Cascading Style Sheets

Giuseppe Della Penna

Università degli Studi di L'Aquila
giuseppe.dellapenna@univaq.it
http://people.disim.univaq.it/dellapenna

> *This document is based on the slides of the Web Engineering course, translated into English and reorganized for a better reading experience. It is not a complete textbook or technical manual, and should be used in conjunction with all other teaching materials in the course. Please report any errors or omissions to the author.*

1. Introduction to CSS

1.1. The visual part of the web

Cascading Style Sheets (CSS) are a language used to define the **visual/perceptual aspect (*presentation*) of HTML documents**.

CSS describes how HTML-defined elements should be represented on **screen**, **paper**, **speech**, or other media.

On visual media, such as screen or paper, CSS can be used, for example, to define the font, color, size, and spacing of content, divide it into columns, or add animations and other decorative features.

On different media, such as speech, CSS can be used to define the tone and speed of voice used to pronounce content.

Since version 3, CSS has become much more powerful and its development is continuous.

CSS (or its derivatives) can also be used to define visual aspects outside the web, for example for graphical user interfaces of desktop applications

1.2. Style Sheets in HTML documents

A CSS is a text document, consisting of a set of *style rules*.

A document can have more than one associated style sheet.

A style sheet, in general, can be *embedded* in a document or *attached* to it. HTML:

embedded CSS style sheets are written in the document <head>, inside a <style> tag with type = "text/css".

it is also possible to write a style rule directly into a specific element using its style attribute.

finally, CSS style sheets can be imported from an external resource by writing, in the <head> section, a <link> tag with type = "text/css", rel = "stylesheet" and href = "style_sheet_uri".

Multiple Style Sheets

If you specify multiple style sheets for the same document, CSS generally combines them in order of inclusion.

However, it is possible to *conditionally* include a style sheet (or a group of sheets) in a document based on various criteria:

- Output media
- User preferences

Media Types

When you embed or link a style sheet in an HTML document, you can specify in the <style> or <link> element the *media* attribute.

The media value is a comma-separated list of names of media descriptors, which identify the type of output for which the style sheet is suitable for:

- `screen` (default)
- `tty`, `tv`, `projection`, `handheld`: indicate different types of visual terminals
- `print`: the style sheet used for printing (useful, e.g., to eliminate background colors and optional elements used in the on-screen rendering)
- `aural`, `braille`: rendering to speech and braille synthesizers
- `all`: for all types of media

Alternative Style Sheets

Moreover, in HTML you can give a page three different styles for the same media type:

- **Persistent style**: it is always loaded by the browser. The styles embedded in the document are always persistent.

- **Preferred style**: this is the default style that will be combined with the persistent, if present. It is indicated by putting a `title = "style_name"` attribute in the <link> tags.
- **Alternative styles**: these are styles that can be loaded alternately to the preferred one, depending on user preferences. They are indicated by inserting the attribute `title = "style_name"` in the <link> tag and changing the rel attribute to `alternate stylesheet`.

2. Cascading Rules

2.1. Rules

A CSS rule defines a **formatting style** and a *class of elements* it must be applied to.

A formatting style is in turn defined by a list of properties with a corresponding value, with the syntax **property: value**, surrounded by curly braces and separated by a semicolon.

The *element classes* are defined through special patterns called **selectors**.

An example of an abstract rule is

```
SEL {P1: V1 [!important] P2: V2 P3: V3}
```

The optional `!important` modifier, written after the value (but before the separator) of any property, is used to increase the *priority of the rule* during the *cascading* process, as we will see later.

Simple selectors

A simple selector is a base selector followed by zero or more *attribute selectors, class selectors, ID selectors, pseudo classes* and *pseudo elements*.

There are two base selectors:

The **universal selector** (`*`) that matches any element.

The **type selectors** (strings representing names of elements), which match any element with the given name.

Attribute Selectors

A base selector can be followed by one or more attribute selectors. The selectors of this type can be written as follows:

- `[A]` (the element must have an attribute A)
- `[A = V]` (the element must have an attribute A with value V)
- `[A ~= V]` (the element must have an attribute A whose value is a space separated list containing

V)
- `[A |= V]` (the element must have an attribute A whose value is a list separated by '-' containing V)

Class Selectors

When working with HTML, there is a special abbreviated syntax, called class selector, to work with the *class* attribute of the elements

The syntax `S.C`, applicable to any simple selector S, is equivalent to `S[class~=C]`.

As a special case, you can write a class selector alone (while in general attribute selectors should follow another valid selector), which implies an universal selector: `.C` is equivalent to `*.C`, i.e., `*[class~=C]`

ID Selectors

In XML (and HTML), you can assign each element a unique ID.

This ID can be used in CSS to apply formatting to a specific item.

The ID selector can be placed after each base selector and has the syntax `S#ID`, where S is a simple selector.

The selector `S#ID` matches the element which corresponds to the selector S and has the specified ID

It is possible (and common) to use ID selectors alone, just like class selectors, implying a universal selector: `#ID` is equivalent to `*#ID`, i.e., the element (of any type) with the specified ID.

Pseudo classes

Pseudo classes identify elements based on some special properties.

- `:first-child` (the element is the *first child* of its parent)
- `:link` *(not visited links)*
- `:visited` *(visited links)*
- `:hover` (the element currently *indicated by the user*, for example by moving the mouse over it)
- `:focus` (the currently *focused* element, namely, the one that accepts keyboard input)
- `:active` (the currently *activated* element, for example by a mouse click)

Pseudo elements

It is also possible to apply formatting to fictitious elements, not really part of the document and/or delimited by tags. These elements are called *pseudo elements*.

