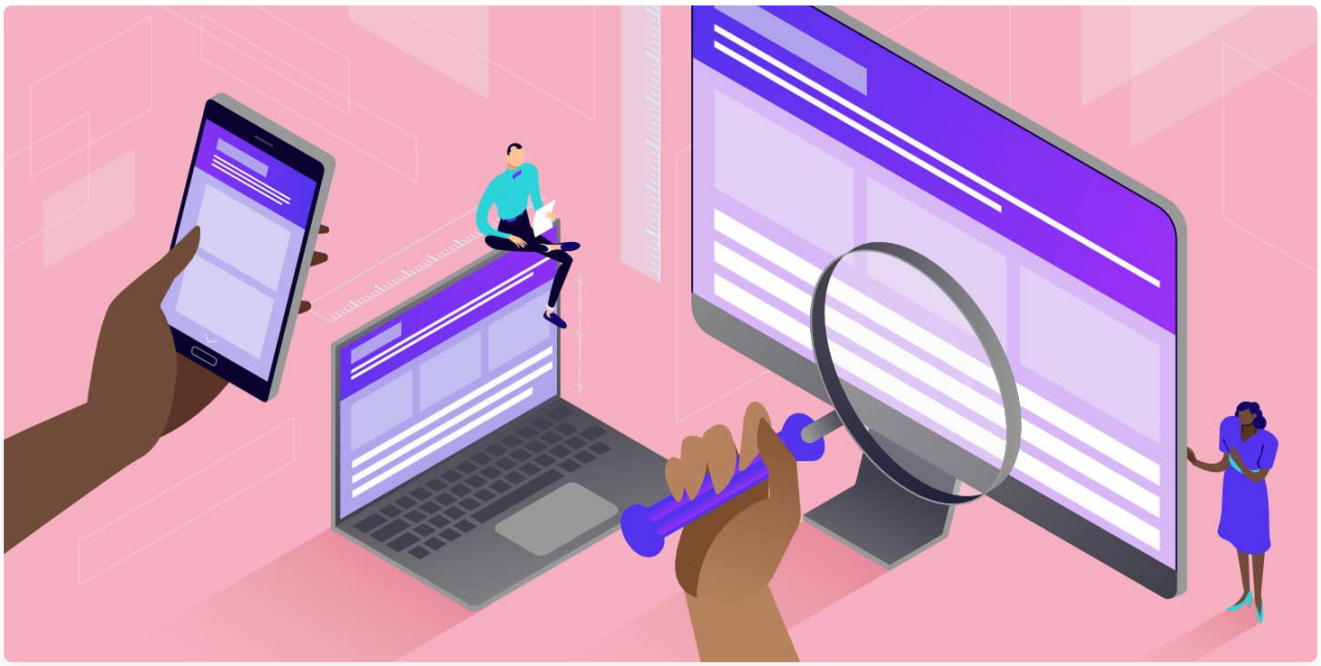
KINSTA BLOG

The Beginner's Guide to Responsive Web Design (Code Samples & Layout Examples)

Matteo Duò, June 18, 2021

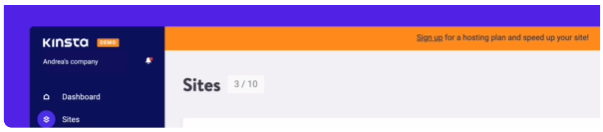


With an internet increasingly accessed from mobile devices, it's no longer enough to have a static website design that only looks good on a computer screen.

Not to mention, you also have to consider tablets, 2-in-1 laptops, and different smartphone models with different screen dimensions when coming up with a design.

So slapping your content into a single column and calling it quits isn't going to cut it.

With responsive web design, you can make sure your website looks its best on cell phones, tablets, laptops, and desktop screens.



And that improvement in user experience means higher conversions and business growth.

This guide will give you everything you need to know about responsive website design, including definitions, a step-by-step walkthrough, examples, and more.

Table of Contents

What Is Responsive Web Design?

Responsive design is an approach to web design that makes your web content adapt to the different screen and window sizes of a variety of devices.

For example, your content might be separated into different columns on desktop screens, because they are wide enough to accommodate that design.

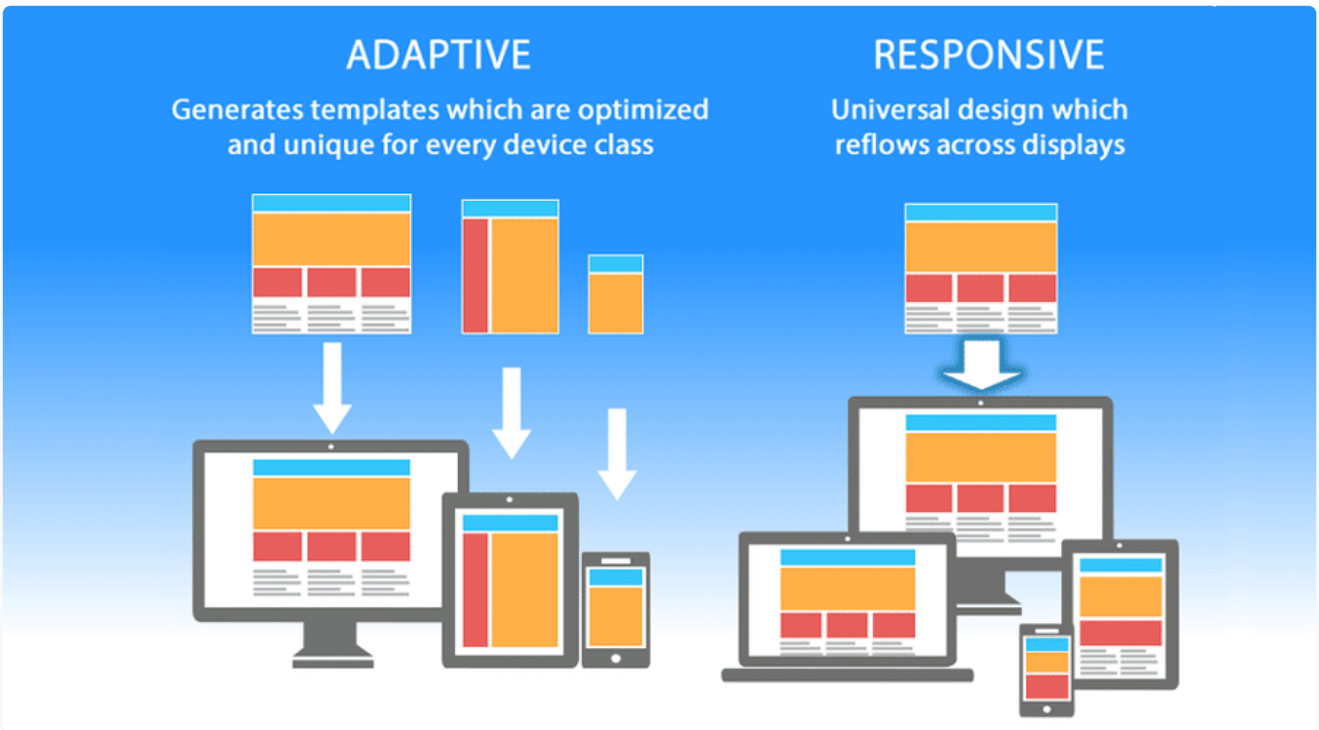
If you separate your content into multiple columns on a mobile device, it will be hard for users to read and interact with.

Responsive design makes it possible to deliver multiple, separate layouts of your content and design to different devices depending on screen size.

> **It's not enough for your website to look good on a computer screen. 🖥️ Tablets, 2-in-1 laptops, and smartphones are all part of the equation... and this guide covers everything you need to know about responsive design 🤳**
>
> CLICK TO TWEET

Responsive Web Design vs Adaptive Design

The difference between responsive design and adaptive design is that responsive design adapts the rendering of a single page version. In contrast, adaptive design delivers multiple completely different versions of the same page.



— Responsive vs adaptive design

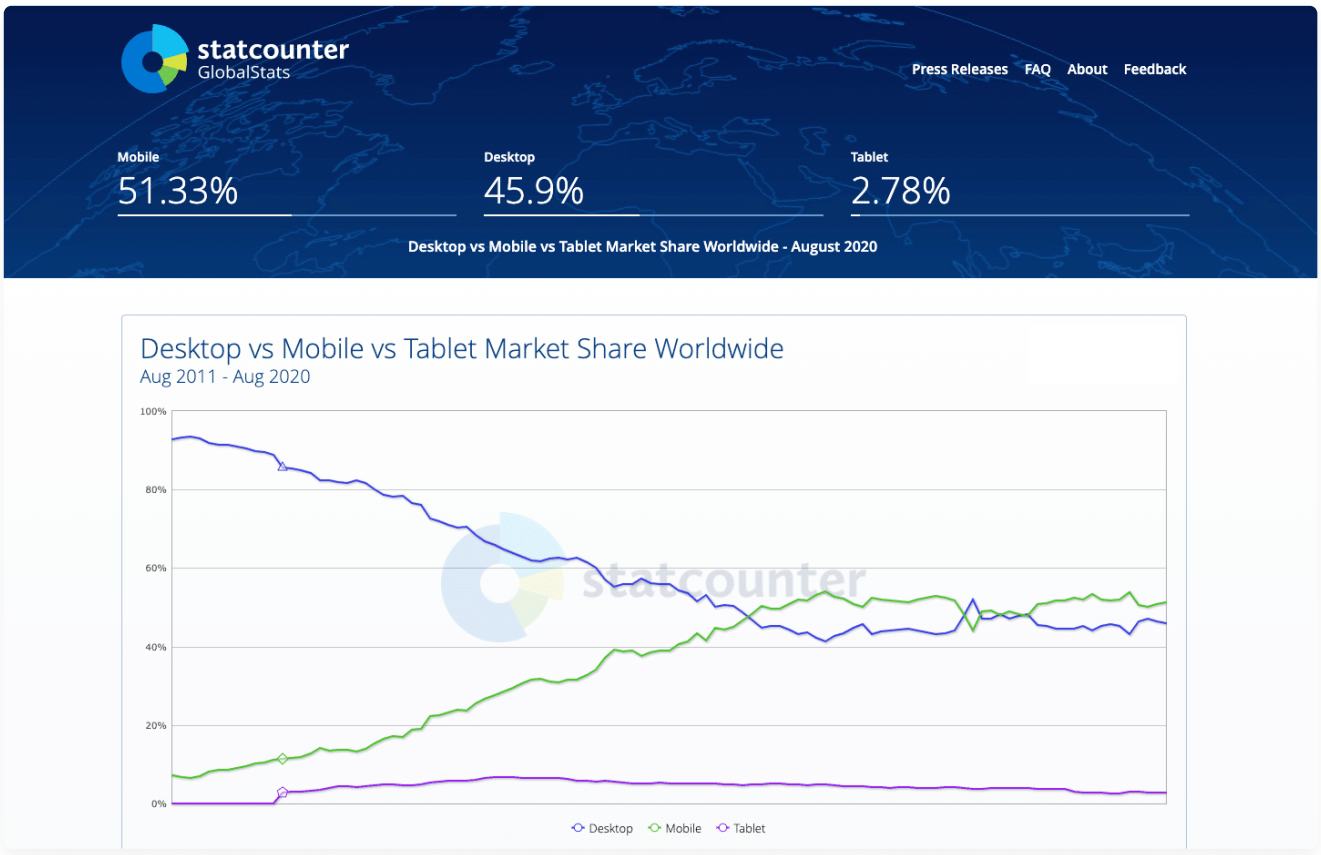
They are both crucial web design trends that help webmasters control how their site looks on different screens, but the approach is different.

With responsive design, users will access the same basic file through their browser, regardless of device, but CSS code will control the layout and render it differently based on screen size. With adaptive design, there is a script that checks for the screen size, and then accesses the template designed for that device.

Why Responsive Design Matters

If you're new to web design, development, or blogging, you might wonder why responsive design matters in the first place.

