



Beste de savoir

Reprenez le contrôle de vos feuilles de
style avec Sass

3 septembre 2018

Table des matières

I. Introduction	4
0.0.1. En utilisant Sass, pardi!	5
II. Les bases de Sass	6
1. Sass, un préprocesseur CSS	8
1.1. Un préprocesseur CSS	8
1.2. À quoi ça sert ?	9
1.3. Installation	11
1.4. Notre projet fil rouge	12
1.5. La compilation	12
1.6. En résumé	13
2. L'imbrication	14
2.1. L'imbrication, pourquoi faire ?	14
2.2. La référence au parent	16
2.3. Imbrication de media-queries	17
2.4. Bonne pratique : « The Inception Rule »	18
2.5. En résumé	19
2.6. Pour s'entraîner	19
3. Les variables	20
3.1. Les variables, pourquoi faire ?	20
3.2. Les différents types de données	22
3.3. L'interpolation	24
3.4. La règle !default	26
3.5. En résumé	26
3.6. Pour s'entraîner	27
4. La directive @import	28
4.1. Importer des feuilles de style	28
4.2. Bonne pratique : Les feuilles partielles	29
4.3. En résumé	30
5. Les mixins (1/2)	31
5.1. Un mixin, pourquoi faire ?	31
5.2. Inclure un mixin	32
5.3. Les arguments	33
5.4. En résumé	34

6. Les mixins (2/2)	36
6.1. Valeurs par défaut, arguments nommés et listes d'arguments	36
6.2. Passer un bloc de contenu à un mixin	39
6.3. Bonne pratique : Arguments & type null	40
6.4. En résumé	42
7. Les fonctions	43
7.1. Un fonction ?	43
7.2. Quelques fonctions bien utiles	44
7.2.1. Pour définir une couleur	45
7.2.2. Pour jouer avec les nombres	45
7.3. La palette du peintre	46
7.3.1. Inverse, Nuance de gris et Mélange	46
7.3.2. Lumière, Saturation, Opacité	46
7.4. Jouons avec les listes	49
7.5. Créer sa propre fonction	51
7.6. En résumé	52
III. Plus loin avec Sass	54
8. L'héritage avec @extend	56
8.1. Il est où le grisbi ?	56
8.2. Les placeholders (un nouvel eldorado ?)	58
8.3. Bonne pratique : Faut-il oublier @extend ?	59
8.3.1. Héritage & Media-queries	59
8.3.2. Comportements imprévus	60
8.3.3. On oublie l'héritage ?	62
8.4. En résumé	62
9. Les conditions	63
9.1. Les bases	63
9.2. @else, @else if...	64
9.3. Mini-TP : un mixin bien compliqué	66
9.3.1. Prérequis	66
9.3.2. La solution	67
9.3.3. Un peu mieux...	67
9.4. En résumé	67
9.5. Pour s'entraîner	68
Contenu masqué	70
10. Les boucles	72
10.1. La boucle @for	72
10.2. La boucle @each	73
10.3. Mini-TP : La gestion d'une sprite	75
10.3.1. Énoncé	75
10.3.2. Correction	76
Contenu masqué	78

11. Et maintenant ?	79
11.1. Le type map	79
11.2. Utiliser des bibliothèques externes	80
11.2.1. Neat	80
11.2.2. Bourbon	82
11.2.3. Susy3	83
11.3. RubySass, LibSass, DartSass, et les autres...	83

IV. Conclusion **86**

% REPRENEZ LE CONTRÔLE DE VOS FEUILLES DE STYLE AVEC SASS % Matouche %
19 septembre 2017

Première partie

Introduction

I. Introduction

Lorsqu'on découvre comment sont faits les sites Web, on est souvent étonné de voir que le contenu (HTML) est clairement séparé de la mise en page (CSS). Et puis, assez vite, on ne peut plus s'en passer. C'est tellement pratique ! Mais au bout d'un moment, nos fichiers CSS deviennent très longs, très très longs, et on ne s'y retrouve plus. Alors, comment **mieux organiser son code**, comment le rendre **plus maintenable** ?

0.0.1. En utilisant Sass, pardi !



FIGURE 0.1. – Faites place à Super-Sass !

i

Prérequis

Connaître le fonctionnement des feuilles de style, avoir des bases solides en CSS.

Objectifs

Découvrir les différents outils que Sass met à notre disposition pour faciliter l'organisation et la réutilisation des feuilles de style.

Autres remarques

Ce tutoriel se basera sur un exemple « fil rouge » dont le code est disponible [sur GitHub](#) . Nous en reparlerons au cours du premier chapitre, ne vous inquiétez pas.

Deuxième partie

Les bases de Sass

II. Les bases de Sass

Dans cette première partie, nous allons installer Sass et découvrir les outils principaux qu'il met à disposition des développeurs pour organiser et gérer les styles. Il sera question d'imbrication, de mixins, de variables, de fonctions, etc.

1. Sass, un préprocesseur CSS

Au départ ce n'était qu'un petit jardin bien entretenu. Et puis, peu à peu, la forêt s'est densifiée, les règles se sont multipliées, la jungle s'est installée. Vos feuilles de styles sont devenues surchargées et illisibles. Vous êtes un jeune aventurier et vous avez peur de de vous lancer dans cette périlleuse expédition ? Il y a encore peu de temps, on vous aurait répondu de vous débrouiller. Mais aujourd'hui, une solution existe.

Un véritable produit miracle, un héros du vingt-et-unième siècle, une chimère devenue réalité ! On appelle cela... un préprocesseur CSS.



Mais, c'est quoi un préprocesseur CSS ?

C'est tout le sujet de ce chapitre, ami lecteur...

1.1. Un préprocesseur CSS

Commençons par regarder sur Wikipédia, qui nous donne une définition générale bien obscure...

En informatique, un préprocesseur est un programme qui procède à des transformations sur un code source, avant l'étape de traduction proprement dite (compilation ou interprétation).

Wikipédia

Essayons d'expliquer cela de manière plus simple. Un préprocesseur est un programme jouant le rôle d'une moulinette : on lui donne du code source et, en échange, il génère un code modifié. Dans le cas d'un préprocesseur CSS comme Sass, on donne des fichiers écrits dans un langage spécifique à Sass et la moulinette génère des feuilles de style CSS qui pourront être comprises par un navigateur. On appelle ce processus **la compilation**.



FIGURE 1.1. – On donne des fichiers SCSS à Sass, et il génère des fichiers CSS.

Ce langage spécifique est le *SCSS* (pour « Sassy CSS »). On retrouve dans ce nom les lettres CSS, car la syntaxe du langage SCSS se base sur celle de CSS. En fait, tout code CSS est compatible SCSS. Cela veut dire que tout travail en CSS est directement transposable en SCSS. Il suffit pour cela de changer l’extension du fichier pour `.SCSS`.

1.2. À quoi ça sert ?

Si le SCSS a une syntaxe basée sur celle de CSS, on peut se demander l’intérêt de Sass. En fait, Sass ajoute à CSS un ensemble de fonctionnalités qui permettent d’organiser de manière plus maintenable les feuilles de style. Sa devise pourrait être “Don’t Repeat Yourself” (ne vous répétez pas). Il ne s’agit donc pas de rajouter des propriétés, mais d’aider le développeur à y voir plus clair, à défricher son code (d’où la métaphore de la forêt vierge, CQFD). Dans ce but, Sass permet de factoriser certains bouts de code, stocker les couleurs/polices/dimensions fréquemment utilisées en un unique endroit, rendre certaines règles plus facilement réutilisables d’un projet à l’autre.

Comme tout est plus simple avec un exemple, je vous propose comme fil rouge à ce cours le site Web de *Citron Inc*, une petite entreprise spécialisée dans la production de limonades biologiques. Parce que oui, j’ai de l’imagination :

