CSS 1

Date de création	@3 juin 2024 16:59
Cours	CSS
Type	O'clock
Relue ?	☐
Mise en page ok ?	oui
Prise de note ajoutée	☐

CSS

Le **CSS** ou **C**ascading **S**tyle **S**heets (feuilles de style en cascade) offre la possibilité de styliser des langages de balises comme le HTML. Appliquer des couleurs, définir des marges, positionner des éléments et bien d'autres choses...

Là où le HTML représente le **fond**, le CSS lui, représente la **forme**

Où écrire du CSS

Dans l'attribut style d'une balise

On parle de *style inline* dans ce cas.

```
<p style="color: blue;">Lorem ipsum dolor sit amet</p>
```

Dans la balise `<head>`

Dans une balise `<style>...</style>`

On parle de *style interne*. Le CSS écrit à l'intérieur de la balise `style` ne s'appliquera que dans la page courante.

```
<html>
  <head>
    <title>Hello</title>
    <style>
       p{ color: blue; }
    </style>
  </head>
  <body>
    <p>Lorem ipsum dolor sit amet</p>
  </body>
</html>
```

Dans un fichier séparé

On parle de *style externe*. Grâce à cette méthode, on pourra réutiliser facilement le CSS dans d'autres pages HTML.

Le fichier CSS prendra `.css` pour extension

```
/* Fichier styles.css */
p{ color: blue; }
```

La liaison avec le HTML est assurée par une balise `<link>`

```
<html>
  <head>
    <title>Hello</title>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <p>Lorem ipsum dolor sit amet</p>
  </body>
</html>
```

Afin d'assurer la séparation entre le contenu (fond) et l'apparence (forme)

SoC ⇒ Separation **o**f **C**oncerns

cette méthode est la voie à suivre

La syntaxe CSS

Structure d'une règle CSS

```
selecteur{
  propriete: valeur;
}
```

- **selecteur** : Cible (élément HTML)
- **declaration** : couple propriété-valeur
- **propriété** : Ce que l'on souhaite modifier
- **valeur** : Comment souhaite-t-on le modifier

Mise en pratique

```
<p>Lorem ipsum dolor sit amet</p>
```

```
p {
  background: black;
}
```

Les sélecteurs peuvent être cumulés grâce à une `,` et une règle CSS peut également avoir plusieurs ensemble `propriete: valeur;`

```
p,
h2 {
  background: black;
  color: white;
}
```

Commentaires

Les commentaires en CSS sont déclarés avec `/* ... */`

```css
/* un commentaire en CSS */
p,
h2 {
  background: black;
  color: white;
}
/*
Idéal pour laisser des instructions
ou des rappels
*/
```

Sélecteurs

Balises

Cible un élément par le nom de la balise

```css
a { /* Tous les liens */ }
p { /* Tous les paragraphes */ }
li { /* Tous les éléments de liste */ }
```

IDs

En se basant sur l'attribut HTML `id` il est possible de cibler l'élément grâce à `#` en CSS

```html
<div id="chapeau">Le chapeau de l'article</div>
```

```css
#chapeau {
  color: blue;
}
```

Classes

En se basant sur l'attribut HTML `class` il est possible de cibler les éléments grâce à `.` en CSS

```html
<div class="important">Un texte très important</div>
```

```css
.important {
  color: red;
}
```

Classes multiples

Il est possible de mettre plusieurs classes pour un seul et même élément. Il suffit pour cela de mettre un espace entre les classes dans l'attribut class :

```
class="classe1 classe2 classe3"
```

C'est très utile lorsque plusieurs éléments ont les même styles sauf quelques propriétés.

On définit les styles communs sur une classe commune, et on définit les styles différents, sur une classe spécifique.

```html
<div class="important rouge"> Un texte très utile et rouge </div>
```

```
<div class="important jaune"> Un texte très utile et jaune </div>
```

```
.important {
  font-weight: bold;
  font-size: 20px;
}
.important.rouge {
  color: red;
}
.important.yellow {
  color: yellow;
}
```

Selecteur descendant

En utilisant un espace : https://developer.mozilla.org/fr/docs/Web/CSS/S%C3%A9lecteurs_descendant

```
header a { /* ... */ }
```

Les `<a>` se trouvant dans un `<header>`

```
<header>
  <a href="">Je serai ciblé</a>
  <p>
    Contenu de paragraphe
    <a href="">Je serai également ciblé</a>
  </p>
</header>
```

Sélecteur enfant

En utilisant le symbole `>` : https://developer.mozilla.org/fr/docs/Web/CSS/S%C3%A9lecteurs_enfant

```
header > a { /* ... */ }
```

Les `<a>` étant **directement** enfants de `<header>`

```
<header>
  <a href="">Je serai ciblé</a>
  <p>
    Contenu de paragraphe
    <a href="">Je ne serai pas ciblé</a>
  </p>
</header>
```

Selecteur voisin direct

En utilisant le symbole `+` : https://developer.mozilla.org/fr/docs/Web/CSS/S%C3%A9lecteur_de_voisin_direct

```
h2 + p { /* ... */ }
```

Un `<p>` étant **directement** voisin de `<h2>`

```
<header>
  <h2>titre</h2>
  <p>paragraphe ciblé</p>
  <p>paragraphe ignoré</p>
  <p>paragraphe ignoré</p>
</header>
```

Sélecteur d'attribut

En utilisant la notation `[attribut]` : https://developer.mozilla.org/fr/docs/Web/CSS/S%C3%A9lecteurs_d_attribut

```
input[type="text"] { /* ... */ }

a[href="http://exemple.com"] { /* ... */ }
```

Combinaisons

En combinant les différents sélecteurs existant il est possible de cibler des éléments très précisément

Dans toutes les `<div>`, trouve les `<p>` avec l'attribut `class` *important*

```
divp.important{ /* ... */ }
```

Trouve l'élément possédant l'attribut `id` puis trouve tous ces `<h2>`, à l'intérieur, trouve toutes les `<span>` avec l'attribut `class` *note*

```
#chapeauh2span.note{ /* ... */ }
```

Pseudo-classes

Permet de cibler des éléments en fonction de leur état (survolé, mis en focus, etc...)

```
a {
  color: black;
}
a:hover {
  color: red;
}
```

Lorsqu'un lien sera survolé, son texte deviendra rouge

> Pour en apprendre plus sur les Pseudo-classes, cette page présente d'autres exemples ou bien MDN propose une liste exhaustive des pseudo-classes disponibles

Héritage

Le principe est simple, la plupart des propriétés relatives au **texte**, passeront d'un élément parent à ses enfants

- font-size
- font-weight
- font-family
- line-height

```
body {
  font-size: 16px;
}
/* les `p` hériterons de la valeur de `font-size` car ils sont les enfants de `body` */
p {}
```

Priorité ou 'c'est moi le plus fort'

Tous les sélecteurs n'ont pas la même force

```
<p id="chapeau" class="important">Lorem ipsum dolor sit amet</p>
```

```
#chapeau { color: blue; }
.important { color: red; }
p { color: green; }
```

On aurait tendance à penser que la dernière règle l'emporterait mais dans la pratique ici notre paragraphe sera de couleur bleue. `color` vaudra `blue` car bien que `p {}` ou `.important {}` soit déclaré après `#chapeau {}`, ils sont plus 'faibles'.

Donc ici, `#chapeau { color: blue; }` est plus que `.important { color: red; }` qui lui même est plus fort que `p { color: green; }`

> id > class > <balise>

Unités

Couleurs

Afin d'exprimer une couleur en CSS il existe plusieurs solutions

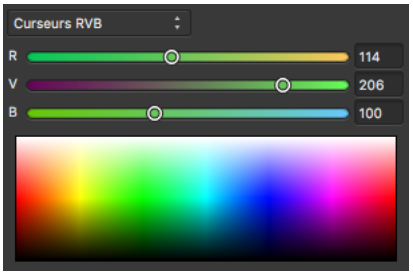
Couleurs nommées

- `red`
- `blue`
- `black`
- `white`
- ...

La liste fournie par MDN contient plus d'une centaine de références; à chaque mot-clé correspond une couleur avec une valeur RVB associée.

RGB & RGBa

RGB ou Red Green Blue



256 valeurs possible dans le Rouge, le Vert et le Bleu soit la bagatelle de **16 777 216** de combinaisons et donc de couleurs possible

En CSS, le RGB s'utilise comme ceci

```
p { color: rgb(114, 206, 100); }

/* Noir */
p { color: rgb(0, 0, 0); }

/* Blanc */
p { color: rgb(255, 255, 255); }

/* Rouge */
p { color: rgb(255, 0, 0); }
```

Il est possible d'exprimer une notion d'alpha (transparence) avec la variante `rgba`. L'alpha se note de 0 à 1.

```
p { color: rgba(114, 206, 100, 0.8); }
```

Hexadécimal

Bien que la notation hexadécimale puisse être déroutante elle est en réalité extrêmement simple à comprendre.

Partant du principe

Base 2**Binaire**	0	1					
Base 10**Décimale**	0	1	2	3	4	5	6
Base 16**Hexadécimale**	0	1	2	3	4	5	6

L'hexadécimal s'exprime avec 16 valeurs possibles et non 10.

Décimale	**Hexadécimale**
0	0
3	3
8	8
13	D
16	10
20	14
30	1E
100	64
150	96
200	C8
255	FF

Même si cette notion n'est pas indispensable, ça peut toujours être utile lors d'un dîner « Absolument, `64 + 64 = C8` ! »

Les couleurs hexadécimales expriment donc du RGB mais sous un autre format

Blanc en RGB `255, 255, 255` en hexadécimal `#FFFFFF`

Toujours pas clair ?