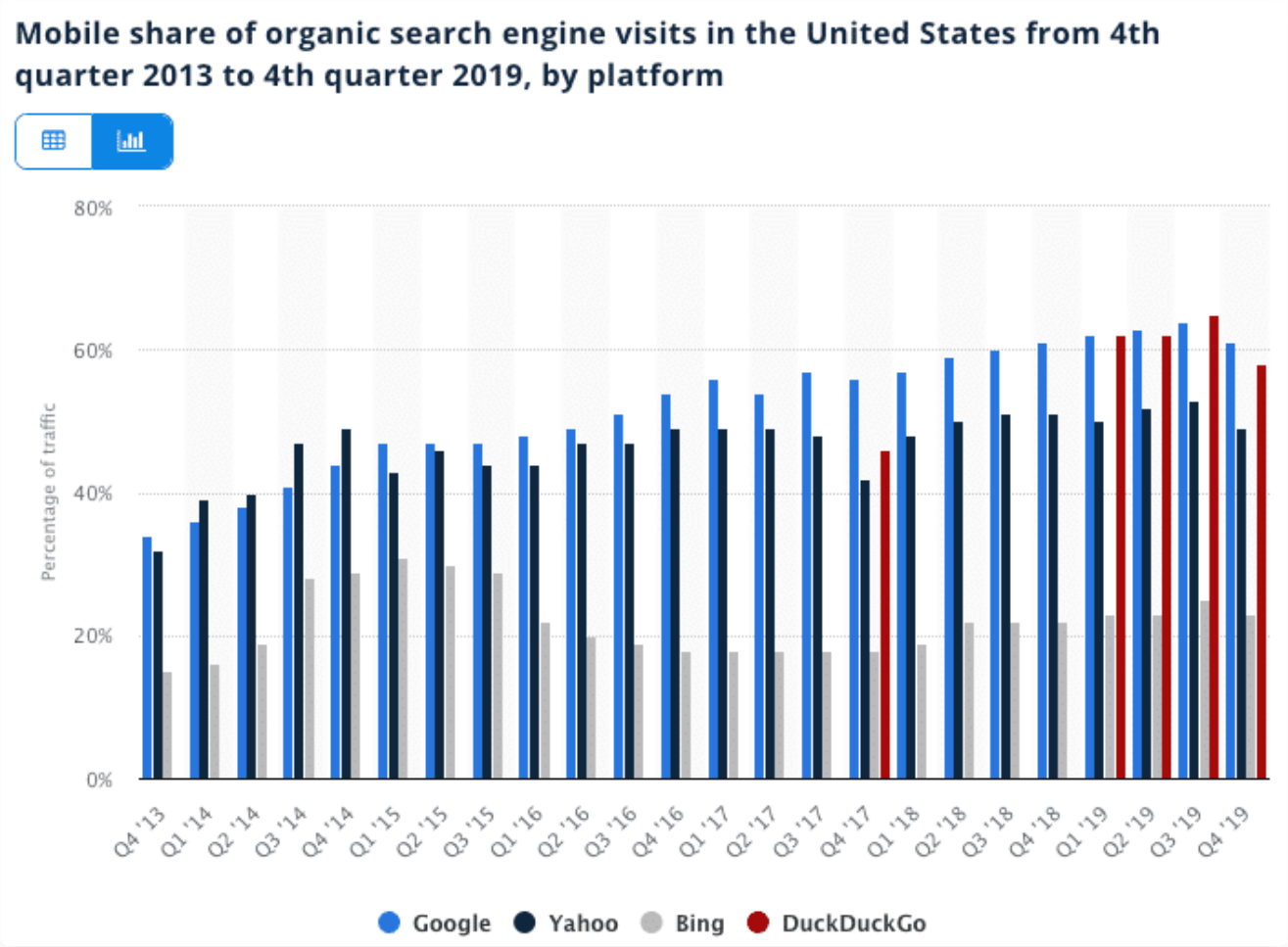
The answer is simple. It's no longer enough to design for a single device. Mobile web traffic has overtaken desktop and now makes up the majority of website traffic, accounting for more than 51%.



— Mobile, tablet, desktop market share

When over half of your potential visitors are using a mobile device to browse the internet, you can't just serve them a page designed for desktop. It would be hard to read and use, and lead to bad user experience.

But that's not all. Users on mobile devices also make up the majority of search engine visits.



— Mobile search traffic

Finally, over the last few years, mobile has become one of the most important advertising channels. Even in a post-pandemic market, mobile ad spending is growing 4.8% to $91.52 billion.

Whether you choose to advertise on social media or use an organic approach like YouTube SEO, the vast majority of your traffic will come from mobile users.

If your landing pages aren't optimized for mobile and easy to use, you won't be able to maximize the ROI of your marketing efforts. Bad conversion rates will lead to fewer leads and wasted ad spend.

Are WordPress Sites Responsive?

Whether or not WordPress sites are responsive depends on the theme of your WP site. A WordPress theme is the equivalent of a template for a static website and controls the design and layout of your content.

If you use a default WordPress theme, like Twenty Twenty, the design is responsive, but since it's a single-column design, you might not realize it when looking at it on different screens.

If you use another WordPress theme, you can test if it's responsive or not by comparing how it looks on different devices or using Chrome Developer Tools.

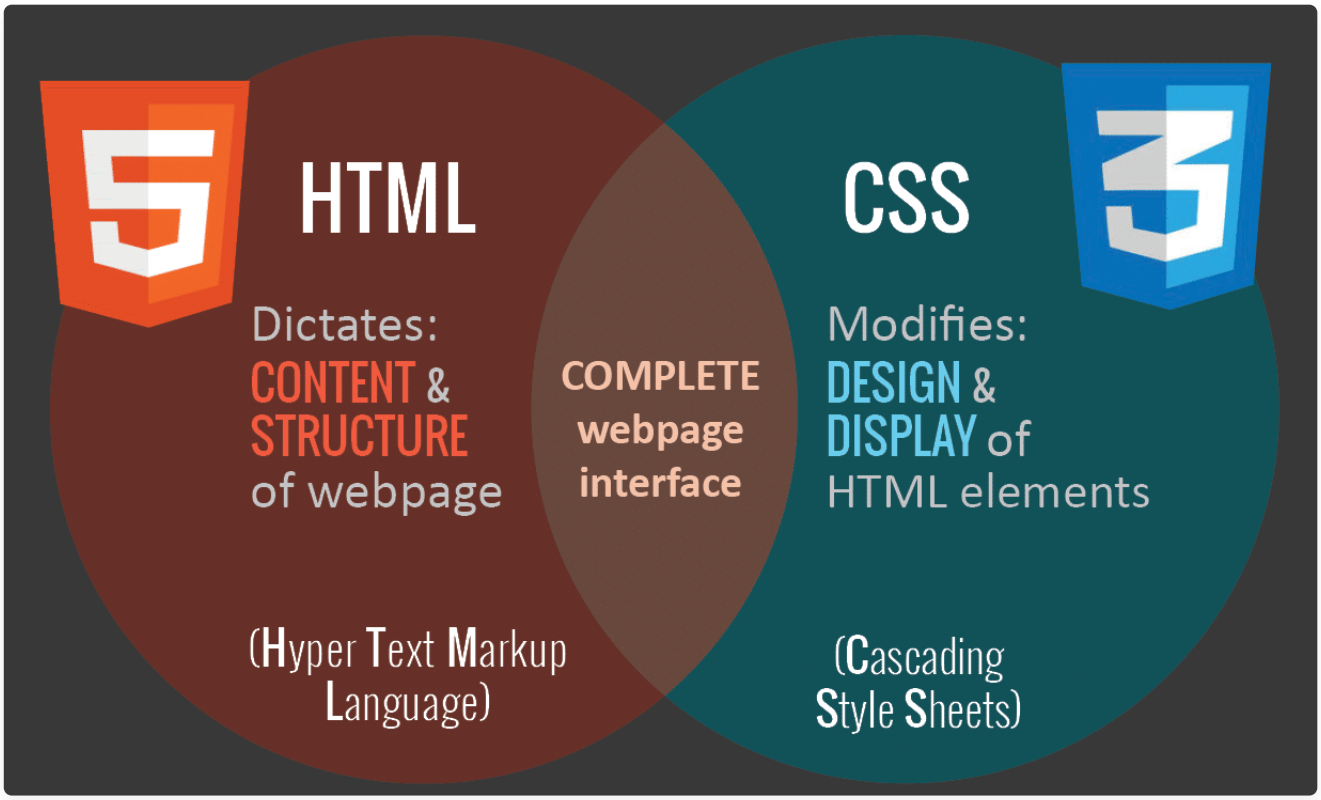
The Building Blocks of Responsive Web Design

In this section, we'll cover the underlying foundation for responsive website design and its different building blocks.

- CSS and HTML
- Media Queries
- Fluid Layouts
- Flexbox Layout
- Responsive Images
- Speed

CSS and HTML

The foundation of responsive design is the combination of HTML and CSS, two languages that control the content and layout of a page in any given web browser.



— HTML vs CSS (Image source: codingdojo.com)

HTML mainly controls the structure, elements, and content of a webpage. For example, to add an image to a website, you have to use HTML code like this:

```
<img src="image.gif" alt="image" class=”full-width-img”>
```

You can set a “class” or “id” that you can later target with CSS code.

You could also control primary attributes such as height and width within your HTML, but this is no longer considered best practice.

Instead, CSS is used to edit the design and layout of the elements you include on a page with HTML. CSS code can be included in a `<style>` section of a HTML document, or as a separate stylesheet file.

For example, we could edit the width of all HTML images at the element level like this:

```
img {
width: 100%;
}
```

Or we could target the specific class “full-width-img” by adding a period in front.

```
.full-width-img {
width: 100%;
}
```

You can also control the design beyond just height, width, and color. Using CSS like this is how you make a design responsive when you combine it with a technique called media query.

Media Queries

A media query is a fundamental part of CSS3 that lets you render content to adapt to different factors like screen size or resolution.

— Media queries for desktop, tablet, smartphone

It works in a similar way to an “if clause” in some programming languages, basically checking *if* a screen’s viewport is wide enough or too wide before executing the appropriate code.

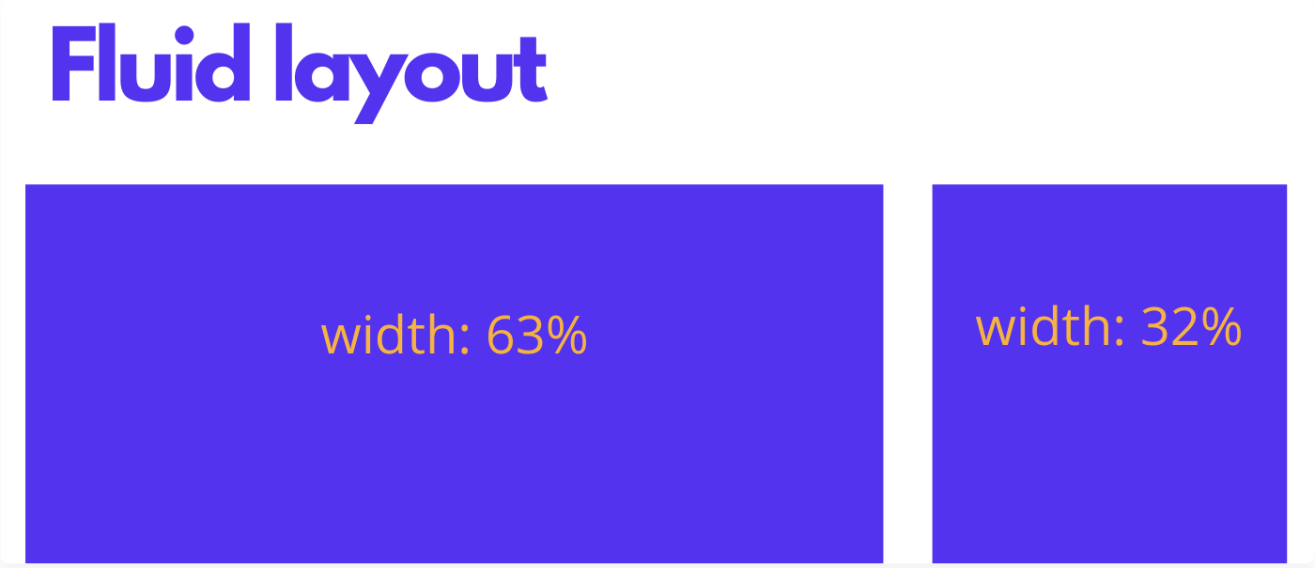
```
@media screen and (min-width: 780px) {
.full-width-img {
margin: auto;
width: 90%;
}
```

If the screen is at least 780 pixels wide, “full-width-img” class images will take up 90% of the screen and be automatically centered by equally wide margins.

Fluid Layouts

A fluid layout is an essential part of modern responsive design. In the good old days, you would set a static value for every HTML element, like 600 pixels.

A fluid layout relies instead on dynamic values like a percentage of the viewport width.



