# ISTA 230

**CSS–Generated Elements**

# CSS

## Pseudo-Elements

As previously discussed, HTML should be used for providing structure, organization, and meaning. Conversely, CSS should be used for modifying the display/presentation of the page (i.e., make it look nice).

CSS pseudo-elements allow us to inject content into our HTML using the :before and :after pseudo selectors. Using pseudo-elements, we can add an inline element before/after an element's content. The generated content is a child of the element, **but is not actually part of the HTML!!!** Because of this, many screen readers and other accessibility devices will ignore the content generated. Because of this, you should only use CSS-generated content for display-related content only (i.e., content that enhances your page but is not vital to understanding the content).

# CSS

## Pseudo-Elements

To create CSS-generated content, you'll need to create a CSS selector that includes the desired pseudo-element. Additionally, you must specify a 'content' property.

### CSS

```
div:before
{
   content: contentGoesHere;
}
```

The 'content' property can contain a string of text, a link to an external file, element attributes, or counters. We'll explore each of these on the following slide.

It is important to note that CSS-generated content does not work with what the CSS specification refers to as 'replaced elements'. This includes the following elements:

- <img>
- <object>
- <input>
- <textarea>

# CSS

## Generated Elements - Text

Using CSS pseudo-elements, we can insert additional text into our web pages. Consider the following  element:

### HTML

```
<div>
   This is my content.
</div>
```

### Browser

**This is my content.**

Using our CSS pseudo-element selectors, we can add content before or after the text, "This is my content."

### CSS

```
div:before
{
   content: "Content: ";


}
```

### Browser

**Content: This is my content.**

As you can see, the text "Content: " was added *before* the text specified in the HTML ("This is my content.") Alternatively, we could have added our dynamically generated content after the text in the HTML by using ":after" instead of ":before".

# CSS

## Generated Elements - Text

In addition to specifying the content for our pseudo-elements, we can also apply all other CSS properties that would apply to an inline element (such a span).

### CSS

```
div:after
{
   content: "... some of which is CSS-generated!";
   font-weight:bold;
   background-color:yellow;
}
```

### Browser

**This is my content.... some of which is CSS-generated!**

# CSS

## white-space Property

CSS

```
pre { white-space: normal; }
```

```
div { white-space: pre; }
```

```
/*
 * Specifies how white-space should be displayed
 * within a given element. Options are:
 * normal, nowrap, pre, pre-line, and pre-wrap.
 *
 *
 * Default Value: normal
 * Inherited: Yes
 *
 */
```

# CSS

## Generated Elements - Text

In much the same way, we can insert HTML elements into our dynamically generated content. Up until this point, we've relied on the HTML entity name when adding them to our content. For example, we would use '♥' to display a small heart character. However, we could use the **HTML entity number** instead of the name if we so desired. For the heart character, the HTML number would be '9829'.

### HTML

```
<p>
   I &hearts; NY<br />
   I &#9829; NY
</p>
```

### Browser

I ♥ NY
I ♥ NY

To add an HTML element to our CSS-generated content, we'll need to use the **hexidecimal** version of the HTML entity number. For our heart character, we know that $9829_{10}$ is equal to $2665_{16}$. Thus, we would use the following to add our HTML heart entity to our CSS-generated content.

### CSS

```
div:after
{
   content: "I \2665 NY";
}
```

### Browser

This is my content. I ♥NY

It's worth noting that in the above example, the space after our heart character is missing in the final display. To get around this, we can take one of two approaches:

### CSS

```
div:after
{
   content: "I \2665" " NY"; /*** Break our content into two strings ***/
}
```

### CSS

```
div:after
{
   content: "I \2665 \0020 NY"; /*** OR use the hexidecimal ASCII code for a blank space ***/
}
```

# CSS

## Generated Elements - External Files

We can also use CSS-generated content to add external files to our page. For example, some sites add a small icon to any links on their page that will take the user to another site. We can do this using pseudo-elements by specifying a URL for our content property.

