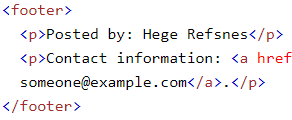
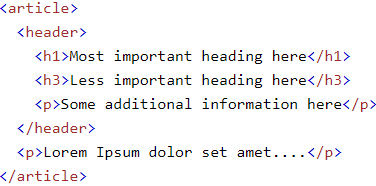
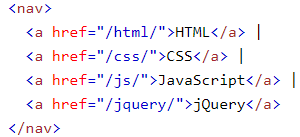
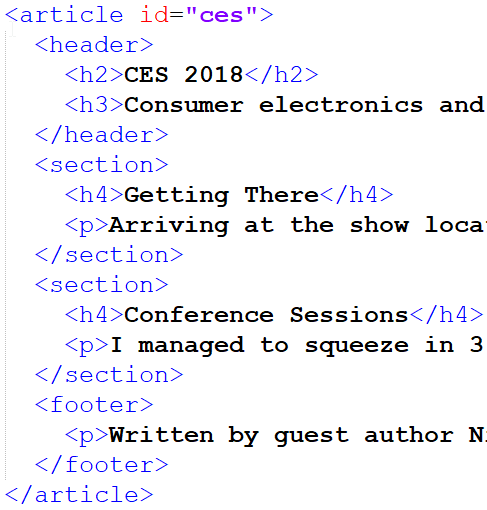
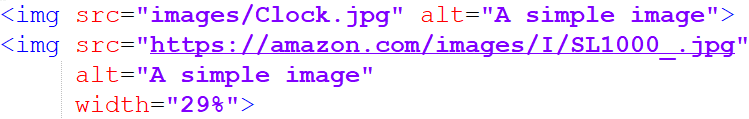
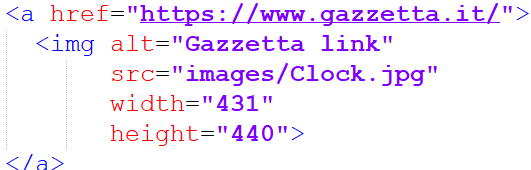
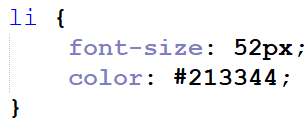
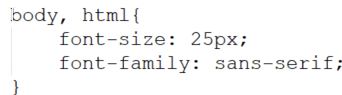
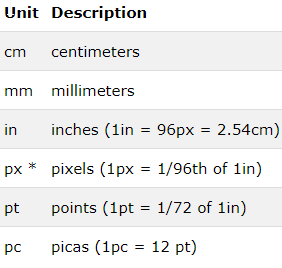
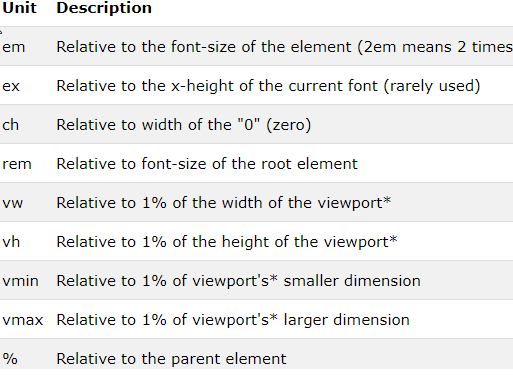
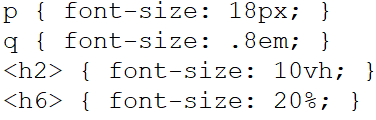
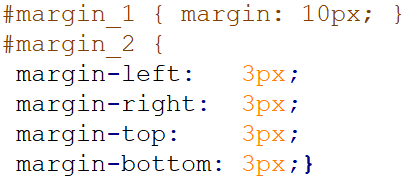
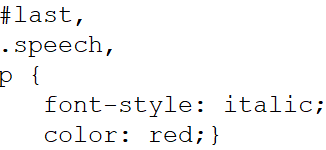
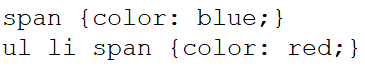
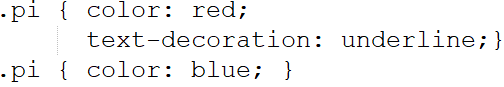
Html 5 and CSS

**W3Cx: HTML5 and CSS Fundamentals**

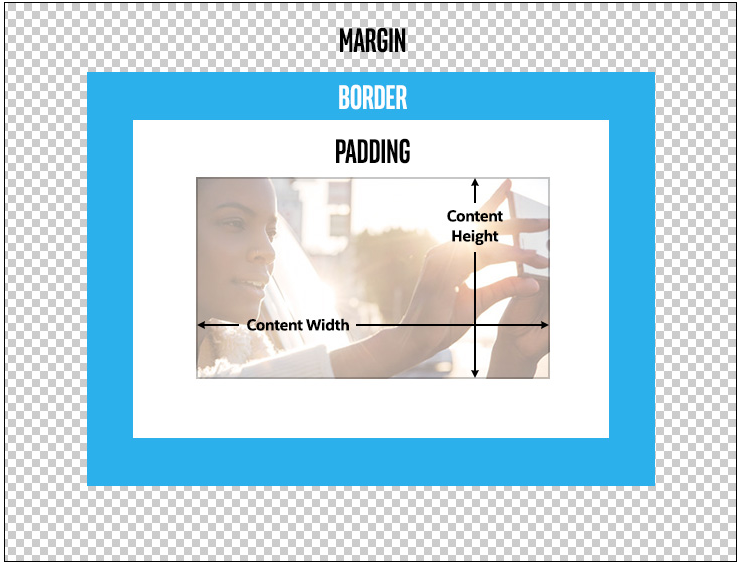
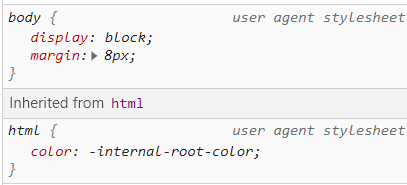
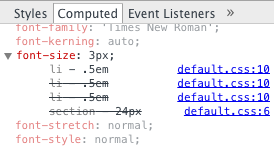
**01 Introduction to html**

* What does Markup really mean? Essentially, it means to annotate a document with extra information, things like where different sections and paragraphs begin and end, which part is the title, which things should be emphasized and so on.
