Let's breakdown the `content` value of this responsive <meta> tag:

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">
```

vh and vw

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

Example:
- height: 100vh; - width: 100vw;

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

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

```css
/* 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%;
}
```

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

rwd
Responsive Web Design

Most of these notes are verbatim txt from: Learning
Web Design - Jennifer Niederst Robbins

Layout

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.

Designers typically start with a one-column layout that fits well on small handheld devices and rearrange elements into columns as more space is available. They may also have the design for the widest screens worked out early on so there is an end-point in mind. The design process may involve a certain amount of switching between views and making decisions about what happens along the way.
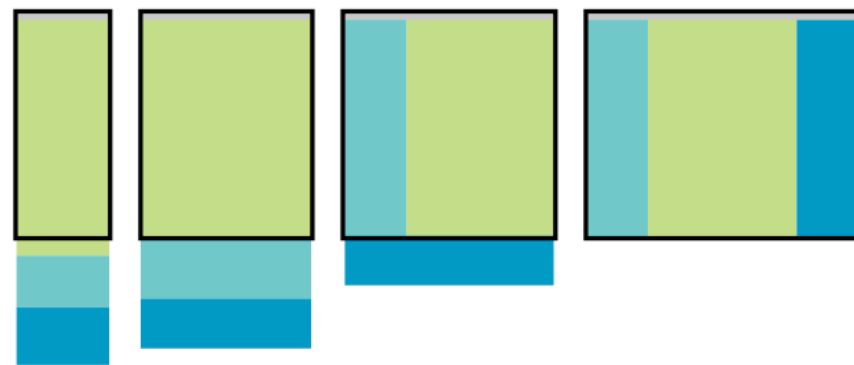
Responsive layout patterns

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**:
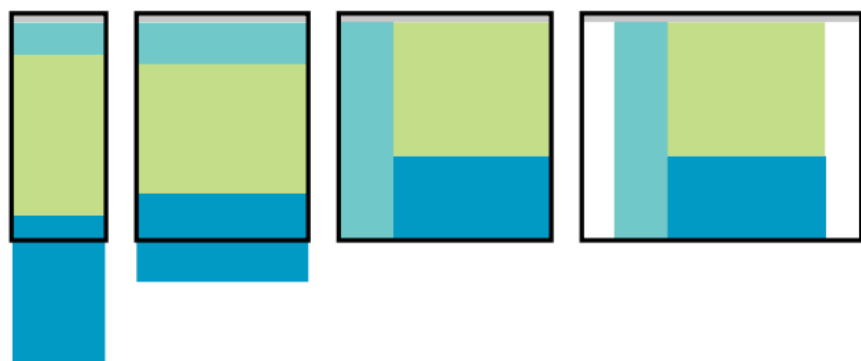
Mostly fluid
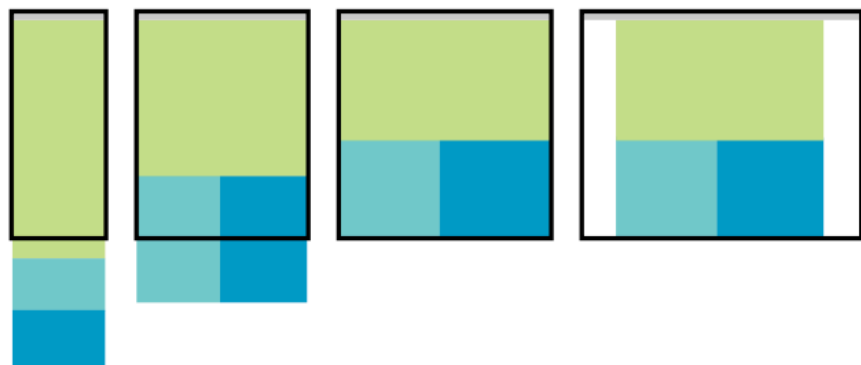
Column drop

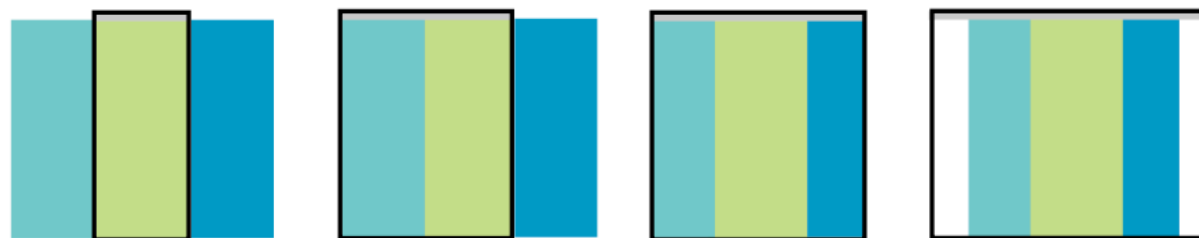**Layout shifter**

Tiny tweaks

Off canvas

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.

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

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.

Display Property

The display property specifies if/how an element is displayed. Every HTML element has a default display value depending on what type of element it is. The default display value for most elements is block or inline. A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can). An inline element does not start on a new line and only takes up as much width as necessary.

    display: none;

commonly used with JavaScript to hide/show elements without deleting and recreating them.

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.

Overflow Property

The CSS overflow property controls what happens to content that is too big to fit into an area. The overflow property specifies whether to clip content or to add scrollbars when the content of an element is too big to fit in a specified area. The overflow property only works for block elements with a specified height. The overflow property has the following values:

visible - Default. The overflow is not clipped. It renders outside the element's box
hidden - The overflow is clipped, and the rest of the content will be invisible
scroll - The overflow is clipped, but a scrollbar is added to see the rest of the content
auto - If overflow is clipped, a scrollbar should be added to see the rest of the content

<u>Properties</u> for left and right

overflow-x
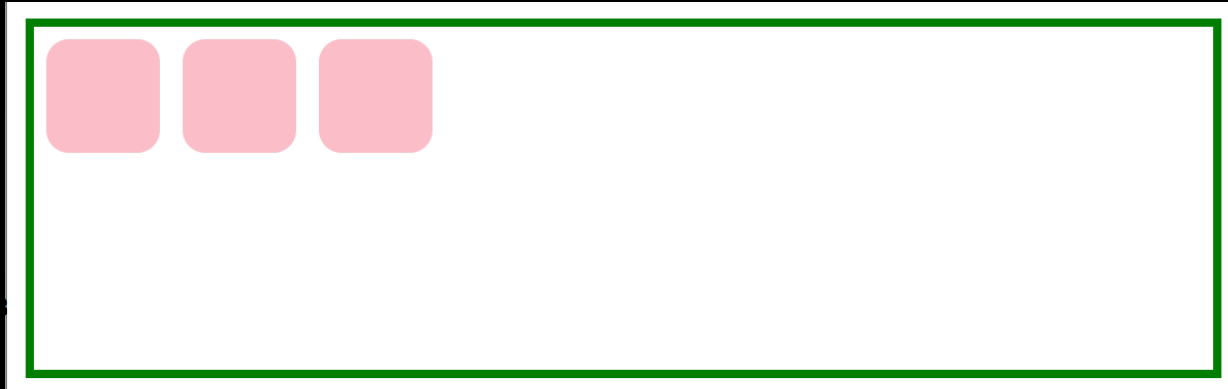   specifies what to do with the left/right edges of the content.
overflow-y
    specifies what to do with the top/bottom edges of the content.