- `:first-line` (the first line of the text block contained in an element)

- `:first-letter` (the first character of the text block contained in an element)
- `:before`, `:after` (indicate the text position preceding and following an element, used in conjunction with the CSS *content* property)

Combination of selectors

Two selectors S and T can be combined in a third selector in various ways:

- `S T` (space between)
  This selector matches the elements designated by T only if they are descendants of an element that matches S
- `S > T`
  This selector matches the elements designated by T only if they are children of an element that matches S
- `S + T`
  This selector matches the elements designated by T only if they immediately follow an element that matches S and has the same parent.
- `S, T`
  This selector matches the elements designated by S or T (logical OR, *selector grouping)*

2.2. Style Properties Value Deduction

During the rendering process, CSS processors must determine the style to be assigned **to each element of the document**.

This involves calculating the value **of each style property** that the element can have.

To calculate the value of a property P for an element E, CSS proceeds as follows:

1. **Cascading**
   The property has a style specified through CSS rules?
2. **Inheritance**
   The property can be inherited from the parent?
3. **Default**
   The property is set to the value given by the default stylesheet.

2.3. Cascading

In a style sheet you can write several rules that match the same element (and it is often very useful).

Moreover, CSS implicitly provides, besides the author style sheet (i.e., the one associated with the document t to be formatted), two other style sheets:

- The *user* style sheet. The user may provide style rules, such as the base font size.

- The *browser* style sheet. Every browser should have its own default style sheet.

When CSS calculates the value of *each individual style property*, it takes in consideration all the rules that match the element to be formatted in any style sheet, and selects one through a **cascading** process.

Selection Rules

When multiple rules match the same element, the final value of each property is selected using the following procedure.

1. **Priority of Origin**
   1.1. !important value from the user style sheet
   1.2. !important value from the author style sheet
   1.3. Value from the author style sheet
   1.4. Value from the user style sheet
   1.5. Default value from the browser style sheet
2. **Specificity** *(for properties with the same priority of origin)*
   - A selector that is more specific to a particular element takes precedence over a more generic one.
   - In HTML, the rules included in the *style* attributes have the greatest specificity.
3. **Order** *(for properties with same priority of origin and specificity)*
   - A rule takes precedence over those that *precede* it.

2.4. Inheritance

Many properties (see the specification) are automatically *inherited* by the children of an element, if there are no specific rules that require a different value.

This default behavior is very useful when creating complex style sheets.

For example, if you specify a font for the P tag, all the tags contained therein (e.g., B) have the same font, unless a style is assigned to them (collectively or individually) that specifies a different font.

It is also possible to force the inheritance of a property by specifying (where possible) the `inherit` keyword as the value of the property itself.

3. Base CSS properties

3.1. Basic Elements of CSS

Measures and measurement units

The measures are expressed in the CSS language by numbers (also floating point and in some cases

negative) immediately followed by the name of the measurement unit.

Inserting a space between the number and the unit name usually makes the property unreadable!

The zero measure can be specified without a unit.

There are two classes of units: *relative* and *absolute*.

The *absolute* units are `in` (inches), `cm` (centimeters), `mm` (millimeters), `pt` (points = 1/72 of an inch), `pc` (picas = 12 points) and `px` (device pixel: and absolute unit relative to the device).

They are useful only when the output device is unique and precisely defined.

The *relative* units are `em` (current font-size), `ex` (current x-height). In most modern browsers, we also find `rem` (font-size of the root element), `vw` (1% of the width of the browser window), `vh` (1% of the height of the browser window), `vmin` (1% of the smallest dimension of the browser window), `vmax` (1% of the largest dimension of the browser window), `ch` (width of the character zero in the current font).

Relative units are very useful if you want to automatically adjust sizes based on the output device and its settings (e.g., printer or browser with various font sizes)

In many cases, the measures can also be expressed as *percentages*. The reference measure is usually the same property of the container element.

Colors

The colors can be defined in the CSS via the numerical RGB specification or through their own name.

The *RGB specification* allows you to define any RGB color through the value of its three components *red, green* and *blue*.

RGB hexadecimal strings have the form `#RRGGBB`, where each pair of digits represents the hexadecimal value of the corresponding component.

The short form `#RGB` is the number where each component has the value of the corresponding digit repeated twice.

RGB decimal strings are obtained using the construct `rgb(R, G, B)`, where R, G and B are numbers between 0 and 255 or percentages (i.e., fractions of the maximum 255).

Finally, the colors for which a *name* is defined are `maroon` (#800000), `red` (#ff0000), `orange` (#ffA500), `yellow` (#ffff00), `olive` (#808000), `purple` (#800080), `fuchsia` (#ff00ff), `white` #ffffff, `lime` (#00ff00), `green` (#008000), `navy` (#000080), `blue` (#0000ff), `aqua` (#00ffff), `teal` (#008080), `black` (#000000), `silver` (#c0c0c0) and `gray` (#808080)

Shortand Properties

The CSS language has many properties that are often set in a group, such as the three properties that define a border (color, width, style) or the font properties(family, size, weight, ...).

For this reason, there are also the so-called *shortand* properties, which allow, with their particular syntax, to set in a single operation the values of several properties.

In a shorthand property the values of each "component" property are simply separated by a space.

If one or more properties are omitted in the shorthand notation, their value is set to the default one.

For example, the CSS font property can be used to set all the font-style font-variant, font-weight, font-size, line-height and font-family properties.

3.2. Borders

Most items can have a border on the four sides of their boxes. Each border can have different characteristics (color, thickness, style). It is also possible, for table cell borders, to specify how neighboring borders should be combined.

Note: table borders specified through CSS are independent from those shown by the `border` attribute of <table>.

The border colors can be specified by symbolic names (e.g., white), or through their RGB components in hex (e.g., #FFFFFF) or decimal (e.g., rgb(255,255,255)) form

The border thickness is a value that can be specified in any of the measurement units understood by CSS (e.g., px, pt, mm, ...)

The main styles for the borders are `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`. However, most browsers only support solid, dotted and dashed styles.