* The most used tags are <p>, <h1> - <h6>, <ul> - <ol>, <hr>, <br>
  + <hr> defines a thematic break in an HTML page
  + <br> tag inserts a single line break.
* Tag names are "case insensitive", instead attributes are case sensitive.
* Double and single quotes are almost interchangeable, but they have to match.
* We have been emphasizing the general rule that HTML is for the logical structure of your content, not what it looks like. Well, nothing is perfect, including this goal. There are some HTML elements that are primarily used to satisfy certain formatting requirements: <hr> horizontal line, <br> or <br/> break line and <pre> stands for "preformatted text", meaning "text is good, don't mess with it."
* Attributes are used in tags to further define the tag: name-value pairs used inside tags and should be added after a space from the tag name. The only exception to the name-value pair is if the attribute is a 'boolean attribute'. These attributes have only two types of values - true or false. But instead of writing "true" or "false" for its value the presence of a boolean attribute on an element represents the true value, and the absence of the attribute represents the false value.
  + <p id="para1" hidden> … </p>
* We should always define the language of our pages, to do this simple add the lang attribute to the html tag:
  + <html lang="en">
* Attribute id gives a unique identify to an element, it’s useful for 2 reasons: 1) Styling your element (css). 2) Specifying a link target internal at the page. You can link to a section of your HTML page using the 'id' of the section: You should reference the 'id' value with a # preceding it:
  + <P id="plutus"> ...
  + <a href="#plutus">link to plutus</a>
* There are two kind of attributes: Global and Non-global. Global attributes can be applied to all tags (id and class are examples), non-global attributes are attributes applied to a specific instance of a tag. Start is an attribute for the <ol> tag and it cannot be applied on the <p> or <h1> tags.
* Attribute src specifies the location (URL) of an external resources:
  + <img src="smiley.gif" alt="Smiley face">
  + <script src="myscripts.js"></script>
  + <audio src="horse.ogg" controls></audio>
* Semantic HTML is HTML that concentrates on the meaning of information in Web pages instead of its presentation or look. Semantic tags, like p, suggest the purpose of the content within the tags From a semantic HTML perspective, using the right tags is important: you should use blockquote to wrap a quote and not use a paragraph tag and then style it to look like a quote. For presentation purposes, you can achieve the same using CSS, but how something looks has very little to do with what it means. This is why in HTML, we separate content and style.
* The <header> element represents a container for introductory content or a set of navigational links. <footer> element is similar it should have information about its referred elements. Header and footer elements can also be used site-wide at the top and bottom of the body of the Web page. This type of header will typically contain logos, main heading, a search area and site-wide navigation and the footer will typically include authoring information, references and other links, copyright information etc.
* <nav> element represents a section of a page that links to other pages or to parts within the page: a section with navigation links.
  + Notice that NOT all links of a document should be inside a <nav> element. The <nav> element is intended only for major block of navigation links.
* <article> and <secion> : An article element rapresents stand-alone content. If you pick an article out of a Web page, it should make sense all by itself. Section elements are used to section the content of a page, but a single section doesn't make sense by itself.
* <div> tag is one you will likely see sprinkled all over an HTML document. It is used to define a division or a section of the document. Div is not a semantic element, however, it is commonly used when there isn't a better semantic assignment for it. It is like a generic container that can hold a variety of elements such as paragraphs, images, links, tables, etc, it’s often be used to group elements for styling purposes.
* Like div, <span> is not a semantic element, you should only use span if no other semantic element is appropriate. div and span serve the same purpose but should be applied at different levels. div is a block level element (for a block of space) while span is an inline element (for within a line or phrase).
  + To add styling to part of a sentence (inline)
  + Manipulate part of a sentence using JavaScript
  + When no other HTML element is applicable, you can use span (and div) to add attributes such as class and id.
  + <p>Hi everyone! My name is Alexa and I work for <span class="company">ABC Company</span></p>
* <img> tag defines an image in an HTML page, it has two required attribute: src and alt
  + src told you where to fetch the image from: from a local patch in your web site or from the web. Use Unix (/) path name separator instead of Windows (\) style.
  + alt attribute stands for alternate text for an image. It’s important because is the text alternative to the image for users who are unable to see the image. It is also useful to provide relevant information for search engines: they do not 'see' images but rely on the alt attribute to find out what the image is about. If you use your target keyword in alt, it will optimize the search.
  + title attribute should provide additional information about the image. The title attribute should not be relied upon for important information, and it should not be used in place of the alt attribute.
  + Attribute width and height. Use of these attributes really depends on how you are using the image. If it is part of an image grid or a list with multiple images of the same size, it is best achieved by CSS. So you don't have to bother adding the same dimensions to every image and it will be repetitive. Plus it is generally bad practice to encode dimensions directly into the HTML. However, if you are adding the image into some content and it needs to be a certain size for the visual flow of the reader, then it is best to add it to HTML using the height and width attributes.
* Hyperlink <a> is any text or image you can click and it will take you to another page. Links have a default appearance in most browsers: blue and underlined. Ensure no other text in your page is underlined to avoid confusing the user. <a< isn't the same of <link>: The anchor element is used for hyperlinks while tag is used to define a link between a document and an external resource like an external style sheet. A link can lead to three different location types:
  + <a href="https://www.gazzetta.it/"> Gazzetta </a> Another web site
  + <a href="RelativePage.html"> relative URL</a> A local link
  + <a href="#plutus">Plutus </a> another part of the same page
* Target attribute specifies the destination where the linked URL in href should be opened:
  + target="\_self" : same page
  + target="\_blank" : new window
* download attribute makes a link download a file instead of navigate to another location.

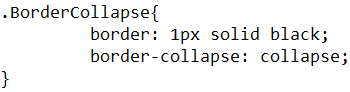
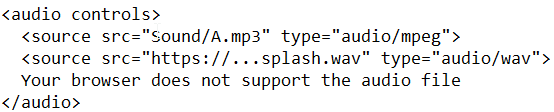
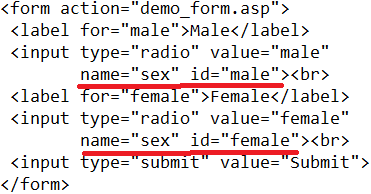
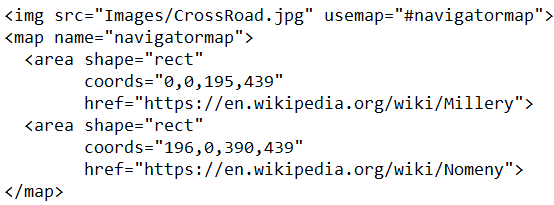
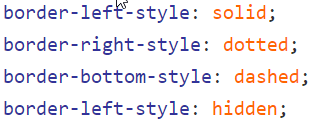
**03 Introduction to CSS**

* CSS is just a list of rules, each rule consist of a selector and a declaration. When a selector appears unprefixed by any punctuation, then it is assumed to match to an HTML tag. This li selector will apply the CSS rule to all <li> tags in the document.
