Les processeurs CSS

- Quelques problèmes de production de codes CSS de grande taille :
  - difficilement lisible
  - difficilement maintenable
  - difficile à réutiliser
    - découper en + fichiers ?
    - construire des classes génériques ?
  - couplage html-css : éviter qu'une modification html impacte trop le css et inversement

Quelques exemples de problèmes

- Évolutivité et maintenance :
  - « foncer tous les bleus et changer les rouges en orange »
- Réutilisation :
  - « je veux le même bouton que sur la page d'accueil, mais avec un bleu dégradé à la place du vert »
- Couplage html-css

couplage html-css

- Css générique réutilisable (framework)
- Html **non "sémantique"** totalement lié au css

```
<input class="btn btn-lg btn-green flat">
```

```
.btn {
    ...
}
 .btn-lg {
    ...
 }
 .btn-green {
    ...
```

couplage html-css

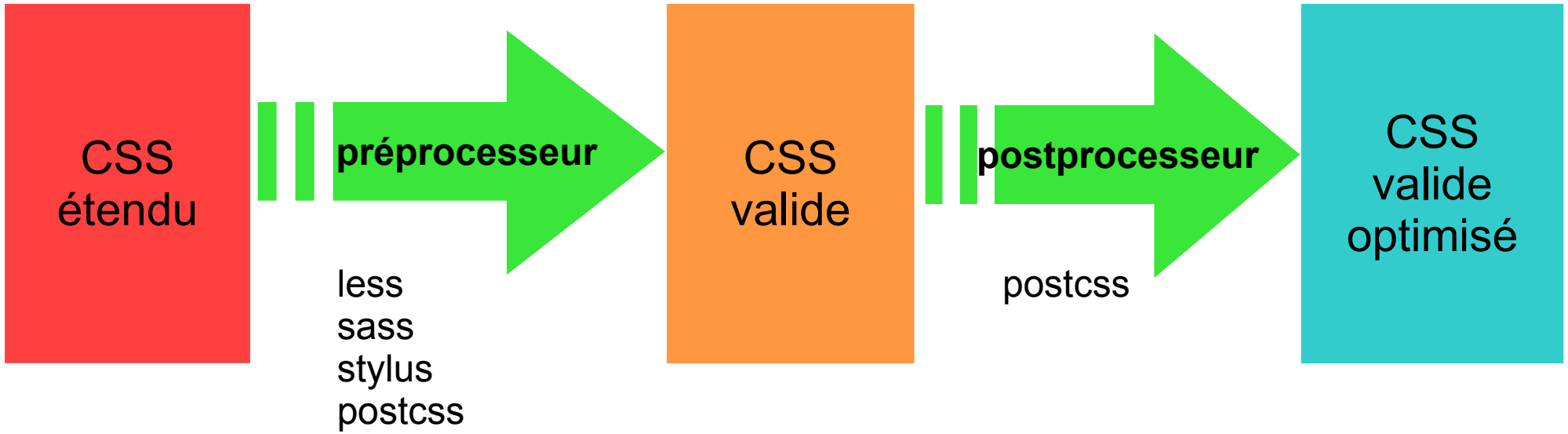
- Html « **sémantique** » : classes liées au document et au rôle de l'élément dans le document
- Css : **peu réutilisable** car lié à 1 application

```
<input class="valid_inscription" type="submit" value="...">
```

```
.valid_inscription {
    border: 1px solid #030303;
    font-size: 1rem;
    line-height: 1.42rem;
}
```

les principes des processeurs css

- **objectif** : améliorer la **lisibilité**, la **maintenabilité** et la **réutilisabilité** de codes css en proposant un langage étendu plus expressif, compilé en css



et ça se passe où ?

- **dans l'environnement de développement :**
  - exécuté à chaque modification de la source
  - intérêt : pas de surcharge du client et du réseau
  - exemple : **sass/scss, stylus, less, PostCss**

les apports des processeurs css

- des **variables**, des **expressions** (ouf !)
  - Facilite la maintenance, plus d'expressivité
- des **sélecteurs imbriqués**
  - Améliore la lisibilité
- **assemblage de fichiers**
  - Améliore la modularité
- **réutilisation** et extension de **classes**, **mixins**
  - Améliore la réutilisation de code, facilite la production de librairies et de frameworks

postCss

- post-processeur css écrit en js
- détection d'erreurs (**lint**)
- syntaxes et fonctionnalités avancées ou en cours de validation par le W3C
- **autoprefixer** : ajout des préfixes navigateurs

sass/scss

- pré-processeur à ajouter dans l'environnement de développement
- disponible en 2 syntaxes :
  - sass : syntaxe ruby-like
  - scss : syntaxe css-like
- source : http://sass-lang.com

principes de base scss

```
#navbar {
    width: 80%;
    height: 23px; }
```

- scss : **sur-ensemble** de css
- compilation :
  - compilation : `$>sass scss/style.scss:css/style.css`
  - compilation à la volée à chaque changement

```
$>sass –watch scss/style.scss:css/style.css
```