### HTML

```
<a class="external" href="http://google.com">http://google.com</a>
```

### CSS

```
a.external:after
{
   content: url("../images/external.png");
}
```

### Browser

http://google.com 

Note: While the above example is valid, it is not the most common approach to implementing an 'external link' icon. Instead, most designers use a combination of padding and a background image to achieve the same effect. While the difference is not noticed by most users, the padding/background-image approach is a better choice since it is better supported and provides additional benefits when we move to CSS 3.

# CSS

## Generated Elements - Attributes

In addition to adding text to our content via CSS, we can also use pseudo-elements to display the value of element attributes. For example, we could dynamically add the value of a hyperlink's href attribute after it.

### HTML

```
<a href="http://google.com">Google</a>
```

### CSS

```
a:after
{
   content: attr(href); /* Notice, no quotes! */
}
```

### Browser

Googlehttp://google.com

We could combine this approach with our previous examples to provide a more readable example:

### CSS

```
a:after
{
   /* We can combine content pieces */
   content: " [" attr(href) "] ";
   font-style:italic;
   color:#aaa;
}
```

### Browser

Google *[http://google.com]*

# CSS

## Generated Elements - Counters

CSS counters are another option for CSS-generated content. Counters can be used to store a value and can be incremented or reset based on the number of times they're used in your page. To use counters, we'll have to rely on two additional CSS properites:

- counter-reset
- counter-increment

To use a counter, you first must 'reset' it to a value (0 by default). Once we have done so, we can use the counter in our content property. Additionally, we can increase the value of our counter by using the counter-increment property. By default, counters are incremented by 1, though we can specify any integer value if we'd like.

If we wanted to, we could use counters to replicate the mechanism used by browsers to provide numeric ordered lists (<ol>).

### HTML

```
<ul>
   <li>Item</li>
   <li>Item</li>
   <li>Item</li>
</ul>
```

### CSS

```
ul { list-style:none; counter-reset: listCounter; }/*** 'Reset' the counter named 'listCounter' to 0.
***/
li { counter-increment: listCounter; } /*** We increment the counter named 'listCounter' by 1 for
every <li> element. ***/
li:before { content: counter(listCounter) ": "; }/*** We need to use a pseudo-element to actually
display the counter value. ***/
```

### Browser

1: Item 1
2: Item 2
3: Item 3

# CSS

## Generated Elements - Counters

By providing a value to our counter-reset property, we can specify what value our counter should be reset to. It is worth noting that counters are incremented **before** they are displayed in our pseudo-elements. That means that if we reset our counter to 5, the first list item will show '6' as the counter value.

### HTML

```
<ul>
   <li>Item</li>
   <li>Item</li>
   <li>Item</li>
</ul>
```

### CSS

```
ul { list-style:none; counter-reset: listCounter 5; }
li { counter-increment: listCounter; }
li:before { content: counter(listCounter) ": "; }
```

### Browser

**6: Item 1**
**7: Item 2**
**8: Item 3**

# CSS

## Generated Elements - Counters

Similarly, we could increment our counter by more (or less) than 1. For example, we could increment by 2, resulting in an even-numbered list of items.

### HTML

```
<ul>
   <li>Item</li>
   <li>Item</li>
   <li>Item</li>
</ul>
```

### CSS

```
ul { list-style:none; counter-reset: listCounter; }
li { counter-increment: listCounter 2; }
li:before { content: counter(listCounter) ": "; }
```

### Browser

2: Item 1
4: Item 2
6: Item 3

# CSS

## Generated Elements - Counters

Lastly, it is worth noting that we can pass an additional argument to the 'counter' function, specifying the style of the counter. This can be any of the styles allowed for the 'list-style-type' property.