II. Les bases de Sass

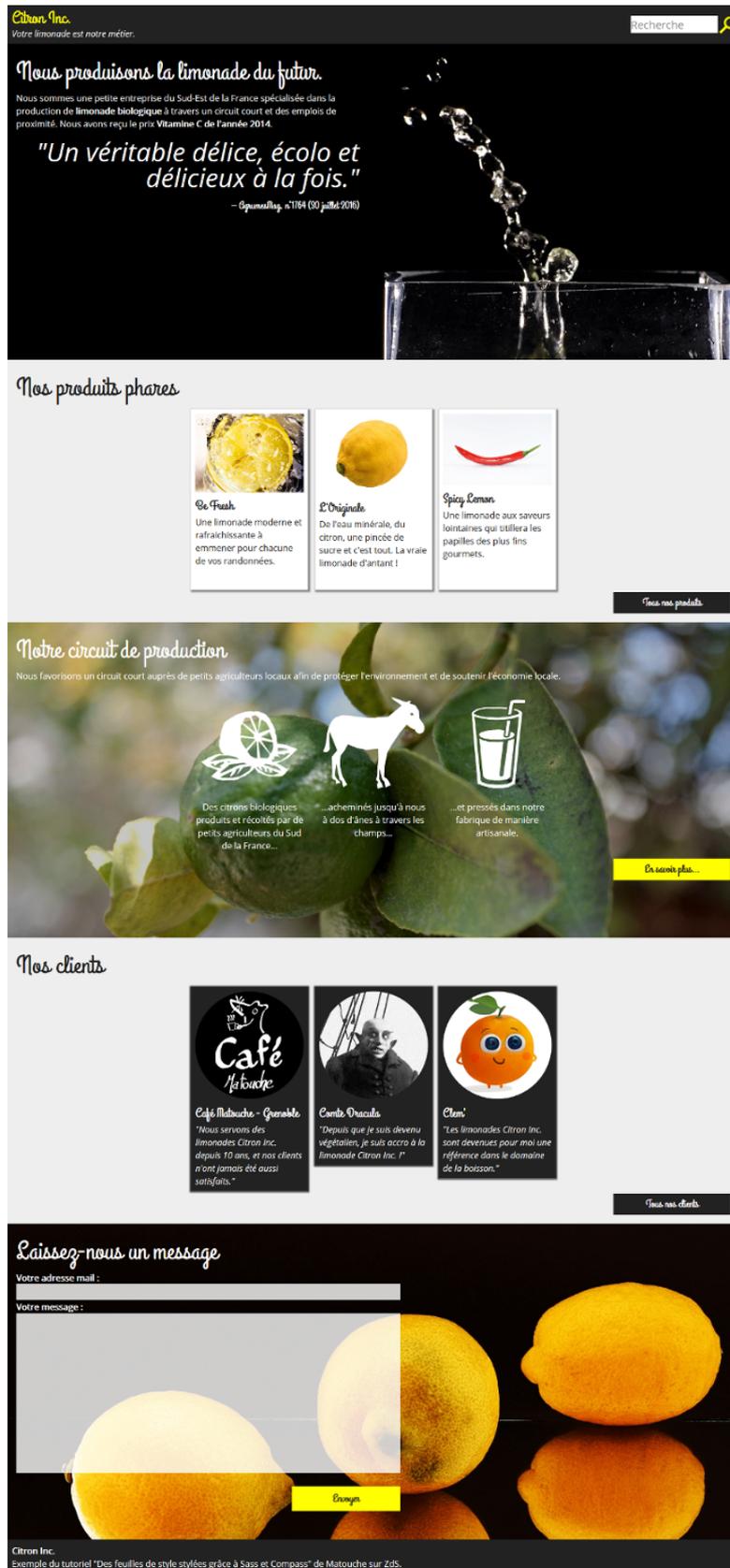


FIGURE 1.2. – Maquette du fil rouge

On remarque plusieurs choses :

II. Les bases de Sass

- Les mêmes couleurs (jaune, gris foncé, etc.) et les mêmes polices sont utilisées pour plusieurs éléments.
- Les "cartes" représentant un produit du catalogue, une étape de fabrication ou un client partagent quelques propriétés. Il serait utile de « factoriser » toutes ces propriétés, pour que le fichier de travail soit moins long, sans pour autant qu'on perde en lisibilité.
- Les boutons ont la plupart de leurs propriétés en commun (dimensions et police, notamment), mais pas toutes (les couleurs sont différentes). De plus, on peut facilement imaginer que des boutons similaires soient présents sur d'autres projets, et il serait fatigant de réinventer la roue à chaque fois.

Cet exemple mis à part, certains problèmes d'organisation reviennent fréquemment dans de nombreux projets :

- Séparer les styles en plusieurs fichiers permet de mieux s'organiser durant le développement, et facilite le recyclage de portions de code, mais demande plus de requêtes, donc plus de ressources en production.
- En CSS il est fréquent de répéter plusieurs fois les mêmes sélecteurs. Dans notre exemple, on va ainsi cibler l'élément `header`, puis `header h1`, et enfin `header p`. À grande échelle, c'est plutôt barbant...

Vous l'aurez compris, Sass propose une solution à tous les problèmes que je viens de citer en ajoutant de nouveaux mots à la syntaxe CSS. Ces nouveaux mots facilitent la vie du codeur, mais sont parfaitement invisibles pour le visiteur du site Web : celui-ci continuera de recevoir un feuille de style CSS qui aura été générée par Sass.

1.3. Installation

Vous n'en pouvez plus attendre d'avoir Sass sur votre machine ? Avant tout, chers lecteurs, il faut que vous disposiez d'un interpréteur Ruby. Ruby est le langage de programmation utilisé par les développeurs de Sass. Pour obtenir l'interpréteur Ruby, tout dépend du système d'exploitation.

- Sous Linux, il est fort probable que votre distribution fournisse Ruby par défaut. Sinon, il est disponible dans les dépôts officiels et s'installe via un gestionnaire de paquets (`dnf`, `apt`, `pacman`, etc.).
- Sous macOS, vous pouvez sauter cette étape : Ruby est installé par défaut.
- Sous Windows, vous pouvez télécharger l'installateur depuis [le site officiel](#)  . Durant l'installation, pensez à cocher la case "Add Ruby executables to your PATH".

Maintenant, nous pouvons installer Sass avec `gem`. Gem est le gestionnaire de paquets de Ruby. Si c'est votre première utilisation de la ligne de commande, jeunes padawans, pas de panique : c'est d'une simplicité déconcertante. Lancez un terminal (sous Windows, vous pouvez utiliser l'invite de commande `cmd`) et entrez : `gem install sass` (attention à ne pas oublier `sudo` sur les systèmes Unix).



Si Windows n'arrive pas à trouver gem, c'est peut-être que vous n'avez pas coché durant l'installation de Ruby la case "Add Ruby executables to your PATH".

Ça y est, c'est fini! Vous pouvez vérifier que sass est bien installé en tapant `sass -v` dans un terminal : si vous obtenez le numéro de version de Sass, c'est que tout s'est bien passé.

1.4. Notre projet fil rouge

Revenons à notre fil rouge : le site de limonade. Je vais vous fournir son code comme exemple tout au long de ce cours. Nous partirons donc d'un gros fichier CSS (et de la page HTML qui va avec) et nous allons chercher à le rendre le plus lisible, compréhensible et maintenable possible avec l'aide de Sass. Cela vous permettra de voir progressivement comment vous pouvez vous faciliter la vie avec Sass, ainsi que d'adopter quelques réflexes.

Vous pouvez récupérer l'ensemble du code de notre fil rouge sur [Github](#) .

À l'intérieur du répertoire CitronInc, vous pouvez voir notre page Web, le dossier `sass` qui sera notre dossier de travail pour les fichiers SCSS et un dossier `stylesheets` pour les fichiers CSS générés. Il est habituellement préférable de séparer les fichiers SCSS des fichiers CSS si on veut s'y retrouver un minimum. J'ai déjà placé dans le dossier `sass` un fichier `main.scss` qui contient exclusivement du code CSS. Nous verrons étape par étape comment Sass nous permet de rendre ce code plus maintenable. Il y a aussi un fichier `main.css` dans le dossier `stylesheets` c'est le fichier CSS compilé par Sass. Pour l'instant, comme `main.scss` ne contient aucune commande spécifique à Sass, les deux fichiers sont strictement identiques.

Vous remarquerez au passage un dossier `img` qui contient des... images (je sais, c'était dur à deviner), et le dossier `correction` qui contient les corrections des exercices que je vous proposerai en fin de chapitres.

Voilà, nous sommes prêts à commencer! Si vous préférez démarrer directement avec l'un de vos projets actuels plutôt que de prendre mon exemple, n'hésitez pas : vous ne devriez pas avoir trop de mal à appliquer ce que je vais vous montrer sur un autre code.

1.5. La compilation

Maintenant que le dossier de notre projet est prêt, voyons comment compiler une première fois notre feuille de style, c'est-à-dire générer `stylesheets/main.css` à partir de `sass/main.scss`.

Tout d'abord, placez-vous à l'intérieur du dossier `CitronInc` en ligne de commande. Utilisez pour cela la commande `cd` (si vous êtes familiers de celle-ci) ou bien, dans l'explorateur de fichier de Windows, faites un Clic-Droit avec la touche Maj enfoncée et choisissez "Ouvrir une fenêtre de commande ici".

La commande pour indiquer à Sass de compiler tous les fichiers SCSS est `sass --watch`. Elle s'utilise ainsi :

```
1 sass --watch sass:stylesheets
```

II. Les bases de Sass

Comme vous pouvez le voir, elle prend deux paramètres, séparés par `:` : le nom d'un fichier/dossier d'entrée et celui d'un fichier/dossier de sortie. Si tout se passe bien, vous devriez voir ceci s'afficher :

```
1 >>> Sass is watching for changes. Press Ctrl-C to stop.
2     write stylesheets/main.css
3     write stylesheets/main.css.map
```

Sass nous indique qu'il est désormais en train de **surveiller** les changements dans le dossier *sass* : chaque fois que l'on modifiera un fichier SCSS, les fichiers CSS correspondants seront régénérés de manière transparente. Ainsi toute modification est directement visible dans le navigateur (et ceci d'autant plus rapidement si vous utilisez des extensions comme LiveReload). Par ailleurs, Sass nous dit qu'il vient de recompiler notre fichier *main.css*, et de créer un fichier *main.css.map*. Ce dernier est utile pour afficher le fichier SCSS original directement dans les outils de développement de votre navigateur (généralement accessibles via F12). Cela permet par exemple de sélectionner un élément dans la page et de voir les règles SCSS qui s'appliquent à ce dernier.

1.6. En résumé

- Sass est un **préprocesseur CSS**, c'est-à-dire qu'il génère du CSS à partir de fichiers `.SCSS` ;
- Utiliser un préprocesseur CSS permet d'**optimiser son code** et de moins se répéter ;
- La syntaxe par défaut de Sass est le **SCSS** et tout code CSS est compatible avec cette syntaxe.
- La commande `sass --watch input:output` indique à Sass qu'il doit **recompiler automatiquement** les fichiers `.scss` du dossier `input` à chaque modification et placer les fichiers CSS générés dans le dossier `output`.