* It’s possible to have a default rules that are always valid for each elements in the document. To obtain this use ‘body, html’
  + , is used to apply the same declaration to mare than one selectors.
* CSS has several different units for expressing a length. Many CSS properties take "length" values, such as width, margin, padding, font-size, border-width, etc. Length is a number followed by a length unit, such as 10px, 2em, etc. There are two types of length units: absolute and relative
  + Absolute length units are not recommended for use on screen, because screen sizes vary so much. However, they can be used if the output medium is known, such as for print layout.
  + Relative length units specify a length relative to another length property. Relative length units scales better between different rendering mediums.
* font-size: 18px; can be used to size the text of a tag, it’s possible to use different units of measure: .8em is relative to the font-size of the element (2em means 2 times the size of the current font), 20% sets the font-size to a percent of the parent element's font size…
* line-height:3 line-height property drives the height of the space where text is drawn into. A large line-height will give the text more spacing. A small line-height will smash the text lines together. The used value (an unit less number) is multiplied by the element's font size to obtain the real height value
* text-align: justify text-align valid values are: left center right justify.
* text-decoration: underline Used to underline (and similar) the text, valid value are: underline, overline, line-through, and none;
* font-weight: bold; While common values are normal and bold, text can also be made bolder (or less bold) than its parent with the values bolder and lighter. Lastly, the font-weight can be set explicitly as a numeric value: 100, 200, 300, 400, 500, 600, 700, 800 and 900 (only this number are valid).
* font-style: italic; Valid values for this property are normal and italic
* font-family: "fantasy", "Verdana", "Arial", sans-serif; Set the font of an item, browsers only guarantee a few standard choices: serif, sans-serif, monospace, cursive, and fantasy. font-family property accepts a list of possible font choices: the browser will start with trying the first font listed, and if not available (or not having a needed glyph) it will then proceed to the next font in the list, and so on.
* margin can be a bit confusing, depending upon context, it will space an item away from its immediate neighbors (in the HTML) or from the edges of its parent. Also, there is not only one margin property, but five.
* color is used to set the color of a text
  + color: blue; this simplest form allow you to choose one of the predefined color
  + color: rgb(142,34,218); any color can be specified by mixing three components together: red, green, and blue. The amount of each component falls within a range between 0 and 255.
  + rgba(142,34,218, 0.5); similarly rgba is used to have semitransparent colors.
  + color: #3B6B0A; lastly we can provide an hexadecimal (hex) code.
* Unit of measure. Many CSS properties expect some sort of dimension value. Dimension values support a wide variety of units the most common and useful ones are: px, em, rem, %, vh and vw
  + margin: 10px; px it's short for 'pixel' which is a single dot on the screen. Text with font-size:20px is 20 pixels tall on-screen.
  + font-size: 0.9em; em maps to the height of one capital letter in the parent context, it's usually used for vertical dimensions.
  + font-size: 2rem; rem is much like em, except that it always derives its size relative to the root.
  + margin-left: 5%; % is relative to the parent dimension. If the parent element doesn't have an explicit width or height set then child percentages may be percentages of 0.
  + margin-left: 5vw; margin-right: 5vw; vh stands for viewport height, and 'vw' for viewport width. They work much like the percentage % unit but instead of percentage of the parent, it is percentage of the screen (aka viewport). Obviously, vh is for vertical dimensions, and vw for horizontal dimensions.
* We can alter the aspect of a list using attribute list-style-type, this attribute can take different value for ordered and unordered list:
  + Unordered: disc, circle, square and none.
  + Ordered: decimal, decimal-leading-zero, lower-roman, upper-alpha, simp-chinese-formal.
* There are two really used selector: id selector # and class selector .
  + #par\_1 {... with # we select a unique id present inside html <p id=” par\_1”>
  + .cl {… we select a class <div class=’cl’>
* Comma separated selectors: It’s possible to combine different selector using ‘,‘ to apply to everyone the same css rules.
* Specialized selectors: If two selectors appear next to each other with no spacing separating them, they are forming a specialized selector. To match, a candidate must match both rules. If a tag selector is used it must appear first.
  + li.talk { color: purple; }
  + p#first { color: purple; }
  + .insect.flying {list-style-type: square; }
* Descending selectors (space): if it’s necessary to enter inside the tag tree we can use tags separated by a space.
  + #intro a { color: yellow; } any <a> tag that is a descendant of #intro.
  + p #guideline { color: #00FF00; }
* Direct Descending Selector > : Sometimes you don't want to apply a style to any \_possible\_ child, but to only to the direct children. This can be done with the > symbol.
  + #direct > a { color: #00FF00; } all the <a> directly inside a tag of class direct
* Everything selector (\*) : The asterisk (\*) can be used to match any tag.
  + #oneforall \* { color: #00FF00; } any tag that is progeny of a tag of class oneforall.
* Most CSS rules once applied to an element are also applied to all the children of that element, and to their children, and so on. There are exceptions, notably the layout properties (margin, padding, position, width, etc.) and the decorative properties (border, background, etc.) do not cascade. This cascading of a CSS property is called inheritance.
  + Inheritance can be explicitly set. Many CSS properties accept the value of inherit, which means to inherit the value from the parent. …{padding: inherit;}
  + There is no reliable rule for which CSS properties are inheritable by default and which are not. Generally properties associated with positioning and layout do not inherited; likewise, the decorative properties (borders, background images, etc.) do not inherit. Most properties that begin with text- or font- inherit.
* Conflicting rules. A tag can matches different selectors and the declarations can go in contrast, the browser has some guidelines for resolving conflicts:
  + A more specific rule takes precedence over a less specific rule. A rule that more tightly matches a particular element than a general rule will be applied, color red wins over blue .
  + #id selector is always the most specific.
