

Responsive Design

Modern technology allows users to browse the Internet via multiple devices, such as desktop monitors, mobile phones, tablets, and more.

How can we ensure that a website is readable and visually appealing across all devices, regardless of screen size?

The answer: *responsive design*! Responsive design refers to the ability of a website to resize and reorganize its content based on:

- 1. The size of other content on the website.**
- 2. The size of the screen the website is being viewed on.**

First we'll size HTML content *relative* to other content on a website.

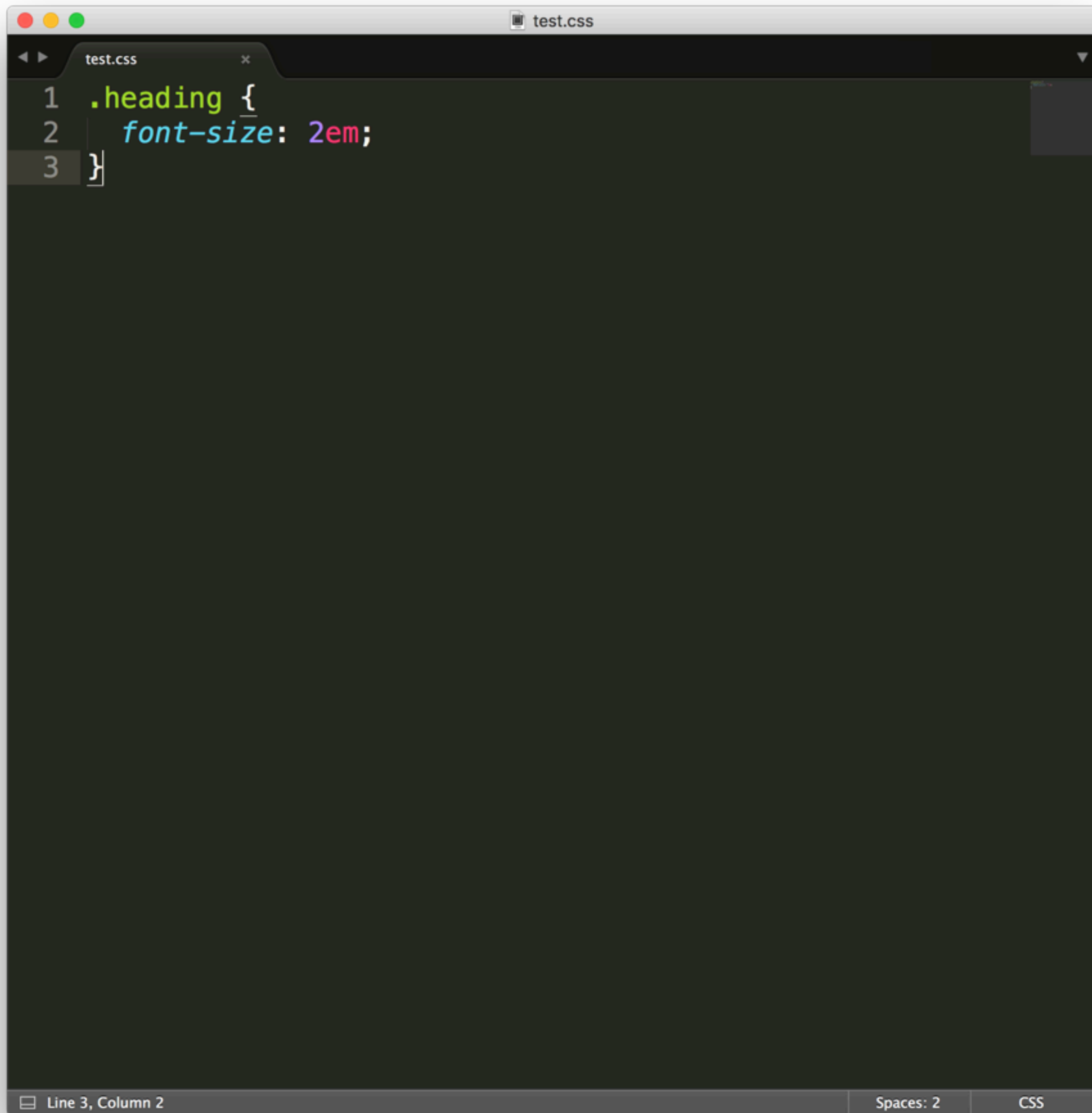
You've probably noticed the unit of **pixels**, or **px**, used in websites. Pixels are used to size content to *exact* dimensions. For example, if you want a div to be exactly 500 pixels wide and 100 pixels tall, then the unit of **px** can be used.

Pixels, however, are fixed, `hard coded` values. When a screen size changes (like switching from landscape to portrait view on a phone), elements sized with pixels can appear too small, overflow the screen, or become completely illegible.

With CSS, you can avoid hard coded measurements and use *relative measurements* instead. Relative measurements offer an advantage over hard coded measurements, as they allow for the proportions of a website to remain intact regardless of screen size or layout.

Incorporating relative sizing starts by using units other than pixels. One unit of measurement you can use in CSS to create relatively-sized content is the *em*, written as **em** in CSS.

Today, the em represents the size of the base font being used. For example, if the base font of a browser is 16 pixels (which is normally the default size of text in a browser), then 1 em is equal to 16 pixels. 2 ems would equal 32 pixels, and so on.

A screenshot of a code editor window titled 'test.css'. The editor shows three lines of CSS code: line 1 is '.heading {' in green, line 2 is 'font-size: 2em;' in blue and red, and line 3 is '}' in green. The editor has a dark background and a light-colored sidebar on the right. The status bar at the bottom shows 'Line 3, Column 2', 'Spaces: 2', and 'CSS'.

```
1 .heading {  
2   font-size: 2em;  
3 }
```

Here no base font has been specified, therefore the font size of the **heading** element will be set relative to the default font size of the browser. Assuming the default font size is 16 pixels, then the font size of the **heading** element will be 32 pixels.

```
test.css
1 .splash-section {
2   font-size: 18px;
3 }
4
5 .splash-section h1 {
6   font-size: 1.5em;
7 }
```

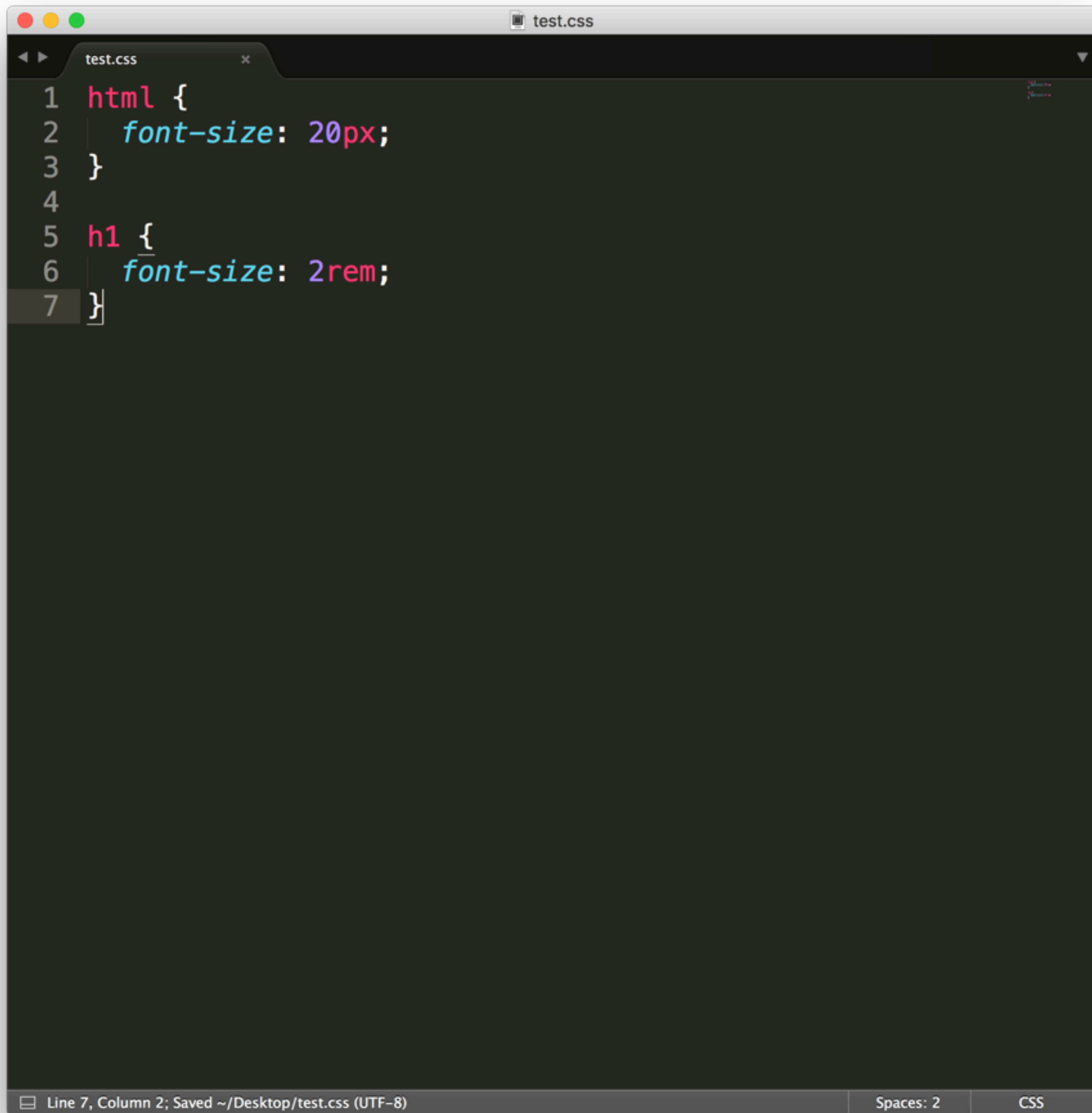
Line 7, Column 2; Saved ~/Desktop/test.css (UTF-8) Spaces: 2 CSS

This example shows how to use ems without relying on the default font size of the browser. Instead, a base font size (**18px**) is defined for all text within the **splash-section** element. The second CSS rule will set the font size of all **h1** elements inside of **splash-section** relative to the base font of **splash-section** (18 pixels). The resulting font size of **h1** elements will be 27 pixels.