### HTML

```
<ul>
   <li>Item</li>
   <li>Item</li>
   <li>Item</li>
</ul>
```

### CSS

```
ul { list-style:none; counter-reset: listCounter; }
li { counter-increment: listCounter 2; }
li:before { content: counter(listCounter, upper-roman) ": "; }
```

### Browser

II: Item 1
IV: Item 2
VI: Item 3

# CSS

## Generated Elements - Counters

Using CSS counters isn't limited to HTML lists. For instance, imagine you're publishing a page containing the contents of a short book on HTML. The book, while short, does include a number of chapters and sections. You've already created the markup using the appropriate heading tags. However, you'd like to number each chapter/section without creating a maintenance nightmare in the future.

### HTML

```
<h1>Introduction</h1>
   <h2>Conventions</h2>
   <h2>Source Code</h2>
   <h2>Errata</h2>
<h1>HTML</h1>
   <h2>Basic Text Formating</h2>
   <h2>Presentational Elements</h2>
   <h2>Phrase Elements</h2>
```

Ultimately, you'd like to have each chapter numbered, as well as each section underneath that chapter.

# CSS

## Generated Elements - Counters

This task can be completed easily by using two separate counters. The first counter, named 'chapter', is used for numbering our chapters. It is 'reset' to 0 on the body element. For each <h1>, we'll increment the 'chapter' counter by 1.

We'll also create a counter named 'section'. This counter will be 'reset' to 0 every time we come across an <h1> (i.e., a new chapter). We'll increment the 'section' counter by 1 for each <h2>.

CSS

```
body {
   counter-reset: chapter;
}
h1 {
   counter-increment: chapter;
   counter-reset: section;
}
h2 {
   counter-increment: section;
}
h1:before {
   content: "Chapter " counter(chapter) ": ";
}
h2:before {
   content: "Section " counter(chapter) "." counter(section) ": ";
}
```

The results can be seen on the next slide.

## CSS

### Generated Elements - Counters

#### Browser

**Chapter 1: Introduction**
- **Section 1.1: Conventions**
- **Section 1.2: Source Code**
- **Section 1.3: Errata**

**Chapter 2: HTML**
- **Section 2.1: Basic Text Formating**
- **Section 2.2: Presentational Elements**
- **Section 2.3: Phrase Elements**

While the output is exactly what we want, the practice of introducing multiple counters to accomplish this task doesn't scale well. For example, imagine if we had subsections as well, or even greater levels of heirarchy... Fortunately, there is a better option (see next slide).

# CSS

## Generated Elements - Counters

When we use the counter-reset property, a new counter is created. In the case of nested lists, a new counter (using the same counter name) is actually created for each nested list. CSS provides us with a means to take advantage of this fact!

In addition to the 'counter' function, we can also use the 'counter**s**' function. The "counters" function allows us to use the nested counter as well as it's ancestors to create a heirarchy. Let's again look at the case of our online book. With multiple chapters, sections, and subsections, maintaining multiple counters simply isn't a great option.

### HTML

```
<ol>
   <li>Chapter</li>
   <li>Chapter
      <ol>
         <li>Section</li>
         <li>Section</li>
         <li>Section
            <ol>
               <li>Subsection</li>
               <li>Subsection</li>
            </ol>
        </li>
      </ol>
   </li>
   <li>Chapter</li>
</ol>
```

# CSS

## Generated Elements - Counters

Instead of using multiple counters, we'll use a single counter for our chapters, sections, and subsections. However, instead of using the "counter" function, we'll use the "counters" function, providing it with the name of our counter as well as the string that should be used to separate each value.

### CSS

```
ol { list-style:none; counter-reset: listCounter; }
li { counter-increment: listCounter; }
li:before { content: counters(listCounter,".") ": "; }
```

Because we added a counter-reset property to our <ol> elements, a new counter is actually created for each of our nested lists. The value for each counter is then concatenated with "." to provide our CSS-generated content (see output on next slide).

## CSS

### Generated Elements - Counters

#### Browser

1: Chapter

2: Chapter

2.1: Section

2.2: Section

2.3: Section

2.3.1: Subsection

2.3.2: Subsection

3: Chapter

# CSS

## Floating Issues

As previously discussed, floating an element that is the only child of it's parent will cause the parent element to collapse. During our lecture on the box model, we found that setting the parent element's overflow property to 'auto' would typically fix this issue. However, that technique isn't full-proof and sometimes leads to small scrollbars being displayed on the parent element!