Voilà, nous arrivons à la fin de ce premier chapitre, j'espère qu'il vous a plu. Dans le prochain chapitre, nous nous intéresserons à un premier concept clef de Sass, l'imbrication de règles.

2. L'imbrication

Pour commencer en douceur, on va s'intéresser dans ce chapitre à l'*imbrication de règles*, une fonctionnalité fondamentale de Sass. Comme vous aller le voir, elle s'inscrit vraiment dans le principe « Don't Repeat Yourself ».

2.1. L'imbrication, pourquoi faire ?

Commençons par observer cet extrait de notre feuille de style :

```
1 blockquote {
2     font-style: italic;
3 }
4 blockquote:before, blockquote:after {
5     content: '""';
6 }
7 .quote {
8     text-align: right;
9     margin: 1rem 0;
10 }
11 .quote blockquote {
12     font-size: 3rem;
13     line-height: 1em;
14 }
15 .quote figcaption {
16     font-family: "Grand Hotel", serif;
17     font-size : 1.2rem;
18     margin-top: 1rem;
19 }
20 .quote figcaption:before {
21     content: '\2012 ';
22 }
```

On remarque que `blockquote`, `.quote` et `figcaption` sont répétés plusieurs fois. Ce n'est pas encore trop rébarbatif à cette échelle là, mais ça l'est au niveau de la feuille de style en entier : on n'a pas forcément envie de répéter plusieurs fois les mêmes sélecteurs. Avec Sass, les règles concernant `.quote` peuvent être écrites de la manière suivante en utilisant l'imbrication de règles :

```
1 .quote {
2   text-align: right;
3   margin: 1rem 0;
4   blockquote {
5     font-size: 3rem;
6     line-height: 1em;
7   }
8   figcaption {
9     font-family: "Grand Hotel", serif;
10    font-size : 1.2rem;
11    margin-top: 1rem;
12  }
13  figcaption:before {
14    content: '\2012 ';
15  }
16 }
```

Avec l'imbrication, nous n'avons écrit `.quote` qu'une seule fois et nous avons mis les blocs `blockquote`, `figcaption` et `figcaption:before` à l'intérieur du bloc `.quote`. Sass comprend lors de la compilation que les règles ne doivent concerner que les éléments `<blockquote>` et `<figcaption>` situées à l'intérieur d'un élément de classe `.quote`. On obtient bien la même chose qu'avec le code précédent. Cependant, vous remarquez que l'on a regroupé toutes les règles concernant un module (la citation) dans un même bloc. Ainsi, le code est plus lisible, car il est mieux *organisé*.

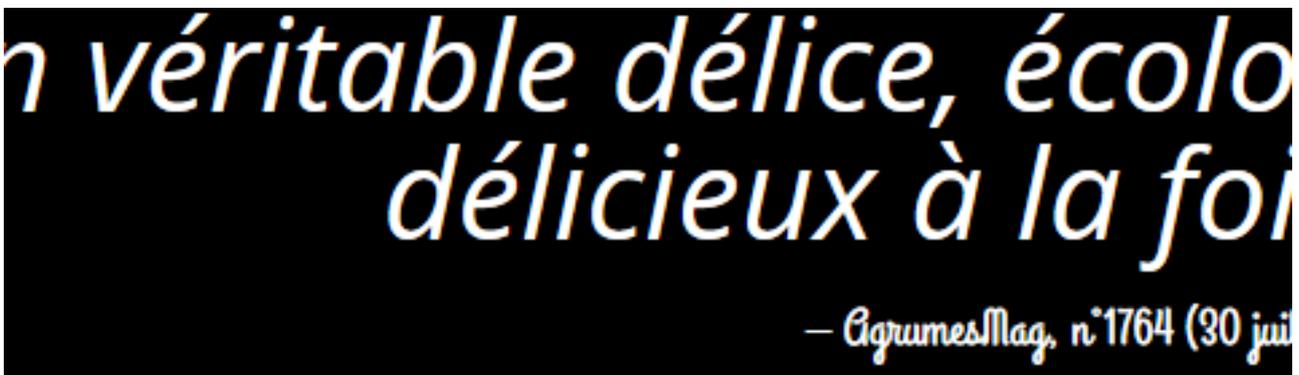


FIGURE 2.1. – Toutes les règles spécifiques à ce module sont regroupées en un bloc dans le code Scss

On peut aussi imbriquer des propriétés CSS. Cela peut être utile pour regrouper les propriétés d'une même catégorie à l'intérieur d'un bloc. Par exemple, nous pouvons mettre dans un même bloc les propriétés `font-size` et `font-family`, comme ceci :

```
1 figcaption {
2   font: {
3     family: "Grand Hotel", serif;
```

```
4     size: 1.2rem;
5   }
6   margin-top: 1rem;
7 }
```



faites bien attention au deux-points (:) après `font`

Voilà pour le principe de base de l'imbrication. Passons maintenant à quelques cas particuliers.

2.2. La référence au parent

On n'a pas encore réussi à imbriquer le sélecteur `figcaption:before`. Vous pensez que le code suivant marchera ?

```
1 figcaption{
2   font: {
3     family: "Grand Hotel", serif;
4     size: 1.2rem;
5   }
6   margin-top: 1rem;
7   :before{
8     content: '\2012 ';
9   }
10 }
```

En réalité, on obtiendra le sélecteur `figcaption :before` (avec une espace) qui ne veut rien dire. Pour corriger cela, on va utiliser le sélecteur `&`, qui *fait référence au parent*. Ainsi, ce code fonctionnera :

```
1 figcaption{
2   font: {
3     family: "Grand Hotel", serif;
4     size: 1.2rem;
5   }
6   margin-top: 1rem;
7   &:before{
8     content: '\2012 ';
9   }
10 }
```

II. Les bases de Sass

Que s'est-il passé ? Avec le sélecteur `&` (l'esperluette), on a indiqué à Sass où il devait insérer le sélecteur du bloc parent. Dans notre cas, comme il n'y a pas d'espace entre `&` et `:before`, il n'y en aura pas après compilation entre `figcaption` et le pseudo-élément.

On peut faire de même avec `blockquote` :

```
1 blockquote{
2     font-style: italic;
3     &:before, &:after{
4         content: "'";
5     }
6 }
```

Il est aussi possible de s'en servir avec une *pseudo-classe* (`&:hover` sélectionne l'élément parent au survol) ou même à la fin d'un sélecteur (`section &` sélectionne l'élément parent lorsqu'il est contenu dans une balise `<section>`). Toujours moins de répétitions, et toujours plus d'imbrications qui rendent le code plus lisible.

2.3. Imbrication de media-queries

Depuis quelque temps et la démocratisation du Responsive Web Design (RWD pour les intimes), les media-queries ont le vent en poupe. Justement, saviez-vous que Sass permet d'imbriquer la directive `@media` ? Intéressons-nous à cet extrait de notre exemple :

```
1 #####description{
2     max-width:22rem;
3 }
4 @media all and (min-width: 740px){
5     #description{
6         max-width:48%;
7     }
8 }
```

Il est possible d'imbriquer le bloc `@media` à l'intérieur du bloc `#description`. L'intérêt est encore un gain de clarté, puisque l'on regroupe toutes les règles concernant un module (`#description`) dans un même bloc :

```
1 #####description{
2     max-width:22rem;
3     @media all and (min-width: 740px){
4         max-width:48%;
5     }
6 }
```

Voilà, pas grand chose d'autre à dire là-dessus, même si on reparlera un peu des media-queries dans le prochain chapitre.

2.4. Bonne pratique : « The Inception Rule »

L'imbrication est un outil puissant, mais cela ne veut pas dire qu'il faut en abuser, au risque de provoquer une pagaille inimaginable. Vous pourriez être tentés de reproduire dans la feuille de style la structure de la page HTML, un peu comme ceci :

```
1  html{
2      ...
3      body{
4          ...
5          header{
6              #hgroup{
7                  ...
8                  h1{...}
9                  h2{...}
10             }
11             #search{
12                 ...
13                 input{...}
14                 button{
15                     ...
16                     &::after{...}
17                 }
18             }
19         }
20         #presentation{...}
21     }
22 }
```

Sauf que... **c'est mal !** Essayez d'imaginer quels sélecteurs seront présents après la compilation :

```
1  html body header #hgroup #search button::after{
2      ...
3  }
```

Beurk. Une telle méthode pose deux problèmes. Le premier, c'est que l'on perd en partie le principe de la cascade : les sélecteurs sont trop spécifiques, et une modification, même peu importante, du HTML risque d'imposer une modification de la feuille de style. On a donc perdu en maintenabilité. Le second, c'est que, tout bêtement, le fichier `.css` sera inutilement lourd, ce qui est embêtant pour le serveur comme pour les visiteurs.

C'est là qu'intervient ce que l'on appelle « *The Inception Rule* ». La règle est la suivante : *n'imbriguez pas plus de quatre niveaux différents*. C'est une règle arbitraire et contraignante, mais je vous invite à la suivre. Si vous dépassez quatre niveaux, c'est que probablement l'un d'entre eux n'est pas nécessaire. Essayez donc de ne pas trop allonger vos sélecteurs. Il y a souvent moyen de faire plus court.

