Push your HTML and CSS skills to the limit and build professional grade, responsive websites

What you will learn from this book

- Install and use Sass with SCSS syntax to create your CSS
- Enhance the experience of users with assistive technology with ARIA roles
- Design with a desktop-first approach, but implement with mobile-first methodology
- Master your layouts by creating your own scalable CSS grid, or using Flexbox for total layout flexibility
- Retrofit your old websites with your new responsive design skills
- Deal with legacy browsers, and learn how to keep your designs consistent
- Implement widely used responsive patterns, such as the "drawer navigation", the "Off-Canvas" and the "Flexbox-based" navigation patterns
- Find the best way to serve the right type of image and how to make videos responsive
- Build custom responsive e-mails to ensure the message gets across regardless of the device

Who this book is written for

If you're already building responsive designs and are looking to level up your skills, then Mastering Responsive Web Design is the book for you. Regardless of whether you’re a web designer, or web developer, you’ll find something here to help you in your responsive design journey.

Ricardo Zea

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Mobile-first or Desktop-first?'
- A synopsis of the book’s content
- More information on Mastering Responsive Web Design
Ricardo Zea, originally from Medellín, Colombia, is a passionate and seasoned full-stack designer now located in Dayton, OH (USA). He is always looking for ways to level up his skills and of those around him. Constantly wondering how things are made on the Web, how they work, and why they work the way they do have made Ricardo a very technical designer, allowing him to explain to others the intricacies of design and the technicalities of the Web in ways that are very easy to understand and assimilate.

With a master's degree in publicity and advertising and having a deep passion for understanding human behavior complemented with a fierce competitive PC gaming hunger have allowed Ricardo to "switch" from the creative side of the brain to the rational side very easily. This has allowed him to visualize and create technically sound web and mobile designs that are responsive, perform well, and convey the proper message through design.

Ricardo is the organizer of the CodePen Dayton meetup group. He's a member of the Dayton Web Developers and UX Dayton meetup groups. He's also one of the first members of SitePoint's Ambassadors program. He is the author of the monthly Web Design & Development, Level Up! Newsletter. He was also a technical reviewer for the books Sass and Compass, Designers Cookbook and Sass Essentials by Packt Publishing. He is a real-time, live one-on-one expert advisor on https://wizpert.com/. For several years, he was also a Flash and CorelDRAW professor at different universities in his home country, Colombia.

Ricardo has 15 years of experience in web design and 20 years of experience in visual and graphic design.

You can contact Ricardo through the following links:

Personal website: http://ricardozea.design

Twitter: @ricardozea (https://twitter.com/ricardozea)

Wizpert: https://wizpert.com/ricardozea
Preface

Before the days of Responsive Web Design, web designers' and frontend developers' efforts were pretty much focused on translating print layouts into websites and applications. Elements and dimensions were rigid, and the need for them to adapt and scale wasn't part of the concepts we mandatorily have to account for today.

Devices come in many shapes and sizes. All sorts of screen densities, aiming our work for operating systems and browsers that support (or don't support) certain HTML, CSS and JavaScript technologies, UX principles, usability best practices, and knowing how mobile devices affect the world we live in are now an "intricate" part of what we do as web designers and frontend developers.

In this book, I provide a lot of information, technical and conceptual, about how RWD provides a pathway to better websites and web apps. Installing and using Sass, handling images and videos, and creating a solid typographic scale to building responsive e-mails are a few of the content gems you'll be able to read in this book.

Time to level up!

What this book covers

Chapter 1, Harness the Power of Sass for Responsive Web Design, starts with the easiest walkthrough you'll ever read on how to install Sass; then we learn how to make Sass "watch" our SCSS files. Then, there are easy-to-understand explanations about basic Sass concepts such as variables, mixins, arguments, nesting, partial files, the @import directive, source maps, and Sass comments. We also learn to automate vendor prefixing and automatically compile our SCSS files with Prepros. We talk about creating mixins to address media queries as easily as possible, considering how content defines the breakpoints.
Chapter 2, Marking Our Content with HTML5, clarifies that HTML is a markup language, not code. Then, we talk about the most used HTML5 elements that allow us to markup our content semantically. Improving the accessibility of our builds with ARIA roles is also something we address in a simple way. We also talk about the different meta tags required for RWD, and then we have an example that brings everything together.

Chapter 3, Mobile-first or Desktop-first?, demystifies why and when mobile-first or desktop-first should be used. With examples, we will learn how to retrofit a site with Adaptive Web Design, and of course, with RWD. We will learn about Respond.js and conditional classes in order to support legacy browsers when building mobile-first.

Chapter 4, CSS Grids, CSS Frameworks, UI Kits, and Flexbox for RWD, helps us understand what a grid is, how to use it, and why. With this understanding, we can make sound decisions when building a website or web app. We also create a custom CSS grid both with the floats technique and then by using Flexbox. We will use conditional classes again to address old browsers, and with the help of a small script, we can take care of the quirks from IE10 with an .ie10 specific selector.

Chapter 5, Designing Small Uls Driven by Large Finger, shows how usability and accessibility play a major role in this chapter. We also find explanations about the different sizes of our target zones, the location of our controls (links, buttons, form fields, and so on), and the touch zones in different devices. There are also three examples on how to create a menu button plus three more examples on mobile navigation patterns.

Chapter 6, Working with Images and Videos in Responsive Web Design, is one of the most interesting chapters of this book because images in RWD are a "thing". We will talk about different ways to serve different images with the picture element and the srcset attribute. Making videos responsive with CSS, jQuery, and JavaScript is also addressed in this chapter. We also learn about using vector-based files such as icon fonts and SVGs.

Chapter 7, Meaningful Typography for Responsive Web Design, talks about using relative units being ideal because they provide scalability, and this is what RWD is all about. The golden egg in this chapter is that we will learn how to use the Modular Scale to create a harmonious typographic scale. We will also use Flowtype.js to increase the legibility of our copy.

Chapter 8, Responsive E-mails, shows that e-mails are opened more on mobile devices than on desktops; that responsive e-mails have more engagement on mobile devices than nonresponsive ones; and that people click on e-mails on desktops more than on mobile. We will also create an e-mail template as an example. We'll learn to use a CSS reset block to normalize those quirky e-mail clients and learn that the best width for e-mails is not more than 600 pixels.
Mobile-first or Desktop-first?

In my years of experience designing and building responsive websites, I've found that in order to have a better view of the content and the messages, it's easier to visualize things with a desktop-first approach during the wireframe and design phases.

