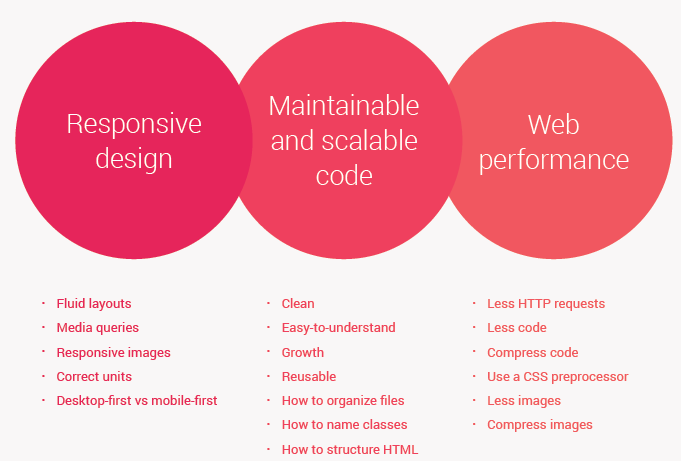
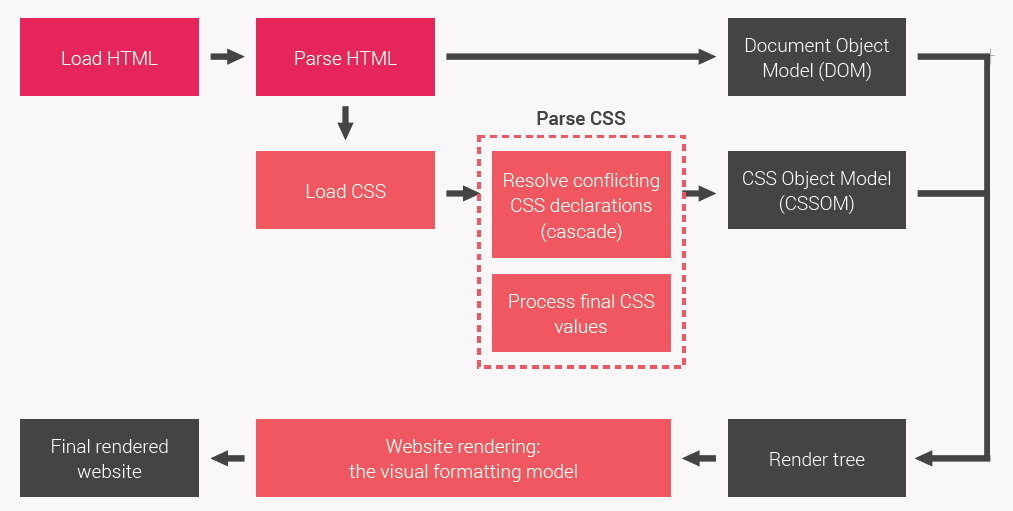
Advanced CSS and Sass

# How CSS Works

## Three pillars of writing good HTML and CSS.

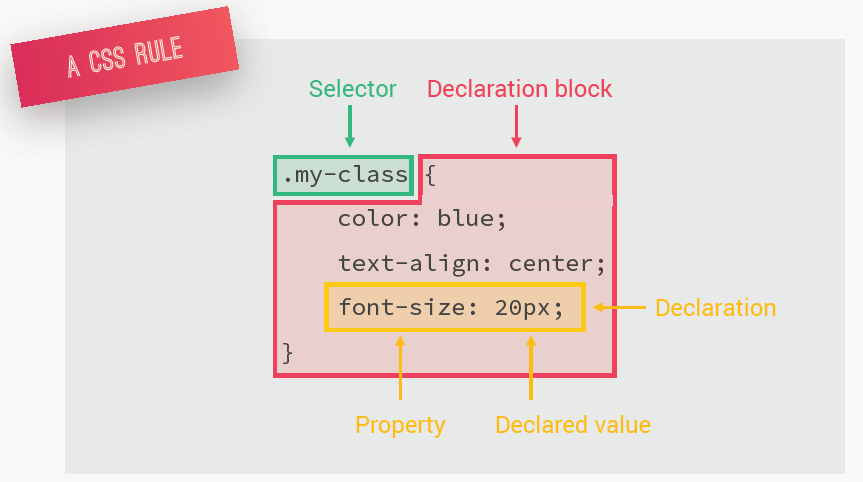


## What Happens To CSS When We Load Up A Webpage?

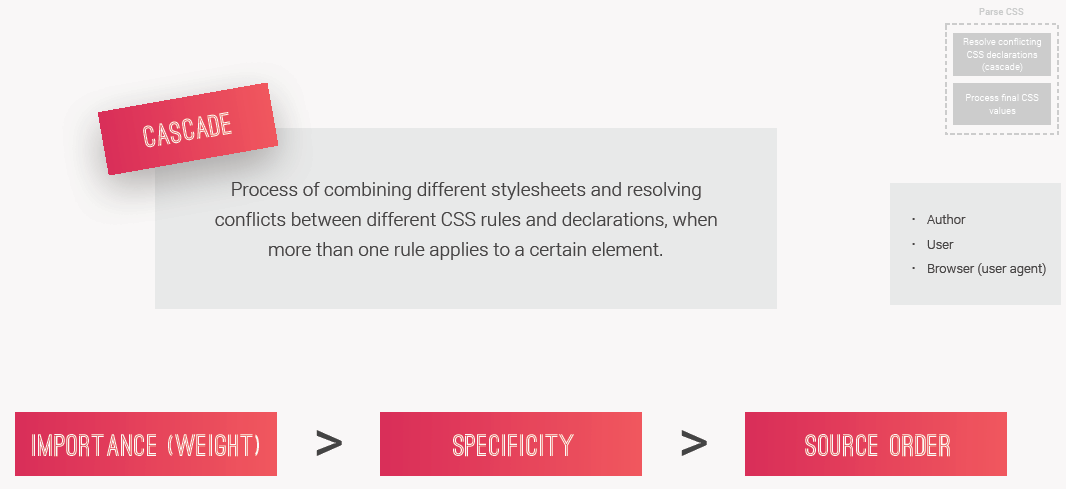


* The Process final CSS values step does for instance, converting a margin defined in percentage units to pixels.
* The Visual formatting Model algorithm calculates and uses a bunch of stuff that you already know about, like the box model, floats, and positioning.

## CSS Terminology



## How CSS is Parsed: Part 1 (Resolve conflicting CSS declaration Step)



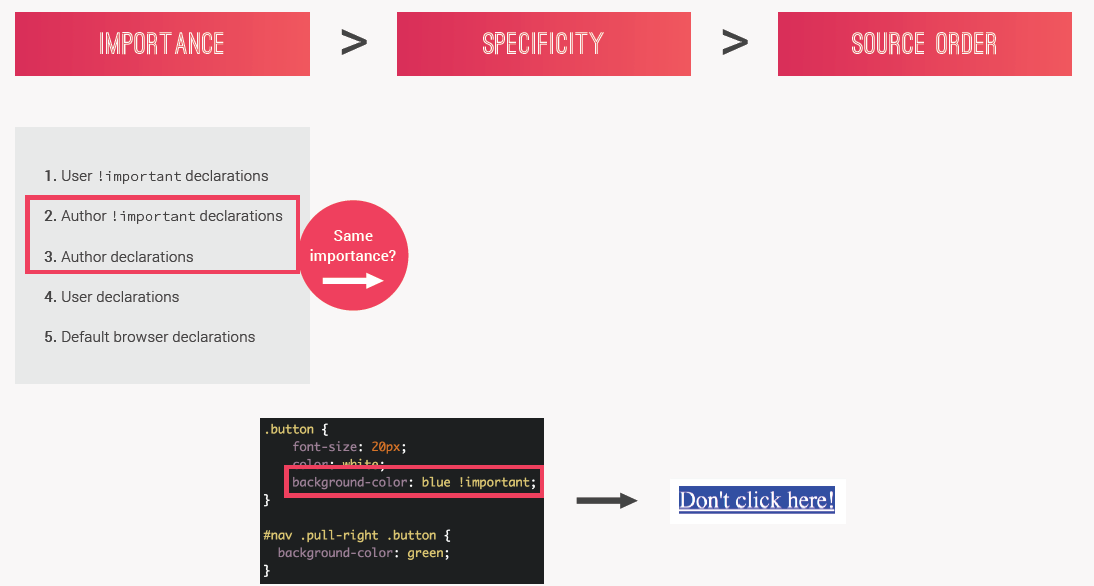
* Cascade is the process of combining different stylesheets and resolving conflicts between different CSS rules and declarations when more than one rule applies to a certain element.
* A CSS declaration for a certain style property like font size can appear in several stylesheets and also several times inside one single stylesheet.
* Also CSS can also come from different sources.
  + **Author** – The most common one is the CSS that we the developers write. These declarations that we put in our stylesheets. These are called the **author** declarations.
  + **User** **CSS** – which is CSS coming from the user. For instance, when the user changes the default font size in the browser.
  + **Default Browser (User Agent) CSS** – For instance, if we put an anchor tag in our HTML for a link and then don't style it at all, it will usually be rendered with blue text and underlined. That's called the user agent CSS, because it's set by the browser.
* The cascade combines the CSS declarations coming from all these different sources.

### How does the cascade actually resolve conflicts when more than one rule applies?

Cascade step uses below order to determine which CSS rule takes precedence.

1. Importance at the selector
2. Specificity
3. Sort order of the conflicting declarations

#### Importance

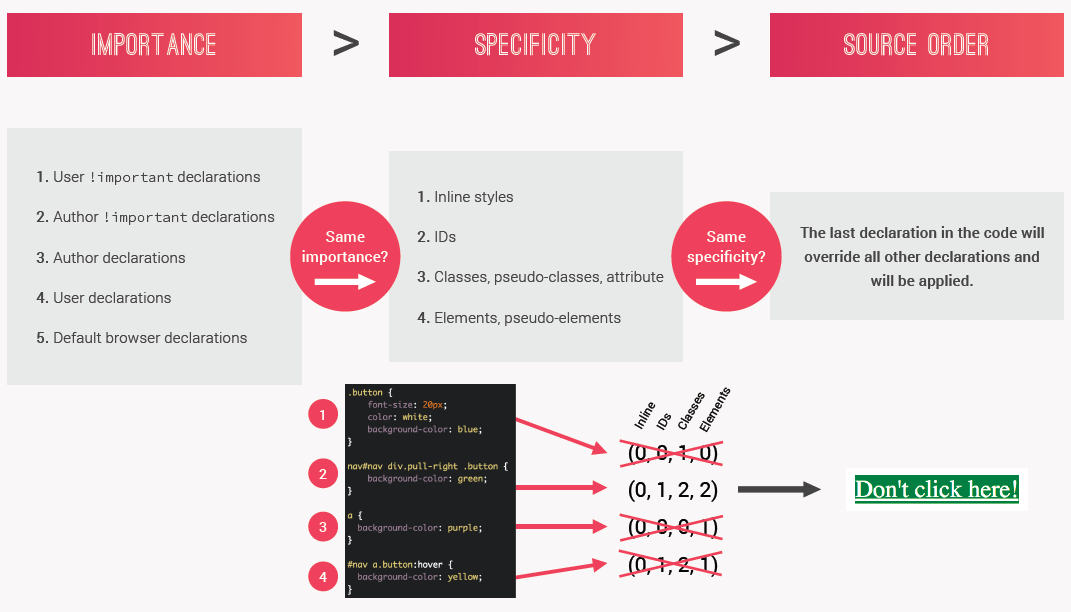


* First off the cascade starts by giving the conflicting declarations different importance's based on where they are declared, i.e. based on their source.
* The important declarations are the ones marked with the important keyword.
* Order from most important to least important declarations –

1. User !important declarations
2. Author !important declarations
3. Author declarations
4. User declarations
5. Default browser declarations

* E.g. As highlighted in above image the declaration with the blue background color contains the important keyword and therefore this declaration is more important. This means that this is the declaration that gets precedence and so we see that our button is indeed blue.

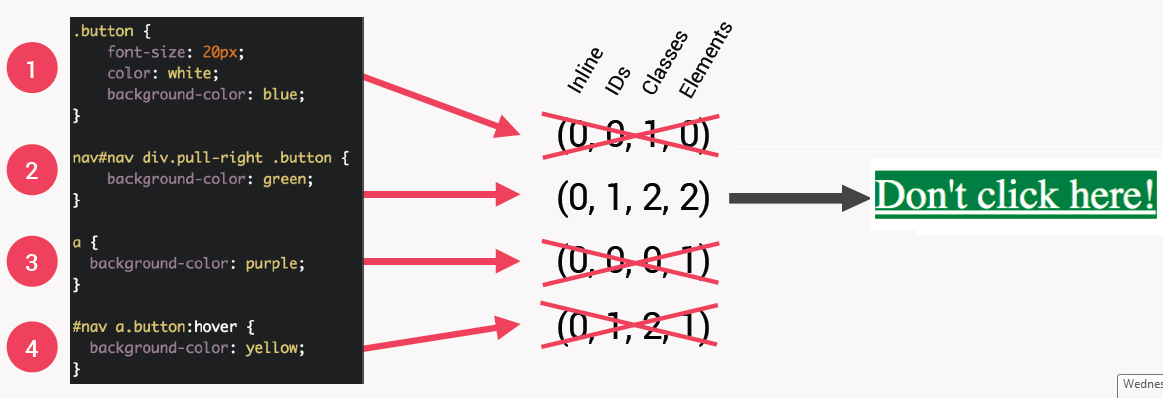
#### Specificity



* Most common scenario is when we have a bunch of conflicting rules in our **author** stylesheets without any important keyword. All the declarations have the exact same importance.
* Now what happens in this case?
* Priorities/Categories – **Inline** styles have the highest specificity followed by **IDs**, then **classes**, **pseudo classes and attribute** selectors, and finally the least specific is the **element** and **pseudo element** selector.
* So when we have conflicting declarations with the same importance we calculate their selector specificity based on above priorities.
* The **specificity** is actually not just one number, but one number for each of the above four categories

Inline styles, IDs, classes, pseudo elements and attributes, and finally elements and for each of these we count the number of occurrences in the selector.

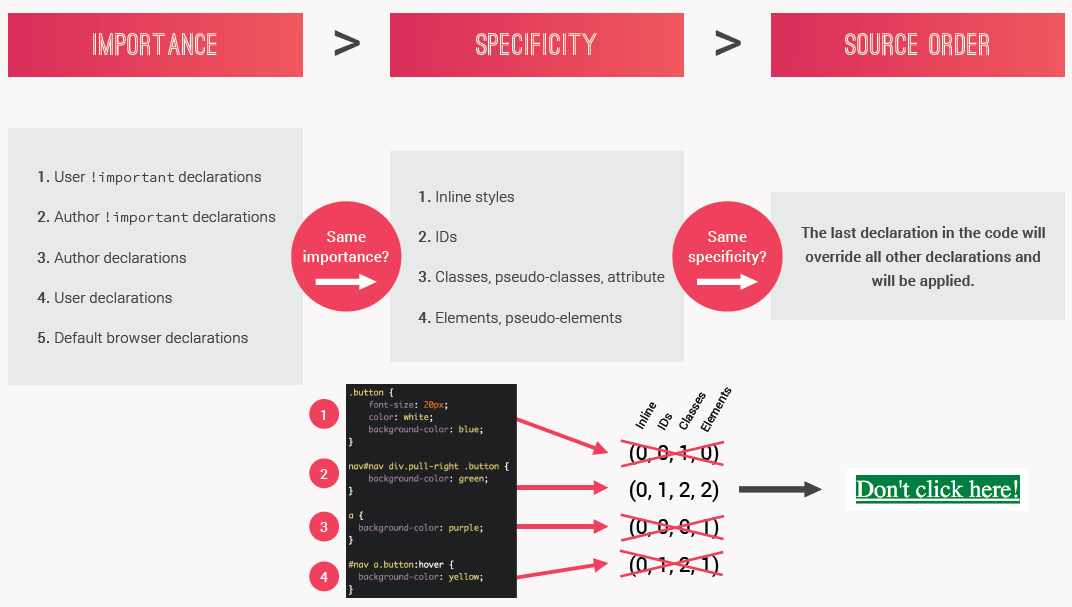
* Selector with maximum specificity wins and that style is applied. The value of the winning declaration is called the **cascaded value**, because it's the result from the cascade.
* E.g. As highlighted in black box above,



* + All these declarations have the same importance, because they're all author declarations without any important keyword.
  + For Selector 1 (highlighted in red circle), the specificity is (0, 0, 1, 0).  
    (0=> no inline style, 0 => no ID style, 1 => 1 button class style, 0 => no element style).
  + For Selector 2 (highlighted in red circle), the specificity is **(0, 1, 2, 2).**  
    (0=> no inline, 1=> 1 ID for #nav, 2=> 1 pull-right and 1 button class, 2=> 1 nav and 1 div element)
  + For Selector 3 (highlighted in red circle), the specificity is (0, 0, 0, 1).  
    (0=> no inline, 0=> no ID, 0=> no class, 1=> 1 anchor element)
  + For Selector 4 (highlighted in red circle), the specificity is (0, 1, 2, 1).  
    (0=> no inline, 1=> ID for #nav, 2=> 1 button class & 1 hover pseudo-class, 1=> 1 anchor el.)

#### Source Order

* We know that the Selector with maximum specificity wins and that style is applied. But what if more than one selector have same maximum specificity? (e.g. If we consider above example, image if 2 selectors have same max specificity as (0, 1, 2, 2).
* In this case, the last CSS declaration written in the code is applied.
* If everything is equal, if all the declarations selectors have the same specificity, then  
  the last declaration in the code will override all other declarations and will be applied.