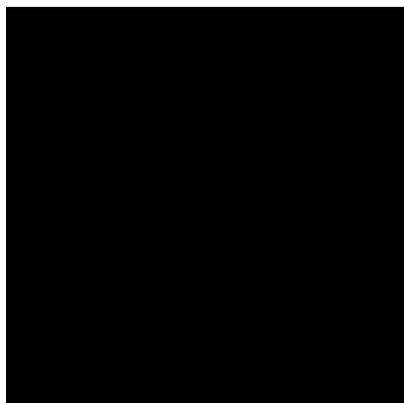


FIGURE 2.2. – Inception

2.5. En résumé

- On peut éviter d'écrire plusieurs fois les mêmes sélecteurs, grâce à l'**imbrication**.
- Lors de l'imbrication, pour **faire référence au bloc parent**, on utilise le sélecteur **&**.
- On peut **imbriquer la directive @media** pour mieux organiser son code.
- Il ne vaut mieux pas dépasser **quatre niveaux d'imbrication**, c'est *The Inception Rule*.

2.6. Pour s'entraîner

Je vous invite à trouver tous les endroits où il est possible d'utiliser l'imbrication dans notre exemple, plu particulièrement en ce qui concerne l'élément `#search`, qui regroupe vraiment tout ce que nous venons de voir (esperluette et media-queries compris). La correction est [ici](#) ↗

Dans le prochain chapitre, nous ferons un sort aux variables, de petites bestioles bien pratiques, héritées des langages de programmation.

3. Les variables

Dans ce chapitre, nous allons aborder un autre aspect fondamental de Sass, les variables.

3.1. Les variables, pourquoi faire ?

Vous avez sans doute remarqué que, dans un design, certaines valeurs reviennent souvent. Si on observe de près notre fil rouge, on peut retrouver plusieurs fois les mêmes couleurs et polices :

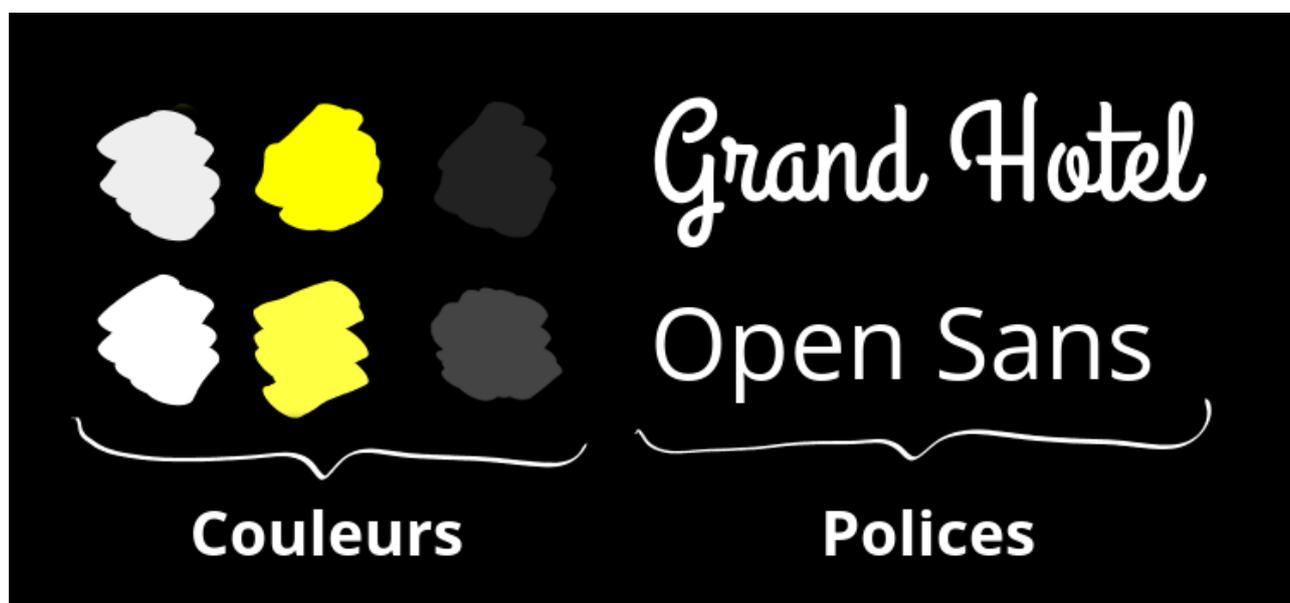


FIGURE 3.1. – Les mêmes couleurs et polices se répètent plusieurs fois dans notre exemple.

Maintenant, imaginez que vous souhaitez changer une de ces valeurs. Vous devez alors effectuer un rechercher-remplacer dans toute votre feuille de style. Mais il y a bien plus simple, vous pouvez utiliser une **variable** !

Une variable, c'est un peu comme une enveloppe. On stocke à l'intérieur une valeur (par exemple, la couleur `#fff0`) et on met une étiquette dessus (c'est le nom de la variable). Ensuite, lorsqu'on a besoin de la valeur de la variable, on a juste à donner son nom, et Sass ira lui-même chercher la valeur à notre place.



FIGURE 3.2. – Une variable a un nom et contient une valeur.

Comment allons-nous dire à Sass qu'il doit mettre la valeur `#ff0` dans la variable `$color`? Ce n'est pas très compliqué, en fait, il suffit d'insérer le code suivant au début de notre feuille de style :

```
1 // À insérer au-début de main.scss
2 $color: #ff0;
```

On dit qu'on a *déclaré* une variable. Le nom de la variable commence nécessairement par `$`. Il est suivi de `:` puis de la valeur que l'on veut donner à notre variable.

Vous remarquez que j'ai écrit dans le code un *commentaire* précédé par `//`. C'est un commentaire d'une ligne, spécifique à Sass, qui n'apparaîtra pas dans le fichier CSS compilé.

Maintenant, on peut utiliser notre variable dans le reste du code, par exemple pour le titre principal :

```
1 #####hgroup {
2     float: left;
3     h1 {
```

```
4   color: $color;  
5   font-size: 2em;  
6   line-height: 1em;  
7   }  
8   ...  
9 }
```

À la compilation, Sass va remplacer le nom `$color` par la valeur `#ff0`. On peut aussi utiliser `$color` pour l'arrière-plan des boutons (`#production .more`, `#contact button`). Désormais, si on ajoute un élément qui utilise cette couleur, on pourra faire appel à notre variable.

Imaginons maintenant que notre client producteur de boissons gazeuses décide de changer la charte graphique : on va prendre du vert, ça fait plus bio ! Il suffit pour cela de changer la valeur de `$color` :

```
1 $color: #af0;
```



FIGURE 3.3. – Greenwashing

Vous l'aurez compris, l'intérêt des variables est avant tout de rassembler au même endroit les valeurs (couleurs, polices, etc.) qui reviennent dans notre feuille de style, afin de faciliter leur modification : plus besoin de parcourir l'ensemble du code à la recherche de chaque occurrence, tout est stocké dans une sorte de palette au début du fichier.

Voilà donc pour l'utilisation basique des variables avec Sass, voyons maintenant plus en détail tout ce que l'on peut mettre dedans.

3.2. Les différents types de données

Il existe quatre principaux types de données que l'on peut stocker à l'intérieur d'une variable : les couleurs, les nombres, les chaînes de caractères et les listes.

II. Les bases de Sass

Nous avons déjà vu comment cela fonctionnait pour **les couleurs**. Elles peuvent être nommées (par exemple `yellow`), ou écrites sous une forme hexadécimale (`#ff0`, `#ffff00`), mais aussi en RGB (`rgb(255,255,0)`) ou RGBA (`rgba(255,255,0,0.5)`), voire même en [HSL](#) / [HSLA](#).

Passons maintenant aux **nombre**s. Les nombres peuvent ne pas avoir d'unité (par exemple `255`) ou en avoir une (par exemple `1em` ou `16px`). On peut s'en servir pour contenir la fonte des caractères (leur taille), la largeur ou la hauteur d'un bloc, la dimension des marges, etc. Très clairement, je ne vois pas d'exemple intéressant dans notre fil rouge, mais ce n'est pas bien compliqué à imaginer :

```
1 //Exemple bidon
2 $largeur: 800px;
3 body {
4   width: $largeur;
5 }
```

On peut effectuer des calculs avec les nombres : addition `+`, soustraction `-`, multiplication `*`, division `/` et modulo `%` (reste de la division euclidienne). Cela pourra être utile plus tard, mais pour l'instant, je vous offre un autre exemple sorti de nulle part :

```
1 //Autre exemple bidon
2 $var: 15px;
3 p {
4   font-size: $var + 5px; // = 20px
5   width: $var * (5+5) - 50px; // = 100px
6 }
```



Petite précision concernant la division. Si vous écrivez juste `font-size: 40px/2;`, Sass n'effectuera pas la division (il laissera ce code dans le CSS). Pour qu'il fasse le calcul, vous devez ajouter des parenthèses : `font-size: (40px/2);`. Ce problème ne se pose pas si vous utilisez des variables (`$var/2`) dans la division, ou si le calcul est composé de plusieurs opérations (`40px/2 + 5px;`).

Pour votre culture, sachez que Sass permet aussi d'effectuer des opérations sur les couleurs, mais on s'en sert assez peu dans la pratique. Nous verrons plus tard que pour modifier une couleur, il existe des *fonctions* spécialisées.