Blanc	**Valeur R**	**Valeur G**	**Valeur B**
RGB	255	255	255

#	FF	FF	FF

Mieux non ?

```
/* Blanc */
p { color: #FFFFFF; }

/* Rouge */
p { color: #FF0000; }

/* Noir */
p { color: #000000; }
```

Évidement, il est inutile de recomposer les couleurs de tête, les couleurs hexadécimales sont très répandues.

La notation hexadécimale permet aussi de gérer la transparence d'une couleur.

🚀 **Pour aller plus loin :** https://developer.mozilla.org/fr/docs/Web/CSS/color_value

---

Tailles

Il est possible d'exprimer les tailles grâce à plusieurs unités

Les unités absolues :

pixels (px)

Cette unité permet de fixer des tailles, des positions ou des espacements par rapport à la résolution de l'affichage.

```
p{
  border-bottom: black solid 2px;
  font-size: 16px;
  margin-top: 20px;
  width: 300px;
}
```

Les unités relatives :

%

Unité relative à la taille de l'élément parent :

```
div{
  width: 300px;
}
divp{
  width: 50%;
```

```
  /* la largeur dépend de la largeur du parent, ici elle fera 150px (50% de 300px)*/
}
```

em

Unité relative à la valeur du `font-size` du parent :

```
body{ font-size: 20px }
h1{ font-size: 2em;} /* 40px (2 fois 20px hérité de body) */
h2{ font-size: 1.5em;} /* 30px (1,5 fois 20px hérité de body) */
p{ font-size: 0.75em;} /* 16px (75% de 20px hérité de body) */
```

rem

Unités relative à la valeur du `font-size` de l'élément racine (le plus souvent, la balise html) :

```
html{ font-size: 10px; }
body{ font-size: 2rem } /* 20px (2 fois 10px hérité de html) */
h1{ font-size: 2rem;} /* 20px (2 fois 10px hérité de html) */
h2{ font-size: 1.5rem;} /* 15px (1,5 fois 10px hérité de html) */
p{ font-size: 0.75rem;} /* 7.5px (75% de 10px hérité de html) */
```

Gestion des textes

font-family MDN

La typo à utiliser

`body { font-family: serif; }`

Classification

Les noms de typos contenant des espaces doivent être notées entre `quotes`

5 groupes de typos existent dans lesquels sont répartis les différentes typos

On parle de typos sécurisées quand elles sont installées par défaut sur n'importe quel système d'exploitation

Groupe	Typos sécurisées	Typos fréquentes
serif	Georgia, « Times New Roman »	« Lucida Bright », « Lucida Fax », « Palatino »
sans-serif	Arial, « Arial Black », Verdana, Impact, « Trebuchet MS »	« Open Sans », « Fira Sans », « Lucida Sans »
monospace	« Courier New »	« Fira Mono », « DejaVu Sans Mono », Menlo
cursive		« Brush Script MT », « Brush Script Std », « Lucida Calligraphy »
fantasy		Papyrus, Herculanum, « Party LET »

Déclaration

Il convient de terminer une déclaration `font-family` par le groupe de typos auquel appartient la typo demandée, si elle n'est pas utilisable l'affichage restera tout de même cohérent

```
body { font-family: "Georgia", serif; }

body { font-family: "Open sans", "Helvetica Neue", Arial, sans-serif; }
```

font-size MDN

Taille de la typo

```
body { font-size: 16px; }
/*
Valeurs possibles  :
  ...px
  ...%
  ...em
  ...rem
*/
```

font-weight MDN

Graisse de la typo

```
body { font-weight: normal; }
/*
Valeurs possibles :
  normal | par défaut
  bold

  100 / Thin
  200 / Extra Light
  300 / Light
  400 / Normal
  500 / Medium
  600 / Semi Bold
  700 / Bold
  800 / Extra Bold
  900 / Ultra Bold
*/
```

font-style MDN

Mise en italique

```
body { font-style: italic; }
/*
Valeurs possibles :
  normal | par défaut
  italic
  oblique
*/
```

font-variant MDN

Petites capitales

```
body { font-variant: normal; }
/*
```

```
Valeurs possibles :
  normal | par défaut
  small-caps
*/
```

line-height MDN

Interlignage

```
body { line-height: 16px; }
/*
Valeurs possibles  :
  normal | par défaut
  valeur numérique sans unité = multiplicateur
  ...px
  ...%
  ...em
  ...rem
*/
```

letter-spacing MDN

Interlettrage

```
body { letter-spacing: normal; }
/*
Valeurs possibles  :
  normal | par défaut
  ...px
  ...em
  ...rem
*/
```

text-align MDN

Alignement du texte

```
body { text-align: left; }
/*
Valeurs possibles :
  left | par défaut
  center
  right
  justify
*/
```

text-decoration MDN

Soulignement du texte

```
body { text-decoration: none; }
/*
Valeurs possibles :
```

```
  none
  underline
  overline
  line-through
  underline overline
*/
```

text-transform MDN

Majuscules / Minuscules

```
body { text-transform: none; }
/*
Valeurs possibles :
  none
  capitalize
  uppercase
  lowercase
*/
```

text-shadow MDN

Ombre du texte

```
body { text-shadow: red 0 2px 5px; }
/*
Valeurs possibles :
  none
  <color> <x> <y> <blur-radius>
*/
```

CSS LevelUp

Un réflexe à adopter très rapidement est de s'assurer de la compatibilité d'une règle ou d'une pratique en se rendant sur Caniuse

Il est possible que certaines règles ne soient pas supportées par certains navigateurs ou que leur usage nécessite l'adoption de préfixe propre à chaque navigateur

Pseudo-classes MDN

La syntaxe comporte `:`

`selector:pseudo-classe{}`

`:hover`

Agira au survol de l'élément ciblé

```
<p class="chapeau">Lorem ipsum dolor <span>sit</span> amet</p>
```

```
.chapeau:hover span{
  background: #EAEAEA;
```

```
}
```

Au survol de `.chapeau` le `<span>` enfant changera de couleur de fond

`:focus`

Agira au focus de l'élément (quand le curseur se place dans un champ par exemple)

```
<input class="adresse" value="Fond vert si sélectionné">
```

```
.adresse:focus{
  background: green;
}
```

Au 'focus' du champ de formulaire `.adresse` sa couleur de fond devient verte

`:first-child` & `:last-child`

Cible le premier/dernier élément de son niveau

```
<ul>
  <li>Un élément</li>
  <li>Un autre élément</li>
  <li>Et encore un élément</li>
  <li>Ensuite...</li>
  <li>Un dernier élément</li>
</ul>
```

```
li{
  color: blue;
}
li:first-child{
  color: red;
}
li:last-child{
  color: green;
}
```

Le texte des `<li>` est bleu sauf pour le premier qui lui est rouge et le dernier qui est vert.

> Il existe beaucoup de pseudo-classes, elles sont toutes listées sur MDN

- Exemples sur CodePen (code + résultat) d'usage complexe des pseudo-classes utilisant `child`.

Pseudo-élément MDN

La syntaxe comporte `::`

`selector::pseudo-element {}`

`::before` & `::after`

Idéal pour la décoration, ces éléments peuvent être facilement manipulés sans impacter le selecteur princip

```
<p class="chapeau">{::before agira ici} Lorem ipsum dolor sit amet {::after agira ici}</p>
```

```css
.chapeau::before {
  content: 'Début de la phrase : ';
  color: red;
}

.chapeau::after {
  content: ' La phrase est finie';
  color: blue;
}
```

`::first-letter` MDN

Idéal pour traiter une lettrine

```css
.chapeau::first-letter {
  color: red;
  font-size: 130%;
}
```

Transitions

Transitions CSS & Utiliser les transitions CSS sur le MDN

```css
div {
  /* Ce que l'on souhaite animer width, background, color, ... \ `all` par défaut*/
  transition-property: all;

  /* Quelle durée pour l'animation 0.3s, 500ms, 2s ... */
  transition-duration: 0.8s;

  /*  Mode d'animation ease, linear, ease-in, ease-out, ease-in-out | `ease` par défaut */
  transition-timing-function: ease;

  /* Durée de décalage du début de la transition 0.3s, 200ms, 1s ... */
  transition-delay: 0;
}

/* méthode courte */
div {
  transition: <property> <duration> <timing-function> <delay>;
}
```

Au survol d'un lien la transition de `background` et de `color` prendra 1 seconde

```css
a{
  background: black;
  color: white;
  transition: 1s;
}

a:hover{
  background: yellow;
```

```
  color: red;
}
```

Animations

Animations CSS & Utiliser les animations CSS sur le MDN

Partant du postulat suivant

`<div class="change-fond">Je suis changeant</div>`

```
.change-fond{
  color: white;
  font-family: monospace;
  width: 300px;
  padding: 20px 0;
  text-align: center;
  margin: 50px auto;
}
```

Nous allons animer la couleur de fond de notre `<div>`

```
.change-fond{
  /* ... contenu précédent ... */

  /* L'identifiant de l'animation déclaré dans @keyframes ... */
  animation-name: changecouleur;

  /* La durée de l'animation 0.3s, 500ms, 5s ... */
  animation-duration: 8s;

  /* Mode d'animation ease, linear, ease-in, ease-out, ease-in-out | `ease` par défaut */
  animation-timing-function: linear;

  /* Durée de décalage du début de l'animation 0.3s, 200ms, 1s ... */
  animation-delay: 0s;

  /* Nombre de fois que l'animation sera jouée 3, 10, infinite  | `1` par défaut */
  animation-iteration-count: infinite;

  /* Sens de l'animation
    normal : démarre à 0% fini à 100% (par défaut)
    reverse : démarre à 100% fini à 0%
    alternate : démarre à 0% va à 100% revient à 0% repart à 100% ...
    alternate  reverse : démarre à 100% va à 0% revient à 100% repart à 0% ...
  */
  animation-direction: alternate;

  /* Quels styles sont appliqués avant le début de l'animation et après la fin de l'animat
ion
    none : aucun style appliqué (par défaut)
    backwards : Les styles du point de départ sont appliqués avant l'animation
    forwards : Les styles du point d'arrêt sont appliqués après l'animation
    both : Les styles sont appliqués au point de départ et au point d'arrêt
  */
  animation-fill-mode: none;

  /* Lecture ou pause d'une animation
```

```
    running : lecture de l'animation (par défaut)
    paused : pause dans l'animation
  */
  animation-play-state: running;
}

/* méthode courte */
.change-fond{
  animation: <name> <duration> <timing-function> <delay> <iteration-count> <direction> <fi
ll-mode> <play-state>;
}

@keyframes changecouleur {
  0% {
    background: #234567;
  }
  50% {
    background: #789654;
  }
  100% {
    background: #987654;
  }
}
```

Pour aider à la création d'animations complexes, des outils comme Bounce.js peuvent être utilisés.