During that same lecture, we also talked about the option of adding a <br /> element and setting it's clear property to 'both'. We didn't pursue this approach since it required us to add additional HTML elements to our page, purely for display purposes.

### HTML

```
<div>
   <img src="cats.jpg" alt="CATS!" />
   <br style="clear:both" /> <!-- Not great... -->
</div>
```

### Browser

That said, the approach of using a <br /> was tempting, as it worked across browsers and didn't have the issue of scrollbars that came with our 'overflow:auto' solution. While adding an element in our HTML file isn't a great solution, we can now use CSS-generated content to provide a more robust solution to the float/collapse issue.

# CSS

## CSS Generated Content - Clear Fix

First, we'll use a class called 'clearFix' for our parent element.

### HTML

```
<div class='clearFix'>
   <img src="cats.jpg" alt="CATS!" />
   <!-- <br style="clear:both" /> -->
</div>
```

Next, we'll use our CSS pseudo-element for ':after' to create a new inline (by default) child of the .

### CSS

```
.clearFix:after
{
   content: ' ';
   display: block;
   clear: both;
   visibility: hidden;
   font-size: 0;
   height: 0;
}
```

Note that we still provided a content property, even though there is no content to be displayed. If we were to leave that out, the rule would be ignored completely.

Additionally, we specified that the pseudo-element should display as a block element, that it should be hidden, that it's height should be 0, and that the font-size should be 0. Most of these properties are set to ensure that the pseudo element isn't actually visible to users. Most importantly, we set the clear property to 'both'. This will ensure that even if every element inside of the  is floated to the left or right, the div will still stretch to contain all of them.

### Browser

# CSS

## Review - Rounded Corners

You may remember, during our lecture on background images, that we discussed creating the effect of 'rounded corners'.

In order to do this, we had to use multiple nested  elements.

### HTML

```
<div id="contentTop">
   <div id="contentBottom">
      <div id="content">
         This is some content.
      </div>
   </div>
</div>
```

However, using CSS-generated content, we can now do this with less HTML markup.

# CSS

## Generated Elements - Rounded Corners

First, we'll create a single  element.

### HTML

```
<div id="rounded">
   This is some content.
</div>
```

Next, we'll set some CSS properties for the , giving it a fixed width as well as some padding, borders, and a top margin. Also, note that we set the position property of the element to 'relative'!

### CSS

```
.rounded {
   width:343px;
   padding:10px;
   background-color:white;
   border:5px solid #597CBB;
   margin-top:5%;

   position:relative;
}
```

We can now create two pseudo-elements for the  (:before and :after). Each of these will have a specific background image that will provide two of the rounded corners.

We'll position each of the pseudo-elements absolutely. Because we specified that the parent element was positioned 'relative', it is the containing element for the pseudo elements.

### CSS

```
.rounded:before {
   content:url("../images/topRounded.png");
   position:absolute;
   height:30px;
   top:-30px;
   left:-10px;
}

.rounded:after {
   content:url("../images/bottomRounded.png");
   position:absolute;
   height:66px;
   top:100%;
   left:-10px;
}
```

Using the 'top' and 'left' properties to position the pseudo-elements (with their rounded corner background images) in the correct location, we've given the impression that the  itself has rounded corners. The output can be seen on the next slide.

## CSS

### Generated Elements - Rounded Corners

#### Browser

# CSS

## Generated Elements - Shapes

### CSS

```
border:100px solid #000;
border-color: red green blue orange;

width:0;
height:0;
```

# CSS

## Generated Elements - Shapes

We can also use CSS-generated content to make useful shapes which can be incorporated into our site design. Before we jump into that, let's review a little bit about our box model and, in particular, how borders are rendered.