For all of the border properties there exist shorthands that allow one to set the same values for all sides at once and/or specify the three properties (color, thickness, style) with a single statement.

CSS Properties

- `border` (`border-top`, `border-right`, `border-bottom`, `border-left`)
  Values: {*width, style, color*}
- `border-color` (`border-top-color`, `border-right-color`, `border-bottom-color`, `border-left-color`)
  Values: *color * | `transparent`
- `border-style` (`border-top-style`, `border-right-style`, `border-bottom-style`, `border-left-style`)
  Values: `none` * | *border style name* | `inherit`
- `border-width` (`border-top-width`, `border-right-width`, `border-bottom-width`, `border-left-width`)

Values: *measure*

- `border-collapse`
  Values: `collapse` | `separate` *
  Elements: tables and internal inline elements

Round corners with CSS3

Before CSS3, smooth the corners of a box required complex tricks or even images. Now you can use the `border-radius property` (**you still need to give the element a border with the properties just described**)

- `border-radius` (`border-top-left-radius`, `border-top-right-radius`, `border-bottom-right-radius`, `border-bottom-left-radius`)
  Values: *radius* | *percentage* | `initial` | `inherit`

Initial is the default value, which in this case is zero.

Image borders with CSS3

It is also possible to create borders using images instead of normal lines.

- `border-image` (`border-image-source`, `border-image-slice`, `border-image-width`, `border-image-outset`, `border-image-repeat`)
- `border-image-source`
  Values: `none` * | *image url* | `initial` | `inherit`
- `border-image-slice`
  Values : *number* | *percentage* | `fill` | `initial` | `inherit`
  The source image is divided into nine parts: four corners, four sides and the center. Up to four values can be specified, those omitted are equal to the last specified. These values determine how the image will be cut into nine parts, indicating a distance from the top, right, bottom, and left of the image.
- `border-image-width`
  Values : *number* | *percentage* | `fill` | `initial` | `inherit`
  Sets the size of the border image. Up to four values can be specified.
- `border-image-outset`
  Values : *number* | *length* | `fill` | `initial` | `inherit`
  Specifies how far the image extends beyond the natural outline of the element (border box). Up to four values can be specified.
- `border-image-repeat`
  Values : `stretch` * | `repeat` | `round` | `initial` | `inherit`
  Specifies whether images should be repeated, rounded, or stretched to cover the entire border.

3.3. Background

All the *block* elements can have a background consisting of a solid color or an image

In the case of image backgrounds, the file to be used must be indicated through the construct url('...') and it is possible to specify:

- In which position to display the image w.r.t. to the element background.
- If the image should be repeated to fill the entire space available to the element.
- If the image has to "scroll" with the content of the window or remain fixed.

Thanks to their versatility, backgrounds are often used for improper purposes, such as creating graphics (buttons, structural elements of the page, etc.) that can not be achieved simply by importing images through the HTML <img> tag.

CSS Properties

- `background-color`
  Values: *color* | `transparent`
- `background-attachment`
  Values: `scroll` | `fixed` | `inherit`
- `background-image`
  Values: *url* | `none` | `inherit`
- `background-position`
  Values: `left top` | `left center` | `left bottom` | `right top` | `right center` | `right bottom` | `center top` | `center center` | `center bottom` | *x% y%* | *measure* | `inherit`
- `background-repeat`
  Values: `repeat` | `repeat-x` | `repeat-y` | `no-repeat` | `inherit`

Shadows with CSS3

In addition to the background color, with CSS3 you can give each element a shadow.

- `box-shadow`
  Values: *horizontal_offset vertical_offset blur_radius* [*spread_radius*] *color*

Offsets (even negative) indicate where and how much the shadow will be visible behind the element.

The blur radius determines the size of the shadow blur area (which sums to the offset).

The spread radius is optionally used to increase or decrease the size of the shadow (in addition to offset+blur).

To shade text, use the text-shadow property instead.

3.4. Formatting

characters

- `color` (font color)
  Values: *color*
- `font-family` (font)
  Values: one or more font names, separated by commas, in order of priority
- `font-size`
  Values: *measure*
- `font-style` (italic text)
  Values: `normal` | `italic` | `oblique`
- `font-variant` (small caps text)
  Values: `normal` | `small-caps`
- `font-weight` (bold text)
  Values: `normal` | `bold` | `bolder` | `lighter`
- `text-decoration` (underlined text)
  Values: `none` | `underline` | `overline` | `line-through`
- `text-transform` (upper/lower case text)
  Values: `capitalize` | `uppercase` | `lowercase` | `none`

paragraphs

- `line-height`
  Values: *measure* | `normal`
- `text-align` (paragraph horizontal alignment)
  Values: `left` | `right` | `center` | `justify`
- `vertical-align` (paragraph vertical alignment)
  Values: `top` | `middle` | `bottom`
  Elements: table cells
- `text-indent` (paragraph left indentation)
  Values: *measure*
- `word-spacing` (space between words)
  Values: *measure* | `normal`
- `letter-spacing` (space between letters)
  Values: *measure* | `normal`

3.5. Lists

Using CSS, you can create bulleted and numbered lists of simple types, with standard images or numbers (Arabic, Roman, etc..) as bullets, using the `list-style-type` attribute.

The most advanced list feature, made available through the `list-style-image` attribute, allow the use

of images as bullets. In this case, the `list-style-type` attribute is ignored.

When you create custom lists with image bullets, you should always adjust the indents, margins and padding of the elements in order to achieve the desired visual effect.

Depending on your browser, the margin and/or the padding of list-item elements determine the indentation of the element itself and/or space between the bullet and the associated text.

The list-type attributes can be applied to all items whose *display* property is set to *list-item*.