As we are able to see more content in a given layout with a desktop-first approach, it allows us to translate the hierarchy of the content that was provided to us into a layout that represents said hierarchy. Doing this in a small canvas of 320 pixel width is more difficult than it needs to be.

When you accomplish that hierarchy, it will remain the same on small-screen devices, and the only thing that changes is the layout.

*Best practices* recommend building mobile-first, but many web professionals don't really know why we build *mobile-first* in the first place. Pun intended.

So, let's clear it up. The reason we build mobile-first is because of three principles mentioned by Luke Wroblewski, the author who actually coined the term *mobile-first* in 2009. You will notice that none of those principles are related to HTML, CSS, and/or JavaScript. In other words, you don't build mobile-first because of any advantage with HTML, CSS, or JavaScript. For more information, visit [http://www.lukew.com/ff/entry.asp?933](http://www.lukew.com/ff/entry.asp?933).

Consider the following points:

- **Mobile is exploding**: Well, mobile has already exploded. This basically means that it's a lot easier, faster, and more convenient for people to use their mobile devices to access the web. So if you build your website to be compatible with mobile devices first, there's a better chance of providing a better user experience and being viewed by more people than if you have a desktop-only website/app.
Mobile-first or Desktop-first?

- **Mobile forces you to focus**: Since there’s a lot less space on a mobile device’s screen than on a desktop screen, there’s a mandatory need to prioritize. This means that the most important tasks and/or messages need to be visible right away.

- **Mobile extends your capabilities**: Mobile devices have features that a desktop device doesn’t have: GPS, accelerometer, multitouch inputs, voice recognition, front and rear cameras, and so on. When going mobile-first, you can use these advanced technologies to create richer, more exciting experiences.

Now that you have a final design, you now need to implement that design into HTML, CSS, and JavaScript. For this phase, you should use the mobile-first approach and take into account the three reasons we mentioned earlier:

- Building mobile-first means your website/app can be seen by more people
- It makes you prioritize the content
- If you need to, it will allow you to use the advanced features and capabilities of mobile devices

In this chapter, we will cover the following topics:

- Create your designs in a desktop-first view, but implement them with mobile-first.
- Sass mixins for mobile-first and desktop-first media queries.
- Dealing with legacy browsers.
- How to deal with high-density screens.
- Why RWD is sometimes not necessarily the right solution.
- Retrofitting an old website with RWD.

**Create your designs in a desktop-first view, but implement them with mobile-first**

Let’s look at some terminology so that we’re on the same page:

- **Wireframe**: This is a very basic visual representation of a layout using only outlines, in other words, in black and white. There are no colors, no branding, and no defined styles of any kind.

- **Design/Comp**: This is a *fleshed out* wireframe with colors, branding, and styles. It’s a very close representation (usually, say, 95 percent close to the final product) of the final page/site/app without going into markup or coding of any kind.
• **HTML mockup or HTML template**: This is when the design has been implemented into an actual HTML page with CSS and—sometimes—JavaScript. It can only be viewed in the browser. It's practically an exact representation (99 percent close to the final product) of how the page/site/web app will look and work like.

With the terminology clear, let's continue.

Some professionals, including me, recommend using more modern and efficient techniques to create visual assets in order to optimize the time spent during the process of wireframing and creating designs/comps. Techniques such as style tiles, mood boards, element collages, and atomic design differentiate themselves from traditional wireframing and designs/comps methodologies. They offer the opportunity to explore layouts and styles, independent of screen widths, technologies, and even content creation.

For the scope of this book, we are going to focus on how a few things of the traditional wireframing and designs/comps methodologies can still be harnessed while maximizing the use of our time in these initial stages of mastering **Responsive Web Design (RWD)** with HTML5 and CSS3.

**Why create designs in a desktop-first view?**

The reason behind creating designs in a desktop-first view is simple: real estate (space).

As designers, we are required to reflect the hierarchy of the content in a visual way. To accomplish this, we use many design principles such as rhythm, proximity, whitespace, patterns, contrast, balance, grid, symmetry, and so on.

When the canvas on which we're creating a wireframe or design/comp is big enough to try different arrangements and layouts, we have the necessary flexibility to explore different ways that can represent the said content hierarchy.

For example, we're using a 12-column grid and the content we were provided with dictates the following content hierarchy:

- The business wants users to be able to provide their e-mail IDs to receive our newsletter.
- We want to display a **featured post** chosen by the editorial department.
Mobile-first or Desktop-first?

With the preceding content hierarchy, we can immediately start picturing different layouts to convey this hierarchy:

- For users to provide their e-mail addresses, we'll create a form with a heading, a paragraph, an input type e-mail, and a button. This form will be on the top-left corner below the header and it could have a width of three to four columns. I'm thinking that maybe four columns is too wide though, but let's wireframe it and see how that feels and what usability, accessibility, and legibility issues or benefits this could have.

- For the featured post, we'll use the remaining columns. If the e-mail form is three-column wide, we'll use the remaining nine; if the e-mail form is four-column wide, we'll just use the remaining eight columns. The featured post has a lot more content, such as the heading, author, date, category, snippet, thumbnail, and a link to the full post.

With a wide canvas in our design/wireframing application, we can play with these different approaches and eventually end with a proposed layout that's sound and represents the content hierarchy as required by the business or stakeholders.

Creating a layout like this one with a mobile-first approach with a small canvas is practically impossible. The small real estate screen is incredibly restrictive and limited. But when things start to grow, we would need to make this exploration process each time we think of a specific breakpoint.

Actually, we shouldn't be thinking about breakpoints at this point (no pun intended), because the content—not specific device widths—is what dictates where a new breakpoint needs to be added.

Once we have a layout defined that reflects the content hierarchy, we will be in a good spot because when that content is rearranged on smaller screens, no matter which widths, the hierarchy will remain intact.

**Why implement with mobile-first?**

Let's clarify a term first: *implement* means create an HTML mockup with CSS and, if necessary, JavaScript, based on a wireframe or design/comp.

The reasons mentioned at the beginning of this chapter are the answer to the question *why implement with mobile-first*? Remember: mobile is exploding (well, it already did), mobile forces you to focus and mobile extends your capabilities.
None of those reasons could be accomplished with a desktop-first implementation, except maybe (that's a huge *maybe*) the second premise.

Let's change gears and move on to a subject a bit more technical that will help us understand how Sass mixins can help us master RWD for the mobile-first and desktop-first methodologies.

So, let's recap. Use desktop-first to create your designs and wireframes. Having a large canvas allows us to explore different layouts and properly arrange the hierarchy of the content. When it's time to implement (create HTML mockups), use mobile-first.