  + .class selector is more specific than a tag selector. Rules employing a class selector (e.g. .someclass) are considered more specific than rules without (but not as specific as an #id selector, which trumps everything)
  + Rules that come later override those that come earlier. This guideline is for two CSS rulesets with the same selector, when there are conflicts, the rules from the later one apply. In the example above, an element with the .pi class will be underlined and its color will be blue
  + You may from time to time encounter a situation where you need to apply a particular CSS property and you want it to take precedence over all others, no matter what. !important will do that. The exclamation point is required, and the whole symbol ( !important ) goes after the value and before the semi-colon ;

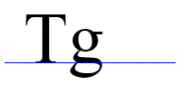
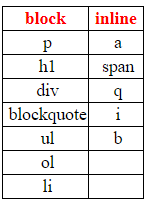
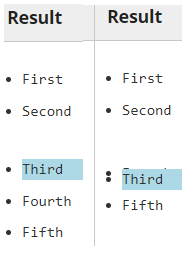
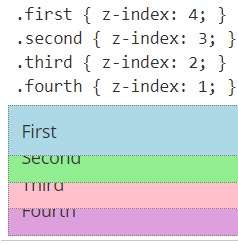
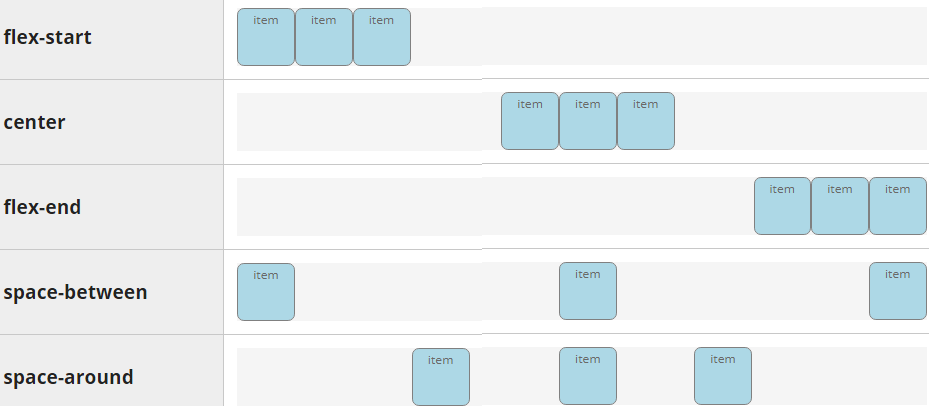
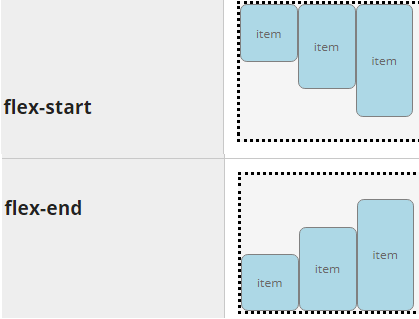
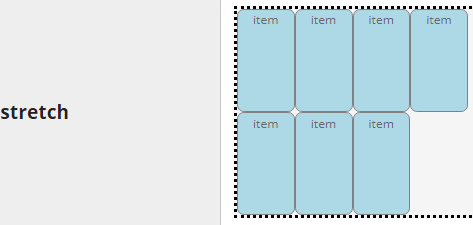
**04 Fixing and debugging**

* The Box Model governs 3 important spacing features of CSS: Margin, Border and Padding.
* All elements in an html document end up being treated as rectangles somewhere in the window. The content of each rectangle corresponds to the innermost rectangle in the image below. Just outside the content is the padding. This is kind of like an internal margin, meaning that it separates the contents from the border. The border essentially traces the sides of the padding rectangle.
* To manage Border we usually set three properties: border-width (the size of your imaginary pen), border-style (dashed, dotted, solid, etc.) and border-color (the color of your pen).
  + There is a shorthand syntax to set all three in one line: “border: 5px dotted red;”
* The margin specifies the position of the element relative to whatever is adjacent to it, either to the right or left, or top or bottom. The margin is always transparent, and each side can be set individually.
  + Values for any of the sides can be negative, even if that means that it overlaps with another element on the page.
* Padding controls the amount of space between the elements content and the border box. If you have no padding, then the contents of the element would be right up against the border. The background of the padded area matches the background of element, so the effective visible size of the element includes the padding.
* Inside css rules there are always two grayed-out section ‘User agent stylesheet’: these are basically the defaults values that the browser will use if nothing else is specified.
* When working with relative measures (.5em) styles panel doesn't tell us a lot about the actually font-size in absolute terms; we can use the "Computed" tab. It contains the values of all the CSS properties that apply to the current element.

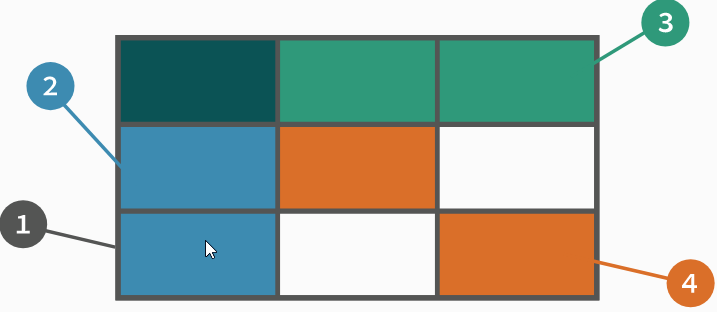
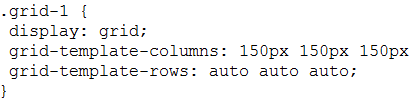
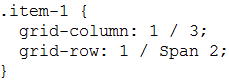
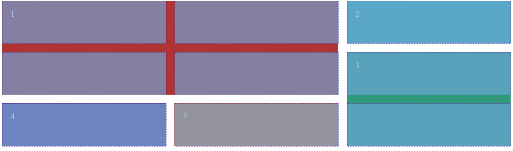
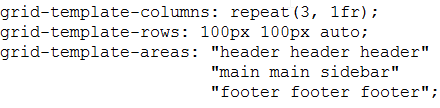
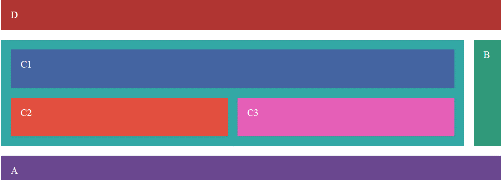
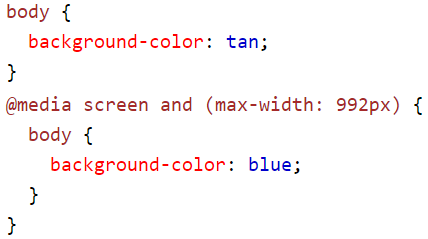
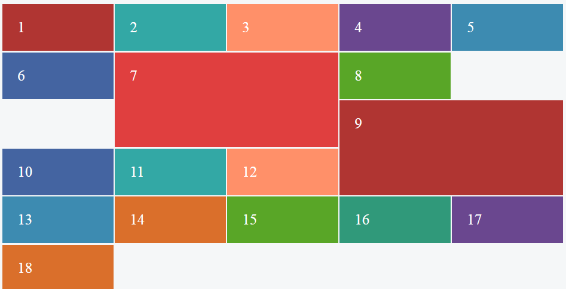
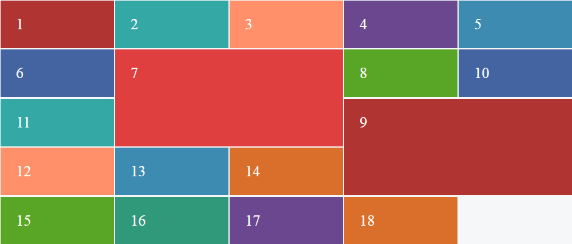
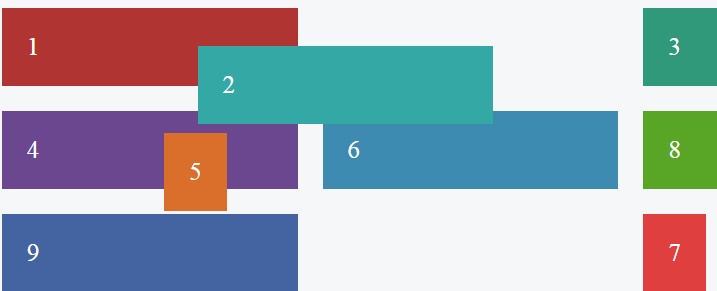
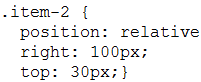
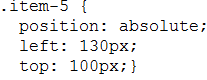
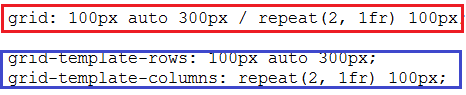
**05 More HTML and CSS**

* <table> are used to arrange data in tabular format - rows and columns of cells. You can put a variety of data like text, images, forms, links and even other tables in your table.
  + In earlier days many developers resorted to tables as a means of layout, there's really no need to do that anymore. To change the layout of the page, you shouldn't be editing your content but only your CSS.
  + To declare a row use <tr> tag.
  + Inside a row we can declare cells, there are two different types: <th> declare a header cell (bold and center by default), <td> a standard one.
    - With attributes colspan and rowspan a cell can extend on more columns and rows.
  + The <thead> element is used in conjunction with the <tbody> and <tfoot> to specify each part of a table (header, body, footer). Browsers can use these elements to enable scrolling of the table body independently of the header and footer.
  + If we gave a border to the table, table-header and table-data this creates a double line. In order to collapse them all into a single border, we use the border-collapse property.