CSS Properties

- `list-style-type` (standard bullets)
  Values: `disc` | `circle` | `square` | `decimal` | `decimal-leading-zero` | `lower-roman` | `upper-roman` | `lower-greek` | `lower-latin` | `upper-latin` | `armenian` | `georgian` | `lower-alpha` | `upper-alpha` | `none`
- `list-style-image` (image bullets)
  Values: *uri* | `none`
- `list-style-position` (position of the bullet relative to the item text)
  Values: `inside` | `outside` *

3.6. Dynamic Content

Prefixes, suffixes, quotes and counters

- `quotes`
  Values: `none` | (*string string*)+
  Indicates the quotes (open and closed) to use for this element, if necessary. It is possible to specify multiple pairs, one for each level of nesting of the quotation marks.
- `counter-reset`
  Values: (*string* [*integer*])+
  Clears (or sets to the given number) the value of given counters for the element.
- `counter-increment`
  Values: (*string* [*integer*])+
  Increments by one (or the given number) the values of the given counter for the element.
- `content`
  Values: `none` | (*string* | `counter(C,S)` | `open-quote` | `close-quote`)+
  Applies only to pseudo elements `:before` and `:after` and specifies the text that should be inserted before or after an element, respectively. The values of `open-quote` and `close-quote` are those set by the `quotes` attribute. *C* can be any counter visible from the element. *S* (optional) is one of the values defined for the `list-style-type`.

3.7. Cursors

Using the `cursor` property allows one to define the shape that the mouse should have when it hovers over a certain element of the page.

When creating rich interfaces, with interactive elements (clickable, movable, editable) it is important to associate them with the right cursor, to "suggest" to the user how to interact with the element itself.

- `cursor`
  Values: [*uri*,] (`auto` | `crosshair` | `default` | `pointer` | `move` | `e-resize` | `ne-resize` | `nw-resize` | `n-resize` | `se-resize` | `sw-resize` | `s-resize` | `w-resize` | `text` | `wait` | `help` | `progress`)
  Sets the shape of the mouse when it is above the element.

The list of URI, optional, indicates one or more external resources to be used as a cursor. The browser uses the first one it can retrieve.

In any case, you must also provide one of the standard cursors, as a single value or as the last choice in the list.

4. CSS Box Model

4.1. Controlling the Box Generation

It is possible to specify how the box associated with an element should be generated.

`display`
Values: `inline` | `block` | `grid` | `flex` | `inline-block` | ... | `none`

- **Block** generates a block box (which occupies a horizontal and vertical space disjoint from the other boxes, like elements div or p)
- **Inline** generates an inline box, which becomes part of the text flow without interrupting it (such as elements b or span)
- **Flex and Grid** activate the new positioning modes (*flexbox* and *grid*) that we will see later
- **Inline-block** generates an inline box that can also have a width and an height set by CSS properties (`width` , `height`), just like block boxes
  There are also many other values that allow the block to "behave" like a certain html element, for example a *list-item*, a *table*, etc.
- **None** disables the generation of the box, removing the associated element from the document.

4.2. Showing and Hiding Elements

After generating the box related to an element, you can specify whether the contents of the box should be rendered or not.

`visibility`

Values: `visible` * | `hidden`

- **Visible** (default) shows the element
- **Hidden** hides the element.

By setting the property to hidden, the element box is not removed from the document flow, so its footprint is still considered in the layout calculation.

4.3. Content Management

In general, the content of a block is limited to the size of the block itself. The content, however, may be larger than its container.

In these cases, you can specify how to handle the part that overflows the container.

`overflow`
Values: `visible` * | `hidden` | `scroll` | `auto`

- `visible` (default) allows the extra content to be rendered outside the container.
- `hidden` hides the piece of content overflowing the container.
- `scroll` brings up the scroll bars inside the container, so that the content can be scrolled. The bars appear in any case, even if the content does not overflow.
- `auto` makes scroll bars appear inside the container, so that its content can be scrolled, only if it overflows the container.

Starting from CSS3, there are also the `overflow-x` and `overflow-y` properties, with the same values but handling only horizontal or vertical overflow.

4.4. Margins and Spaces

- `margin` (`margin-right` , `margin-left` , `margin-top` , `margin-bottom`)
  Values: *measure* | `auto`
  The margin is the space between the border of an element box and that of surrounding objects.
- `padding` (`padding-top` , `padding-right` , `padding-bottom` , `padding-left`)
  Values: *measure*
  The padding is the empty space between the border of an element box and the contents of the box itself.

In many cases, margin and padding have the same visual effect, but you must always use the attribute that is logically suitable for your purpose.

By setting the horizontal margins (`margin-right` , `margin-left`) of an element to the `auto` value, the browser will *center* it in its container.

4.5. Sizing

The size of each element can be set in various ways, completely overwriting or constraining the natural size calculated by the browser.

In the first case, you can specify measures or percentages that are applied to the corresponding dimensions of the container.

In the second case, you can specify minimum or maximum sizes, expressed as measures or percentages.

Sizing Properties

- `width`, `height`
  Values: *measure* | `auto` * | `inherit`
  Set the element width and height, respectively. The `auto` value corresponds to the natural size, calculated through the other element properties.
- `max-height`, `max-width`, `min-height`, `min-width`
  Values: *measure* | `inherit`
  Set minimum or maximum size of the element.
- `box-sizing`
  Values: `content-box` | `border-box`
  Determine how the width and height of the element are actually calculated, whether or not taking into account padding and edge thickness.
  - In `content-box` mode (default), the values assigned to `width` and `height` are applied only to the internal space available for content, so the padding and the border increase these dimensions, often making the box larger than the dimensions set.
  - In `border-box` mode the values assigned to `width` and `height` will always determine the actual final dimensions of the box: in other words, these dimensions will be distributed between border, padding and interior space. *Usually this mode is the most useful when the elements need to be precisely positioned*.