#### Summary

* CSS declarations marked with !important have the highest priority;
* But, only use !important as a last resource. It’s better to use correct specificities — more maintainable code! If something doesn't work the way you think it should then look at your selector specificities and figure out what's going on. That's always better than just adding important, which may solve the problem in an easier way, but will cause you problems in the long run.
* Inline styles will always have priority over styles in external stylesheets;   
  Writing inline styles isn't a good practice.
* A selector that contains even 1 ID is more specific than one with 1000 classes.  
  If you want to be really specific use an ID.
* A selector that contains even 1 class is more specific than one with 1000 elements.
* By default, browsers apply a certain margin, and/or a certain padding, to some elements. And we don't want that, obviously for consistency. Hence we must declare Universal selector in our CSS (\* means Universal Selector). Along with real elements, we should also declare selector for all after and before pseudo-elements so that the declaration block will apply to all elements and we won’t have any inconsistent behaviour.  
  e.g. \*,

\*::after,

\*::before {

margin: 0;

padding: 0;

box-sizing: inherit;

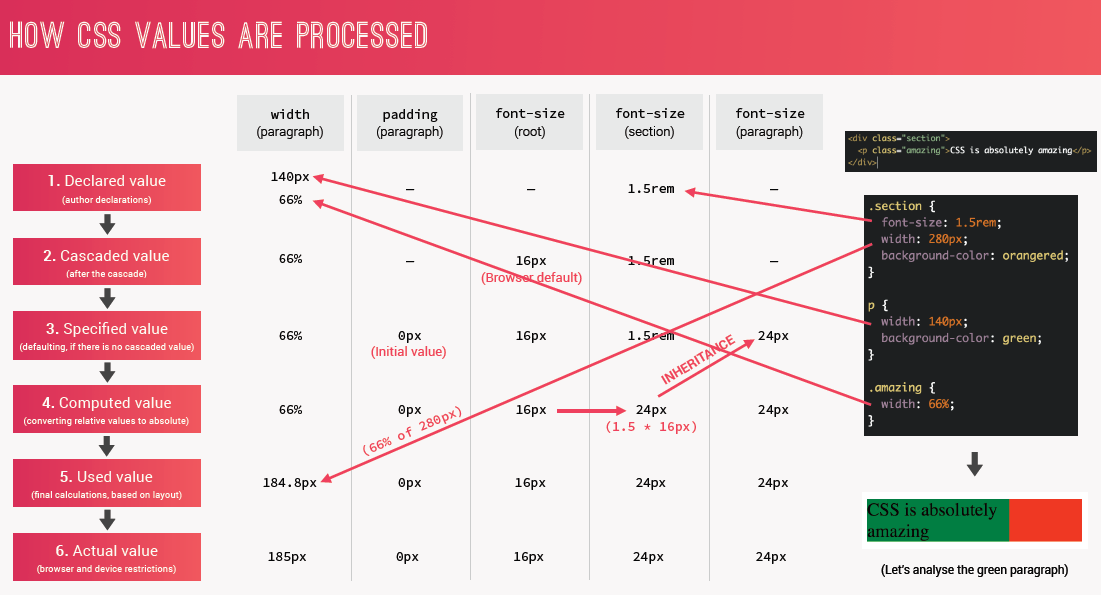
}

* The universal selector \* has no specificity value (0, 0, 0, 0) which means that all other selectors have a precedence over it.
* Rely more on specificity than on the order of selectors. This way if you someday rearrange all your CSS code you won't have any surprises which could mess up your entire code. And this makes your code a lot easier to maintain.
* Rely on order when using 3rd-party stylesheets — **always put your author stylesheet last**.

## How CSS is Parsed: Part 2 (Value Processing)

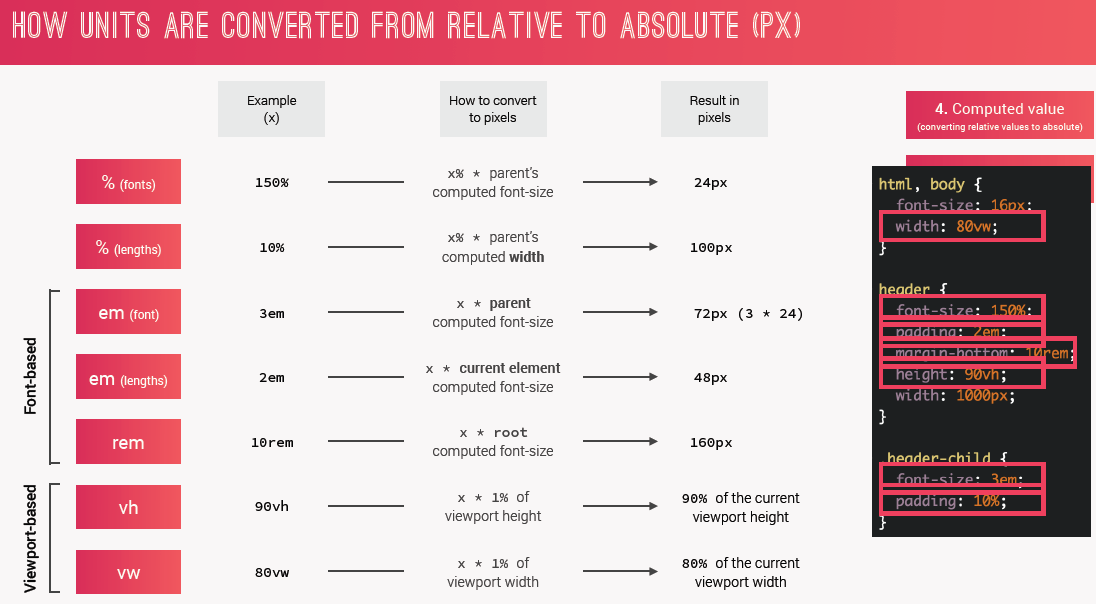
* Why is it important to know this?
* Two reasons –
  + Each time you use a unit other than pixels, like rem or vh or some other relative unit, it will ultimately be converted to pixels.
  + Also knowing this stuff, will is predict and control better which property takes which value in the end.

### How CSS Values Are Processed



* Let’s analyze the width, padding, and font-size as declared in above example.
* Cascaded Value – value calculated in the first step of Parsing CSS – Resolving Conflicts.
* Specified Value – the default value of a certain CSS property. (This is used if we don’t have any cascaded value already. Otherwise not used.)
* Computed Value –
  + In this step, values with relative units are converted to pixels so that they can be inherited.
  + CSS key words like “orange”, “oral”, “boulder” and a lot more are computed and replaced here in this step.
* Used Value – In this calculation step, the CSS engine uses the rendered layout to figure out some of the remaining values, for example percentage values that depend on the layout.
* Actual Value – browsers usually cannot really display decimal pixels e.g. 184.8 pixels. That's just way too specific and so usually, such is simply rounded. This is the final value which is used in the layout for rendering by considering the browser and device restrictions.
* Each and every CSS property needs to have a value even if we don't even declare it at all. That is where the “Specified Value” comes into picture. Each CSS property has something called an "initial value" which is simply the value used if there is no cascaded value.
* So basically, if we don't declare a value and if neither the browser nor the user define a value, then the initial value will be used. Actually, inheritance also plays a role here.
* Different properties have different initial values of course, and for padding, it's zero pixels, which is obvious.
* Root font-size means the overall font-size of a document.
* Browsers has default value of a root font-size for each page (usually 16px). But we can also set it in our HTML selector.
* Some properties like the ones related to text, such as font-size, inherit the computed value of their parent elements.

### How Units Are Converted From Relative to Absolute (px)



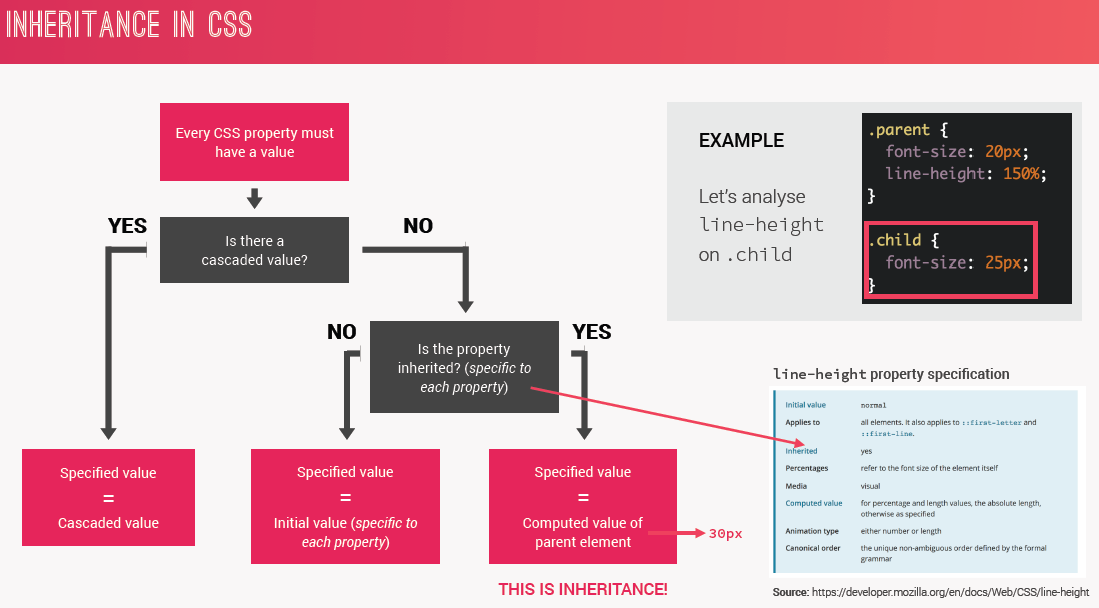
* It is important to know how the CSS engine converts relative units to pixels in order to calculate **Computed** and **Used** values. Because this is crucial to know when we write our own code so this has actually a very practical application in the real world and relative units are fundamental to build modern responsive layouts.
* There's a distinction between using percentages for fonts and for length/distance measurements.
* Percentages are measured relative to their parent’s **font-size, if used to specify font-size**.
* Percentages are measured relative to their parent’s **width, if used to specify lengths** (e.g. like height, padding, margin, etc.).
* Both **em** and **rem** are **font-based**, but the difference between them is that **em** use the parent or the current element as a reference while **rem** use the root font-size as the reference.
* If **em** is used for **font-sizes**, then the reference is simply the **parent's** computed font-size (similar to what happens with percentages).
* If **em** is used to specify **lengths,** then they are measured **relative to the** **current** font-size.
* **rem** unit is always relative to the root font-size.
* For **both font-sizes and lengths**, if **rem** unit is used, it is always relative to the **root font-size.**
* Why should we actually size (length) stuff with em and rem if they are based on font-size?
  + by doing so, we can build more robust responsive layouts because just by changing font sizes, we will automatically change length since it depend on a font size and that gives us a lot of flexibility, and it's just a great technique.
* 1**vh** is just 1% of the browser’s view port height.
* 1**vw** is just 1% of the browser’s view port width.
* **vh** is really useful to build nice hero sections for a webpage.

### Summary

* Each property has an initial value, used if nothing is declared (and if there is no inheritance — see next lecture)
* Browsers specify a root font-size for each page (usually 16px)
* Percentages and relative values are always converted to pixels in order for the CSS engine to be able to render the page on the screen.
* Percentages are measured relative to their parent’s font-size, if used to specify font-size.
* Percentages are measured relative to their parent’s width, if used to specify lengths.
* **em** are measured relative to their parent font-size, if used to specify font-size.
* **em** are measured relative to the current font-size, if used to specify lengths.
* **rem** are always measured relative to the document’s root font-size.
* **vh** and **vw** are simply percentage measurements of the viewport’s height and width.

## How CSS is Parsed: Part 3 (Inheritance)

Inheritance is a way of propagating property values from parent elements to their children.



### Summary

* Inheritance passes the values for some specific properties from parents to children — Inheritance allows the developers to write less code and code that will be more maintainable.
* Properties related to text are inherited: font-family, font-size, color, etc;
* Other properties like margins or paddings are, of course, not inherited.
* The computed value of a property is what gets inherited, not the declared value.
* Inheritance of a property only works if no one declares a value for that property;
* We can use the inherit keyword to force inheritance on a certain property;
* The initial keyword can be used to reset a property to its initial value.

## Converting px to rem

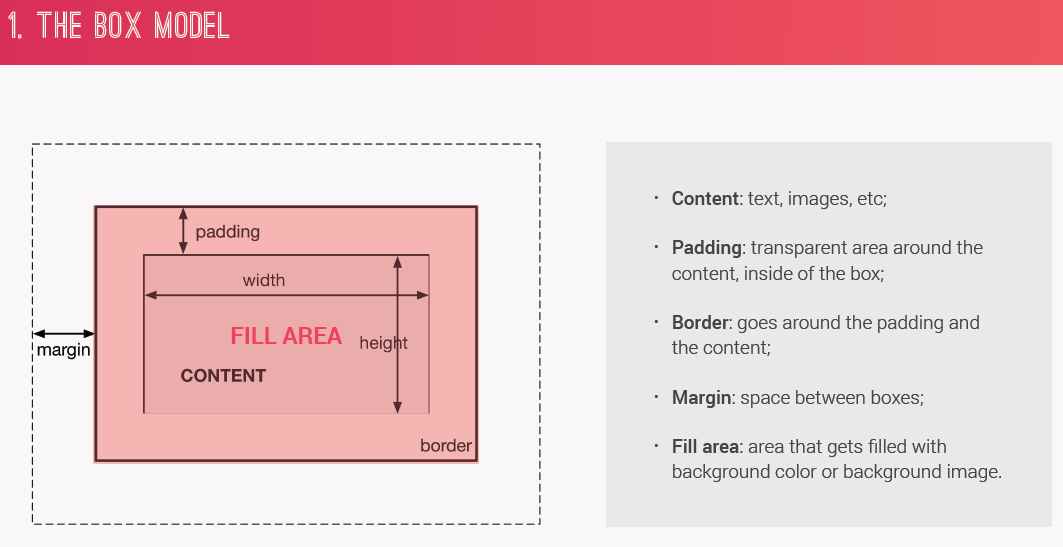
* We should use rem instead of px.
* Why?
  + That’s because we want an easy way to change all the measurements on our page with one simple setting.
  + For instance, when we hit a break point to display our page on a mobile device, we want a way to decrease all the measurements in our site at the same time, and instead of writing hundreds of lines of code and in media queries, we can just change one global setting, and that is the global font size.
  + We know that rem unit is always in relation to the root font size. Now if we now set that root font size, we can and very easily change all the other measurements on our page to rem. The root font size is set in the HTML selector.
* When we are doing responsive design and modern design, and modern CSS development, then it's really good practice to use relative units like rem.
* Using em is a whole lot of work because as we know em depends on its parent’s font-size so we have to do a lot of calculations and it becomes really, really difficult to manage.
* Note: rems are not supported below Internet Explorer 9.
* Root font size should be set/evaluate to 10px.
  + Why 10, bcz 1 rem = root font-size. So here 1 rem = 10px. So now it's easier for us to do calculations (w.r.t 10) while assigning rem based font-sizes/length units.
  + If it were say 16px, then we would have to do lot more calculations in each selector which takes away lot of our time.
* We should **NOT** set Root font-size in **px** but should be set in **percentage**.
  + The problem is that by doing so we actually override the browser font size setting that the user can manually change in the browser settings and many people actually do that. e.g. when they have bad sight and cannot see so good, then some people actually increase the default font size of their browser.
  + So if we set our ROOT font-size in px, then we remove the ability for these people to see our website properly because they can no longer change the default font size (they can, but our root font size will not be affected by that anymore).
  + So this is a really bad practice, and that is why we should set overall font size to a percentage, which will translate to a percentage of the font size given by the browser.
* Let’s take an example –
  + Now the default font size, if the user doesn't change anything is 16 pixels.   
    But we want root font-size to be 10px for our ease (why? As mentioned in one of above points – for easier calculations and assignment on rem based font-sizes/length units.
  + So Default browser font-size is 16px.
    - But we want root font-size to be 10 px, then 10/16 = 0.625 => 62.5%   
      So by default 62.5% translates to 10px (0.625 \* 16).
    - If user changes to 20px, then 62.5% translates to = 12.5px (0.625 \* 20).
    - If user changes to 22px, then 62.5% translates to 13.75px (0.625 \* 22).
  + So with this we allow the user to actually zoom in on the page automatically, not only to change the font size, but to also to zoom in on the page, because basically what happens is Not exactly zooming, but increasing all of the measurements that we have on our page.
  + This is really widely used technique in the CSS developer community because it's so simple and yet so powerful.
* Whenever we can use inheritance, we should use it instead of using something like the universal selector (\*).
* box-sizing property by itself is not inherited, but by setting the box-sizing property on universal selector (basically each and every element on the entire page) to inherit, it'll automatically inherit whatever we set the border-box property in **body** selector. So this is considered a slightly better practice by the CSS community because it makes it a bit easier to change the box-sizing like in plugins or other components where we might want to use a different property for box-sizing. It doesn't make a huge difference, just a slightly better practice to do the box-sizing border-box like this.