  + Browsers automatically set the width and height for the rows and columns for your table based on the content in your cells Explicitly setting width/height of one cell will not only affect that cell but the whole column/row.
  + Using css we can select all the even and odd lines <tr> of a table:
    - .Table tr:nth-child(even){
    - . Table tr:nth-child(odd){
    - . Table tr:nth-child(3n) { with the same logic we can select all the third rows.
  + To select a row when the mouse is over it we can use: ‘.Table tr:hover{…’
* We can use <audio> tag to embed audio in our web page, in conjunction with the tag <source> it’s possible to specify multiple different source file. The advantage of providing multiple source files in different formats is that if the browser doesn't support the first format, it will automatically try the second one.
  + It’s also possible to add a test to use if the browser doesn’t support any file.
* Similarly to the previous one we can use <video> tag to embed video in our page. To select a source we can use the src attribute (single source) or the source tag (multiple source).
* Tag <iframe> allows you to put another Web page in your Web page, they are generally used in Web pages to show external content/resources. The type of content is not limited to other Web pages. You can add YouTube videos or display a PDF file. There is one significant problem with iframes: suppose you create your Web page, containing only an iframe with src="http://foo.com", with no borders, padding or margin. By all appearances, you would seem to be on the Web site foo.com. If you don't look at the URL, it might be difficult to tell, for reasons like this, some Web sites disallow their inclusion.
  + Iframes load separately from the main page, however, they do block the main page's load command until its content finishes loading. You can avoid this by applying some Javascript.
  + Useful for third party content like ads.
* Using <map> tag it’s possible to add a map to an image to enable the user to interact differently with various image’s parts.
  + The image must declare that it’s using a map with the attribute usemap.
  + A map can include any number of different <area> tags, each providing a different type of user interaction.
* What is the difference between name and id attribute? The ID of an element has nothing to do with the data contained within the element. IDs are for hooking the element with JavaScript and CSS. The name attribute, however, is used in the HTTP request sent by your browser to the server as a variable name associated with the data contained in the value attribute. With form elements the name attribute is used to determine the name-value pairs to be sent to a server-side program and should not be eliminated. Now in most elements, the name attribute has been deprecated in favor of the more ubiquitous id attribute. However, in some cases, particularly form fields (<button>, <input>, <select>, and <textarea>), the name attribute lives on because it continues to be required to set the name-value pair for form submission.
  + To ensure compatibility, having matching name and id attribute values when both are defined is a good idea. However some tags, particularly radio buttons, must have nonunique name values, but require unique id values.
  + When using radio buttons the importance is that the name remains the same throughout the options so there are no multiple selections.
* Decorative images are incorporated via CSS, not using the <img> tag. The simplest method use the background-color CSS property, it fills the rectangle of the given element with a solid background color. The background-image property is used to set an external image file as the background to a particular HTML element, the image can be local (starting point of search is the position of css file) or taken from the web. With this property there are quite a few different usage scenarios: for instance, an image can be used as repeating tile, or a background image can fit its parent element, or be a large panoramic image not fully viewed. These scenarios can be constructed with other CSS properties, like background-repeat, background-size, and background-attac.
  + By default, if the rectangular area of an element is bigger than the image itself, then the image will repeat and fill the space. The background-repeat property can be used to control this behavior, it's more commonly used values are: repeat, repeat-x, repeat-y, space and no-repeat.