The second relative unit of measurement in CSS is the *rem*, coded as **rem**

Rem stands for *root em*. It acts similar to em, but instead of checking parent elements to size font, it checks the *root element*. The root element is the `<html>` tag

Most browsers set the font size of `<html>` to 16 pixels, so by default rem measurements will be compared to that value. To set a different font size for the root element, you can add a CSS rule.



```
1 html {  
2   font-size: 20px;  
3 }  
4  
5 h1 {  
6   font-size: 2rem;  
7 }
```

Line 7, Column 2; Saved ~/Desktop/test.css (UTF-8) Spaces: 2 CSS

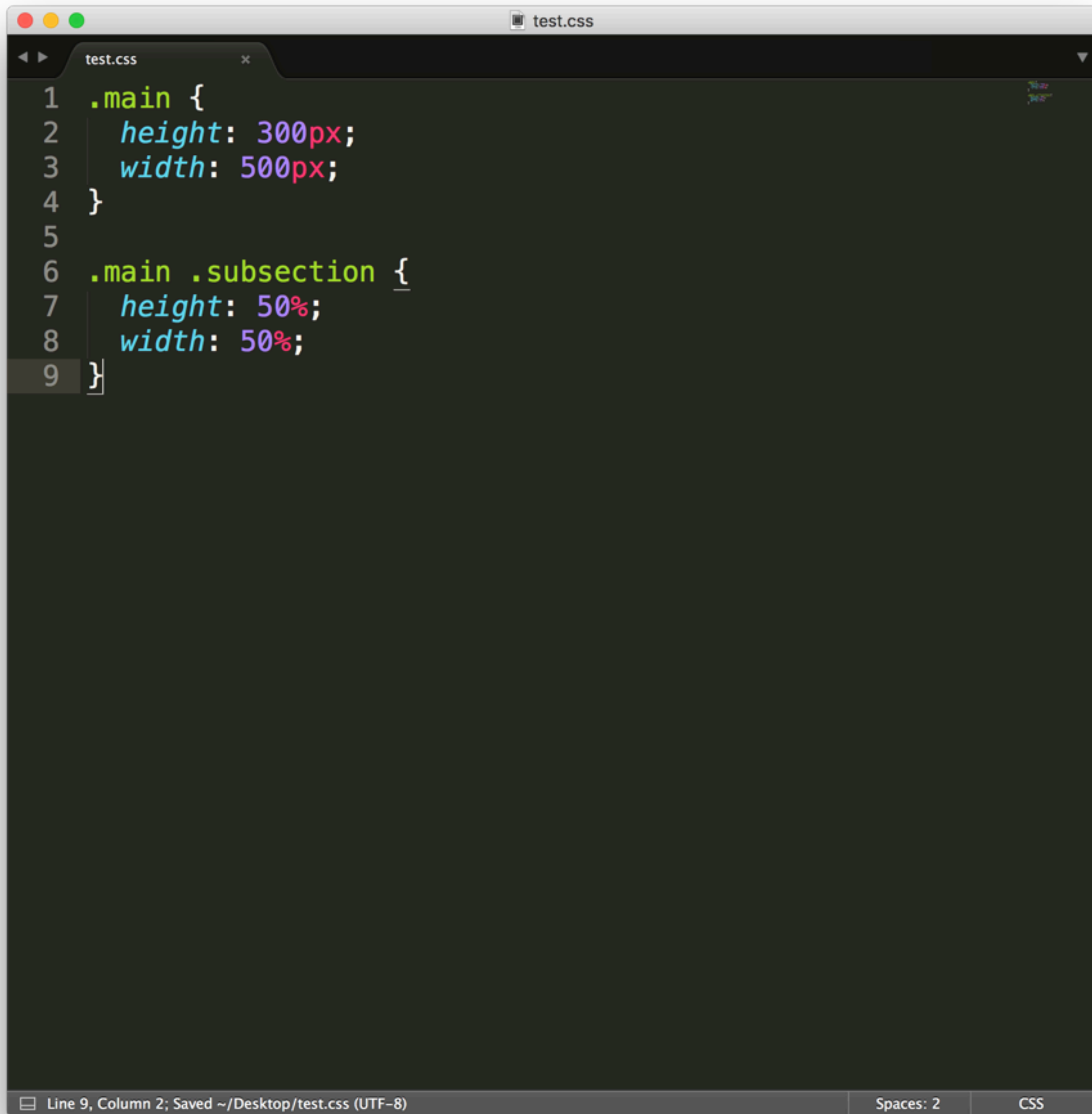
The font size of the root element, **<html>**, is set to 20 pixels. All subsequent rem measurements will now be compared to that value and the size of **h1** elements in the example will be 40 pixels.

One advantage of using rems is that all elements are compared to the same font size value, making it easy to predict how large or small font will appear.

If you are interested in sizing elements consistently across an entire website, the rem measurement is the best unit for the job. If you're interested in sizing elements in comparison to other elements nearby, then the em unit would be better suited for the job.

To size non-text HTML elements relative to their parent elements on the page you can use *percentages*.

Percentages are often used to size box-model values, like width and height, padding, border, and margins. They can also be used to set positioning properties (top, bottom, left, right).

A screenshot of a code editor window titled 'test.css'. The editor shows two CSS rules. The first rule, on lines 1-4, is for the class '.main' with a height of 300px and a width of 500px. The second rule, on lines 6-9, is for the class '.main .subsection' with a height of 50% and a width of 50%. The code is syntax-highlighted: selectors are green, properties are blue, and values are purple. The status bar at the bottom indicates 'Line 9, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

```
1 .main {  
2   height: 300px;  
3   width: 500px;  
4 }  
5  
6 .main .subsection {  
7   height: 50%;  
8   width: 50%;  
9 }
```

When percentages are used, elements are sized relative to the dimensions of their parent element (also known as a container). Therefore, the dimensions of the **.subsection** div will be 150 pixels tall and 250 pixels wide. Be careful, a child element's dimensions may be set erroneously if the dimensions of its parent element aren't set first.

Percentages can also be used to set the padding and margin of elements.

When height and width are set using percentages, you learned that the dimensions of child elements are calculated based on the dimensions of the parent element.

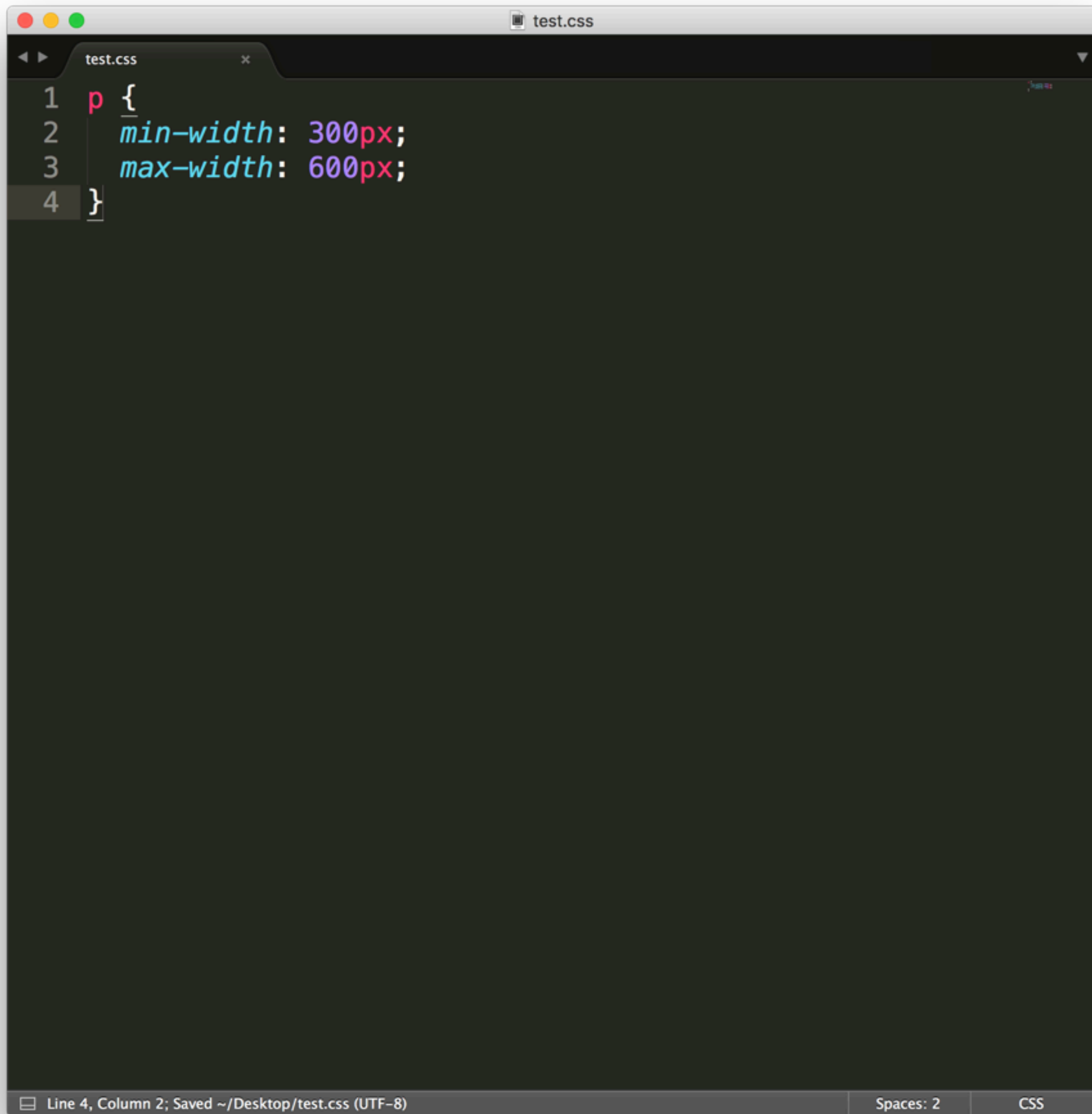
When percentages are used to set padding and margin, however, they are calculated based only on the *width* of the parent element.

For example, when a property like `margin-left` is set using a percentage (say `50%`), the element will be moved halfway to the right in the parent container (as opposed to the child element receiving a margin half of its parent's margin).

Although relative measurements provide consistent layouts across devices of different screen sizes, elements on a website can lose their integrity when they become too small or large.