```css
div.theExample1 {
    background-color: lightblue;
    height: 40px;
    width: 200px;
    overflow-y: scroll;
}

div.theExample2 {
    background-color: lightblue;
    height: 40px;
    width: 200px;
    overflow-y: hidden;
}
```
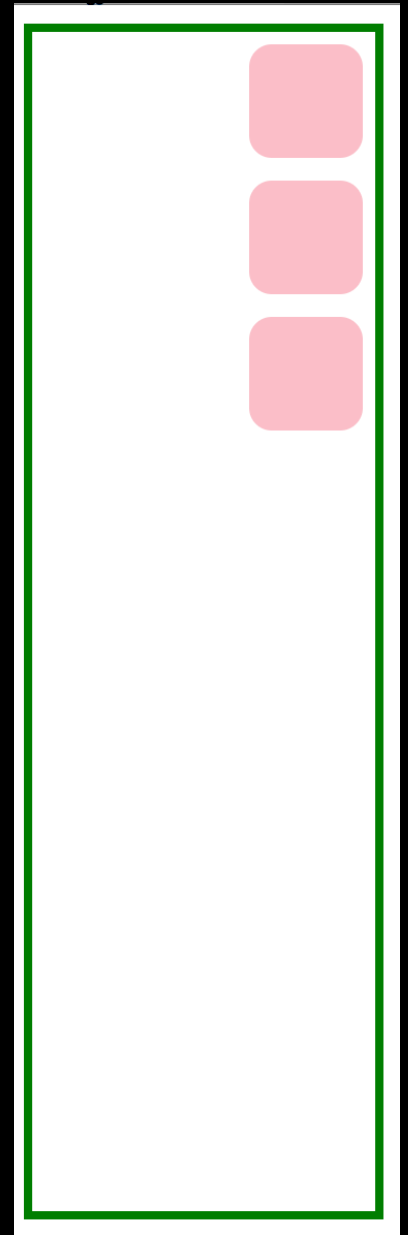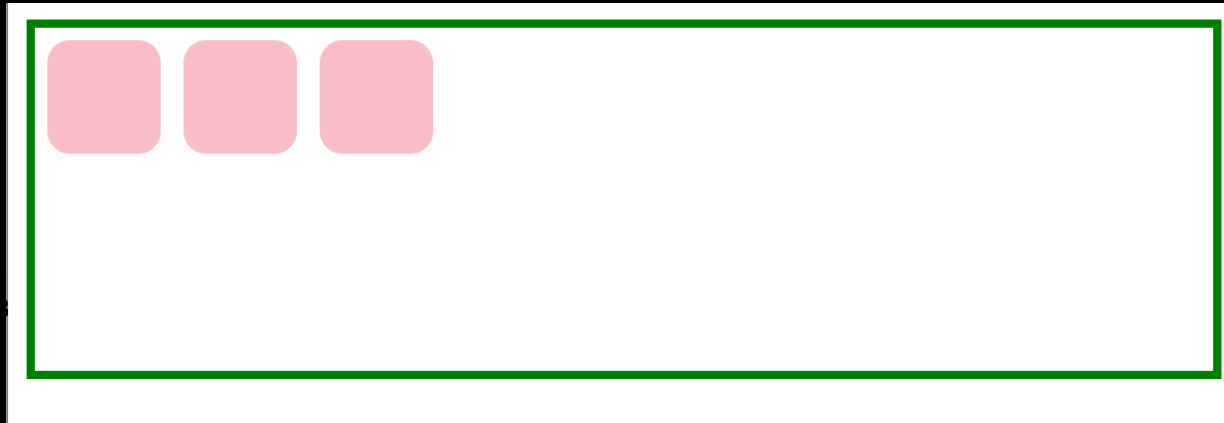
<u>code from W3</u>

Flex Layout