  + When not repeating, it is very useful to size a background image to fit its element using the background-size property. It has two very useful values: contain and cover. The contain value will put the entire image into the space of the element, however, the space of the element may not be completely filled if the aspect ratio of the element and the image do not match (it shrinks but doesn’t enlarge the image). The cover value is the opposite. It will completely fill the element but the image may be cropped off two opposite sides (it enlarges but doesn’t shrink). Neither contain or cover will distort or squish the image, Its aspect ratio is maintained.
  + If we need a fixed dimension for the background image we can use the property background-size: it takes two values separated by a space: the first governs the width, the second the height. If none of the two values is set to ‘auto we can have image distortion.
* Using the CSS property border-style is possible to change the border of an element. We can specify four different styles one for each element’s side.
* "Pseudo Classes" is a fancy term for simply being able to refine our CSS selection to something that isn't just another element this mechanism allows us to apply styles to the different states of an element or to various children of an element based on their index, or to other interactions with the browser.
  + The most common example is the change applied to the link already visited: a:visited { color: purple; }.
  + li:hover lets you change the style of an element when the mouse is hover it. Note that the mouse is rarely hovering or clicking over/into "just one" item because if it is over a child element, it will be over the parent, grandparent, and great grandparent.
  + li:active pseudo class is applied when the mouse is depressed into its area.
  + We have already seen the selectors tr:nth-child(odd) and tr:nth-child(even), used to select the odd/even sons of en element.
* The css cursor property let you change the cursor that is displayed when the mouse is over the element. cursor: url(".images/my\_pointer.png")

**06 Base of page layout**

* The text "baseline" is a key concept to understanding how the browser makes its layout decisions, it determines how and where the characters are positioned. The baseline is never drawn by the browser, it is not exposed directly to you as a developer. As the browser is rendering your page, every time it encounters the next tag it has a simple question: "Do I give this element its own line?", css property display manages this mechanism. Different tags have different default value: <p> tag (default value block) gets a new line, but <a> (default value display) tags do not. 'Block' values always gives you a new line instead 'inline' use the current one.
* The block level:
  + Appears below and to the left of their block level neighbors (like a carriage return on a typewriter going to the next new line)
  + Will expand to fill the width of the parent container by default
  + Respects all margin properties
  + Can have its width property set, which will make it narrower and cause its children to wrap.
  + Takes on the height of all its children as long as its own height is unset.
  + Ignores the vertical-align property
* Inline elements:
  + Simply appear to the right of their preceding inline neighbor.
  + The width is simply the width of the content of the element, plus any padding
  + Ignore top and bottom margin settings
  + Ignore width and height properties
* There’s a third option for display property inline-block, it’s normally used by <img> tags
  + Inline-block elements still cleave to the text baseline of the line they are on.
  + If top or bottom margins or paddings are used, then the entire line is adjusted to make room, the vertical padding for inline-block elements contributes to the calculation of the height their line.
  + Inline-block elements respect margin-top and margin-bottom
  + Inline-block elements respect width and height properties.
* The CSS position property governs how an element is positioned on the page and how it responds to the position adjusting properties (left, top, right, and bottom).
* The default value for position is static, when using it all elements follow the standard "flowing text" model of layout and the only properties influencing their position are margins, padding, and the display property (block, inline or inline-block). Static elements ignore the positioning properties (left, top, right, and bottom).
* Relative value really means "relative to itself". If you set position: relative; on an element but no other positioning attributes (top, left, bottom or right), it will have no effect on it's positioning at all, it will be exactly as it would be if you left it as position: static; But if you do give it some other positioning attribute, say, top: 10px;, it will shift its position 10 pixels down from where it would normally be. I'm sure you can imagine, the ability to shift an element around based on its regular position is pretty useful.
  + The positioning properties (left, top, right, and bottom) adjust the placement of the element independently of its siblings. If we have a list and we want to move one of the items a little further down the page. Should we use margin-top to move it? Or position:relative in conjunction with the top property? The answer to this question depends on whether you want to move only this element(use position:relative) or you want the siblings to move down as well (use margin-top).
  + No automatic size adjustment when you use position:relative and the left or right positional properties.
* A fixed positioned element respects the positioning properties (left, top, right, and bottom). A fixed positioned element is positioned against the browser window (aka the viewport), which means it always stays in the same place even if the page is scrolled.
  + The best practice is to ensure that one of the horizontal positioning properties (that is, left or right) and one of the vertical properties (top or bottom) are both set.
  + A margin might be able to move the element but will not move any siblings
* Absolute. This is a very powerful type of positioning that allows you to literally place any page element exactly where you want it. You use the positioning attributes top, left, bottom. and right to set the location. Remember that these values will be relative to the next parent element with relative (or absolute) positioning, if there is no such parent, it will default all the way back up to the <html> element itself meaning it will be placed relative to the page itself. The trade-off (and most important thing to remember) about absolute positioning is that these elements are removed from the flow of elements on the page. An element with this type of positioning is not affected by other elements and it doesn't affect other elements (do not contribute to size of parent).
* We have seen four position properties (left, top, right, bottom) but there is another one: z-index (int value). Like the other positioning properties, z-index only applies to positioned elements (elements that have their position property set to relative, absolute or fixed, but not static), using it you can control overlapping: the higher the number, the more "topmost" or "overlapping" the element will be
  + If z-index is not set, siblings that appear later in the HTML document overlap (are "higher than") earlier siblings.
  + z-index is used to figure out which sibling is higher than another, but if two elements are not siblings (same level inside html tree), then the z-index of their respective sibling ancestors will need to be calculated to figure out which is higher.
* The Flexbox Layout module aims at providing a more efficient way to lay out, align and distribute space among items in a container, even when their size is unknown and/or dynamic. The main idea is to give the container the ability to alter its items' width/height (and order) to best fill the available space (mostly to accommodate to all kind of display devices and screen sizes). A flex container expands items to fill available free space, or shrinks them to prevent overflow.
* To designate an element as a flex container, we simply set the display property to be flex or inline-flex. A flex element will itself be a block level element, and an inline-flex element will itself be an inline element, in both cases the element is now a flex container and will be handling the layout of its children.
* Flexbox containers can lay out their children both horizontally, as in a row, and vertically, as in a column, and both at the same time. To decide how the container is going to work we use the flex-flow property. It’s an abbreviation of two different properties: ‘flex-flow: <flex-direction> <flex-wrap>’
  + The possible values for the flex-direction are: row, row-reverse, column, and column-reverse.