You can limit how wide an element becomes with the following properties:

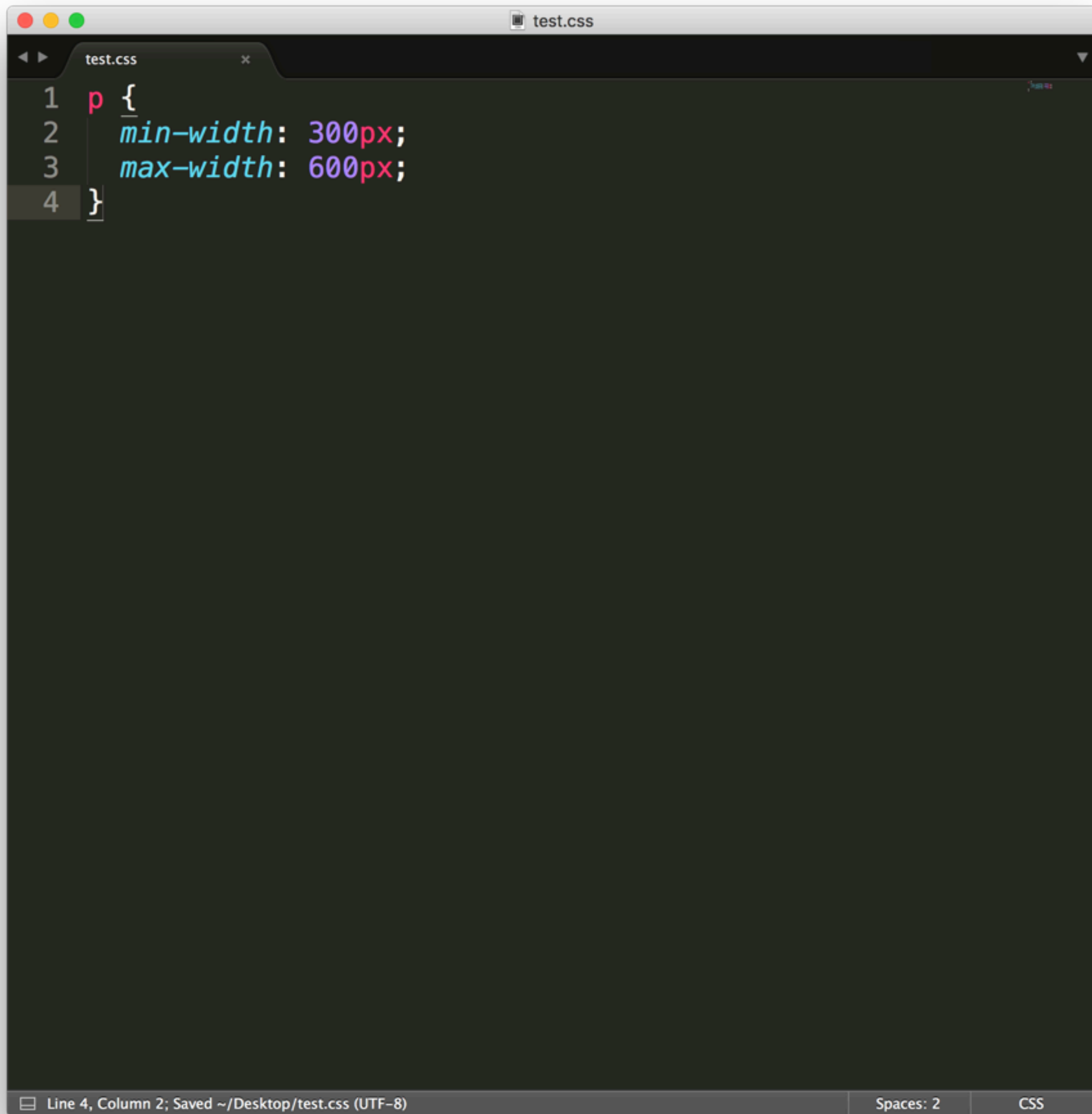
1. **min-width** — ensures a minimum width for an element.
2. **max-width** — ensures a maximum width for an element.



```
1 p {  
2   min-width: 300px;  
3   max-width: 600px;  
4 }
```

The image shows a code editor window titled 'test.css'. The code defines a CSS rule for the 'p' (paragraph) element, setting a minimum width of 300px and a maximum width of 600px. The editor has a dark theme and a sidebar on the left. The status bar at the bottom indicates 'Line 4, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

In this example, when the browser is resized, the width of paragraph elements will not fall below 300 pixels, nor will their width exceed 600 pixels.

A screenshot of a code editor window titled 'test.css'. The editor has a dark theme and shows four lines of CSS code. Line 1: 'p {'; Line 2: 'min-width: 300px;'; Line 3: 'max-width: 600px;'; Line 4: '}'. The code is color-coded: 'p' is red, '{' is blue, 'min-width' and 'max-width' are cyan, '300px' and '600px' are purple, and ';' and '}' are blue. The status bar at the bottom indicates 'Line 4, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

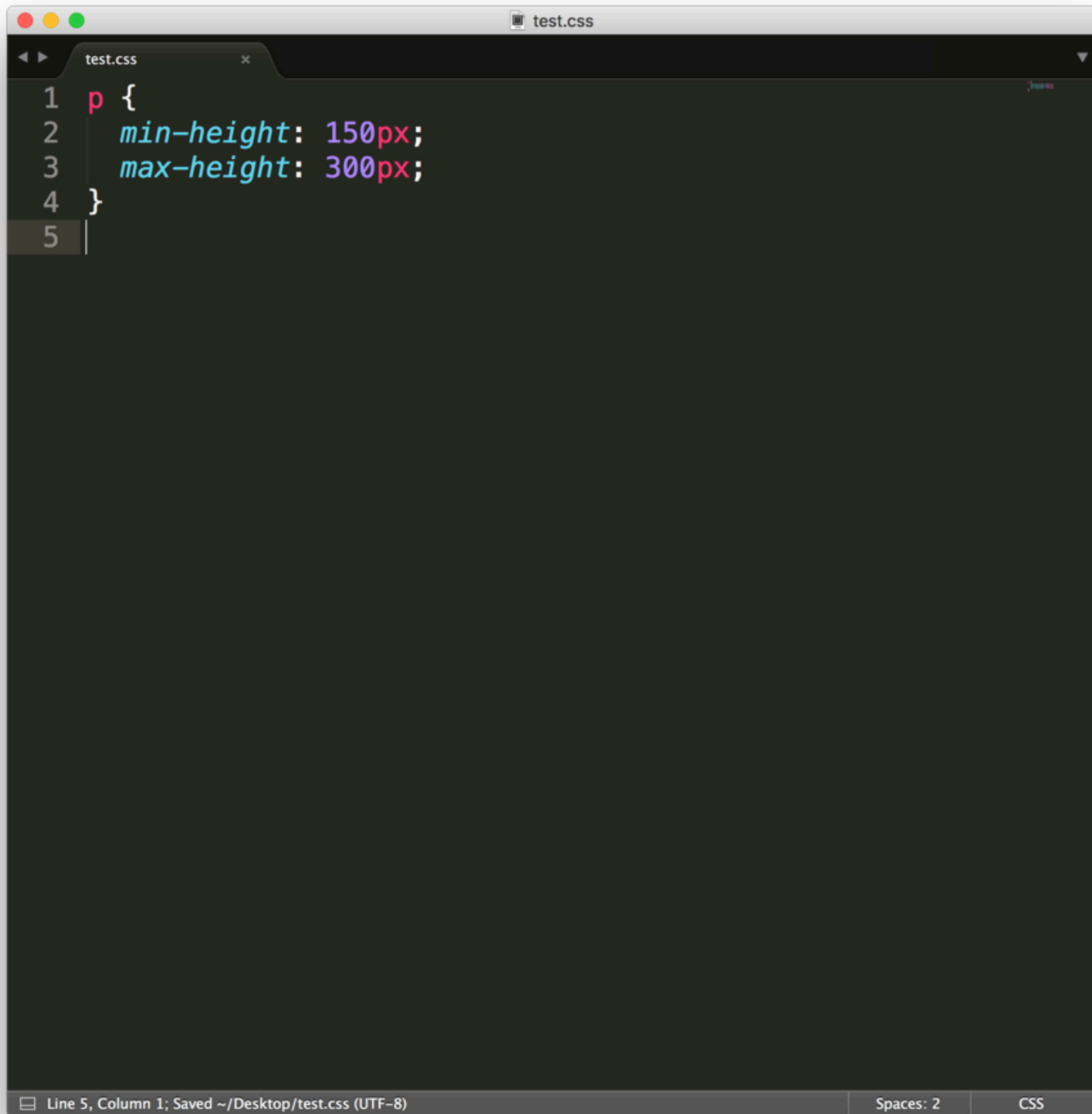
```
1 p {  
2   min-width: 300px;  
3   max-width: 600px;  
4 }
```

When a browser window is narrowed or widened, text can become either very compressed or very spread out, making it difficult to read. These two properties ensure that content is legible by limiting the minimum and maximum widths.

You can also limit the minimum and maximum *height* of an element.

min-height — ensures a minimum height for an element's box.

max-height — ensures a maximum height for an element's box.

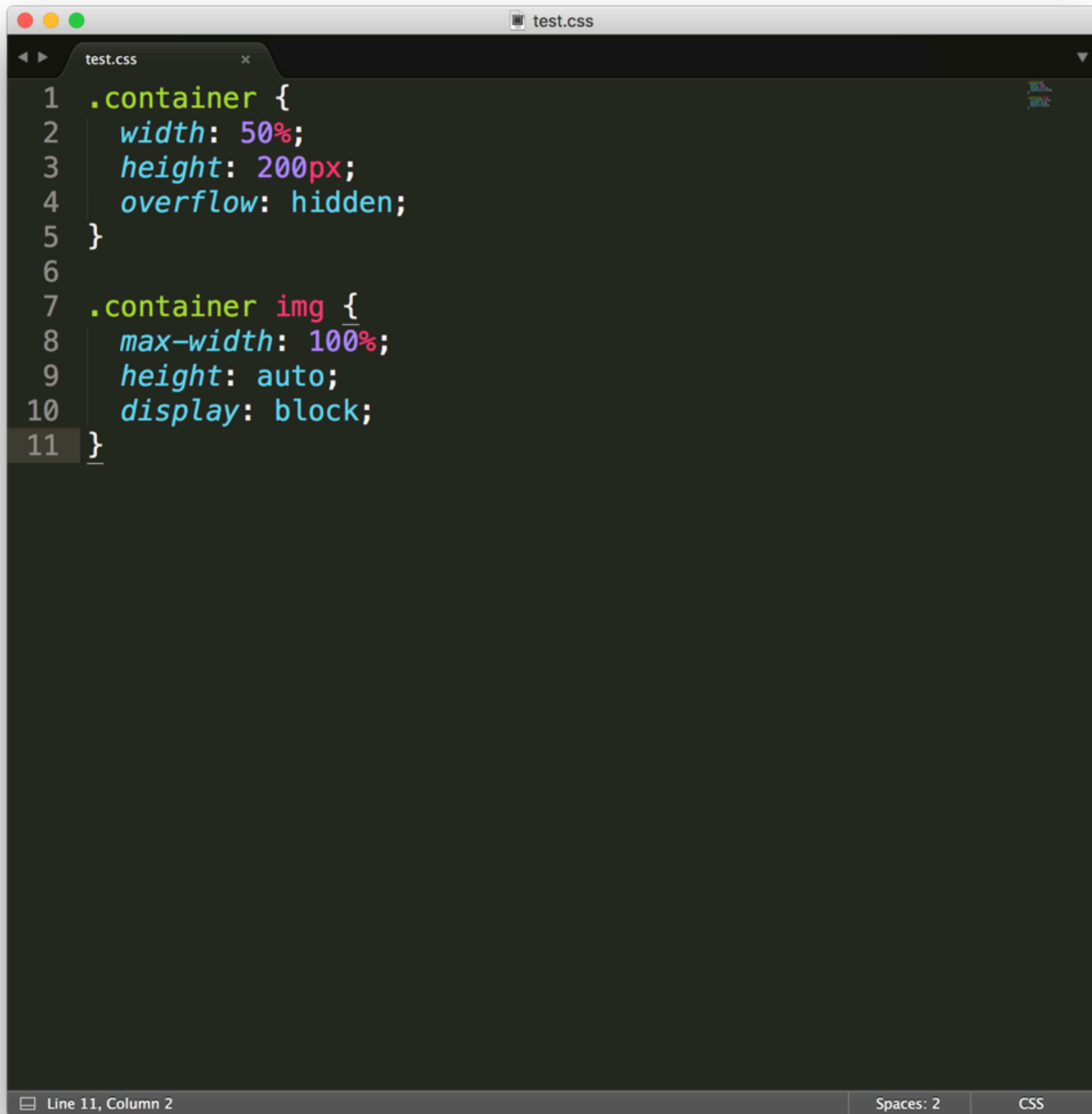
A screenshot of a code editor window titled 'test.css'. The editor has a dark background with light-colored text. The code is as follows:

```
1 p {  
2   min-height: 150px;  
3   max-height: 300px;  
4 }  
5 |
```