Après cette parenthèse, passons au troisième type : **les chaînes de caractères** (le texte). On peut stocker beaucoup de chose dans une chaîne de caractères : une police, une URL, ou n'importe quelle valeur composée de lettres et de chiffres. Elle peut être encadrée par des guillemets (par exemple `"Open Sans"`) ou non (par exemple `bold`). Prenons notre fil rouge, et ajoutons une variable pour la police de caractère des titres :

```
1 $head-font: "Grand Hotel";
```

Dans la suite du document, on peut faire appel à cette variable par un simple :

```
1 h1, h2 {  
2   font-family: $head-font;  
3 }
```

Alors j'avoue, j'ai un peu triché. Ce code n'est pas exactement l'équivalent du CSS que nous avons avant, puisque nous ne transmettons qu'une seule police à notre règle. Or, il est souvent conseillé de proposer plusieurs polices, séparées par des virgules, dans le cas où la première n'est pas chargée.

Sass nous offre la possibilité de gérer cette situation avec un quatrième type : **les listes**. Une liste est un ensemble de valeurs séparées par des virgules ou des espaces. Il peut s'agir d'une liste de chaînes de caractères, comme dans notre cas ("Open Sans", Helvetica, Arial, sans-serif), d'une liste de nombres (1rem 0), ou bien encore d'une liste de listes (box-shadow 1s, background-color 1s). Notez qu'on peut tout à fait avoir différents types de données dans une même liste.

Pour notre fil rouge, on gèrera les polices avec deux listes de chaînes de caractères :

```
1 $head-font: "Grand Hotel", serif;  
2 $body-font: "Open Sans", Helvetica, Arial, sans-serif;
```

i

On découvrira trois autres types de données au cours des prochains chapitres :

- les *booléens*, qui peuvent valoir `true` (vrai) ou `false` (faux),
- le type *null* qui représente une valeur vide,
- le type *map* qui est une sorte de liste dont chaque item a un nom (sa clef) et une valeur.

3.3. L'interpolation

Jusqu'à présent, on a vu comment utiliser une variable en tant que valeur passée à une propriété. Cependant, il peut arriver qu'on veuille insérer la valeur d'une variable autre part dans le code : dans un sélecteur, dans le nom d'une propriété, ou en tant que media-query.

Justement, revenons à notre fil rouge et intéressons-nous aux media-queries. Dans le chapitre précédent nous les avons imbriquées dans d'autres blocs. Or on remarque dans notre code qu'il y en a deux qui reviennent :

II. Les bases de Sass

```
1 @media screen and (max-width: 540px) {...} // pour les petits
   écrans
2 @media screen and (min-width: 740px) {...} // pour les grands
   écrans
```

On peut donc facilement gérer cela en passant une variable entre parenthèses :

```
1 // On stocke la règle dans une variable
2 $large-screen: "min-width: 740px";
3 // On l'insère entre parenthèses en tant que chaîne de caractères
4 ####description {
5     max-width: 22rem;
6     @media screen and ($large-screen) {
7         max-width: 48%;
8     }
9 }
```

Mais on pourrait aller plus loin en stockant l'intégralité de nos media-queries dans des variables, sous la forme de chaînes de caractère :

```
1 $small-screen: "screen and (max-width: 540px)";
2 $large-screen: "screen and (min-width: 740px)";
```

Comment insérer ces chaînes de caractères ? Le code suivant ne fonctionne malheureusement pas :

```
1 ####description {
2     max-width: 22rem;
3     @media $large-screen {
4         max-width: 48%;
5     }
6 }
```

On va devoir utiliser une syntaxe particulière, l'**interpolation**. Le principe est assez simple : on place le nom de la variable entre `#{` et `}`. Ainsi, le code suivant aura l'effet escompté :

```
1 ####description {
2     max-width: 22rem;
3     @media #{ $large-screen } {
4         max-width: 48%;
5     }
```

```
6 }
```

Personnellement, je trouve qu'on gagne déjà un peu en lisibilité, puisque, intuitivement, cela veut dire :

▮ Ce qui se trouve dans ce bloc est destiné aux écrans larges.

C'est tout pour l'interpolation, qui pourra aussi vous être utile si vous souhaitez modifier un sélecteur, et, plus rarement, le nom d'une règle. Retenez bien sa syntaxe, on en reparlera dans plusieurs chapitres.

i

Parlons au passage rapidement de la **concaténation**. Il s'agit tout simplement de la possibilité d'ajouter deux chaînes de caractère, l'une après l'autre, avec l'opérateur `+` :

```
1 font-family: sans- + "serif";
```

Rien de bien compliqué, mais il est possible que je m'en serve à un moment ou un autre dans un exemple.

3.4. La règle `!default`

Actuellement vous n'en verrez pas forcément l'intérêt, mais je souhaite tout de même vous parler de `!default`. Lorsque vous attribuez une valeur à une variable, il est possible que cette variable existe déjà. Dans ce cas là, la nouvelle valeur remplace l'ancienne. C'est le comportement par défaut. Pour que la variable garde sa valeur d'origine si elle existe déjà, alors il faut ajouter `!default`, comme ceci :

```
1 $macouleur: orange;  
2 $macouleur: skyblue !default;  
3 h1 {  
4   color: $macouleur; // orange;  
5 }
```

Je n'insiste pas trop là-dessus, car je ne vois pas d'application concrète à votre niveau. Sachez juste que cela existe.

3.5. En résumé

— Une variable a un nom et permet de **stocker** une valeur.

II. Les bases de Sass

- On **déclare** une variable ainsi : `$nom: valeur;`. Ensuite, pour accéder à la valeur, on donne le nom de la variable.
- Il existe 4 principaux **types de données** : les nombres, les chaînes de caractères, les couleurs et les listes.
- Pour insérer une variable dans le nom d'une propriété, dans un sélecteur, ou en tant que media-query, on utilise la syntaxe d'interpolation `#{ $variable }`.

i

Il resterait d'autres choses à dire sur le système de variables de Sass. Je ne vous ai pas parlé des types Booléen et Null parce que nous en reparlerons dans d'autres chapitres, lorsqu'ils seront réellement utiles. Sinon, il faudrait aussi évoquer la portée des variables et la règle `!default`, mais ça me semble assez anecdotique actuellement, je vous invite à vous référer à la doc si cela vous intéresse.

3.6. Pour s'entraîner

Déclarez les variables suivantes au début de la feuille de styles, puis utilisez-les à chaque fois que c'est nécessaire :

```
1 //Couleurs
2 $color: #ff0;
3 $color-hover: #ff4;
4 $dark: #222;
5 $dark-hover: #444;
6 $light: #eee;
7 $ultra-light: #fff;
8
9 //Polices
10 $head-font: "Grand Hotel", serif;
11 $body-font: "Open Sans", Helvetica, Arial, sans-serif;
12
13 //Media-queries
14 $small-screen: "screen and (max-width: 540px)";
15 $large-screen: "screen and (min-width: 740px)";
```

La solution est [disponible ici](#) ↗ .

Lorsque vous aurez fini, nous pourrons passer à `@import`, pour voir comment Sass améliore l'import de feuilles de styles.

4. La directive `@import`

La directive `@import`, vous la connaissez probablement déjà : c'est elle qui permet d'indiquer au navigateur qu'il doit charger d'autres feuilles de styles, à la manière de la balise `<link>` en HTML. Si je vous en parle dans ce chapitre, c'est parce que son comportement change considérablement avec Sass, et c'est tant mieux.

4.1. Importer des feuilles de style

Durant le processus de développement, il est souvent intéressant de diviser son code en plusieurs fichiers, pour mieux s'organiser. Mais, d'un autre côté, cela augmente le nombre de requêtes effectuées vers le serveur de production, et ralentit donc le chargement de la page.

Avec Sass, on n'a plus à se poser la question : notre outil favori modifie le comportement de la directive `@import` : en fait, plutôt que de laisser le navigateur charger une à une les feuilles de styles importées, il les importe dès à la compilation. Ainsi, on a plusieurs fichiers SCSS, pour une meilleure organisation, mais on n'a qu'un fichier CSS, pour plus de performance.

Regardons de plus près notre feuille de style fil rouge : j'ai fait exprès de séparer les styles en différentes sections, pour garder un peu de clarté : on a ainsi la section *configuration* contenant toutes nos variables, la section *Reset*, la section *typographie*, la section *boutons*, etc.

Ce serait plus clair si chaque section avait son propre fichier, non ? Je vous propose donc de créer un fichier pour chacune d'entre elles, de manière à obtenir cette arborescence :

```
1 sass
2 | sections.scss
3 | button.scss
4 | cards.scss
5 | contact.scss
6 | header_footer.scss
7 | reset.scss
8 | typo.scss
9 | config.scss
10 | main.scss
```

Placez dans chaque fichier le contenu de la section correspondante. Maintenant, on devrait se retrouver avec un `main.scss` totalement vide. Que va-t-on utiliser désormais ? Mais oui, la commande `@import` !

```
1 // Fichier main.scss
2 @import 'reset'; // en premier, évidemment
3 @import 'config'; // important de le mettre ici, sinon les autres
4                       // fichiers n'auront pas accès à nos variables
5 @import 'buttons';
6 @import 'typo';
7 @import 'cards';
8 @import 'header_footer';
9 @import 'sections';
10 @import 'contact';
```

Comme vous pouvez le voir, la directive `@import` demande le nom du fichier à importer entre guillemets. Détail intéressant : Sass est suffisamment intelligent pour deviner l'extension du fichier et on n'a donc pas à la préciser. Si vous regardez désormais dans le fichier `stylesheets/main.css`, vous verrez que son contenu n'a pas changé : chaque directive `@import` a été remplacée par le contenu du fichier correspondant.

