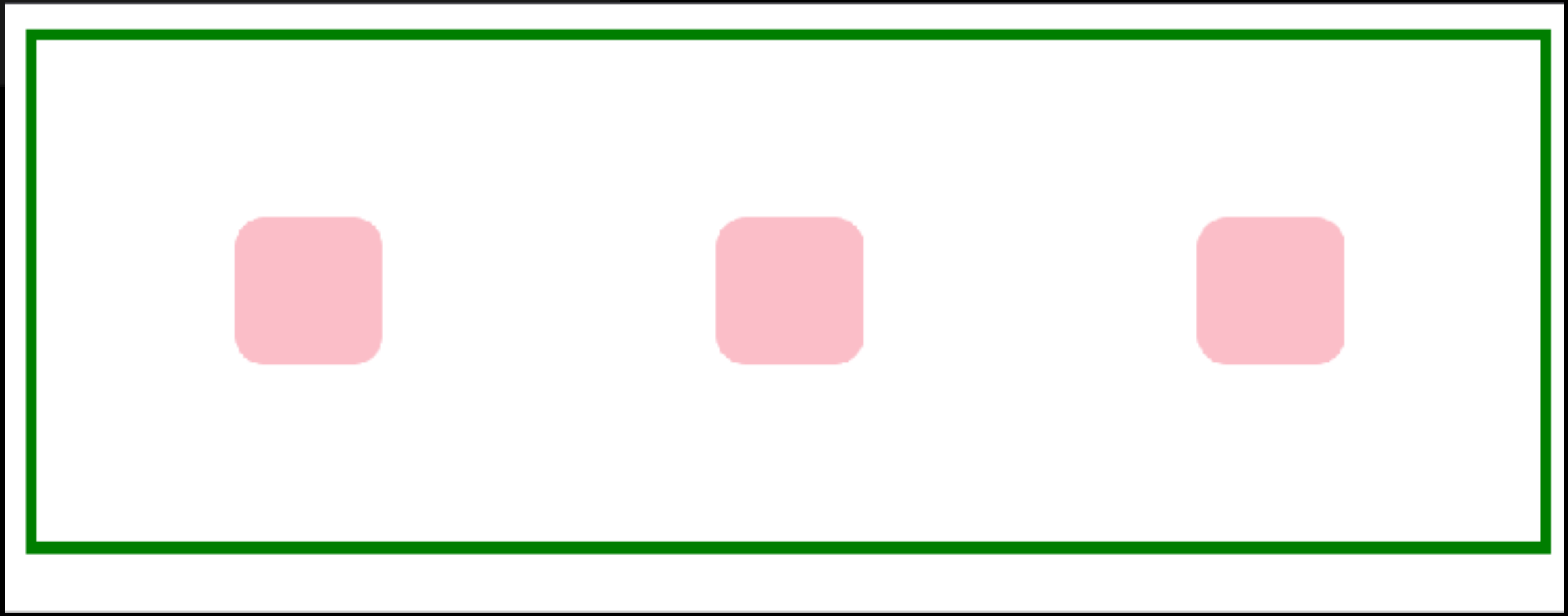


Flex Basics:

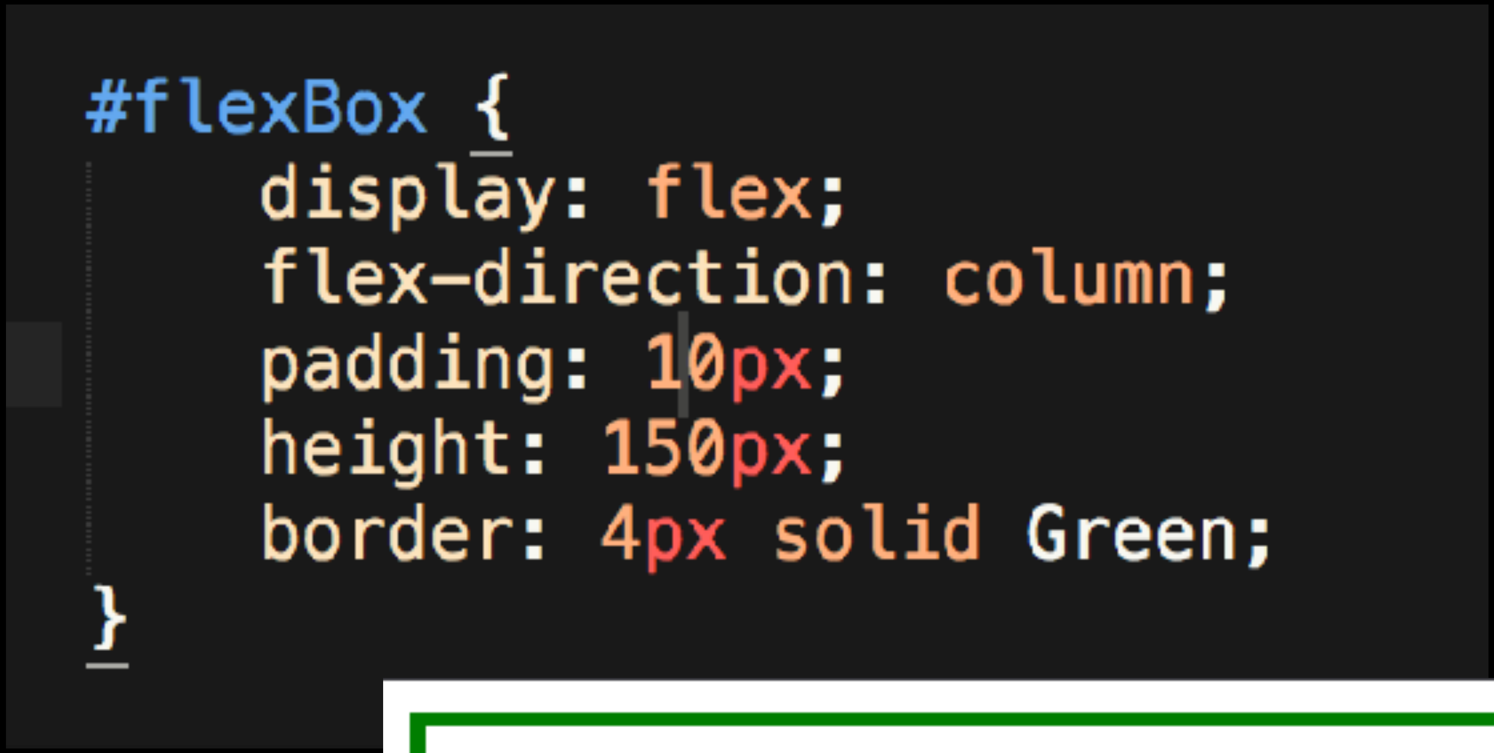
space-between + space-around



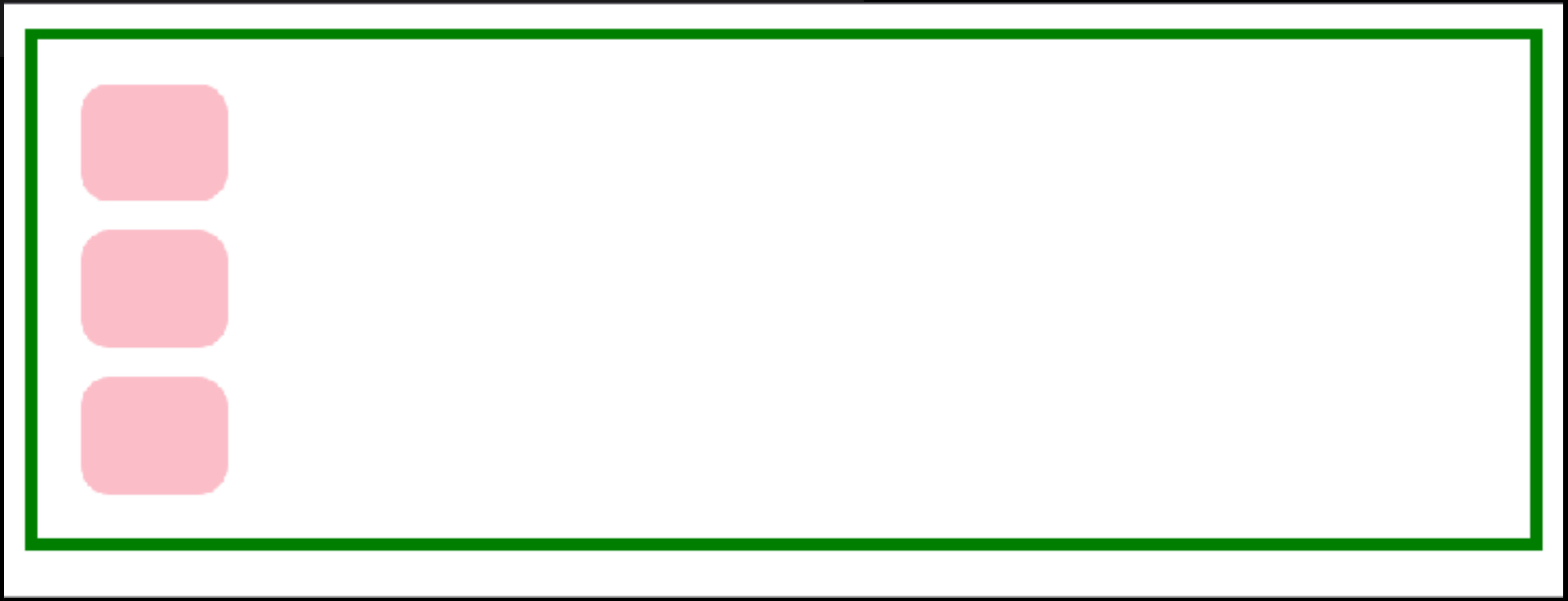
```
#flexBox {
    display: flex;
    justify-content: space-around;
    align-items: center;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



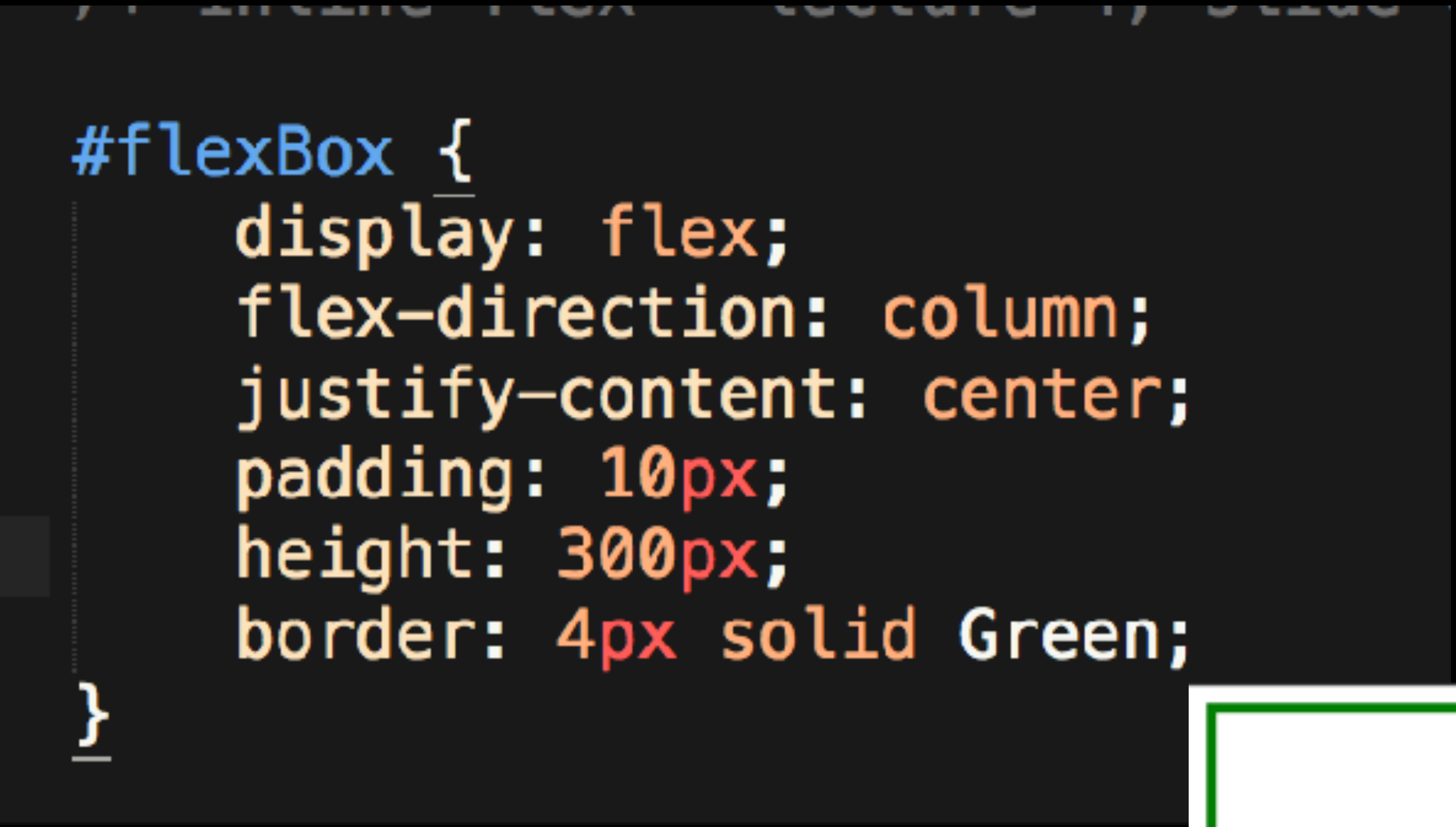
Flex Basics: flex-direction



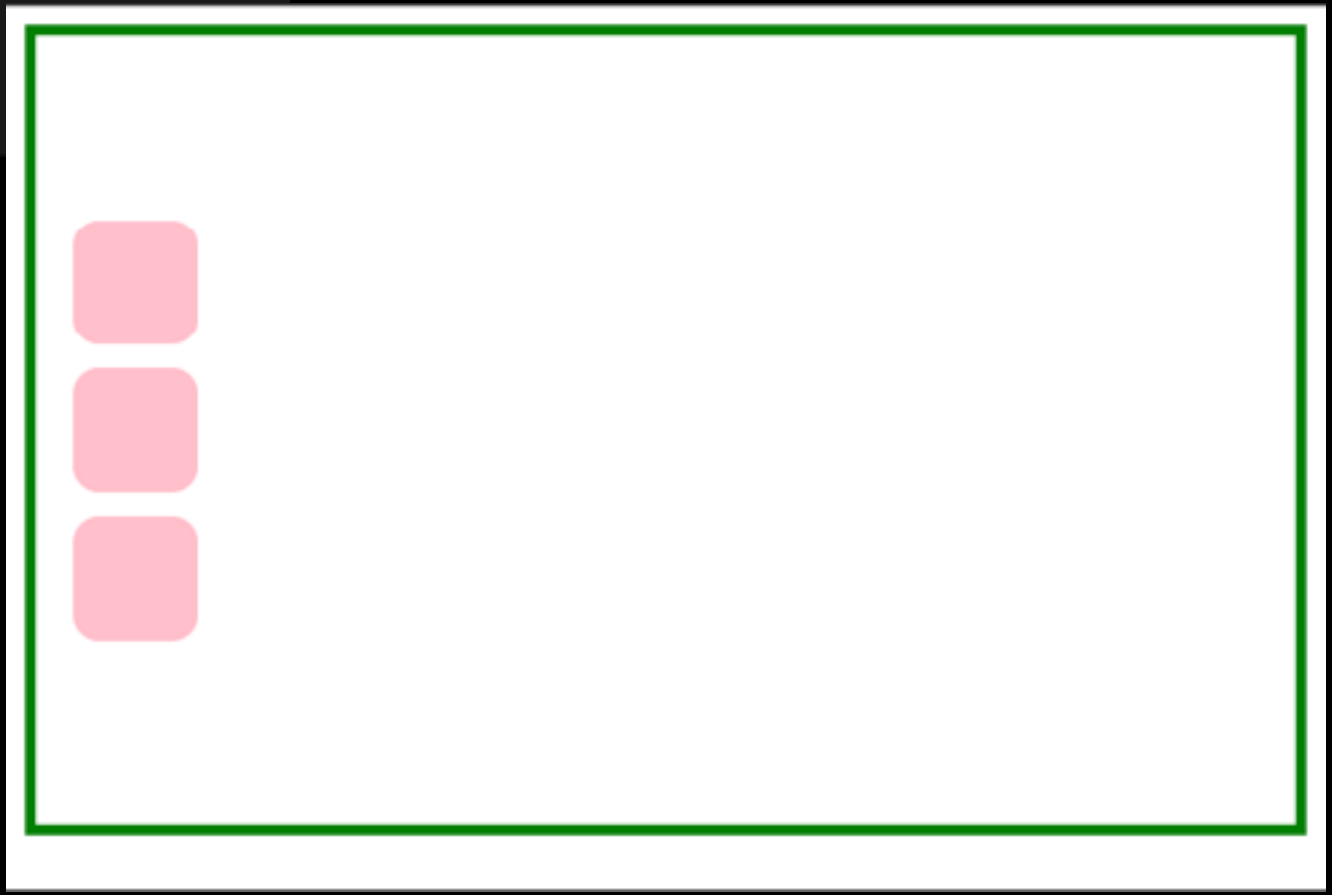
```
#flexBox {
    display: flex;
    flex-direction: column;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```



Flex Basics: flex-direction

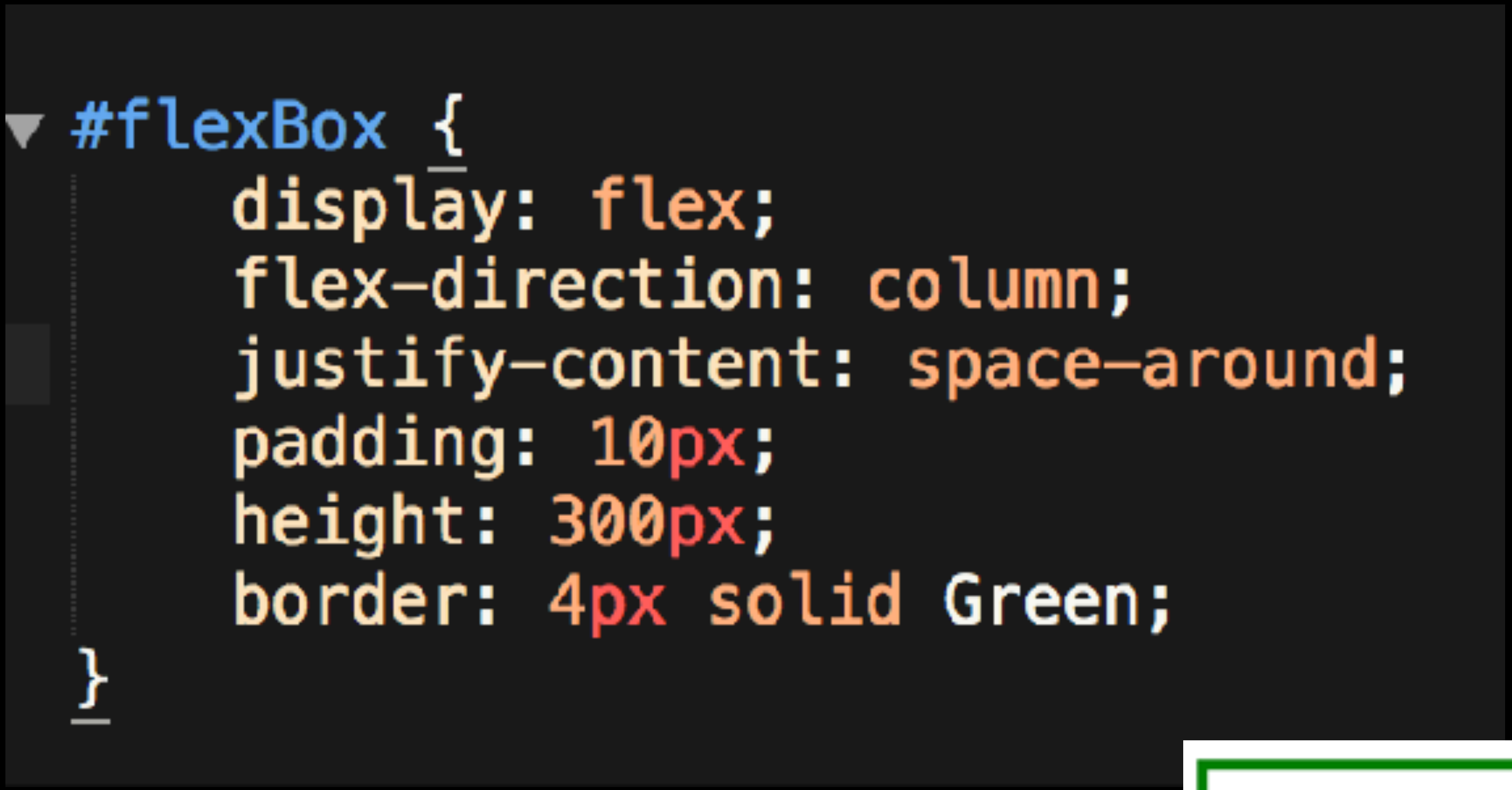


```css
#flexBox {
    display: flex;
    flex-direction: column;
    justify-content: center;
    padding: 10px;
    height: 300px;
    border: 4px solid Green;
}
```