The line numbers 1 through 5 are on the left. The code is color-coded: 'p' is red, '{' is white, 'min-height' and 'max-height' are cyan, '150px' and '300px' are purple, and '}' is white. The status bar at the bottom shows 'Line 5, Column 1; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

In this example, the height of all paragraphs will not shrink below 150 pixels and the height will not exceed 300 pixels.

Many websites contain a variety of different media, like images and videos. When a website contains such media, it's important to make sure that it is scaled proportionally so that users can correctly view it.



```
1 .container {  
2   width: 50%;  
3   height: 200px;  
4   overflow: hidden;  
5 }  
6  
7 .container img {  
8   max-width: 100%;  
9   height: auto;  
10  display: block;  
11 }
```

The image shows a code editor window titled 'test.css'. The code defines a CSS class '.container' with a width of 50%, a height of 200px, and an overflow of hidden. It also defines a rule for '.container img' with a max-width of 100%, height of auto, and display of block. The editor has a dark theme and a status bar at the bottom showing 'Line 11, Column 2', 'Spaces: 2', and 'CSS'.

.container represents a container div. It is set to a width of **50%** (half of the browser's width, in this example) and a height of 200 pixels.

Setting **overflow** to **hidden** ensures that any content with dimensions larger than the container will be hidden from view.

```
test.css
1 .container {
2   width: 50%;
3   height: 200px;
4   overflow: hidden;
5 }
6
7 .container img {
8   max-width: 100%;
9   height: auto;
10  display: block;
11 }
```

Line 11, Column 2 Spaces: 2 CSS

The second CSS rule ensures that images scale with the width of the container. The **height** property is set to **auto**, meaning an image's height will *automatically* scale proportionally with the width. Finally, the last line will display images as block level elements (rather than inline-block, their default state). This will prevent images from attempting to align with other content on the page (like text), which can add unintended margin to the images.

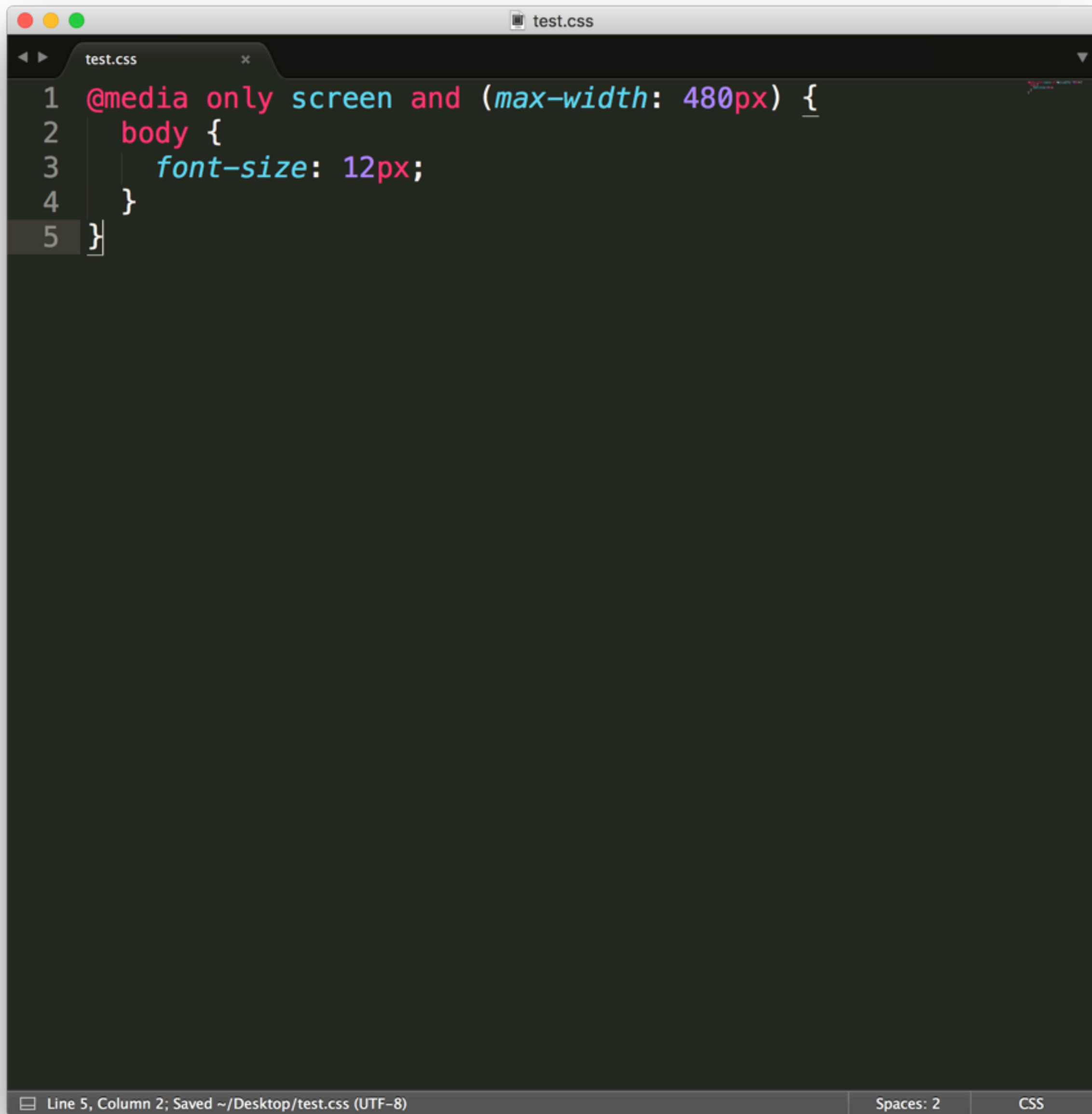
```
test.css
1 .container {
2   width: 50%;
3   height: 200px;
4   overflow: hidden;
5 }
6
7 .container img {
8   max-width: 100%;
9   height: auto;
10  display: block;
11 }
```

Line 11, Column 2 Spaces: 2 CSS

It's worth memorizing this entire example. It represents a *very common* design pattern used to scale images and videos proportionally.

Media Queries

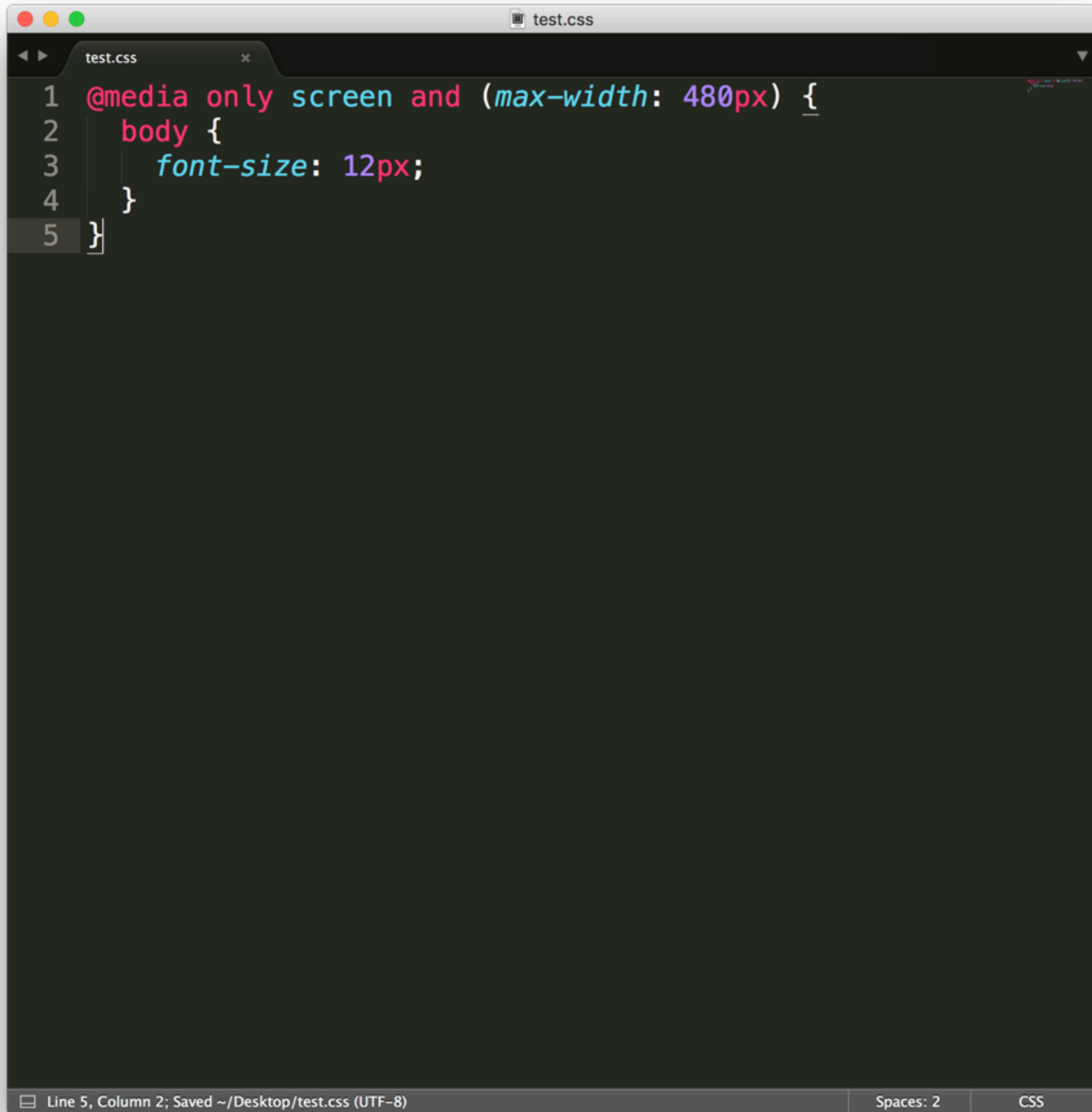
CSS uses *media queries* to adapt a website's content to different screen sizes. With media queries, CSS can detect the size of the current screen and apply different CSS styles depending on the width of the screen.