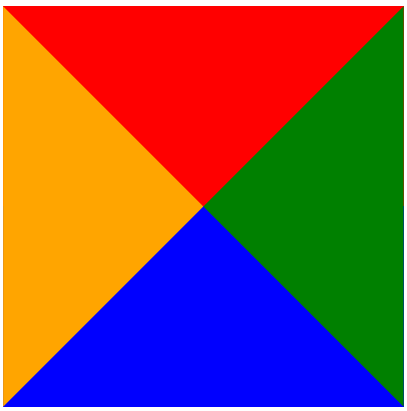
Consider the following: An element is given an explicit height of 0 pixels and an explicit width of 0 pixels. It is effectively non-existent to the viewer. However, if we were to add a 100 pixel border to the element, it's true dimensions would be 200 × 200 pixels. If we applied a different color to each border, we could see exactly how the browser renders each one.

### CSS

```
    border:100px solid #000;
    border-color: red green blue orange;

    width:0;
    height:0;
```

### Browser



When an element is 0 × 0 pixels tall, all four borders are drawn as triangles, their apexes meeting in the middle. If we experiment with the size of our borders, we'll find that we can manipulate how these triangles are drawn. Additionally, we don't have to display all of these borders if we change their color to 'transparent'. This opens up a number of options for us to play with.

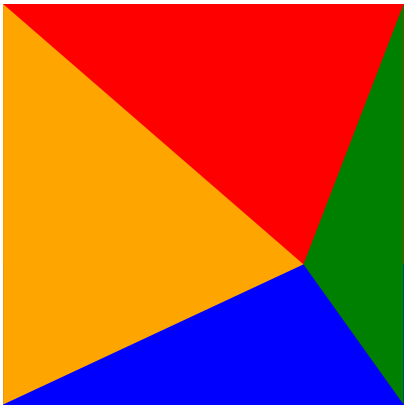
# CSS

## Generated Elements - Shapes

### CSS

```
border:100px solid #000;
border-color: red green blue orange;
border-width: 130px 50px 70px 150px;
width:0;
height:0;
```