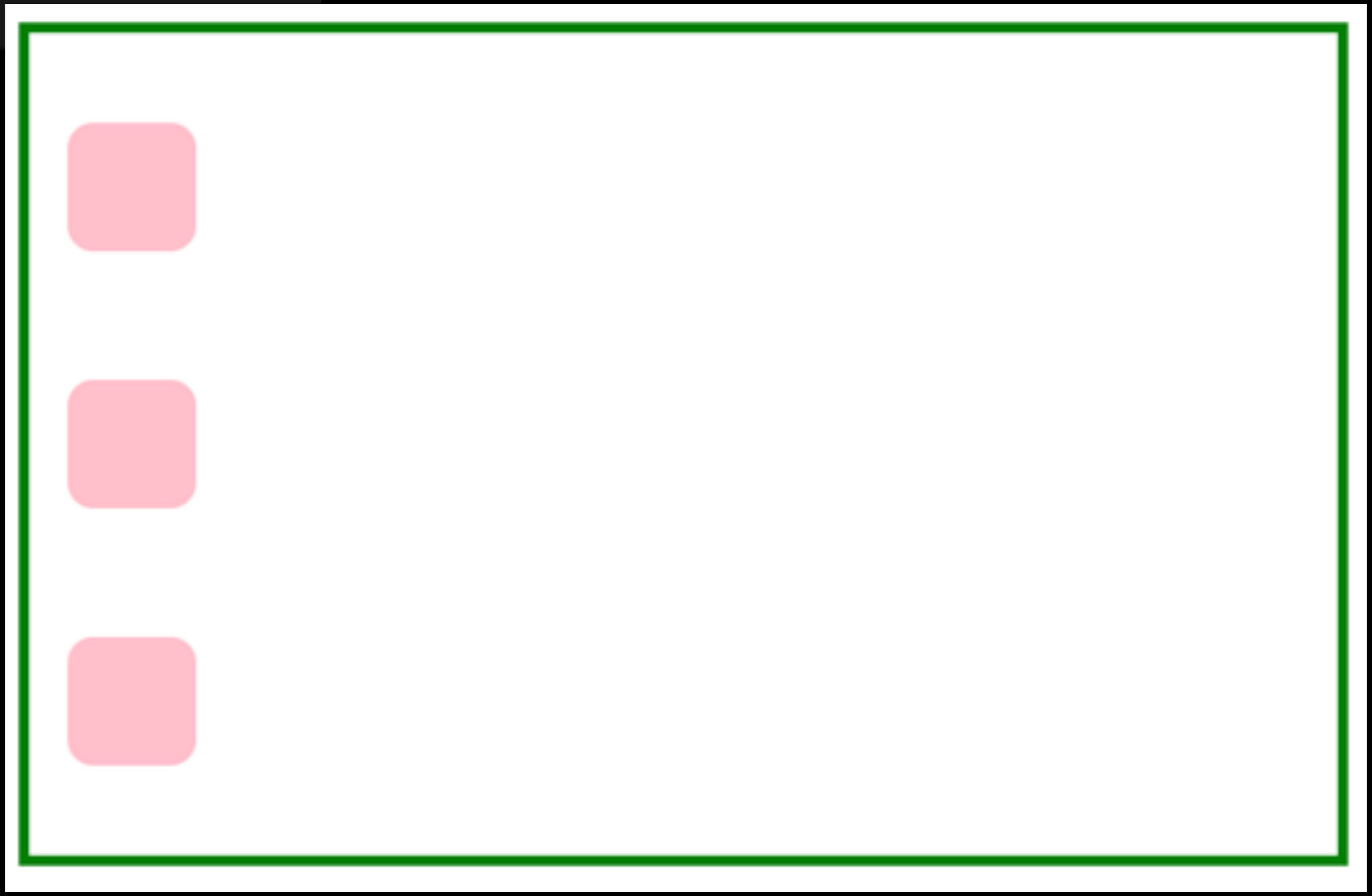
Now **justify-content** controls where the column is vertically in the box.

Flex Basics: flex-direction



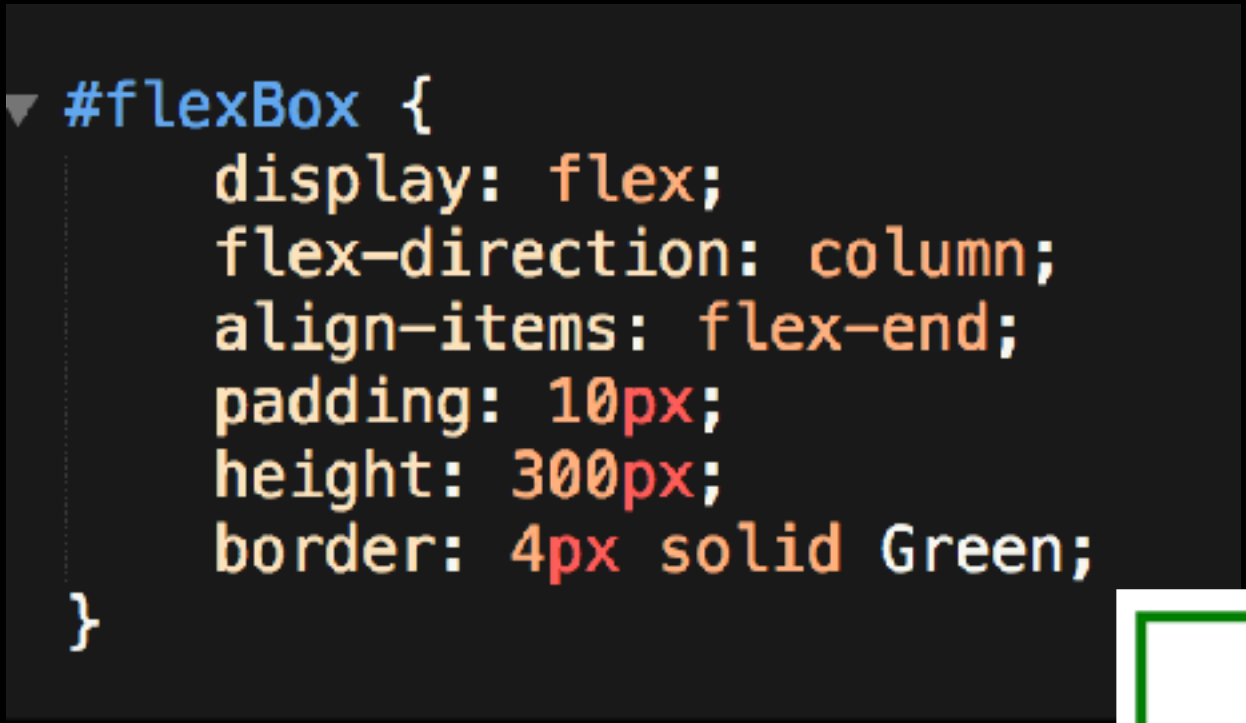
```
#flexBox {
      display: flex;
      flex-direction: column;
      justify-content: space-around;
      padding: 10px;
      height: 300px;
      border: 4px solid Green;
}
```

And you can also lay out columns instead of rows.



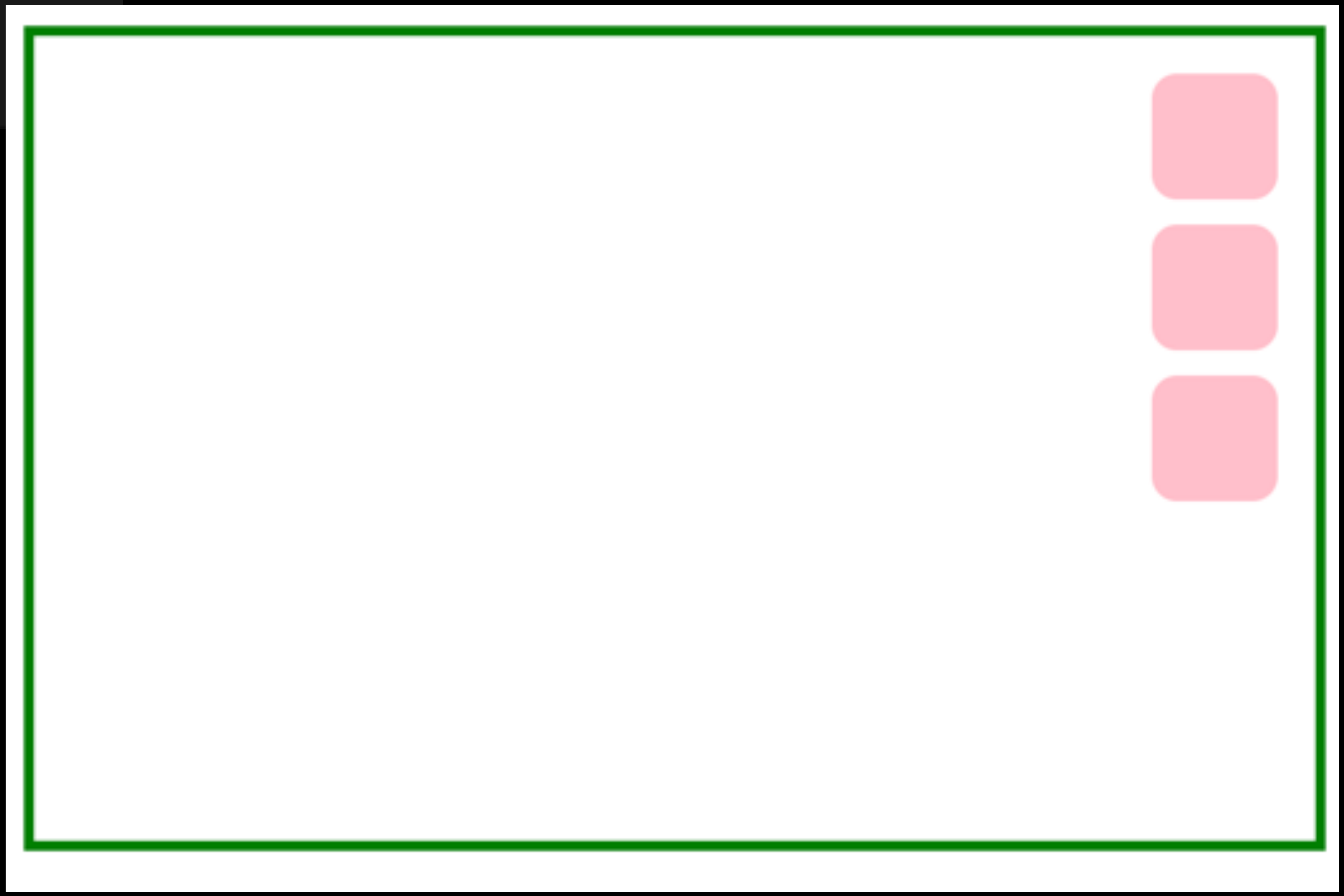
Now **justify-content** controls where the column is vertically in the box.

Flex Basics: flex-direction



```
#flexBox {
      display: flex;
      flex-direction: column;
      align-items: flex-end;
      padding: 10px;
      height: 300px;
      border: 4px solid Green;
}
```

And you can also lay out columns instead of rows.



Now **align-items** controls where the column is horizontally in the box.

Flex Basis

Flex items have an initial width*, which, by default is either:

- The content width, or
- The explicitly set **width** property of the element, or
- The explicitly set **flex-basis** property of the element

This initial width* of the flex item is called the **flex basis**.