Animate.css, une collection d'animations CSS peut également être utile

Transformations

Le guide d'utilisation des transformations CSS du MDN est très simple et permet de comprendre rapidement le fonctionnement des transformations en CSS.

En passant par la propriété `transform`

```
p {
    transform: translate(0, 20);
}
```

translate

`translate` peut prendre 1 ou 2 arguments

- 1 argument : la valeur du déplacement sur `x`
- 2 arguments : le 1er pour la valeur du déplacement sur `x` le 2eme pour `y`

```
/* déplacement de 240px sur l'axe `x` et de 100px sur `y` */
transform: translate(240px, 100px);
```

Il existe les variantes `translateX()` et `translateY()` pour ne spécifier que la valeur de `x` ou de `y`

rotate

`rotate()` prend 1 argument pour spécifier l'angle de rotation le plus exprimé en degrés `deg` mais d'autres possibilités existent)

```
/* rotation de 30 degrées */
```

```
transform: rotate(30deg);
```

scale

`scale` peut prendre 1 ou 2 arguments

- 1 argument : proportion de redimensionnement uniforme pour la largeur et la hauteur
- 2 arguments : le 1er, redimensionnement de la largeur, le 2eme, de la hauteur

```
/* 2 fois plus petit */
transform: scale(0.5);

/* taille normale */
transform: scale(1);

/* 2 fois plus grand */
transform: scale(2);

/* symétrie, effet miroir */
transform: scale(-1);
```

Il existe les variantes `scaleX()` et `scaleY()` pour ne spécifier que le redimensionnement horizontal ou vertical

skew

`skew` peut prendre 1 ou 2 arguments

- 1 argument : angle de déformation horizontale
- 2 arguments : le 1er, déformation horizontale, le 2eme, verticale

l'angle de déformation est exprimé le plus souvent en degrés mais d'autres possibilités existent

```
/* pas de déformation */
transform: skew(0deg);}

/* déformation de 45 degrées */
transform: skew(45deg);

/* équivalent à -60deg */
transform: skew(120deg);
```

Timing

Il est possible de définir soi-même la temporisation (`transition-timing-function` , `animation-timing-function`) : https://developer.mozilla.org/fr/docs/Web/CSS/transition-timing-function

Un outil très pratique permet d'aider dans la création `cubic-bezier()` : http://cubic-bezier.com/

Layout

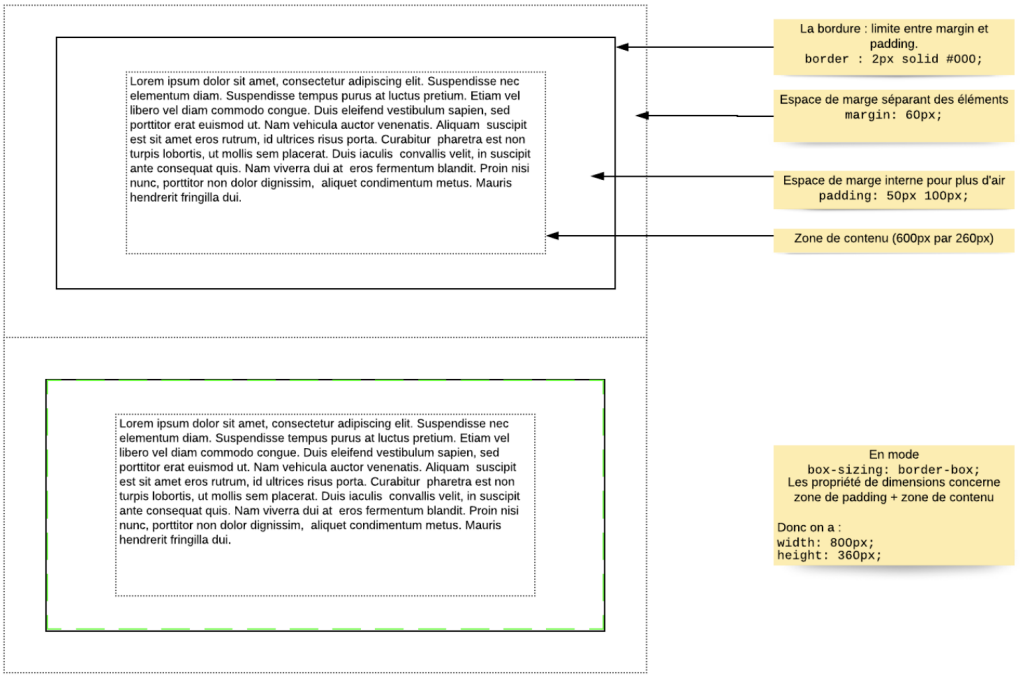
`Layout` signifie `Disposition`. On parle donc ici des différentes méthodes pour « placer » des éléments dans une page HTML.

Box model

> Un élément HTML, c'est comme un oignon : ça a des couches.

On distingue **4 espaces principaux** dans un élément HTML :

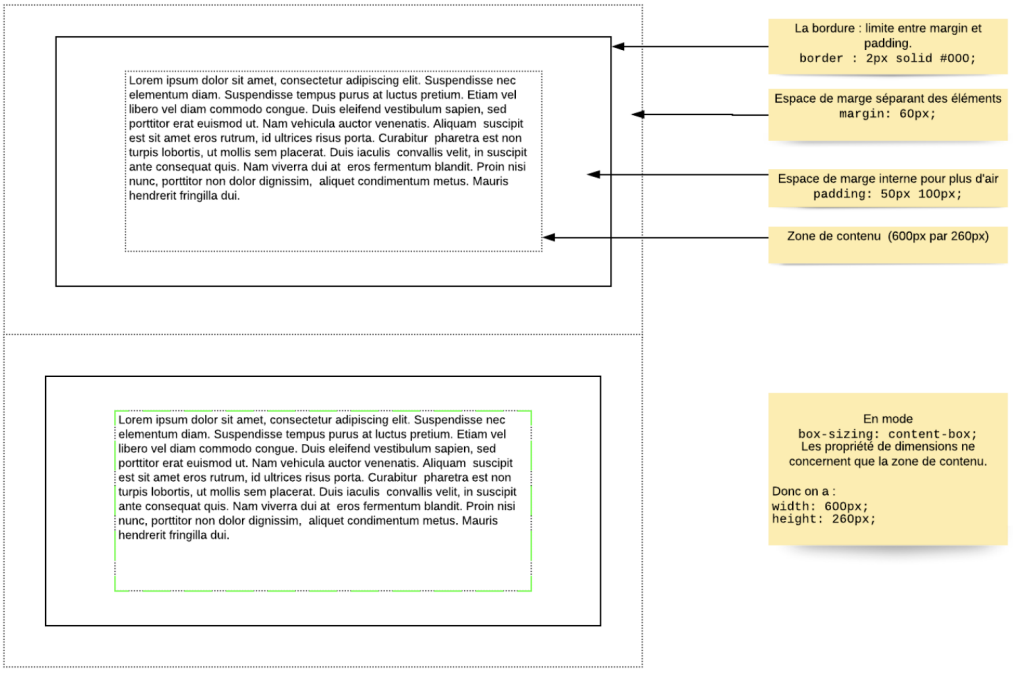
-



Box sizing

Un élément HTML possède 2 tailles : une largeur (`width`), et une hauteur (`height`). Ces tailles peuvent être fixées à la main, ou calculées par le navigateur.

Mais on peut aussi définir quelle est la zone qui sera prise en compte pour appliquer ou calculer ces tailles, en utilisant la propriété box-sizing



Aller plus loin

Ressources pour comprendre la mise en page en CSS

- MDN : Introduction au positionnement CSS
- LearnLayout : Apprendre les mises en page CSS
- MDN : Le modèle de boite
- CSS Reference.io : Le modèle de boite
- CSS Reference.io : Le positionnement
- MDN : Disposition multi-colonnes
- MDN : Utilisation des flexbox

StyleLint

L'idée est d'avoir une vérification automatique de la syntaxe de notre code CSS. Cette vérification s'appuie sur un ensemble de règles d'écriture choisies librement et qui peuvent donc varier d'un projet à l'autre.

Selon les règles d'écriture, on peut retrouver certains des avantages suivants :

- ***uniformisation de l'écriture*** du code (exemple : toujours indenter avec 2 espaces)
- ***éviter le code inutile*** en signalant les règles CSS vides sans aucune propriété associée (exemple : `form { }`)
- ***optimiser*** l'écriture du code (exemple : si on écrit `color: #FFFFFF;` le linter pourrait nous inciter à écrire `color: #FFF;`), celapeut permettre de réduire légèrement la taille du fichier CSS)
- ***travail en équipe :*** il sera plus facile de travailler à plusieurs si chacun doit respecter les mêmes règles d'écriture

Fichier `.stylelintrc`

Le fichier caché `.stylelintrc` va contenir un ensemble de règles à respecter dans l'écriture de tout le code CSS de notre projet.

Pour qu'il soit pris en compte, il faut placer ce fichier à la racine du projet.