4.2. Bonne pratique : Les feuilles partielles

Formidable ? Un petit détail vient quand même assombrir le tableau : Sass a généré un fichier CSS pour chaque fichier SCSS. Or, on souhaite seulement qu'il génère le fichier `main.css`. En fait, il faudrait pouvoir dire à Sass de ne pas compiler les autres feuilles de styles. Et, comme vous pouvez vous en douter, les concepteurs de Sass y ont pensé pour nous.

On appelle **feuille partielle** (traduction de *partial*, en anglais), un fichier SCSS qui a uniquement vocation à être importée dans une autre feuille de styles et qui ne doit donc pas être compilée par Sass. Pour indiquer à Sass qu'un fichier est une feuille partielle, on a juste à ajouter un *underscore* (le symbole `_`) au début du nom de fichiers. Ainsi, `buttons.scss` devient `_buttons.scss`. Aucun changement à faire dans notre fichier `main.scss` pour autant, Sass devine qu'il doit ajouter le `_`.

Si vous regardez dans notre dossier `stylesheets` vous devriez remarquer le changement : Sass a supprimé tous nos fichiers, à l'exception de `main.css`, car il a compris qu'on n'avait besoin que de ce dernier.

Dernière chose à savoir : il est courant de regrouper toutes ses feuilles partielles dans un sous-dossier `partials`, ça permet de mieux s'y retrouver. Je vous invite à le faire aussi pour notre projet. On se retrouve donc avec cette arborescence :

```
1 sass
2   └─ partials
3     └─ _sections.scss
4     └─ _buttons.scss
5     └─ _cards.scss
```



N'oubliez pas bien sûr de modifier le fichier *main.scss* en conséquence pour indiquer le nouveau chemin de nos partials (mais toujours pas besoin du `_` ou de `.scss`, évidemment).

4.3. En résumé

- Avec Sass, la directive `@import` permet d'importer durant la compilation le contenu d'un fichier SCSS dans un autre fichier SCSS.
- Il est inutile de préciser l'extension du fichier importé à Sass.
- Une **feuille partielle** ou *partial*, dont le nom commence par un *underscore*, est un fichier qui a uniquement vocation à être importé dans d'autres feuilles de styles. Aucun fichier CSS n'est généré pour lui à la compilation.

5. Les mixins (1/2)

Si une variable est bien utile pour stocker une valeur, que faut-il utiliser pour stocker un bloc de code entier afin de faciliter sa réutilisation ? Un mixin, évidemment !

Les mixins sont un gros morceau de Sass, que j'ai divisé en deux chapitres. Dans celui-ci, nous allons voir à quoi ils servent, puis nous allons en créer un et l'utiliser, et nous verrons enfin comment lui ajouter des arguments.

5.1. Un mixin, pourquoi faire ?

Intéressons-nous à nos boutons. En soi, ils sont assez banals. Si banals que, à l'exception de la couleur et de la police, on devrait pouvoir les réutiliser sur beaucoup de projets. Il serait donc pas mal de pouvoir emballer le code de nos boutons, pour le réutiliser plus tard, tout en se laissant la possibilité de changer certaines options.

Ce paquet a un nom, c'est un **mixin**. On peut le comparer à une *macro* : une macro est un ensemble d'instructions à effectuer, un mixin est un ensemble de styles à appliquer à un élément. On reste dans l'idée *Don't repeat yourself*.

Je vous propose de construire notre mixin **button** pas à pas, à l'intérieur de la feuille partielle `_button.scss`. La première étape consiste à mettre les styles communs à tous nos boutons dans un bloc précédé de la directive `@mixin` et du nom que l'on souhaite donner à notre mixin. Voici donc le code de notre mixin :

```
1 @mixin button {
2   font: {
3     family: $head-font;
4     size: 1.25rem;
5   }
6   text: {
7     decoration: none;
8     align: center;
9   }
10  padding: .5rem;
11  width: 12.5rem;
12  border: none;
13  display: block;
14  float: none;
15  margin: .5rem auto 0;
16  @media #{$large-screen}{
```

```
17     display: inline-block;
18     float: right;
19   }
20 }
```

Si vous tentez de compiler maintenant, ce code ne devrait générer aucune règle. Pourquoi ? Parce qu'on n'a pas dit à Sass où il doit insérer le contenu de notre mixin. C'est un peu comme pour les variables : on doit d'abord les déclarer pour pouvoir ensuite les inclure dans notre code.

5.2. Inclure un mixin

Il nous reste donc à inclure notre mixin. Pour cela, on se sert de la directive `@include` suivie du nom du mixin, comme ceci :

```
1  #####produits .more, #clients .more {
2    @include button;
3    background-color: $dark;
4    color: $ultra-light;
5    &:hover {
6      background-color: $dark-hover;
7    }
8  }
9  #####production .more, #contact button {
10   @include button;
11   background-color: $color;
12   color: $dark;
13   &:hover {
14     background-color: $color-hover;
15   }
16 }
```

Maintenant Sass comprend qu'il faut inclure le contenu du mixin **button** avant la compilation vers CSS. On peut inclure autant de mixins que l'on veut dans un même bloc. Ainsi, le code suivant est tout à fait envisageable :

```
1 //Code bidon
2 h1{
3   @include mandarine;
4   @include clementine;
5   @include citron;
6   @include button;
7 }
```

Voilà, vous savez désormais inclure un mixin, reste à voir comment le paramétrer.

Vous noterez cependant que le code généré n'est pas exactement celui que l'on avait auparavant. En effet, le contenu du mixin a été inclus *deux fois*, d'abord pour `#produits .more`, `#clients .more`, et ensuite pour `#production .more`, `#contact button`. Cela crée une répétition dans notre code CSS. On pourrait croire que cela alourdit le fichier envoyé au client et rallonge donc le temps de chargement. Sauf que... pas vraiment.

En fait, il est fort probable que vous utilisiez **gzip** sur votre serveur pour réduire la taille des fichiers envoyés à vos visiteurs (si ce n'est pas le cas, vous devriez¹). Or l'un des mécanismes principaux de gzip consiste à supprimer toutes les répétitions (en les remplaçant par des pointeurs vers la première occurrence). Le fichier CSS généré par Sass sera donc probablement plus lourd, mais ce ne sera pas le cas du fichier envoyé à votre visiteur.

5.3. Les arguments

Pour l'instant, les règles concernant les couleurs de nos boutons sont en-dehors de notre mixin, puisque ce sont elles qui changent selon l'utilisation. Pour les insérer dans le mixin `button`, on va devoir utiliser des **arguments**. Un argument, c'est une variable à l'intérieur d'un mixin, qui peut prendre une valeur différente à chaque directive `@include`. C'est un peu comme les options de commande sur une livraison.

Pour notre mixin, il nous faudrait donc trois arguments :

- `$color` : la couleur du texte du bouton ;
- `$background-color` : la couleur de l'arrière-plan du bouton ;
- `$hover-color` : la couleur de l'arrière-plan au hover.

La première étape va donc consister à écrire les règles dans notre mixin, en leur passant pour valeur nos trois arguments, comme n'importe quelles variables :

```
1 @mixin button {
2   font-family: $head-font;
3   font-size: 1.25rem;
4   text-decoration: none;
5   text-align: center;
6   padding: .5rem;
7   width: 12.5rem;
8   border: none;
9   display: block;
10  float: none;
11  margin: .5rem auto 0;
12  background-color: $background-color;
13  color: $color;
14  &:hover {
15    background-color: $hover-color;
16  }
17  @media #{$large-screen} {
```

1. Pour vérifier que gzip est bien configuré, il y a [ce test](#) .

```
18     display: inline-block;
19     float: right;
20   }
21 }
```

Maintenant, il faut déclarer ces arguments, et pour cela, on va ajouter leurs noms entre parenthèses après le nom du mixin. Si vous avez déjà fait de la programmation, cette syntaxe devrait vous rappeler les arguments d'une fonction.

```
1 @mixin button ($color, $background-color, $hover-color) {
2     ...
3 }
```

Pour finir, doit renseigner la valeur de ces arguments à chaque `@include`, en écrivant leurs valeurs entre parenthèses, **dans le même ordre** que précédemment :

```
1 #####produits .more, #clients .more {
2     @include button($ultra-light, $dark, $dark-hover);
3 }
4 #####production .more, #contact button {
5     @include button($dark, $color, $color-hover);
6 }
```

Voilà, ce code nous économise de recopier plusieurs fois les mêmes règles lorsque seules les valeurs changent. Avec le même mixin, on obtient deux boutons de même forme mais de couleurs différentes.

5.4. En résumé

- Un mixin est un **bout de code réutilisable**.
- Il se **déclare** ainsi :

```
1 @mixin nom-du-mixin {
2     //Code à réutiliser
3 }
```

- A chaque fois qu'on veut l'utiliser, on se sert de la directive `@include nom-du-mixin`.
- Les **arguments** sont les options du mixin. On s'en sert comme ceci :

II. Les bases de Sass

```
1 @mixin nom-du-mixin($arg1, $arg2) {  
2   color: $arg1;  
3   font-size: $arg2;  
4 }  
5 // Plus loin  
6 p {  
7   @include nom-du-mixin(red, 22px);  
8 }
```

Je vous donne rendez-vous dans le prochain chapitre pour parler... des mixins, parce qu'on n'a pas encore tout vu.