*width in the case of rows; height in the case of columns

Flex Basis

Flex items have an initial width*, which, by default is either:

- The content width, or

- The explicitly set **width** property of the element, or

- The explicitly set **flex-basis** property of the element

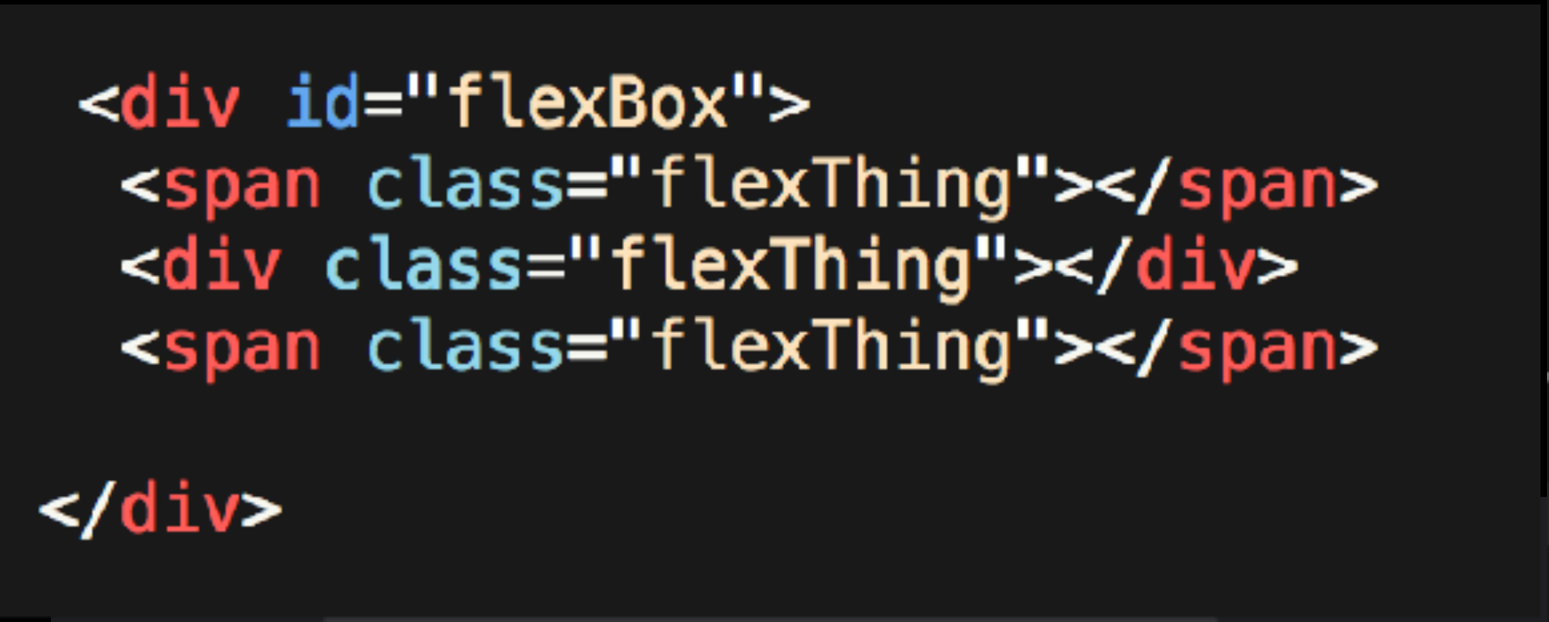
This initial width* of the flex item is called the **flex basis**.

The explicit width* of a flex item is respected **for all flex items**, regardless of whether the flex item is inline, block, or inline-block.

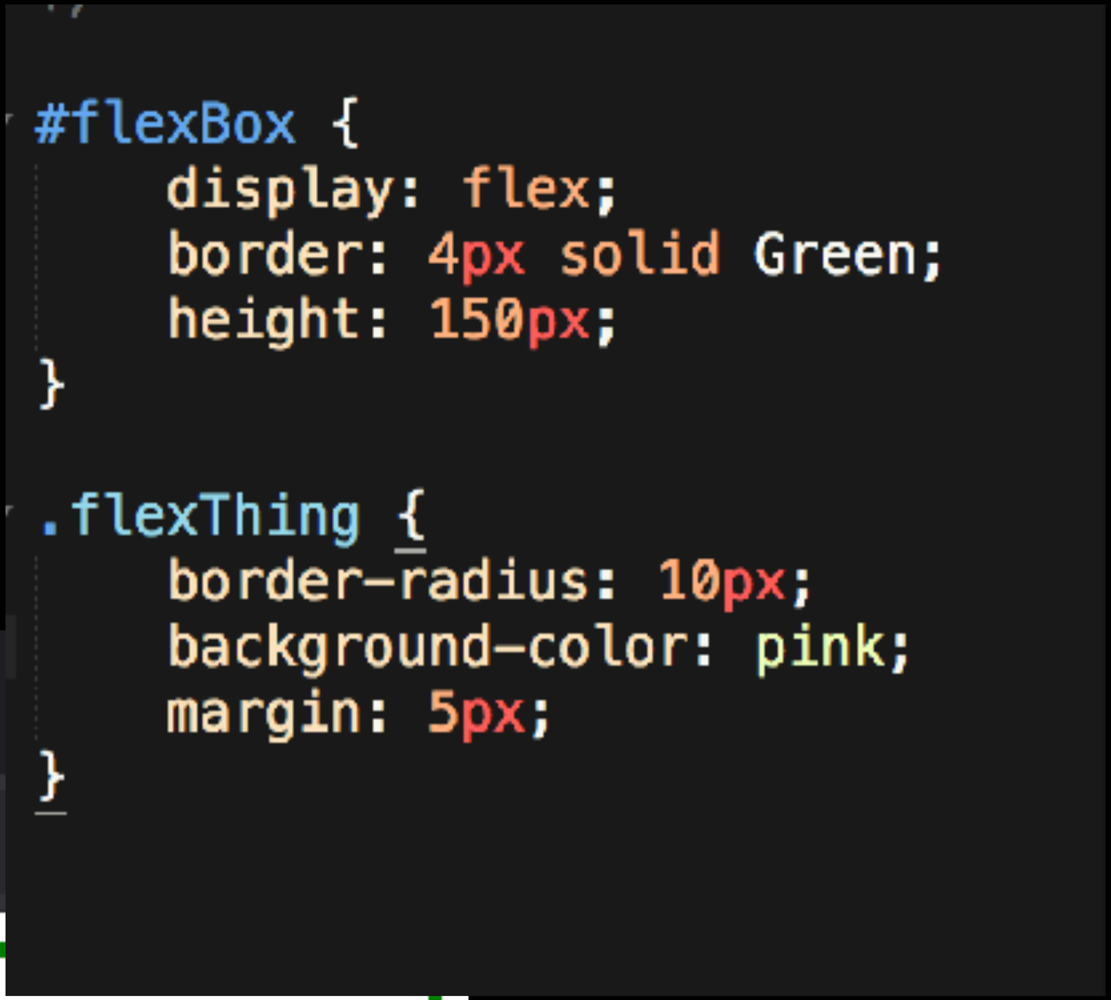
*width in the case of rows; height in the case of columns