A screenshot of a code editor window titled 'test.css'. The editor has a dark background with syntax-highlighted CSS code. The code is as follows:

```
1 @media only screen and (max-width: 480px) {  
2   body {  
3     font-size: 12px;  
4   }  
5 }
```

The status bar at the bottom indicates 'Line 5, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

The example above demonstrates how a media query is applied. The media query defines a rule for screens smaller than 480 pixels (approximately the width of many smartphones in **landscape** orientation).

A screenshot of a code editor window titled 'test.css'. The editor has a dark background with syntax-highlighted CSS code. The code is as follows:

```
1 @media only screen and (max-width: 480px) {  
2   body {  
3     font-size: 12px;  
4   }  
5 }
```

The status bar at the bottom indicates 'Line 5, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

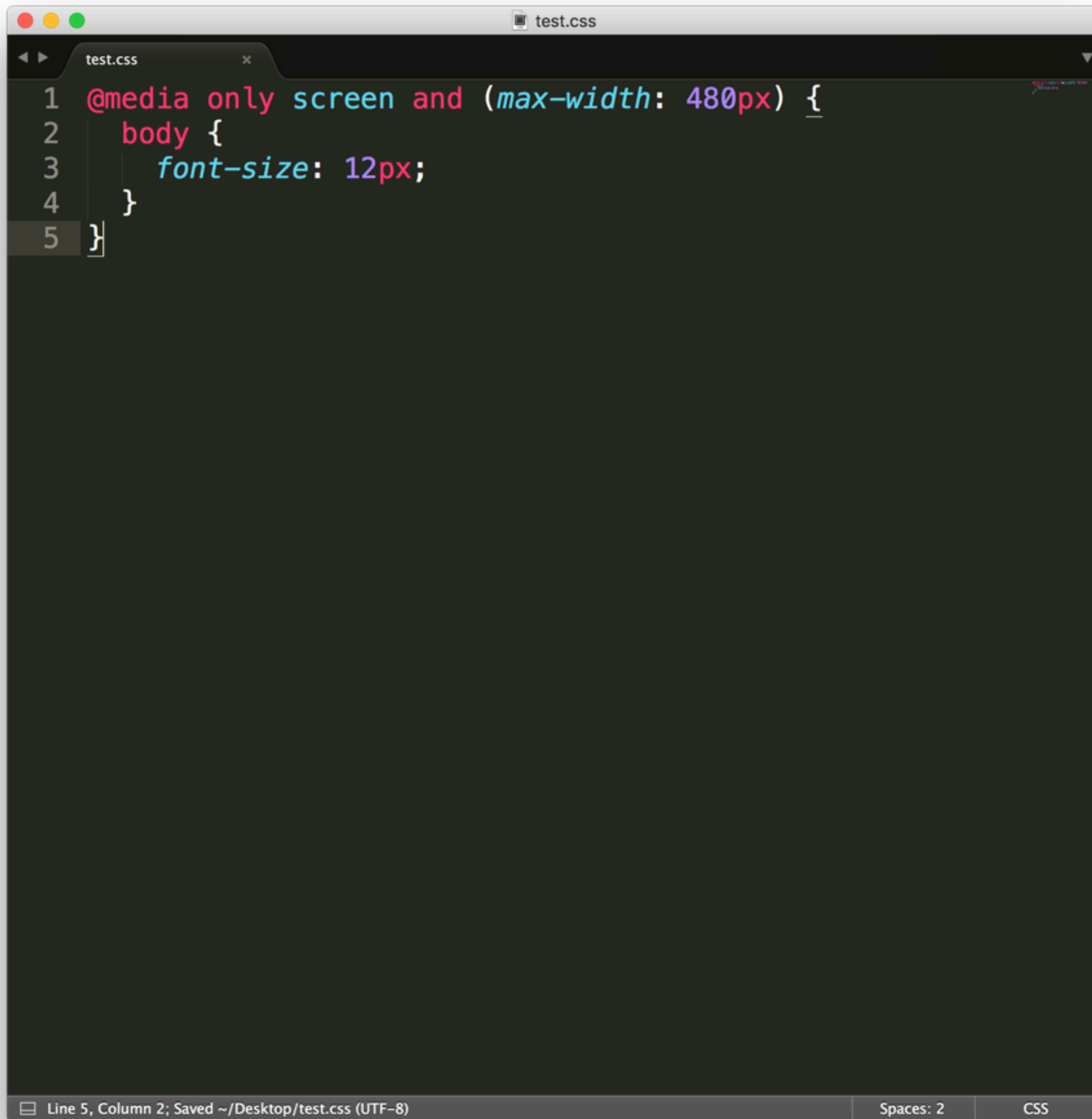
Let's break this example down into its parts:

1. **@media** — This keyword begins a media query rule and instructs the CSS compiler on how to parse the rest of the rule.

```
test.css
1 @media only screen and (max-width: 480px) {
2   body {
3     font-size: 12px;
4   }
5 }
```

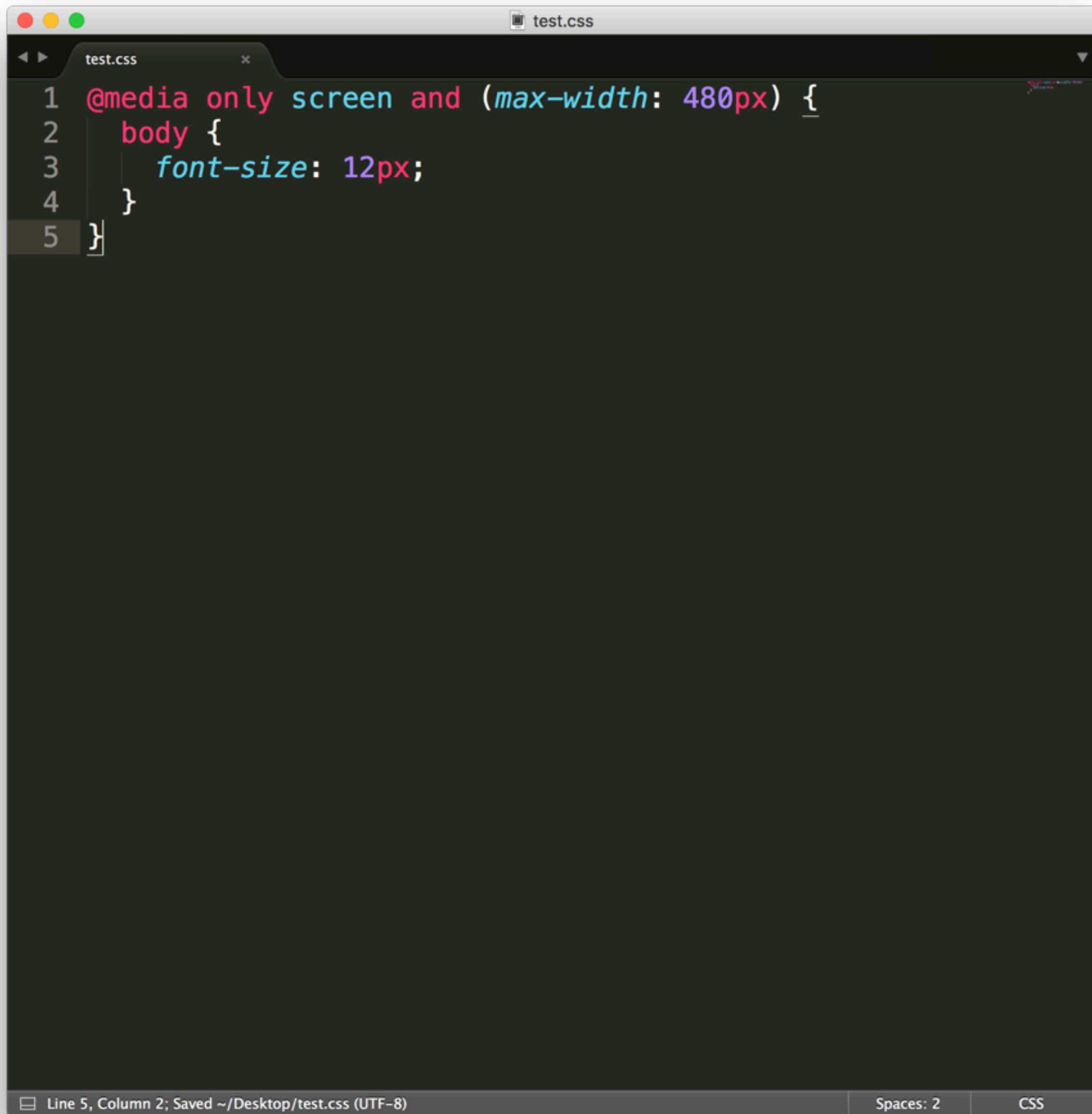
Line 5, Column 2; Saved ~/Desktop/test.css (UTF-8) Spaces: 2 CSS

2. **only screen** — Indicates what types of devices should use this rule. **screen** is the media type always used for displaying content, no matter the type of device. The **only** keyword is added to indicate that this rule only applies to one media type (**screen**).