Ensuite, c'est grâce à une extension comme linter-stylelint qu'on verra dans VSCode les lignes de code CSS qui ne respectent pas les règles définies dans le fichier `.stylelintrc` avec un message d'erreur nous indiquant le problème rencontré. Cela nous incite alors à corriger et à uniformiser notre code CSS.

Exemples de configuration

Il est possible d'ajouter une multitude de règles, le site de stylelint fourni une documentation très claire et des exemples complémentaires. On peut également directement repartir des exemples ci-dessous.

Exemple simple

Pour tester, c'est toujours bien de commencer petit. Juste pour voir si ça fonctionne bien 😃

```
{
  "rules": {
    "block-no-empty": true,
    "color-hex-case": "upper",
    "color-hex-length": "short",
    "color-no-invalid-hex": true,
    "indentation": 2
  }
}
```

Chaque règle joue un rôle spécifique :

- `"block-no-empty": true` signale toute règle CSS vide (sans aucune propriété associée) car cela correspond à du code inutile
- `"color-hex-case": "upper"` les couleurs hexadécimales doivent être écrites en majuscules `#F9EA26`
- `"color-hex-length": "short"` les codes hexadécimaux doivent utiliser le format court quand cela est possible
- `"color-no-invalid-hex": true` les codes hexadécimaux doivent être valides
- `"indentation": 2` l'indentation doit utiliser 2 espaces

Exemple complet

OK, une fois qu'on a compris le principe, on peut sortir le bazooka.

Voici le `.stylelintrc` étant défini comme le standard actuel :

```
{
  "rules": {
    "at-rule-empty-line-before": [ "always", {
      "except": [
        "blockless-after-same-name-blockless",
        "first-nested"
      ],
      "ignore": ["after-comment"]
    } ],
    "at-rule-name-case": "lower",
    "at-rule-name-space-after": "always-single-line",
    "at-rule-semicolon-newline-after": "always",
    "block-closing-brace-empty-line-before": "never",
    "block-closing-brace-newline-after": "always",
    "block-closing-brace-newline-before": "always-multi-line",
    "block-closing-brace-space-before": "always-single-line",
    "block-no-empty": true,
    "block-opening-brace-newline-after": "always-multi-line",
```

```
    "block-opening-brace-space-after": "always-single-line",
    "block-opening-brace-space-before": "always",
    "color-hex-case": "lower",
    "color-hex-length": "short",
    "color-no-invalid-hex": true,
    "comment-empty-line-before": [ "always", {
      "except": ["first-nested"],
      "ignore": ["stylelint-commands"]
    } ],
    "comment-no-empty": true,
    "comment-whitespace-inside": "always",
    "custom-property-empty-line-before": [ "always", {
      "except": [
        "after-custom-property",
        "first-nested"
      ],
      "ignore": [
        "after-comment",
        "inside-single-line-block"
      ]
    } ],
    "declaration-bang-space-after": "never",
    "declaration-bang-space-before": "always",
    "declaration-block-no-duplicate-properties": [ true, {
      "ignore": ["consecutive-duplicates-with-different-values"]
    } ],
    "declaration-block-no-redundant-longhand-properties": true,
    "declaration-block-no-shorthand-property-overrides": true,
    "declaration-block-semicolon-newline-after": "always-multi-line",
    "declaration-block-semicolon-space-after": "always-single-line",
    "declaration-block-semicolon-space-before": "never",
    "declaration-block-single-line-max-declarations": 1,
    "declaration-block-trailing-semicolon": "always",
    "declaration-colon-newline-after": "always-multi-line",
    "declaration-colon-space-after": "always-single-line",
    "declaration-colon-space-before": "never",
    "declaration-empty-line-before": [ "always", {
      "except": [
        "after-declaration",
        "first-nested"
      ],
      "ignore": [
        "after-comment",
        "inside-single-line-block"
      ]
    } ],
    "font-family-no-duplicate-names": true,
    "function-calc-no-unspaced-operator": true,
    "function-comma-newline-after": "always-multi-line",
    "function-comma-space-after": "always-single-line",
    "function-comma-space-before": "never",
    "function-linear-gradient-no-nonstandard-direction": true,
    "function-max-empty-lines": 0,
    "function-name-case": "lower",
    "function-parentheses-newline-inside": "always-multi-line",
    "function-parentheses-space-inside": "never-single-line",
    "function-whitespace-after": "always",
    "indentation": 2,
```

```
    "keyframe-declaration-no-important": true,
    "length-zero-no-unit": true,
    "max-empty-lines": 1,
    "media-feature-colon-space-after": "always",
    "media-feature-colon-space-before": "never",
    "media-feature-name-case": "lower",
    "media-feature-name-no-unknown": true,
    "media-feature-parentheses-space-inside": "never",
    "media-feature-range-operator-space-after": "always",
    "media-feature-range-operator-space-before": "always",
    "media-query-list-comma-newline-after": "always-multi-line",
    "media-query-list-comma-space-after": "always-single-line",
    "media-query-list-comma-space-before": "never",
    "no-empty-source": true,
    "no-eol-whitespace": true,
    "no-extra-semicolons": true,
    "no-invalid-double-slash-comments": true,
    "no-missing-end-of-source-newline": true,
    "number-leading-zero": "always",
    "number-no-trailing-zeros": true,
    "property-case": "lower",
    "property-no-unknown": true,
    "rule-empty-line-before": [ "always-multi-line", {
      "except": ["first-nested"],
      "ignore": ["after-comment"]
    } ],
    "selector-attribute-brackets-space-inside": "never",
    "selector-attribute-operator-space-after": "never",
    "selector-attribute-operator-space-before": "never",
    "selector-combinator-space-after": "always",
    "selector-combinator-space-before": "always",
    "selector-descendant-combinator-no-non-space": true,
    "selector-list-comma-newline-after": "always",
    "selector-list-comma-space-before": "never",
    "selector-max-empty-lines": 0,
    "selector-pseudo-class-case": "lower",
    "selector-pseudo-class-no-unknown": true,
    "selector-pseudo-class-parentheses-space-inside": "never",
    "selector-pseudo-element-case": "lower",
    "selector-pseudo-element-colon-notation": "double",
    "selector-pseudo-element-no-unknown": true,
    "selector-type-case": "lower",
    "selector-type-no-unknown": true,
    "shorthand-property-no-redundant-values": true,
    "string-no-newline": true,
    "unit-case": "lower",
    "unit-no-unknown": true,
    "value-list-comma-newline-after": "always-multi-line",
    "value-list-comma-space-after": "always-single-line",
    "value-list-comma-space-before": "never",
    "value-list-max-empty-lines": 0
  }
}
```

Responsive Web-Design

RWD en 3 étapes

RWD ou **R**esponsive **W**eb **D**esign : du Webdesign (**UI – User Interface**) qui s'adapte à la taille et aux spécificités des périphériques.

- ☑ ~~[x] *Device* – Prendre en compte le périphérique~~
- ☑ ~~[x] *Mobile First* – Concevoir son application pour le mobile avant-tout~~
- ☑ ~~[x] *Breakpoints* – Définir des « points de décrochage »~~

1. *Device* – Prendre en compte le périphérique

Simplement avec un élément HTML `<meta name="viewport">` en spécifiant 3 informations

- `width=device-width` on se base sur la largeur du device
- `initial-scale=1` zoom réglé à 1
- `user-scalable=no` L'utilisateur ne peut pas zoomer (optionnel, en fonction des cas)

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

2. *Mobile First* – Concevoir son application pour le mobile avant-tout

L'objectif est d'intégrer avant-tout pour la plus petite taille d'écran. Voici quelques outils utiles

Outils

- `width` sur MDN
- `height` sur MDN
- Unités sur MDN
- Flexbox

min & max

Permet de définir une largeur (ou hauteur) minimun ou maximun qu'un élément pourra avoir.

> En gardant à l'esprit que
>
> `min-width` > `max-width` > `width`
>
> `min-height` > `max-height` > `height`

em & %

Le calcul `cible / contexte = résultat` est très pratique pour adapté du fixe vers du fluide.

Exemple

Pour afficher un titre d'une taille de `24px`, tout en sachant que `<body>` est configuré avec une `font-size` de `16px`

> 24 / 16 = 1.5

Notre titre pourra donc être déclaré comme ceci

```
h1 {
    font-size: 1.5em;
}
```

vh & vw

Donner la hauteur de la fenêtre à un élément est parfois bien compliqué.

L'unité `vh` s'appuie sur le `viewport`

- `vh` 1/100e de la hauteur du viewport.
- `vw` 1/100e de la largeur du viewport.
- `vmin` 1/100e de la valeur minimale entre la hauteur et la largeur du viewport.
- `vmax` 1/100e de la valeur maximale entre la hauteur et la largeur du viewport.

```
section {
    height: 100vh;
}
```

3. *Breakpoints* – Définir des points de décrochage

Afin de respecter l'approche **Mobile First**, le mode de conception va du plus petit au plus grand. La principale caractéristique du CSS (Cascading) va rendre la tâche relativement simple.

Structure d'une `Media Query`

@media sur MDN

```
@media <media-query-list> {
    /* Règles CSS appliquées si la requête est vérifiée */
}
```

Types de média

Type	Description
`all`	Peut être appliqué quel que soit l'appareil.
`print`	Destiné aux œuvres paginés et aux documents qui sont vus sur des écrans pour l'aperçu avant impression.
`screen`	Destiné à tous les écrans.
`speech`	Destiné aux synthétiseurs vocaux.