Flex Basis

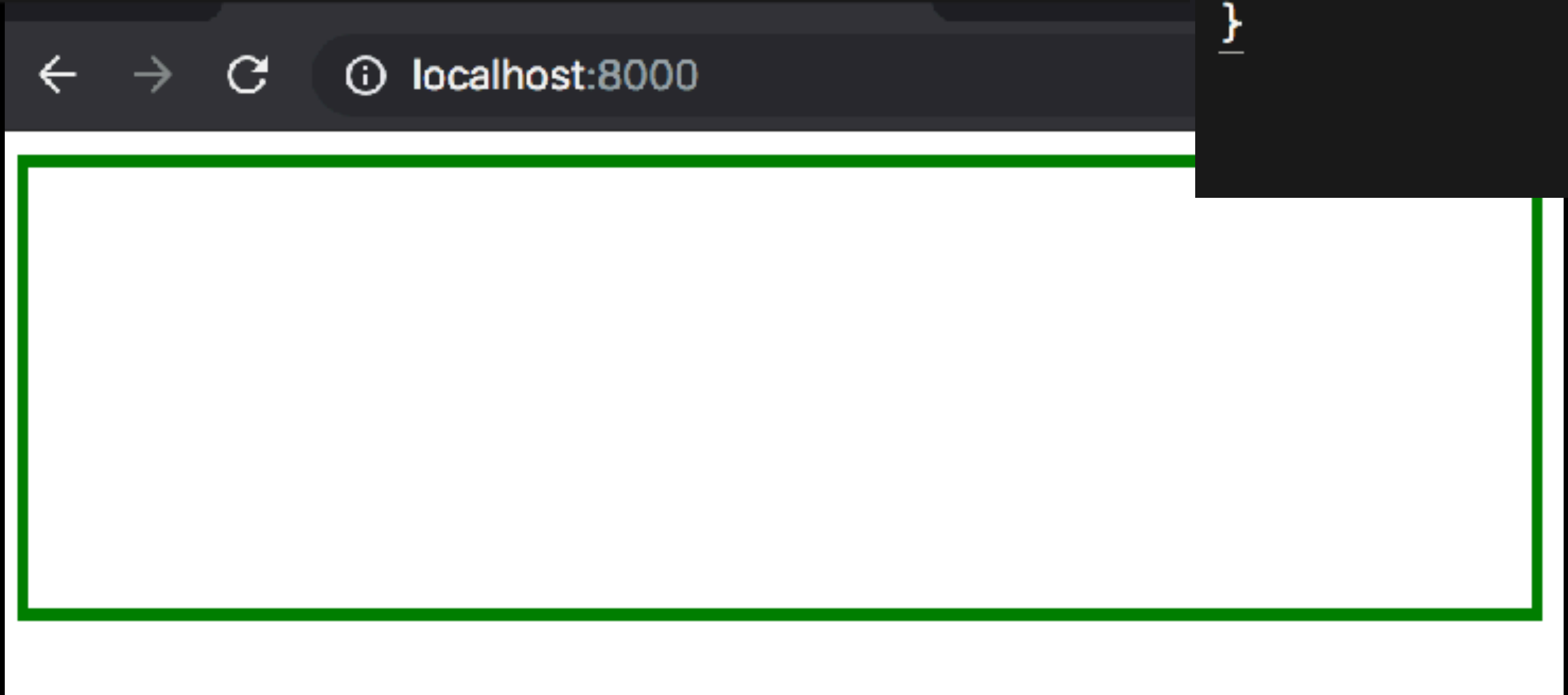
If we unset the height and width, our flex items disappears, because the **flex basis** is now the content size, which is empty:



```html
<div id="flexBox">
 <span class="flexThing"></span>
 <div class="flexThing"></div>
 <span class="flexThing"></span>

</div>
```



```css
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```



flex-shrink

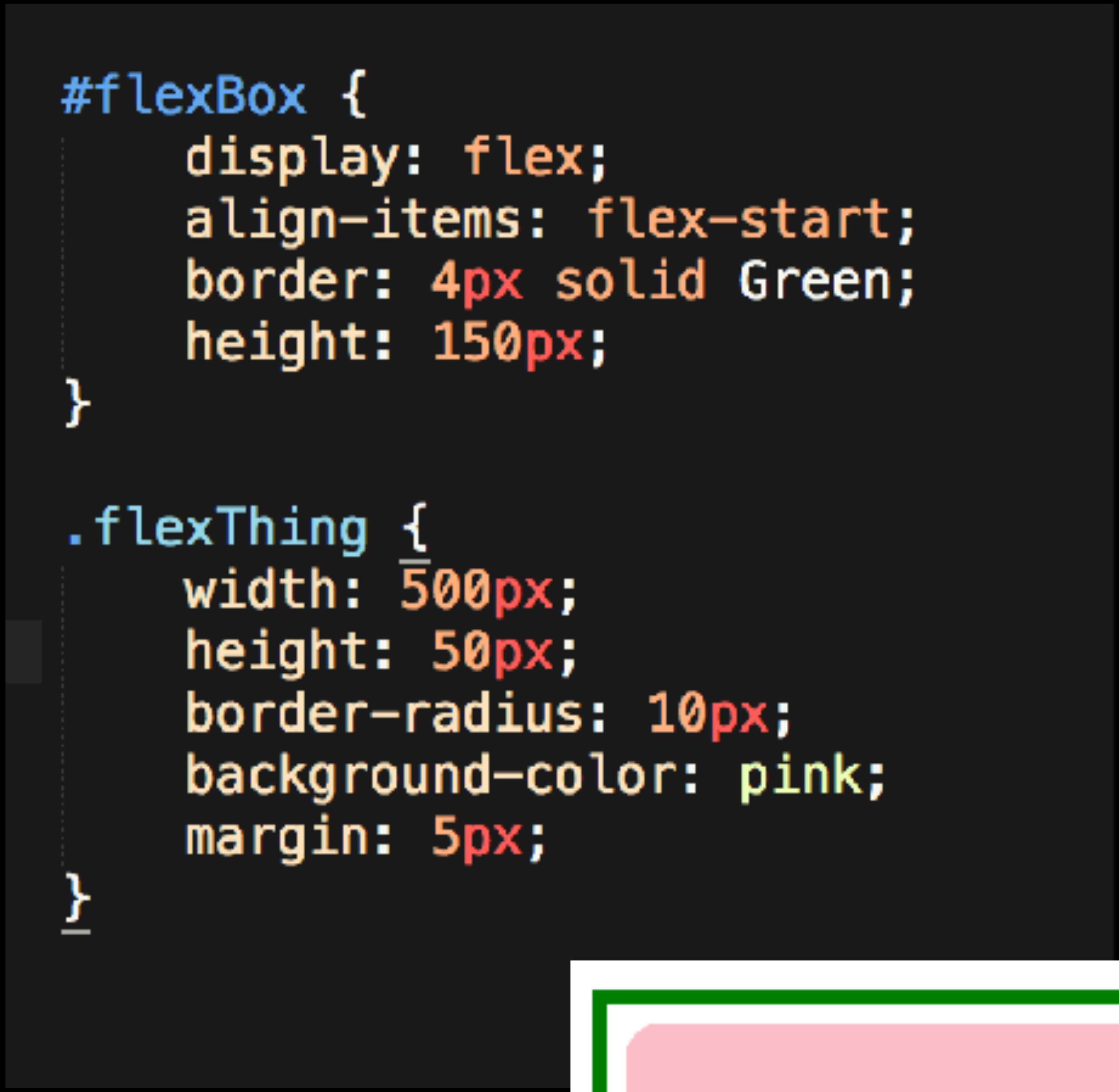
The width* of the flex item can automatically shrink **smaller** than the **flex basis** via the **flex-shrink** property:

flex-shrink:
- If set to **1**, the flex item shrinks itself as small as it can in the space available
- If set to **0**, the flex item does not shrink.

Flex items have **flex-shrink**: **1 by default**.

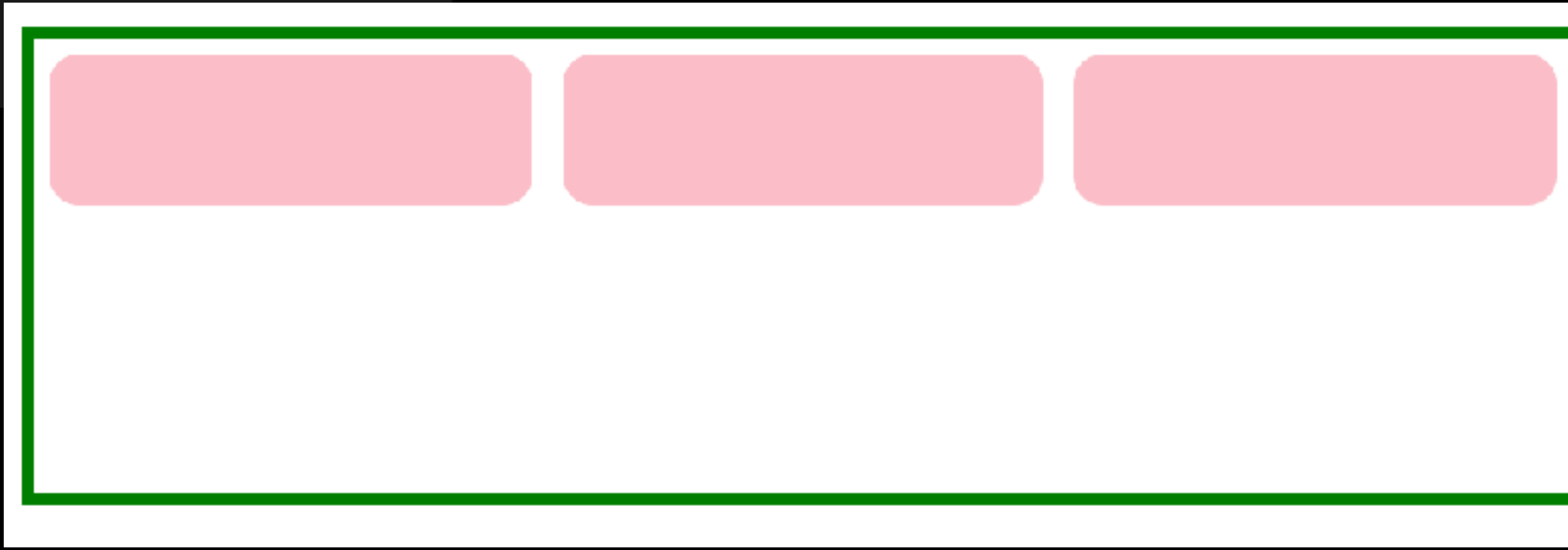
*width in the case of rows;
height in the case of columns

flex-shrink

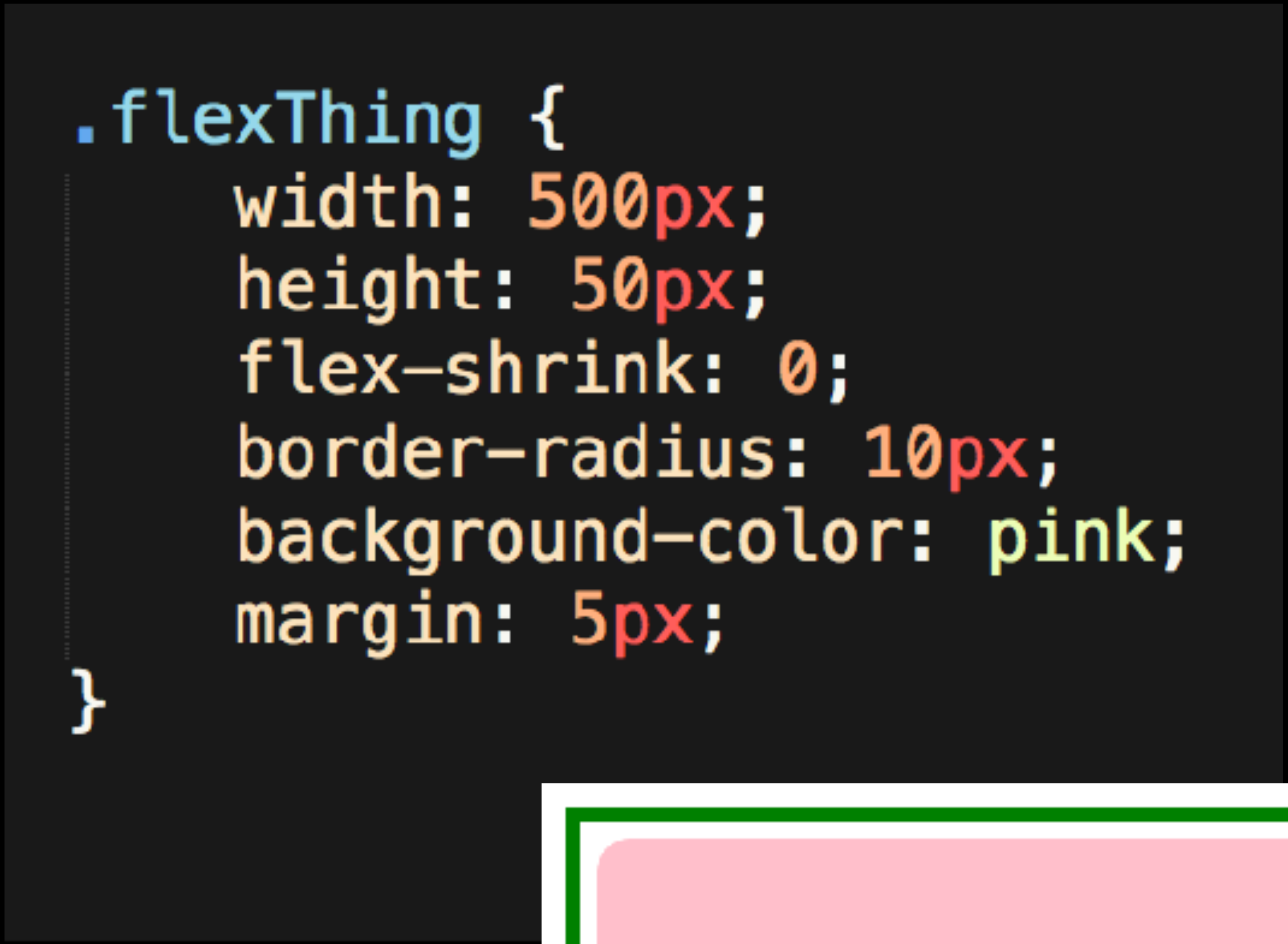


```
#flexBox {
    display: flex;
    align-items: flex-start;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    width: 500px;
    height: 50px;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

The flex items' widths all shrink to fit the width of the container.