**Sass mixins for the mobile-first and desktop-first media queries**

For our examples, there are two types of Sass mixins we're going to use in this book: a mobile-first mixin that uses the `min-width` property and a desktop-first mixin that uses the `max-width` property. We already saw the following mixins and how they worked in *Chapter 1, Harness the Power of Sass for Responsive Web Design*, but here's a refresher.

**The mobile-first mixin**

We're going to use the following mobile-first mixin:

```scss
@mixin forLargeScreens($media) {
  @media (min-width: $media/16+em) { @content; }
}
```

This is how we use it:

```scss
header {
  //Properties for small screens
  width: 50%;
  background: red;
  @include forLargeScreens(640) {
    //Properties for large screens
    width: 100%;
    background: blue;
  }
}
```
Mobile-first or Desktop-first?

This compiles to the following:

```css
header {
  width: 50%;
  background: red;
}

@media (min-width: 40em) {
  header {
    width: 100%;
    background: blue;
  }
}
```

The desktop-first mixin

Here's the desktop-first mixin we're going to use:

```css
@mixin forSmallScreens($media) {
  @media (max-width: $media/16+em) { @content; }
}
```

This is how we use it:

```css
header {
  //Properties for large screens
  width: 100%;
  background: purple;
  @include forSmallScreens(640) {
    //Properties for small screens
    width: 50%;
    background: yellow;
  }
}
```

@include forSmallScreens
This compiles to the following:

```css
header {
  width: 100%;
  background: purple;
}

@media (max-width: 40em) {
  header {
    width: 50%;
    background: yellow;
  }
}
```

The great thing about using these mixins is that it's incredibly easy to find out which approach is being used, because we can see either the term `forLargeScreens` or `forSmallScreens` is repeated all over our SCSS file. If someone else were to edit any of the work we initially did, they will get a clear idea of which approach we used to build our site/app just by scanning the SCSS file.

**Dealing with legacy browsers**

Within the question "mobile-first or desktop-first?" there's an area that we need to cover about legacy browsers. Each project, each client, and their corresponding analytics (if they have any, which they should) have different requirements that affect how we are supposed to deal with those older browsers.

If you're building with a desktop-first approach, your current workflow should remain the same as this is pretty much what we've been doing since before RWD became practically mandatory.
Mobile-first or Desktop-first?

This means that you would still use something like this:

```css
header {
    // Desktop-first declaration
    width: 50%;
    @include forSmallScreens(768) {
        // Target small screens (mobile devices)
        width: 100%;
    }
}
```

This compiles to the following:

```css
header {
    width: 50%;
}

@media (max-width: 48em) {
    header {
        width: 100%;
    }
}
```

IE7 and IE8 do not support media queries, but the preceding code will work just fine because the `header { width: 50%; }` rule is not inside a media query.

However, if you’re doing mobile-first, then `header { width: 50%; }` is going to be inside a media query so that IE7 and IE8 won’t be able to see that rule:

```css
.article {
    // Mobile-first declaration
    width: 100%;
    // IE7 and IE8 won’t be able to see this rule.
    @include forLargeScreens(768) {
        width: 50%;
    }
}
```

This compiles to the following:

```css
header {
    width: 100%;
}

@media (min-width: 48em) {
    header {
        width: 50%;
    }
}
```

What do you do then? The solution is quite simple: use the `Respond.js` script.
How to use Respond.js for RWD

Respond.js is a type of script called a polyfill. A polyfill, according to the one who coined the term in the first place, Remy Sharp, is a piece of code that provides the technology that we, web developers, expect browsers to provide natively.

In web design and development, polyfills are more abundant as JavaScript implementations, in our case, Scott Jehl's Respond.js. But we could also say that there are polyfills in CSS too, for example, the well-known reset.css from Eric Meyer and Normalize.css from Nicolas Gallagher and Jonathan Neal.

The Respond.js script is a polyfill that makes legacy browsers (IE6/7/8) support a particular CSS feature they were never made to support: media queries.

You can download Respond.js from https://github.com/scottjehl/Respond.

Although I'm suggesting the use of a polyfill, we need to be mindful of the additional HTTP request the site/app needs to make in order to fetch this JavaScript file. The fewer requests our sites/apps make, the faster they are going to be creating many benefits such as improved user experience and positive SEO impact.

So, here's what you need to do:

• Make sure the call to Respond.js is after the call to your CSS file(s) (hopefully it is just one CSS file).
• Call the Respond.js script.

Performance best practices recommend placing nonessential scripts at the bottom of the markup right before the closing </body> tag. Since Respond.js is aimed at legacy browsers, let's go ahead and do that. Another benefit of placing scripts at the bottom of the markup is that it helps to avoid blocking the rendering of the page.

Here's our example HTML:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Mastering RWD with HTML5 &amp; CSS3</title>
  <link href="styles.css" rel="stylesheet">
</head>
```
Mobile-first or Desktop-first?

```html
<body>
  <header>Logo goes here…</header>
  <article>Content goes here…</article>
  <script src="js/respond.min.js"></script>
</body>
</html>
```

In our `styles.scss` file, we type the following lines:

```scss
//Mobile-first declaration
article { background: red;
  //Target screens 640px wide and larger
  @include forLargeScreens(640) {
    & { background: green; }
  }
}
```

This compiles to the following:

```css
article {
  background: red;
}

@media (min-width: 40em) {
  article {
    background: green;
  }
}
```

So, when you resize an IE7 or IE8 browser window, it will be able to display a red background if the window width is 640 pixels or less, and a green background if the window is 641 pixels or more.

The days of an IE-specific style sheet are dead

I've avoided creating IE-specific style sheets since I started writing CSS. The reasons for this are simple:

- **File management**: The fewer files there are to manage when going to production, the smoother every process goes; not to mention being less prone to errors.
- **Scalability**: If you need to add, remove, or edit a style, you and your team know that the final change(s) needs to end up in your main and only CSS file, in our case, the SCSS file.
Chapter 3

- **Organization**: Keep everyone on the same page when adding, removing, or editing IE-specific styles in the right CSS file(s), in our case, SCSS file(s).

- **Performance**: One less HTTP request is a good thing, a very good thing. Anything we can do for performance, no matter how small, can go a long way for a good user experience; not to mention a fast website is good for SEO.

### Other benefits of not using an IE-specific style sheet

In legacy browsers, page rendering is not blocked when they try to download the IE-specific style sheet. Also, troubleshooting is easier. So what do we use then?

There are several ways to deal with IE by keeping everything in one style sheet:

- Use CSS hacks (not recommended).
- Use Modernizr.js.
- Use conditional classes in the `<html>` tag.

Let's talk a bit more about a popular method, using conditional classes.

### Use conditional classes in the `<html>` tag


- It's easy to implement; it's just a matter of copying and pasting this block of markup at the top of our HTML file.
- It's not intrusive, since there's no need for anyone in the chain (users, browsers, servers, and us) to deal with additional files.
- It doesn't require JavaScript to work; if a visitor has JavaScript unavailable or disabled, everything will still work.

This is the one I use:

```html
<!-[if IE 8]> <html class="no-js ie8" lang="en"> <![endif]-->
<!-[if IE 9]> <html class="no-js ie9" lang="en"> <![endif]-->
<!-[if gt IE 9]><!-[endif]--><html class="no-js" lang="en">
<!-[endif]-->
```
IE10 and above does not support *conditional comments* anymore, that's why there isn't any mention of IE10 in the conditional classes markup.

With the preceding conditional classes in place, targeting a specific IE (IE7 in this example) looks like this:

```
.ie7 nav li {
    float: left;
}
```

If we need to target all IEs, we would do this:

```
.ie7, .ie8, .ie9 {
    nav li {
        float: left;
    }
}
```

This compiles to the following:

```
.ie7 nav li,
.ie8 nav li,
.ie9 nav li {
    float: left;
}
```

For all other browsers, we would do this:

```
nav {
    display: flex;
}
```

It doesn't matter which of the methods you use, Modernizr.js or conditional classes, it's all personal preference. You'll be doing *the right thing* by using either of those two methods.

Remember, avoid CSS hacks at all costs. As web designers and web developers, we have a moral responsibility of creating a better web for everyone.

**How to deal with high-density screens**

There are many articles on the Web that explain what Dots Per Inch (DPI), Pixels Per Inch (PPI), and Density-independent Pixel (DP/DiP) are. Although it may be important to understand the intricate details of such technologies and terms, let's keep the scope of the book in the realms of what the basis of high density screens is and what we need to understand to create sound responsive designs.
Bitmaps or vectors for high-density screens?

Vectors like SVGs, Icon Fonts, or regular fonts are a visual representation of mathematical equations, thus they never lose quality, no matter their size.

In order for bitmap images to display well on high-density screens, we have to export a high-resolution version of the normal-quality image. This means that we need to create two files (or more) for every bitmap image we plan to use: one normal-quality image for non-high-density screens (standard LCD displays, old TFT monitors, some TVs, and so on) and one (or more) high-quality image for high-density screens (any retina devices and Super AMOLED displays, for example).

This is where good design judgment comes into play, because sometimes we may not necessarily need to export two (or more) bitmap images every time.

There are several techniques that we can use to deal with images when we have to consider high-density screens. These techniques are explained in detail in Chapter 6, Working with Images and Videos in Responsive Web Design.

Sometimes RWD is not necessarily the right solution

Take, for example, the booking section of most travel sites. The sheer amount and type of information a site like this manages makes it quite difficult to have a responsive site. When visiting the eight highest ranked travel sites in Google's search results, this is what I saw:

  - **Homepage**: Responsive
  - **Booking page**: Not responsive

  - **Homepage**: Responsive
  - **Booking page**: Responsive

- [https://www.hotwire.com/](https://www.hotwire.com/)
  - **Homepage**: Not responsive
  - **Booking page**: Responsive
Mobile-first or Desktop-first?

- http://www.travelocity.com/
  - **Homepage**: Responsive
  - **Booking page**: Responsive
- http://www.orbitz.com/
  - **Homepage**: Not responsive
  - **Booking page**: Not responsive
- http://www.priceline.com/
  - **Homepage**: Not responsive
  - **Booking page**: Not responsive
- http://www.tripadvisor.in/
  - **Homepage**: Not responsive
  - **Booking page**: Not responsive
- https://www.hipmunk.com/
  - **Homepage**: Not responsive
  - **Booking page**: Not responsive

Here is a brief list of our findings:

- Since Expedia acquired Travelocity, they share the same platform. The difference is in the branding; thus, I will consider these two sites as one.
- The homepages of five out of seven sites (71 percent) are not responsive.
- The booking pages of five out of seven sites (71 percent) are not responsive.
- Only one site (Expedia/Travelocity) out of seven (14 percent) is fully responsive.
- Four out of seven sites (57 percent) have no RWD whatsoever.

We can conclude that the most popular travel sites have not fully embraced RWD yet, but some are hybrids between fixed width and responsive layouts. That's why all of those sites have separate mobile apps. For them, RWD may not be a priority, so they rely on their mobile apps to balance this deficiency.

Although very rare these days, sometimes we may need to build a site or page that is not responsive. Actually, there are some pages out there today that are not responsive.
Chapter 3

CodePen is one of the most popular frontend sandboxes out there and the editor of CodePen is not responsive. Why? Because it doesn't need to be. It's very unlikely that a developer would go to CodePen to write HTML, Sass, and JavaScript using their phone.

With that being said, if you ever need to build a site/page that doesn't need to be responsive, there are two good options as far as CSS grid systems go:

- Use our old friend, the 960 Grid System (http://960.gs/).
- Use the 1140 Grid System (http://www.1140px.com/).

There are a few things to consider:

- The 960 Grid System is aimed at screens 1024px wide.
- The 1140 Grid System is aimed at screens 1280px wide.
- The 1140 Grid System includes media queries by default, so we need to take this into account and decide whether it's best to leave them or if it's best to delete them to reduce file size and selector limitations in IE6-IE9.

Because I always thought that the 10px padding on the left and right of the 960 Grid System left the content too close to the edges of the main container, I added 10 more pixels to each side, increasing the padding to 20px — turning the 960 Grid System into a 980 Grid System. From now on, we will refer to it as the 980GS.

Retrofitting an old website with RWD

If and when the moment comes, we need to be prepared to make a nonresponsive or fixed-width site/app responsive.

There are two ways of retrofitting a nonresponsive or fixed-width site/app. One way is using the Adaptive Web Design (AWD) technique that uses absolute units (that is, pixels). The other way is using RWD and transforming all pixel values to percentages with a very simple formula.

Regardless of which techniques we use, we are going to have to use a desktop-first approach since the site we're dealing with was built for wide screens only. This means that we're going to use the max-width property in our media queries.

Before we look at both retrofitting techniques, we need a base page to start with.
Mobile-first or Desktop-first?

The base page

The graphic you see here is proportional to a 12-column 980GS layout. The browser window is 1024px wide and the page is 980px wide:

Our main container in gray, which is 980px wide, already has 10px padding to the left and right. This means that the sections inside always need to add up to 960px.

The following are the container's components:

- The main container in gray is 980px wide with 10px padding on the left and right.
- The **Header** in green and **Footer** in red are 960px or 12-column wide each: 940px with a 10px margin on the left and right.
- The **Nav** section in blue is 240px or 3-column wide: 220px with 10px left margin and right margins.
- The **Content** section in yellow is 710px or 9-column wide: 700px with 10px right margin.
• The gutter in white is 20px wide, that is, a 10px right margin from **Nav** and a 10px left margin from **Content**.
• So, 220px **Nav** + 710px **Content** + 20px gutter + 10px margins = 960px.

**HTML**

Here's the markup that represents our base page:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Retrofitting with Adaptive Web Design</title>
  <link href="css/styles.css" rel="stylesheet">
</head>
<body>
<main class="container_12 clear">
  <header class="grid_12">Header</header>
  <nav class="grid_3">Nav</nav>
  <section class="grid_9">Content</section>
  <footer class="grid_12">Footer</footer>
</main>
</body>
</html>
```