A screenshot of a code editor window titled 'test.css'. The editor shows five lines of CSS code. Line 1: '@media only screen and (max-width: 480px) {'; Line 2: ' body {'; Line 3: ' font-size: 12px;'; Line 4: ' }'; Line 5: '}' (highlighted). The status bar at the bottom indicates 'Line 5, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (max-width: 480px) {  
2   body {  
3     font-size: 12px;  
4   }  
5 }
```

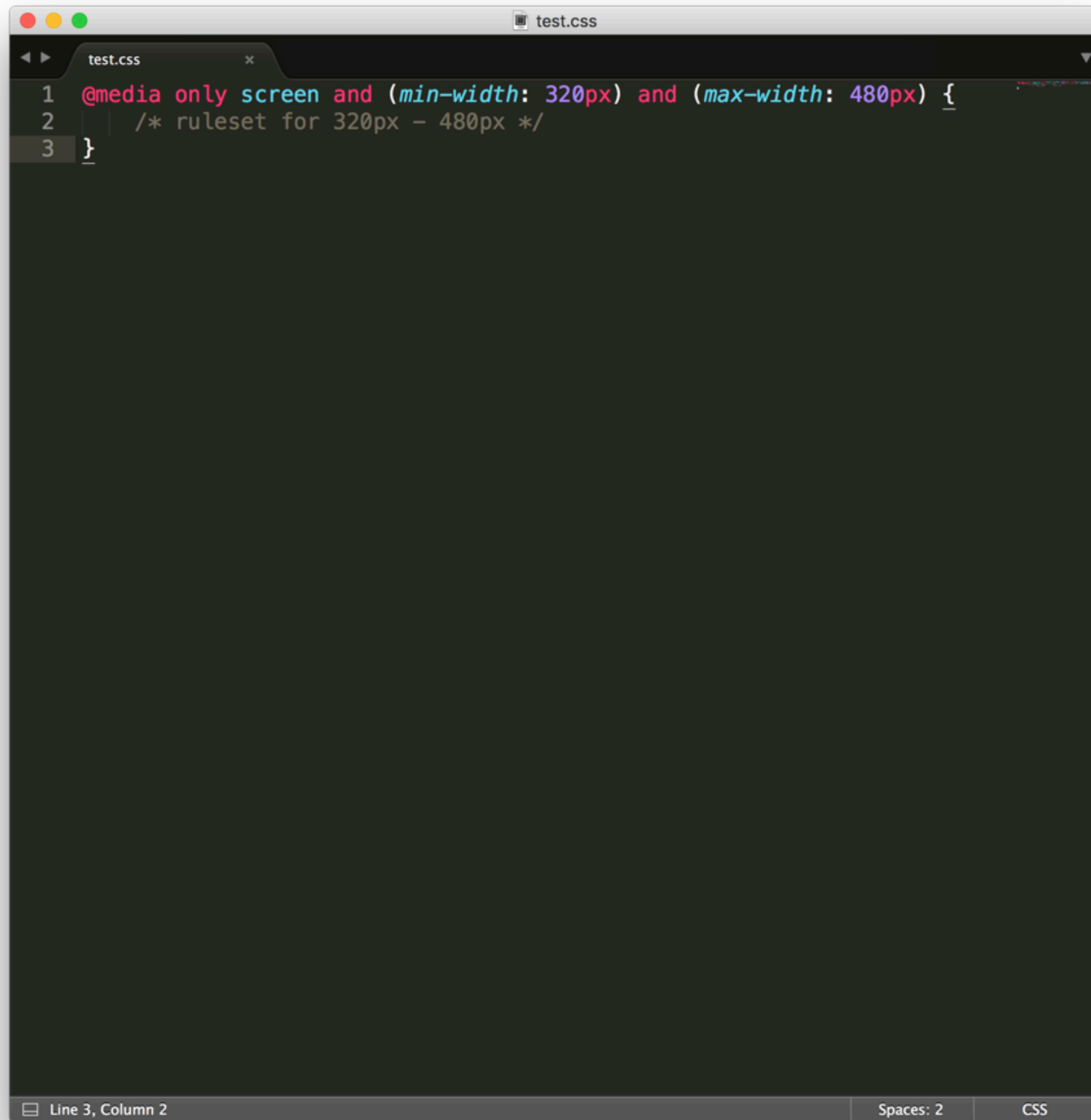
3. **and (max-width : 480px)** — This part of the rule is called a *media feature*, and instructs the CSS compiler to apply the CSS styles to devices with a width of 480 pixels or smaller. Media features are the conditions that must be met in order to render the CSS within a media query.

A screenshot of a code editor window titled 'test.css'. The editor has a dark background with syntax-highlighted CSS code. The code consists of five lines: a media query '@media only screen and (max-width: 480px) {' on line 1, followed by a 'body {' selector on line 2, a 'font-size: 12px;' property on line 3, a closing brace for the selector on line 4, and a closing brace for the media query on line 5. The status bar at the bottom indicates 'Line 5, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (max-width: 480px) {  
2   body {  
3     font-size: 12px;  
4   }  
5 }
```

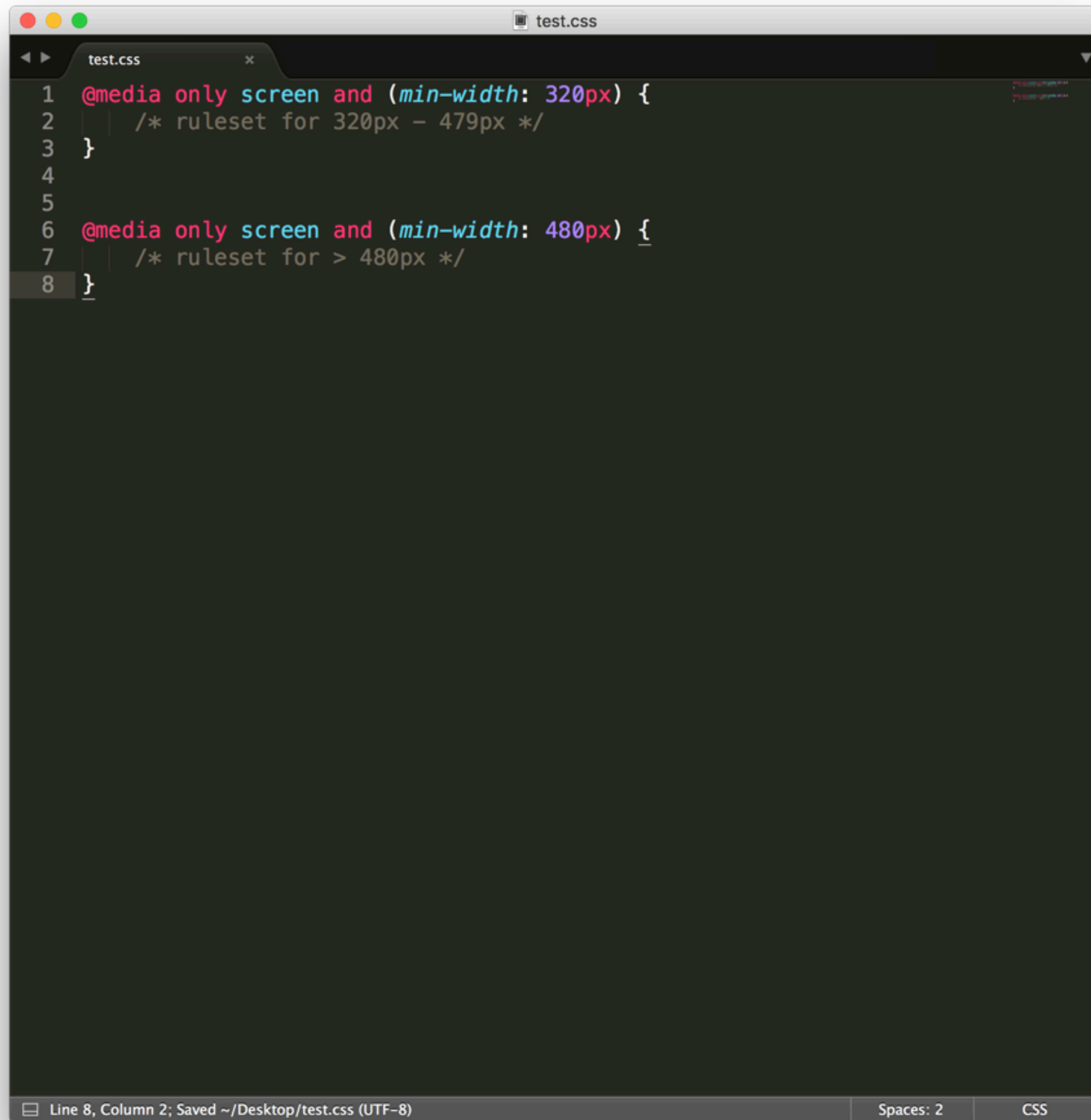
4. CSS rules are nested inside of the media query's curly braces. The rules will be applied when the media query is met. In the example above, the text in the **body** element is set to a **font-size** of **12px** when the user's screen is less than 480px.

Specific screen sizes can be targeted by setting multiple width and height media features. **min-width** and **min-height** are used to set the minimum width and minimum height, respectively. Conversely, **max-width** and **max-height** set the maximum width and maximum height, respectively.

A screenshot of a code editor window titled 'test.css'. The editor shows three lines of CSS code: line 1: '@media only screen and (min-width: 320px) and (max-width: 480px) {'; line 2: '/* ruleset for 320px - 480px */'; line 3: '}'. The code is syntax-highlighted. The status bar at the bottom indicates 'Line 3, Column 2', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (min-width: 320px) and (max-width: 480px) {  
2   /* ruleset for 320px - 480px */  
3 }
```

This example would apply its CSS rules only when the screen size is between 320 pixels and 480 pixels. Notice the use of a second **and** keyword after the **min-width** media feature. This allows us to chain two requirements together.

A screenshot of a code editor window titled 'test.css'. The editor shows two media queries. The first query is on lines 1-3: '@media only screen and (min-width: 320px) { /* ruleset for 320px - 479px */ }'. The second query is on lines 6-8: '@media only screen and (min-width: 480px) { /* ruleset for > 480px */ }'. The editor has a dark theme and a status bar at the bottom indicating 'Line 8, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (min-width: 320px) {
2   /* ruleset for 320px - 479px */
3 }
4
5
6 @media only screen and (min-width: 480px) {
7   /* ruleset for > 480px */
8 }
```

The last example can be written using two separate rules as well.