## How CSS renders a Website? (The Visual Formatting Model)

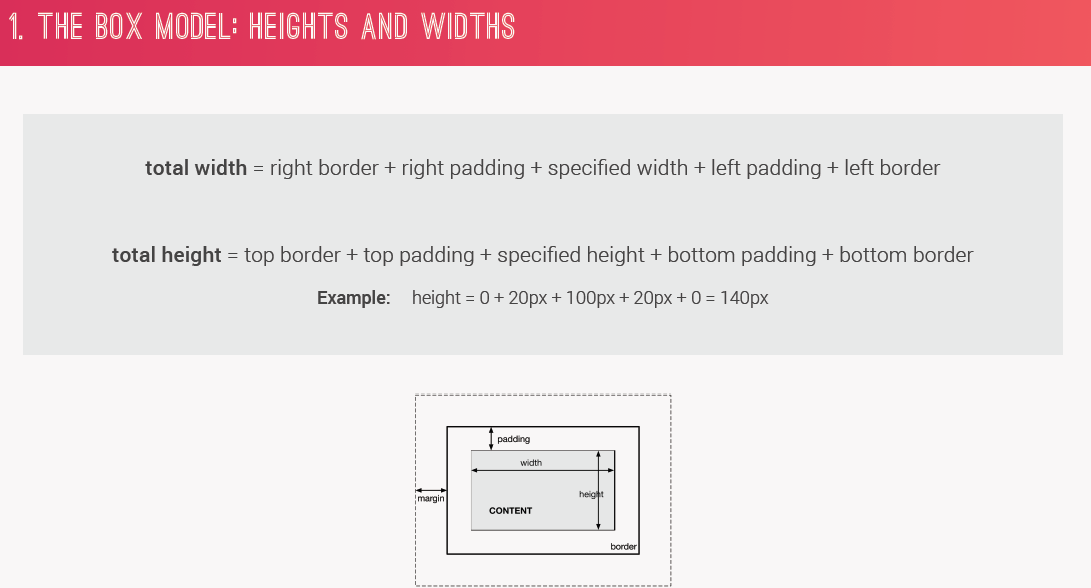
(Refer Flow: [What Happens To CSS When We Load Up A Webpage?)](#_What_Happens_To)

* The Visual Formatting Model – It’s an Algorithm that calculates boxes and determines the layout of theses boxes, for each element in the render tree, in order to determine the final layout of the page.
* This model is one of the fundamental concepts of CSS.
* In order to do this, the algorithm takes into account factors like
  + dimensions of the boxes, which are calculated by the box model.
  + The box type, which can be inline, block or inline-block.
  + The positioning scheme which includes concepts like floats and absolute or relative positioning.
  + Stacking context
  + The other elements that are present in the render tree such as siblings or parents.
  + external information such as the current viewpoint size, dimensions of images, or other factors.
* By putting all these factors together, the browser figures out how the final website will look for the user.
* Let’s look into those one by one.

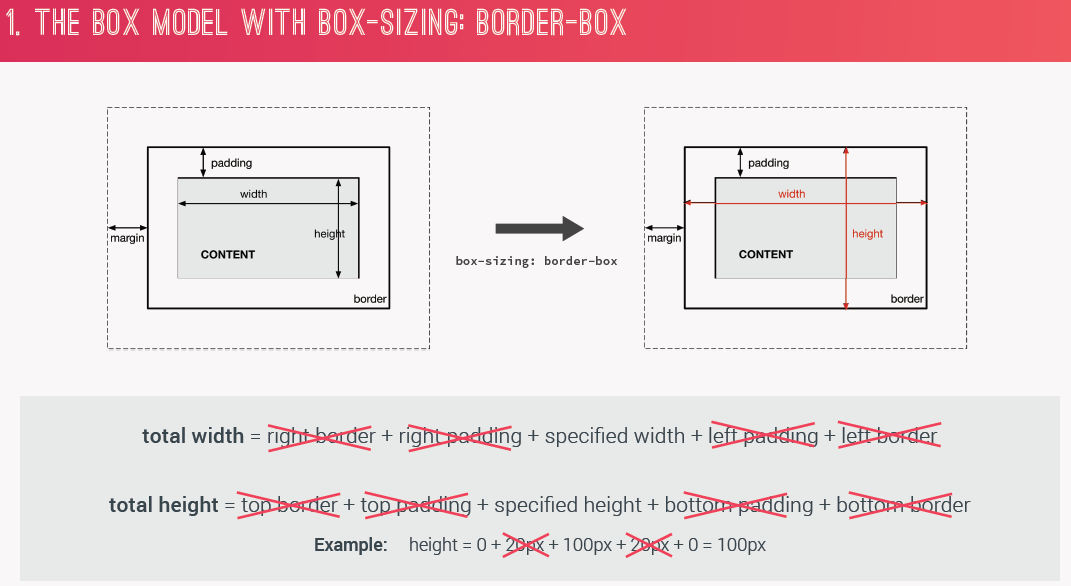
### The Box Model



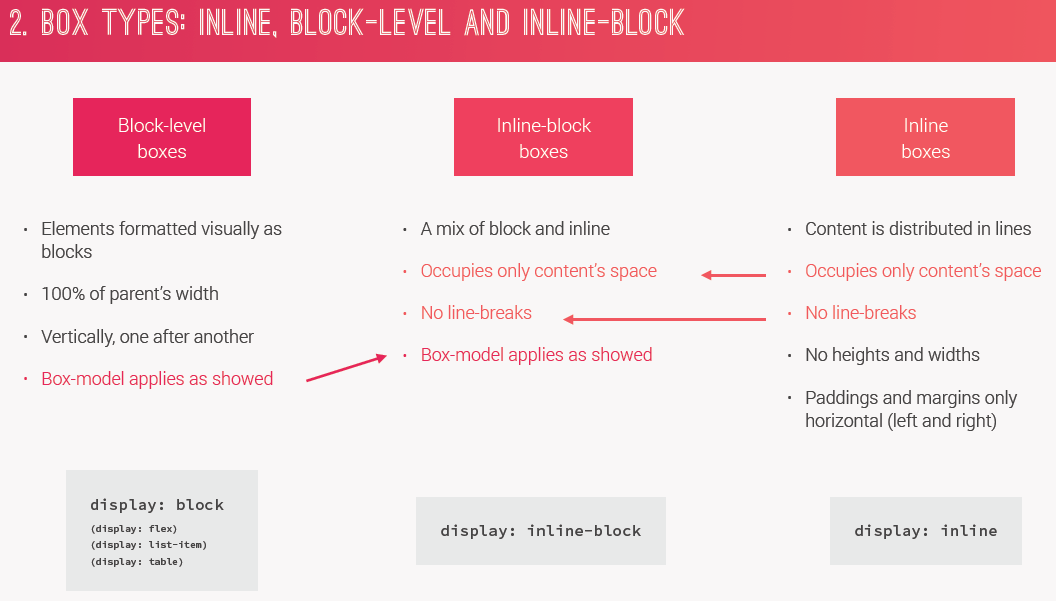
#### Box Model: total Height & Width (box-sizing: content-box)



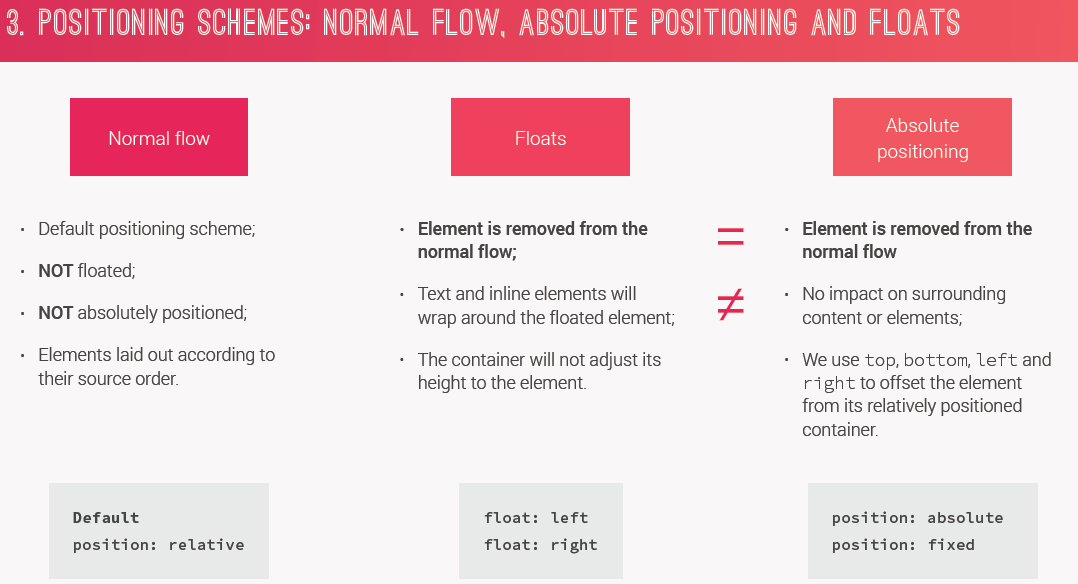
#### Box Model with box-sizing: border-box



### Box Types: inline, block and inline-block

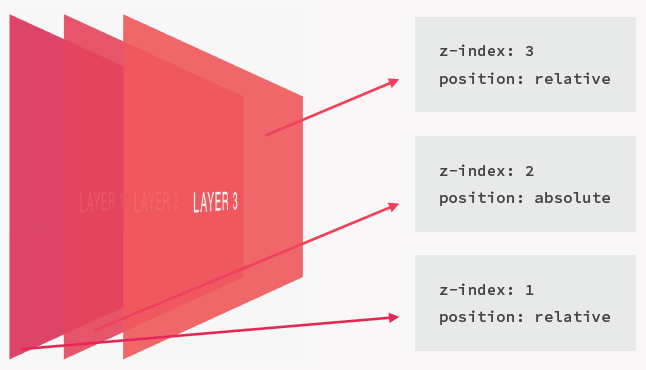


### Positioning Schemes: normal flow (default and relative), absolute positioning (absolute and fixed) and floats (float: left, float: right)



* Floats –
  + The float property causes an element to be completely taken out of the normal flow and shifted to the left or right as far as possible, until it touches the edge of its containing box, or another floated element.
  + When this happens, text and inline elements will wrap around the floated element.
  + Also, when an element is floated, its container will not adjust its height to the element, which sometimes can be problematic. The usual solution to this is to use clear fixes.
* Since an absolutely positioned element has no impact on surrounding content or elements, it can overlap other elements occupying the same space. But CSS solves it for us using something called stacking context.

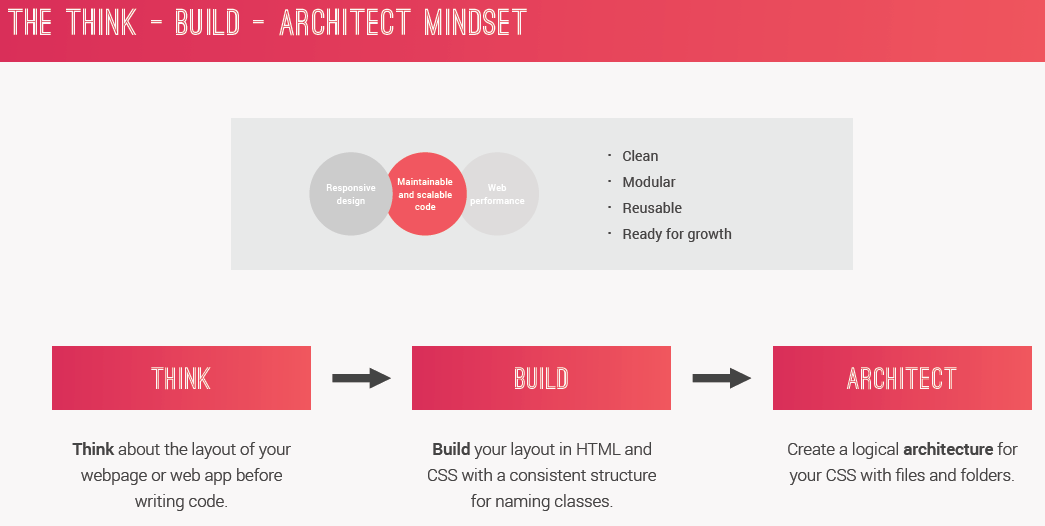
### Stacking Contexts



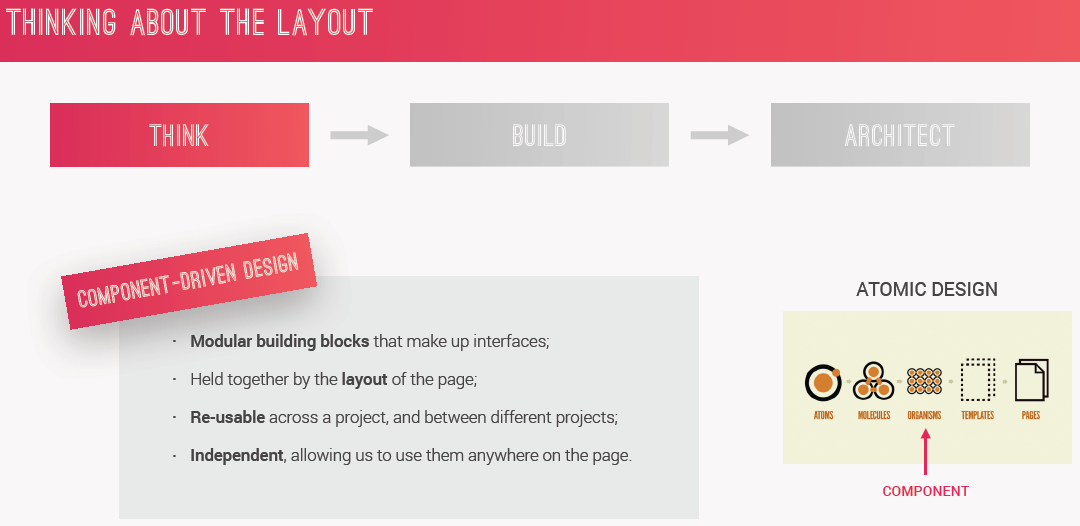
* Stacking contexts determine in which order elements are rendered on the page.
* A new stacking context can be created by a different CSS properties, where the most widely known is z-index. But there are other properties that also create new stacking contexts.
* Stacking contexts are like layers that form a stack. Layers on the bottom of the stack appear at first, and elements higher up the stack appear on top, overlapping the elements below them.
* Between these elements, the one with the higher z index appears on the top, and the one with the lowest z index appears at the bottom.
* Properties which can create stacking context are z-index, opacity, transform, filter, etc.

## CSS Architecture, Components and BEM

* This is about CSS architecture, how we should think about layouts, and how we should mark up our code in a professional way.
* Remember that we want code that is clean, modular, reusable, that makes it easy to add more and more features to our webpage or app if it needs to grow.
* Because of all of these reasons, we cannot just dive into a new project and leave architecture behind just as a second thought.
* We need to take important decisions concerning our HTML and CSS code right from the beginning of the project.
* We need a good strategy – a good mindset.



### Think

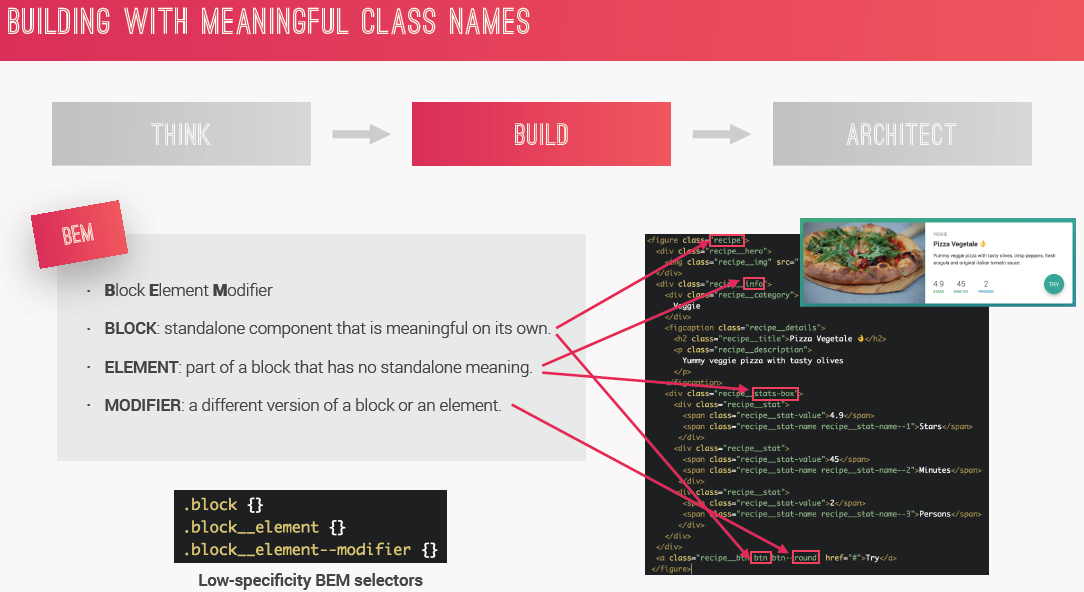


* With component-driven design, we try to divide our page into modular components.
* Components are the building blocks that we use to construct our interfaces.
* So we can basically think of our interface as a collection of components held together by the overall layout of the page.
* Components should be reusable across a project and also between different projects.

So you can build a library of your components and then reuse them across projects, which will speed up your development.

* Components should be independent so that we can use them completely on their own no matter where on the page. What this means is that components should not depend on their parent elements.
* All of this will make your CSS code easier to maintain and to scale if necessary.
* According to atomic design, the smallest units on a page are atoms, which together form molecules, which combine together to form organisms. And these organisms can be seen as our components in some situations.

### Build

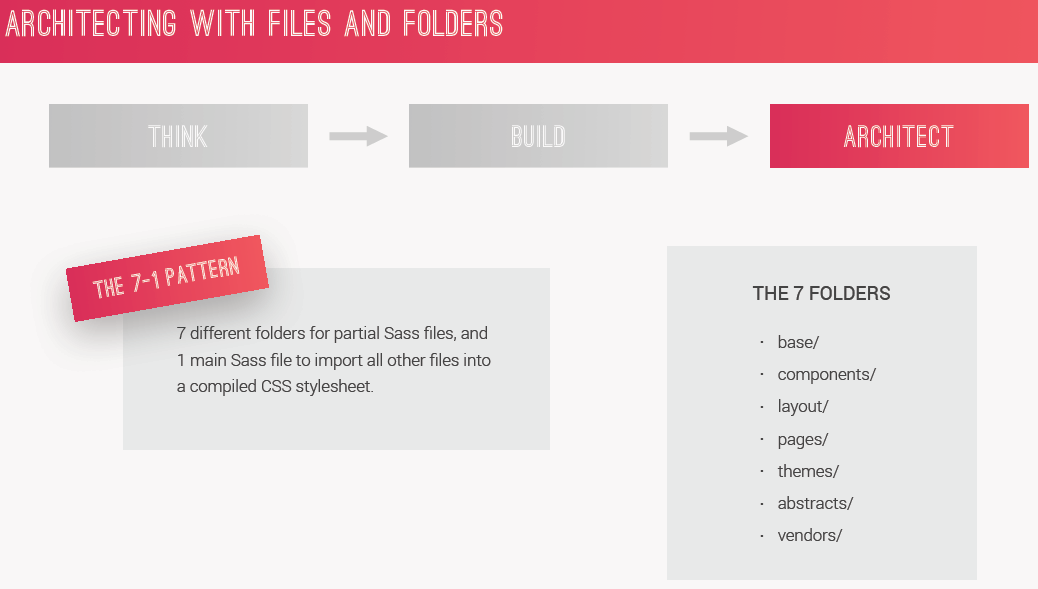


* After thinking about the design, we need to code the design using HTML and CSS, and in this step, it's important to have a consistent strategy and structure for naming our classes.
* Approaches for naming our classes – Object-oriented CSS or S Max, BEM.
* BEM, which is becoming more and more popular by developers around the world.