Principales caractéristiques de média

Caractéristique	Description	Notes
`width`	Largeur de la zone d'affichage (*viewport*).	`min-...`, `max-...`
`height`	Hauteur de la zone d'affichage (*viewport*).	`min-...`, `max-...`
`aspect-ratio`	Proportion de la largeur sur la hauteur pour la zone d'affichage (*viewport*).	`min-...`, `max-...`
`orientation`	Orientation de la zone d'affichage (*viewport*).	`portrait` ou `landscape`
`resolution`	Densité de pixels pour l'appareil d'affichage	`min-...`, `max-...`

Et dans un avenir proche

Caractéristique	Description	Notes
`light-level`	Le niveau de lumière ambiante actuel.	Ajouté avec la spécification sur les requêtes média de niveau 4
`scripting`	La manipulation via les scripts (ex. JavaScript) est-elle disponible ?	Ajouté avec la spécification sur les requêtes média de niveau 4

Opérateurs

`and`

Combinaison de caractéristiques

```
@media (min-width: 700px) and (orientation: portrait) {
    /* … */
```

```
}
```

Les styles seront appliqués si le périphérique a au moins une largeur de 700px et est en orientation portrait.

`,`

Association de plusieurs requêtes

```
@media (orientation: portrait), screen and (max-width: 500px) {
    /* … */
}
```

Les styles seront appliqués pour tous les types de périphériques en orientation portrait et aux périphériques `screen` avec au maximum une largeur de 500px.

D'autres opérateurs logiques sont disponibles comme `only`, le détail est disponible sur le MDN

Breakpoints types

```
/* Mobile First CSS */

/* Mobile + */
@media (min-width: 400px) { /* ... */ }

/* Phablets */
@media (min-width: 550px) { /* ... */ }

/* Tablettes */
@media (min-width: 750px) { /* ... */ }

/* Desktop */
@media (min-width: 1000px) { /* ... */ }

/* Desktop HD */
@media (min-width: 1200px) { /* ... */ }

/* Desktop Wide */
@media (min-width: 1400px) { /* ... */ }
```

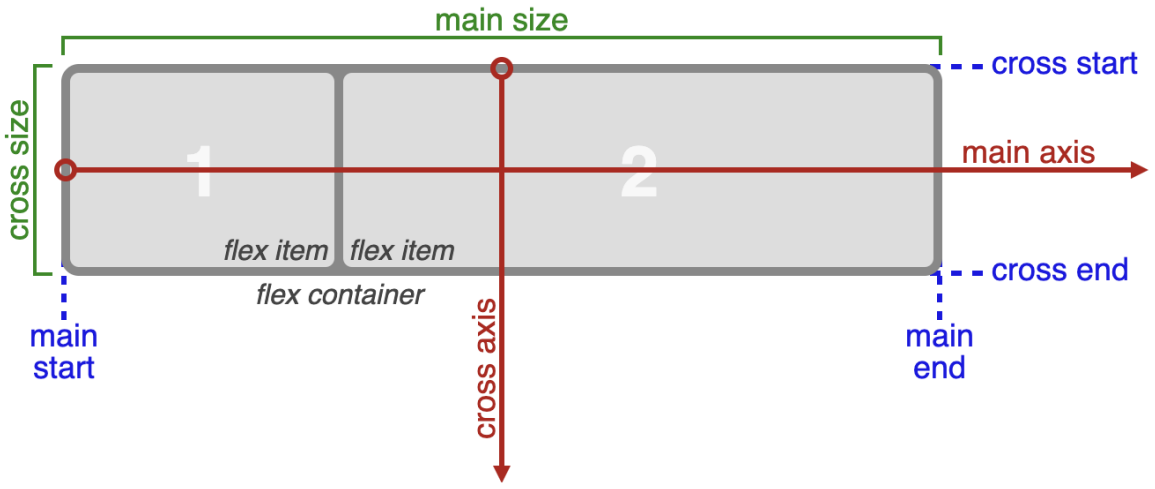
C'est alléchant de pouvoir cibler précisément certains appareils en fonctions de leurs caractéristiques, pour autant il vaut mieux ne pas tomber dans ce genre de piège et rester sur des « fourchettes » globales; qui sait quelle taille d'écran sortira demain…

Flexbox

- Flexbox sur W3C
- Flexbox sur MDN
- Flexbox playground
- CSS-tricks flexbox
- Guide flexbox

`flexbox` pour Flexible box, est une manière très puissante de réaliser une mise en page.

Avant tout, la connaissance de la terminologie est très utile pour saisir quelques subtilités.



flex container

L'élément parent dans lequel chaque élément flex sera contenu. Un conteneur flex est défini lorsque la propriété `display` possède la valeur flex ou inline-flex.

flex item

Chaque enfant d'un conteneur flex devient un élément flex. Le texte directement contenu dans un conteneur flex est englobé dans un élément flex anonyme.

Axes

Toute boîte suit deux axes : L'axe principal (main axis) sur lequel les éléments flex se suivent. L'axe secondaire (cross axis) est perpendiculaire à l'axe principal.

- La propriété `flex-direction` établit l'axe principal.
- La propriété `justify-content` définit comment les éléments flex sont positionnés le long de l'axe principal sur la ligne courante.
- La propriété `align-items` définit comment les éléments flex sont positionnés le long de l'axe secondaire sur la ligne courante.
- La propriété `align-self` définit comment un seul élément flex est aligné sur l'axe secondaire et surcharge le comportement par défaut défini par `align-items`.

Directions

Le début/fin du côté principal et du côté secondaire du conteneur flex décrit l'origine et la fin du flux d'éléments flex. Ils suivent l'axe principal et secondaire du conteneur flex dans le sens établi par writing-mode (gauche-vers-droite, droite-vers-gauche, etc.).

- La propriété `order` ordonne les éléments d'un groupe et détermine quel élément va apparaitre en premier.
- La propriété `flex-flow` raccourcis les propriétés `flex-direction` et `flex-wrap` pour positionner les éléments flex.

Lines

Les éléments flex peuvent être positionnés soit sur une seule ligne, soit sur plusieurs lignes via la propriété `flex-wrap`, qui contrôle la direction de l'axe secondaire et la direction dans chaque nouvelle lignes rajoutées.

Dimensions

Les termes désignant la hauteur et la largeur des éléments flex sont la taille principale (main size) et la taille secondaire (cross size), qui suivent respectivement l'axe principal et l'axe secondaire du conteneur flex.

- Les propriétés `min-height` et `min-width` ont une valeur initiale de auto.
- La propriété `flex` est un raccourci des propriétés `flex-grow`, `flex-shrink` et `flex-basis` pour établir la flexibilité des éléments flexibles.

> Provenant de MDN

La clé d'un monde merveilleux

Prérequis indispensable

`display: flex;` ou `display: inline-flex;`

La différence entre les 2 est la même que entre `block` et `inline-block`, aucun changement donc sur le comportement de flexbox.

Main dans la main, 2 lots de règles

un 'flex-container' et des 'flex-item' fonctionnent ensemble, le container donne des règles de formatage et les items les respectent. Les items peuvent avoir des règles propres pour surpasser celles du container.

flex-container

le `display: flex` s'applique sur le container

```
.flex-container {
    display: flex; /* la clé ;) */
}
```

dans quelle direction vont les items ?

Basée sur le `main-axis`

`flex-direction`

- `row` * de gauche à droite
- `column` de haut en bas
- `row-reverse` de droite à gauche
- `column-reverse` de bas en haut

Comment les items seront alignés sur l'axe principal ?

`justify-content`

- `flex-start` * les items commencent à `main-start`
- `flex-end` les items commencent à `main-end`
- `center` les items sont centrés sur l'axe principal `main-axis`
- `space-between` les items sont répartis sur l'axe principal depuis `main-start` jusqu'à `main-end`
- `space-around` les items sont répartis sur l'axe principal, espacés de manière égale.

Comment les items seront alignés sur l'axe secondaire `cross-axis` ?

`align-items`

- `stretch` * les items occupent tout l'espace disponible (sauf indication contraire sur un item)
- `flex-start` les items commencent à `cross-start`
- `flex-end` les items commencent à `cross-end`
- `center` les items sont centrés sur l'axe secondaire `cross-axis`
- `baseline` les items sont alignés sur la même baseline mdn

Les items tiennent sur une seule ligne ou sur plusieurs lignes ?

`flex-wrap`

- `nowrap` * les items tiennent sur une seule ligne
- `wrap` les items seront rangés sur plusieurs lignes de haut en bas
- `wrap-reverse` les items seront rangés sur plusieurs lignes de bas en haut

Quand les items sont sur plusieurs lignes, comment aligner ce bloc de lignes globalement ?

`align-content` se comporte de la même façon que `justify-content` mais sur l'axe secondaire; *ne s'applique que sur plusieurs lignes d'items*

`align-content`

- `stretch` * les items sont réparties pour occuper l'espace
- `flex-start` les lignes sont regroupées à `cross-start`
- `flex-end` les lignes sont regroupées à `cross-end`
- `center` les lignes sont regroupées au centre du container
- `space-between` les lignes sont réparties sur l'axe secondaire depuis `cross-start` jusqu'à `cross-end`
- `space-around` les lignes sont réparties sur l'axe secondaire, espacées de manière égale.

flex-item

```
.flex-item { /* ... */ }
```

un item peut se ré-ordonner par rapport aux autres ?

`order`

0 représente la position par défaut, tous les nombres plus grands déplaceront l'item vers la fin, les plus petits vers le débuts.

un item peut être aligné différemment que les autres ?