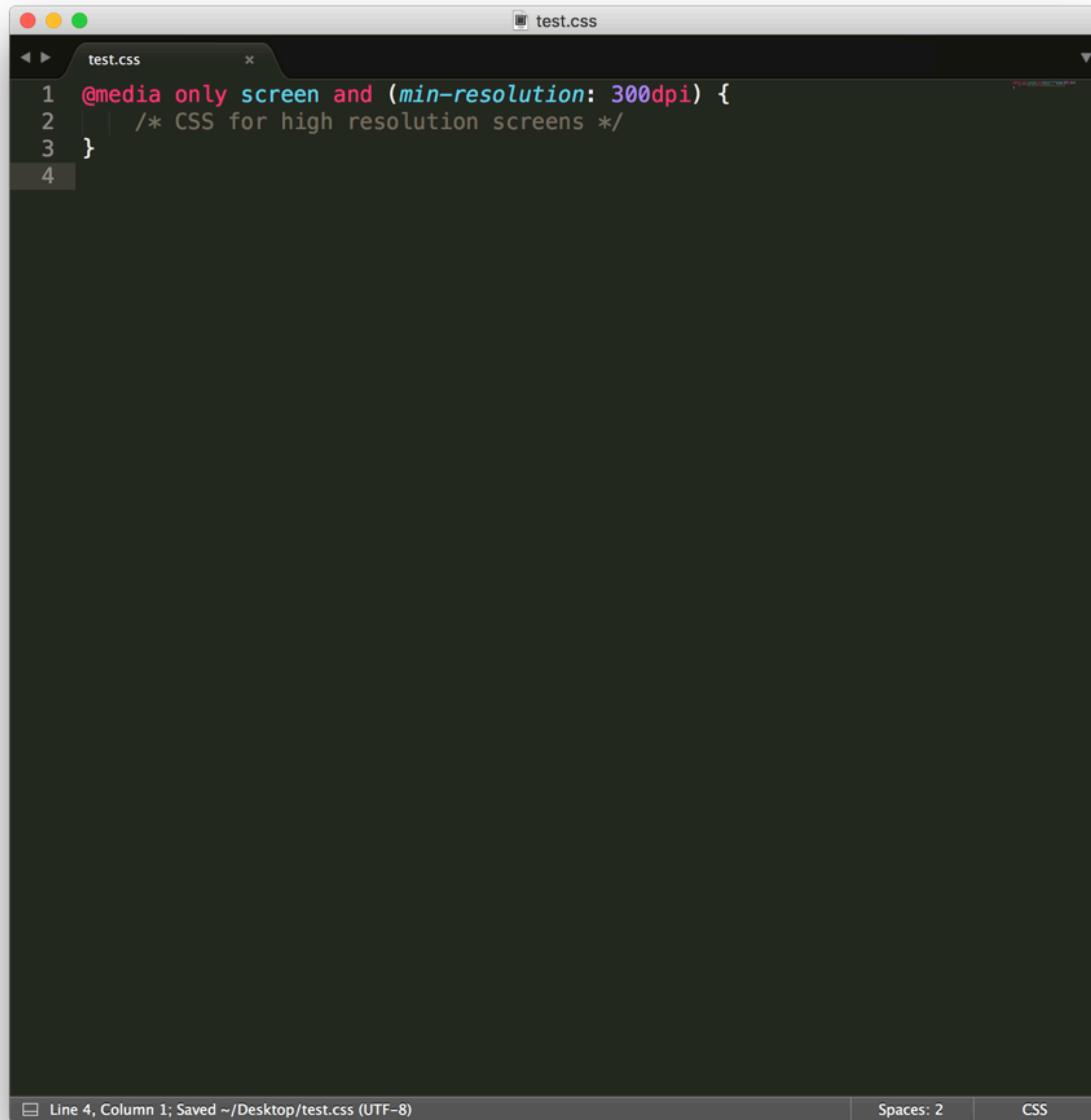
The first media query in this example will apply CSS rules when the size of the screen meets or exceeds 320 pixels. The second media query will apply CSS rules when the size of the screen meets or exceeds 480 pixels, meaning that it will override the CSS rules present in the first media query.

Both examples above are valid, and it is likely that you will see both patterns used when reading another developer's code.

Another media feature we can target is screen resolution. Many times we will want to supply higher quality media (images, video, etc.) only to users with screens that can support high resolution media.

Targeting screen resolution also helps users avoid downloading high resolution (large file size) images that their screen may not be able to properly display.

To target by resolution, we can use the **min-resolution** and **max-resolution** media features. These media features accept a resolution value in either dots per inch (dpi) or dots per centimeter (dpc).

A screenshot of a code editor window titled 'test.css'. The editor has a dark theme and shows the following CSS code:

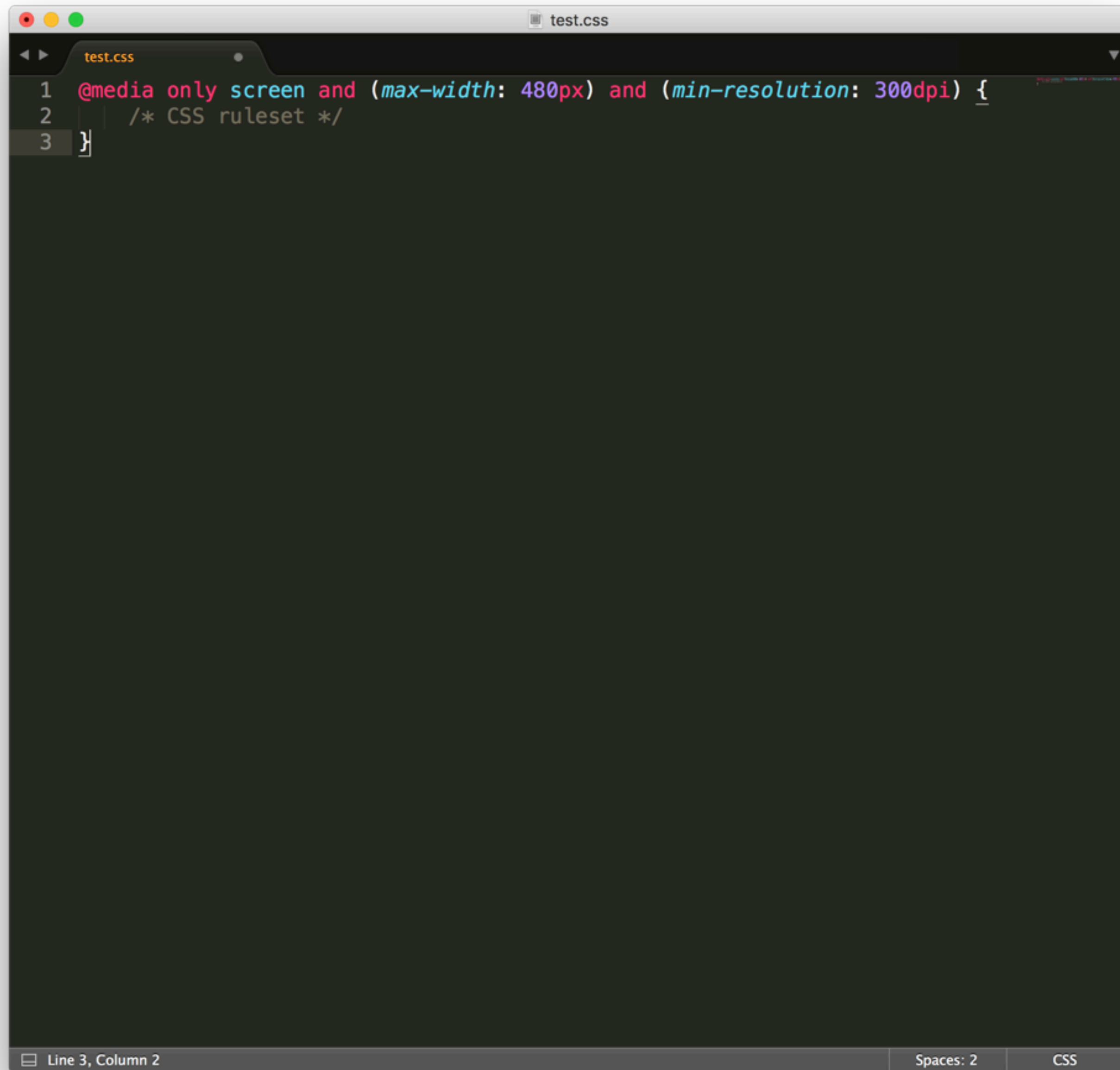
```
1 @media only screen and (min-resolution: 300dpi) {  
2     /* CSS for high resolution screens */  
3 }  
4
```

The line numbers 1 through 4 are visible on the left side of the editor. The status bar at the bottom indicates 'Line 4, Column 1; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

The media query in this example targets high resolution screens by making sure the screen resolution is at least 300 dots per inch. If the screen resolution query is met, then we can use CSS to display high resolution images and other media.

In previous examples, we chained multiple media features of the same type in one media query by using the **and** operator. It allowed us to create a range by using **min-width** and **max-width** in the same media query.

The **and** operator can be used to require multiple media features. Therefore, we can use the **and** operator to require both a **max-width** of **480px** *and* to have a **min-resolution** of **300dpi**.