#### BEM (Block Element Modifier)

* It's a nice clean system for marking up our layouts.
* In BEM, a **Block** is a standalone component that is meaningful on its own, and can be reused anywhere in the project.
* We can have nested blocks and that is perfectly normal and acceptable.
* An **Element** is a part of a block and has **no meaning on its own.** i.e. if we take one of these elements out of the block then they wouldn't be useful at all.
* The **Modifier** is a flag that we can put on a block or an element in order to make it different from the regular blocks or elements. To make a different version, let's say.
  + For instance, there may be some rules for all buttons, and then we use a modifier to make a more specific, different button. E.g. big button, small button, etc.
* BEM creates selectors with really low specificities because we always use classes and they are never nested. And so they always have a pretty low specificity, and this is one of the big reasons why BEM is so widely used for easy to maintain and reusable code.
* If we follow BEM, then just by reading the markup with those class name, we can instantly see how all elements are related and what each of them does.

### Architect



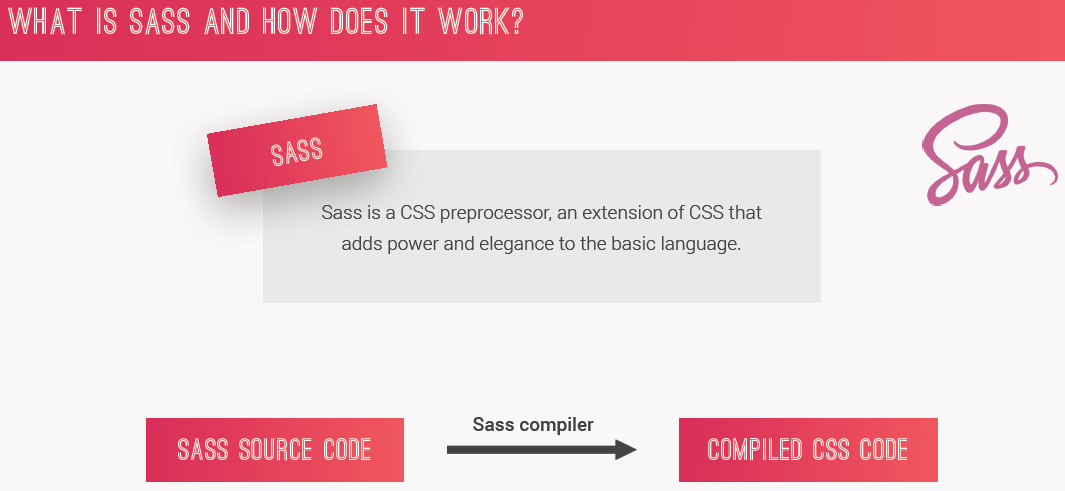
* Architecting CSS means to create a logical folder and file structure for our CSS to live in.
* There are different options like the ITCSS method or S Max, the 7-1 Pattern.

#### The 7-1 Pattern

* The 7-1 Pattern is extremely simple. All it means is that we have seven different folders where we put partial Sass (or any other CSS Preprocessor) files, and then one main Sass file in which we import all of our partial files into one final compiled CSS stylesheet.
* The 7 folders are –
  + **base** folder where we put the basic product definitions.
  + **components** folder where we have one file for each component. Components are our building blocks which are independent and reusable everywhere across our website.
  + **layout** folder where we define the overall layout of the project e.g. global footer, header, etc. and holds all of the components together.
  + **pages** folder where we have styles for specific pages of the project.
  + **themes** folder if you want to implement different visual themes.
  + **abstracts** folder where we put code that doesn't output any CSS, such as variables or mix-ins
  + **vendors** folder where put all third party CSS.
* We don't always need to use all of these folders. It's gonna depend on the size and the scope of the project.

# Introduction to Saas and NPM

## What is Saas?

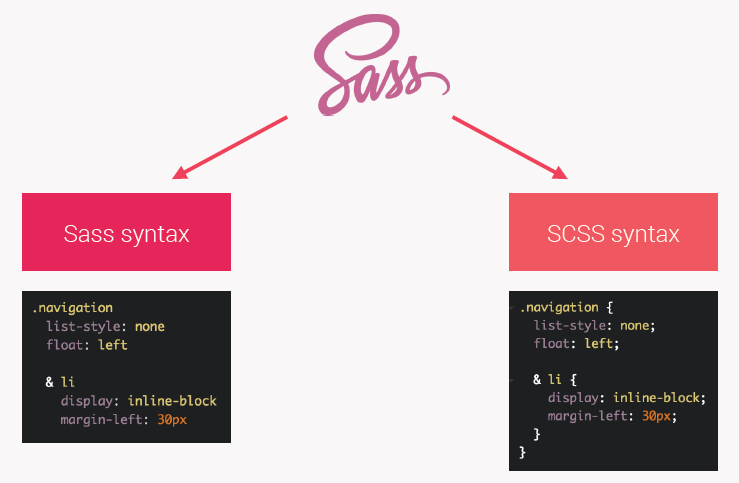


* **Sass** is a CSS preprocessor, an extension of CSS that adds a lot of power and elegance to the basic language.
* So basically, we use Sass to fix the problems that we have with CSS.
* CSS gets very messy, very quickly.
* For each project, having a single CSS file with thousands of lines of code without any reusable pieces, without any logic, gets completely unmanageable after some time.
* That's why we use Sass which provide us with a couple of handy features and tools that CSS simply doesn't have, while at the same time, not changing the fundamental way that CSS works.
* The way Sass works is like this –
  + Instead of writing a CSS file with regular CSS code, we write Sass code in Sass files.
  + Then, we run a compiler, and that compiler converts the Sass code we wrote into regular CSS code as if all we ever did was write in regular CSS in the first place.
* So, we need to process our Sass code first, and that's why it's called a CSS preprocessor.
* Our website has no idea that the code we wrote in the beginning was in Sass because it only ever sees the final output of the compiler in the form of CSS.
* There are actually other CSS preprocessors like **Less** or **Stylus**, but **Sass** is the most popular one.
* Note: If some code in Saas does not reflex on webpage, check the generated .css code. You might get some clue.

### Features of Saas –

* **Variables** – which allow us to have reusable values such as colors, font sizes, spacing, etc.
* **Nesting** – in order to nest selectors inside of one another, allowing us to write less code.
* **Operators** – for mathematical operations right inside of CSS.
* **Partials and Imports** – which are one of the most important and most useful features of Sass, allowing us to write CSS in different files, and then importing them all into one single file.
* **Mixins** – to write reusable pieces of CSS code. The code is copied as it is where it is used.
* **Functions** – which are quite similar to mixins with the difference that they produce a value that can be used later.
* **Extends** –
  + **Extends** can be used to make different selectors inherit declarations that are common to all of them.
  + We should **only** ever use extends if the rules that we're extending are inherently and thematically **related**. E.g. buttons of different types, etc. Otherwise it may cause maintenance problem.
  + Use extends when the selectors or the elements that you are extending are actually pretty related to one another.
  + CSS code written inside Extends is not copied to each and every element where is it used (but the selectors where is it used themselves get copied in place of Extends declaration).
  + There is this subtle difference between Mixins and Extends. (Use codepen to see the difference in final compiled CSS.)
* **Control directives** – allow developers to write complex code using conditionals and loops like in real programming language. It's nearly not needed in a real world project. It's more for writing CSS frameworks or something.
* SASS comes in with a couple of built-in color function which can make life our life easier. Instead of having to calculate another color, we can simply use one of the built-in color functions. E.g. darken(), lighten(), etc.
* The function, mixin are just like the variables, they will not produce any final CSS code, because their only purpose is to help us writing our code.
* Basic introduction to how to use variables, functions, mixins, extends –
  + Sass Introduction: Variables, Nesting:   
    <https://codepen.io/sameer-bhilare/pen/ExKKYLP>
  + Sass Introduction: Mixins, Functions, Extends:   
    <https://codepen.io/sameer-bhilare/pen/NWNNPKr>
* A valid CSS is also a valid SCSS as the syntax is quite same.

### Sass syntaxes



* There are two Sass syntaxes as you can see above.
* One syntax, which is actually the original one, is just called Sass.
  + The Sass syntax is indentation sensitive and doesn't use any curly braces and semicolons.
  + this syntax a bit confusing, and more difficult to learn, and also more difficult to convert original CSS projects.
* The other one is called SCSS which stands for Sassy CSS. And the SCSS syntax preserves the way that original CSS looks like. (More preferred way)
* Whichever syntax you choose, it will work the same way in the end.

## NPM

* NMP package ecosystem is where we can find all kinds of open source tools and libraries and frame works needed for modern web development.
* Node JS is an open source JavaScript run time that allows developers to write and run JavaScript application on the server.
* Developers started using Node to also write tools like for test automations, or to compile SASS to help them with local web development.
* In order to use and share these packages we need some kind of tool to install and to manage them.

That is where the NPM comes in.

* NPM is a simple command line interface that allows developers to install and manage packages on their local computers. Plus NMP also allows us to do some automations of boring tasks.
* There are all kinds of open source tools, libraries and frameworks needed for modern development.
* Modern web development could simply not exist without a package manager.

### Installing Saas

* Before installing our first package, usually the first thing that we always do in a new project is to create a package.json file which basically will contain the definitions of our project and where NMP will write the packages that we use.
* To create package.json,
  + npm init
* Now let’s install Sass
  + npm install node-sass --save-dev
  + --save-dev because we want this to be dev dependency only as it is just development tool and will not be part of final deployable. And this will install the package **locally**, that is only for that project in which is it installed.
* The important reason we should use --save-dev or --save option in the package.json file is that by this way we can then share our project without sharing the entire node\_modules folder. And so if someone else receives our project, they can just do npm install and then all of the dependencies will be automatically installed.
* Why do we have to worry about writing dependencies and dev dependencies in package.json?
  + That’s because we can then share the project along with package.json file and without the node\_modules folder to others and others can exactly recreate the project with exactly same dependencies and version on their local by just running npm install command.
  + npm install will take a look at our package.json file and download exactly the packages that we specified there.
  + So this package.json file is really useful in each project where we have NMP modules.
* To uninstall a package from our project, use npm uninstall <package-name> -- save
  + E.g. say we want to uninstall jquery from a project  
    npm uninstall jquery --save  
    This will uninstall jquery and because of the --save flag will the dependency will be removed from package.json file.

### Running Sass package

* Since we have installed node-sass, in order to actually run it against our project, we need to write an NPM script for that.
* We need to write that script in the package.json under key “scripts”.
* E.g.

"scripts": {

    "compile:sass": "node-sass sass/main.scss css/style.css"

 }

* And you can now run this script like this – npm run compile:sass
* This will compile your sass input file from given source folder and will create css file at given destination folder.
* This is only one of the many possible scripts that we could write in the package.json.
* There's so many packages on NPM. There are packages to automatically prefix our code in order to support all of the web browsers. There are packages to compress our code, and so much more.
* Now every time we make changes to sass file, we need to compile it to css file. That is not really practical. But luckily we have a solution for this. The node-sass package allows us to to keep watching our main "sass" file by just adding –w option to our node-sass command.

"scripts": {

    "compile:sass": "node-sass sass/main.scss css/style.css –w"

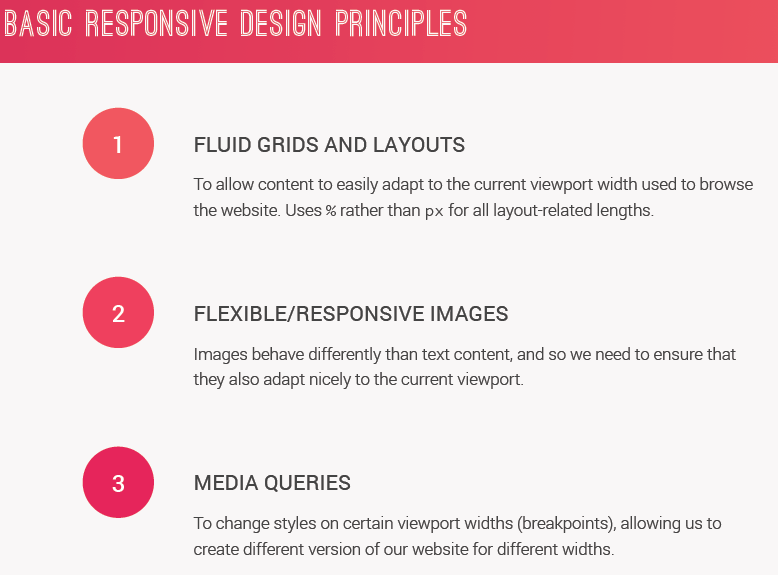
}

* Once you make this change, run the command again and then start making changes in your sass file, it will automatically create css file from it and your changes will be automatically reflected. npm run compile:sass
* So now each time that we write some more SASS code, it automatically compiles and we can then check out our page directly. The only thing that you have to keep in mind is that the the npm run command above should be running in some command window or terminal.

### Automatically Reload your Project on any file changes

* We just need to install an npm package that automatically reloads on a complete project as soon as we change any file in the project folder.
* The npm package to do this is **live-server**.
* We want to install this package globally, because we want to be able to use it, not only in a particular project, but also on all future projects and anywhere on our computer.
* So the command to do so is   
  npm install live-server -g
* -g means globally. And you will see that this package will neither be in the dependencies of package.json nor in node\_modules folder. That’s because we installed this one as a global package.
* Since it's a global package, we can actually call it directly from the command line. And we don’t need to write any script in the package.json file.
* Command to the this live-server package –   
  live-server
* Once you run this command, you will see your webpage will open in your browser at <http://127.0.0.1:8080/> (BTW we can change the port number in configuration.)
* Now all we need to do was to change the code and come back to the browser and everything updated for us very nicely and we didn't have to click the reload button.

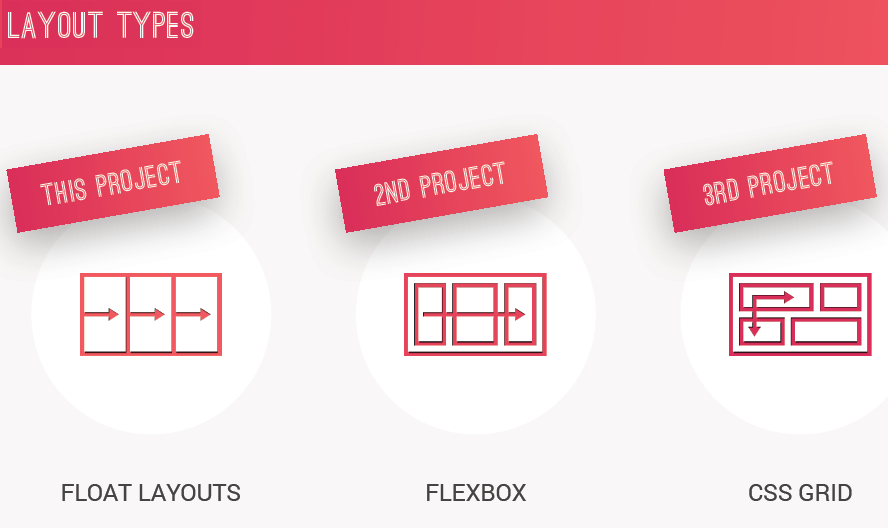
# Three main ingredients to responsive web design.



1. Fluid Grid –   
   In a fluid grid all layout elements are sized in relative units like percentages instead of absolute units like pixels.
2. Flexible Images –   
   Are also sized in relative units.
3. Media Queries –

Allow us to specify different CSS style rules for different browser widths.  
With media queries we can target different devices like mobile phones and tablets and make the website respond to their screen size.

## Layout Types



* In modern web design there are currently three major ways of laying out a webpage or app.
* Float Layouts –
  + We simply put a bunch of boxes side by side, using floats.
  + This method is still being used because the alternatives are still not 100% supported by all browsers.
* Flexbox –
  + Offers an amazing way of laying out elements in a one dimensional row.
* CSS Grid –
  + Perfect for creating the overall layout of a page in a fully-fledged 2D grid.
* The layout is the single most fundamental aspect of our design, and so if a browser does not support a technology like Flexbox or CSS grid, then that's a major problem and can ruin our entire designs.
* Moving forward in the future, we will always have to know how to use all of these three layout types.

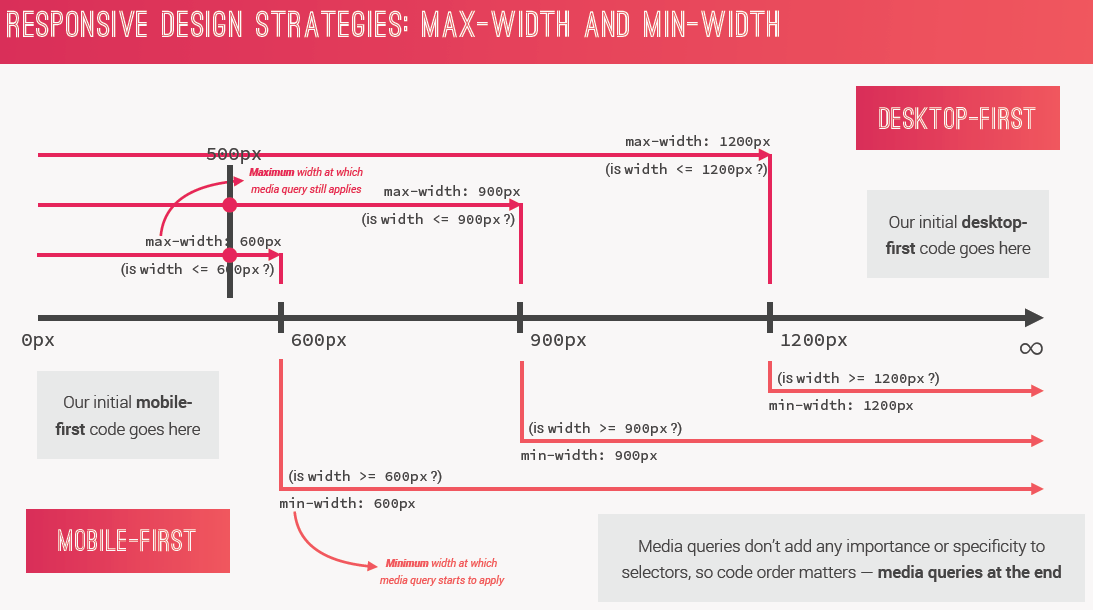
# Advanced Responsive Design

* Two fundamental aspects of modern responsive design are –
  + Deciding about doing mobile-first or desktop-first for website or app and
  + Selecting breakpoints for our project, when writing media queries.