    - à chaque changement dans le fichier scss/style.scss, le compilateur le transforme et écrit le résultat dans le fichier css/style.css
  - ou sur des répertoires :

```
$>sass –watch scss:css
```

imbrication

- imbrication des sélecteurs

scss

```
.tside {
   width: 23em  ;
   padding: 1em ;
    p {
      font-size:0.9em ; }
}
```

sass

css

```
.tside {
   width: 23em  ;
   padding: 1em ;
}
.tside p {
    font-size:0.9em ;
}
```

- référence au parent :

scss

```
.tside {
   width: 23em  ;
   padding: 1em ;
   & > ul {
    font-size:0.9em ;
    &>li > a {
      color: red ;
      &:hover {
        color: maroon ;
       } } }
}
```

sass

css

```
.tside {
   width: 23em  ;
   padding: 1em ;
}
.tside>ul {
    font-size:0.9em ;
}
.tside>ul>li>a {
    color: red ;
}
.tside>ul>li>a:hover {
    color: maroon;
}
```

utilisation de l'imbrication

- **attention** : ne pas abuser de l'imbrication car conduit à des sélecteurs avec des expressions de chemins très longues
  - problèmes de **performance** pour le navigateur
  - rend le code **css très dépendant de l'arbre html**
- à utiliser pour définir des "widgets" html/css réutilisables

Variables

- **variables** : à utiliser sans modération

scss

```
$bcolor: #fafafa ;
$fcolor: #020202 ;
nav {
    background-color: $bcolor ;
    color: $fcolor ;
}
```

sass

css

```
nav {
    background-color:#fafafa;
    color: #020202 ;
}
```

- valeurs par défaut : affectation seulement si la variable n'a pas déjà une valeur définie par ailleurs

```
$bcolor: #fafafa !default;
$fcolor: #020202 !default;
```

expressions

- permet d'utiliser des expressions pour calculer des valeurs
- fournit une bibliothèque de fonctions prédéfinies

scss

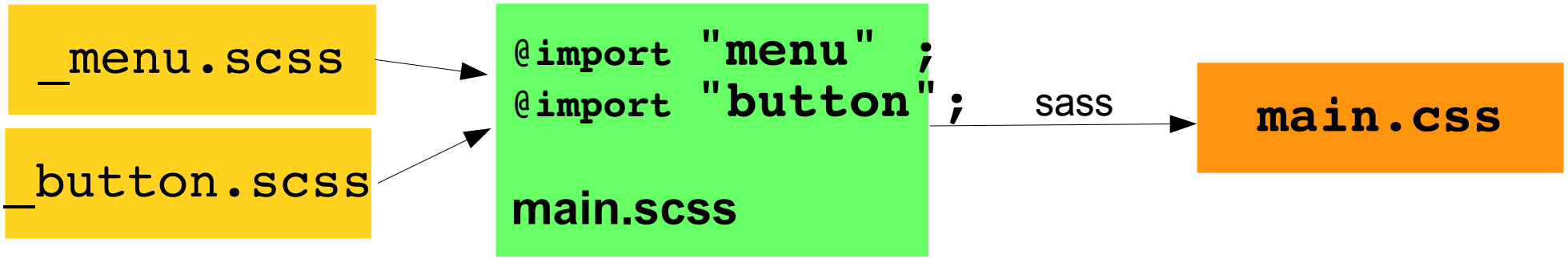
```
$items: 5  ;
$bcolor: rgba(#f6f6f6,1);

article {
   float: left ;
   margin: 0 1% ;
   width: (100%/$items) - 2% ;
   background-color: $bcolor ;
   color: darken($bcolor,50) ;
}
```

css

```
article {
  float: left ;
  margin: 0 1% ;
  width: 18% ;
  background-color: #f6f6f6 ;
  color: #767676 ;
}
```

import

- @import : permet d'importer un "partial" (fichier scss) dans un fichier
- intérêt : découper et réutiliser du côté du serveur
- 1 seul fichier produit et transféré au client

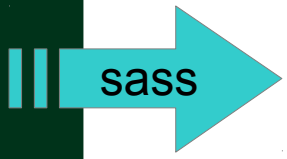


extension

- @extend : réutilisation et extension d'une règle existante

scss

```
.message {
  padding: 10px ;
  border: 2px solid #ccc ;
  color: #333 ;
}

.mess-success {
  @extend .message;
  border-color: green ;
}

.mess-error {
  @extend .message;
  border-color: red ;
}
```



css

```
.message,
.mess-success,
.mess-error {
  padding: 10px ;
  border: 2px solid #cccccc
 ;
  color: #333333  ;
}


.mess-success {
  border-color: green ;
}

.mess-error {
  border-color: red ;
}
```