4.6. Positioning

Items can be placed in four different ways, specified with the CSS attribute `position`:

- `static`: the object is placed in its natural position, given by the text flow (default).
- `relative`: the object is located at the coordinates set by the `left`, `right`, `top` and `bottom` attributes *relative to its natural position*.
- `absolute`: the object is located at the coordinates set by the `left`, `right`, `top` and `bottom` attributes, *relative to the upper left corner of its nearest container with non static positioning*.
- `fixed`: The object is located at the coordinates set by the `left`, `right`, `top` and `bottom` attributes , *relative to the upper left corner of the viewport*.

Elements with absolute or fixed positioning are **removed from the text flow**. Their position does not depend on the elements that surround them, though, if absolute, they continue to flow along with the

rest of the page.

Positioning Properties

- `position`
  Values: `static` | `relative` | `absolute` | `fixed`
  Determines the type of positioning for the element
- `top` , `left` , `right` , `bottom`
  Values: *measure*
  Determines the element position, according to the rules defined by the value of the `position` property
- `z-index`
  Values: *number* | `auto` | `inherit`
  Determines the element position on the Z axis. Higher values move the element towards the user.

4.7. Floats

The floating technique allows to remove elements from the text flow and place them in a dynamic way on the left or right edge of the container.

Floating elements are always distributed in the best possible way according to the available space.

The text outside floating elements flows around their margin.

This type of effect is often used to create menus, column layouts, etc.

Floats Properties

- `float`
  Values: `left` | `right` | `none`
  Set the object as floating on the left or right side of the container. The value `none` disables floating.
- `clear`
  Values: `left` | `right` | `both`
  The `clear` property set on an element requires all the floats of the type specified (`left` , `right` or `both`) to be placed on the page before the element itself.

5. CSS Flexbox

Flexboxes are a new way of arranging objects, now implemented in all browsers, and widely used also for the creation of complex layouts.

Flexboxes are containers with **unidirectional disposition**, i.e. the elements are placed one after the other on a single axis, although they can optionally wrap. For grid layouts there are CSS grids

however, *since the latter are still partially implemented* , flexboxes are still the preferred way to create (with some tricks) also grid layouts.

When defining a flexbox layout we must take into account two elements:

- The layout container, or *flex container*
- The elements arranged in the layout, or *flex items*

5.1. Contaniers

To make an element a flex container, the well known `display` property is used, which in this case must be set to the `flex` value (or `inline-flex` , if the container must be inline).

All items nested directly in a flex container are considered *flex items*

To configure how the flex items will be arranged in the container we can use various properties:

- `flex-direction`
  Defines the main axis of the arrangement. Possible directions are `row` , `row-reverse` , `column` , and `column-reverse` .
- `flex-wrap`
  Allows flex items to wrap if they don't all fit on the main axis. The values are `nowrap` (default), `wrap` (wrap from top to bottom in row mode) and `wrap-reverse` (wrap from bottom to top in row mode).

To configure how the flex items will **align in the container** with respect to the main axis and the secondary one we can use the following properties:

`justify-content`
defines the alignment **on the main axis**. The possible values are:

- `flex-start` , `flex-end` : items are arranged from the beginning/end of the axis (e.g. from left/right in the row direction).
- `start` , `end` : items are arranged from the beginning/end of the writing direction (e.g. from left/right in RTL mode).
- `left` , `right` : items are arranged from left/right, if this does not conflict with flex-direction.
- center: items are centered on the axis.
- `space-between` : items are arranged on the axis from end to end with uniform spacing.
- `space-around` : items are arranged on the axis so that everyone has the same amount of space around.
- `space-evenly` : items are arranged on the axis so that the space between the items and the space between the items and the edges of the container are equal.

`align-items`
defines how items are arranged **on the secondary axis *of each row***. The possible values are:

- `flex-start`, `flex-end`: items are aligned to the beginning/end of the secondary axis (e.g. from top/bottom in the row direction).
- `center`: items are centered on the secondary axis of each row.
- `stretch`: items are stretched on the secondary axis so that all the items *in each row* have the same height
- `baseline`: items are arranged so that their baseline is aligned on each row.

`align-content`
defines the alignment ***of all rows* on the secondary axis**. The possible values are the same as `justify-content`. For example, this property allows you to distribute the space between rows when `flex-direction` is `row` and elements wrap using `flex-wrap`.

5.2. Items

Each flex item can be configured individually, possibly overwriting some of the global properties of the container, using the following properties:

- `order`
  defines the items order, which are arranged in the container following the numerical order given by this property (which can also have negative values). If omitted, the internal ordering of the container obviously applies.
- `flex-grow`
  defines the item's ability to expand on the main axis if necessary. The numeric value indicates the fraction of the space to be occupied that will be used by the specific item. For example, if three elements have grow equal to 1, 2, 1, 50% of the space (2/4) will be used by the second item, and 25% (1/4) to the other two. *By default, items do not expand*.
- `flex-shrink`
  like flex-grow, but handles the shrinkage of items if there is insufficient space. The number in this case indicates the fraction of the necessary space that will be "subtracted" to the specific item.
- `flex-basis`
  indicates the base size of the item, before any empty space is redistributed. It can also be a relative measure. *Support for this property, and in particular in combination with other properties such as width, min-width etc., is still not advanced*.
- `flex`
  it is a shorthand to set `flex-grow`, `flex-shrink` and `flex-basis`.
- `align-self`
  allows one to change the `align-items` alignment mode of the container for this specific item.

6. CSS Grids

Grids are an advanced CSS feature that allows you to **define grids** (rows by columns) and **distribute content on them** . This is the "ultimate solution" to the problem of the well-known *grid layouts*,

currently made with floats or flexboxes using some tricks.

However, this specification is still not well implemented in the browsers, and it is not completely safe to adopt it, for example, to create layouts as a (better) replacement of the flexbox solution.

In a grid you first configure a *container* by dividing the internal space with a *virtual grid* (which does not correspond to any HTML element, as for grids made with other techniques or tables)

Elements inside the *grid container* (*grid items*) occupy the grid spaces according to the directives associated with them, or automatically. In the latter case, the elements will be distributed naturally, in the given order, trying to occupy all the cells, row by row.