## Desktop-First Vs Mobile-First



* **In the Desktop-First approach,** 
  + We optimize our interfaces for the large screens. And therefore, we start by writing CSS code for these large screens, into our CSS file.
  + Then later, when we make the website responsive, we simply write media queries, in order to shrink the design to fit the smaller screens (writing media queries that test for **max-width** -The maximum width till which the media query is applicable).
  + This is the more traditional way of doing things and also the easier way to learn.
* **In Mobile-First approach,**
  + We start writing CSS code for smaller screens, in order to optimize for mobile.
  + And then, move up to larger screens from there using media queries (writing media queries that test for **min-width** - The minimum width at which the media query starts to apply).
  + The main strategy behind Mobile-First is we really have to think about the mobile experience, by reducing our websites or apps to the absolute essentials. Stripping away everything that's not really necessary, in order to end up with a smaller and faster final product.
  + We have to use min-width type of media query in the mobile-first approach, because basically we want the queries to stay away from our smallest screen styles. So, we want them to work for bigger screens, and so the max-width approach wouldn't work.



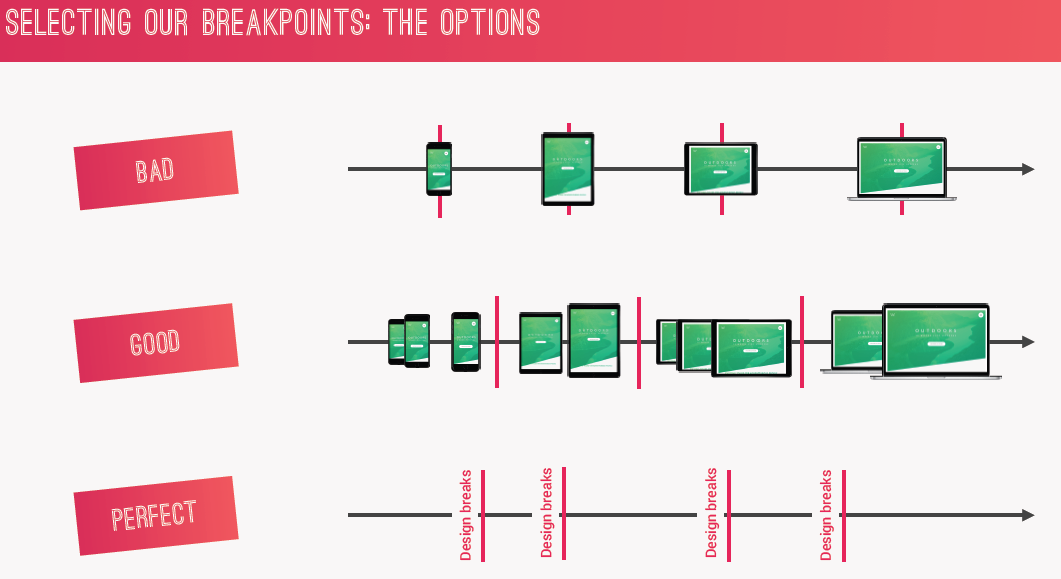
* Think of media queries as tools, to overriding specific parts of our CSS for specific viewport width.
* If you have conflicting CSS rules in media queries, which is usually the case, like defining the global font size in both media queries, then the one which appears last in the code is the one that takes precedence. That's because media queries don't add any importance or specificity to our selectors. And so, the order of the code matters a lot. And, that's why we **keep our media queries always at the end.**

## What’s best for us?



* No matter what you decide, always keep both the desktop and mobile versions in mind and understand that they are both important. You should never design only for one of the solutions and just leave the other one as an afterthought.

## Selecting Our Breakpoints



* Breakpoints are the viewport width at which we want our design to change.   
  In other words, where we want to put our media queries.
* Usually, we'll need one breakpoint for phones, one for portrait tablets, one for landscape tablets and one for the desktop. There can be more, of course, but these are the basics.
  + The phone ranging from 0-600 pixels,
  + portrait tablets from 600-900
  + The landscape tablet from 900-1200
  + Desktops from 1200+
  + Big desktops starting at 1800 pixels.
* There are three major ways of selecting breakpoints – Bad, Good and Perfect.

### Bad Way

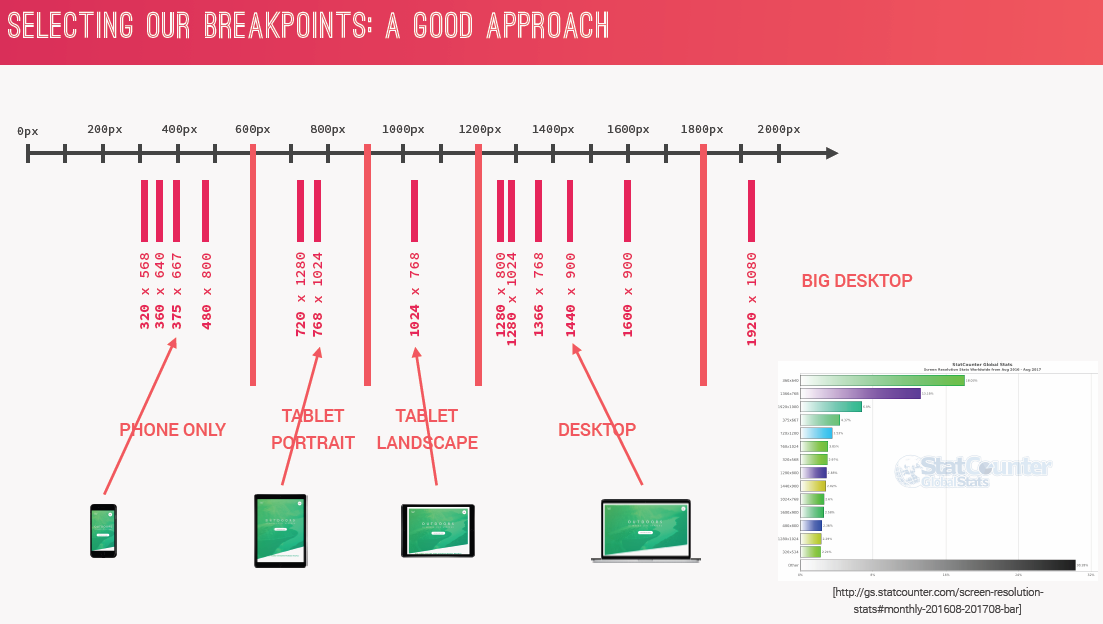
* Unfortunately, the bad way is also the most used way of choosing breakpoints.
* It consists of simply using the width of popular devices as the breakpoints like the iPhone and the iPad, to set breakpoints for their screen width.
* While this is the easiest way, it has a couple of **problems**.
  + First, you're optimizing for one very specific device, ignoring all the users of other devices, which may be even more used than the Apple products.
  + And, second and even more important is that this strategy is not future proof at all.   
    E.g. If tomorrow Apple decides to completely change resolutions on all their devices, we would need to change all of our media queries.
  + So using specific devices as breakpoints is completely against the logic of reusable and maintainable code.

### Good Way

* In the good way, we look at all the most-used device width on the entire internet, try to group them together in a logical way and then pick our breakpoints from that.
* It's a lot better than the first way because we're using a lot of devices, and we're also using the most popular device width. Plus, we're not setting breakpoints at one specific point but between similar device widths.

### Perfect Way

* The perfect way to do this is to ignore devices all together and only look at your content and your design.
* Ideally, it works like this – You begin at one size, either mobile or desktop and then start increasing your screen width or decreasing for desktop first. Then, as soon as the design breaks, which means that the design no longer works or it no longer looks okay, then you insert a new breakpoint.
* So you just put the breakpoints wherever your design starts to look weird and out of place, and don't think about devices, at all.
* This approach can be extremely difficult. And, that's why not many people are doing it.
* Without the constraints of at least a couple of pre-defined breakpoints, it's hard to find the best ones and not to end up with really a bunch of them.



## Writing Media Queries

* We should not use pixels in the media queries. That's because if the user changes the default font size in the browser, then the media queries would not be affected by that. And we won't get desired results.
* Important note about rems and ems in media queries –
  + ems and rems in media queries are NOT affected by a root font size setting.
  + What that means is that 1rem or 1em in a media query is always equal to the font size which is coming from the browser.
  + And that's by default 16 pixels, but if the user changes it to 20 pixels, then it will be 20 pixels no matter what we have in our base/root font definition.
* We should use ems instead of rems in media queries, because rems fail to work as intended in some of the browsers. There's been some experiments done by people who understand a lot of all of this, and the conclusion basically is that **ems are the BEST option for media queries**.
* So the ems will be converted to pixels and will be adopted to the font size that the user set for the browser. So, it's a more accessible and dynamic solution here instead of using the pixels.
* First and foremost, Responsive web design does not work without having below meta element in the HTML head.

<meta name="viewport" content="width=device-width, initial-scale=1.0">

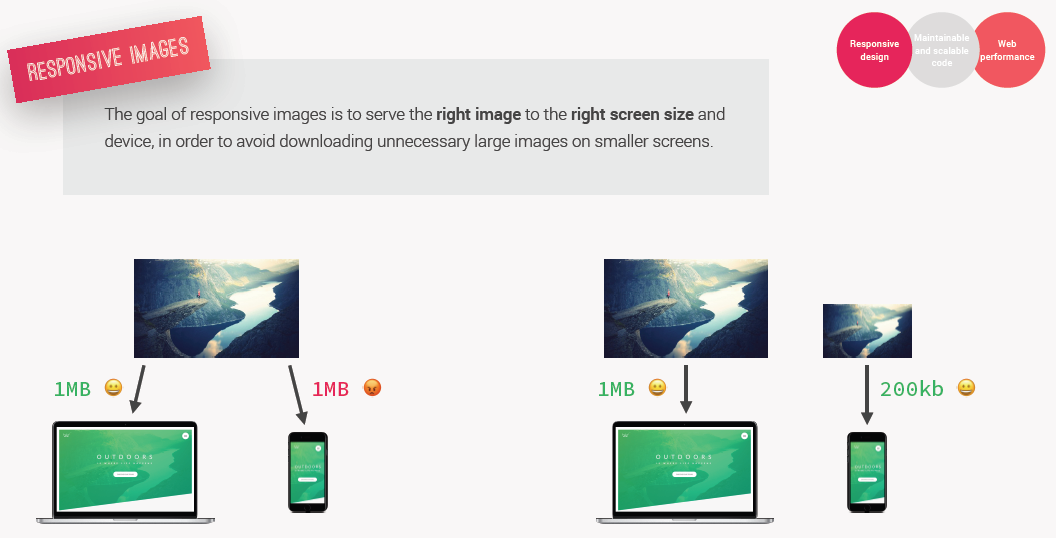
width=device-width, basically says that our website should be rendered with the width of the device. But if we do not specify this here in the HTML, then the browser will basically zoom out our page, so that the largest possible version of our page fits the screen.

* While writing media queries, sequence does matter that’s because one media query can be applicable to more than one breakpoints e.g. in cases when we are only using max-width, and NOT both max-width and min-width while writing one media query.   
  @media (max-width: 800px) { … } // only using max-width  
  @media (min-width: 600px) and (max-width: 800px) { … } // both min & max
* Always write the larger (meant for bigger sizes) media queries before the smaller (meant for smaller screen sizes) ones.
* Always use 'only screen' in the media query. It’s a best practice to put this. This actually means is that this media query should only apply to screens. So if someone tries to print out our page, then these media queries don't apply.   
  e.g. @media only screen and (max-width: 37.5em) { . . . };
* We cannot simply identify a touch device by the width of the screen alone. But, good news is that in CSS we can actually write a media query which can identify if the user **can hover** over elements, or not, and with that, we can very easily identify touch devices and non-touch devices (devices with a mouse).
  + only screen and (hover: none) =>

It applies for device where we cannot hover, which basically means, whenever we have a device which uses touch as the primary input.

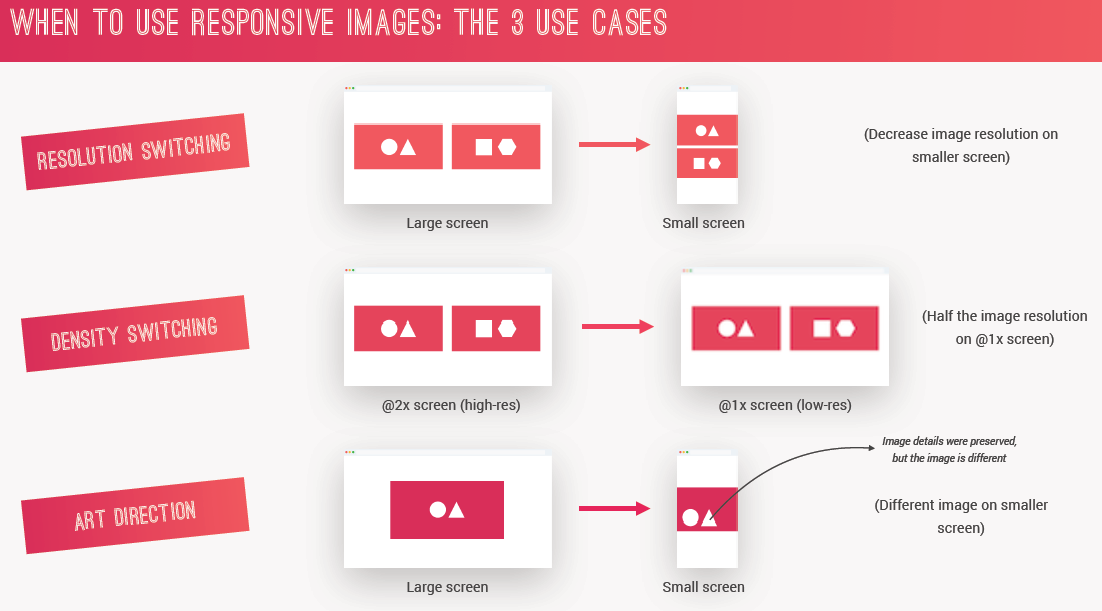
* + Opposite of (hover:none) is (hover:hover).
  + E.g. @media only screen and (hover: none) { . . .}

## Responsive Images



* Responsive images take things to a whole new level. That's because responsive images are not only an aspect of responsive design but even more importantly of web performance.
* The philosophy behind responsive images is to **serve the right image to the right screen size** **and device** using different techniques in HTML or CSS so that users don't have to download images that are way too large for their devices.
* It makes way more sense to send the 1MB image to a device with a large screen that actually needs the image and a much smaller version of the image to a device with a small screen that doesn't need such a large image anyway.

### Responsive Images in HTML



* There are mainly three use cases where it makes sense to use Responsive Images (in HTML).
* In conclusion, Resolution and Density Switching is when you want to serve the same image but with different resolutions. And Art Directions is when you want to serve a completely different image.

#### 1. Resolution Switching

* All that we do in resolution switching is to serve up the same image for a smaller screen but with a smaller resolution.
* So basically, the same image but a smaller version for a device that doesn't need such a big image.

#### 2. Density Switching

* It’s a special case of resolution switching but where the screen size does not matter but a screen pixel density does instead.
* **Pixel density** is the amount of pixels found in an inch or a centimeter.
* There are low resolution screens, and high resolution screens.
* Low Resolution Screens
  + Low resolution screens are just typical PC screens, and they can be called 1x Screen because they use one pixel to display one pixel of our design.
  + So if we say an image should be 100 pixels high, they will actually use 100 physical pixels in the screen to display these 100 pixels that we specified.
* High Resolution Screens
  + Now high-resolution screens are the ones found in all modern smart phones and even some computers already have them, like the MacBooks with retina displays.
  + These are high-resolution screens and can be called 2x screens because they actually use two physical pixels to display one pixel of our design.
  + So if we say we want an image to be 100 pixels high it will actually be 200 pixels in a physical screen. And this all happens automatically of course.
  + Phones usually have a pixel density of two, that is 2x screens.
* What does all of that mean for our images?
  + It means that if we want our images to look sharp on high-resolution displays which use double the amount of pixels then we actually have to serve an image with double the resolution of the original image.
  + And that's what density switching is for, to serve one image to a high-resolution screen and another one if the screen is low resolution.
* Example
  + <img srcset="img/logo-green-1x.png 1x, img/logo-green-2x.png 2x">
  + srcset attribute allows the browser to choose the best of these two images according to the screen that is used by the user to display the webpage.
  + Syntax for srcset => "<image\_path> <density\_descriptor>, ..." e.g. img/logo-green-1x.png 1x So if it's a low density screen, the browser will use the image with 1x density descriptor and when a device has a high resolution screen, then it will use the image with 2x density descriptor as described in the srcset attribute.