To achieve more complicated layouts, we can enable a different kind of CSS layout rendering mode: **Flex layout.**

Flex layout defines a special set of rules for laying out items in rows or columns.
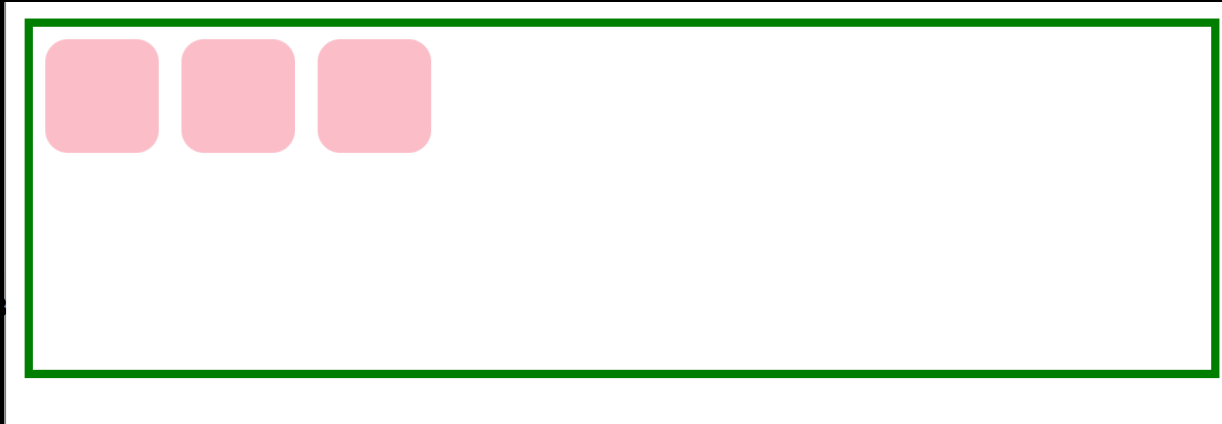
Flex Basics

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;
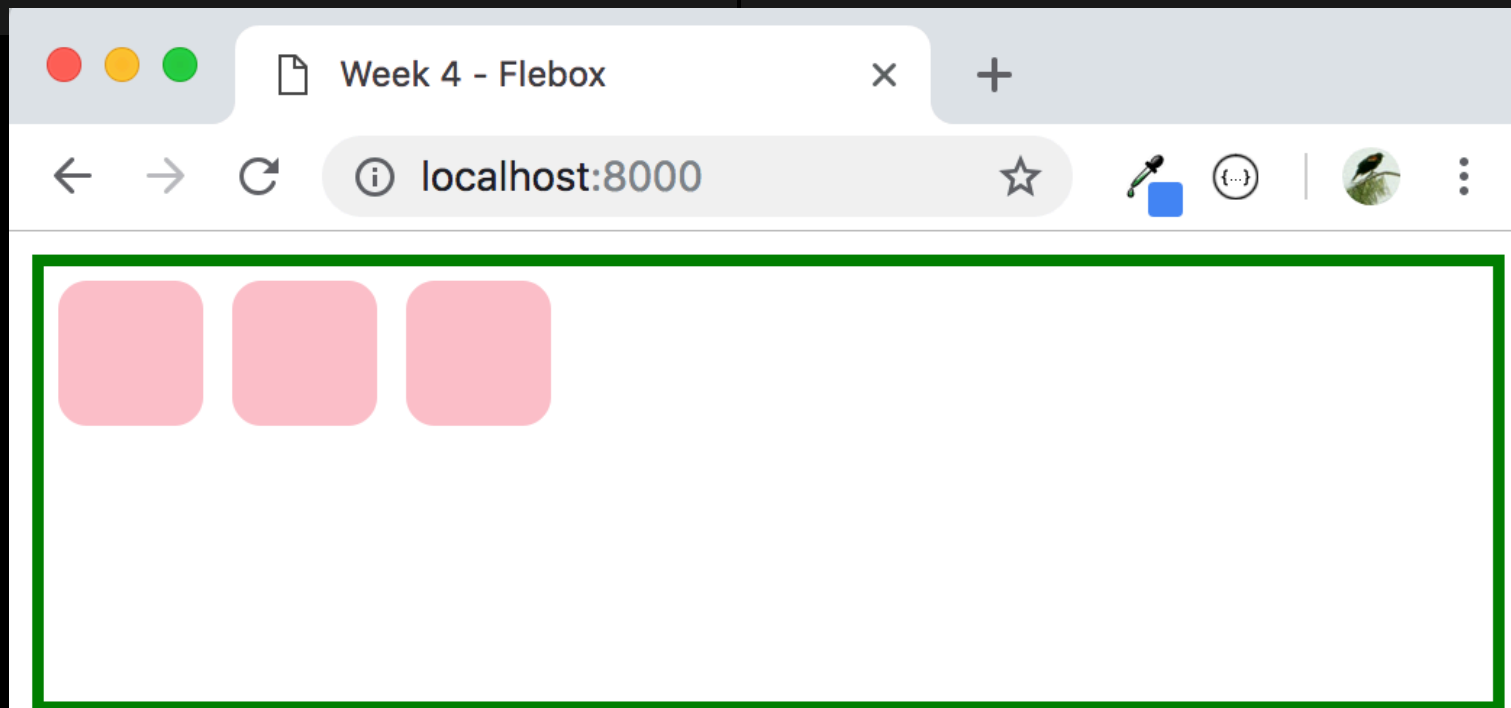 - Inline container: display: inline-flex;

Flex Basics