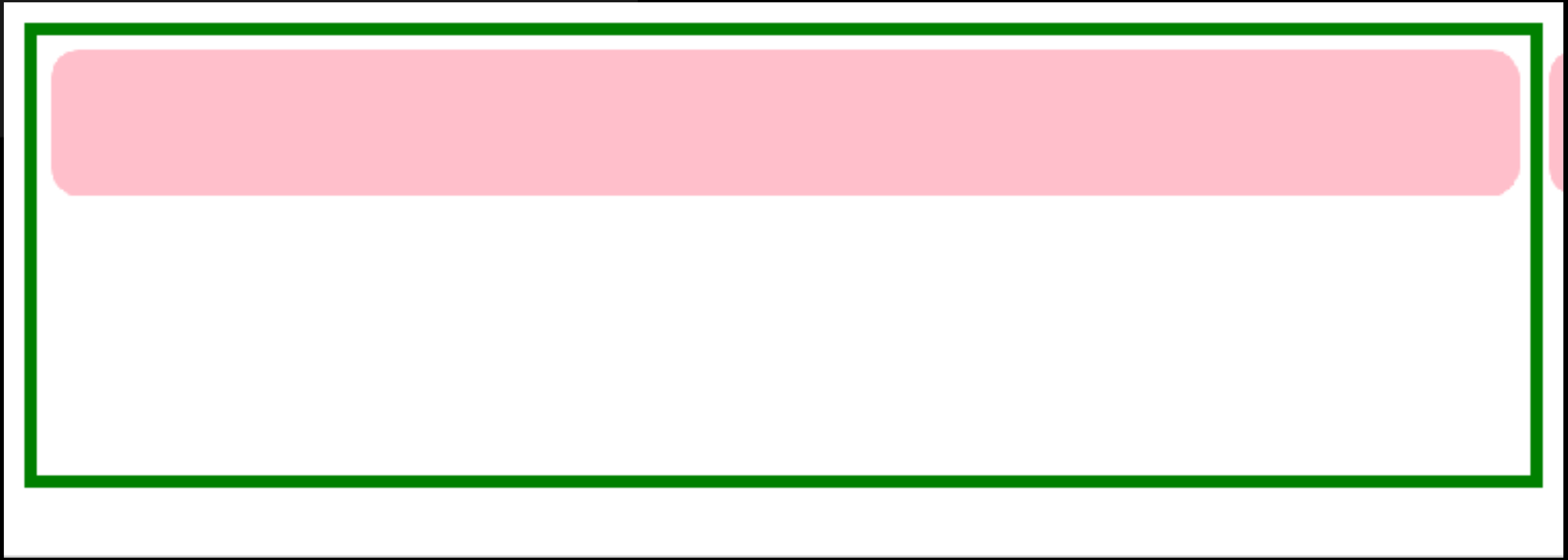


flex-shrink



```
.flexThing {
    width: 500px;
    height: 50px;
    flex-shrink: 0;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

Setting **flex-shrink**: 0;
undoes the shrinking behavior,
and the flex items do not
shrink in any circumstance:



flex-grow

The width* of the flex item can automatically **grow larger** than the flex basis via the flex-grow property:

flex-grow:

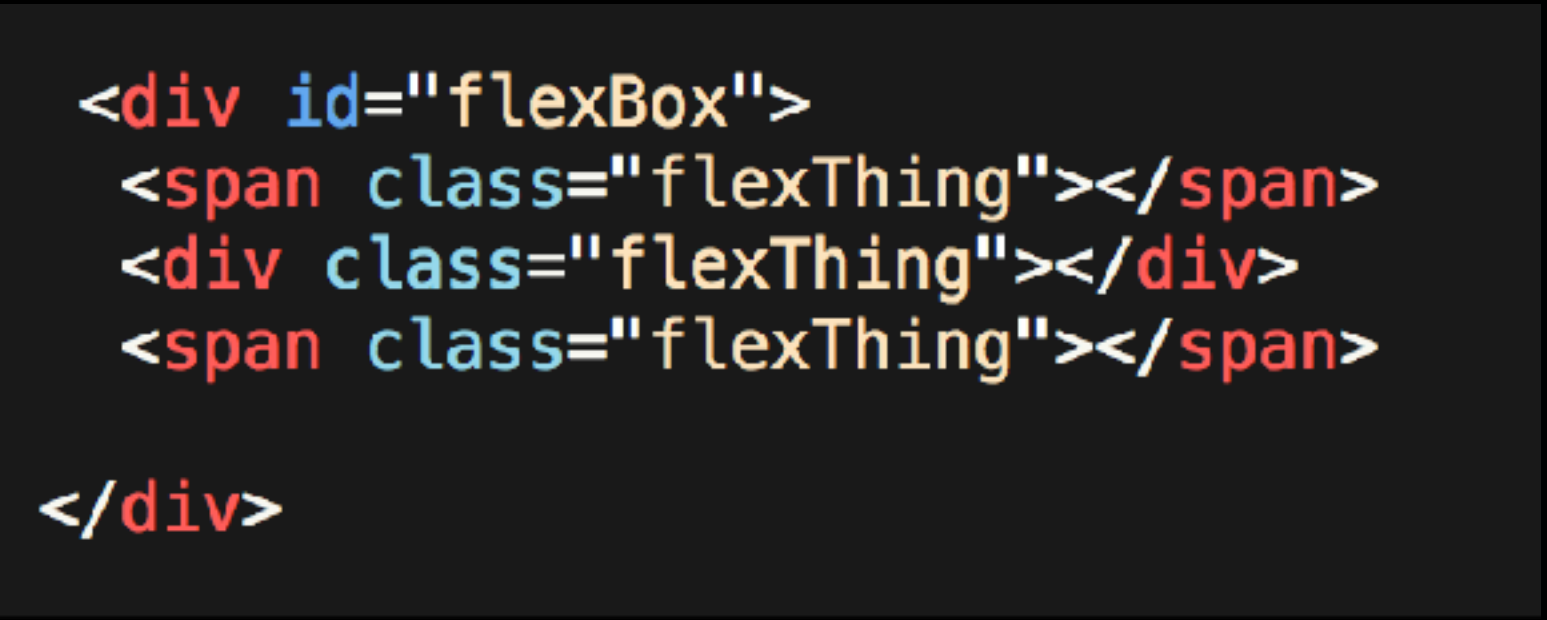
- If set to 1, the flex item grows itself as large as it can in the space remaining
- If set to 0, the flex item does not grow

Flex items have flex-grow: **0 by default.**

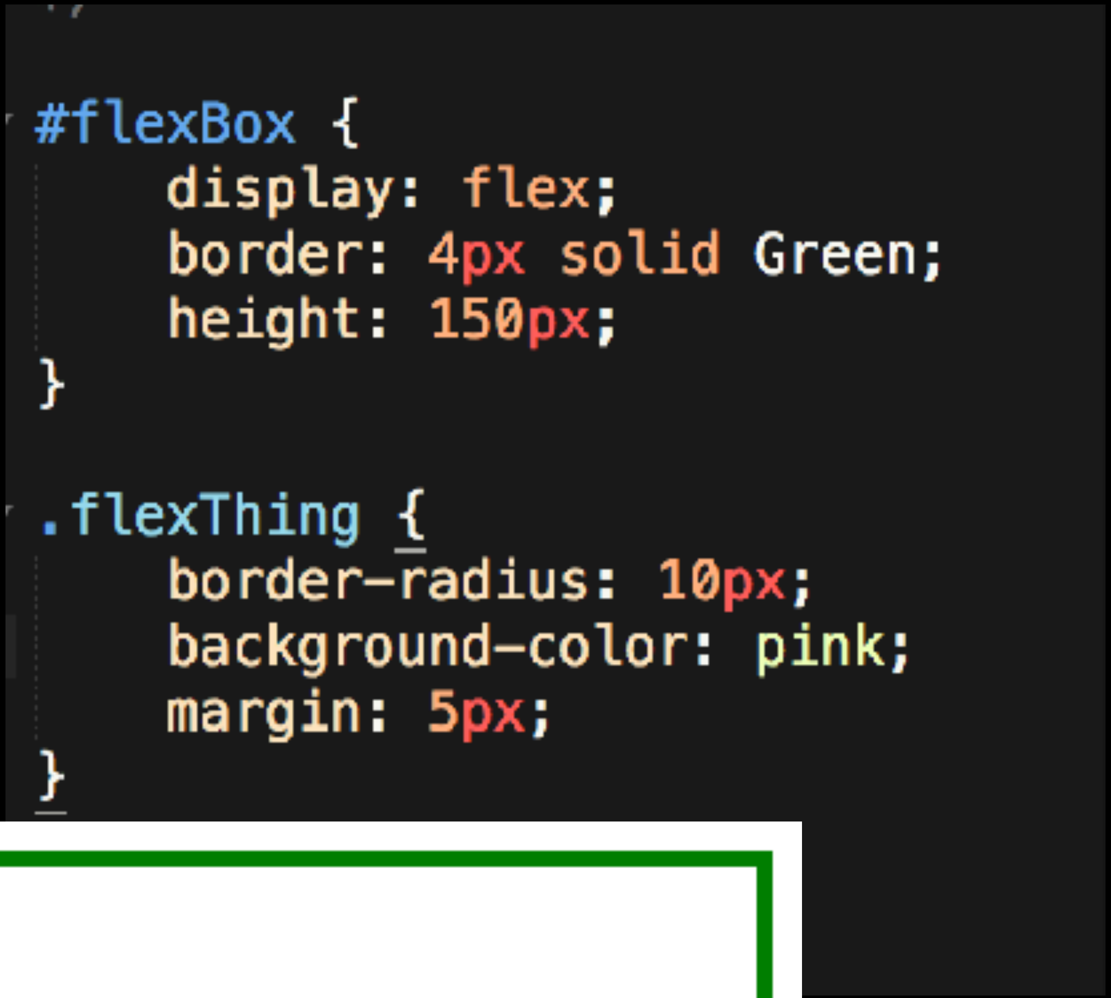
*width in the case of rows;
height in the case of columns

flex-grow

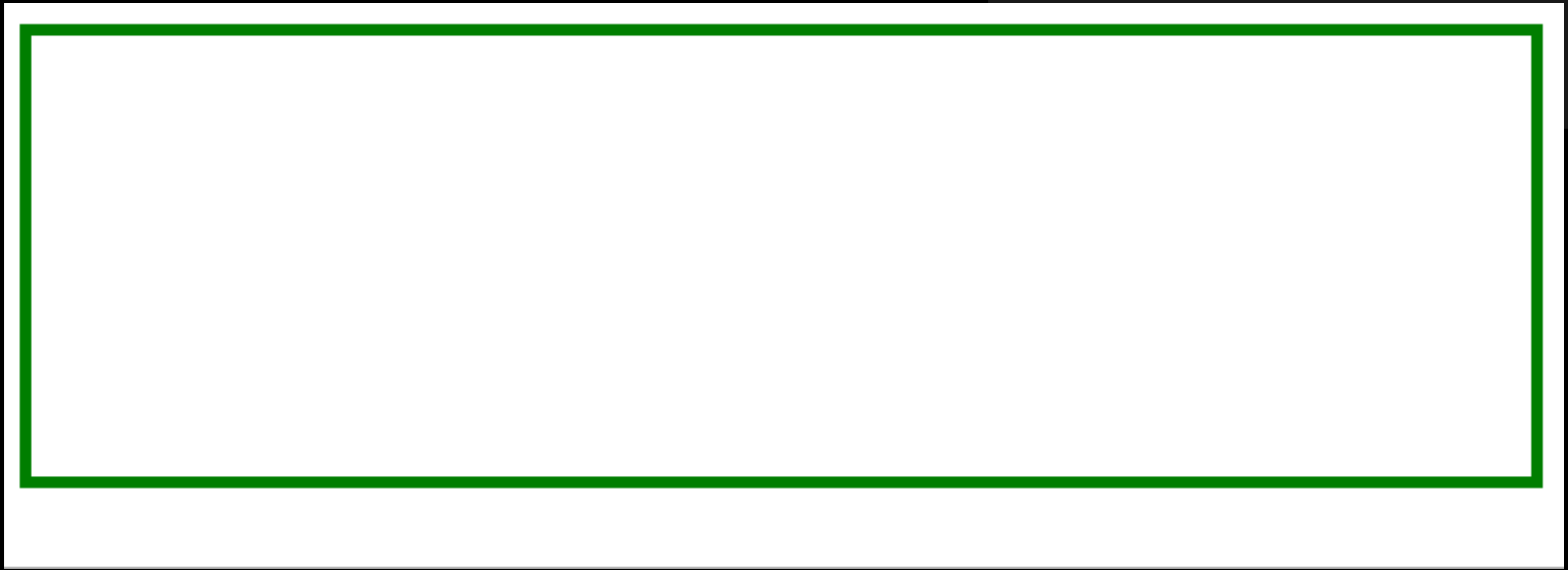
Let’s unset the height + width of our flex items again.