6. Les mixins (2/2)

Les mixins, ça continue! On va parler de quelques trucs au sujet des mixins qui peuvent vous faciliter la vie.

6.1. Valeurs par défaut, arguments nommés et listes d'arguments

Tout d'abord, parlons arguments. Les arguments, on l'a vu, permettent de changer certaines valeurs à l'intérieur de notre mixin. Il arrive cependant qu'un mixin soit souvent utilisé avec les mêmes valeurs (mais pas toujours, sinon on ne voit pas pourquoi on a créé des arguments). Dans ce cas on peut attribuer des valeurs par défaut à nos arguments. Si l'on fournit une valeur à l'inclusion, le mixin l'utilisera, sinon, il se servira de la valeur par défaut. Une valeur par défaut se déclare à la création du mixin, comme ceci :

```
1 @mixin button($color: $dark, $background-color: $color,  
2     $hover-color: $color-hover) {  
3     ...  
}
```

Dans ce cas, il est possible d'omettre les valeurs de tous les arguments, ou de certains arguments.

Ainsi :

```
1 @include button;  
2 // Correspondra à :  
3 @include button($dark, $color, $color-hover);  
4 // Tandis que :  
5 @include button(#000);  
6 // Correspondra à :  
7 @include button(#000, $color, $color-hover);
```

Vous remarquez que lorsque seulement certaines valeurs sont données à `@include`, Sass "remplit" les arguments de gauche à droite : on ne peut pas renseigner une valeur pour `$hover-color`, sans donner au préalable une valeur à `$color`. L'ordre dans lequel on déclare les arguments dans l'entête du mixin a donc une importance.

Une autre syntaxe permet cependant de contourner cette limite, **les arguments nommés**. En effet, il est possible d'utiliser les noms des mixins lors de l'inclusion :

```
1 @include button($color:#000, $background-color:#eee, $hover-color:
  #ccc);
2 @include button($hover-color: #eee, $color: #000);
3 @include button($hover-color: #ccc);
```

Analysons ces trois lignes :

- Dans la première, les arguments `$color`, `$background-color` et `$hover-color` prennent respectivement les valeurs `#000`, `#eee` et `#ccc`.
- Dans la deuxième, les arguments `$color` et `$hover-color` prennent respectivement les valeurs `#000` et `#eee`, tandis que `$background-color` garde sa valeur par défaut (si on n'avait pas donné de valeur par défaut à la création du mixin, Sass aurait signalé une erreur).
- Dans la troisième, on donne juste la valeur `#ccc` à `$hover-color`, les autres arguments se voient attribués leurs valeurs par défaut.

Pourquoi utiliser les arguments nommés (*keyword arguments* en anglais) ? Certes, c'est plus long, mais on gagne probablement un peu en lisibilité. De plus, on ne se pose plus la question de l'ordre des arguments : cela permet, par exemple, de n'attribuer une valeur spécifique qu'au dernier argument, comme on le fait ici. Cela n'aurait pas été possible sans nommer les arguments.

Dernière chose à voir concernant les arguments : le cas particulier des listes. Imaginons que l'on souhaite rajouter un argument `$font` à notre mixin de bouton, pour pouvoir paramétrer la police utilisée.

```
1 @mixin button($color, $background-color, $hover-color, $font) {
2   font-family: $font;
3   ...
4 }
```

Habituellement, on utilise donne une liste de polices. Si on passe notre variable `$head-font` (qui est une liste de chaînes de caractères) à l'inclusion, tout se passe comme prévu :

```
1 @include button($dark, $light, $ultra-light, $head-font);
```

Cependant, imaginons un cas exceptionnel pour lequel notre liste de polices n'est pas stockée dans une variable, comment les passer à nos mixins ?

```
1 @include button($dark, $light, $ultra-light, "Source Sans Pro",
  "Segoe UI", "Trebuchet MS", Helvetica, "Helvetica Neue", Arial,
  sans-serif);
```

II. Les bases de Sass

Si vous essayez la ligne précédente, vous allez obtenir une erreur, et c'est logique : Sass attend 4 arguments, et il en reçoit 10. Deux solutions s'offrent en fait à nous. On peut tout d'abord encadrer notre liste de polices par des parenthèses pour faire comprendre à Sass qu'il s'agit d'un seul argument :

```
1 @include bouton($dark, $light, $ultra-light, ("Source Sans Pro",  
    "Segoe UI", "Trebuchet MS", Helvetica, "Helvetica Neue", Arial,  
    sans-serif));
```

Sinon, on peut utiliser la syntaxe des **arguments variables**. En ajoutant `...` après le nom du *dernier* argument lors de la création du mixin, on indique à Sass que cet argument devra contenir la liste de toutes les valeurs passées au mixin qui restent. On ne peut logiquement avoir qu'un seul argument variable par mixin (vu qu'il prend toutes les valeurs restantes). Ainsi, le code suivant marche :

```
1 @mixin bouton($color, $background-color, $hover-color, $font...) {  
2     font-family: $font;  
3     ...  
4 }  
5  
6 #####bouton {  
7     @include bouton($dark, $light, $ultra-light, "Source Sans Pro",  
    "Segoe UI", "Trebuchet MS", Helvetica, "Helvetica Neue",  
    Arial, sans-serif);  
8 }
```

Dans cet exemple `$font` contiendra la liste de toutes valeurs restantes à partir de "Source Sans Pro". Cette syntaxe est particulièrement pratique pour des mixins dont on ne connaît pas le nombre d'arguments. C'est le cas par exemple des mixins utilisés pour préfixer certaines propriétés CSS3 comme *box-shadow*, *transition* ou *animation* :

```
1 // Exemple issu de la doc Sass  
2 @mixin box-shadow($shadows...) {  
3     -moz-box-shadow: $shadows;  
4     -webkit-box-shadow: $shadows;  
5     box-shadow: $shadows;  
6 }
```



Les mixins pour préfixer des règles CSS3 ont probablement été l'une des raisons du succès de Sass, à travers des bibliothèques de mixins prêts-à-l'emploi. Aujourd'hui, ce genre de mixin a tendance à être remplacé par des outils comme *Autoprefixer*.

6.2. Passer un bloc de contenu à un mixin

Dans le chapitre sur les variables, je vous ai montré comment l'interpolation pouvait être utilisée avec les media-queries. Un autre moyen de gérer ces dernières passe par l'utilisation de mixins. Observez par exemple ce code :

```
1 @mixin tablette {
2     @media (max-width: 950px) {
3         @content;
4     }
5 }
6
7 h1{
8     font-size: 3.5rem;
9     @include tablette {
10        font-size: 2.5rem;
11    }
12 }
```

Deux détails à remarquer :

- le mixin `tablette` contient la directive `@content`, que nous n'avons pas encore vue;
- l'inclusion du mixin est suivie d'un bloc de code entre accolades.

En fait, lorsque Sass voit un `@content` à l'intérieur d'un mixin, il s'attend à ce qu'on lui passe un bloc de code lors de l'inclusion. Ce bloc de code va prendre la place de la directive `@content` partout où elle est présente dans le mixin inclus. Dans notre exemple, le code CSS généré sera :

```
1 h1 {
2     font-size: 3.5rem;
3 }
4 @media (max-width: 950px) {
5     h1 {
6         font-size: 2.5rem;
7     }
8 }
```

Comme vous pouvez le constater, `@content` a effectivement été remplacé par `font-size: 2.5rem`. Si on avait passé plusieurs règles à notre mixin, elles auraient toutes été insérées à l'emplacement de `@content`.

Vous l'aurez compris, on peut tout à fait remplacer nos variables `$small-screen` et `$large-screen` par des mixins et employer `@include` à la place de l'interpolation. On obtiendrait ce genre de code :

```
1 @mixin small-screen {
2     @media screen and (max-width: 540px) {
3         @content;
4     }
5 }
6 @mixin large-screen {
7     @media screen and (min-width: 740px) {
8         @content;
9     }
10 }
```

Les deux techniques se valent à mon avis pour ce qui est des media-queries. À vous de choisir celle qui vous convient le mieux.

6.3. Bonne pratique : Arguments & type null



Le contenu de cette sous-partie s'inspire de l'article [Smarter Sass Mixins with Null\(en\)](#) de Guil Hernandez.

Imaginons maintenant que l'on veuille un bouton qui ne change pas de couleur au hover. Le seul moyen qu'on a d'obtenir cela consiste à passer la même valeur pour `$background-color` et `$hover-color`. Sauf que c'est un peu idiot, parce que le code concernant `$hover-color` va quand même être généré, alors qu'il ne sert à rien.

Il faudrait pouvoir dire à Sass que `$hover-color` est optionnel et que s'il n'est pas renseigné, on doit ignorer la règle concernant le hover. Et ça tombe bien, parce qu'on peut faire cela avec un **argument de type null**.