```html
</head>

<body>

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

   </div>
</body>
```

```css
/* inline flex  -  lecture 4/ Slide

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

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


Week 4 - Flebox

localhost:8000

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;
}
```
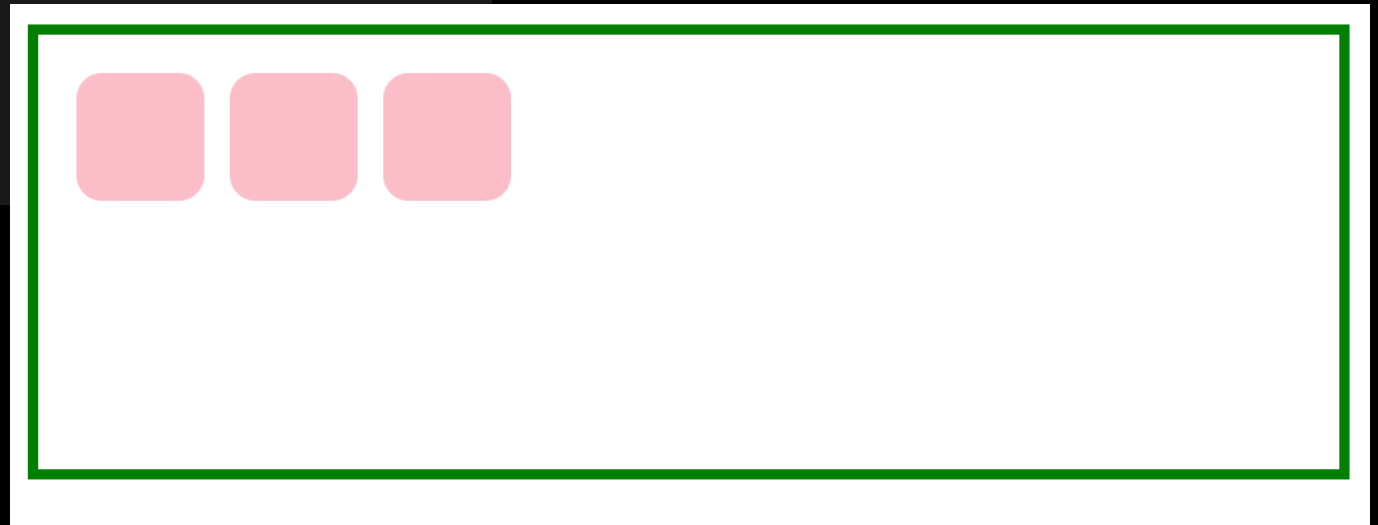
Flex Basics: justify-content

You can control where the item is horizontally in the box by setting **justify-content** in the flex container.
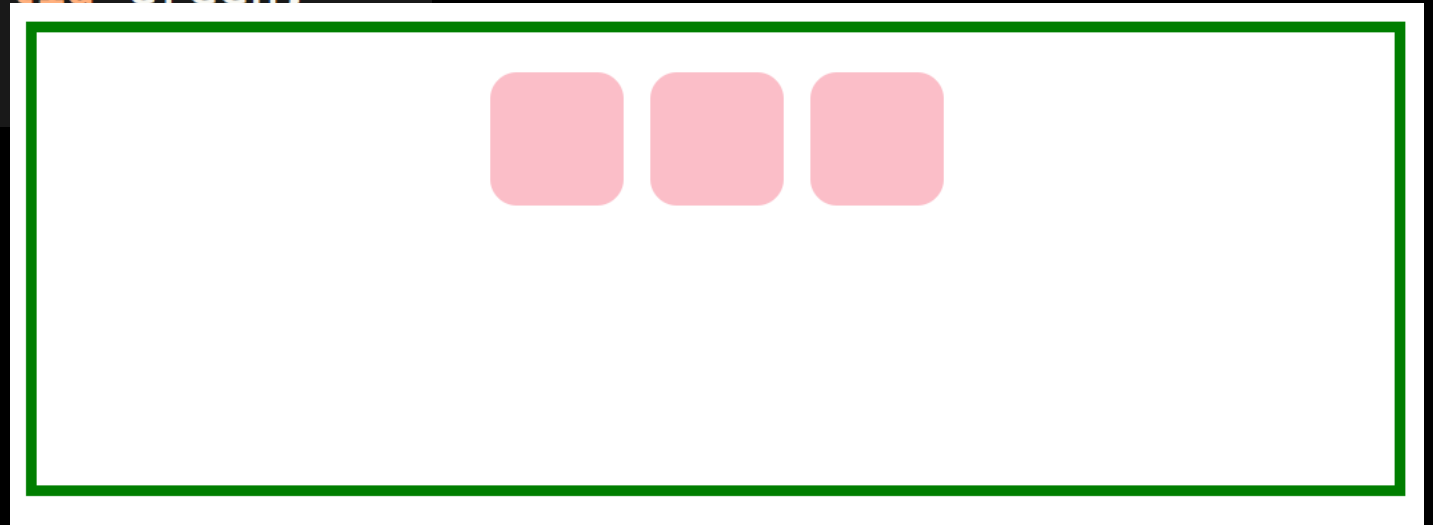
```css
#flexBox {
    display: flex;
    justify-content: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

Flex Basics: justify-content

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

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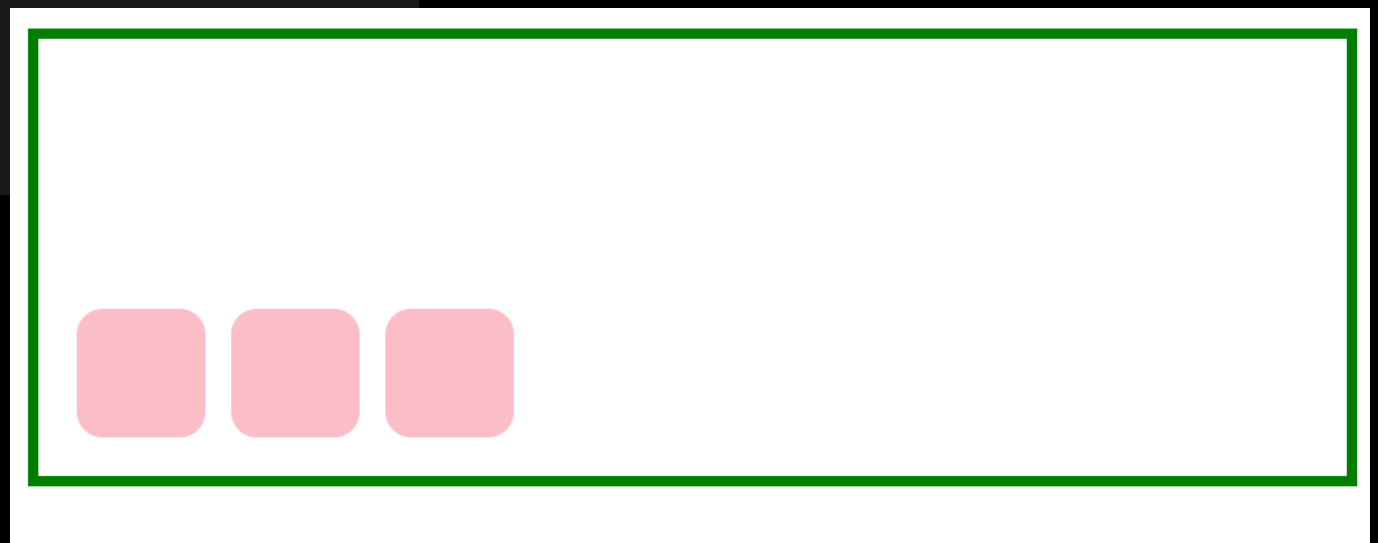
```
#flexBox {
    display: flex;
    align-items: flex-start;
    padding: 10px;
    height: 150px;
    border: 4px solid Green;
}
```

Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.
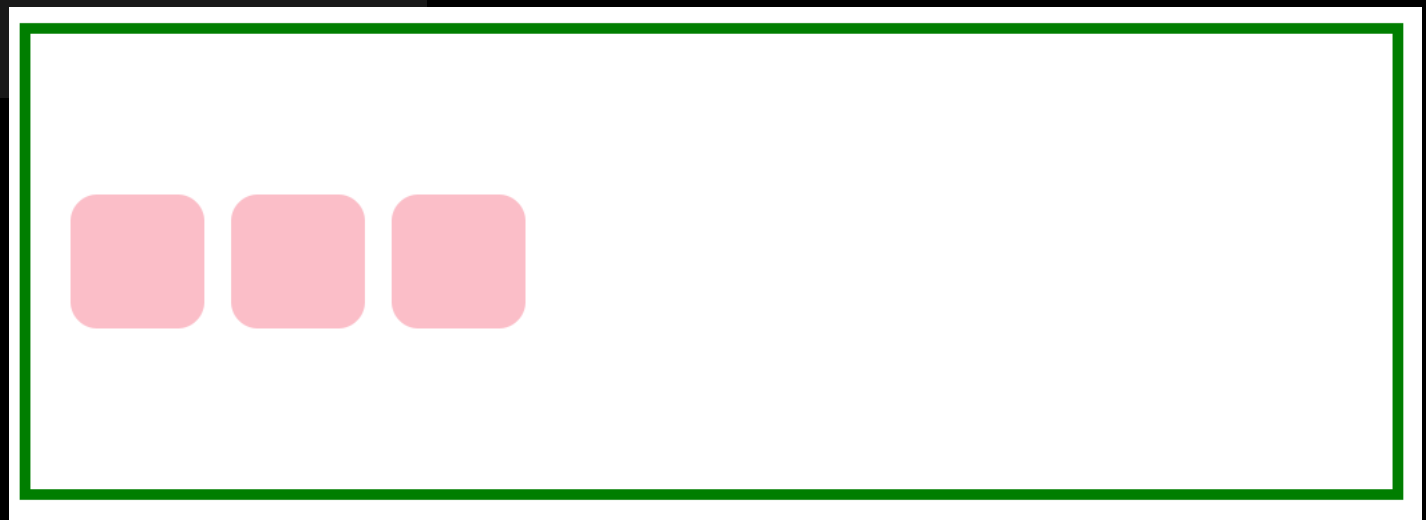
```
#flexBox {
    display: flex;
    align-items: flex-end;
    padding: 10px;
    height: 150px;
    border: 4px solid
}
```

Flex Basics: align-items

You can control where the item is vertically in the box by setting **align-items** in the flex container.
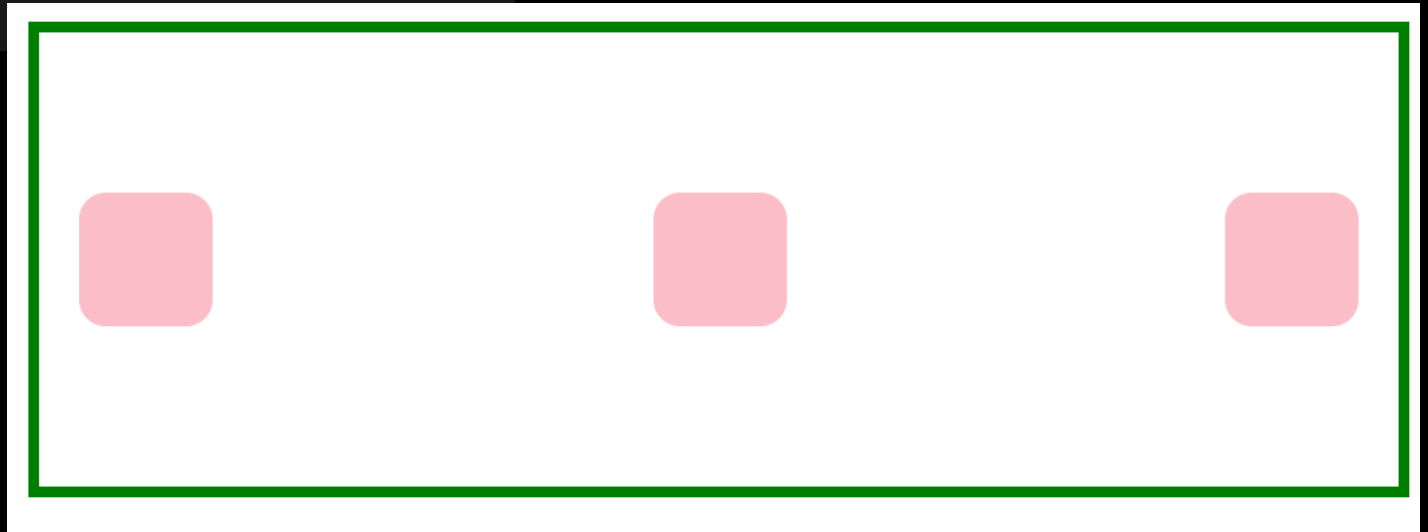
```
▼ #flexBox {
      display: flex;
      align-items: center;
      padding: 10px;
      height: 150px;
      border: 4px solid Green;
  }
```
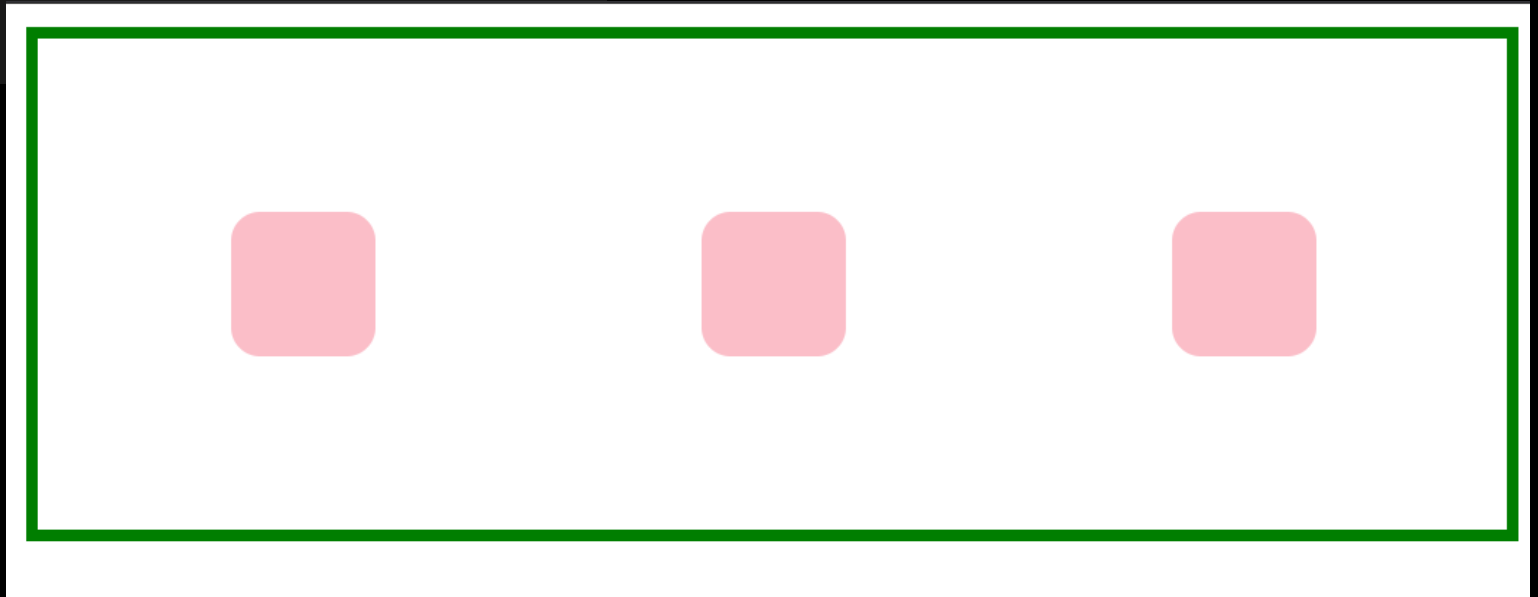
Flex Basics:

space-between + space-around

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

Flex Basics:

space-between + space-around

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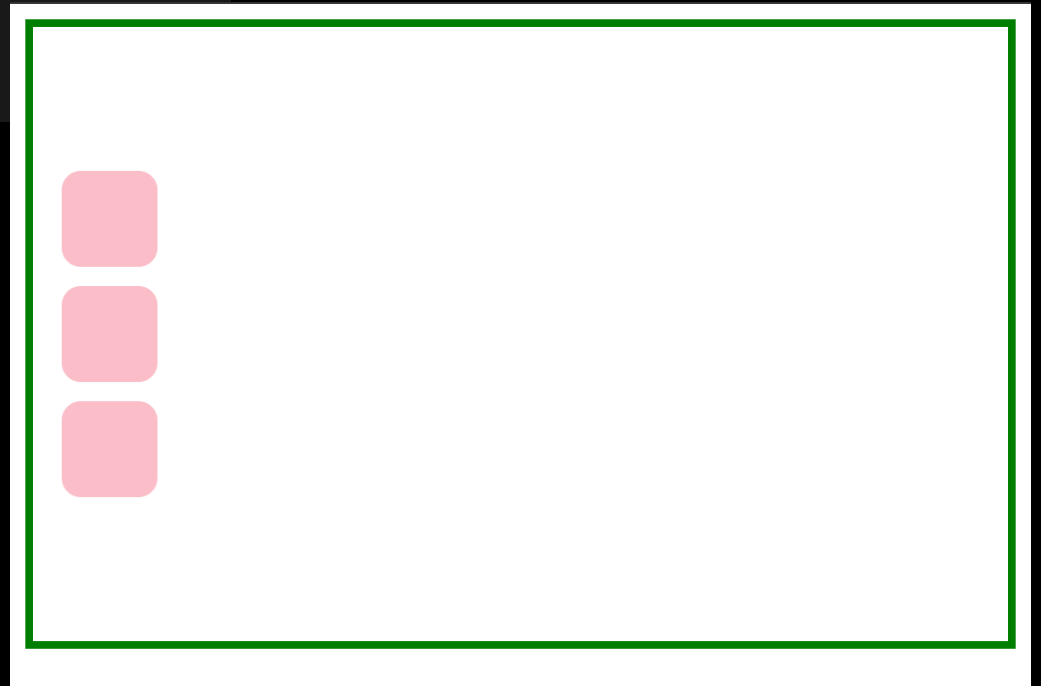
Flex Basics: flex-direction

```css
#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.
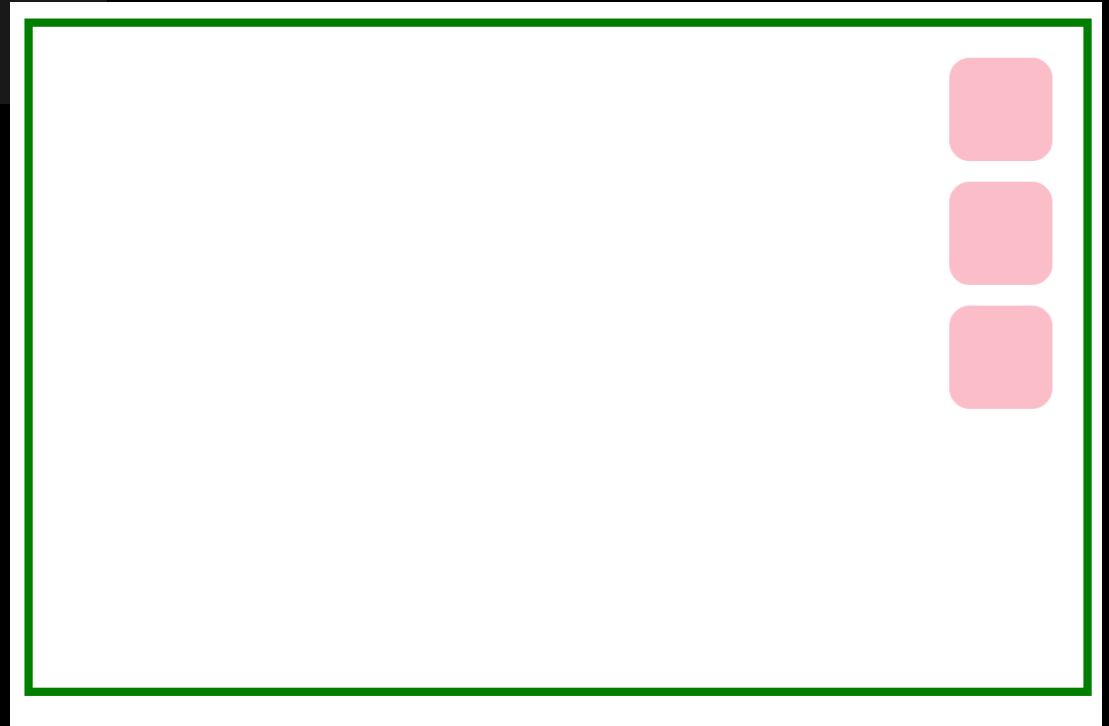
Flex Basics: flex-direction