#### 3. Art Direction

* Art direction happens when you don't just want to serve the same image but in smaller resolution, but a whole different image for a different screen size.
* For example, you may want to keep the important details in an image but remove parts of the image around these details, or you could also want to load a completely different image altogether.
* Example

<picture class="footer\_\_logo">

    <source srcset="img/logo-green-small-1x.png 1x,

                    img/logo-green-small-2x.png 2x,"

            media="(max-width: 37.5em)">  
 <!-- followed by other source elements for different media queries or img element -->

</picture>

#### Example of Art Direction & Density Switching

* Art Direction is telling the browser to use one image on one screen width and another image on another screen width. And the solution for that is to use an HTML element - <picture>
* We can specify multiple sources for one image and in the source element we can write a media query, so with this we force the browser to use the image from source element’s source set in case that max width is less than 600 pixels, and in case it's larger than 600 pixels, then the browser is forced to use the image from img element's source set.   
  This is called art direction - Different images for different viewport width.

<source srcset="img/logo-green-small-1x.png 1x,

                img/logo-green-small-2x.png 2x,"   
        media="(max-width: 37.5em)">

* Now inside each of these sources, we then do Density Switching and that's why we're using the source set (srcset) attribute instead of the source (src) attribute. This means, serve a larger version of the same image for high resolution screens and serve a smaller version of the same image for a low density screen.
* srcset attribute allows the browser to choose the best of these two images according to the screen that is used by the user to display the webpage.  
  Syntax for srcset => "<image\_path> <density\_descriptor>, ..."   
   e.g. img/logo-green-1x.png 1x
* So if it's a low density screen, the browser will use the image with 1x density descriptor and when a device has a high resolution screen, then it will use the image with 2x density descriptor as described in the srcset attribute.

<img srcset="img/logo-green-1x.png 1x,   
 img/logo-green-2x.png 2x"   
 alt="Full Logo">

* We could now of course go ahead and add another source element with another media query here and then add more and more different versions of the image.
* Final code

<picture class="footer\_\_logo">

    <source srcset="img/logo-green-small-1x.png 1x,

                    img/logo-green-small-2x.png 2x,"

            media="(max-width: 37.5em)">

    <img srcset="img/logo-green-1x.png 1x, img/logo-green-2x.png 2x"

            alt="Full Logo"

            src="img/logo-green-2x.png">

</picture>

* Note: It is important to also provide src attribute along with srcset, in case a user is using older browser which does not support srcset.

#### Example of Resolution Switching + Density Switching

* For Resolution Switching, we will be again using the srcset attribute, but along with that we will be using the width descriptor.
* Using width descriptors, we basically inform the browser of the width of each of the images without the browser having to download them to get access to that information.
* Syntax: <img srcset="<image\_path> <width\_descriptor>, ..."  
  E.g. <img srcset="img/nat-1.jpg 300w, img/nat-1-large.jpg 1000w">  
   nat-1.jpg has a width of 300px, and nat-1-large.jpg has 1000px width.
* Width descriptor alone is not enough for the browser to figure out which image to download, because the browser knows only its view port width and its display density.
* Along with Width descriptor, we need to also provide approximate width of the image at different viewport widths, to the browser in order for it to decide which of these two images to choose. We pass that additional information using img element’s the 'sizes' attribute.
* And so, with the ‘sizes’ information, together with the width descriptor, the browser can then figure out which is the perfect image to use for the current viewport width and the current display resolution.
* Basically for 'sizes' we pass a list of breakpoints that we define and approximate width of the image at those breakpoints.  
  E.g. For breakpoint 900px, considering the image width is 171px (use devtools at 900px), hence 171/900 = 0.19 = approx 20%, so 20vw   
   For breakpoint 600px, considering the image width is 171px,  
   hence 171/600 = 0.285 = approx 30%, so 30vw  
   Then default size (desktop version), considering image width is 291px  
   so using 300px (approx). This is used when none of previous breakpoints apply.
* Final <img> tag with both Resolution & Density Switching.

<img srcset="img/nat-1.jpg 300w, img/nat-1-large.jpg 1000w"

     sizes="(max-width: 900px) 20vw, (max-width: 600px) 30vw, 300px"

     alt="Photo 1"

     src="img/nat-1-large.jpg">

* This entire img element will take care of actually both resolution switching and density switching since we are giving the browser enough information to figure out exactly what to do depending on both the view port width and the pixel density, so the screen resolution.
* Note: It is important to also provide src attribute along with srcset and sizes, in case a user is using older browser which does not support srcset and sizes.

## Responsive Images in CSS

* Responsive images in CSS are pretty straight forward. Because all we have to do is to write media queries, in order to load different images, for different situations.
* We can not only use Media queries to target just screen or viewport width, but also we can write media queries depending on other things, such as the device resolution.
* Example –
  + Large version of the hero image, is shown when the resolution is higher than 192dpi (dots per inch), so when we have 2x screen, and at the same time, whenever the width is larger than 600 pixels (37.5em). That’s because If the screen is below 600 and if the pixel density is 2, well then the image that we need at this size doesn't have to be larger than 1200 pixels. So 600 times 2 is 1200, and so an image, which has 1200 pixels of width, is more than enough to display on a high density screen like this.
  + Also we want to show high resolution image for really big screens (2000px = 125em) even if they are 1x screens. (OR condition)

@media (min-resolution: 192dpi) and (min-width: 37.5em), //37.5em = 600px

       (min-width: 125em) {

        background-image: linear-gradient(

                        to right bottom,

                        rgba($color-primary-light, 0.8

                        rgba($color-primary-dark, 0.8)),

                      url(../img/hero.jpg);

}

# Browser Support



* You should always start by checking a very handy tool called caniuse.com before using a property in production.
* Even if a browser doesn't support a property that we really want to use on a site, then we can still apply it to the modern browsers and show a different, simpler style on the older browsers. And that's a concept called **Graceful Degradation**.
* So providing a top notch experience for modern browsers and providing an adapted reduced version to browsers that don't understand all of these shiny new features.
* Using @supports, we can check if the browser supports a certain property by writing a feature query. And accordingly we can add necessary code to support both browsers – those which support and those which not.
* Example

// GRACEFUL DEGRADATION using @supports feature query

// asks the browser if it supports a certain property along with a value (it could be any value. e.g. blur(0).

@supports ((-webkit-backdrop-filter: blur(10px)) or (backdrop-filter: blur(10px))) {

/\* backdrop filter - to apply a filter to what's behind the selected element.So here we will apply a backdrop filter to this background here with a blur. \*/

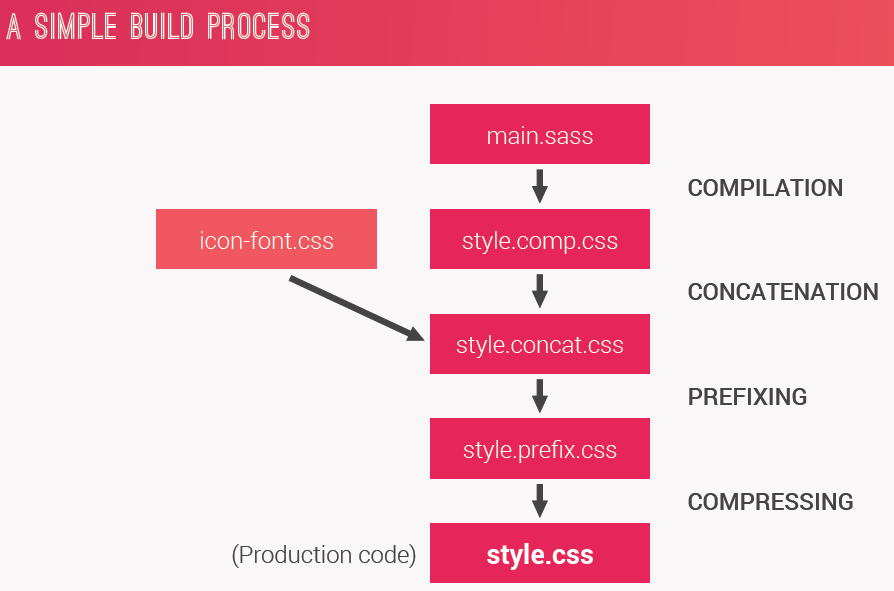
        -webkit-backdrop-filter: blur(10px);

        backdrop-filter: blur(10px);

        background-color: rgba($color-black, .3);

}

# Build Process



* What is build process?
* Well, it's basically just a sequence of tasks that we perform automatically after we finish developing a product or a certain feature of a product.
* And the result of the build process are one or more final files, which are then ready for production. Which means basically, ready to be deployed to a web server.
* Here are the simple steps involved –
  + Compilation – here we compile sass code into CSS code.
  + Concatenation – here we merge the content of one css file with another css file so that we can go from including two css files in our page to only one, which is a lot better, because like this, we only need one HTTP request instead of two.
  + Prefixing – here we **automatically** add prefixes to our code.
  + Compression – here we compress the entire code we have at this stage.

## Production Build Process with NPM Scripts

1. Compilation

* "compile:sass": "node-sass sass/main.scss css/style.comp.css"
* What this means is compile main.scss to style.comp.css

1. Concatenation

* "concat:css": "concat -o css/style.concat.css css/icon-font.css css/style.comp.css"
* For this to work, we need to first install npm package called concat,   
  npm install concat --save-dev

1. Prefixing

* "prefix:css": "postcss --use autoprefixer -b \"last 10 versions\" css/style.concat.css -o css/style.prefix.css"
  + -b means browsers
  + 'last 10 versions' - it'll target the last 10 versions of all the major browsers.  
    And this data of browser versions actually comes from the caniuse.com website. So it's always super up-to-date.
  + -o means output file.
* For this we need to install npm package called autoprefixer   
  npm install autoprefixer --save-dev
* Also in order for the autoprefixer to work, we actually need to install another package which is called postcss. We need this bcz the autoprefixer is actually part of this postcss plugin or package.  
  npm install postcss-cli --save-dev

1. Compressing
   * "compress:css": "node-sass css/style.prefix.css css/style.css --output-style compressed"
   * With the already installed node-sass package, we can compress the final css by just adding a flag --output-style compressed
2. Sequencing above 4 tasks into single task
   * "build:css": "npm-run-all compile:sass concat:css prefix:css compress:css"
   * In order for these steps to work on all platforms, install npm package npm-run-all
   * So the final command for our production ready build will be –   
     npm run build:css

## Development Build Process with NPM Scripts

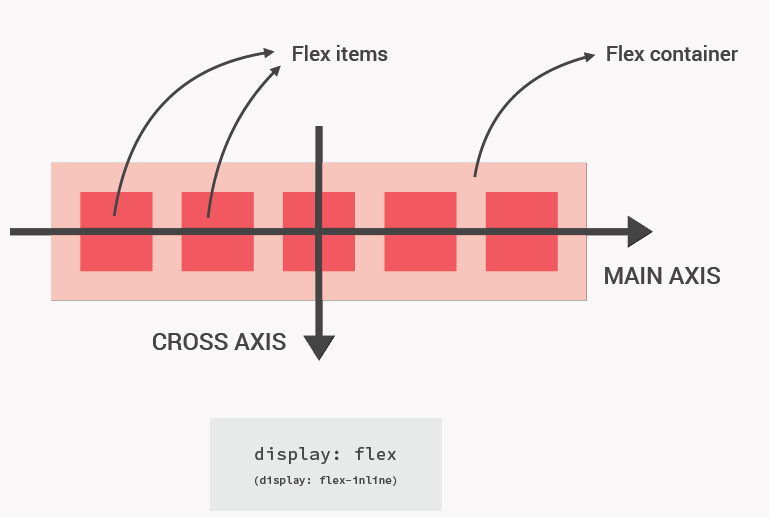
* Similarly we can also create **Development workflow**.
* "start": "npm-run-all --parallel devserver watch:sass"
  + --parallel - because I want both tasks to run at exact same time.
* So final command to be used during development is –   
  npm run start

# Flexbox

## What is Flexbox?

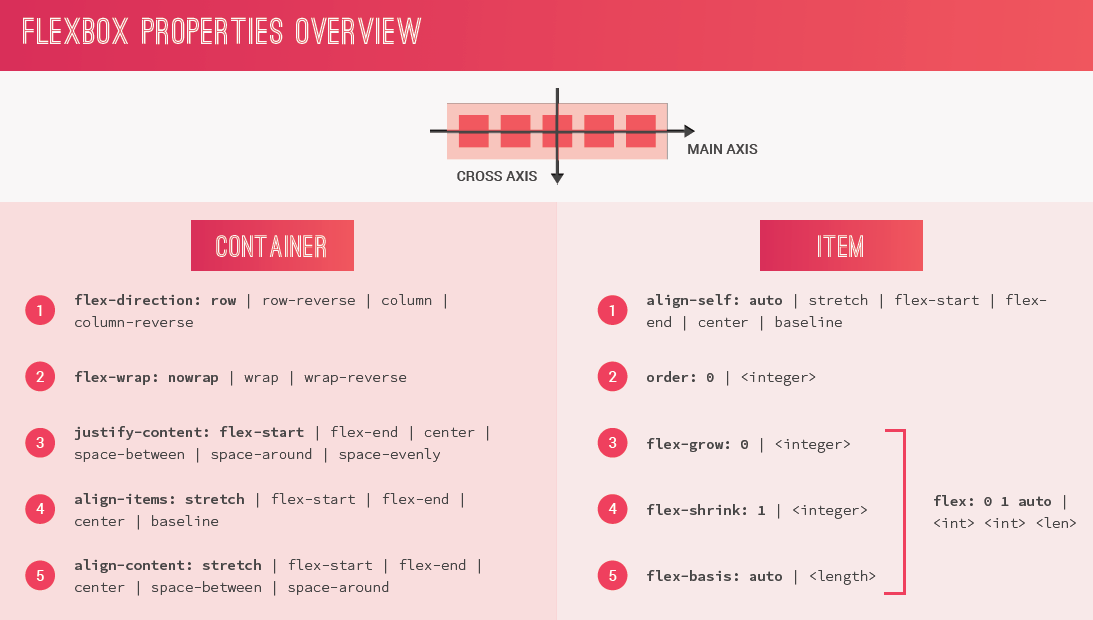
* Flexbox is a new module in CSS3 that makes it very easy to align elements to one another in different directions and orders, and also to change these alignments if necessary, like for example, for responsive design.
* This works perfectly even if you have no idea about the dimensions of these elements because the main idea behind Flexbox is to actually give the container the ability to expand and to shrink elements to best use all the available space.
* With Flexbox, we can easily replace float layouts with much less and much more readable and logical code.
* And because of that, Flexbox completely changes the way that we build layouts, especially one-dimensional layouts because that's where Flexbox really shines.
* For two-dimensional layouts, there are actually better ways of building these, like using CSS grid layouts.
* For all of these reasons, Flexbox is a true revolution in CSS. It makes our lives as CSS developers so much easier.

## Flexbox Concepts



* The element on which we use Flexbox is called the **Flex container**.
* And all we have to do in order to create a Flex container is to set its display property to flex.
* We can also set it to flex-inline, which will create a Flex container that behaves like an in-line element.
* All the direct children of the Flex container are called the **Flex items**.
* The direction which these Flex items are laid out is called the **main axis**, and then the other perpendicular axis is simply called the **cross axis**.
* It is important to be aware of these names because we can change the direction of the main axis, and also because there are different ways of aligning elements both along the main axis and the cross axis. This will help us to fully understand how the different Flexbox related properties work.
* Instead of messing with margins and translates, and floats, and text-align center, and all of that stuff, we can just use Flexbox.
* Flexbox also works with text element. E.g. If we want to center text in the rounded circle.
* The biggest advantage of using flexbox for responsive design is a lot of times we just change the flex direction from row to column for smaller screens.

## Flexbox Properties



* The Flexbox specification defines a couple of properties that we use on the Flex container in order to position and align all the Flex items.
* And then some other properties that we use directly on the Flex items themselves.
* These properties and their allowed values (bold values are the default values) are shown in above picture.

### Flex Container Properties

* flex-direction
  + It specifies in which direction the main axis goes.
  + The direction can go as a row, which is the default direction, but it can also go in the opposite direction and even up or down, which will then form a column.
  + This is really helpful for responsive design. E.g. for desktop we may have flex-direction as row and for mobile, we can then set it to column.
* flex-wrap
  + It simply defines if the Flex items should wrap into a new line if there is not enough space in the Flex container or not.
* justify-content
  + This one defines how the Flex items will be aligned along the main axis. So it's important to know where the main axis is actually.
* align-items
  + It is very similar to justify-content with a difference that this one defines how the Flex items will be aligned along the cross axis, not the main axis.
  + align-items is many times used together with justify-content, and these two are the most important Flexbox properties of all.
* align-content
  + It only applies when there is more than one row of Flex items.
  + In that case, align-content controls how the rows are aligned along the cross axis if there is some empty space.

### Flex Item Properties

* align-self
  + It is very similar to align-items but for one individual Flex item.
  + For example, if we set align-items to center, but then we want one of the items to be aligned to the bottom, then we can use align-self for that.
* order
  + It defines the order in which one specific Flex item should appear inside the container.
  + This is really helpful to reorder items, for example for smaller screens.
* flex-grow
  + It defines how much an item can grow.
* flex-shrink
  + It defines how much it can shrink.
* flex-basis
  + It can define its base width.
* These three properties together, basically helps Flexbox decide on the width of a Flex item.  
  There's actually a shortened property of these three, which is simply called **flex**.

## CSS Custom properties / Variables

* With CSS variables, we don't even need a pre-processor like Sass because CSS can now natively handle CSS variables.
* These CSS variables can be manipulated in JavaScript and we can also edit them in our dev tools.
* Also it's easier to use CSS variables in the calc() function and
* Also CSS variables cascade and they are inherited. So they are very different from SASS variables in that regard.
* These custom properties, they have to be defined inside a scope and so within a declaration block.
* We could put these variables just in any normal selector that we want but then these variables would only be available in that element and its children.
* So it's easier to just have one central place where we collect all the variables in one place.

This one place is usually the root pseudo class (:root).

* Root pseudo class (:root) is essentially like the same thing as the HTML selector but with a higher specificity.
  + By putting all of our variables in in root pseudo class, they will be accessible throughout the entire document, because that's what the root of the document is.
  + It's like the global parent element. And therefore all the variables declared here will be available to all the child elements.
* This is how we declare our CSS Custom property.  
  E.g.