— Example of fluid layout

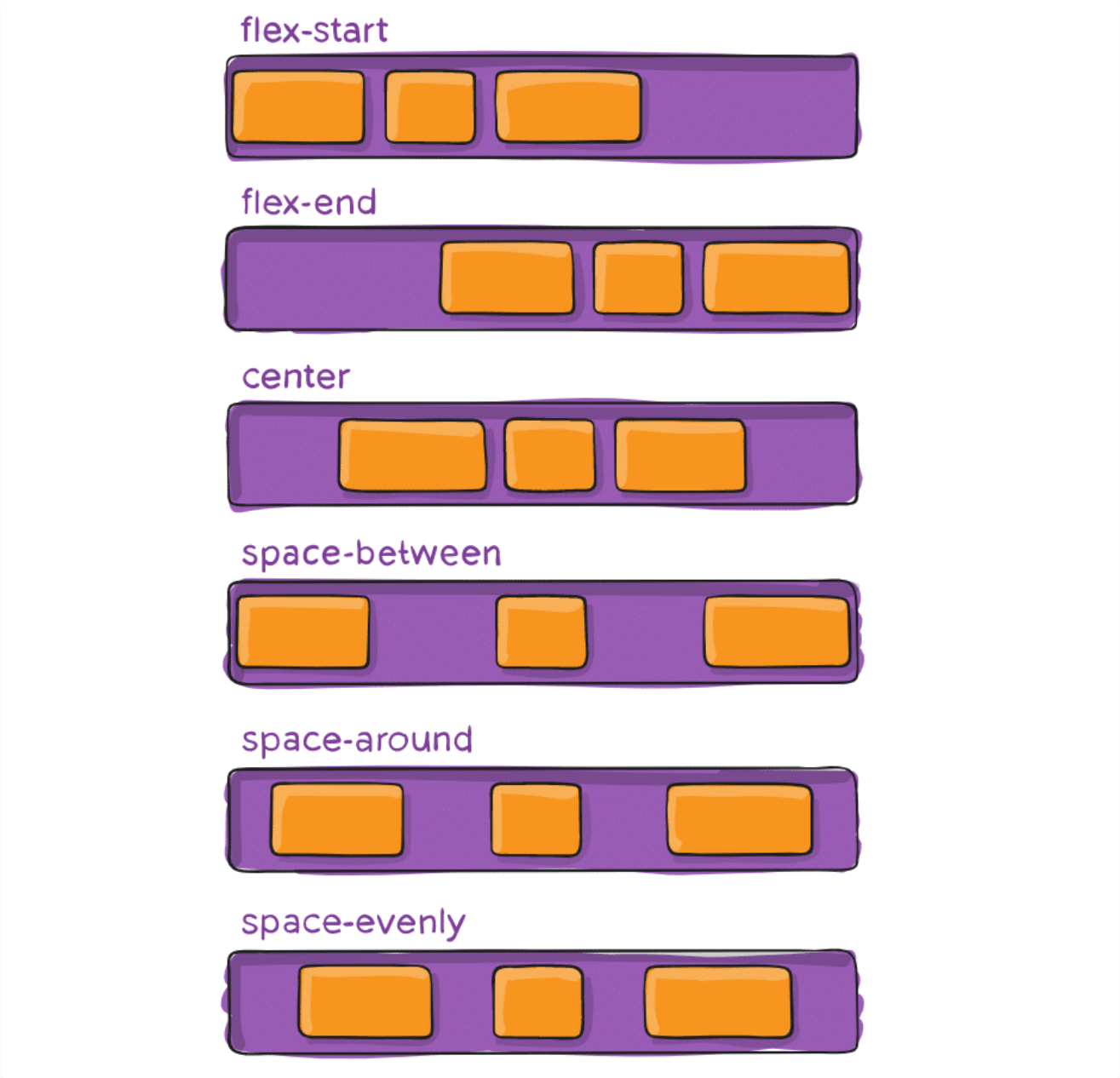
This approach will dynamically increase or decrease the different container element sizes based on the size of the screen.

Flexbox Layout

While a percentage-based layout is fluid, many designers and web developers felt it was not dynamic or flexible enough. Flexbox is a CSS module designed as a more efficient way to lay

out multiple elements, even when the size of the contents inside the container is unknown.

A flex container expands items to fill available free space or shrinks them to prevent overflow. These flex containers have a number of unique properties, like justify-content, that you can't edit with a regular HTML element.



— Flexbox container

It's a complicated topic, so if you want to use it in your design, you should read CSS Tricks' flexbox guide.

Responsive Images

The most basic iteration of responsive images follows the same concept as a fluid layout, using a dynamic unit to control the width or height. The sample CSS code we covered earlier already accomplishes this:

```
img {
width: 100%;
}
```

The % unit approximates to a single percentage of the width or height of the viewport and makes sure the image remains in proportion to the screen.

The problem with this approach is that every user has to download the full-sized image, even on mobile.

To serve different versions scaled for different devices, you need to use the HTML `srcset` attribute in your img tags, to specify more than one image size to choose from.

```
<img srcset="large-img.jpg 1024w,
middle-img.jpg 640w,
small-img.jpg  320w"
src="small.jpg"
/>
```

WordPress automatically uses this functionality for images included in posts or pages.

Speed

When you're attempting to create a responsive design for your website, the loading speed should be a top priority.

Pages that load in 2 seconds have an average 9% bounce rate, while pages that take 5 seconds lead to a 38% bounce rate.

Your approach to responsiveness must not block or delay your page's first render any more than it needs to.

There are several way you could make your pages faster. Optimizing your images, implementing caching, minification, using a more efficient CSS layout, avoiding render-blocking JS, and improving your critical rendering path are all great ideas you should consider.

You could also try to implement Google AMP for your mobile pages, but in our Google AMP case study, our mobile leads dropped by a whopping 59%.

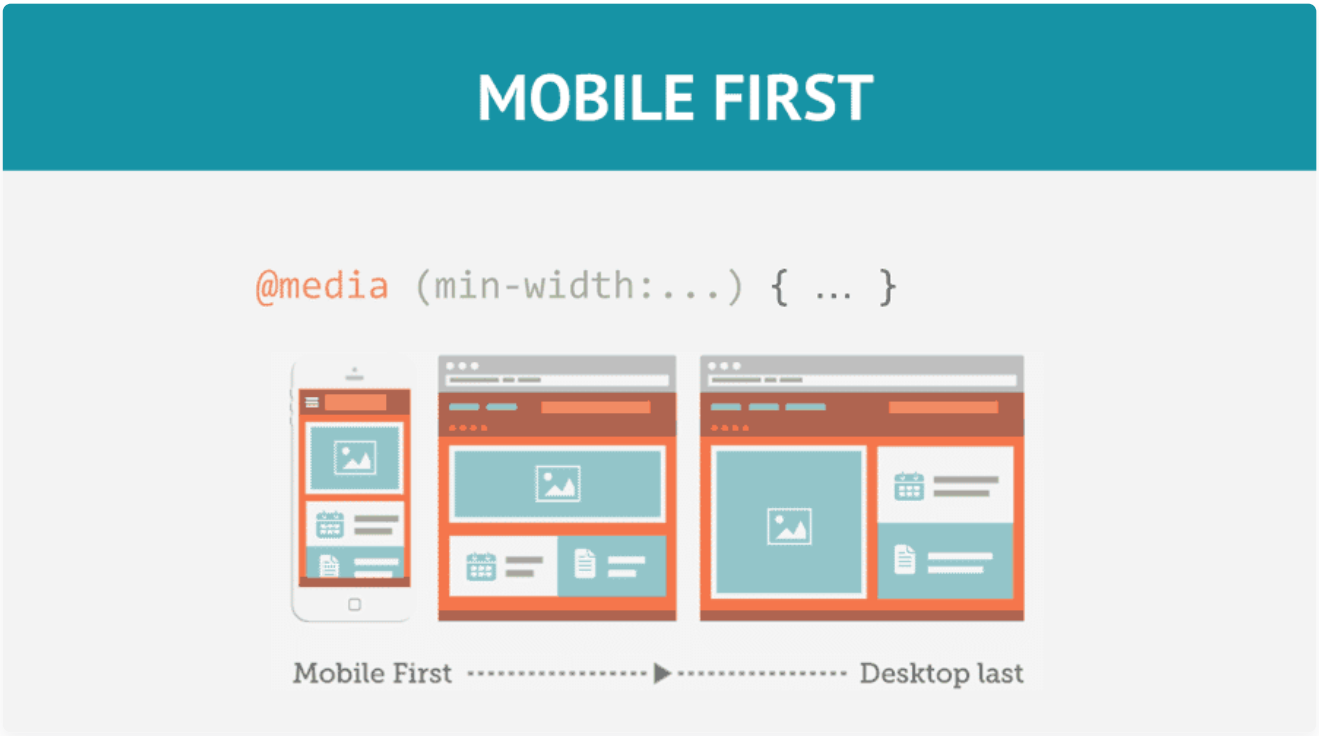
Common Responsive Breakpoints

To work with media queries, you need to decide on the “responsive breakpoints” or screen size breakpoints. A breakpoint is the width of the screen where you use a media query to implement new CSS styles.

Common screen sizes

- Mobile: 360 x 640
- Mobile: 375 x 667
- Mobile: 360 x 720
- iPhone X: 375 x 812
- Pixel 2: 411 x 731
- Tablet: 768 x 1024
- Laptop: 1366 x 768
- High-res laptop or desktop: 1920 x 1080

If you choose a mobile-first approach to design, with a single column and smaller font sizes as the basis, you don’t need to include mobile breakpoints — unless you want to optimize the design for specific models.



— Mobile-first design (Image source: silocreativo.com)

So you can create a basic responsive design with just two breakpoints, one for tablets and one for laptops and desktop computers.

Bootstrap's Responsive Breakpoints

As one of the first, and most popular, responsive frameworks, Bootstrap led the assault on static web design and helped establish mobile-first design as an industry standard.

As a result, many designers to this day still follow Bootstrap's screen-width breakpoints.

Responsive breakpoints

Since Bootstrap is developed to be mobile first, we use a handful of media queries to create sensible breakpoints for our layouts and interfaces. These breakpoints are mostly based on minimum viewport widths and allow us to scale up elements as the viewport changes.

Bootstrap primarily uses the following media query ranges—or breakpoints—in our source Sass files for our layout, grid system, and components.

Copy

```
// Extra small devices (portrait phones, less than 576px)
// No media query for `xs` since this is the default in Bootstrap

// Small devices (landscape phones, 576px and up)
@media (min-width: 576px) { ... }

// Medium devices (tablets, 768px and up)
@media (min-width: 768px) { ... }

// Large devices (desktops, 992px and up)
@media (min-width: 992px) { ... }

// Extra large devices (large desktops, 1200px and up)
@media (min-width: 1200px) { ... }
```

Since we write our source CSS in Sass, all our media queries are available via Sass mixins:

— Bootstrap responsive breakpoints

They use media queries to target landscape phones (576px), tablets (768px), laptops (992px) and extra large desktop screens (1200px).

How to Make Your Website Responsive

Now that you're familiar with the building blocks, it's time to make your website responsive.

Set Your Media Query Ranges (Responsive Breakpoints)

Set your media query ranges based on the unique needs of your design. For example, if we wanted to follow the Bootstrap standards for our design, we would use the following media queries:

- 576px for portrait phones
- 768px for tablets
- 992px for laptops
- 1200px for large devices

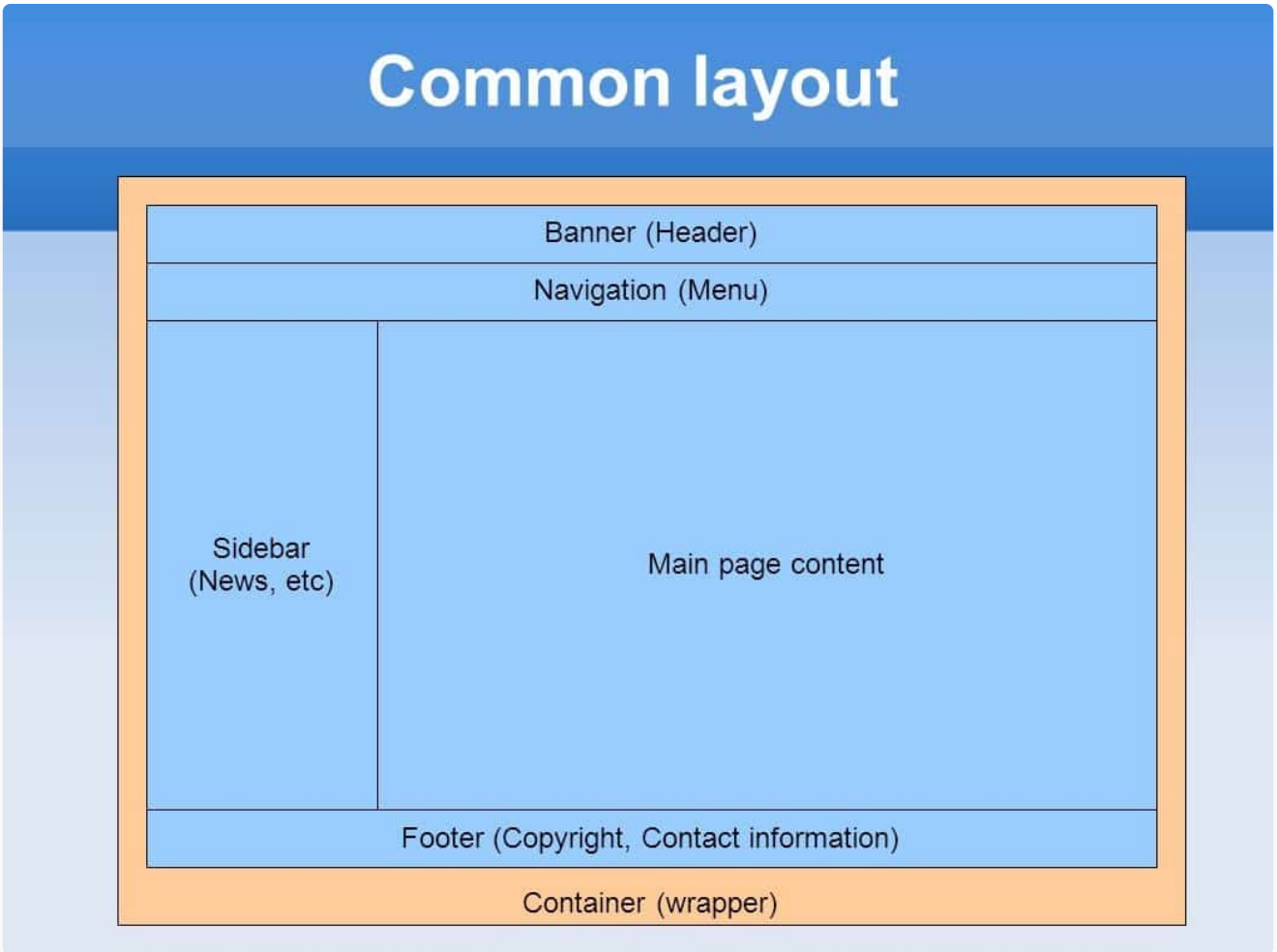
Size Layout Elements with Percentages or Create a CSS Grid Layout

The first and most important step is to set up different sizes for different layout elements depending on the media query or screen breakpoint.