To make an element a grid container, you use the well known `display` property, which in this case must be set to the `grid` value (or `inline-grid`, if the container must be inline).

6.1. Rows and columns

The structure of the grid inside the container can be defined in various ways.

You can define **rows and columns** by specifying their size using the `grid-template-columns` and `grid-template-rows` properties.

Each property accepts a *sequence of measures* (which can also be percentages or the `auto` keyword), for instance

```
grid-template-columns: auto 50px 20px 40px;
grid-template-rows: 25% 100px auto
```

define a grid with four columns (of which the first will take all the "remaining" space) and three rows.

If you insert more rows into a grid container than you declare, new rows are automatically created. In this case, their size can be specified using the `grid-auto-rows` property.

You can **assign names to the ends of each cell (border)** by placing them in square brackets. *Attention! We are not giving names to the cells, but to their edges!* For example

```
grid-template-columns: [c1 cstart] auto [c2 c1end] 50px [c3] 20px [c4] 40px [cend];
grid-template-rows: [r1 rstart] 25% [r2] 100px [r3] auto [rend]
```

define the same grid as in the previous example, where the left end of the first column is called "c1" or "cstart", its right end (which is also the left end of the second column) is called "c2" or "c1end", and so on.

The same goes for the upper and lower ends, defined in the row specification. You do not need to specify names for all borders.

When specifying row or column sizes, you can also use the special unit of measurement `fr` (*fraction*), which indicates a fraction of the available space.

You can define the spaces between rows and columns using the `grid-column-gap` and `grid-row-gap` properties.

6.2. Areas

It is also possible to **give names to the actual grid cells** (always after defining them with the two properties `grid-template-rows` and `grid-template-columns`), which is often conceptually clearer, using the `grid-template-areas` property. For example

```
grid-template-areas: "header header header header" "main main
. sidebar" "footer footer footer footer"
```

The property value consists of a sequence of strings (in quotation marks), each representing a line.

Each string contains area/cell names, separated by spaces. A dot indicates an unnamed cell (which will likely remain empty)

Repeating the same name creates an area that expands (spans) across multiple cells.

The number of rows and columns specified must match what is defined with `grid-template-rows` and `grid-template-columns`.

6.3. Templates

Finally, it is possible (but complex) to summarize the three properties just seen in a single definition using the `grid-template` property, for example

```
grid-template: [r1 restart] "header header header header" 25% [r2] "main main . sidebar"
100px [r3] "footer footer footer footer" auto [rend] / [c1 cstart] auto [c2 c1end] 50px [c3] 2
```

The part of the specification preceding the slash (/) defines the rows (possibly giving names to the top and bottom edges) with the specification of their areas and their size.

The specification part after the slash defines the columns of all rows exactly as done with `grid-template-columns`.

6.4. Alignment

To configure how the content cells content will align with respect to the their edges we can use the following properties:

`justify-items`

defines the horizontal alignment of cell contents. The possible values are:

- `start`, `end`: elements are arranged at the beginning/end of the cell.
- `center`: elements are centered on the cell.
- `stretch`: elements are stretched so that they occupy the entire width of the cell (default).

`align-items`
defines the vertical alignment of cell contents. The possible values are the same as `justify-items`.

We omit other more advanced properties, as they are poorly supported.

Each cell can have its own values for alignment properties, which override global properties: for example `justify-self` and `align-self`.

6.5. Items

You can specify in which cell (or cells) each item should be placed, instead of letting it be arranged according to the natural order.

The first arrangement mode refers to the edges of rows and columns (defined by grid-template-rows and grid-template-columns) through the four properties `grid-column-start`, `grid-column-end`, `grid-row-start`, `grid-row-end`, for example

```
<div style="grid-column-start: cstart; grid-column-end: cend">
 C1
</div>
<div style="grid-column-start: c2; grid-column-end: c4; grid-row-start: r2; grid-row-end: rend
 C2
</div>
<div style="grid-column-start: c4; grid-column-end: cend; grid-row-start: r2; grid-row-end: r3
 C3
</div>
<div style="grid-column-start: c4; grid-column-end: cend; grid-row-start: r3; grid-row-end: re
 C4
</div>
```

With reference to the grid defined in the previous slides,

- C1 will extend horizontally from the edge "cstart" to the edge "cend" and vertically will occupy the first row (since grid-row-start and grid-row-end have not been specified, but in this case the result may vary),
- C2 will extend horizontally from edge 'c2' to edge 'c4' and vertically from edge 'r2' to edge 'rend',
- C3 will extend horizontally from edge 'c4' to edge 'cend' and vertically from edge 'r2' to edge 'r3',
- C4 will extend horizontally from edge 'c4' to edge 'cend' and vertically from edge 'r3' to edge 'rend'.

These properties also have a more complex syntax, which is not explored here.

The second way of arranging items in a grid refers to areas defined with `grid-template-areas` through the `grid-area` property, for example

```
<div style="grid-area: header"> C-h </div>
<div style="grid-area: main"> C-m </div>
<div style="grid-area: sidebar"> C-s </div>
<div style="grid-area: footer"> C-f </div>
```

With reference to the grid defined in the previous slides,

- C-h will occupy the four cells of the first row called "header",
- C-m will occupy the two cells of the second row called "main",
- C-s will occupy the cell at the far right of the second row called "sidebar",
- C-f will occupy the four cells of the third row called "footer".

This property also has a more complex syntax, which is not delved into here.

7. Responsive Design with CSS

7.1. CSS Media Queries

Media queries are one of the most important features introduced by CSS3 and supported by all modern browsers.