```
<div id="flexBox">
  <span class="flexThing"></span>
  <div class="flexThing"></div>
  <span class="flexThing"></span>

</div>
```

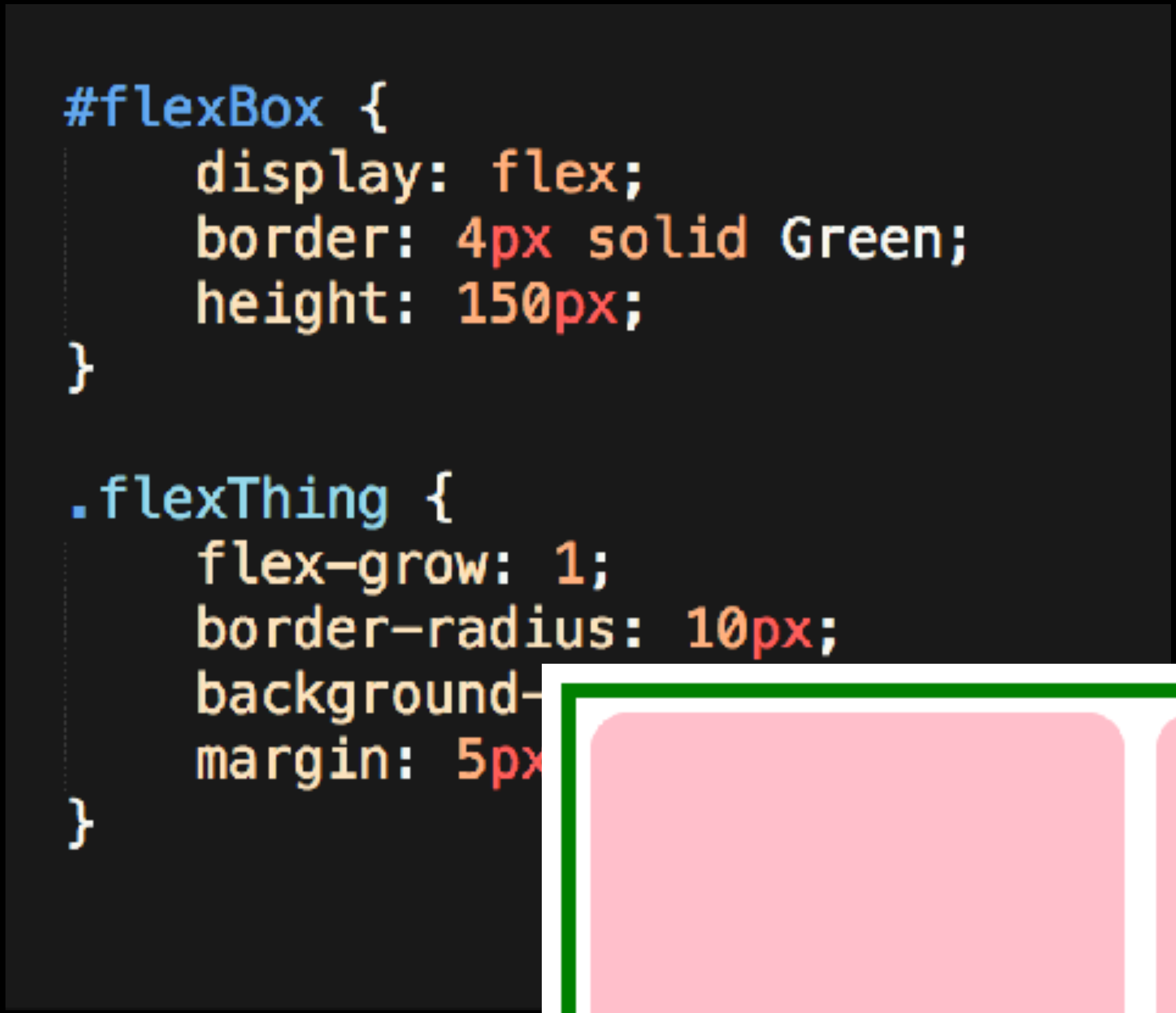


```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```

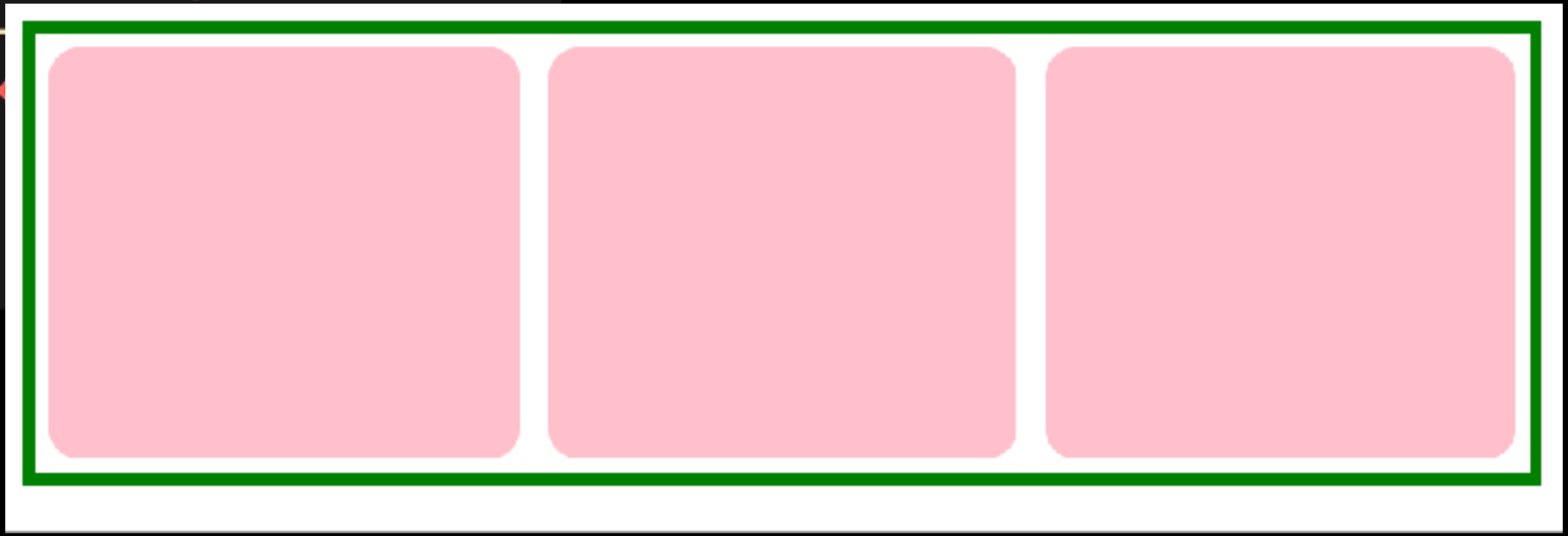


flex-grow

if we set **flex-grow**: 1;
the flex items fill the empty space.



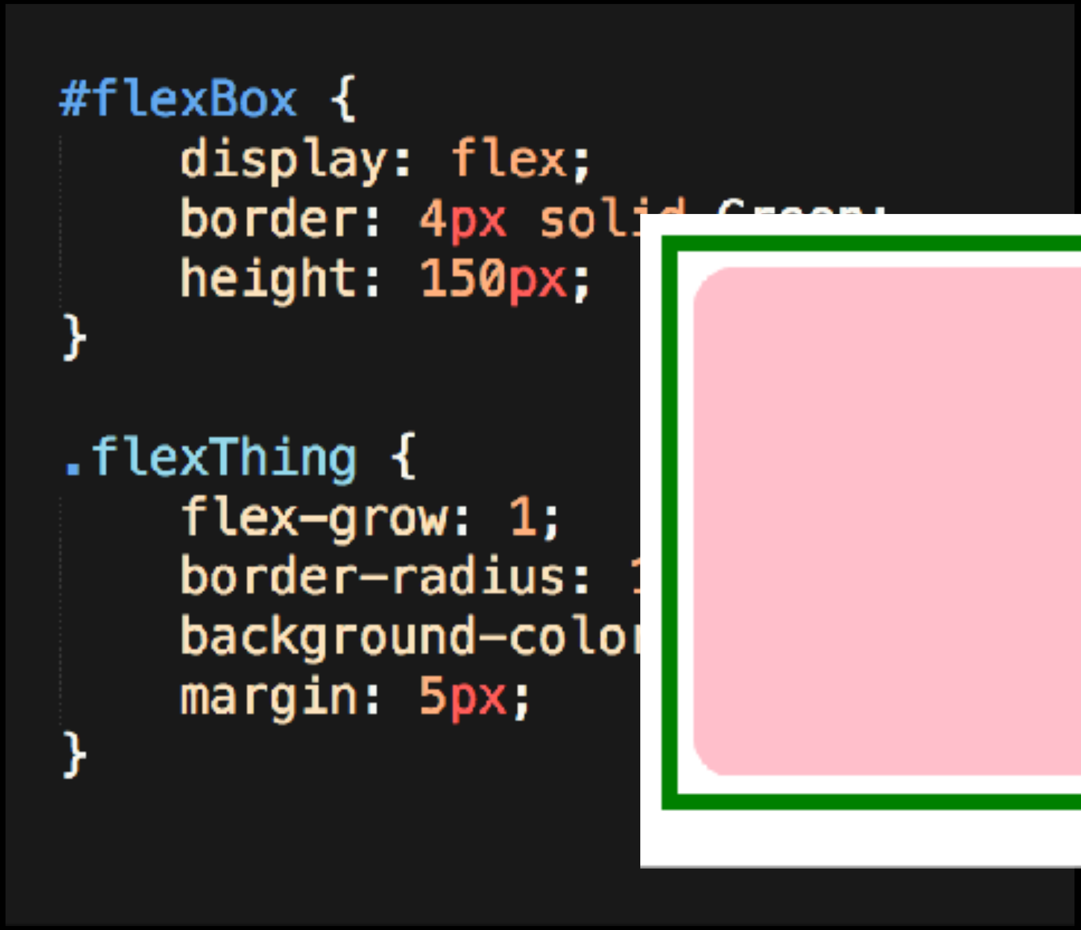
```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-
    margin: 5px
}
```