  + The values for the flex-wrap part are: wrap, wrap-reverse, and nowrap. It sets whether flex items are forced onto one line (nowrap) or can wrap onto multiple line
* The direct children of a flex container are automatically converted into flex items, with the exception of children that are position-fixed or position-absolute, which are taken out of the "flow" of the flex container. Empty flex items are automatically removed from the flex container. To configure a child we can use the 3 CSS properties: flex-grow, flex-shrink, and flex-basis or the abbreviate form flex.
  + The flex-grow property is a positive number. It specifies how much the item will grow relative to the rest of the flexible items inside the same container. The content of each sibling is first accounted for by the flex container when creating any row or column and only after that has been settled is any extra space distributed among the children, for this reason a higher number doesn’t always mean a bigger element.
  + The flex-shrink is the opposite of flex-grow: if the flex container needs to take away some space from the children, then those with the highest flex-shrink values contribute more of the needed space. Like flex-grow, setting the flex-shrink to 0 will prevent the flex item from shrinking. However, this may not be as desirable mechanism
  + flex-basis can be used instead of the sizing properties on a flex item. If the flex-direction of the parent flex container is row or row-reverse, then the flex-basis will govern the width of the flex item. If the flex-direction is column or column-reverse, it governs the height. The flex-basis provides the starting dimension for the flex-item, it may be grown or shrunk from that. If you do not want it to change at all, then set the flex-grow and flex-shrink to 0, and the box-sizing to border-box. BAD PRACTICE.
  + Usually all 3 properties are set toghether using the abbreviation form flex 🡪 flex: 1 1 87px; /\* use flex: <flex-grow> <flex-shrink> <flex-basis> \*/
* Some best practice with flex:
  + Try to avoid using explicit width and height properties. Instead, use the flex-basis to set a desired dimension (e.g. flex: 1 1 200px; ) or consider using min-width (or max-width) and min-height (or max-height). This will make your flex item a bit more responsive and malleable.
  + Do not over constrain your flex items. With flexbox you give the browser some general guidelines and allow it to figure it out; don't micromanage, let the flexbox do its job.
  + AVOID margin: auto on flex items
* If we are using ‘flex-flow: row wrap’ and we want to force the container to go to the next row we must simple impose the minimal dimension of the item to all the width of the screen (using min-width: 100vw; or flex: 0 1 100vw;). In this way there will not be any more space on this row and the next item will always goes on the following line.
  + Same thing for column using (using min-height: 100vh; or flex: 0 1 100vh;).
* If the flex items are fixed size, or cannot grow anymore, then the flexbox container will put the remaining extra space between or outside the items. The justify-content property is applied to the flex container, it governs how any extra space along the main layout axis is distributed between the flexbox items. The possible values are: flex-start, flex-end, center, space-between, and space-around. Remember that this is only spacing in the direction of the main axis, it does not affect any spacing or placement in the direction of the cross axis.
  + flex-start, flex-end, and center values do not distribute any space between the flex items. Instead, these values determine where the flex items should be positioned within the flex container, and any extra space is outside them.
  + The space-between and space-around values both put space evenly between the flex items, but space-between places the flex items flush against the main start and main ends of the flexbox container.
* Property align-items determines how items are aligned in the cross axis direction. This is applied to the flexbox container. The possible values are stretch, flex-start, flex-end, center, and baseline. In the context of alignment, flex-start and flex-end refer to the cross start and cross end sides.
  + align-items defaults to stretch.
* Property align-content determines how the wrapped lines are positioned or spaced, it is not applied to individual items, but rather to the wrapped lines. The align-content property supports the values stretch, flex-start, center, flex-end, space-between and space-around. They work more or less in the same way of the equivalent value of justify-content possible values, with the exception of stretch that tries to fill all the cross-axis.
* align-self property makes possible to override the align-items (that determines how items are aligned in the cross axis direction) value for a specific flex items. It will only come into play if there is extra space in the cross axis direction to be exploited.
  + flex-start: cross-start margin edge of the item is placed on the cross-start line
  + flex-end: cross-end margin edge of the item is placed on the cross-end line
  + center: item is centered in the cross-axis
  + baseline: items are aligned such as their baseline are aligned
  + stretch (default): stretch to fill the container (still respect min-width/max-width)
* Property order allows you to determine the order in which the items appears in the flexbox. This allows you to present the information in the flexbox layout independent of its order in the HTML itself. By default, the first item in a flexbox container has the order value of 1, the second is 2, etc
  + .item { order: 2; }

**07 CSS Grid**

* Grid allows us to arrange elements on a page, according to regions created by guides. These guides, or grid lines (1), frame horizontal and vertical grid tracks. Grid tracks serve as rows (3) and columns (2), with gutters running between them. Where horizontal and vertical grid tracks intersect, we’re left with cells (4), much like we use with tables. A grid area is any part of our grid fenced in by four grid lines; it can comprise any number of grid cells.
* The basic properties to transform a container element into a grid are: display, grid-template-columns and grid-template-rows (The last two define grid tracks). If we need a grid 3X3 with each column large 150px and auto seizing rows (default value) we can do as in the left figure.
  + If we place child items inside the grid container they will each take one of the nine available cells
* It’s possible to add some space between the cells using the grid-gap property on the containing element:
  + grid-gap: 20px;
* When defining gutters we can use the repeat function: *grid-template-columns: repeat(3, 33.33%);* this is exaclty the same of : *grid-template-columns: 33.33% 33.33% 33.33%;*
* An important addiction is the fraction unit ‘fr’. A single fr unit describes “one piece of however many pieces we’re splitting this into”
  + *Grid-template-columns: 2fr 1fr 1fr;* Now there’s a total of four fr units, so the first column would take up half the available width, with the other two columns each taking a quarter.
  + These units are really powerful, especially in combination with other units of measurement: *grid-template-columns: 300px 1fr 3fr 20%;* we’ve declared four columns: the first is fixed at 300px wide, the last is a flexible 20% of the grid container element wide. Then the fr units are calculated, also taking gutters gap into account, leaving the second column with one piece of the remaining space and the third with three pieces.
  + Grid accepts flexible units in combination with fixed units of measurements.
* So far our grid items have each been contained within a single grid cell, but we can achieve more useful layouts using grid area. To change the number of cells used by an item use properties grid-column and grid-row. There are three possible use:
  + *grid-column: 1 / 3;* tell the item to start at grid line 1 and end at grid line 3, using a two columns.