:root {

    --color-primary: #eb2f64; // CSS custom property

}

* To use our CSS Custom property, we use var() function.  
  E.g.

p {

    color: var(--color-primary); // using CSS Custom property

}

* We use double dash (- -) to distinguish between our CSS custom properties and a normal like built-in CSS properties.
* CSS Custom properties/variables of course not only work for colors only but we can store other things as well. E.g. --shadow-dark: 0 2rem 6rem rgba(0,0,0,.3);

## SVG (Scalable Vector Graphic)

* There are several **problems with icon fonts**,
  + Like the fact that it's really just a hack to display icons, which are like images using a font.
  + Also, icon fonts fail more often than you would think, and in that case, the browser just displays a blank square or something like that on the screen.
  + There are actually other issues, like screen readers for blind people trying to read out the icon fonts and of course, failing to do so.
* Because of all those issues with icon fonts, it's actually considered a best practice to move away from icon fonts and to move to a better alternative which is – SVG.
* SVG stands for Scalable Vector Graphics, and it's just basically a way of writing vector graphics with code.
* For SVGs to use, we need a sprite file.
  + A Sprite file is simply an SVG file which contains all of our SVG images in it.
  + This way, we only have to use one global file and from there we can select each of the icons.
  + So this way, we only have one 'http' request for one file instead of having to include all of different (.svg) files.
  + So icon sprites is the way to go, and that's how all the professionals do it.
* Another advantage of SVG is that it makes it really easy to format these icons. With icon fonts, it always really hard to put them exactly the way we want it.
* To format an SVG icon, all we have to do is to set its height and width in order to size it the way we want.
* Note: svg icons are sometimes really not 100 percent vertically aligned, if you open devtools and see the svg element and the container in which the svg item is placed, you see that there is like, a small space below each of the svg icon. And that happens because these SVG elements behave like in-line elements, and so it's a bit like text that produces that small white space that we see. We can remove that by setting the line height and the font size to zero for example, but we have a simpler way of doing that by setting their parent element to Flexbox.

### Using SVG with sprite file in HTML –

<svg class="search\_\_icon">

     <use xlink:href="img/sprite.svg#icon-magnifying-glass"></use>

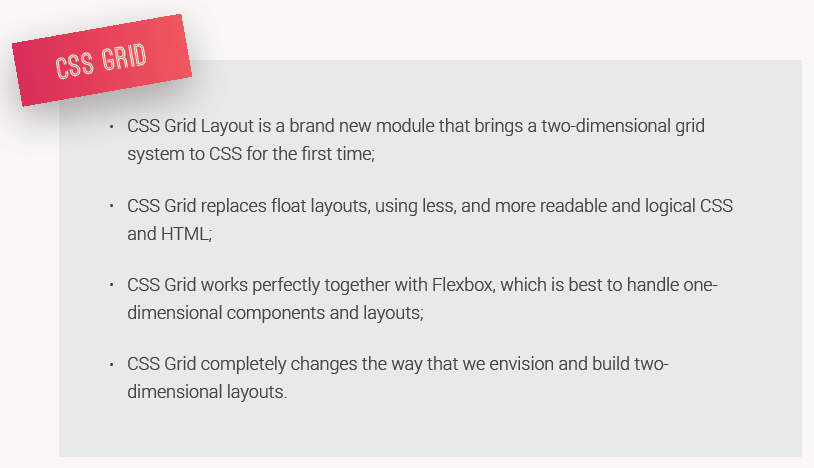
</svg>

* xlink:href="img/sprite.svg#icon-magnifying-glass"
  + that's just how we can include the SVG file and then reference each separate icon in there.
  + syntax: <svg\_sprite\_file\_path>#<svg\_icon\_name>
* This is really the best way to include SVG's in HTML using a SVG sprite.
* This way, we can actually change the color of each of the icons using CSS.
* Please Note that – Using SVG's with the 'xlink href' attribute is only going to work on a web server. So only on a web server, we will be able to see this sprite SVG file. That means if you directly open the .html file from your explorer, you won’t be able to see the SVG icons.

### Using SVG directly in CSS

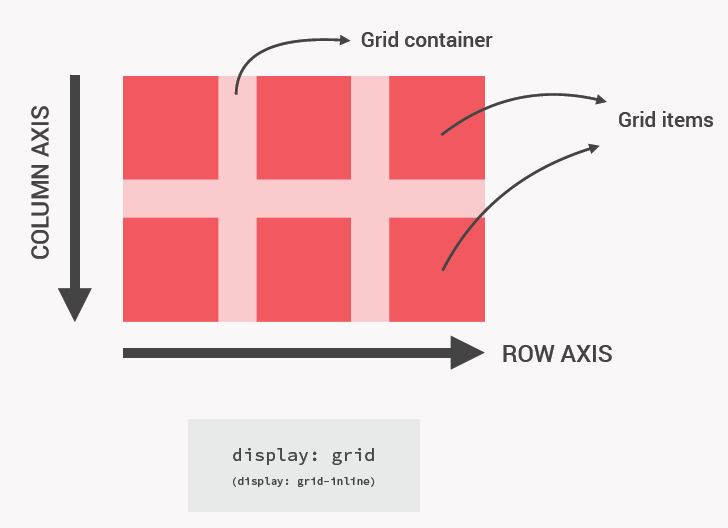
* In CSS, it's really difficult to use a sprite image like we could in the HTML.
* But in CSS, it's actually much easier to use single icon itself.
* How do we actually include an SVG file in our CSS?
  + The easiest way is to use a background-image (also the only approach for older browsers). But there is actually a problem with this approach –
    - If we include it as a background image, then we have no way of actually changing the color. But if that's not a problem for your situation, then we can of course go ahead and make it like this.
  + Another approach - is to use masks (works only in modern browsers)
    - Basically a mask defines an area where we can look through the element and see what's behind that element.
    - So, basically, what we're going do is to set the background-color of this element to a solid color, and then use the icon as the mask (by putting our mask on top of this, and only in the positions where the icon is). And so we can then look through that mask, basically, and see the background color in the area where the icon is and everything else is covered,
    - So it's basically a bit like clipping (background-clip).

# CSS Grids

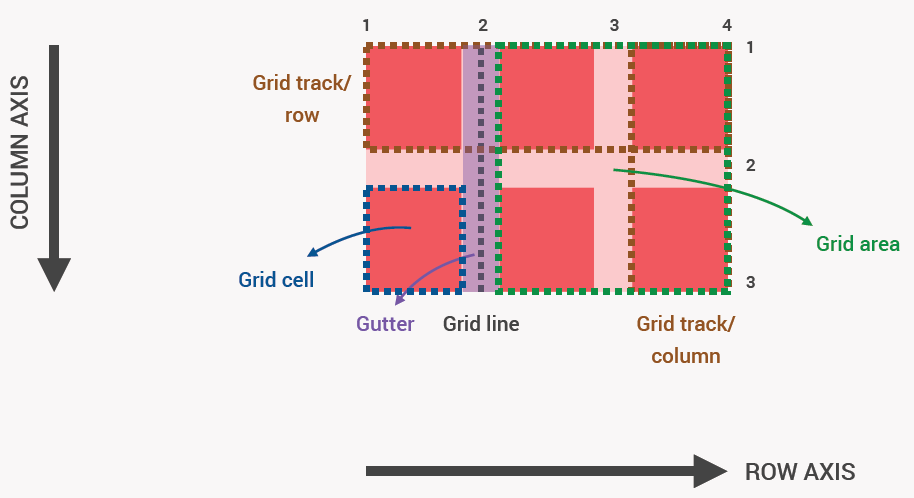


* CSS Grids allow us to build layouts so easily, like you've never done before.
* It really is a game changer for CSS, because it changes everything: how we think about layout, how we write our CSS, and even how we write our markup in HTML.
* All of that is much easier, and more logical too, with CSS grids, and so we can now finally design much more complex layouts, because we now have the tools to actually build them in our code with CSS Grid, and that actually also holds true for Flexbox, which works very well together with CSS grids, where Flexbox can handle the one-dimensional layouts, and Grids, the two-dimensional layout parts.
* Together, these two new technologies solve all the layout problems that we could never really solve before, at least not in a clean and straightforward way.
* So, by having native grids now right in our browser, we can even throw CSS frameworks, like Bootstrap out the window, because CSS is now the framework itself.
* We don't need to be bound by a simple and boring 12 column layout anymore, like Bootstrap and all these other frameworks. We're free to do whatever we want, because it's now so simple to implement without any weird hacks in our CSS, or even HTML code.
* CSS Grid requires a whole new mindset, a whole new way of thinking about design, HTML, and CSS.

## CSS Terminology

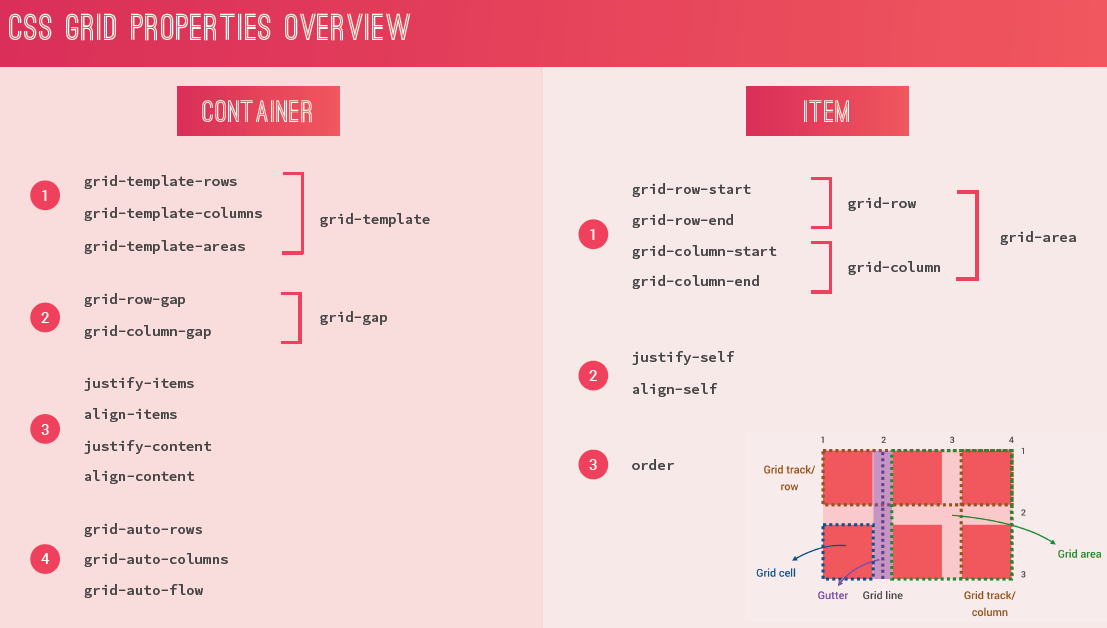


* Just like we have a flex container in Flexbox, in CSS grids, we have a **Grid Container**, and we create a new grid container by setting the display property of an element to grid.
* All the direct children of the Grid container are the **Grid Items**.
* We also have axes in Grid just like in Flexbox, but the difference is that in Grid these are called the **Row Axis in the x direction**, and the **Column Axis in the y direction**, and these axes are always the same unlike Flexbox.



* Grid Line
  + The vertical and horizontal lines that divide up the grid, and separate the columns and the rows, are called the grid lines.
  + They are automatically numbered for the columns as well as for the rows, starting from one, all the way to the number of rows plus one, and the number of columns plus one.
* Gutter
  + The actual space that is between the rows and columns is called the gutter.
  + The row gutter can be different from the column gutter.
* Grid Rows and Columns
  + The space between two grid lines is called a track, no matter if it's vertical or horizontal.
  + If it's horizontal, then it's called a row, and if it's vertical, then it's called a column.
* Grid Area
  + The area between two vertical and two horizontal grid lines is called a grid area.
  + If that area is between two adjacent grid lines and two adjacent column lines, then that area is just called a **Grid Cell**.
  + So a Grid Cell is also a Grid Area.

## CSS Grid Properties



### Grid Container Properties

* display: grid;
  + to set the Grid Container
* grid-template-rows
  + It describes number of row tracks in the grid with their height.
  + E.g. To have two rows, each with 150px height  
    grid-template-rows: 150px 150px;
* grid-template-columns
  + It describes number of column tracks in the grid with their width.
  + E.g. To have 3 columns, each with 150px width  
    grid-template-columns: 150px 150px 150px;
* grid-template
  + It’s a shorthand for grid-template-rows and columns (but not intuitive).
  + Syntax: grid-template: <grid template rows> / <grid template columns>
  + E.g. to have 2 by 3 grid

grid-template: 150px 150px / 150px 150px 150px;

* ~~grid-row-gap~~ / row-gap
  + To define grid row gutter (space between the rows)
  + ~~grid-row-gap~~ is deprecated and is replaced by row-gap
  + E.g. To have 30px space between rows  
    grid-row-gap: 30px;
* ~~grid-column-gap~~ / column-gap
  + To define grid column gutter (space between the columns)
  + ~~grid-column-gap~~ is deprecated and is replaced by column-gap
  + E.g. To have 30px space between columns  
    grid-column-gap: 30px;
* ~~grid-gap~~ / gap
  + To define same gutter space for both rows and columns at one time.
  + ~~grid-gap~~ is deprecated and is replaced by gap.
  + E.g. To have 30px space between rows and columns,  
    grid-gap: 30px;
* grid-auto-flow
  + It is by default set to row, but we can set it to column as well.
  + grid-auto-flow: row;
    - With this the automatic placement algorithm places ALL items in order to fill up the rows and not columns.
    - And the remaining items (implicit grid) are added as rows not columns.
  + grid-auto-flow: column;
    - With this the automatic placement algorithm places ALL items in order to fill up the columns and no longer the rows.
    - And the remaining items (implicit grid) are added as columns not rows.
* grid-auto-rows: 80px;
  + E.g. grid-auto-rows: 80px;
  + This will have set the implicit grid cell height to 80px, however this will be applied only if the grid-auto-flow is set to row (default)
* grid-auto-columns: 100px;
  + E.g. grid-auto-columns: 100px;
  + This will have set the implicit grid cell width to 100px, however this will be applied only if the grid-auto-flow is set to column.
* align-items
  + It will align the Grid items across the column axis (vertically) inside a grid area (or a grid cell).
  + By default, it is set to stretch. Other values are – center, start, end.
* justify-items
  + It will align the Grid items across the row axis (horizontally) inside a grid area (or a grid cell).
  + By default, it is set to stretch. Other values are – center, start, end.
  + This is not present in Flexbox because Flexbox is one dimentional and not 2-D.
* justify-content
  + It aligns the tracks inside the container across the row axis (horizontally).
  + Just as we align flex items inside of a flex container with justify-content, we do it the exact same way with the grid tracks in CSS Grid.
  + Allowed values – center (default), start, end, space-between, space-around, space-evenly.
* align-content
  + It aligns the tracks inside the container across the column axis (vertically).
  + Allowed values – center (default), start, end, space-between, space-around, space-evenly.

### Grid Item Properties

* grid-row-start
  + It is the number of the Grid row line in which we want this Grid Item to start.
* grid-row-end
  + It is the number of the Grid row line in which we want this Grid Item to end.
* E.g. To place the grid item in 2nd row   
   grid-row-start: 2;

grid-row-end: 3;

* grid-row
  + This is like a shorthand of grid-row-start and grid-row-end.
  + It is the number of the Grid row line in which we want this Grid Item to start and to end.
  + E.g. We can rewrite this  
     grid-row-start: 2;

grid-row-end: 3;

AS  
 grid-row: 2 / 3;

* grid-column-start
  + It is the number of the Grid column line in which we want this Grid Item to start.
* grid-column-end
  + It is the number of the Grid column line in which we want this Grid Item to end.
* E.g. To place the grid item in 2nd row – 2nd column (start at row line number two end at three and also start at column line number two and end at three.)  
   grid-row-start: 2;

grid-row-end: 3;

grid-column-start: 2;

grid-column-end: 3;

* grid-column
  + This is like a shorthand of grid-column-start and grid-column-end.
  + It is the number of the Grid column line in which we want this Grid Item to start and to end.
  + E.g. We can rewrite this  
     grid-column-start: 2;

grid-column-end: 3;

AS  
 grid-column: 2 / 3;

* grid-area
  + This is like a shorthand of grid-row and grid-column.
  + Syntax: grid-row-start / grid-column-start / grid-row-end / grid-column-end
  + E.g. We can rewrite this  
     grid-row: 1 / 2;

grid-column: 3 / 4;  
AS  
 grid-area: 1 / 3 / 2 / 4;

* + Honestly this is quite confusing, so it is better to use grid-row and grid-column shorthand.
* align-self
  + It will align a **specific** Grid item across the column axis (vertically) inside a grid area (or a grid cell).
  + It will basically override the align-items property set for the container.
* justify-self
  + It will align a **specific** Grid item across the row axis (horizontally) inside a grid area (or a grid cell).
  + It will basically override the justify-items property set for the container.

## Repeat Function

* Using the repeat function, we can define multiple rows or columns in an easier way. Otherwise we have to define all of them one by one. E.g.
* To have 2 row tracks, each with 150px size.  
  grid-template-rows: repeat(2, 150px);
* To have 3 column tracks each with 150px.  
  grid-template-columns: repeat(3, 150px);
* Imagine we want 1st two columns as 150px and 3rd column as 300px  
  grid-template-columns: repeat(2, 150px) 300px;
* We can put more than one tracks in the repeat function  
  E.g. grid-template-columns: repeat(3, min-content max-content);

## Fractional Unit (fr)

* What fr does is that it expands to all the space that it can occupy.
* The fractional unit represents a fraction of the available space.
* It works with both width and height depending upon where it is used.
* CSS Grid specification says that a fractional unit fills up the entire remaining space but it is never smaller than the minimum content of a row or a column. Because of this, even if you use say 1fr for 3 columns, then actual width will depend upon the content of that column. So ideally, those 3 columns should be all the same because they are all 1fr, but at the limit, when the content gets too wide, then this fractional unit simply takes the minimum width that it needs to fill the column without overflowing. This is an important thing to remember.
* E.g. 1   
  Imagine we want 1st two columns as 150px and 3rd column as 300px  
  grid-template-columns: repeat(2, 150px) 300px;  
    
  What if you wanted to make last column to occupy the rest of our container.  
  grid-template-columns: repeat(2, 150px) 1fr;
* E.g. 2  
  What if we want all of the three columns to have the same size?  
  grid-template-columns: repeat(3, 1fr);
* E.g. 3  
  If we want to have 1st and 3rd column of same size and 2nd column as double the size of 1st?   
  grid-template-columns: 1fr 2fr 1fr;  
    
  So all the empty space here is divided and then two of the fractions are assigned to the second column and then one each is assigned to the first and last.
* So you see that this fractional unit can actually be very helpful. That's exactly why it was developed specifically for CSS Grids.