The number of layout containers you have will depend on the design, but most websites focus on the elements listed below:

- Wrapper or Container
- Header

- Content
- Sidebar
- Footer



— Common layout

Using a mobile-first approach, you can style the main layout elements like this (with no media query for the basic styles for mobile phones):

```
#wrapper {width:95%;  margin: 0 auto; }

#header {width:100%; }

#content {width:100%; }

#sidebar {width:100%; }

#footer {width:100%; }
```

```
// Small devices (landscape phones, 576px and up)

@media (min-width: 576px) {

// Medium devices (tablets, 768px and up)

@media (min-width: 768px) {

#wrapper {width:90%;  margin: 0 auto; }

#content {width:70%; float:left; }

#sidebar {width:30%; float:right; }

// Large devices (desktops, 992px and up)

@media (min-width: 992px) { ... }

}

// Extra large devices (large desktops, 1200px and up)

@media (min-width: 1200px) {

#wrapper {width:90%;  margin: 0 auto; }

}
```

In a percentage-based approach, the “float” attribute controls which side of the screen an element will appear on, left, or right.

If you want to go beyond the basics and create a cutting-edge responsive design, you need to familiarize yourself with the CSS flexbox layout and its attributes like box-sizing and flex.

Implement Responsive Images

One way to make sure that your images don’t break is merely using a dynamic value for all pictures, as we covered earlier.

dashboard. Check out our plans

```
img {
width: 100%;
}
```

But that won't reduce the load placed on your mobile visitors when they access your website.

Make sure you always include a `srcset` that with different sizes of your photo when you add images to your pages.

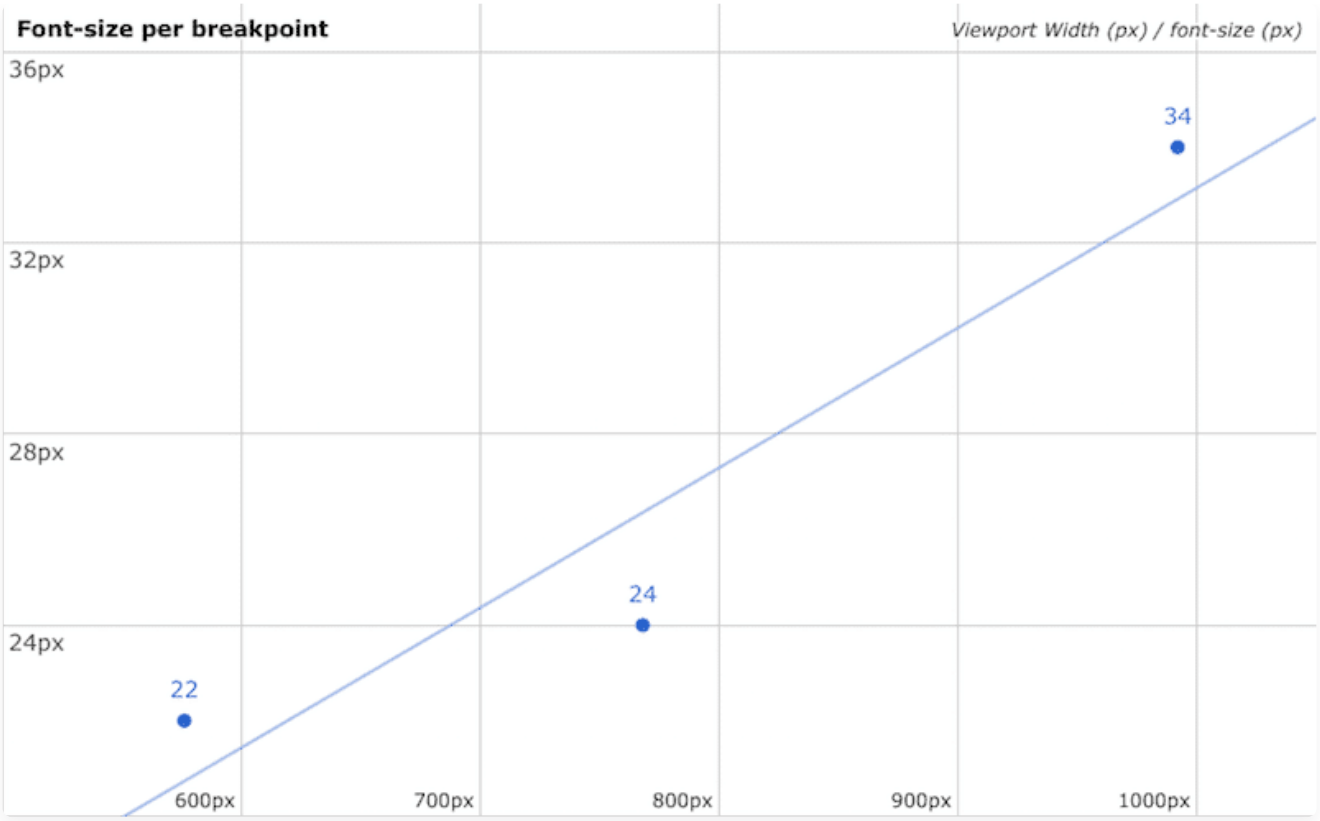
Doing this manually can be quite time-consuming, but with a CMS like WordPress, it automatically happens when you upload media files.

Responsive Typography For Your Website Text

The main focus of responsive web design is on the responsiveness of the layout blocks, elements, and media. Text is often treated as an afterthought.

But for a truly responsive design, you should also adjust your font-sizes appropriately to match screen size.

The easiest way to do so is to set a static value for font-size, like 22 px, and adapt it in each media query.



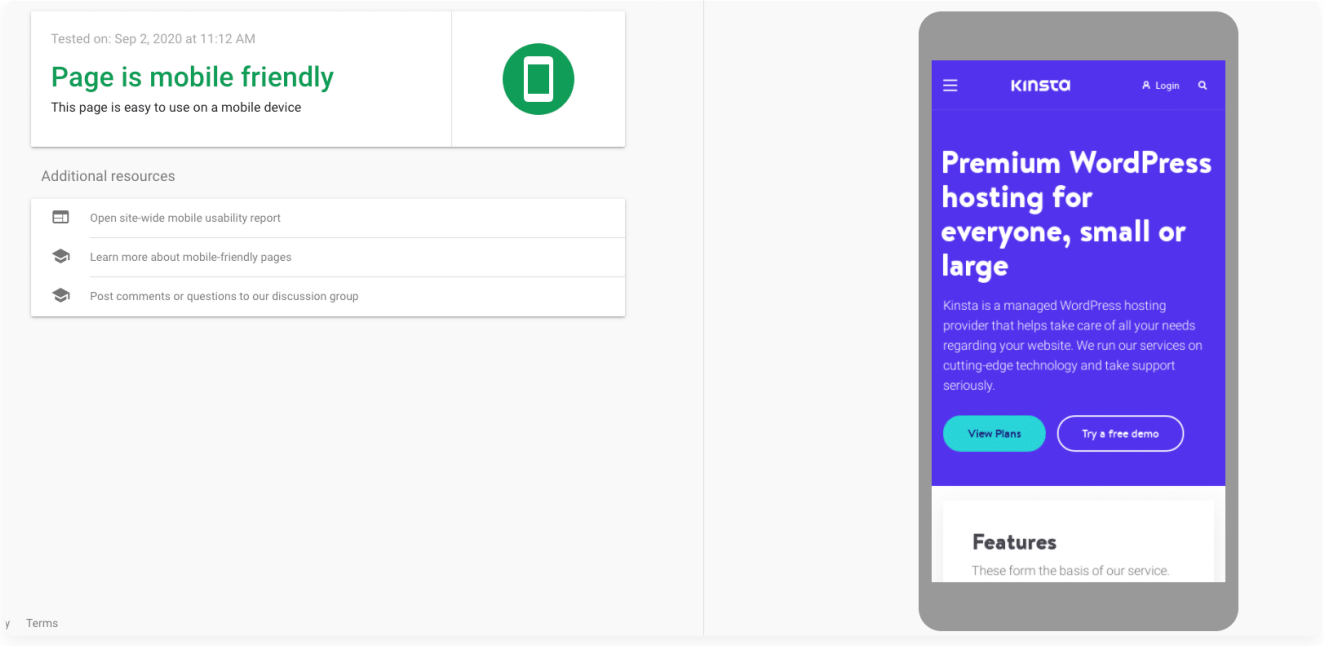
— Font size vs view size scatter points

You can target multiple text elements at the same time by using a comma to separate each one.

```
@media (min-width: 992px) {

body, p, a, h4 {

font-size: 14px;

}

}
```

Test Responsiveness

First, you want to test whether your site is mobile-friendly with Google's mobile-friendly test. Simply enter the URL of your website and click the "test URL" button to get the results.



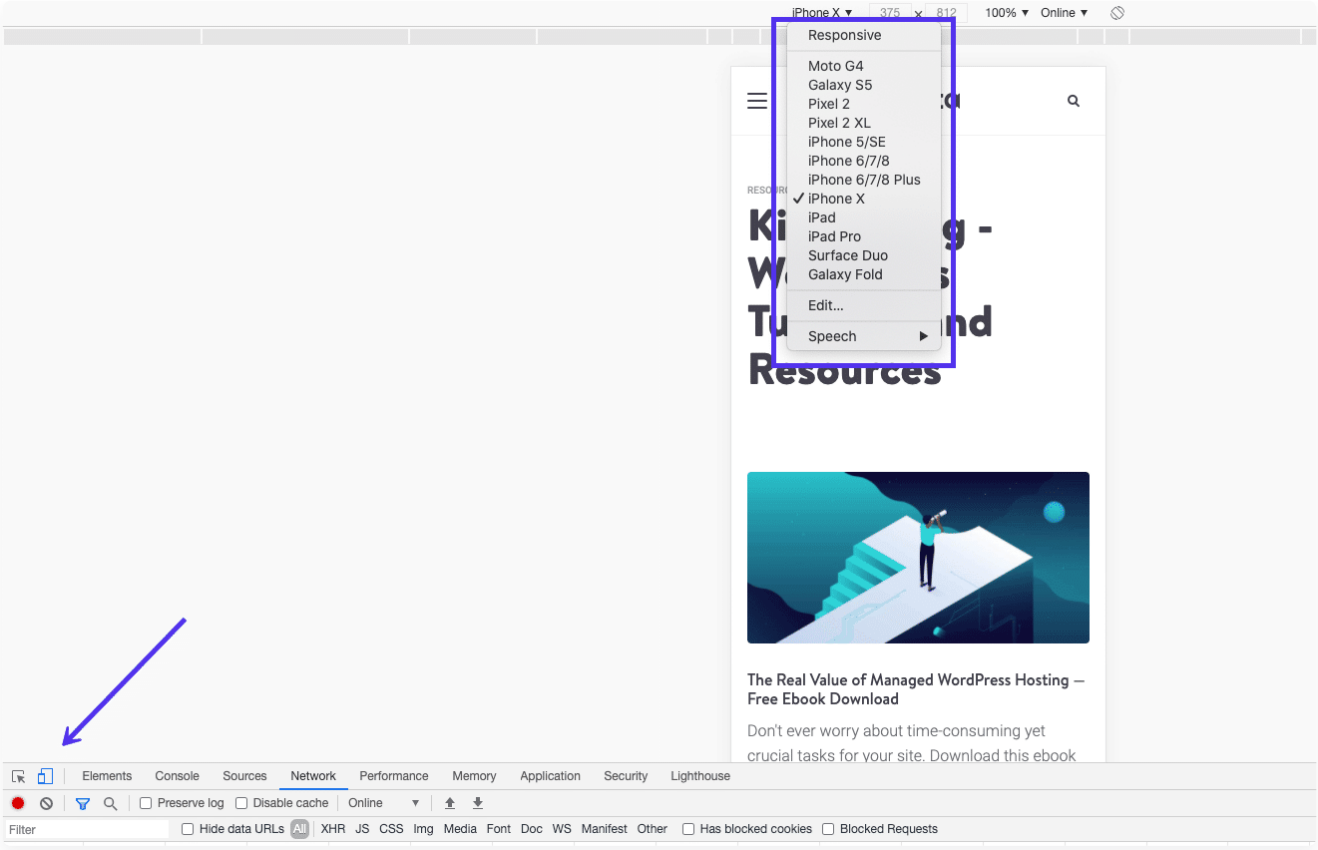
— Google's Mobile-Friendly Test

Don't worry if it takes a while to fetch your site. That doesn't reflect your page loading speed.