A screenshot of a code editor window titled 'test.css'. The editor shows three lines of CSS code. Line 1: '@media only screen and (max-width: 480px) and (min-resolution: 300dpi) {'; Line 2: '/* CSS ruleset */'; Line 3: '}'. The code is syntax-highlighted. The status bar at the bottom indicates 'Line 3, Column 2', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (max-width: 480px) and (min-resolution: 300dpi) {  
2   /* CSS ruleset */  
3 }
```

By placing the **and** operator between the two media features, the browser will require both media features to be true before it renders the CSS within the media query.

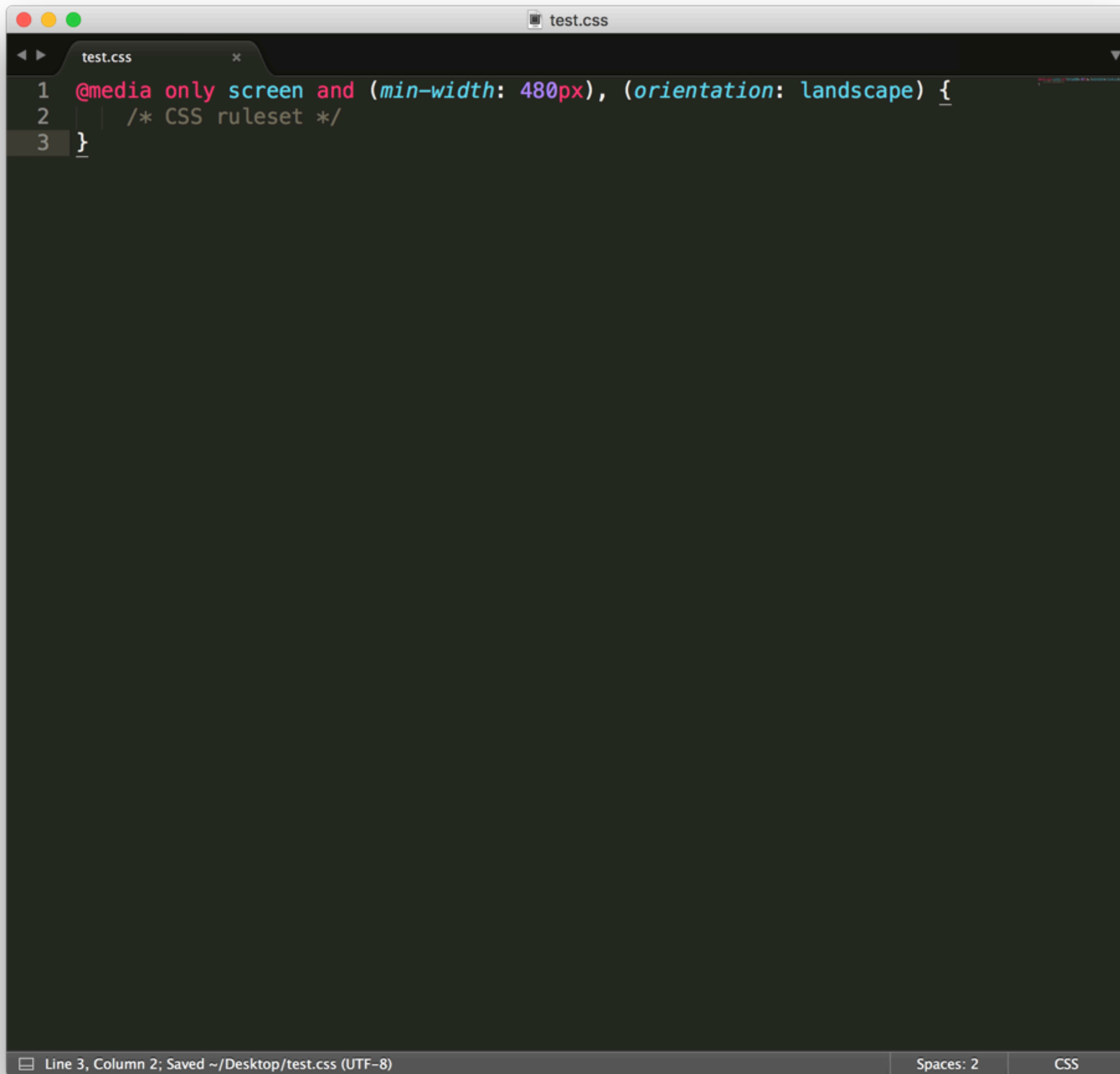
The **and** operator can be used to chain as many media features as necessary.

If only one of multiple media features in a media query must be met, media features can be separated in a comma separated list.

For example, if we needed to apply a style when only one of the below is true:

The screen is more than 480 pixels wide

The screen is in landscape mode

A screenshot of a code editor window titled 'test.css'. The editor shows three lines of CSS code: line 1: '@media only screen and (min-width: 480px), (orientation: landscape) {'; line 2: '/* CSS ruleset */'; line 3: '}'. The code is syntax-highlighted with colors: '@media' is red, 'only' is blue, 'screen' is green, 'and' is red, '(min-width: 480px)' is blue, ',' is red, '(orientation: landscape)' is blue, '{' is red, '/* CSS ruleset */' is green, and '}' is red. The status bar at the bottom indicates 'Line 3, Column 2; Saved ~/Desktop/test.css (UTF-8)', 'Spaces: 2', and 'CSS'.

```
1 @media only screen and (min-width: 480px), (orientation: landscape) {
2   /* CSS ruleset */
3 }
```

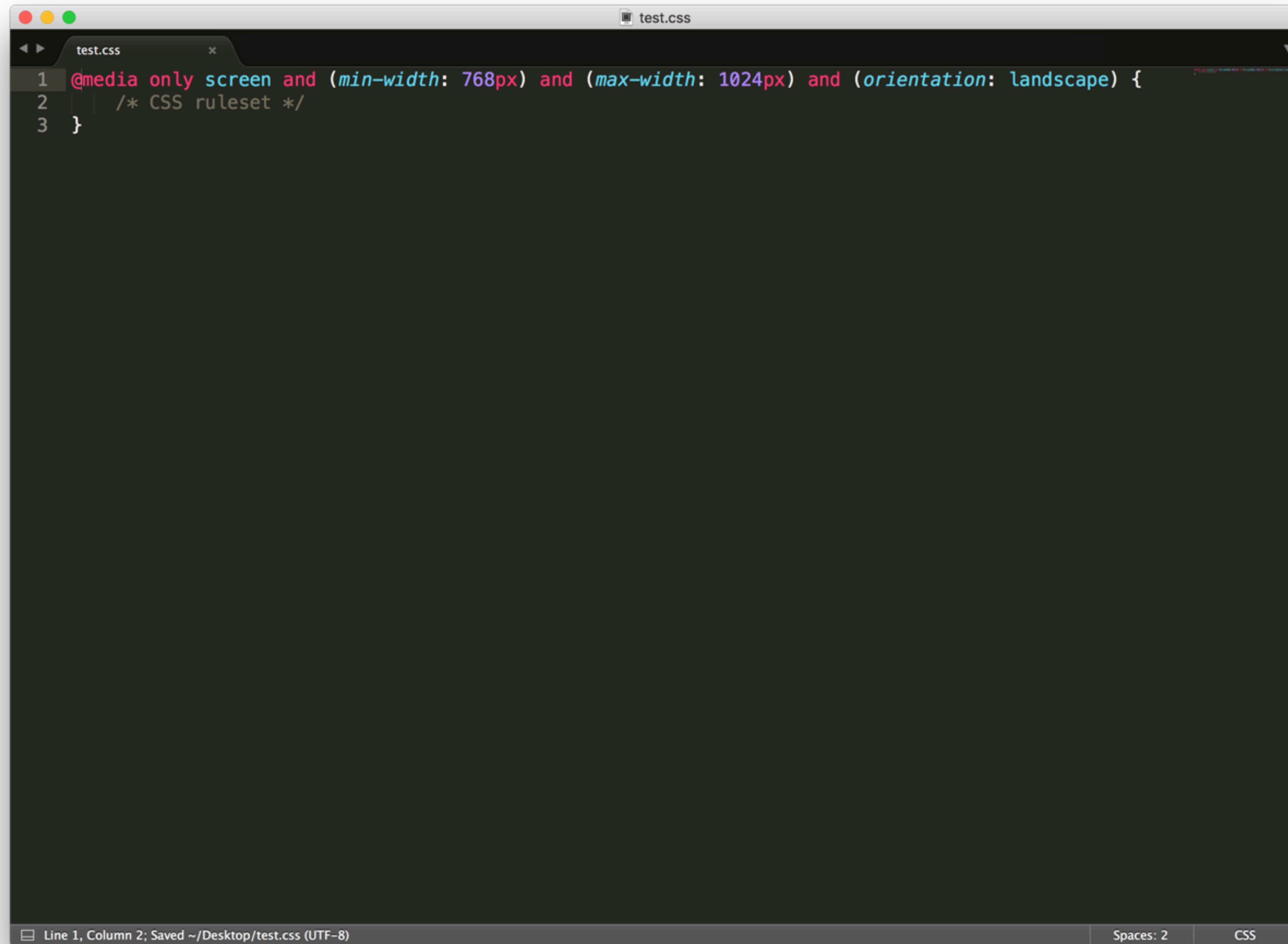
In the example above, we used a comma (,) to separate multiple rules. The example above requires only one of the media features to be true for its CSS to apply.

Note that the second media feature is **orientation**. The **orientation** media feature detects if the page has more width than height. If a page is wider, it's considered **landscape**, and if a page is taller, it's considered **portrait**.

We know how to use media queries to apply CSS rules based on screen size and resolution, but how do we determine what queries to set?

The points at which media queries are set are called *breakpoints*. Breakpoints are the screen sizes at which your web page does not appear properly.

For example, if we want to target tablets that are in landscape orientation, we can create the following breakpoint:



The image shows a code editor window with a dark theme. The title bar at the top has three colored window control buttons (red, yellow, green) on the left and a document icon followed by the text 'test.css' on the right. Below the title bar is a tab bar with a single tab labeled 'test.css' and a close button. The main editing area contains three lines of CSS code, with line numbers 1, 2, and 3 on the left. The code is:

```
1 @media only screen and (min-width: 768px) and (max-width: 1024px) and (orientation: landscape) {  
2     /* CSS ruleset */  
3 }
```

 The code is syntax-highlighted: '@media' is blue, 'only' is red, 'screen' is blue, 'and' is red, '(min-width: 768px)' is blue, 'and' is red, '(max-width: 1024px)' is blue, 'and' is red, '(orientation: landscape)' is blue, '{' is blue, '/* CSS ruleset */' is red, and '}' is blue. At the bottom of the window is a status bar with a document icon, the text 'Line 1, Column 2; Saved ~/Desktop/test.css (UTF-8)', a separator, the text 'Spaces: 2', another separator, and the text 'CSS'.

```
1 @media only screen and (min-width: 768px) and (max-width: 1024px) and (orientation: landscape) {  
2     /* CSS ruleset */  
3 }
```

Line 1, Column 2; Saved ~/Desktop/test.css (UTF-8) Spaces: 2 CSS

However, setting breakpoints for every device imaginable would be incredibly difficult because there are many devices of differing shapes and sizes. In addition, new devices are released with new screen sizes every year.

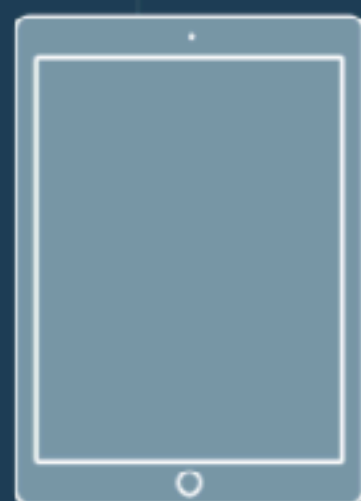
Rather than set breakpoints based on specific devices, the best practice is to resize your browser to view where the website naturally breaks based on its content. The dimensions at which the layout breaks or looks odd become your media query breakpoints. Within those breakpoints, we can adjust the CSS to make the page resize and reorganize.

By observing the dimensions at which a website naturally breaks, you can set media query breakpoints that create the best possible user experience on a project by project basis, rather than forcing every project to fit a certain screen size.

0 200 400 600 800 1000 1200 1400 1600 1800 2000+



Mobile
 $\leq 480\text{px}$



Tablet
 $\leq 768\text{px}$



Tablet
Landscape
 $\leq 1024\text{px}$



Laptop
 $\leq 1600\text{px}$



Desktop
 $> 1600\text{px}$