Media queries can be used:

- Within the `media` attribute of the <link> tag, to import a stylesheet only if the query is satisfied, or
- Directly in the stylesheet, enclosing a set of rules between braces preceded by the `@media` *at-rule*

A media query consists of a *media type* (*screen*, *print*, etc.) combined through an AND with e sequence of *media features*, whose support may vary in browsers.

- `orientation` : [`portrait` | `landscape`]
- `width` : *measure*, `min-width` : *measure*, `max-width` : *measure*
- `prefers-color-scheme` : [`light` | `dark`]

It is possible to merge (OR) several media queries using a comma, for example

```
@media screen and (min-width: 300px), screen and (orientation: landscape) {...}
```

This media query is true if we are viewing on screen and the horizontal size is at least 300 pixels or the screen orientation is horizontal.

7.2. Responsive Design

Responsive design is a layout design technique introduced by Ethan Marcotte in his 2010 article on A List Apart:
http://alistapart.com/article/responsive-web-design

With this technique the website layout **dynamically adapts to the size and the characteristics of the output device**.

The three components of a responsive design are:

- fluid layout (typically grid-based)
- CSS3 media queries
- flexible images

When CSS media queries were not available, responsive design was achieved with the help of fluid design supported by scripts, but now you can get much more advanced effects using only stylesheets.

By applying specific changes to the stylesheets through media queries, it is possible, for example, to hide items, relocate others, decrease borders and spacings, etc..

Breakpoints

The so-called *responsive breakpoints* are the limit values of certain properties (typically the width of the *viewport*) below and above which media queries activate to modify the visual structure of the site. For example:

- Default ("normal" browsers)

Fluid design, with percentage widths (possibly also for images). A liquid design makes the layout robust also for devices that do not support media queries.

- Viewport from 768 to 959 pixels (tablets):

```
@media only screen and (min-width: 768px) and (max-width: 959px)
```

Remove "creative" paddings and spacings, slightly decrease the font size, etc..

- Viewport smaller then 768 pixels (mobiles):

```
@media only screen and (max-width: 767px)
```

Use a fixed design, for example 320 pixels wide, or use a min-width on the main elements so that they don't become too narrow. Linearize columns, hide secondary elements (parts of the header and footer, etc.), show more compact menus...

7.3. A Grid Layout with Floats

Now we will see how to create a **liquid grid layout**, useful when webpage elements should be aligned in rows and columns, exploiting only *floats*.

This kind of structure is the basis for **responsive** layouts, because it adjusts to the size of the browser. It is also possible to **nest** these grids, to obtain very complex effects.

Sources:

- The base of this layout is the 960 grid system (http://960.gs/) which, however, is fixed (width 960 pixels) and does not use media queries.
- The media queries part is inspired by Skeleton (http://www.getskeleton.com/), which, however, remains fixed for widths greater than 960 pixels.
- Finally, the inspiration for the liquid version of the grid is taken from http://www.designinfluences.com/fluid960gs/.

See also the introductory article on *fluid grids* by Ethan Marcotte: http://alistapart.com/article/fluidgrids

The rows

This design allows to place up **to 16 columns on each row**.

Row cells have a 1% **spacing** on both sides.

The *container* and *row* classes represent the grid and the row containers, respectively:

```
.container { position: relative; width: 98%; padding: 0;}
```

The grid container has a small extra margin. It is not strictly needed, and it is safe to write rows outside a container.

```
.row { margin-bottom: 10px; }
```

Rows have a little spacing below, that can be removed.

The columns

The *column* class, defines the common column properties:

```
.container .column,
.container .columns { float: left; display: inline; margin-left: 1%; margin-right: 1%; }
```

alpha and *omega* (i.e., first and last) columns have no lateral margin:

```
.column.alpha,
.columns.alpha { margin-left: 0; }

.column.omega,
.columns.omega { margin-right: 0; }
```

Classes *one*, *two*, etc. define the horizontal span of the cell, in columns (1-16):

```
.container .one.column { width: 4.25%; }
.container .two.columns { width: 10.5%;}
```

Horizontal widths for columns from three to sitexteen are, respectively, 16.75%, 23%, 29.25%, 35.5%, 41.75%, 48%, 54.25%, 60.5%, 66.75%, 73%, 79.25%, 85.5%, 91.75%, 98%

Floats encapsulation

To conclude, some cross-browser tricks help to maintain the float columns inside their row:

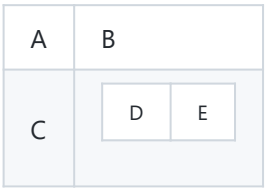
```
.row:before,
.row:after {
 content: '\0020';
 display: block;
 overflow: hidden;
 visibility: hidden;
 width: 0; height: 0;
}

.row:after { clear: both; }

.row{ zoom: 1; }
```

Example

```
<div class="row">
 <div class="three columns">
  A
 </div>
 <div class="thirteen columns">
  B
 </div>
</div>
<div class="row">
 <div class="three columns">
  C
 </div>
 <div class="thirteen columns">
  <div class="row">
   <div class="eight columns">
    D
   </div>
   <div class="eight columns">
    E
   </div>
  </div>
 </div>
</div>
```



Media queries

The grid layout *becomes linear* below a certain width: this is achieved by disabling the *floating* and not forcing a column width, so that the columns can "wrap".

In addition, in this example we fix the size of the entire layout to prevent it from falling below a minimum threshold.

```
@media only screen and (max-width: 767px) {
.container {width: 300px;}
.container .columns, .container .column {margin: 0;}
.container .one.column,
.container .one.columns, ... {float: none; width: auto;}
}
```

7.4. A Grid Layout with Flexbox

We will now see how to create a **liquid grid layout** using flexboxes.

The rows

The *container* class is identical to the floats version:

```
.container { position: relative; width: 98%; padding: 0;}
```

The *row* class activates flexbox positioning within it:

```
.row {
 display: flex;
 flex: 1 0 auto;
 flex-direction: row;
 flex-wrap: wrap;
 margin-bottom: 10px;
}
```

The row is declared as a flex container (`display: flex`) whose direct children will be aligned horizontally (`flex-direction`), but will wrap if necessary (`flex-wrap`).