If you followed the steps outlined in this article, it should say that you have a mobile-friendly website.

Then you want to test your site on multiple screen sizes with a tool like Chrome developer tools.

Press CTRL + Shift + I on Windows computers, or Command + Option + I on Macs to open the relevant device view. From here, you can select the mobile device or tablet of your choice to test the responsiveness of your design.



— Testing responsive & mobile layouts in Chrome

There are a couple of questions you want to answer when going through this process.

- Does the layout adjust to the correct amount of columns?
- Does the content fit well inside the layout elements and containers on different screens?
- Do the font sizes fit each screen?

CSS Units and Values for Responsive Design

CSS has both absolute and relative units of measurement. An example of an absolute unit of length is a cm or a px. Relative units or dynamic values depend on the size and resolution of the screen or the font sizes of the root element.

PX vs EM vs REM vs Viewport Units for responsive design

- PX – a single pixel
- EM – relative unit based on the font-size of the element.
- REM – relative unit based on the font-size of the element.
- VH, VW – % of the viewport's height or width.

- % – the percentage of the parent element.

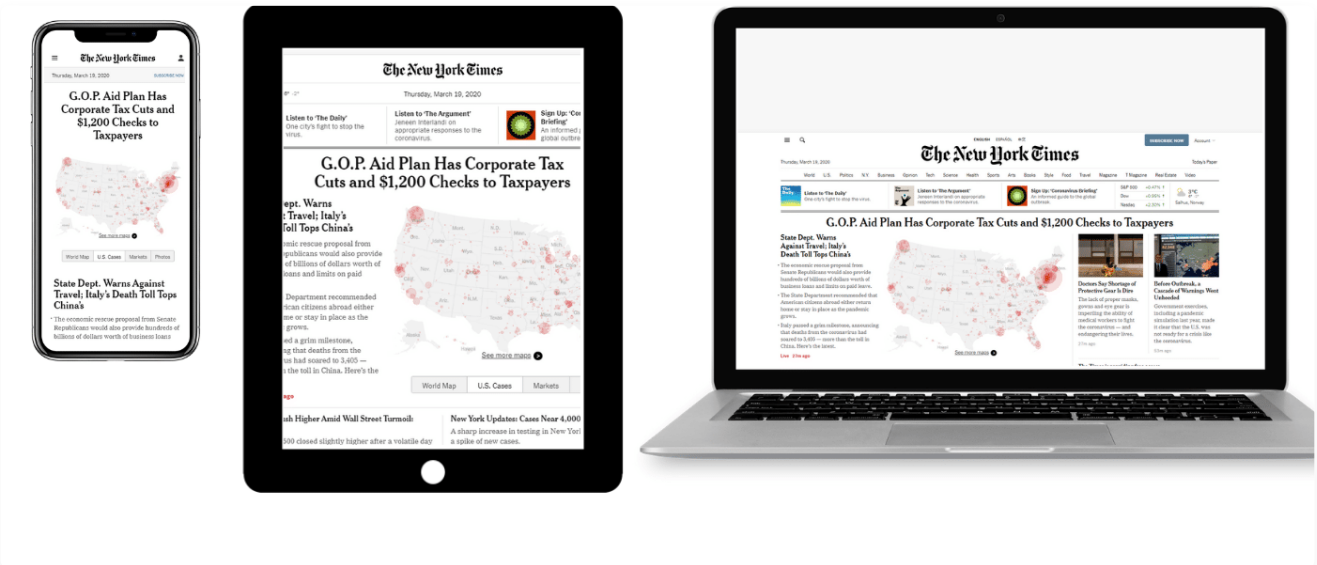
A new web designer or developer should probably stick with pixels for text because they are the most straight-forward unit of length in CSS.

But when setting the width and max-width of images and other elements, using % is the best solution. This approach will make sure the components adjust to the screen size of every device.

Responsive Design Examples

Below we will cover a few examples of responsive web design from different industries — and learn from what they do right and wrong.

1. Online Newspaper: New York Times

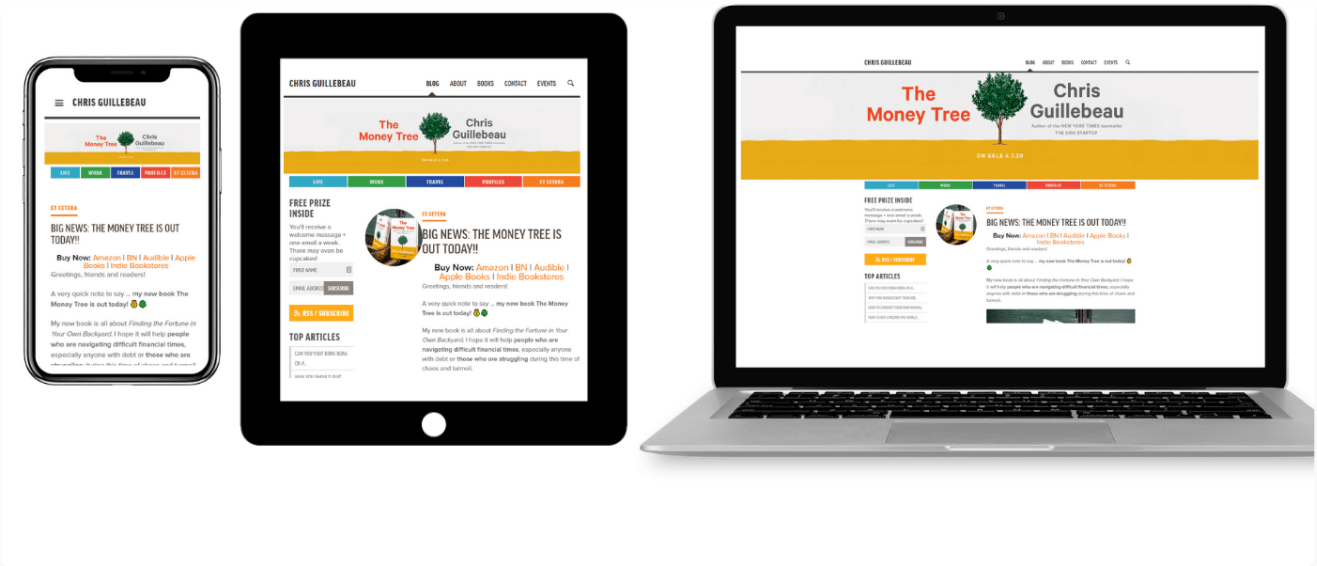


— NYT on mobile, tablet, and laptop

On desktop, the NYT layout reminds you of a traditional newspaper, crowded with visuals and different rows and columns of content. There seems to be a separate column or row for every category of news.

On mobile, it conforms to the single-column standard and also adjusts the menu to be in the accordion format to be easier to use.

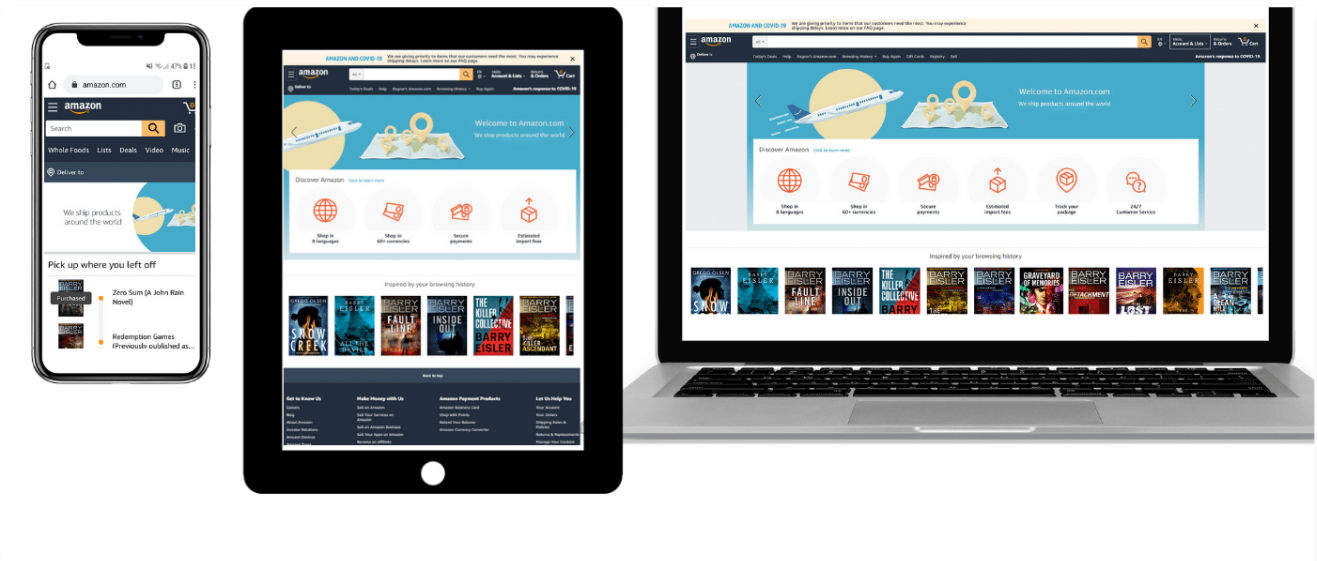
2. Blog: The Art of Non-Conformity



— The Art of Non-Conformity on mobile, tablet, and laptop

Chris Guillebeau's blog "The Art of Non-Conformity" has been going strong for over a decade. While the design isn't the most cutting edge, it's responsive and adapts the two-column sidebar and main content layout to a single-column design on mobile devices.

3. Ecommerce: Amazon



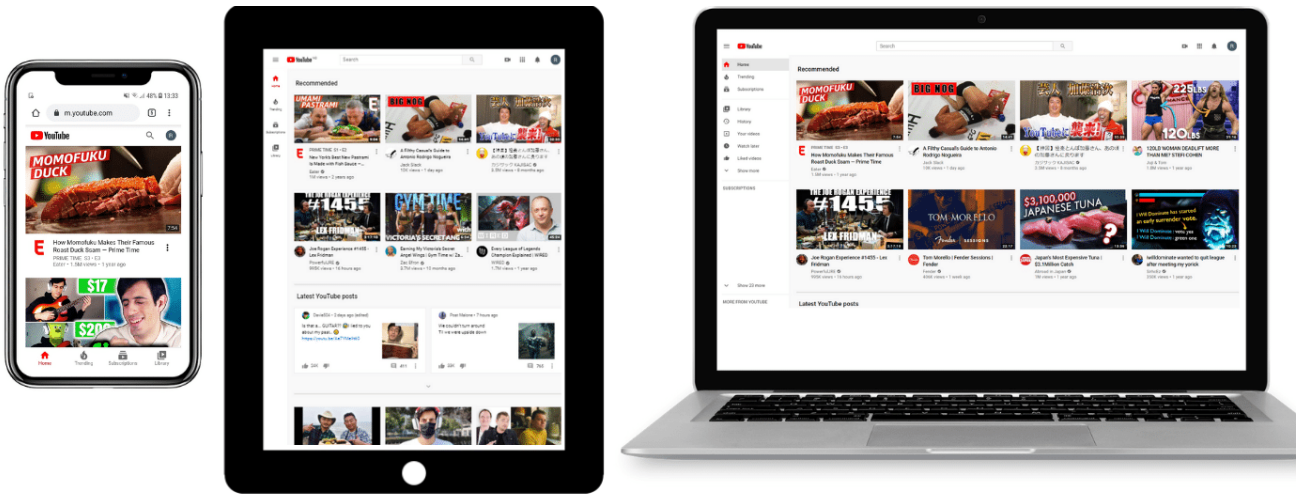
— Amazon on mobile, tablet, and laptop

Amazon is a global leader in ecommerce for a reason, their user interface is perfectly fluid across all devices.

Their tablet layout simply removes some of the white space and adds a scrollable section of icons to fit more content into a smaller package.

Their mobile layout brings it into a single column, focusing on the essentials, like recent purchase history, rather than the different section link icons from their main homepage.