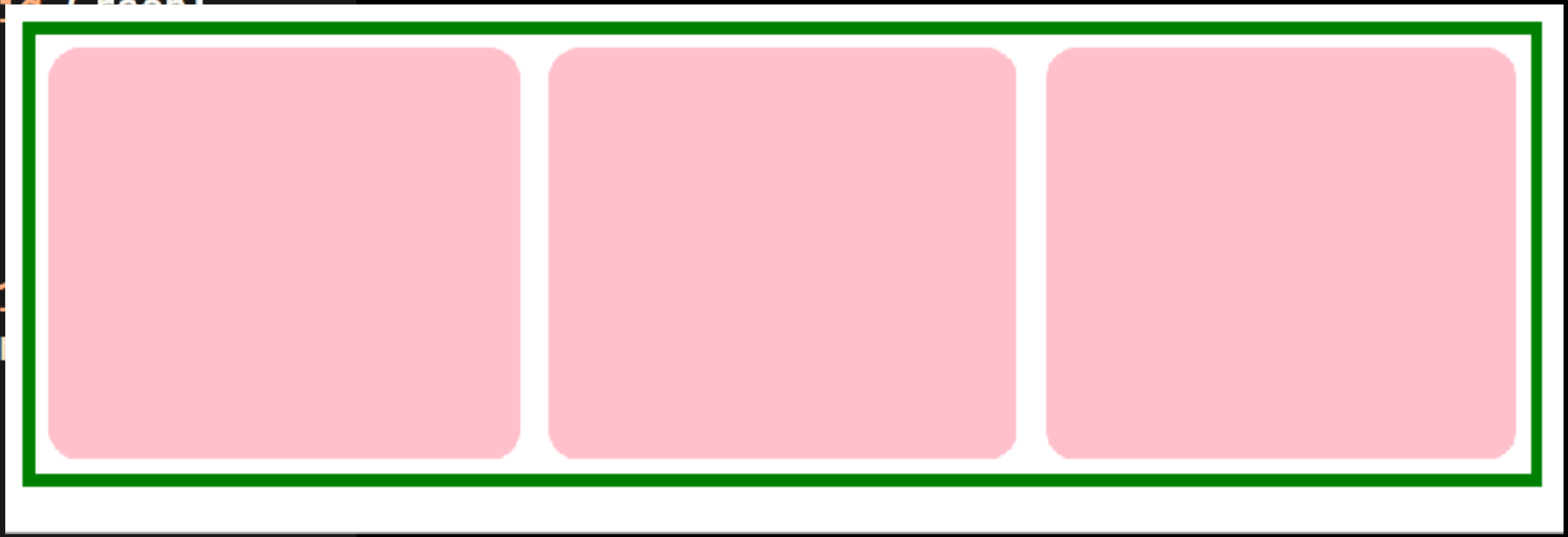
flex item height**?

note that **flex-grow** only controls width*

So why does the height** of the flex items seem to ‘grow’ as well?



```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 1
    background-colo
    margin: 5px;
}
```

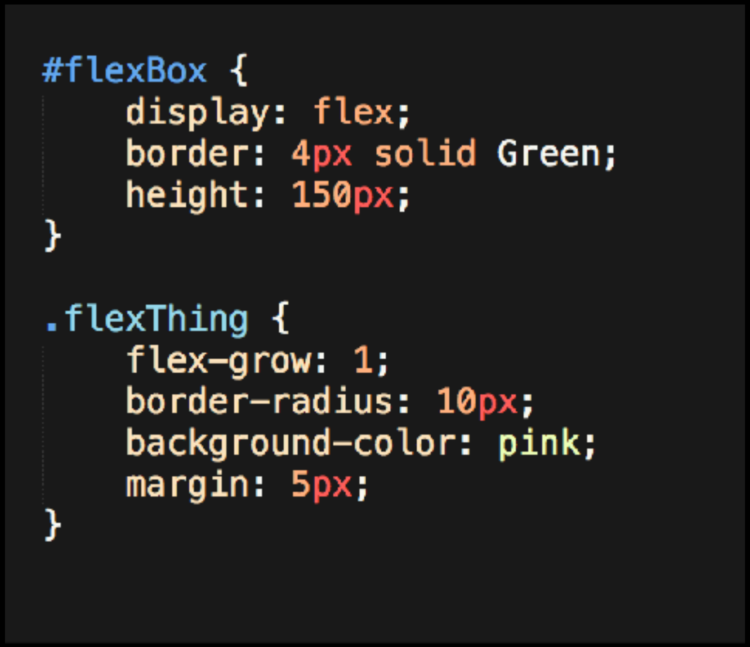


*width in the case of rows; height in the case of columns
**height in the case of rows; width in the case of columns

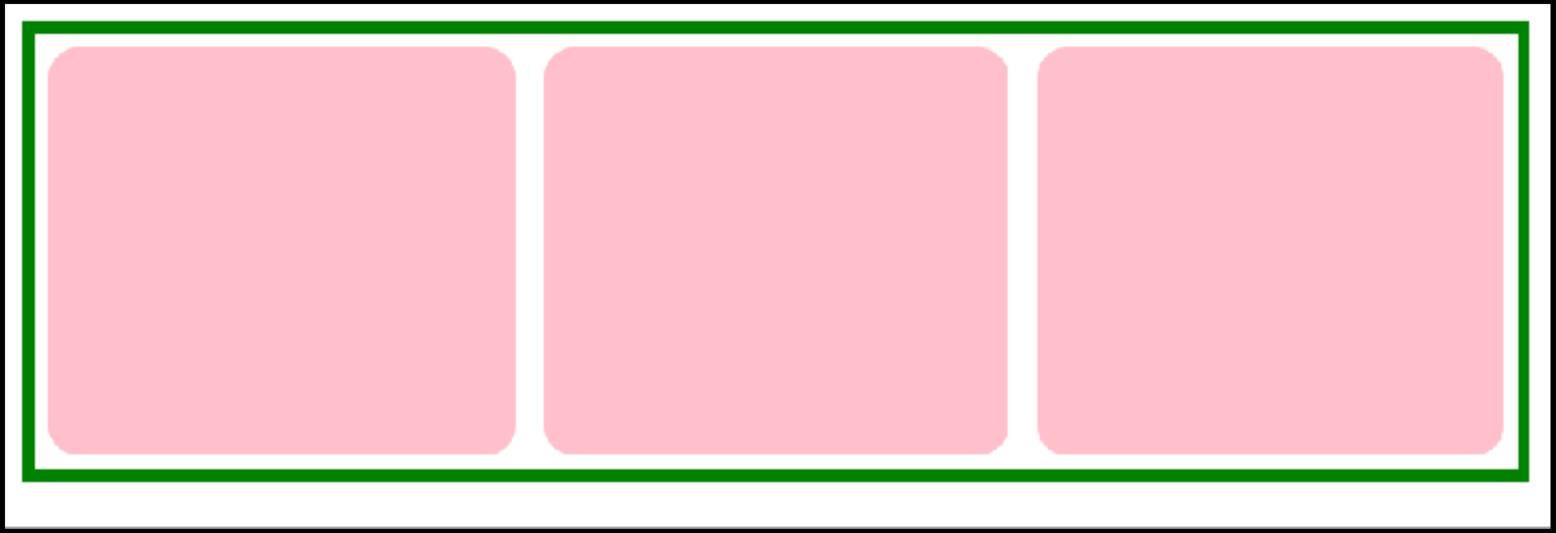
align-items: stretch;

The default value of **align-items** is stretch, which means every flex item grows vertically* to fill the container by default.

(This will not happen if the height on the flex item is set)



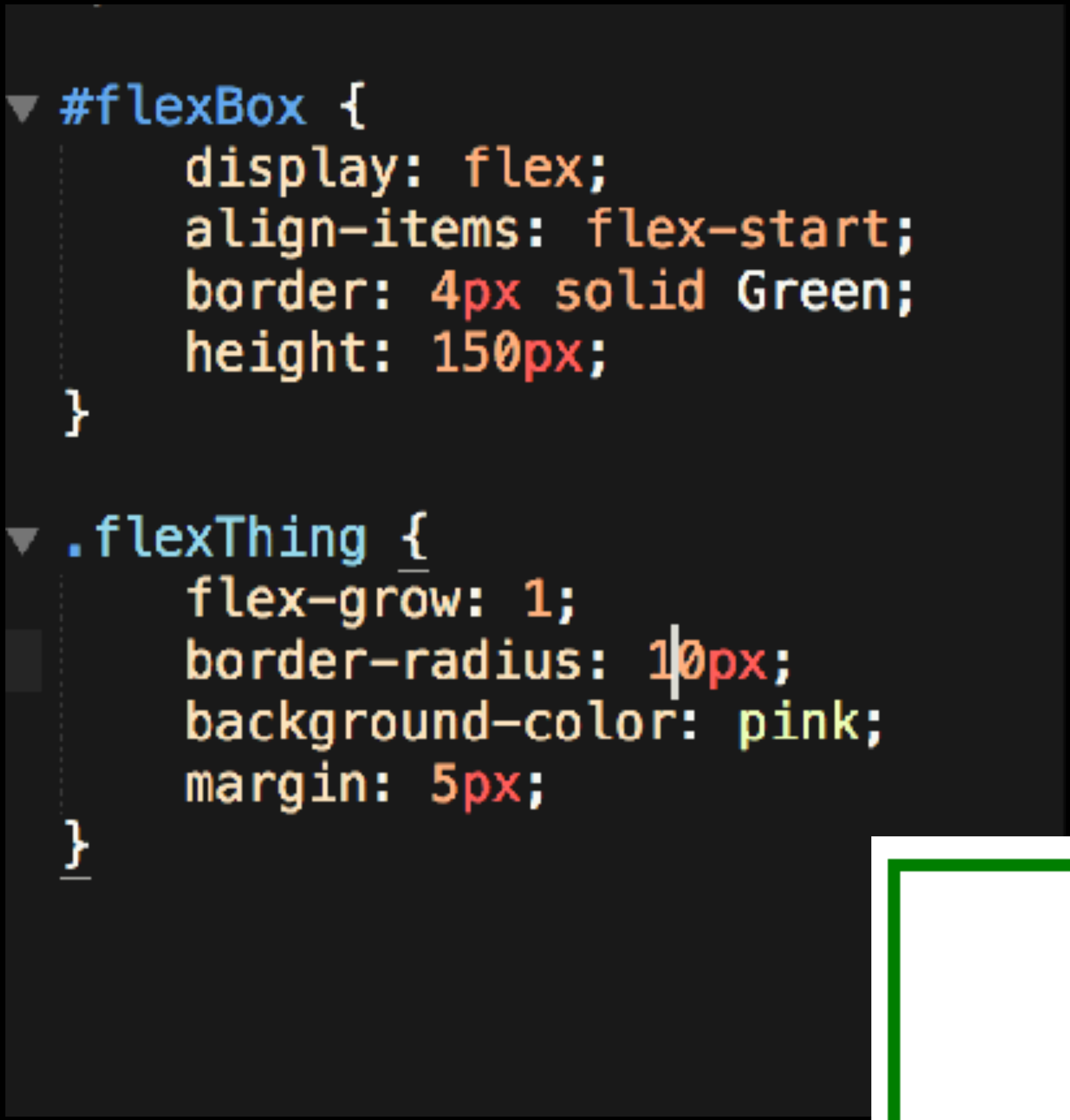
```
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```



*veritcally in the case of rows; horizontally in the case of columns

align-items: stretch;

If we set another value for align-items, the flex items disappear again because the height is now content height, which is 0:



```
#flexBox {
    display: flex;
    align-items: flex-start;
    border: 4px solid Green;
    height: 150px;
}

.flexThing {
    flex-grow: 1;
    border-radius: 10px;
    background-color: pink;
    margin: 5px;
}
```