`align-self`

écrase l'alignement par défaut fourni par `align-items`

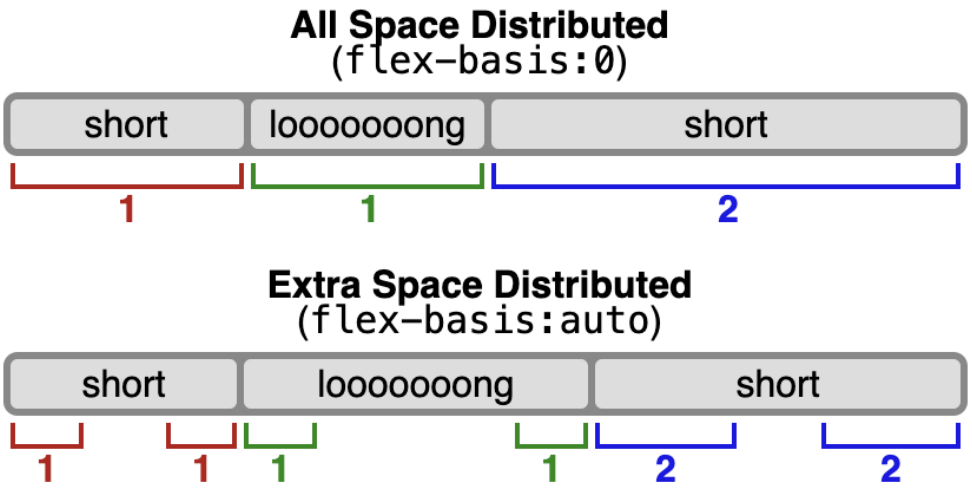
`align-self: auto | flex-start | flex-end | center | baseline | stretch;`

Un item peut avoir une taille différente des autres ?

`flex-basis`

`auto` * flex utilisera `width` ou `height`

valeur – l'item prendra cette valeur par défaut comme taille mais avant l'attribution d'espace fait par flex



un item peut occuper plus ou moins d'espace que les autres ?

Démo explicative

`flex-grow`

si tous les items possèdent une `flex-grow` de 0, un item possédant un `flex-grow` de 1 occupera le maximum d'espace par rapport aux autres, donc si plusieurs items possède le même `flex-grow` ils s'entendront pour se repartir l'espace.

`flex-shrink`

Facteur de rétrécissement autorisé

CSS orientée-objet (OOCSS)

> Écrire de la CSS, c'est facile. Écrire de la CSS qui tienne la route, et dans le temps, c'est autre chose !

Il y aurait comme une sorte de malédiction planant sur CSS. Impossible d'avoir un corpus CSS un tant soit peu volumineux qui ne devienne pas un sac de nœuds. Un secret bien gardé (ou peu connnu ?) dans l'univers CSS est qu'il ne faut pas *écrire* de la CSS, il faut la *construire*. Avec des règles et tout et tout 💪

Un problème de spécificité

Pour comprendre pourquoi on se retrouve très facilement avec de la CSS spaghettis, étudions attentivement l'exemple suivant. On se place dans le cas où on cherche à modifier le style du titre d'un « module », une boîte de contenu texte, pour le mettre en avant. Par exemple parce que le module transmet une information urgente.

Le premier essai est simplement d'ajouter une classe sur le titre :

```
# La modification.

 <div class="some-module">
-  <h2>
+  <h2 class="urgent">
     Special Header
   </h2>
 </div>
```

```
<!-- Le résultat. -->

<div class="some-module">
  <h2 class="urgent">
    Special Header
  </h2>
</div>
```

Mais voilà que les ennuis commencent :

```
/* La CSS associée. */

.some-module > h2 {
  /* Styles par défaut pour le titre d'un module. */
}

.urgent {
  /* Ce style ne s'appliquera en fait *jamais* ! */
}

.some-module .urgent {
  /* Ah, celui-là s'applique, cool. */
}
```

Ce code CSS à deux problèmes principaux, qui sont liés au sélecteur `.some-module > h2`.

1. Ce sélecteur définit le style de base, mais il est fortement couplé au DOM.
2. Il possède une spécificité de 11 (voir ce calculateur de spécificité CSS).

Le couplage CSS-HTML est problématique car si le `<h2>` est déplacé, ou remplacé par un `<h3>` ou un `<header>`, il faudra modifier la CSS — alors même que le *rôle* du titre d'un module n'a pas changé. La spécificité haute sur la règle de base est quant elle problématique car pour surcharger ce sélecteur, il faudra se placer soit dans une spécificité de 11 et *après* (donc il faudra faire très attention à l'ordre de chargement), soit dépasser 11 : `.some-module .urgent` (20), `.some-module > h2.urgent` (21) ou encore `#special-case .urgent` (110). La haute spécificité du sélecteur de base « contamine » tout le corpus CSS, et représente une dette technique.

Ces problèmes sont très fréquents et jouent bien souvent de concert à travers des sélecteurs longs qui ont plusieurs niveaux de couplage et une très haute spécificité : `#details-pane #products-wrapper .product > img:first-child`, ce genre de choses. Tout ça devient très pénible à gérer dès lors que le corpus CSS est un peu volumineux et/ou que la spécificité moyenne des sélecteurs est élevée. Comme en général, pour s'en sortir, on ajoute des sélecteurs *encore plus spécifiques et encore plus longs* que ce qui existe déjà, le problème ne fait qu'empirer. On se retrouve alors au mieux à traîner des sélecteurs alambiqués dans le seul but d'essayer de *défaire* ou d'*outrepasser* l'existant ; au pire, plus personne ne veut toucher à la CSS.

Afin de se débarasser de ces problèmes, toutes les métholodogies modernes de CSS recommandent de suivre une règle simple : viser une complexité de la plus basse possible en utilisant un nommage sémantique basé sur des `.classes-sémantiques`. Pourquoi ?

Génèse d'une `.classe-sémantique`

Pour améliorer notre code et pouvoir utiliser cette classe `.urgent` qui à l'air si pratique, on peut penser à se doter d'une classe `.title` pour gérer le style de titre par défaut :

```
- <div class="some-module">
+ <div>
-   <h2 class="urgent">
+   <h2 class="title urgent">
      Special Header
    </h2>
  </div>
```

```
<div>
  <h2 class="title urgent">
    Special Header
  </h2>
</div>
```

```
.title {
  /* Styles par défaut (optionel). */
}

.urgent {
  /* Styles pour un titre urgent. */
}
```

À ce stade, le couplage au DOM a bel et bien été éradiqué. Contrat rempli ?

Et bien, certes, il est beaucoup plus simple et rapide de faire des modifications ; mais les classes `.title` et `.urgent` sont désormais tellement génériques — on dit « réutilisables » pour être dans le coup — que dans la pratique, elles vont encourager la composition de « micro-classes » (`title urgent`, puis `title urgent big`, puis…) pour arriver à produire le « macro-état » (le style final désiré). Cela va mécaniquement augmenter la spécificité des sélecteurs. Pas vraiment une solution satisfaisante à moyen terme.

Une classe *wrapper* semble alors tout indiquée :

```
- <div>
+ <div class="module">
   <h2 class="title urgent">
     Special Header
   </h2>
 </div>
```

```
<div class="module">
  <h2 class="title urgent">
    Special Header
  </h2>
</div>
```

```
.module .title {
  /* Styles par défaut. */
}

.module .title.urgent {
  /* Styles pour un titre urgent. */
}
```

C'est mieux : le sélecteur `.module .title` permet de comprendre tout de suite à quoi on a affaire, en l'ocurrence au *titre d'un module*. Les micro-classes elles-mêmes ne sont pas vraiment très précises, mais leur combinaison permet de créer du sens. Le problème, c'est que les sélecteurs possèdent maintenant une spécificité de 20 ou plus. Les symptômes *et* la maladie sont toujours les mêmes, on a juste nettoyé la plaie.

Pour *réellement* abaisser la spécificité des sélecteurs, tout en évitant couplage et micro-classes globales, CSS ne laisse d'autre choix que d'intégrer un espace de nom explicite dans les classes elles-même :

```
<div class="module">
-  <h2 class="title urgent">
+  <h2 class="module-title urgent">
    Special Header
  </h2>
 </div>
```

```
<div class="module">
  <h2 class="module-title urgent">
   Special Header
 </h2>
</div>
```

```
.module-title {
  /* Styles par défaut. */
}

.module-title.urgent {
  /* Styles pour un titre urgent. */
}
```

Ou en SASS, syntaxe scss :

```
.module-title {
  /* Styles par défaut. */
```

```
  &.urgent {
    /* Styles pour un titre urgent. */
  }
}
```

Le principal intérêt d'un tel « préfixe », ou *namespace*, réside dans le fait qu'il permet de transférer de CSS vers HTML la gestion de la relation entre des concepts — ici, un module et son titre. Avec un sélecteur comme `.module .title`, la CSS essayait de gérer cette relation parent-enfant en utilisant le seul outil à sa disposition : un sélecteur à spécificité haute (20). Cette technique fonctionne à court terme, mais comme on l'a vu, elle n'est pas scalable à plusieurs titres. Avec le *namespace*, le HTML peut désormais dire à un élément « comporte toi comme un titre de module ! » et coté CSS, la complexité moyenne des sélecteurs à été effectivement réduite : `.module.title` est devenu `.module-title` (20 ⇒ 10) et `.module.title.urgent` est devenu `.module-title.urgent` (30 ⇒ 20). C'est déjà beaucoup mieux !

Mais est-ce *vraiment* mieux ? Notre problème initial était de contrecarrer `.some-module > h2` (11). Au final, aussi bien la toute première solution (`.some-module .urgent`) que la dernière (`.module-title.urgent`) ont une spécificité de 20. Certes, chemin faisant, le style de base est devenu `.module-title` (11 ⇒ 10), mais manifestement quelque chose cloche.

