Responsive web design

Modern technology allows users to browse the Internet via multiple devices, such as desktop monitors, mobile phones, tablets, and more. Devices of different screen sizes, however, pose a problem for web developers: 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.

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.

Sizing elements:

Em

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.

Let's take a look at two examples that show how em can be used in CSS.

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

In the example above, 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 headingelement will be 32 pixels.

```
.splash-section {
  font-size: 18px;
};
.splash-section h1 {
  font-size: 1.5em;
}
```

The example above 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-sectionelement. The second CSS rule will set the font size of all h1 elements inside of splash-sectionrelative to the base font of splash-section (18 pixels). The resulting font size of h1 elements will be 27 pixels.

Rem

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.

```
html {
  font-size: 20px;
}

h1 {
```

```
  font-size: 2rem;
}
```

In the example above, 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 h1elements 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.

Percentages: Height & Width

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).

To start, let's size the height and width of an element using percentages.

```
.main {
  height: 300px;
  width: 500px;
}

.main .subsection {
  height: 50%;
  width: 50%;
}
```

In the example above, .main and .subsection each represent divs. The .subsection div is nested within the .main div. Note that the dimensions of the parent div (.main) have been set to a height of 300 pixels and a width of 500 pixels.

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.

Note: Because the box model includes padding, borders, and margins, setting an element's width to 100% may cause content to overflow its parent container. While tempting, 100% should only be used when content will not have padding, border, or margin.

Percentages: Padding & Margin

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).

Vertical padding and margin are also calculated based on the width of the parent. Why? Consider the following scenario:

1. A container div is defined, but its height is not set (meaning it's flat).
2. The container then has a child element added within. The child element *does* have a set height. This causes the height of its parent container to stretch to that height.
3. The child element requires a change, and its height is modified. This causes the parent container's height to also stretch to the new height. This cycle occurs endlessly whenever the child element's height is changed!

In the scenario above, an unset height (the parent's) results in a constantly changing height due to changes to the child element. This is why vertical padding and margin are based on the width of the parent, and not the height.

Note: When using relative sizing, ems and rems should be used to size text and dimensions on the page related to text size (i.e. padding around text). This creates a consistent layout based on text size. Otherwise, percentages should be used.

Width: Minimum & Maximum

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.

```
p {
  min-width: 300px;
  max-width: 600px;
}
```

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

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.

Height: Minimum & Maximum

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

1. min-height — ensures a minimum height for an element's box.
2. max-height — ensures a maximum height for an element's box.

```
p {
  min-height: 150px;
  max-height: 300px;
}
```

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

What will happen to the contents of an element if the max-height property is set too low for that element? It's possible that content will overflow outside of the element, resulting in content that is not legible.

Scaling Images and Videos

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.

```
.container {
  width: 50%;
  height: 200px;
  overflow: hidden;
}

.container img {
  max-width: 100%;
  height: auto;
  display: block;
}
```

In the example above, .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.

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.

Note: The example above scales the width of an image (or video) to the width of a container. If the image is larger than the container, the vertical portion of the image will overflow and will not display. To swap this behavior, you can set max-height to 100% and width to auto(essentially swapping the values). This will scale the *height* of the image with the height of the container

instead. If the image is larger than the container, the horizontal portion of the image will overflow and not display.

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.

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

Range

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.

By using multiple widths and heights, a range can be set for a media query.

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

The example above 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.

The example above can be written using two separate rules as well:

```
@media only screen and (min-width: 320px) {
    /* ruleset for 320px - 479px */
}
@media only screen and (min-width: 480px) {
    /* ruleset for > 480px */
}
```

The first media query in the example above 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.

Dots Per Inch (DPI)

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). Learn more about resolution measurements here.

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

The media query in the example above 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.

And Operator

In previous exercises, 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`.

For example:

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

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.

Comma Separated List

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

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

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.

Breakpoints

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:

```
@media only screen and (min-width: 768px) and (max-width: 1024px) and (orientation: landscape) {
```

```
    /* CSS ruleset */
}
```

The example above creates a screen size range the size of a tablet in landscape mode and also identifies the orientation.

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. Different projects have different needs, and creating a responsive design should be no different.

Check out this list of breakpoints by device widths. Use it as a reference of screen widths to test your website to make certain it looks great across a variety of devices.