### Browser



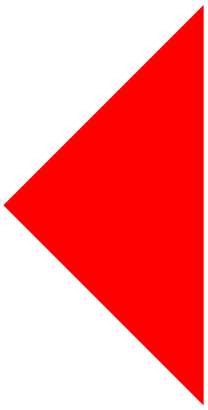
# CSS

## Generated Elements - Shapes

### CSS

```
border:100px solid #000;
border-color: transparent red transparent transparent;
width:0;
height:0;
```

### Browser



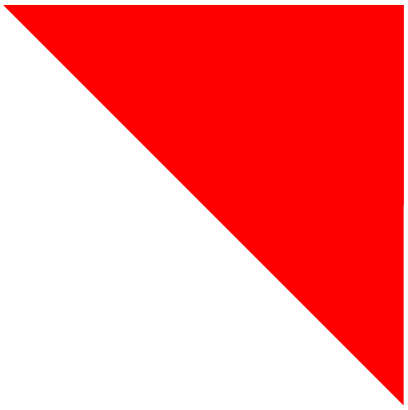
# CSS

## Generated Elements - Shapes

### CSS

```
border:100px solid #000;
border-color: red red transparent transparent;
width:0;
height:0;
```

### Browser



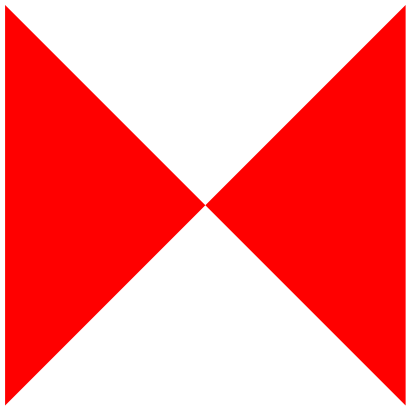
# CSS

## Generated Elements - Shapes

### CSS

```
border:100px solid #000;
border-color: transparent red;
width:0;
height:0;
```

### Browser



## CSS

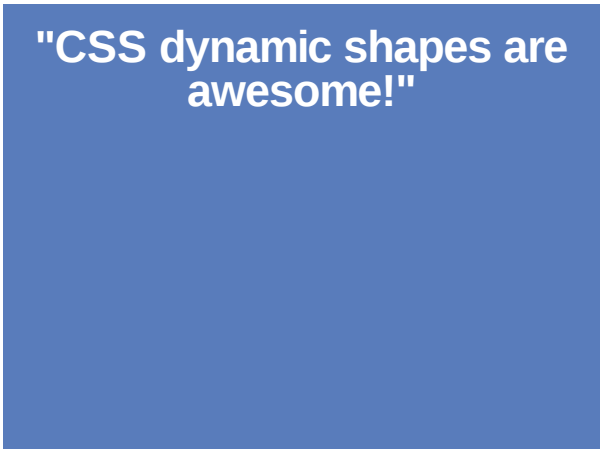
### Generated Elements - Speech Bubble

Consider the following HTML snippet:

**HTML**

```
<div class="speechBubble">
   "CSS dynamic shapes are awesome!"
</div>
```

**Browser**




The output, while perfectly fine, isn't anything spectacular. To add a little flare to it, we'd like to add a small triangle at the bottom of the , making it look more like a "speech bubble" as is often used in comic books and other similar formats.

## CSS

### Generated Elements - Speech Bubble

We can use pseudo-elements to create an additional element which can then be positioned absolutely to give us the effect we're looking for.

#### CSS

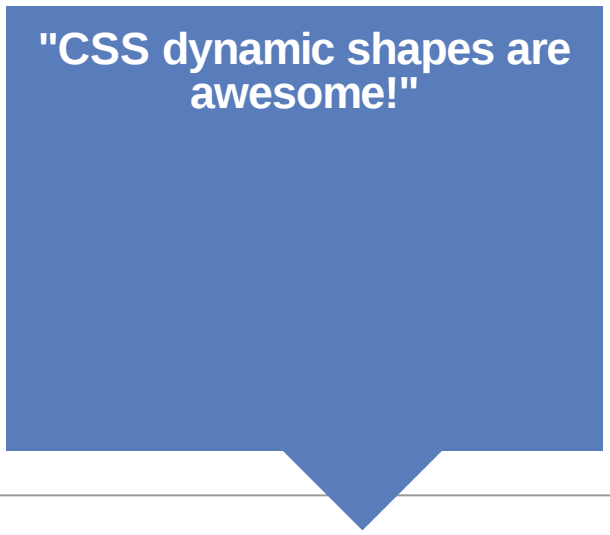
```
.speechBubble
{
   background-color:#597CBB;
   position:relative;
   min-height:20px;
}
```

We'll set our parent element to be positioned relatively, making it the containing element for it's absolutely positioned descendants. For our pseudo-element, we could use a background image to get our small triangle. However, this requires an additional HTTP request to download the image. Additionally, image colors sometimes don't match those rendered by the browser. Instead of using an image, we'll use CSS borders to create our triangle decoration.

#### CSS

```
.speechBubble:before
{
   content: "";
   position:absolute;
   right:20px;
   top:100%;
   width:0;
   height:0;
   border:20px solid #597CBB;
   border-color: #597CBB transparent transparent transparent;
}
```

#### Browser



# CSS

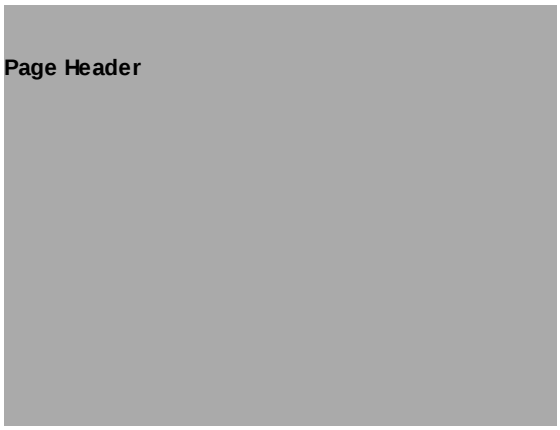
## Generated Elements - CSS Ribbon

Next, let's consider the following HTML snippet:

### HTML

```
<div class="ribbon">
   <h1>Page Header</h1>
</div>
```

### Browser




We have a page heading that is pretty plain. We'd like to spruce it up by making it look like a ribbon wrapping around the parent div.