Notion de modificateur d'état

Le problème sous-jacent, c'est que `.urgent` est toujours un concept global qu'on peut rattacher aussi bien à un titre qu'à un paragraphe, un `<div>`, etc. Si la notion d'urgence, et le style qui y est associé, est *effectivement* quelque chose pensé et designé pour être réutilisable, pas de problème : le gain de 10 points de spécificité dans les sélecteurs du type `.foobar.urgent` sera le prix à payer. Mais bien souvent, cette flexibilité et cette réutilisabilité ne sont ni utiles, ni souhaitables. Pourquoi alors ne pas suivre notre logique jusqu'au bout ? Relions la notion d'urgence à la notion de titre d'un module, directement dans la classe CSS :

```
<div class="module">
-  <h2 class="module-title urgent">
+  <h2 class="module-title module-title-urgent">
     Special Header
   </h2>
 </div>
```

```
<div class="module">
  <h2 class="module-title module-title-urgent">
    Special Header
  </h2>
</div>
```

```
.module-title {
  /* Styles par défaut. */
}

.module-title-urgent {
  /* Styles pour un titre urgent. */
}
```

Ou en SASS (les exemples à suivre seront tous en SASS, syntaxe scss) :

```
.module-title {
  /* Styles par défaut. */

  &-urgent {
    /* Styles pour un titre urgent. */
```

```
  }
}
```

Bien que manifestement plus verbeux, `.module-title-urgent` possède deux grandes qualités :

- une spécificité de 10
- une sémantique complète

Sa spécificité basse rend ce sélecteur très facile et fiable à surcharger, et ce depuis n'importe quel endroit du corpus CSS. Pas de dette technique, et on s'affranchit en plus de la notion d'ordre de déclaration / chargement de la CSS. Le coté sémantique tient au fait qu'en lisant `module-title-urgent`, on comprend d'emblée qu'il s'agit d'un *titre*, dans sa variante *urgente*, trouvé *dans un module* et pas ailleurs (ce n'est pas juste `title-urgent`, par exemple). Le sélecteur devient très résilient au changement, on s'affranchit de la dépendance au DOM.

> Le terme « sémantique » fait référence au fait que la classe est à la fois explicite et déclarative (j'y viens ci-après). En anglais, on pourrait dire self-explanatory. Je n'aime en fait pas trop « sémantique », qui est traditionnellement un concept HTML à destination des machines (parser <article> plutôt que , etc.), et préfèrerais parler de fully-qualified selector. Mais au final, peu importe le jargon tant qu'on comprend de quoi on parle !

Utiliser une haute spécificité à bon escient

Cette technique de nommage basé sur des *namespaces* explicites règle les problèmes de couplage et de spécificité mis en évidence au départ. Mais en plus de ça, à partir du moment où la plupart des sélecteurs du corpus CSS ont une spécificité basse, il devient beaucoup plus simple de gérer les cas particuliers. Par exemple, l'utilisation d'un *wrapper* n'est pas du tout proscrite, à condition qu'il exprime l'intention de gérer un tel cas particulier :

```
<!-- J'adopte à partir d'ici une convention consistant à utiliser deux espaces entre les n
oms
     de classes CSS, pour améliorer la lisibilité. -->
<div class="module  module-special">
  <h2 class="module-title">
    Special Header
  </h2>
</div>
```

```
.module-title { … }

.module-special {
  .module-title {
    /* Styles pour un titre de module spécial. */
  }
}
```

Il est particulièrement intéressant de constater que cette architecture à spécificité basse permet de facilement gérer le cas d'un module « spécial » muni d'un titre « urgent », sans augmenter la spécificité des sélecteurs :

```
<div class="module  module-special">
  <h2 class="module-title  module-title-urgent">
    Special Header
  </h2>
</div>
```

```
.module-special {
  .module-title-urgent {
    /* … */
```

```
  }
}
```

Le code CSS requis est très simple, et on *sait* que tout va bien se passer et s'appliquer correctement, car tout a été construit en ce sens.

Selon nos besoins, on pourrait tout aussi bien imaginer relier la notion d'urgence, non pas au titre d'un module, mais au module lui-même :

```
<div class="module  module-urgent">
  <h2 class="module-title">
    Special Header
  </h2>
</div>
```

```
/* Module -- styles de base. */
.module { … }
.module-title { … }

/* Module -- variante Urgente. */
.module-urgent {
  .module-title { … }
}
```

> En résumé : plus on est précis dans son nommage pour les règles de base, moins les sélecteurs sont spécifiques (en valeur numérique), et moins on a de question à se poser et de travail à effectuer.

OOCSS

Si vous avez un peu suivi l'actualité de CSS, tout ça devrait vous rappeler un mot-clé en vogue : OOCSS (*Object Oriented CSS*). On parle en effet beaucoup de cette CSS « orientée objet », malheureusement sans forcément comprendre ce dont il s'agit.

La programmation orientée objet (POO) est un paradigme de programmation où l'unité de base, l'objet, permet de mettre en place une architecture de code basée sur quatre grands piliers :

1. abstraction
2. encapsulation
3. héritage
4. polymorphisme

Des concepts plutôt abscons, qui paraissent bien éloignés de CSS, mais qui sont en fait plutôt simples et qui peuvent, dans une certaine mesure, être appliqués à CSS pour se simplifier la vie au quotidien. Voyons comment, en partant d'un exemple concret qui n'a rien à voir avec CSS.

- On souhaite représenter la notion très générale de « forme géométrique » dans un programme.
- On trouvera des carrés, des cercles, etc.
- Chaque forme doit avoir une couleur.
- Chaque forme doit pouvoir dire combien de coté(s) elle a.
- Chaque forme doit pouvoir calculer son aire.

Dans un langage de POO comme Ruby, cela donnerait :

```
class Shape
  attr_accessor :color
  attr_reader :sides
```

```
  def initialize(color = :transparent)
    @color = color
    @sides = nil
  end
end

class Square < Shape
  attr_accessor :width, :height

  def initialize(width, height, *args)
    @sides = 4
    @width = width
    @height = height
    super(*args)
  end

  def area
    width * height
  end
end

class Circle < Shape
  attr_accessor :radius

  def initialize(radius, *args)
    @sides = 1
    @radius = radius
    super(*args)
  end

  def area
    2 * Math::PI * radius
  end
end
```

Une fois cette implémentation en place, elle nous donne accès à une API, une interface « orientée objet » pour manipuler des formes géométriques :

```
square1 = Square.new(10, 10, :red)
square2 = Square.new(2, 4, :blue)

square1.color # :red
square1.sides # 4
square1.area  # 100

square2.color # :blue
square2.sides # 4
square2.area  # 8

a_circle = Circle.new(5)
a_circle.color # :transparent
a_circle.sides # 1
a_circle.area # 31.42
a_circle.radius=(10)
a_circle.area # 62.83
```

Revisitons les quatres concepts de la POO à la lumière de ce code :

1. Abstraction : il est sans doute possible de donner une définition mathématique du concept de forme géométrique, mais pour nous autres programmeurs chargés de résoudre un problème précis, une forme géométrique (`Shape`) sera quelque chose qui possède une couleur (`@color`), un nombre de cotés (`@sides`) et qui est capable de calculer son aire (`area`). Point à la ligne. Le terme « abstraction » est à prendre dans le sens de « simplification » — simplification du réel, s'entend. On pourrait ainsi se doter d'abstractions `Car`, `User`, `ForumTopic`, etc. selon nos besoins.
2. Encapsulation : `square1` est une « instance » de l'abstraction `Square` ; `square2` également. Ces deux instances possèdent des caractéristiques communes, qu'elles tirent de l'abstraction dont elle sont issues (par exemple, en tant qu'instances de `Square`, aussi bien `square1` que `square2` sont capables de calculer leur aire). Mais elles ont aussi un état propre : `square1` a une hauteur différente de `square2`, etc. Quand on parle d'encapsulation, on parle à la fois d'état interne, et d'interface pour gérer cet état, le tout au niveau d'une instance. L'encapsulation, c'est l'idée que chaque objet, c'est-à-dire chaque instance d'une abstraction, se comporte tel un individu particulier, avec ses propres caractéristiques (son état), et ses propres moyens d'actions (son interface, dont tout ou partie peut-être tirée de l'abstraction à laquelle l'instance appartient). Cette architecture de code est très pratique, car une instance devient alors un petit soldat dans un corps d'armée, tous les objets collaborant pour échanger et modifier des données selon les règles définies par le programmeur, dans le but de résoudre des problèmes. Un objet à un nom (`a_circle`), un état (`@color`, `@radius`, ...), et on peut lui donner des ordres (`a_circle.radius=(10)` ⇒ change ton rayon !) ou lui poser des questions (`a_circle.radius` ⇒ quelle est ton aire ?).
3. Héritage : `Square` et `Circle` sont deux cas particuliers de la notion générale de forme géométrique, `Shape`. À ce titre, ils ont tout aussi bien des traits commun (une couleur...) que des spécificités (un nombre de coté différent, une façon de calculer leur aire différente...). L'héritage permet de « factoriser » ce qui est commun entre plusieurs abstractions. La façon de gérer cet héritage d'état et d'interface dépend du langage. En Ruby, par exemple, l'héritage est géré au niveau des catégories d'objets (`class`), tandis qu'en JavaScript, des objets spécialisés sont utilisés pour en relier d'autres entre-eux (héritage prototypal). CSS aussi possède sa propre logique d'héritage, un mix de *cascade* (le *C* de CSS) et de règles de spécificités.
4. Polymorphisme : quand plusieurs abstractions partagent tout ou partie d'une interface, on parle de polymorphisme — littéralement, plusieurs formes. L'idée est que des objets appartenant à des abstractions différentes (un carré n'est pas un cercle...) sont capables de réagir à des instructions identiques : `a_square.area` et `a_circle.area`. Cette capacité à créer des interfaces qui vont jouer le rôle de « méta-abstraction » est important, car il permet de se découpler de la connaissance fine de ce qu'on manipule. À partir du moment où je sais qu'on me donne une forme géométrique, n'importe laquelle, je serai en mesure de calculer son aire. L'objet peut bien avoir plusieurs formes, être un carré, un cercle ou autre chose encore, peu importe : parce que l'interface est commune, je n'ai plus à me soucier de savoir comment faire, ni comment demander — calculer l'aire, ce sera toujours `.area`.

Ces concepts fondamentaux de POO sont très utiles pour concevoir des programmes complexes, résilients et maintenables. CSS n'a pas de notion d'objet, de classe, d'état ou d'interface, mais en fait, ces mêmes concepts y existent à l'état latent et ne demandent qu'à être structurés.