**CSS/SCSS**

Regarding our CSS/SCSS, we are only going to need to create one partial, the `_980gs.scss` file that contains the fixed-width grid.

Then, we're going to create a `styles.scss` file with which we are going to perform the following operations:

• Import the `_980gs.scss` file.
• Include our simple desktop-first Sass mixin to handle the media queries.
• Create all the necessary media queries using the `max-width` property.
• Compile it to `styles.css` and use it in our page.
Creating the _980gs.scss file

The _980gs.scss file contains the basic grid and looks like this:

```css
//Globals
*, *:before, *:after {
    box-sizing: border-box;
}

//Container
.container_12 {
    width: 980px;
    padding: 0 10px;
    margin: auto;
}

//Grid >> Global
.grid {
    &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10,
    &_11, &_12 {
        float: left;
        margin: 0 10px;
    }
}

//Grid >> 12 Columns
.container_12 {
    .grid_1 { width: 60px; }
    .grid_2 { width: 140px; }
    .grid_3 { width: 220px; }
    .grid_4 { width: 300px; }
    .grid_5 { width: 380px; }
    .grid_6 { width: 460px; }
    .grid_7 { width: 540px; }
    .grid_8 { width: 620px; }
    .grid_9 { width: 700px; }
    .grid_10 { width: 780px; }
    .grid_11 { width: 860px; }
    .grid_12 { width: 940px; }
}

//Clear Floated Elements - http://davidwalsh.name/css-clear-fix
.clear, .row {
    &:before,
...
Retrofitting with AWD

Unlike RWD where the widths are fluid and elastic (ems and percentages), hence the term relative units, in AWD, the widths are fixed (pixels). Hence, we use the term absolute units and elements will snap to these fixed widths when we resize our browser window.

In AWD, we use pixels for practically every width, even our media queries.

Creating the styles.scss file

The first thing we’re going to do in the styles.scss file is to import the partial _980gs.scss file:

```scss
//Retrofitting with Adaptive Web Design
@import "980gs";
```

Then, we’re going to include our simple desktop-first mixin to handle the media queries. However, remember I mentioned before how this is mixin is scalable and we could make it compile pixel-based values if we wanted to? All we need to do is remove the value /16+em from the division $media/16+em:

```scss
//Retrofitting with Adaptive Web Design
@import "980gs";

//Desktop-first Media Query Mixin
@mixin forSmallScreens($media) {
  @media (max-width: $media) { @content; }
}
```
Mobile-first or Desktop-first?

The following rules are merely for styling purposes in order to accomplish the same design we saw in the screenshot before:

```css
//Retrofitting with Adaptive Web Design
@import "980gs";

//Desktop-first Media Query Mixin
@mixin forSmallScreens($media) {
  @media (max-width: $media) { @content; }
}

//Basic styling
.container_12 {
  background: #aaa;
  font-size: 30px;
  text-shadow: 0 1px 1px rgba(black,.5);
}
header { background: #429032; }
nav { background: #2963BD; }
section { background: #c90; }
footer { background: #c03; }

//Give heights to elements for better perception of sections
header, footer { height: 150px; }
nav, section { height: 440px; }
```

At this point, our page is 980px wide and it looks like the screenshot we initially saw.

Let's define the widths at which we are going to make our base page snap to:

- At 980px, we're going to snap the page to 768px.
- At 768px, we're going to snap the page to 640px.
- At 640px, we're going to snap the page to 480px.
- At 480px, we're going to snap the page to 320px.

This is where the fun begins. Let's start retrofitting this page by creating the media queries for each section.

### 980px to 768px (AWD)

The following media queries are aimed at 768px:

```css
.container_12 {
  @include forSmallScreens(980px) {
    width: 768px;
  }
}
.grid_12 { //Header and Footer sections
  @include forSmallScreens(980px) {
```
Admittedly, it is a bit hard to perceive the difference in the book from 980px to 768px, but believe me, the following screenshot fully represents a browser window 980px wide and a page 768px wide:
As you can see, the moment the screen is 980px, the width of our main container (.container_12) goes from 980px to 768px. Our main container has 10px padding to the left and the right, so the widths of all other sections should add up to match 748px.

Let's take a look.

Our Header and Footer that use the same class .grid_12 are now 728px wide. So if we add: 728px + 10px left margin + 10px right margin = 748px.

If we add the widths of our Nav (.grid_3) and Content (.grid_9) sections:

- 200px Nav + 508px Content = 708px
- 708px + 20px gutter = 728px
- 728px + 10px left margin on Nav + 10px right margin on Content = 748px

Stay with me, I promise this will be very interesting.

768px to 640px (AWD)

The following media queries are aimed at 640px:

```
.container_12 {
    @include forSmallScreens(980px) {
        width: 768px;
    }
    @include forSmallScreens(768px) {
        width: 640px;
    }
    .grid_12 { //Header and Footer sections
        @include forSmallScreens(980px) {
            width: 728px;
        }
        @include forSmallScreens(768px) {
            width: 600px;
        }
    }
    .grid_3 { //Nav section
        @include forSmallScreens(980px) {
            width: 200px;
        }
        @include forSmallScreens(768px) {
            width: 160px;
        }
    }
```
Ok, this layout is now a single column page. We’re starting to see some results. Nice!

Again, remember that our main container has 10px padding to the left and the right, thus the widths of all other sections should add up to match 620px.
Mobile-first or Desktop-first?

Let's make sure our numbers add up:

Our **Header** and **Footer** that use the same class `.grid_12` are now 600px wide. So if we add: 600px + 10px left margin + 10px right margin = 620px.

If we add the widths of our **Nav** (.grid_3) and **Content** (.grid_9) sections:

- 160px Nav + 420px Content = 580px
- 580px + 20px gutter = 600px
- 600px + 10px left margin on Nav + 10px right margin on Content = 620px

Let's make this page even smaller!

### 640px to 480px (AWD)

The following media queries are aimed at 480px:

```css
.container_12 {

  @include forSmallScreens(980px) {
    width: 768px;
  }
  @include forSmallScreens(768px) {
    width: 640px;
  }
  @include forSmallScreens(640px) {
    width: 480px;
  }
}
.grid_12 { //Header and Footer sections
  @include forSmallScreens(980px) {
    width: 728px;
  }
  @include forSmallScreens(768px) {
    width: 600px;
  }
}
.grid_3 { //Nav section
  @include forSmallScreens(980px) {
    width: 200px;
  }
  @include forSmallScreens(768px) {
    width: 160px;
  }
}
.grid_9 { //Content section
  @include forSmallScreens(980px) {
    width: 508px;
  }
  @include forSmallScreens(768px) {
```
We're making some well-deserved progress! Here, the browser window is 640px wide and the page is 480px wide:

![Diagram showing section widths]

Remember that our main container has 10px padding to the left and the right, thus the widths of all other sections should add up to match 460px.
Mobile-first or Desktop-first?

Now, we are going to change from a 2-column to a 1-column layout. This means that all sections now have the exact same width.

This also means that in our SCSS file, we can create a single media block for all three classes:

```
.grid_3, 
.grid_9, 
.grid_12 { 
  @include forSmallScreens(640px) { 
    width: 440px; 
  }
}
```

Now, let's make sure our numbers add up:

Our Header, Nav, Content, and Footer sections are now 440px wide, stacked one on top of the other. So if we add: 440px of all sections + 10px left margin + 10px right margin = 460px.

Here we go, the last piece of this puzzle!

480px to 320px (AWD)
The following media queries are aimed at 320px:

```
.container_12 { 
  @include forSmallScreens(980px) { 
    width: 768px; 
  }
  @include forSmallScreens(768px) { 
    width: 640px; 
  }
  @include forSmallScreens(640px) { 
    width: 480px; 
  }
  @include forSmallScreens(480px) { 
    width: 320px; 
    padding: 0; 
  }
}
```
.grid_12 { //Header and Footer sections
  @include forSmallScreens(980px) {
    width: 728px;
  }
  @include forSmallScreens(768px) {
    width: 600px;
  }
}
.grid_3 { //Nav section
  @include forSmallScreens(980px) {
    width: 200px;
  }
  @include forSmallScreens(768px) {
    width: 160px;
  }
  @include forSmallScreens(640px) {
    height: 50px; //This is only for styling
  }
}
.grid_9 { //Content section
  @include forSmallScreens(980px) {
    width: 508px;
  }
  @include forSmallScreens(768px) {
    width: 420px;
  }
}
.grid_3,
.grid_9,
.grid_12 {
  @include forSmallScreens(640px) {
    width: 440px;
  }
  @include forSmallScreens(480px) {
    width: 300px;
  }
}
Mobile-first or Desktop-first?

There we go! In this screenshot, the browser window is 320px wide—the content is 320px wide as well and fits very nicely:

![Diagram of a responsive layout with main container, header, navigation, content, and footer.]

We already know that our main container has 10px padding to the left and the right. In this case, we are going to remove that padding to gain those 20 pixels, since our screen real estate is now very small:

```css
@include forSmallScreens(480px) {
    width: 320px;
    padding: 0;
}
```
The 10px spacing on the left and right are now going to be created by the left and right margins from the other sections. This means that the width of each section should be 300px.

Adding the new 320px breakpoint is easy:

```scss
.grid_3, .grid_9, .grid_12 { 
  @include forSmallScreens(640px) { 
    width: 440px; 
  } 
  @include forSmallScreens(480px) { 
    width: 300px; 
  } 
}
```

Now, let's make sure our numbers add up:

Our **Header**, **Nav**, **Content**, and **Footer** sections are now 300px wide, stacked one on top of the other. So if we add: 300px of all sections + 10px left margin + 10px right margin = 320px.

That's it. We have now retrofitted a fixed-width page to be responsive with the AWD technique.

The final SCSS is as follows:

```scss
.container_12 { 
  @include forSmallScreens(980px) { 
    width: 768px; 
  } 
  @include forSmallScreens(768px) { 
    width: 640px; 
  } 
  @include forSmallScreens(640px) { 
    width: 480px; 
  } 
  @include forSmallScreens(480px) { 
    width: 320px; padding: 0; 
  } 
}
.grid_12 { //Header and Footer sections 
  @include forSmallScreens(980px) { 
    width: 728px; 
  } 
}```
Mobile-first or Desktop-first?

```css
@include forSmallScreens(768px) {
  width: 600px;
}
.grid_3 { //Nav section
  @include forSmallScreens(980px) {
    width: 200px;
  }
  @include forSmallScreens(768px) {
    width: 160px;
  }
  @include forSmallScreens(640px) {
    height: 50px; //This is only for styling
  }
}
.grid_9 { //Content section
  @include forSmallScreens(980px) {
    width: 508px;
  }
  @include forSmallScreens(768px) {
    width: 420px;
  }
}
.grid_3, .grid_9, .grid_12 {
  @include forSmallScreens(640px) {
    width: 440px;
  }
  @include forSmallScreens(480px) {
    width: 300px;
  }
}

It compiles to the following CSS:

```css
@media (max-width: 980px) {
  .container_12 {
    width: 768px;
  }
}
@media (max-width: 768px) {
  .container_12 {
    width: 640px;
  }
}
```
Chapter 3

@media (max-width: 640px) {
    .container_12 {
        width: 480px;
    }
}
@media (max-width: 480px) {
    .container_12 {
        width: 320px;
        padding: 0;
    }
}
@media (max-width: 980px) {
    .container_12 .grid_12 {
        width: 728px;
    }
}
@media (max-width: 768px) {
    .container_12 .grid_12 {
        width: 600px;
    }
}
@media (max-width: 980px) {
    .container_12 .grid_3 {
        width: 200px;
    }
}
@media (max-width: 768px) {
    .container_12 .grid_3 {
        width: 160px;
    }
}
@media (max-width: 640px) {
    .container_12 .grid_3 {
        height: 50px;
    }
}
@media (max-width: 980px) {
    .container_12 .grid_9 {
        width: 508px;
    }
}
@media (max-width: 768px) {
    .container_12 .grid_9 {
        width: 420px;
    }
Mobile-first or Desktop-first?

As you can see, several breakpoints are repeated in our final CSS file. This is an issue with Sass. However, it's really not an issue or something we need to worry about because when this file is gzipped by the server, it will compress it at its maximum. If we minimize the final output (which we should anyhow), we’ll be compressing the file even more. The repeated @media breakpoints have very little if any impact on performance.

Now, let's see how retrofitting the same page looks when using percentages and RWD.

Retrofitting with RWD

We just saw how using AWD is accomplished, using pixels. With RWD and a very simple equation, we can retrofit a site using relative units, in our case percentages. Not to mention it will be a lot easier than using AWD.

The RWD magic formula

Discovered/created by Ethan Marcotte, who coined the term Responsive Web Design, the RWD magic formula is a very simple equation:

\[(\text{target} \div \text{context}) \times 100 = \text{result} \%\]

Before we start turning pixels into percentages, we need to see which width our context is going to be.
The main container

Our context is going to be the main container of the page .container_12, which has a maximum width of 980px. However, there’s a catch involving the main container and the columns that will turn this 980px context into 960px. Notice the 10px left-right padding on the .container_12 section and the 10px left-right margin in the .grid rules:

```css
.container_12 {
  width: 980px;
  padding: 0 10px;
  margin: auto;
}
.grid {
  &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10, &_11, &_12 {
    float: left;
    margin: 0 10px;
  }
}
```

The 10px left-right margin in the .grid rule means that the widths of all the columns have an additional 20px. So, for example, the header and footer that are 940px wide are really 960px wide. The box-sizing: border-box; property only accounts for subtracting what’s inside the box model (padding), not what’s outside (margin).

One solution would be to remove the 10px left-right padding on .container_12 and increase the left-right margin in the .grid rule to 20px in order to keep the gutters; otherwise, the columns would touch.

Now, the gutters become wider and this may not be intended for design purposes, and—believe it or not—somehow an extra 1px is added to the widest containers. In our case, it is added to the header and footer.

As a designer, I know I do not want to deal with any of those issues if I don’t have to.

The second solution is simpler: make the context 960px. This way, we can remove the 10 extra pixels globally without affecting the integrity of the main container and the columns, and the resulting values are almost the same since we’re getting percentages.

In other words: \((960\text{px} + 980\text{px}) \times 100 = 97.95918367346939\%\) (97.95%)

It’s practically the same as: \((940\text{px} + 960\text{px}) \times 100 = 97.916666666666667\%\) (97.91%)
In the second solution, the 1px issue does happen, but happens at random widths when resizing the browser. However, the 1px issue is permanent with the first solution, regardless of the browser's width.

With this part clear, we are then going to turn all pixel-based widths into percentages using 960px as their context.

**The Header and Footer sections**
Both the **Header** and **Footer** sections have the same width, 940px. Knowing that their context is 960px, let's go ahead and find their widths in percentages using the magic formula: \((940\text{px} ÷ 960\text{px}) \times 100 = 97.91666666666667\%\).

You might be asking yourselves, "are that many decimals necessary?" Not all of them, but at least two are recommended.

So we end up with the **Header** and **Footer** sections of 97.91 percent.

Some developers recommend using all the decimals and letting the browser decide how many it wants to use. In the past, I decided to challenge this recommendation and use only two decimals to see what happened. Since I started using two decimals, I haven't experienced any unwanted behavior or width issues whatsoever in any browser.

Firefox and IE11 trim the excess decimals to two. Chrome, on the other hand, leaves all the decimals. I recommend using no less than two decimals, which is what we're going to use in the book to keep things simple and short. However, if you prefer to use all the decimals, by all means, go for it! At this point, it is a matter of personal preference.

Avoid rounding up the values, and let the browsers deal with the decimals as they are. Doing this also keeps you focused on what matters most: being efficient and trying to create something memorable for the users.

**The Nav section**
To find the width of the **Nav** section in percentages, we use 960px as its context as well: \((220\text{px} ÷ 960\text{px}) \times 100 = 22.91666666666667\%\).

Using two decimals, we end up with a **Nav** section of 22.91 percent.
The Content section

To find out the width of the Content section in percentages, our formula looks almost identical. The only difference is that we are changing the first value which corresponds to the width of the Content section in pixels:

\[(700px + 960px) \times 100 = 72.91666666666667\%\].

Using only two decimals, our final value is a Content section of 72.91 percent.

This is what our initial retrofitting RWD SCSS file starts to look like:

```scss
.container_12 {
  .grid_12 { //Header and Footer sections
    width: 97.91%;
  }
  .grid_3 { //Nav section
    width: 22.91%;
  }
  .grid_9 { //Content section
    width: 72.91%;
  }
}
```

Now, let's take a step back and address a few other pixel-based widths before we continue. Remember the 10px padding to the left and the right of the main container .container_12? We need to turn those 10px into percentages as well.

With our magic formula, we do it like this:

\[(10px + 960px) \times 100 = 1.041666666666667\%\].

Using only two decimals, our final value is a left and right padding of 1.04 percent.

Let's add this value to our SCSS:

```scss
.container_12 {
  width: 980px;
  padding: 0 1.04%;
  margin: auto;
}
```

Now, let's take a step back and address a few other pixel-based widths before we continue. Remember the 10px padding to the left and the right of the main container .container_12? We need to turn those 10px into percentages as well.

With our magic formula, we do it like this:

\[(10px + 960px) \times 100 = 1.041666666666667\%\].

Using only two decimals, our final value is a left and right padding of 1.04 percent.
Mobile-first or Desktop-first?

Also, all our columns have a 10px margin to the left and right. Since we already know that 10px is 1.04 percent, let's add this value to all our columns in our SCSS:

```
.container_12 {
  width: 980px;
  padding: 0 1.04%;
  margin: auto;
}
.grid {
  &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10, &_11, &_12 {
    float: left;
    margin: 0 1.04%;
  }
}
```

Now, we have a browser window 1024px wide, a layout 980px wide, and all the columns at their corresponding percentage values. In reality, this is practically impossible without looking at the code to visually tell the differences between the fixed width and the percentage-based layouts.

We're doing good here!
Let the fun begin. Let's add our first media query.

980px to 768px (RWD)

The following media query is aimed at 768px:

```css
.container_12 {
    width: 980px;
    padding: 0 1.04%;
    margin: auto;
}
.grid {
    &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10,
    &_11, &_12 {
        float: left;
        margin: 0 1.04%;
    }
}
```
Since the widths of the **Header**, **Footer**, **Nav**, and **Content** sections, their paddings, and their margins are set in percentages now, we don't have to declare any media queries for them—at least not yet because the layout hasn't changed.

When we resize our browser window, the **Header**, **Footer**, **Nav**, and **Content** sections automatically respond, shrink proportionally, snap properly, and fit the new width of the main container `.container_12` without breaking the layout. This is shown in the following screenshot:
This is awesome!

Let's add another breakpoint.

### 768px to 640px (RWD)

In the following breakpoint (640px), our layout is going to change to a single column. So we are going to add a new media query that will make the **Nav** and **Content** sections as wide as the **Header** and **Footer** sections, and make them stack on top of each other.

The following media query is aimed at 640px and makes the **Nav** and **Content** sections full width:

```css
.container_12 {
  width: 980px;
  padding: 0 1.04%;
  margin: auto;
}
.grid {
  _1, _2, _3, _4, _5, _6, _7, _8, _9, _10,
  _11, _12 {
    float: left;
    margin: 0 1.04%;
  }
}
.container_12 {
  @include forSmallScreens(980px) {
    width: 768px;
  }
  @include forSmallScreens(768px) {
    width: 640px;
  }
  .grid_12 { //Header and Footer sections
    width: 97.91%;
  }
  .grid_3 { //Nav section
    width: 22.91%;
  }
  .grid_9 { //Content section
    width: 72.91%;
  }
}
Mobile-first or Desktop-first?

Ok, we now have a single-column layout. Not bad, not bad!

640px to 480px (RWD)

We are now going as small as 480px and the one-column layout won't change, only the widths of all the containers will change.

The following media query is aimed at 480px:

```css
.container_12 {
  width: 980px;
  padding: 0 1.04%;
}
```
margin: auto;
}
.grid {
  &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10,
  &_11, &_12 {
    float: left;
    margin: 0 1.04%;
  }
}
.container_12 {
  @include forSmallScreens(980px) {
    width: 768px;
  }
  @include forSmallScreens(768px) {
    width: 640px;
  }
  @include forSmallScreens(640px) {
    width: 480px;
  }
  .grid_12 { //Header and Footer sections
    width: 97.91%;
  }
  .grid_3 { //Nav section
    width: 22.91%;
    @include forSmallScreens(640px) {
      height: 50px; //This is only for styling
    }
  }
  .grid_9 { //Content section
    width: 72.91%;
  }
  .grid_3, .grid_9 { //Nav and Content sections
    @include forSmallScreens(640px) {
      width: 97.91%;
    }
  }
}
Our layout is getting narrower and all we needed to do was add a new media query and that was it! No need to mess around with the other containers; they all adapt perfectly to any width we define.

480px to 320px (RWD)

Finally, we address the 320px width without modifying the one-column layout. We remove the padding on `.container_12` to make use of all the available screen real estate.

The following media query is aimed at 320px:

```css
.container_12 {
    width: 980px;
    padding: 0 1.04%;
    margin: auto;
}
.grid {
```
&_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10,
&_11, &_12 { 
  float: left;
  margin: 0 1.04%;
}
.container_12 { 
  @include forSmallScreens(980px) { 
    width: 768px;
  }
  @include forSmallScreens(768px) { 
    width: 640px;
  }
  @include forSmallScreens(640px) { 
    width: 480px;
  }
  @include forSmallScreens(480px) { 
    width: 320px; padding: 0;
  }
.grid_12 { //Header and Footer sections
    width: 97.91%;
  }
.grid_3 { //Nav section
    width: 22.91%;
    @include forSmallScreens(640px) {
      height: 50px; //This is only for styling
    }
  }
.grid_9 { //Content section
    width: 72.91%;
  }
.grid_3, .grid_9 { 
    @include forSmallScreens(640px) {
      width: 97.91%;
    }
  }
}
Mobile-first or Desktop-first?

Once more, we do not have to add anything to the Header, Footer, Nav, and Content sections, since all of them are now 97.91 percent wide. This makes them responsive and we don't have to worry about anything else.

The final SCSS, combining all breakpoints and widths, is as follows:

```css
.container_12 {
  width: 980px;
  padding: 0 1.04%;
  margin: auto;
}
.grid {
  &_1, &_2, &_3, &_4, &_5, &_6, &_7, &_8, &_9, &_10,
  &_11, &_12 {
    float: left;
    margin: 0 1.04%;
  }
}
.container_12 {
  @include forSmallScreens(980px) {
...
It compiles to the following CSS:

```css
.container_12 {
    width: 980px;
    padding: 0 1.04%;
    margin: auto;
}

.grid_1, .grid_2, .grid_3, .grid_4, .grid_5, .grid_6, .grid_7, .grid_8, .grid_9, .grid_10, .grid_11, .grid_12 {
    float: left;
}
```
Mobile-first or Desktop-first?

```css
margin: 0 1.04%;
}
@media (max-width: 980px) {
  .container_12 {
    width: 768px;
  }
}
@media (max-width: 768px) {
  .container_12 {
    width: 640px;
  }
}
@media (max-width: 640px) {
  .container_12 {
    width: 480px;
  }
}
@media (max-width: 480px) {
  .container_12 {
    width: 320px;
    padding: 0;
  }
}
.container_12 .grid_12 {
  width: 97.91%;
}
.container_12 .grid_3 {
  width: 22.91%;
}
.container_12 .grid_9 {
  width: 72.91%;
}
@media (max-width: 640px) {
  .container_12 .grid_3, .container_12 .grid_9 {
    width: 97.91%;
  }
}
```

As you can see, it's a lot less code using RWD than AWD to retrofit a site. Granted, these examples are an extreme simplification of a site/app layout, but now you are aware of the basic concepts of each technique when the time to make the call of using AWD or RWD knocks on your door.
Summary

We discussed a lot of interesting stuff in this chapter, for sure. We saw how using desktop-first to create our designs and wireframes is beneficial because having a large canvas allows us to explore different layouts and properly arrange the hierarchy of the content.

When creating HTML mockups, using mobile-first is better because a mobile-friendly site will have more reach, allow focused content, and leverage mobile devices' technologies.

We were able to retrofit with AWD and RWD a fixed-width site using the magic formula. We also discussed the benefits of RWD, since it required a lot less code. However, the analysis of the travel sites clearly shows us that RWD sometimes isn't the right solution.

We also saw how Respond.js can be used to make legacy browsers support media queries if we are building with a mobile-first approach. Using conditional classes is a good technique because it's not intrusive, it's very easy to implement, and it has no JavaScript dependencies.

In the next chapter, we're going to talk about some of the most interesting subjects in the world of RWD: CSS grids, CSS frameworks, and the power of Flexbox.

Let's do this!
Where to buy this book
You can buy Mastering Responsive Web Design from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.