- **Attention** : non utilisable à l'intérieur d'un @media

media query

- identique à css, mais peuvent être placées à l'intérieur d'un sélecteur :

scss

```
.section article
{

  padding: 10px ;
  border: 2px solid #ccc ;
  color: #333 ;
  float: left ;
  width: 100% ;

   @media screen and
    (min-width: 35em){
       width: 50%
   }

}
```

css

```
.section article {
  padding: 10px ;
  border: 2px solid #ccc ;
  color: #333 ;
  float: left ;
  width: 100% ;
}

@media screen and
  (min-width: 35em){
  .section article {
      width: 50%
 }
}
```

utilisations

- Produire du code réutilisable et évolutif
- Découpler les classes utilisées dans l'arbre html des classes de mise en forme définies dans le code css
  - Html : classes « sémantiques »dont le nom est lié au document ou à l'application, et au rôle de l'élément dans ce contexte
  - Css : classes dont le nom est lié à l'effet produit

enfin du bon code ?

```
<input class="valid_inscription">
```

```
$btn-green-bg: #318054;

@import "mySassyButtons";

.valid_inscription {
  @extend .btn;
  @extend .btn-lg;
  @extend .btn-green;
  @extend .flat;
  font-weight: 600;
}
```

```
.typography {
  i, em { @extend .italic; }
  b, strong { @extend .bold; }

  h1, .h1 { @extend .h1; margin: 1em 0 0.5em; }
  h2, .h2 { @extend .h2; margin: 1em 0 0.5em; }
  h3, .h3 { @extend .h3; margin: 1em 0 0.5em; }
  h4, .h4 { @extend .h4; margin: 1em 0 0.5em; }
  h5, .h5 { @extend .h5; margin: 1em 0 0.5em; }
  h6, .h6 { @extend .h6; margin: 1em 0 0.5em; }

  p, ul, ol, pre { @extend .block-margins; }

  ul { @extend .unordered-list; }
  ol { @extend .ordered-list; }

  pre, code { @extend .fixed; }
}
```

```
<div class="blog-post typography">
  <h1>Blog post</h1>
  ...
</div>
```

http://thesassway.com/advanced/modular-css-typography

bénéfices (1)

- on obtient du code :
- mieux lisible grâce à l'imbrication de sélecteurs
- mieux maintenable et plus évolutif grâce aux variables, expressions et mixins
- plus facilement réutilisable grâce aux variables, extends et mixins
- tout en bénéficiant de fonctions prédéfinies

bénéfices (2)

- On peut se faire sa bibliothèque/framework d'éléments de mise en page prédéfinis mais adaptables et paramétrables, sous forme de classes ou mieux, de mixins

- Les gros frameworks sont faits avec un préprocesseur css :
  - Twitter bootstrap : less & sass
  - Foundation, Materialize : sass

découplage html-css : le cas des framework css

- les apports d'un framework css : des classes prédéfinies facilement utilisable et permettant une mise en forme rapide
  - grille générique prédéfinie, typographie,
  - composants : navbar, boutons, alertes, forms ...
  - des media-query
  - des plugin javascript

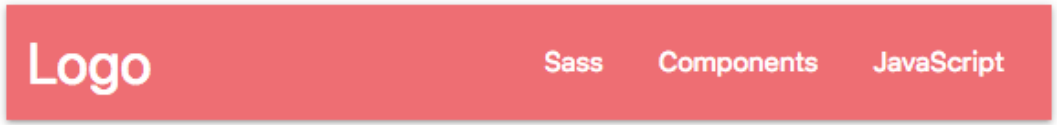
les inconvénients

- uniformise les designs
- code html envahi par le framework css
  - introduit un couplage très fort entre document html et mise en forme css
  - rend quasiment impossible la refonte css en utilisant un autre framework
- conduit à un code html non sémantique
  - beaucoup de class liés à la mise en forme et non pas à la structure du document

utiliser un framework avec sass

- **1ere solution :** utiliser un framework fournit sous forme de classes css avec la directive sass @extend
  - permet de réaliser effectivement le découplage html / css
  - html : classes sémantiques
  - css : définition des classes sémantiques en réutilisant les classes du framework
  - rappel: @extend dans une media-query

exemple materialize




```
<nav>
 <div class="nav-wrapper">
   <a href="#" class="brand-logo">
           Logo</a>
   <ul id="nav-mobile"
    class="right hide-on-med-and-down">
     <li><a href="sass.html">
           Sass</a></li>
     <li><a href="badges.html">
        Components</a></li>
     <li><a href="collapsible.html">
        JavaScript</a></li>
    </ul>
  </div>
</nav>
```

```
<nav class="navigation">
  <a href="#">Maison</a>
  <ul>
    <li><a href="#">pomme</a></li>
    <li><a href="#">poire</a></li>
    <li><a href="#">pêche</a></li>
    <li><a href="#">melon</a></li>
  </ul>
</nav>
```

```
nav.navigation {
  @extend .nav-wrapper;

  &>a:first-child {
    @extend .brand-logo, .right;
  }

  &>ul {
    @extend .left,
            .hide-on-med-and-down;
    &>li:nth-child(3) {
      @extend .active;
    }
}}
```

2ème solution

- on utilise la version sass/scss du framework/grille sous la forme de classes réutilisables et de mixins
  - bootstrap, foundation :
    - écrits en sass,
    - mixins fournis
  - susy, neat, griddle : des grilles écrites en sass

conclusion

- outils d'ingénierie
- utiles pour des projets
- facilite la réutilisation, l'adaptation, l'évolution et le partage de codes css