## Using Percentages while defining the Grid Tracks (rows/columns)

* We cannot only use pixels and fractional units, we can use all kinds of units including percentages.  
  E.g. grid-template-columns: 50% 1fr 2fr;
* The percentage unit doesn't take the gap into account. So basically it takes 50% of the entire width (of the Grid container) and then the rest is for the other columns AND also for the gap.

## Grid Item Order

* The default behavior is that the items are laid out just like they are in the code. That is because of CSS grid's automatic placement algorithm.
* We can change the order of the Grid items using one or more of these properties

grid-row-start, grid-row-end OR shorthand grid-row

grid-column-start, grid-column-end OR shorthand grid-column

OR just grid-area

* E.g. To place the grid item in 2nd row   
   grid-row-start: 2;

grid-row-end: 3;

OR  
 grid-row: 2 / 3;

* E.g. To place the grid item in 2nd row – 2nd column (start at row line number two end at three and also start at column line number two and end at three.)  
   grid-row-start: 2;

grid-row-end: 3;

grid-column-start: 2;

grid-column-end: 3;

OR  
 grid-column: 2 / 3;

* E.g. We can rewrite this  
   grid-row: 1 / 2;  
   grid-column: 3 / 4;  
  OR  
   grid-area: 1 / 3 / 2 / 4;

## Spanning Grid Items

* We can span grid items across multiple grid cells in order to occupy entire areas.
* When we place each grid item into single grid cell, then in that case we always have grid-\*-end as grid-\*-start + 1.
* However if we want a grid item to span across multiple grid cells, then we need to have the end as start + those many lines.
* E.g. If we want an item to occupy 2 cells starting at line 2 and ending at line 4,

grid-column: 2 / 4;

Now what happens to the grid item next to that item? That item simply moves next available cell (either next column or next row), because that's what the CSS grid algorithm does.

* So if we have more items than we actually define cells, then CSS grid will automatically add a new row or a new column.
* If we have conflicting explicitly set Grid Item Orders, then you will see all these grid items stacked together. So we can then use z-index to decide the show specific item or stacking order. So basically **we can have multiple grid items in the same cell**. This is actually a technique that is used quite a lot with CSS grids, like for image galleries or stuff like that where we can have elements overlying each other to create some great effects.
* **Three ways to define spanning –**

1. Simply define the end number of the row or the column line  
   E.g. grid-column: 1 / **3**;  
   E.g. grid-row: 1 / **3**;
2. Using span keyword  
   E.g. grid-column: 2 / span 2; // to span across 2 grid cells horizontally  
   E.g. grid-row: 2 / span 2; // to span across 2 grid cells vertically
3. To span all the way until the end of our explicit grid

E.g. grid-column: 2 / **-1**;

E.g. grid-row: 2 / **-1**;

## Position grid items on the grid by naming grid lines

* To name our grid lines, all we have to do is to use the squared brackets and then put the name of the line.
* The best practice is to basically describe the content that will be in the track, and then say whether it's the start or the end.

E.g. grid-template-rows: [header-start] 100px [header-end] 200px 400px 100px;

* We can also give a line more than one name.

E.g. grid-template-rows: [header-start] 100px **[header-end box-start]** 200px 400px 100px;

* We can actually use these names to position our grid items.
* E.g. grid-row: box-start / main-end;  
  instead of grid-row: 2 / span 2;
* We can also name column grid lines the same way as row grid lines.
* We can also name grid lines along with repeat() function.
* E.g. grid-template-columns: repeat(3, [col-start] 1fr [col-end]) 200px [grid-end];

This will create named-grid lines. E.g. col-start 1, col-end 1 and at the same time col-start-2, etc.

You can use these named-grid lines like this

grid-column: col-start 1 / col-end 3;

* **For complete example** - <https://codepen.io/sameerbhilare/pen/wvGqjrd>

Refer Method 1 in this codepen

* In professional layouts, we just name all our tracks, both the row tracks and the column tracks, and then use these names to position our grid items to build our layouts.

## Position grid items on the grid by naming entire Grid areas

* The third method of positioning grid items on the grid, is by actually naming entire Grid areas.
* We simply use grid-template-areas property for creating the Grid Template Area definition. So, basically we write a couple of strings, and in there we give each grid cell a name. And then use the names defined in the template area definition to position the grid items.
* E.g. for a 4 by 4 grid, we can define the grid template area as

grid-template-areas: "head head head head"

"box box box side"

"main main main side"

"foot foot foot foot";

So here 1st row 4 cells will be called “head”,   
2nd row, first 3 cells will be called box and last will be called side,  
3rd row, first 3 cells will be called main and last as side,  
finally last row, all 4 cells will be called foot.

* So basically the grid template area just creates a representation with text of how our final layout is going to look like.
* We can now use the defined grid template areas to position our grid items.

E.g. grid-area: head;

This will place this grid item on the entire area that we defined to be the “head”.

* **For complete example** - <https://codepen.io/sameerbhilare/pen/wvGqjrd>

Refer Method 3 in this codepen

* Positioning the grid items by naming the grid areas is a completely different philosophy. In this case, we first think about layout, so where all of these items should be placed, and then all we have to do is to give the names to each of the items.
* An important point to remember is that when we create this template area definition, we need to be careful and create a complete representation of our grid, otherwise our overall layout will be not be as we intend it to be. So it is a must that all the cells must be named in the template area definition.
* To put empty cells, all we have to do is to just put a dot (.).

E.g.

grid-template-areas: "head head head **.**"

"box box box side"

"main main main side"

"foot foot foot foot";

* This method with the template area definition, is really great for smaller layouts (like upto 5 by 5).
* For huge layouts (like more than 12 by 12), it becomes very impractical to write out all of these area definitions. In such case, it's much simpler to just name the grid lines (Method 2).

## Explicit Grids vs Implicit Grids

* Suppose if we define a 2 by 2 grid using grid-template-rows and grid-template-columns, then the part of the grid with only those 4 grid cells are called Explicit Grid because we explicitly defined them to be 4 cells.  
  e.g. grid-template-rows: repeat(2, 150px);

grid-template-columns: repeat(2, 1fr);

* However if we have more grid items than we have space, then CSS grid automatically adds some more tracks to the grid so that we can then fit all of those remaining items. That particular part of the grid which is formed automatically by CSS grid is called Implicit Grid.
* How to style the Implicit Grid?
  + grid-auto-rows: 80px;
    - This will have set the implicit grid cell height to 80px, however this will be applied only if the grid-auto-flow is set to row (default)
  + grid-auto-columns: 100px;
    - This will have set the implicit grid cell width to 100px, however this will be applied only if the grid-auto-flow is set to column.
* Why are these additional grid items that we cannot fit into our explicit grid added as rows and not as columns?
  + This is because of the property grid-auto-flow.
  + grid-auto-flow is by default set to row, but we can set it to column as well.
  + When grid-auto-flow is set to column, then the remaining items are added as columns not rows. And with this the automatic placement algorithm now places ALL items in order to fill up the columns and no longer the rows.
* For complete example, refer <https://codepen.io/sameerbhilare/pen/YzqxdOw>
* This implicit grid and the ability to style these additional rows and columns is great for use cases where you don't really know how many rows or columns there will be in your grid, in your layout.

## Aligning grid items to grid areas

* For complete example, refer – <https://codepen.io/sameerbhilare/pen/xxVLMXV>
* In order to do so, we use align-items and justify-items properties of grid container.

And also align-self and justify-self properties on Grid item.

## Aligning entire grid tracks to the grid container

* For complete example, refer – <https://codepen.io/sameerbhilare/pen/BaKdbBM>
* In order to do so, we use align-content and justify-content properties of grid container.

## Managing holes (empty cells) in the Grid

* Sometimes empty cell are formed in a grid container, but that comes from the fact that the automatic placement algorithm kind of tries to keep the sequence of the items in the source code. It kind of tries to follow that when it places the grid items on the grid. And so sometimes, that can create holes (empty cells or tracks),
* But we may not always want them. So this how we can actually fix that.
  + We just use the dense keyword in order to fill that hole.

E.g. grid-auto-flow: row **dense**;

* + So with no holes (dense), all the items packed so that all the grid cells are filled with some kind of item.
  + This can be useful sometimes, like if we have an image gallery or something like that.

## Advanced topics like min-content, max-content, and minmax() function

* max-content keyword
  + Just like other units like fr, percentages, px, there is something called max-content.
  + We can use it for both grid-template-rows and grid-template-columns.
  + When used in grid-template-columns, max-content will occupy the exact width so that it can accommodate actual content.
  + E.g. grid-template-columns: max-content 1fr 1fr 1fr ;
  + It makes that column as wide as it has to be to fit its content, without having to cause any line breaks in the content.
* min-content keyword
  + When used in grid-template-columns, the column track takes the largest width that is needed to fit the largest word in the content without overflowing on the sides.
  + E.g. grid-template-columns: max-content 1fr 1fr min-content;  
    grid-template-rows: 150px min-content;
* minmax() function
  + The minmax() function is just that we pass two values in and then CSS ensures that the track will always stay between these two values.
  + E.g. If we pass into the minmax() function 100 pixels and 200 pixels for a column, CSS grid will ensure that no matter what happens, this grid column will always stay between 100 and 200 pixels width.
  + E.g. grid-template-rows: repeat(2, minmax(150px, min-content));

This means the rows should be at least 150 pixels height and at most min-content.

* + E.g. grid-template-columns: minmax(200px, 300px) repeat(3, 1fr);

Here the first column will always be between 200 and 300px no matter what the size of the viewport. (try resizing the window to understand it better.)

* + E.g. grid-template-columns: minmax(200px, 50%) repeat(3, 1fr);

Here the first column will always be between 200 and 50% of viewport width no matter what the size of the viewport. (try resizing the window to understand it better.) It will never have width less than 200px.

* + E.g. grid-template-columns: minmax(200px, 1fr) repeat(3, 1fr);
    - Here
* For complete example, refer – <https://codepen.io/sameerbhilare/pen/LYNzZPj>

## Build truly responsive layouts with auto-fit and auto-fill keywords without writing a single media query

* auto-fill
  + When used for columns, auto-fill will automatically create total number of columns based on width of the container. Total number of columns will be width divided by specified size.
  + E.g. If the container width is 1000px and   
    grid-template-columns: repeat(auto-fill, 100px);  
    then this will create 10 columns (1000 / 100 = 10)
  + So auto-fill automatically creates as many tracks with the desired width (that we specify) as fit into the container.
  + BTW, auto-fill works with both rows and columns. With rows, it manipulates height while with columns it manipulates width.
* auto-fit
  + auto-fit also works the same way as auto-fill, but the difference is that the auto-fit actually collapses unused empty tracks and gives it a width of zero and leave the empty space in the container.
  + E.g. If the container width is 1000px and   
    grid-template-columns: repeat(auto-fit, 100px);

and say we have only 8 items,

then auto-fit will create 10 columns (1000 / 100 = 10) but the last 2 unoccupied columns are simply collapsed and given a width of 0.

* + BTW, auto-fit works with both rows and columns. With rows, it manipulates height while with columns it manipulates width.
* Responsive design trick using auto-fit along with minmax() function with fractional unit.
  + E.g. If the container width is 90% (of 1000px) and   
    grid-template-columns: repeat(**auto-fit**, **minmax**(200px, **1fr**));

and say we have only 8 items,

then it occupies the entire grid container filling that extra space that was left over by expanding these 8 columns into more than 200 pixels of width.

* + Here as we try resizing the viewport the items which cannot accommodate on the available viewport width, will simply stack down on the next rows.
  + This is a huge trick that we can use to build our layouts, which will then automatically adapt to the screen size without us having to worry about anything, without having to write any media query, or anything like that.
* For complete example, refer – <https://codepen.io/sameerbhilare/pen/VwaMjxG>

## Some things missing in CSS Grid (as of 2017)

* There some things missing in CSS Grid as of 2017, but may be part of next CSS Grid specifications

1. CSS Subgrids
   1. Subgrid something grid inside a grid that can snap the tracks to the parent grid.
   2. This will be really helpful. It is I guess already arrived.
2. If we could change individual gaps (gutters). E.g. in in row 3 and 4, if we could have different gap than the rest of the rows or columns.
3. Selector where we could select nth row and nth column. E.g. if we want to just select 2nd column or 2rd row of a grid or all even/odd rows or odd rows/columns. As of now, we can’t do this with CSS Grid.

# Tips & Tricks

* There are generally two types of animations in CSS.
* The first one, which is also the easier one, is to just use the transition property, and then change the properties that you want to animate on an event, like when we hover the element.
* Second one is advanced and cooler and actually a bit more advanced and cooler. Because these animations allow us to put more options and so they are a bit more complex but they allow us to put more options. (using keyframes)
* For the browser performance, it's best to only ever animate two different properties.

One is opacity, and the other one is the transform property. The browsers are optimized for these two properties. But with transform, we can do a whole lot.

* :link - pseudo class. pseudo-classes are a special state of a selector. We use pseudo-classes to style elements under a special condition. E.g. :link, :hover, :active, etc.
* pseudo-elements allow us to style certain parts of elements.  
   E.g. ::after pseudo-element adds like a virtual element right after the element that we're selecting. And we can then style that element in order for an after pseudo-element to actually appear on the page, we need to specify its content property. It doesn't matter what the content is. It can even be empty but we have to specify it. Otherwise it's not going to appear. Same thing with the display property.
* absolute positioning needs to have a reference. And the reference is the first element with the relative position that it can find.
* To use the icons, we need to add one of the icon classes to an element. And it doesn't really matter what element is, but it’s a convention to use the <i> element. The <i> element doesn't actually stand for icon but for italics in previous versions on HTML. We no longer use i for italics, so we can use it for icons in HTML5.
* Best way to move around an element that is floated is to actually not mess with margins or anything like that, but we should just use a transform. e.g. transform: translateX(-3rem);
* Flexible images for responsive web design always need a width. Always, without any exception!
* transparent is also a valid CSS color.
* Usually all elements automatically inherit the font family, but not the input elements. So we set it manually to what we have defined for this webpage using font-family: inherit
* When you click on Full Name textbox, you will see that Email textbox moves down a bit, and that's because of the 3px that are a for the border in :focus selector, are being added. So we should actually add that border in the initial state as well, but reset it to transparent. So it will be invisible in the normal state and then as soon as we click it, as it's focused.
* currentColor is just the color of the current element or the parent element. This is really useful and powerful feature.
* z-index only works if we have position specified for that element other than static (it can be any other position like relative, absolute, fixed).
* There's an ability in the transition property to add different settings for different properties. i.e. we can have different durations and also different animation timing functions, differemt delays for different properties.

E.g.

transition: transform .2s,

            width .4s cubic-bezier(1, 0, 0, 1) .2s,

            background-color .1s;

* Flexbox Trick to use margin auto in situations like below
  + The problem:  
    With justify-content different options, we will get even or predefined spacing, however sometimes we want uneven or custom alignment of the flex items along the main axis. So in such case, we can’t use justify-content.
  + Solution 1:  
    We may use flex: 1 for the element which we want to occupy say uneven or custom space. So with flex: 1, that flex item should grow and occupy all the available space.
  + Problem with Solution 1 (using flex: 1):
    - This is not ideal, because that flex-item/element should only be the size of its content. We don't want that element to have all of the size (actual size required by the element + All the whitespaces. This can be easily visualized in devtools).
    - Imagine that we would have a hover effect on this and then all of that entire element would be hovered. So what we want is a way to create the space on right side of actual stars without actually stretching out that entire element/flex-item. And that is where the power of margin auto comes in.
  + Proper Solution (using margin auto):
    - Use margin: auto (or margin-left: auto or margin-right: auto whichever applicable depending on situation) instead of flex: 1.
    - So, the same result, but with our element actually only occupying the space that it needs and then all of the white space is the automatically calculated margin.
    - Each time we have a situation like this, or to create some space, we can use margin auto.
* :last-child applies to all of the children of the current element,   
  while :last-of-type applies only to all of the same type of element.
* With background-image we cannot change the color in CSS.
* figure element in HTML5 are perfect for reviews. figures are not only for pictures or for images, but also figures are great for some text and descriptions of the images.
* Remember that layout is a fundamental part of any website or any web app, and so if a browser doesn't support it, well, it will look really messy to that user.
* In modern CSS development, flexbox is many times used in conjunction with the newer CSS Grid Layout module, so that the overall layout of the page or the app can be done with Grid and then the smaller details can be handled by using flexbox.
* In CSS Grids, align always means vertically, and justify always means horizontally.

Also align-items and justify-items, are used for the individual items. And justify-content and align-content are used for the entire tracks.

* We can use grids not only for big layouts but also for small sections on our layouts.   
  We can nest as many grids as we can. Sometimes nested grids are very helpful to achieve smaller tasks.
* Viewport width unit (vw) us very helpful in building really responsive pages.
* CSS Automatic Placement Algorithm does not place grid items one on top of each other, but instead it moves one of the items (least specific configuration) in next cell. In order to have both the grid items on top of each other, we need to specify same grid-row value for both those items and also provide higher z-index for the item which should be on top of other.
* In order for object-fit property to work (with image or video), both width and height must be set for that element (image or video).
* Progressive Enhancements – With this we build a page with old CSS methods pretending that new CSS methods like Flexbox, Grid doesn’t even exist. And on top of top, we build the layout with modern CSS methods (Flexbox, Grids) if the browser supports it.