# CSS

## Generated Elements - CSS Ribbon

We'll start by giving the heading a background color, a minimum height, and positioning it relatively (so it is the containing element of any absolutely positioned descendants). Lastly, we'll give the heading a top and bottom margin of 2% and a left and right margin of **negative** 30 pixels!

### CSS

```
.ribbon h1
{
   background-color:#597CBB;

   position:relative;
   min-height:30px;
   margin:2% -30px;
}
```

Next, we'll create our pseudo-elements. The goal is to create two small triangles that will sit just below our header, filling in the 30 pixel spaces on the left and right. To create these triangles, we'll once again use CSS borders. Because both elements share the same border style and color, we'll specify those here. We'll also position both absolutely, 100% down from the top of the containing element (the <h1>).

### CSS

```
.ribbon h1:before,
.ribbon h1:after
{
   content: "";
   width:0;
   height:0;
   border-style:solid;
   border-color: transparent #142E5B;
   position:absolute;
   top:100%;
}
```

Lastly, we'll create our two triangles and position them to the left and right respectively.

### CSS

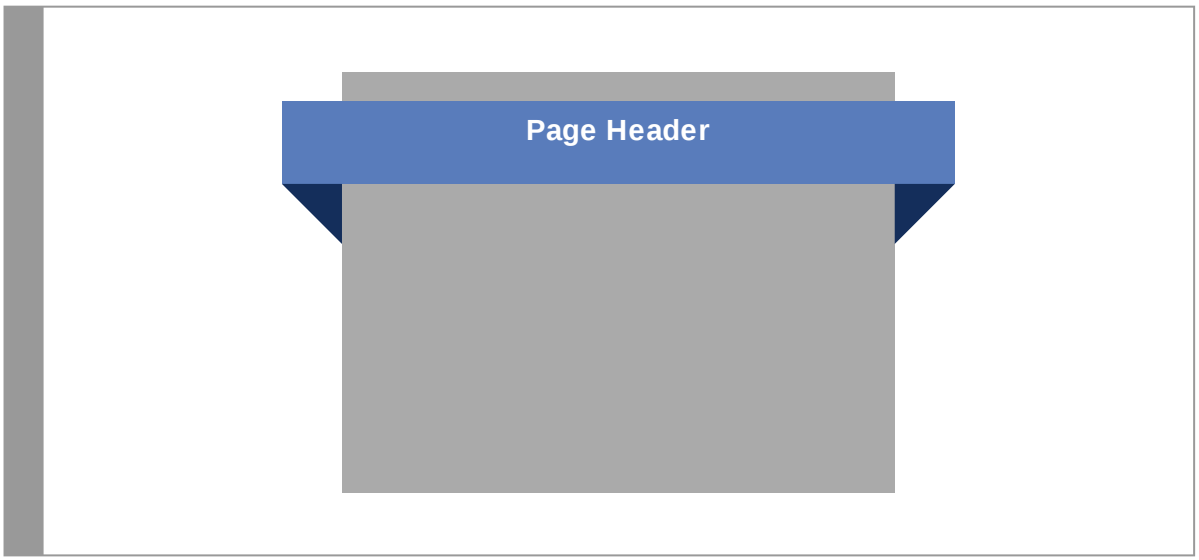
```
.ribbon h1:before
{
   right:0;
   border-width: 0 0 30px 30px;
}

div.ribbon h1:after
{
   left:0;
   border-width: 0 30px 30px 0;
}
```

The results can be seen on the next slide.