4. Video Site: YouTube

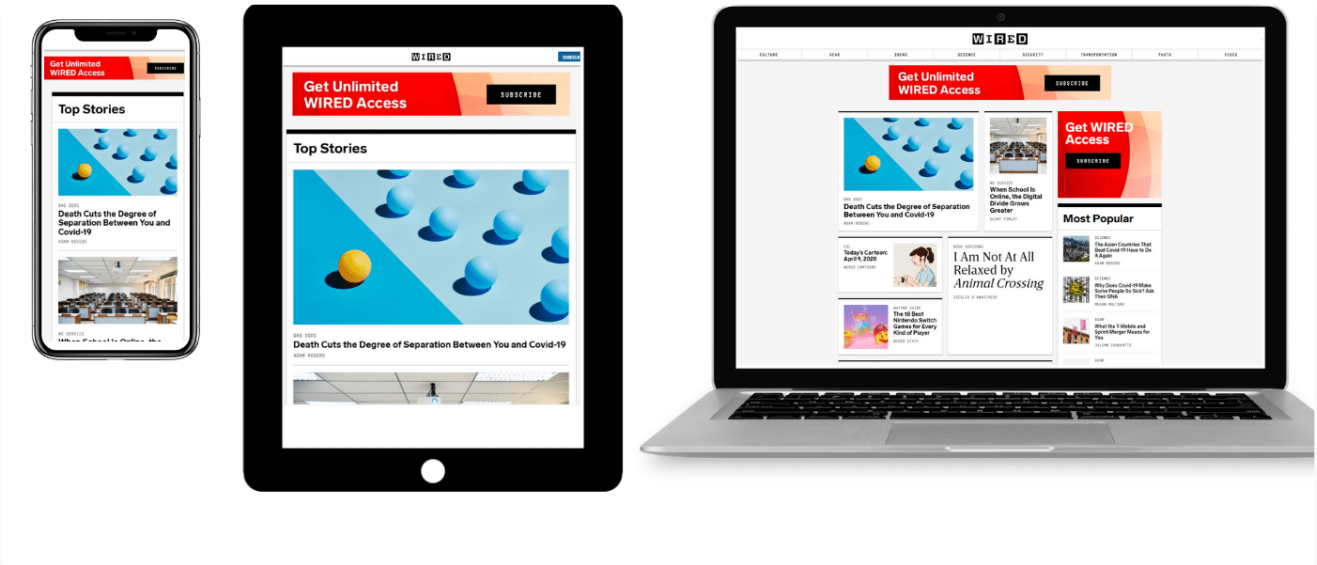


YouTube on mobile, tablet, and laptop

The core of YouTube's homepage design is a flexible grid of videos that are relevant to each user. On tablets, the number of columns in each row goes down to three. On mobile, it's reduced to a single-column design.

The mobile version also moves the main menu to the bottom of the screen, closer to the thumbs of their smartphone users. This simple move improves navigation and UX.

5. Online Magazine: Wired



— Wired on mobile, tablet, and laptop

Wired's approach to responsive web design is focused on implementing a single-column layout on all smaller screens, starting with tablets.

It's a basic layout but makes it easier to draw user attention to top stories and their CTA to subscribe.

> **Make sure your website looks ✨ stunning ✨ across devices with this guide to responsive design!**
>
> CLICK TO TWEET

Summary

There are a lot of different elements that go into responsive web design. Without a basic understanding of HTML and CSS, it can be easy to make mistakes.

But through familiarizing yourself with the different building blocks, analyzing the examples with web dev tools, and testing as you go using the sample code, you should be able to make your website responsive without any major issue.

If that sounds too much to achieve, you can always either hire a WordPress developer or simply make sure your theme is already responsive.

5th April 2021

Developer console

Code is prone to errors. You will quite likely make errors... Oh, what am I talking about? You are *absolutely* going to make errors, at least if you're a human, not a robot.

But in the browser, users don't see errors by default. So, if something goes wrong in the script, we won't see what's broken and can't fix it.

To see errors and get a lot of other useful information about scripts, "developer tools" have been embedded in browsers.

Most developers lean towards Chrome or Firefox for development because those browsers have the best developer tools. Other browsers also provide developer tools, sometimes with special features, but are usually playing "catch-up" to Chrome or Firefox. So most developers have a "favorite" browser and switch to others if a problem is browser-specific.

Developer tools are potent; they have many features. To start, we'll learn how to open them, look at errors, and run JavaScript commands.

Google Chrome

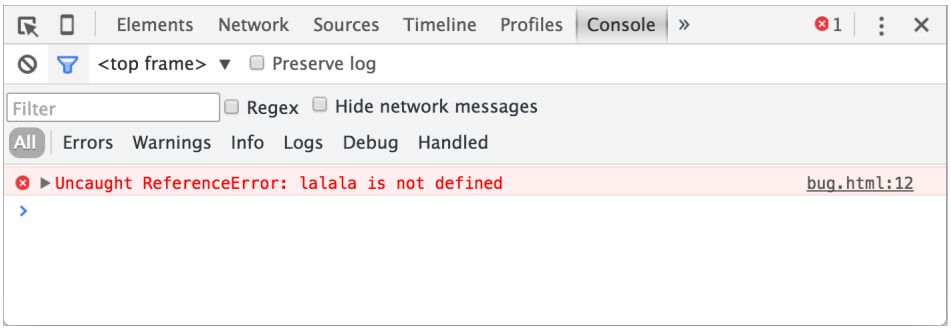
Open the page bug.html.

There's an error in the JavaScript code on it. It's hidden from a regular visitor's eyes, so let's open developer tools to see it.

Press `F12` or, if you're on Mac, then `Cmd+Opt+J`.

The developer tools will open on the Console tab by default.

It looks somewhat like this:



The exact look of developer tools depends on your version of Chrome. It changes from time to time but should be similar.

- Here we can see the red-colored error message. In this case, the script contains an unknown "lalala" command.
- On the right, there is a clickable link to the source `bug.html:12` with the line number where the error has occurred.

Below the error message, there is a blue `>` symbol. It marks a "command line" where we can type JavaScript commands. Press `Enter` to run them.

Now we can see errors, and that's enough for a start. We'll come back to developer tools later and cover debugging more in-depth in the chapter Debugging in the browser.

Multi-line input

Usually, when we put a line of code into the console, and then press `Enter`, it executes.

To insert multiple lines, press `Shift+Enter`. This way one can enter long fragments of JavaScript code.

Firefox, Edge, and others

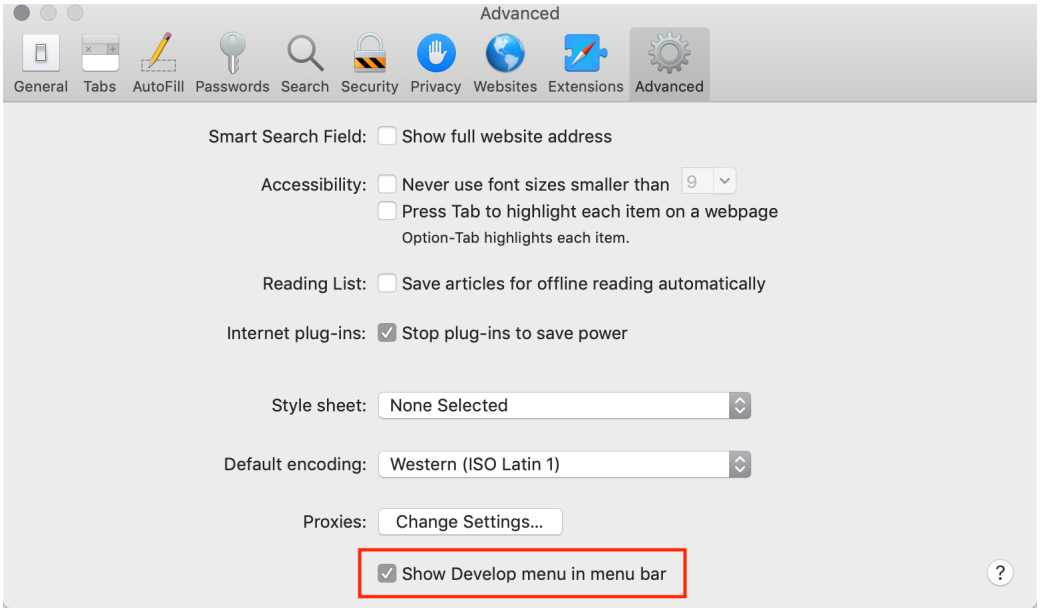
Most other browsers use `F12` to open developer tools.

The look & feel of them is quite similar. Once you know how to use one of these tools (you can start with Chrome), you can easily switch to another.

Safari

Safari (Mac browser, not supported by Windows/Linux) is a little bit special here. We need to enable the "Develop menu" first.

Open Preferences and go to the "Advanced" pane. There's a checkbox at the bottom:



Now `Cmd+Opt+C` can toggle the console. Also, note that the new top menu item named "Develop" has appeared. It has many commands and options.

Summary

- Developer tools allow us to see errors, run commands, examine variables, and much more.
- They can be opened with `F12` for most browsers on Windows. Chrome for Mac needs `Cmd+Opt+J`, Safari: `Cmd+Opt+C` (need to enable first).

Now we have the environment ready. In the next section, we'll get down to JavaScript.

Previous lesson
Next lesson
Share
Tutorial map

22nd November 2020

Hello, world!

This part of the tutorial is about core JavaScript, the language itself.

But we need a working environment to run our scripts and, since this book is online, the browser is a good choice. We'll keep the amount of browser-specific commands (like `alert`) to a minimum so that you don't spend time on them if you plan to concentrate on another environment (like Node.js). We'll focus on JavaScript in the browser in the next part of the tutorial.

So first, let's see how we attach a script to a webpage. For server-side environments (like Node.js), you can execute the script with a command like `"node my.js"`.

The "script" tag

JavaScript programs can be inserted almost anywhere into an HTML document using the `<script>` tag.

For instance:

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Before the script...</p>

  <script>
    alert( 'Hello, world!' );
  </script>

  <p>...After the script.</p>

</body>

</html>
```

You can run the example by clicking the "Play" button in the right-top corner of the box above.

The `<script>` tag contains JavaScript code which is automatically executed when the browser processes the tag.

Modern markup

The `<script>` tag has a few attributes that are rarely used nowadays but can still be found in old code:

The `type` attribute: `<script type=…>`

The old HTML standard, HTML4, required a script to have a `type`. Usually it was `type="text/javascript"`. It's not required anymore. Also, the modern HTML standard totally changed the meaning of this attribute. Now, it can be used for JavaScript modules. But that's an advanced topic, we'll talk about modules in another part of the tutorial.

The `language` attribute: `<script language=…>`

This attribute was meant to show the language of the script. This attribute no longer makes sense because JavaScript is the default language. There is no need to use it.

Comments before and after scripts.

In really ancient books and guides, you may find comments inside `<script>` tags, like this:

```
<script type="text/javascript"><!--
    ...
//--></script>
```

This trick isn't used in modern JavaScript. These comments hide JavaScript code from old browsers that didn't know how to process the `<script>` tag. Since browsers released in the last 15 years don't have this issue, this kind of comment can help you identify really old code.

External scripts

If we have a lot of JavaScript code, we can put it into a separate file.

Script files are attached to HTML with the `src` attribute:

```
<script src="/path/to/script.js"></script>
```

Here, `/path/to/script.js` is an absolute path to the script from the site root. One can also provide a relative path from the current page. For instance, `src="script.js"` would mean a file `"script.js"` in the current folder.

We can give a full URL as well. For instance:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/l
```

To attach several scripts, use multiple tags:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
…
```

Please note:

As a rule, only the simplest scripts are put into HTML. More complex ones reside in separate files.

The benefit of a separate file is that the browser will download it and store it in its cache.

Other pages that reference the same script will take it from the cache instead of downloading it, so the file is actually downloaded only once.

That reduces traffic and makes pages faster.

If `src` is set, the script content is ignored.

A single `<script>` tag can't have both the `src` attribute and code inside.

This won't work:

```
<script src="file.js">
  alert(1); // the content is ignored, because src is set
</script>
```

We must choose either an external `<script src="…">` or a regular `<script>` with code.

The example above can be split into two scripts to work:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Summary

- We can use a `<script>` tag to add JavaScript code to a page.
- The `type` and `language` attributes are not required.
- A script in an external file can be inserted with `<script src="path/to/script.js"></script>`.

There is much more to learn about browser scripts and their interaction with the webpage. But let's keep in mind that this part of the tutorial is devoted to the JavaScript language, so we shouldn't distract ourselves with browser-specific implementations of it. We'll be using the browser as a way to run JavaScript, which is very convenient for online reading, but only one of many.

Previous lesson Next lesson

Share

Tutorial map

29th May 2021

Code structure

The first thing we'll study is the building blocks of code.

Statements

Statements are syntax constructs and commands that perform actions.

We've already seen a statement, `alert('Hello, world!')`, which shows the message "Hello, world!".

We can have as many statements in our code as we want. Statements can be separated with a semicolon.

For example, here we split "Hello World" into two alerts:

```
alert('Hello'); alert('World');
```

Usually, statements are written on separate lines to make the code more readable:

```
alert('Hello');
alert('World');
```

Semicolons

A semicolon may be omitted in most cases when a line break exists.

This would also work:

```
alert('Hello')
alert('World')
```

Here, JavaScript interprets the line break as an "implicit" semicolon. This is called an automatic semicolon insertion.

In most cases, a newline implies a semicolon. But "in most cases" does not mean "always"!

There are cases when a newline does not mean a semicolon. For example:

```
alert(3 +
1
+ 2);
```

The code outputs `6` because JavaScript does not insert semicolons here. It is intuitively obvious that if the line ends with a plus `"+"`, then it is an "incomplete expression", so a semicolon there would be incorrect. And in this case, that works as intended.

But there are situations where JavaScript "fails" to assume a semicolon where it is really needed.

Errors which occur in such cases are quite hard to find and fix.

An example of an error

If you're curious to see a concrete example of such an error, check this code out:

```
1  alert("Hello");
2
3  [1, 2].forEach(alert);
```

No need to think about the meaning of the brackets `[]` and `forEach` yet. We'll study them later. For now, just remember the result of running the code: it shows `Hello`, then `1`, then `2`.

Now let's remove the semicolon after the `alert`:

```
1  alert("Hello")
2
3  [1, 2].forEach(alert);
```

The difference compared to the code above is only one character: the semicolon at the end of the first line is gone.

If we run this code, only the first `Hello` shows (and there's an error, you may need to open the console to see it). There are no numbers any more.

That's because JavaScript does not assume a semicolon before square brackets `[...]`. So, the code in the last example is treated as a single statement.

Here's how the engine sees it:

```
1  alert("Hello")[1, 2].forEach(alert);
```

Looks weird, right? Such merging in this case is just wrong. We need to put a semicolon after `alert` for the code to work correctly.

This can happen in other situations also.

We recommend putting semicolons between statements even if they are separated by newlines. This rule is widely adopted by the community. Let's note once again – *it is possible* to leave out semicolons most of the time. But it's safer – especially for a beginner – to use them.

Comments

As time goes on, programs become more and more complex. It becomes necessary to add *comments* which describe what the code does and why.

Comments can be put into any place of a script. They don't affect its execution because the engine simply ignores them.

One-line comments start with two forward slash characters `//`.

The rest of the line is a comment. It may occupy a full line of its own or follow a statement.

Like here:

```
1  // This comment occupies a line of its own
2  alert('Hello');
3
4  alert('World'); // This comment follows the statement
```

Multiline comments start with a forward slash and an asterisk `/*` and end with an asterisk and a forward slash `*/`.

Like this:

```
/* An example with two messages.
This is a multiline comment.
*/
alert('Hello');
alert('World');
```

The content of comments is ignored, so if we put code inside `/* … */`, it won't execute.

Sometimes it can be handy to temporarily disable a part of code:

```
/* Commenting out the code
alert('Hello');
*/
alert('World');
```

Use hotkeys!

In most editors, a line of code can be commented out by pressing the `Ctrl+/` hotkey for a single-line comment and something like `Ctrl+Shift+/` – for multiline comments (select a piece of code and press the hotkey). For Mac, try `Cmd` instead of `Ctrl` and `Option` instead of `Shift`.

Nested comments are not supported!

There may not be `/*...*/` inside another `/*...*/`.

Such code will die with an error:

```
/*
  /* nested comment ?!? */
*/
alert( 'World' );
```

Please, don't hesitate to comment your code.

Comments increase the overall code footprint, but that's not a problem at all. There are many tools which minify code before publishing to a production server. They remove comments, so they don't appear in the working scripts. Therefore, comments do not have negative effects on production at all.

Later in the tutorial there will be a chapter Code quality that also explains how to write better comments.

Previous lesson Next lesson

Share

Tutorial map

1st December 2020

Variables

Most of the time, a JavaScript application needs to work with information. Here are two examples:

1. An online shop – the information might include goods being sold and a shopping cart.
2. A chat application – the information might include users, messages, and much more.

Variables are used to store this information.

A variable

A variable is a "named storage" for data. We can use variables to store goodies, visitors, and other data.

To create a variable in JavaScript, use the `let` keyword.

The statement below creates (in other words: *declares*) a variable with the name "message":

```
let message;
```

Now, we can put some data into it by using the assignment operator `=`:

```
let message;

message = 'Hello'; // store the string
```

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

```
let message;
message = 'Hello!';

alert(message); // shows the variable content
```

To be concise, we can combine the variable declaration and assignment into a single line:

```
let message = 'Hello!'; // define the variable and assign the value

```

```
3  alert(message); // Hello!
```

We can also declare multiple variables in one line:

```
1  let user = 'John', age = 25, message = 'Hello';
```

That might seem shorter, but we don't recommend it. For the sake of better readability, please use a single line per variable.

The multiline variant is a bit longer, but easier to read:

```
1  let user = 'John';
2  let age = 25;
3  let message = 'Hello';
```

Some people also define multiple variables in this multiline style:

```
1  let user = 'John',
2    age = 25,
3    message = 'Hello';
```

...Or even in the "comma-first" style:

```
1  let user = 'John'
2    , age = 25
3    , message = 'Hello';
```

Technically, all these variants do the same thing. So, it's a matter of personal taste and aesthetics.

`var` instead of `let`

In older scripts, you may also find another keyword: `var` instead of `let`:

```
1  var message = 'Hello';
```

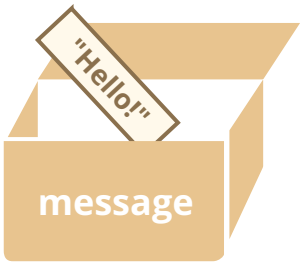
The `var` keyword is *almost* the same as `let`. It also declares a variable, but in a slightly different, "old-school" way.

There are subtle differences between `let` and `var`, but they do not matter for us yet. We'll cover them in detail in the chapter The old "var".

A real-life analogy

We can easily grasp the concept of a "variable" if we imagine it as a "box" for data, with a uniquely-named sticker on it.

For instance, the variable `message` can be imagined as a box labeled `"message"` with the value `"Hello!"` in it:

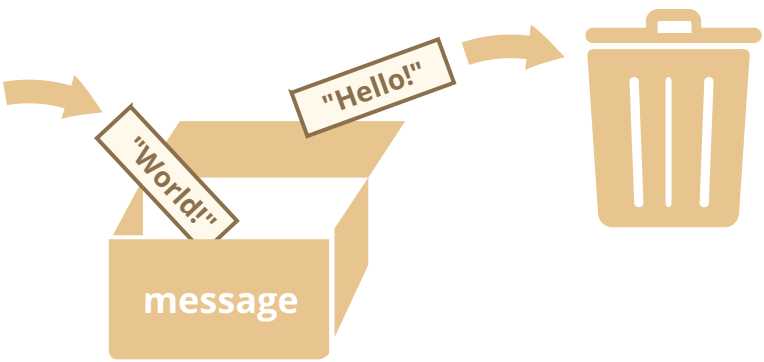


We can put any value in the box.

We can also change it as many times as we want:

```
let message;

message = 'Hello!';

message = 'World!'; // value changed

alert(message);
```

When the value is changed, the old data is removed from the variable:



We can also declare two variables and copy data from one into the other.

```
let hello = 'Hello world!';

let message;

// copy 'Hello world' from hello into message
message = hello;

// now two variables hold the same data
alert(hello); // Hello world!
alert(message); // Hello world!
```

Declaring twice triggers an error

A variable should be declared only once.

A repeated declaration of the same variable is an error:

```
let message = "This";

// repeated 'let' leads to an error
let message = "That"; // SyntaxError: 'message' has already been dec
```

So, we should declare a variable once and then refer to it without `let`.

Functional languages

It's interesting to note that there exist functional programming languages, like Scala or Erlang that forbid changing variable values.

In such languages, once the value is stored "in the box", it's there forever. If we need to store something else, the language forces us to create a new box (declare a new variable). We can't reuse the old one.

Though it may seem a little odd at first sight, these languages are quite capable of serious development. More than that, there are areas like parallel computations where this limitation confers certain benefits. Studying such a language (even if you're not planning to use it soon) is recommended to broaden the mind.

Variable naming

There are two limitations on variable names in JavaScript:

1. The name must contain only letters, digits, or the symbols `$` and `_`.
2. The first character must not be a digit.

Examples of valid names:

```
let userName;
let test123;
```

When the name contains multiple words, camelCase is commonly used. That is: words go one after another, each word except first starting with a capital letter: `myVeryLongName`.

What's interesting – the dollar sign `'$'` and the underscore `'_'` can also be used in names. They are regular symbols, just like letters, without any special meaning.

These names are valid:

```
let $ = 1; // declared a variable with the name "$"
let _ = 2; // and now a variable with the name "_"

alert($ + _); // 3
```

Examples of incorrect variable names:

```
let 1a; // cannot start with a digit

let my-name; // hyphens '-' aren't allowed in the name
```

Case matters

Variables named `apple` and `AppLE` are two different variables.

Non-Latin letters are allowed, but not recommended

It is possible to use any language, including cyrillic letters or even hieroglyphs, like this:

```
let имя = '...';
let 我 = '...';
```

Technically, there is no error here. Such names are allowed, but there is an international convention to use English in variable names. Even if we're writing a small script, it may have a long life ahead. People from other countries may need to read it some time.

Reserved names

There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

For example: `let`, `class`, `return`, and `function` are reserved.

The code below gives a syntax error:

```
let let = 5; // can't name a variable "let", error!
let return = 5; // also can't name it "return", error!
```

An assignment without `use strict`

Normally, we need to define a variable before using it. But in the old times, it was technically possible to create a variable by a mere assignment of the value without using `let`. This still works now if we don't put `use strict` in our scripts to maintain compatibility with old scripts.

```
// note: no "use strict" in this example

num = 5; // the variable "num" is created if it didn't exist

alert(num); // 5
```

This is a bad practice and would cause an error in strict mode:

```
"use strict";

num = 5; // error: num is not defined
```

Constants

To declare a constant (unchanging) variable, use `const` instead of `let`:

```
const myBirthday = '18.04.1982';
```

Variables declared using `const` are called "constants". They cannot be reassigned. An attempt to do so would cause an error:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

When a programmer is sure that a variable will never change, they can declare it with `const` to guarantee and clearly communicate that fact to everyone.

Uppercase constants

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution.

Such constants are named using capital letters and underscores.

For instance, let's make constants for colors in so-called "web" (hexadecimal) format:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Benefits:

- `COLOR_ORANGE` is much easier to remember than `"#FF7F00"`.
- It is much easier to mistype `"#FF7F00"` than `COLOR_ORANGE`.
- When reading the code, `COLOR_ORANGE` is much more meaningful than `#FF7F00`.

When should we use capitals for a constant and when should we name it normally? Let's make that clear.

Being a "constant" just means that a variable's value never changes. But there are constants that are known prior to execution (like a hexadecimal value for red) and there are constants that are *calculated* in run-time, during the execution, but do not change after their initial assignment.

For instance:

```
const pageLoadTime = /* time taken by a webpage to load */;
```

The value of `pageLoadTime` is not known prior to the page load, so it's named normally. But it's still a constant because it doesn't change after assignment.

In other words, capital-named constants are only used as aliases for "hard-coded" values.

Name things right

Talking about variables, there's one more extremely important thing.

A variable name should have a clean, obvious meaning, describing the data that it stores.

Variable naming is one of the most important and complex skills in programming. A quick glance at variable names can reveal which code was written by a beginner versus an experienced developer.

In a real project, most of the time is spent modifying and extending an existing code base rather than writing something completely separate from scratch. When we return to some code after doing something else for a while, it's much easier to find information that is well-labeled. Or, in other words, when the variables have good names.

Please spend time thinking about the right name for a variable before declaring it. Doing so will repay you handsomely.

Some good-to-follow rules are:

- Use human-readable names like `userName` or `shoppingCart`.
- Stay away from abbreviations or short names like `a`, `b`, `c`, unless you really know what you're doing.

- Make names maximally descriptive and concise. Examples of bad names are `data` and `value`. Such names say nothing. It's only okay to use them if the context of the code makes it exceptionally obvious which data or value the variable is referencing.
- Agree on terms within your team and in your own mind. If a site visitor is called a "user" then we should name related variables `currentUser` or `newUser` instead of `currentVisitor` or `newManInTown`.

Sounds simple? Indeed it is, but creating descriptive and concise variable names in practice is not. Go for it.

Reuse or create?

And the last note. There are some lazy programmers who, instead of declaring new variables, tend to reuse existing ones.

As a result, their variables are like boxes into which people throw different things without changing their stickers. What's inside the box now? Who knows? We need to come closer and check.

Such programmers save a little bit on variable declaration but lose ten times more on debugging.

An extra variable is good, not evil.

Modern JavaScript minifiers and browsers optimize code well enough, so it won't create performance issues. Using different variables for different values can even help the engine optimize your code.

Summary

We can declare variables to store data by using the `var`, `let`, or `const` keywords.

- `let` – is a modern variable declaration.
- `var` – is an old-school variable declaration. Normally we don't use it at all, but we'll cover subtle differences from `let` in the chapter The old "var", just in case you need them.
- `const` – is like `let`, but the value of the variable can't be changed.

Variables should be named in a way that allows us to easily understand what's inside them.

Tasks

Working with variables

importance: 2

1. Declare two variables: `admin` and `name`.
2. Assign the value `"John"` to `name`.
3. Copy the value from `name` to `admin`.
4. Show the value of `admin` using `alert` (must output "John").

solution

In the code below, each line corresponds to the item in the task list.

```
let admin, name; // can declare two variables at once

name = "John";

admin = name;

alert( admin ); // "John"
```

Giving the right name

importance: 3

1. Create a variable with the name of our planet. How would you name such a variable?
2. Create a variable to store the name of a current visitor to a website. How would you name that variable?

solution

The variable for our planet

The name of the current visitor

Uppercase const?

importance: 4

Examine the following code:

```
const birthday = '18.04.1982';

const age = someCode(birthday);
```

Here we have a constant `birthday` date and the `age` is calculated from `birthday` with the help of some code (it is not provided for shortness, and because details don't matter here).

Would it be right to use upper case for `birthday` ? For `age` ? Or even for both?