In addition, the row has also the properties of a flex child (see below), so that it can be nested in cells.

The columns

The *column* class defines common column properties:

```
.container.column,
.container.columns {
 display: flex;
 flex: 1 0 auto;
 flex-direction: column;
 max-width: 100%;
 margin-left: 1%;
 margin-right: 1%;
}
```

The `flex` property (shortcut for `flex-grow, flex-shrink` , and `flex-basis`) governs the behavior of flex elements in the container.

`flex-grow` (here 1) indicates the proportion with which the element will grow relative to the others, while `flex-shrink` indicates the proportion with which it will shrink (here 0 indicates that the element will not shrink beyond its base size).

`flex-basis` (`auto`) sets the base size on the basis of which the remaining space will be distributed.

The `display` and `flex-direction` properties are used to configure cells also as containers for other

flex groups (rows, hence flex-direction set to `column`)

Classes *one*, *two*, etc. allow one to define the width of each cell as the number of occupied columns (1-16) and overwrite `flex-basis`:

```
.container.one.column { flex-basis: 4.25%; max-width: 4.25%;}
```

The sizes for cells three to sixteen columns wide are: 16.75%, 23%, 29.25%, 35.5%, 41.75%, 48%, 54.25%, 60.5%, 66.75%, 73%, 79.25%, 85.5%, 91.75%, 98% respectively.

Media queries

Media queries are adapted accordingly, disabling the flex when necessary:

```
@media only screen and (max-width: 767px) {
.container { width: 300px; }
.container .columns, .container .column { margin: 0; flex: none }
.container .one.column,… { flex: none; display: block; width: auto; max-width: 100%;}
.row { flex: none; display: block; width: 100% }
}
```

7.5. Cross-Browser Compatibility

Reset Stylesheet

Different browsers apply to the properties **different default values**, corresponding to their *default stylesheet*.

To achieve crossbrowser CSS, you may want **to reset these differences** and create style sheets from a **minimum common basis**. This can be achieved by inserting at the beginning of the style sheet some rules like the following.
(http://meyerweb.com/eric/tools/css/reset/)

```
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code, del,
dfn, em, img, ins, kbd, q, s, samp, small, strike,
strong, sub, sup, tt, var, b, u, i, center, dl, dt, dd,
ol, ul, li, fieldset, form, label, legend, table,
caption, tbody, tfoot, thead, tr, th, td, article,
aside, canvas, details, embed, figure, figcaption,
footer, header, hgroup, menu, nav, output, ruby,
section, summary, time, mark, audio, video
{ margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}

article, aside, details,
figcaption, figure, footer,
header, hgroup, menu, nav, section /* HTML5 */
{ display: block; }

body
{ line-height: 1; }

ol, ul
{ list-style: none; }

blockquote, q
{ quotes: none; }

blockquote:before,
blockquote:after,
q:before, q:after
{ content: ''; content: none; }

table
{ border-collapse: collapse; border-spacing: 0; }
```

8. References

Cascading Style Sheets (CSS)
https://www.w3.org/Style/CSS

9. Examples

The main samples explained or developed during the classes are outlined below. These examples are all available in the GitHub, at the address https://github.com/orgs/WebEngineering-Univaq, and are a *key component* of the classes itself, since they illustrate the practical use of the concepts presented during lectures and reported in this documentation (where, when possible, references to these examples can also be found).

The list below may not always be up to date: in the repository you may often find useful new examples that have just been developed.

Basic Examples

- css_inclusion_print.html
  Shows how to insert a print-specific stylesheet into an HTML document
- css_selectors.html
  Shows various examples of use and combination of CSS selectors
- css_cascade.html
  Examples and exercises on cascading in CSS
- css_properties_common.html
  Shows the use of various basic CSS properties such as `font`, `color`, `background`, `border`
- css_3_webfonts.html
  Show the use of the CSS at-rule `@font-face`
- css_3_round_borders.html
  Show the use of the `border-radius` *CSS property*
- css_3_imageborders.html
  Show the use of the `border-image` *CSS property*
- css_3_shadows.html
  Shows the use of the `box-shadow` *and* `text-shadow` *CSS properties*
- css_menu_multilev_static.html
  Shows how to turn a simple list into a menu using the `list-style` *property along with other basic CSS properties*
- css_properties_misc.html
  Shows the use of various advanced CSS properties such as `content`, `quotes`, `counter-increment`, `cursor`

Box model and positioning

- css_properties_display.html
  Shows the use of the CSS property `display`
- css_properties_overflow.html
  Shows the use of the CSS property `overflow`
- css_properties_positioning_relative.html
  Shows how to use relative positioning (`position: relative` *)**

- css_properties_positioning_absolute.html
  Shows how to use absolute positioning (`position: absolute`)
- css_properties_positioning_fixed.html
  Shows how to use fixed positioning (`position: fixed`)
- css_properties_positioning_float.html
  Shows the use of the CSS properties `float` and `clear`
- css_menu_multilev_flyout.html
  Shows how to turn a simple list into a flyout menu using the `list-style`, `display` and `position` properties along with other basic CSS properties

Flexbox and Grid

- css_3_flexbox.html
  Shows various flexbox properties
- css_3_grid.html
  Shows various grid properties

Advanced Topics

- css_3_transitions.html
  Shows the use of the CSS property `transition`
- css_3_animation.html
  Shows the use of the CSS property `animation` (and the at-rule `@keyframes`)
- css_sprites.html
  Shows how to use the CSS sprites technique to download multiple images efficiently
- css_3_variables.html
  Shows the use of CSS variables and constructs `var()` and `calc()`

Title: Cascading Style Sheets
Author: Giuseppe Della Penna, *University of L'Aquila*
Version: 20240220