Le cas Bootstrap

On connaît tous l'impératif, ce mode grammatical qui sert à exprimer une injonction, à donner un ordre : « mange ta soupe ! » Si on veut qu'un ordre soit suivi d'effet, mieux vaut être spécifique, non ?

Bootstrap est une librairie présentée comme OOCSS, qui met à disposition un ensemble de classes CSS qu'on va parfois devoir mixer pour graduellement changer le look 'n feel d'un élément, jusqu'à ce qu'il ressemble à ce qu'on souhaite. Prenons un exemple de code HTML qui utilise Bootstrap :

```
<div class="container">
  <div class="row">
    <section class="content">
      <h1>Table Filter</h1>
      <div class="col-md-8 col-md-offset-2">
        <div class="panel panel-default">
          <div class="panel-body">
            <div class="pull-right">
              <div class="btn-group">
                <button type="button" class="btn btn-success btn-filter" data-target="paga
do">Pagado</button>
                <button type="button" class="btn btn-warning btn-filter" data-target="pend
```

```
iente">Pendiente</button>
                // …
```

Pour faire le lien avec la POO, Bootstrap considère qu'un nœud du DOM (un `<div>`, etc.) fait office d'objet, que son état est le style visuel final désiré, et que l'interface de l'objet correspond à l'ensemble des classes impératives comme `row` ou `pull-right`.

C'est une approche intéressante, notamment parce qu'elle est très simple à prendre en main et permet de prototyper rapidement. Mais cet exemple démontre avant tout pour moi l'absence d'un modèle de pensée cohérent dans Bootstrap, et explique pourquoi Bootstrap représente une grosse dette technique sur bien des projets.

Certaines classes de Bootstrap sont impératives par elles-même (`pull-right`), d'autres le sont par composition transverse (`container`, `row` et `col-*`) ou composition directe (`btn` et `btn btn-success`), tandis qu'enfin d'autres « sonnent » sémantiques mais doivent quand même être associées à des micro-classes impératives généralistes (`panel-body` précisé par `pull-right`). Cette composition de classes dans tous les sens et à plusieurs niveaux d'abstractions phagocyte peu à peu tout le DOM, et représente un terrain ultra-favorable aux problèmes de spécificités CSS. En outre, les tentatives sémantiques comme `panel` ne sont qu'une illusion : un tel `panel` Bootstrap a été conçu pour être réutilisable, donc générique, de sorte que *by design*, il va être nécessaire de faire de la composition de classes et/ou de la surcharge agressive pour personnaliser le design. Bootstrap est efficace et résilient à partir du moment où on l'utilise pour ce qu'il a été pensé : une bibliothèque de styles *figée*. Au-delà, danger.

Plus fondamentablement, Bootstrap essaye d'englober sous une même logique POO deux technologies aux responsabilités fondamentalement orthogonales : HTML (structure) et CSS (présentation). Pour ce faire, il extrait un maximum de chose de CSS pour les exposer dans des micro-classes impératives, ce qui aboutit à un fort couplage, pour notre plus grand déplaisir. À la clé, un code pas maintenable, pas scalable et qui se met en travers du chemin des évolutions stylistiques et structurelles.

Dans ces conditions, est-il réaliste, possible voire souhaitable d'appliquer un modèle dit « orienté objet » à CSS et HTML ? La réponse à toutes ces questions est **oui**, mais en abandonnant toute démarche impérative. En clair, en laissant CSS bosser tranquille.

Une entreprise CSS qui roule

Le duo HTML/CSS a été pensé pour se faire une *déclaration* d'amour permanente, pas pour s'échanger une liste de course au détail. CSS, en particulier, saura se montrer le plus performant si on lui tient un langage hautement déclaratif, qui lui laisse un maximum de marge de manœuvre. Et il se trouve — oh comme la nature est bien faite ! — que CSS possède une architecture tout à fait adaptée à la programmation déclarative, la *cascade*.

> Si la différence entre impératif et déclaratif n'est pas claire, voici un bon résumé.

On souhaite obtenir une CSS scalable, facilement extensible, à laquelle on puisse apporter des modifications en toute confiance. Pour ça, il faut que le corpus CSS reste simple, donc que les sélecteurs aient une spécificité moyenne basse. La meilleur manière de faire ça, on l'a vu, c'est de mettre en place des *namespaces*. Ils vont agir comme des **abstractions**, minimisant le couplage entre CSS et HTML. Et de ce premier concept d'abstraction, découlent tous les autres.

À partir d'une abstraction donnée (`button`), sur un projet qui grossit, on cherchera inévitablement à déployer des variations. Tout le sel de l'intégration CSS tient en effet à la gestion de cas particuliers : « ce bouton-ci est actif », « celui-là est désactivé », etc. Chaque variation sera telle une **instance** spécifique d'une abstraction commune, et en tant qu'instance, elle aura un état et une interface. En OOCSS, état et interface sont une seule et même chose, un nom de classe : `button-active`.

On mettra également à profit une forme d'**héritage** un peu particulière intégrée nativement à CSS, la *cascade*, pour rester DRY : `button button-active`, `button button-disabled`.

Et dans les quelques cas particuliers où état et interface gagneraient à être séparés, de sorte qu'un style profite au plus grand nombre, le **polymorphisme** pourra être implémenté par des micro-classes globales : `button hidden` et `link hidden`.

Les employés, tous syndiqués chez BEM

Tout ça, c'est très bien, mais encore faut-il l'appliquer, et sur la durée. Rien de mieux pour ça que de se doter de quelques règles simples et efficaces. En CSS, tout passe par le nom des classes, donc ces règles seront avant tout liées à une convention de nommage. L'une d'entre elles s'appelle BEM : *Bloc, Element, Modifier*. Elle est populaire car simple, efficace, et de surcroît pas *trop* contraignante (elle l'est forcément un peu !)

> BEM is a development methodology that allows team members to collaborate and communicate ideas using the unified language consisting of simple yet powerful terms: blocks, elements, modifiers. It is a collection of ideas and methods. Each company and each team may integrate it into an existing workflow gradually, finding out what works best for them.

L'idée de BEM, c'est de faire de l'OOCSS, mais de remplacer la terminologie classique de la POO (abstraction / encapsulatruc / blablabla / *boooring!*) par quelque chose qui colle plus à l'interface utilisateur (UI). Pour plus de détails, voir cet article qui explique « pourquoi BEM ? ».

- Un bloc (*block*) représente un objet de l'UI : un formulaire de login, un menu, un profil utilisateur…
- Un élément (*element*) est un composant interne d'un bloc : un bouton de validation *du formulaire*, un item *du menu*, la preview de l'avatar *de l'utilisateur*…)
- Un modificateur (*modifier*) représente une variation possible d'un bloc ou d'un élément : le formulaire de login, mais *replié* ; un item *actif* du menu ; le menu, mais en version *condensé pour téléphones portables*…

Ces relations sont matérialisées au travers d'une syntaxe hautement déclarative, avec des *fully-qualified selectors* (en gros, des classes sémantiques). La syntaxe officielle est la suivante :

```
.human-being {}                 /* block */
.human-being--female {}         /* block--modifier */
.human-being__hand {}           /* block__element */
.human-being__hand--left {}     /* block__element--modifier */

/* Mais aussi : */
.human-being--female .human-being__hand {}         /* block--modifier  block__element */
.human-being--female .human-being__hand--left {}   /* block--modifier  block__element--modifi
```

On peut tout à fait envisager une autre convention, tout en gardant la même architecture :

```
.humanBeing {}
.humanBeing--female {}
.humanBeing-hand {}
.humanBeing-hand--left {}
.humanBeing--female .humanBeing-hand {}
.humanBeing--female .humanBeing-hand--left {}
```

Avec BEM, on obtient un code HTML purement déclaratif vis-à-vis de CSS :

```
<form class="site-search  site-search--full">
  <input type="text" class="site-search__field">
  <input type="Submit" value ="Search" class="site-search__button">
</form>
```

Pour plus d'exemples, voir ici, là et bien sûr sur le site officiel.

On retrouve ainsi les concepts fondamentaux de la POO, mais appliqués à une interface utilisateur. Ces principes nous permettent de construire une CSS fiable, résiliente et facile à modifier. Contrairement à Bootstrap qui travaillait contre CSS, BEM cherche à remettre l'intégrateur en position de responsabilité. Vous allez pouvoir décider de l'organisation interne la plus adéquate, en fonction de ce que déclare être un élément du DOM, tel que `.site-search__button`. Envie d'utiliser uniquement de la CSS pur jus ? Chaque classe sera alors un assemblage « impératif » de règles CSS. Plutôt versé dans les préprocesseurs comme Sass ou Less ? Vous pourrez mettre à profit variables, mixins et autres *extend* pour éviter de la duplication et composer vos classes haut-niveau à partir de briques réutilisables.

> En utilisant la convention officielle, voilà ce que deviendraient certains des sélecteurs pour le module et son titre :

Avant	Après
`.module-title`	`.module__title`
`.module-title-urgent`	`.module__title--urgent`
`.module-special .module-title`	`.module--special .module__title`
`.module-special .module-title-urgent`	`.module--special .module__title--urgent`

En résumé

La logique un peu verbeuse de BEM découle d'une analyse fine des problèmes et solutions habituellement rencontrés en CSS. Cette convention permet de construire pro-activement un corpus CSS :

- maintenable
- compréhensible
- facile à modifier et à étendre
- performant
- dépourvu de code inutile
- gèrant bien le *responsive*

Vu comme ça, peu importe si le nom des classes est un peu long, l'auto-complétion et les pré-processeurs sont là pour ça !

Liens utiles

CSS3 module: W3C Selectors

https://www.w3.org/Style/css3-selectors-updates/WD-css3-selectors-20010126.fr.html#selectors