```
const BIRTHDAY = '18.04.1982'; // make uppercase?

const AGE = someCode(BIRTHDAY); // make uppercase?
```

solution

We generally use upper case for constants that are "hard-coded". Or, in other words, when the value is known prior to execution and directly written into the code.

In this code, `birthday` is exactly like that. So we could use the upper case for it.

In contrast, `age` is evaluated in run-time. Today we have one age, a year after we'll have another one. It is constant in a sense that it does not change through the code execution. But it is a bit "less of a constant" than `birthday`: it is calculated, so we should keep the lower case for it.

Previous lesson

Next lesson

Share

Tutorial map

about the project contact us terms of usage privacy policy

codecademy

Introduction

JavaScript

JavaScript is a programming language that powers the dynamic behavior on most websites. Alongside HTML and CSS, it is a core technology that makes the web run.

Methods

Methods return information about an object, and are called by appending an instance with a period `.`, the method name, and parentheses.

```
// Returns a number between 0 and 1
Math.random();
```

Libraries

Libraries contain methods that can be called by appending the library name with a period `.`, the method name, and a set of parentheses.

```
Math.random();
// ☝ Math is the library
```

`console.log()`

The `console.log()` method is used to log or print messages to the console. It can also be used to print objects and other info.

```
console.log('Hi there!');
// Prints: Hi there!
```

Numbers

Numbers are a primitive data type. They include the set of all integers and floating point numbers.

```
let amount = 6;
let price = 4.99;
```

String `.length`

The `.length` property of a string returns the number of characters that make up the string.

```
let message = 'good nite';
console.log(message.length);
// Prints: 9

console.log('howdy'.length);
// Prints: 5
```

Data Instances

When a new piece of data is introduced into a JavaScript program, the program keeps track of it in an instance of that data type. An instance is an individual case of a data type.

Booleans

Booleans are a primitive data type. They can be either `true` or `false`.

```
let lateToWork = true;
```

`Math.random()`

The `Math.random()` function returns a floating-point, random number in the range from 0 (inclusive) up to but not including 1.

```
console.log(Math.random());
// Prints: 0 - 0.9
```

`Math.floor()`

The `Math.floor()` function returns the largest integer less than or equal to the given number.

```
console.log(Math.floor(5.95));
// Prints: 5
```

Single Line Comments

In JavaScript, single-line comments are created with two consecutive forward slashes `//`.

```
// This line will denote a comment
```

Null

Null is a primitive data type. It represents the intentional absence of value. In code, it is represented as `null`.

```
let x = null;
```

Strings

Strings are a primitive data type. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes `'` or double quotes `"`.

```
let single = 'Wheres my bandit hat?';
let double = "Wheres my bandit hat?";
```

Arithmetic Operators

JavaScript supports arithmetic operators for:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division
- `%` modulo

```
// Addition
5 + 5
// Subtraction
10 - 5
// Multiplication
5 * 10
// Division
10 / 5
// Modulo
10 % 5
```

Multi-line Comments

In JavaScript, multi-line comments are created by surrounding the lines with `/*` at the beginning and `*/` at the end. Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

```
/*
The below configuration must be
changed before deployment.
*/

let baseUrl
= 'localhost/taxwebapp/country';
```

Remainder / Modulo Operator

The remainder operator, sometimes called modulo, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can.

```
// calculates # of weeks in a year,
rounds down to nearest integer
const weeksInYear
= Math.floor(365/7);

// calcuates the number of days left
over after 365 is divded by 7
const daysLeftOver = 367 % 7 ;

console.log("A year has "
+ weeksInYear + " weeks and "
+ daysLeftOver + " days");
```

Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. Here are some of them:

- `+=` addition assignment
- `-=` subtraction assignment
- `*=` multiplication assignment
- `/=` division assignment

```
let number = 100;

// Both statements will add 10
number = number + 10;
number += 10;

console.log(number);
// Prints: 120
```

String Interpolation

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc).
It can be performed using template literals: `text ${expression} text`.

```
let age = 7;

// String concatenation
'Tommy is ' + age + ' years old.';

// String interpolation
`Tommy is ${age} years old.`;
```

Variables

Variables are used whenever there's a need to store a piece of data. A variable contains data that can be used in the program elsewhere. Using variables also ensures code re-usability since it can be used to replace the same value in multiple places.

```
const currency = '$';
let userIncome = 85000;

console.log(currency + userIncome + '
is more than the average income.');
// Prints: $85000 is more than the
average income.
```

Undefined

`undefined` is a primitive JavaScript value that represents lack of defined value. Variables that are declared but not initialized to a value will have the value `undefined`.

```
var a;

console.log(a);
// Prints: undefined
```

Learn Javascript: Variables

codecademy

A variable is a container for data that is stored in computer memory. It is referenced by a descriptive name that a programmer can call to assign a specific value and retrieve it.

```
// examples of variables
let name = "Tammy";
const found = false;
var age = 3;
console.log(name, found, age);
// Tammy, false, 3
```

Declaring Variables

To declare a variable in JavaScript, any of these three keywords can be used along with a variable name:

- `var` is used in pre-ES6 versions of JavaScript.
- `let` is the preferred way to declare a variable when it can be reassigned.
- `const` is the preferred way to declare a variable with a constant value.

```
var age;
let weight;
const numberOfFingers = 20;
```

Template Literals

Template literals are strings that allow embedded expressions, `${expression}`. While regular strings use single `'` or double `"` quotes, template literals use backticks instead.

```
let name = "Codecademy";
console.log(`Hello, ${name}`);
// Prints: Hello, Codecademy

console.log(`Billy is ${6+8} years
old.`);
// Prints: Billy is 14 years old.
```

`let` Keyword

`let` creates a local variable in JavaScript & can be re-assigned. Initialization during the declaration of a `let` variable is optional. A `let` variable will contain `undefined` if nothing is assigned to it.

```
let count;
console.log(count); // Prints:
undefined
count = 10;
console.log(count); // Prints: 10
```

`const` Keyword

A constant variable can be declared using the keyword `const`. It must have an assignment. Any attempt of re-assigning a `const` variable will result in JavaScript runtime error.

```
const numberOfColumns = 4;
numberOfColumns = 8;
// TypeError: Assignment to constant
variable.
```

String Concatenation

In JavaScript, multiple strings can be concatenated together using the `+` operator. In the example, multiple strings and variables containing string values have been concatenated. After execution of the code block, the `displayText` variable will contain the concatenated string.

```
let service = 'credit card';
let month = 'May 30th';
let displayText = 'Your '
+ service  + ' bill is due on '
+  month + '.';

console.log(displayText);
// Prints: Your credit card bill is
due on May 30th.
```

codecademy

Conditionals

Control Flow

Control flow is the order in which statements are executed in a program. The default control flow is for statements to be read and executed in order from left-to-right, top-to-bottom in a program file.
Control structures such as conditionals (`if` statements and the like) alter control flow by only executing blocks of code if certain conditions are met. These structures essentially allow a program to make decisions about which code is executed as the program runs.

Logical Operator | |

The logical OR operator `||` checks two values and returns a boolean. If one or both values are truthy, it returns `true` . If both values are falsy, it returns `false` .

A	B	A || B
false	false	false
false	true	true
true	false	true
true	true	true

```
true || false;         // true
10 > 5 || 10 > 20;     // true
false || false;        // false
10 > 100 || 10 > 20;   // false
```

Ternary Operator

The ternary operator allows for a compact syntax in the case of binary (choosing between two choices) decisions. It accepts a condition followed by a `?` operator, and then two expressions separated by a `:` . If the condition evaluates to truthy, the first expression is executed, otherwise, the second expression is executed.

```
let price = 10.5;
let day = "Monday";

day === "Monday" ? price -= 1.5
: price += 1.5;
```

`else` Statement

An `else` block can be added to an `if` block or series of `if` - `else if` blocks. The `else` block will be executed only if the `if` condition fails.

```
const isTaskCompleted = false;

if (isTaskCompleted) {
  console.log('Task completed');
} else {
  console.log('Task incomplete');
}
```

Logical Operator `&&`

The logical AND operator `&&` checks two values and returns a boolean. If *both* values are truthy, then it returns `true`. If one, or both, of the values is falsy, then it returns `false`.

```
true && true;      // true
1 > 2 && 2 > 1;    // false
true && false;     // false
4 === 4 && 3 > 1;  // true
```

`switch` Statement

The `switch` statements provide a means of checking an expression against multiple `case` clauses. If a case matches, the code inside that clause is executed.
The `case` clause should finish with a `break` keyword.
If no case matches but a `default` clause is included, the code inside `default` will be executed.
Note: If `break` is omitted from the block of a `case`, the `switch` statement will continue to check against `case` values until a break is encountered or the flow is broken.

```
const food = 'salad';

switch (food) {
  case 'oyster':
    console.log('The taste of the sea
🦪');
    break;
  case 'pizza':
    console.log('A delicious pie 🍕');
    break;
  default:
    console.log('Enjoy your meal');
}

// Prints: Enjoy your meal
```

`if` Statement

An `if` statement accepts an expression with a set of parentheses:

- If the expression evaluates to a truthy value, then the code within its code body executes.
- If the expression evaluates to a falsy value, its code body will not execute.

```
const isMailSent = true;

if (isMailSent) {
  console.log('Mail sent to
recipient');
}
```

Truthy and Falsy

In JavaScript, values evaluate to `true` or `false` when evaluated as Booleans.

- Values that evaluate to `true` are known as *truthy*
- Values that evaluate to `false` are known as *falsy*

Falsy values include `false`, `0`, empty strings, `null` `undefined`, and `NaN`. All other values are truthy.

Logical Operator `!`

The logical NOT operator `!` can be used to do one of the following:

- Invert a Boolean value.
- Invert the truthiness of non-Boolean values.

```
let lateToWork = true;
let oppositeValue = !lateToWork;

console.log(oppositeValue);
// Prints: false
```

Comparison Operators

Comparison operators are used to comparing two values and return `true` or `false` depending on the validity of the comparison:

- `===` strict equal
- `!==` strict not equal
- `>` greater than
- `>=` greater than or equal
- `<` less than
- `<=` less than or equal

```
1 > 3       // false
3 > 1       // true
250 >= 250  // true
1 === 1     // true
1 === 2     // false
1 === '1'   // false
```

`else if` Clause

After an initial `if` block, `else if` blocks can each check an additional condition. An optional `else` block can be added after the `else if` block(s) to run by default if none of the conditionals evaluated to truthy.

```
const size = 10;

if (size > 100) {
  console.log('Big');
} else if (size > 20) {
  console.log('Medium');
} else if (size > 4) {
  console.log('Small');
} else {
  console.log('Tiny');
}
// Print: Small
```

codecademy

Functions

Arrow Functions (ES6)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the `function` keyword and uses a fat arrow `=>` to separate the parameter(s) from the body.
There are several variations of arrow functions:

- Arrow functions with a single parameter do not require `()` around the parameter list.
- Arrow functions with a single expression can use the concise function body which returns the result of the expression without the `return` keyword.

```
// Arrow function with two arguments
const sum = (firstParam, secondParam)
=> {
  return firstParam + secondParam;
};
console.log(sum(2,5)); // Prints: 7

// Arrow function with no arguments
const printHello = () => {
  console.log('hello');
};
printHello(); // Prints: hello

// Arrow functions with a single
argument
const checkWeight = weight => {
  console.log(`Baggage weight
: ${weight} kilograms.`);
};
checkWeight(25); // Prints: Baggage
weight : 25 kilograms.


// Concise arrow functions
const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); //
Prints: 60
```

Functions

Functions are one of the fundamental building blocks in JavaScript. A *function* is a reusable set of statements to perform a task or calculate a value. Functions can be passed one or more values and can return a value at the end of their execution. In order to use a function, you must define it somewhere in the scope where you wish to call it.
The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function:
sum(3, 6); // 9
```

Anonymous Functions

Anonymous functions in JavaScript do not have a name property. They can be defined using the `function` keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

```
// Named function
function rocketToMars() {
  return 'BOOM!';
}

// Anonymous function
const rocketToMars = function() {
  return 'BOOM!';
}
```

Function Expressions

Function *expressions* create functions inside an expression instead of as a function declaration. They can be anonymous and/or assigned to a variable.

```
const dog = function() {
  return 'Woof!';
}
```

Function Parameters

Inputs to functions are known as *parameters* when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is *passed* in as arguments. It is possible to define a function without parameters.

```
// The parameter is name
function sayHello(name) {
  return `Hello, ${name}!`;
}
```

`return` Keyword

Functions return (pass back) values using the `return` keyword. `return` ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the `return` keyword, in which case the function will return `undefined` by default.

```
// With return
function sum(num1, num2) {
  return num1 + num2;
}

// Without return, so the function
doesn't output the sum
function sum(num1, num2) {
  num1 + num2;
}
```

Function Declaration

Function *declarations* are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

- The `function` keyword.
- The function name.
- An optional list of parameters separated by commas enclosed by a set of parentheses `()`.
- A function body enclosed in a set of curly braces `{}`.

```
function add(num1, num2) {
  return num1 + num2;
}
```

Calling Functions

Functions can be *called*, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. *Arguments* are values passed into a function when it is called.

```
// Defining the function
function sum(num1, num2) {
  return num1 + num2;
}

// Calling the function
sum(2, 4); // 6
```