```
▼ #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 - 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 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;
}
```

← → C ⓘ localhost:8000

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
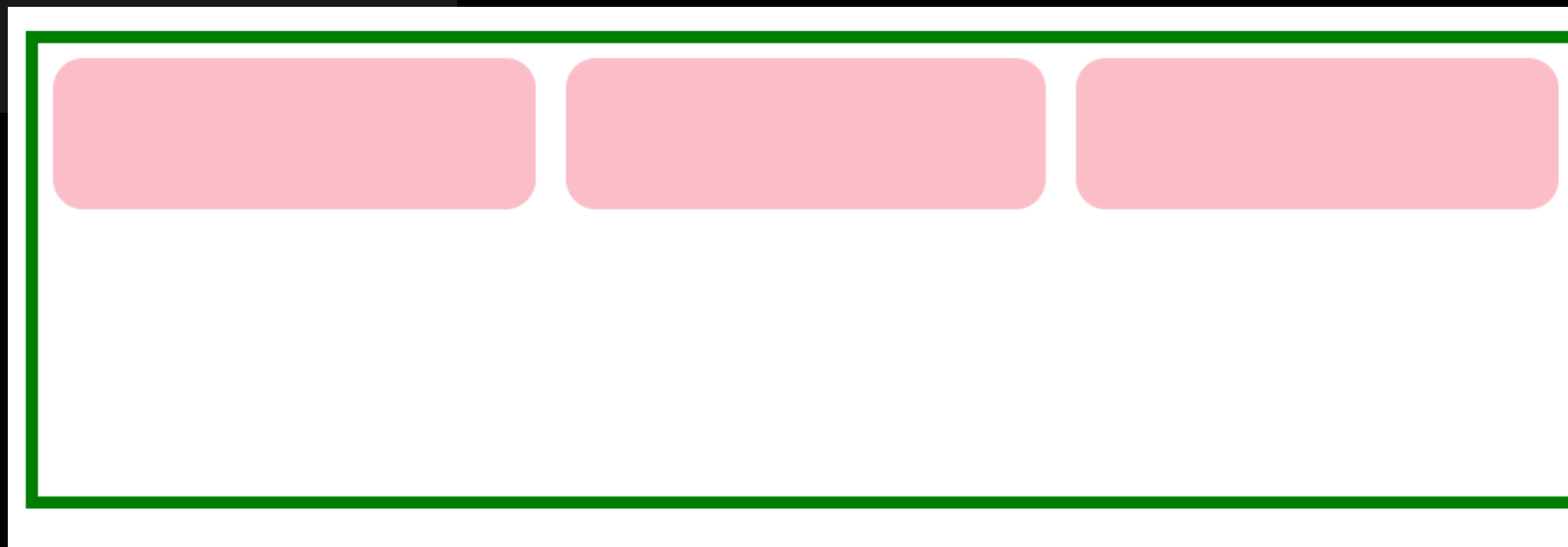- If set to **0**, the flex item does not shrink.

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

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

flex-shrink

```css
#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;
}
```
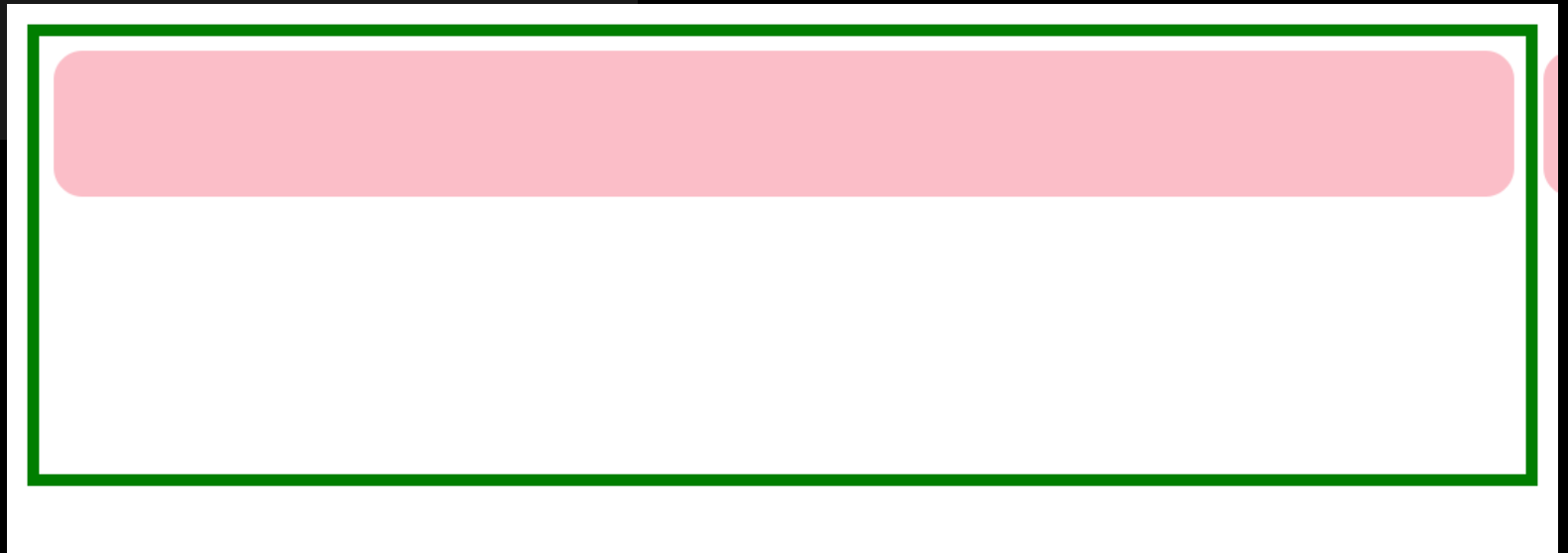
The flex items' widths all shrink to fit the width of the container.

flex-shrink

```css
.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.