  + *grid-row: 1 / Span 2;* tell the item to start at grid row 1 and to spread for two tracks.
  + *grid-row: span 2;* tell the item to spread for two tracks but doesn’t set a starting position; grid auto-placement algorithm will place it in the next available (and large enough) slot.
  + The expressing power of the first two form is the same but the second is less prone to errors because doesn’t specify the ending column but only the space taken.
  + The remaining items fill the available space automatically. This highlights perfectly how a grid layout doesn’t have to reflect the source order of the elements.
* It’s important to notice that the browser tool (F12) will show the different cells of a grid even if an item use more than one.
* Property Grid Template Areas allow us to name areas on the grid, with those areas named, we can reference them (instead of line numbers) to position our items . We define these areas on our grid container, almost as though we’re drawing them out. Working on the classic 3X3 grid we have defined four areas: an header spanning across all three columns of the first line, a main taking 2 columns of the 2° line, a sidebar taking the remaning one and a footer spanning across all three columns of the last line.
* To assign an item to an area use the property grid-area followed by the destination name.
* When we declare a grid container using display: grid; only its direct descendants become grid items. Tags added inside those child elements will be completely unaffected by Grid unless we specifically say otherwise. It’s perfectly possible for an element inside a grid to be a grid on its own if that’s what we wanted.
* Media queries are used to include a block of CSS properties only if a certain condition is true (usually concerning the screen dimension). In this way we can have different rules for different devices.
  + *@media only screen and (max-width: 600px) {...}* Extra small devices (phones, 600px and down)
  + @media only screen and (min-width: 600px) {...} Small devices (portrait tablets and large phones, 600px and up
  + *@media only screen and (min-width: 768px) {...}* Medium devices (landscape tablets, 768px and up)
  + *@media only screen and (min-width: 992px) {...}* Large devices (laptops / desktops, 992px and up)
  + *@media only screen and (min-width: 1200px) {...}* Extra-large devices (large laptops and desktops, 1200px and up).
  + Always Design for Mobile First: Mobile First means designing for mobile before designing for desktop or any other device (This will make the page display faster on smaller devices).
* Our approach up until now has been to dictate how many tracks there are and watch the items fit accordingly, with the auto-fill keyword we can dictate how wide our tracks are and let Grid figure out how many will fit in the available space. We use *grid-template-columns: repeat(auto-fill, 9em);* which says “make the columns 9em wide each, and fit as many as you can into the grid container”.
* The minmax() function allows us to set a minimum and a maximum size for a track, enabling Grid to work within them. *grid-template-columns: 1fr 1fr minmax(160px, 1fr)*; first two columns being 1fr wide, the last being a maximum of 1fr, but shrinking no smaller than 160px:
* It’s possible to obtain really nice result if we combine together auto-fill and minmax: *grid-template-columns: repeat(auto-fill, minmax(9em, 1fr));* place as many 9em columns as possible, but then expand them to a maximum of 1fr until the container is filled.
  + Grid will recalculate the tracks upon page reload but it won’t do so on window resize.
* Sometime using grid-column and grid-row properties we can have some unexpected behavior: If we give Item 7 and 9 a 2x2 dimension something strange happened: item 9 no longer fits into the last column so it is pushed down to the next row. Item 10, you might imagine, would stuck itself in underneath .item-6 again, but, if you remember, it searches for a vacant column, then failing that it moves down a row and shunts across to the left again. This is an important concept to grasp.
* This grid only has five rows but item 18 is on the sixth one, Grid has assumed we’ll want another row tacking on. This is owing to grid-auto-flow, which belongs to the grid element, and whose default value is row. Possible value:
  + row Places items by filling each **row**
  + *column* Places items by filling each **column**
  + *dense* Place items to fill any holes in the grid
  + *row dense* Places items by filling each row, and fill any holes in the grid
  + *column dense* Places items by filling each **column**, and fill any holes in the grid
* We can add another keyword to our grid-auto-flow property: dense. Its default counterpart is spars. Dense packing algorithm attempts to fill in holes earlier in the grid, if smaller items come up later; this may cause items to appear out-of-order, when doing so would fill in holes left by larger items. If not specified a default sparse placement algorithm is used: it only moves forward in the grid when placing items never backtracking to fill holes. This ensures that all of the auto-placed items appear "in order", even if this leaves holes that could have been filled by later items.
* When we declare a grid, each line is given an index number, these numbers begin at the top left of the grid working their way to the bottom right. With more complex grids referencing everything by numbers might get a bit confusing, the Grid module allows us to explicitly name our lines when we declare our grid columns and rows: *grid-template-rows: [header-start] 60px [main-start] 130px [main-end] 50px [row-end];*
  + We can now position items with names: *grid-row: header-start;*
  + The original line numbers remain in operation, so you can still use them.
  + You can declare multiple names for one line, for example: *[main-end footer-start row-5]* etc.
  + There are also circumstances where lines are given names implicitly: naming a grid area header automatically assigns names to its four boundary lines too. The row lines around it become header-start and header-end, and likewise the two column lines also become header-start and header-end.
* grid-auto-columns property sets a default size for the columns in a grid container, being a default it only affects columns with a not set size. Obviously grid-auto-rows will do the same for rows as grid-auto-columns does for columns.
  + Useful when a new row/column is automatically created to make space for an item that doesn’t find a place in the explicit declared rows.
* It’s possible to use CSS positioning on grid items, just as you would with most other elements.
* We declare that item-2 is to be positioned relatively, then define some offset properties. You’ll notice, if you resize the window, that the grid item continues to behave (resize) exactly as it was before we repositioned it, and it’s still reserving its place in the grid in case it feels like coming back.
* So what happens when we absolutely position an item (5)? Firstly, it will position itself against its closest ancestor which has a declared position value, it has effectively been removed from the document flow, as is normal with absolutely positioned elements and its slot in the grid has been filled by item-6. It no longer has the dimensions it was using when it was part of the grid: it has shrunk to the size of its contents. The grid doesn’t influence the sizing of the element, and the element doesn’t influence the sizing of the grid in any way.
* It might take some getting used to, but in addition to the normal offsets you can also position a grid item using the grid-placement properties. For example, let’s place our item-7 absolutely on grid-area: 3 / 3;
  + It ‘s still unaffected by the sizing of the grid and still outside the flow.
  + We can position items on those implicit grid tracks if they exist, but Grid won’t create those tracks for elements outside of the flow.
* There are some shorcut when declaring grid, this two declaration are absolute equivalent.
* fafas