## CSS

### Generated Elements - CSS Ribbon




Our page header now has a unique look and feel, giving it some distinction on the page. Because this was implemented only using CSS, it will load very quickly in the browser without using much processing power or additional HTTP requests.

## CSS

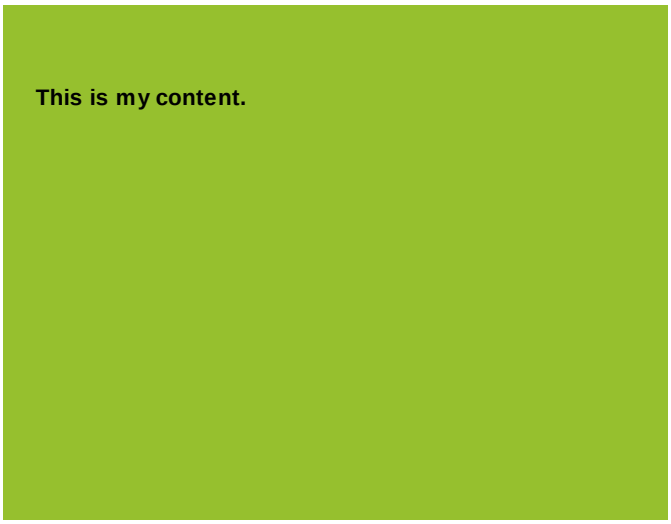
### Generated Elements - Page Fold

Lastly, we'll consider the following HTML snippet:

#### HTML

```
<div class="folded">
   This is my content.
</div>
```

#### Browser




Again, nothing too fancy. Just some plain text in a . To enhance the display of this content, we want to make it look like one corner of the page is folded over.

# CSS

## Generated Elements - Page Fold

Similar to our previous examples, we'll start by formatting the element itself. We give our  some padding, set the background color, and specify that it should be positioned relatively.

### CSS

```
.folded
{
   padding-top:40px;
   padding-right:40px;
   background-color:#96c02e;
   position:relative;
}
```

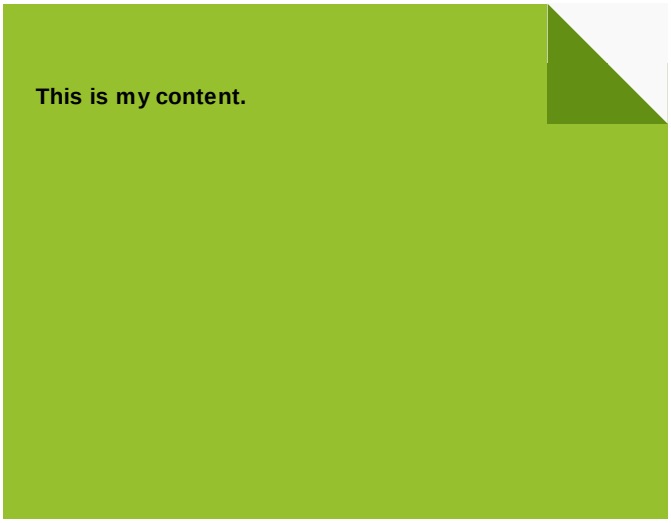
Next, we'll create a single pseudo-element and position it absolutely in the top right corner of the containing element. Lastly, we'll create a triangle shape using CSS borders. It's important to note that while the shape appears to be a triangle, we've actually just matched the background color of the parent's parent element.

### CSS

```
.folded:after
{
   content:"";
   width:0px;
   height:0px;
   position:absolute;
   right:0;
   top:0;
   border:30px solid #638f14;
   border-color: #fff #fff #638f14 #638f14;
}
```

The result is a page that looks as though the corner has been folded over.

### Browser



# CSS

## Reminders

- CSS-generated content is not actually part of the HTML
- CSS-generated content is not accessible by most screen readers
- Don't use this technique for adding non-display/non-trivial content to your w eb pages