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

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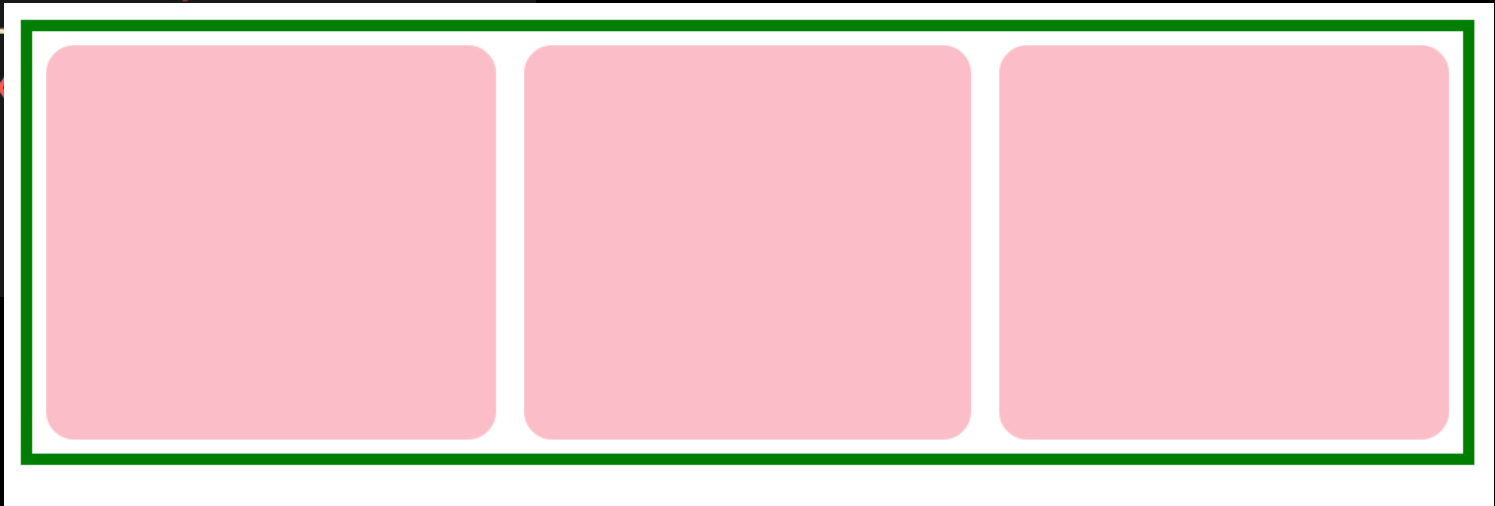
```css
#flexBox {
    display: flex;
    border: 4px solid Green;
    height: 150px;
}

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

flex-grow

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

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

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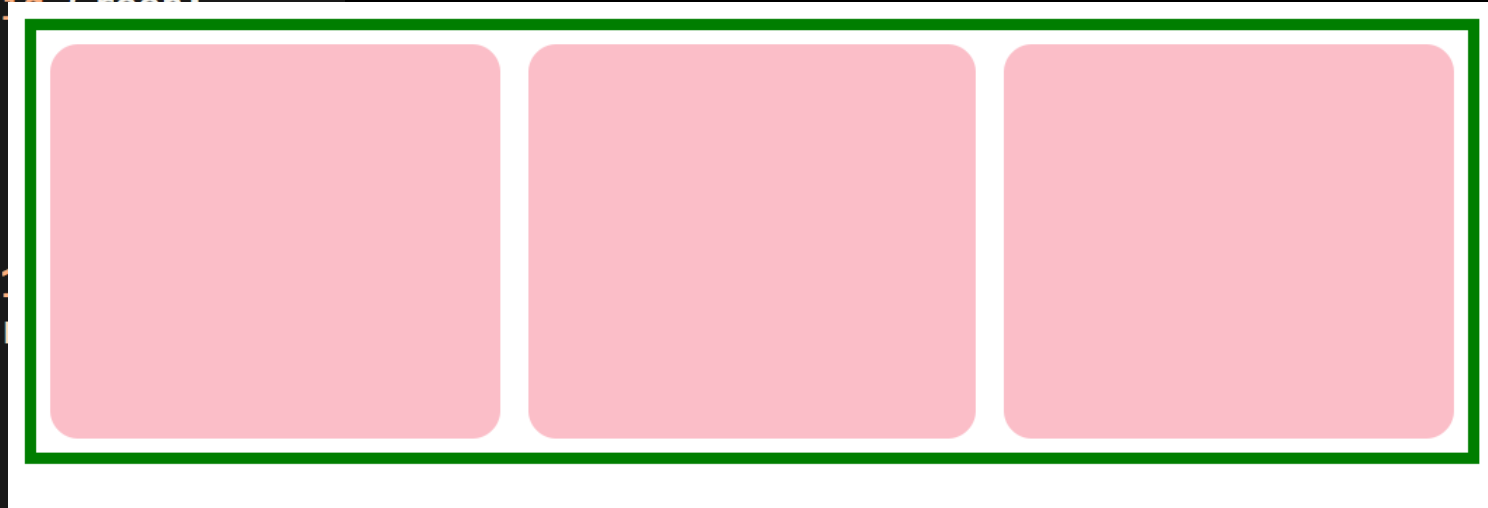
flex item height**?

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

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

```css
#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
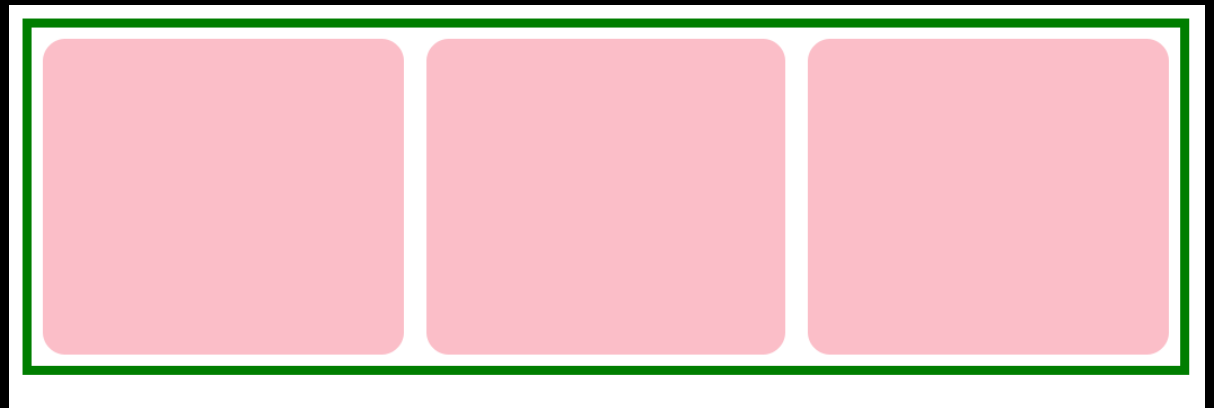**height in the case of rows; width in the case of columns

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)

```css
#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:

```css
#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;
}
```