Je l'avais très rapidement évoqué dans le chapitre sur les variables : le type `null` permet d'indiquer à Sass qu'une variable ne contient rien. La seule valeur que peut prendre une variable de ce type est `null` :

```
1 $variable: null;
```

Vous vous demandez sans doute à quoi ça peut bien servir. En fait, il faut savoir que lorsqu'une propriété reçoit la valeur `null`, elle est ignorée par Sass et n'apparaît pas dans le code compilé. Ainsi, le code suivant n'apparaîtra pas après compilation :

```
1 $variable: null;
2 p {
3     color: $variable;
```

II. Les bases de Sass

```
4 }
```

Si un argument reçoit la valeur `null` et qu'on passe cet argument à une propriété, la propriété est donc exclue du fichier CSS. Mieux, si on attribue `null` comme valeur par défaut à un argument et qu'on passe cet argument à une propriété à l'intérieur du mixin, alors la propriété sera exclue du fichier CSS à moins que l'on donne explicitement une valeur à l'argument lors de l'`@include`. Modifions donc la valeur par défaut de `$hover-color` :

```
1 @mixin button($color, $background-color, $hover-color: null) {  
2     ...  
3 }
```

Désormais, si on inclut notre mixin sans préciser de valeur pour `$hover-color`, on obtiendra un bouton sans hover :

```
1 .my-button {  
2     font-family: "Grand Hotel", serif;  
3     font-size: 1.25rem;  
4     text-decoration: none;  
5     text-align: center;  
6     padding: .5rem;  
7     width: 12.5rem;  
8     border: none;  
9     display: block;  
10    float: none;  
11    margin: .5rem auto 0;  
12    background-color: #ff0;  
13    color: #222;  
14 }  
15 @media screen and (min-width: 740px) {  
16     .my-button {  
17         display: inline-block;  
18         float: right;  
19     }  
20 }
```

La valeur `null` est donc particulièrement utile pour rendre certains arguments optionnels sans générer de code inutile.



Petite limite quand même : cette technique ne marche pas si on effectue des calculs avec cet argument. En effet, toute opération mathématique impliquant la valeur `null` provoquera une erreur.

6.4. En résumé

- On peut définir des **valeurs par défaut** pour les arguments de notre mixin, comme ceci :

```
1 @mixin nom_du_mixin($argument: "Valeur par défaut") {...}
```

- Il est possible de **nommer** explicitement les arguments à l'inclusion du mixin :

```
1 @include nom_du_mixin($nom_argument: valeur);
```

- Si le nom du dernier argument déclaré dans la directive `@mixin` est suivi de `...`, alors cet argument prendra pour valeur **la liste de toutes les valeurs restantes** lors de l'`@include`.
- On peut passer un bloc de code à un mixin. Ce code se place entre accolades `{}` après les arguments lors de l'inclusion du mixin. Lors de la création du mixin, on indique où le bloc doit être placé avec la commande `@content;`. Cette technique peut servir afin de gérer les media-queries globalement avec des mixins.
- En attribuant à un mixin la valeur par défaut `null`, on rend cet argument optionnel : les propriétés qui reçoivent cet argument seront ignorées par Sass lors de la compilation et n'apparaîtront pas dans le fichier CSS généré.

7. Les fonctions

Dans le chapitre sur les variables, j'ai évoqué très rapidement les fonctions. Il est temps de revenir plus en détail sur ces petites moulinettes bien pratiques. Dans ce chapitre, nous nous concentrerons sur l'utilisation des fonctions proposées par Sass, avant de voir rapidement comment créer les nôtres.

7.1. Un fonction ?

Si vous avez un peu écouté en Maths au collège et/ou au lycée, vous devriez savoir ce qu'est une fonction mathématique. Du coup j'ai ressorti mon cahier de 3^e :

Définition : Une fonction est un procédé par lequel on associe à un nombre donné (initial) un seul nombre correspondant.

Pour Sass, c'est un peu la même chose : on passe des données à la fonction (l'équivalent du nombre initial en Maths), elle fait ses calculs, et elle *renvoie* le résultat. Les données initiales, appelées arguments (comme pour les mixins) peuvent être des nombres, des couleurs, des chaînes de caractères, des listes, etc. tout comme le résultat.

i

Si, en plus d'écouter en Maths, vous avez déjà fait de la programmation, non seulement vous êtes formidable, mais en plus vous devriez savoir ce qu'est une fonction pour les concepteurs de Sass.

Un exemple ? Je vous propose d'essayer la fonction `darken`, incluse avec Sass, qui permet d'assombrir une couleur. Elle prend deux arguments : `$color`, la couleur de départ, et `$amount` le pourcentage d'assombrissement. Elle renvoie la couleur assombrie du pourcentage demandé :



FIGURE 7.1. – La fonction `darken` prend deux arguments (`#ff0000` et `10 %`) et renvoie `#cc0000`.

On utilise la fonction `darken` ainsi :

```
1 background-color: darken(#ff0000, 10%);
```

Comme attendu, ce code donne le CSS suivant :

```
1 background-color: #cc0000;
```

Comme vous pouvez le remarquer, cette syntaxe est assez proche de celle utilisée par les mixins (les arguments entre parenthèses), sauf qu'on n'utilise pas `@include`. D'ailleurs, vous pouvez, comme pour les mixins, nommer vos arguments :

```
1 background-color: darken($color: #ff0000, $amount: 10%);
```

Je pense que vous voyez bien en quoi cela peut nous être utile dans notre fil rouge. La couleur du hover de notre mixin bouton par exemple : il s'agit à chaque fois d'une version légèrement assombrie de la couleur d'arrière-plan de base. On peut tout à fait supprimer cet argument et le remplacer par un `darken($background-color, 10%);`. Notre mixin n'accepte donc désormais plus que deux arguments (ou plus, si vous avez ajouté la liste des polices).

Sass met à votre disposition tout un tas de fonctions très utiles, que nous allons voir plus en détail dans la suite de ce chapitre.

7.2. Quelques fonctions bien utiles

Commençons tout d'abord par voir quelques fonctions très utiles que Sass nous sert sur un plateau. Pour limiter l'effet "liste de courses", je les ai classées en différentes catégories.

7.2.1. Pour définir une couleur

Vous avez sans doute déjà utilisé `rgb()`, `rgba()`, `hsl()` ou `hsla()` pour définir une couleur dans une feuille de style. Sass considère que ce sont des fonctions qui renvoient une couleur. Cela veut dire qu'il les convertira automatiquement au format le plus adapté (le plus court) dans le fichier final.



Si vous ne connaissez pas le modèle **HSL** (Teinte Saturation Lumière), allez donc voir par [ici](#) ou par [là](#).

7.2.2. Pour jouer avec les nombres

De toutes les fonctions que Sass propose concernant les nombres, je pense que `percentage($number)` et `round($number)` sont les deux seules que vous devez retenir.

La première demande un nombre sans unité et retourne un pourcentage, comme dans cet exemple :

```
1 div {
2   width: percentage(0.5); // 50%
3 }
```

Cependant, cela devient vraiment intéressant si vous avez des valeurs en pixels (tirées d'une maquette par exemple), car vous n'avez plus besoin de faire les calculs vous-mêmes :

```
1 div {
2   width: percentage(200px/960px); // 20,8333% (vous aviez prévu de
3     le faire de tête ?)
}
```

Si vous calculez l'ensemble des dimensions de vos blocs avec `percentage()` (en vous basant sur les dimensions dans la maquette statique), vous pouvez obtenir une grille fluide en un rien de temps.

Enfin, la fonction `round()` renvoie l'arrondi à l'unité du nombre donné. Très utile lorsque vous faites des divisions impliquant des pixels :

```
1 small {
2   font-size: round(15px/2); // 8px
3 }
```

Bon, là je vous ai montré leur fonctionnement avec des valeurs inscrites "en dur", mais vous comprenez bien qu'utiliser des fonctions est surtout intéressant avec des variables. Ainsi, si vous changez la couleur principale de votre design, définie dans le fichier *config.scss* (ou quelque soit le nom que vous lui avez donné), les fonctions recalculeront à votre place les couleurs secondaires.

7.3. La palette du peintre

Voyons maintenant quelques unes des très nombreuses fonctions à notre disposition pour modifier les couleurs.

7.3.1. Inverse, Nuance de gris et Mélange

Commençons par celles qui changent radicalement une couleur :

```
1 invert($color);
2 grayscale($color);
3 mix($color1, $color2);
```

Les deux premières prennent une couleur pour seul argument :

- `invert()` renvoie l'inverse ou négatif de la couleur donnée (ex : `#ff0000` devient `#00ffff`),
- `grayscale()` renvoie la nuance de gris correspondant à la couleur, c'est-à-dire son équivalent totalement dé-saturé (ex : `#ff0000` devient `#808080`).

Enfin, la fonction `mix()` prend deux couleurs, et renvoie le mélange des deux :

```
1 h1{
2   color: mix(#ff0000, #0000ff); // Renvoie #7f007f
3   // Rouge + Bleu = Violet
4 }
```

A noter que `mix()` accepte un troisième argument optionnel en pourcentage, `$weight`, qui indique s'il faut plus de la première ou de la deuxième couleur.

7.3.2. Lumière, Saturation, Opacité

Continuons avec des fonctions qui proposent des altérations plus fines de lumière, de saturation ou d'opacité :

```
1 // Lumière
2 lighten($color, $amount); // plus clair
```

II. Les bases de Sass

```
3 darken($color, $amount); // plus sombre
4
5 // Saturation
6 saturate($color, $amount); // plus saturé
7 desaturate($color, $amount); // moins saturé
8
9 // Opacité
10 opacify($color, $amount); // plus opaque
11 transparentize($color, $amount); // moins opaque
```

Nous avons déjà vu `darken()` et, comme vous pouvez le constater, toutes ces fonctions demandent une couleur et un « pourcentage d'efficacité » de la transformation. Je pense que les noms sont assez explicites, donc plutôt que de vous décrire tout en détail, je vous propose plutôt un joli schéma bilan :

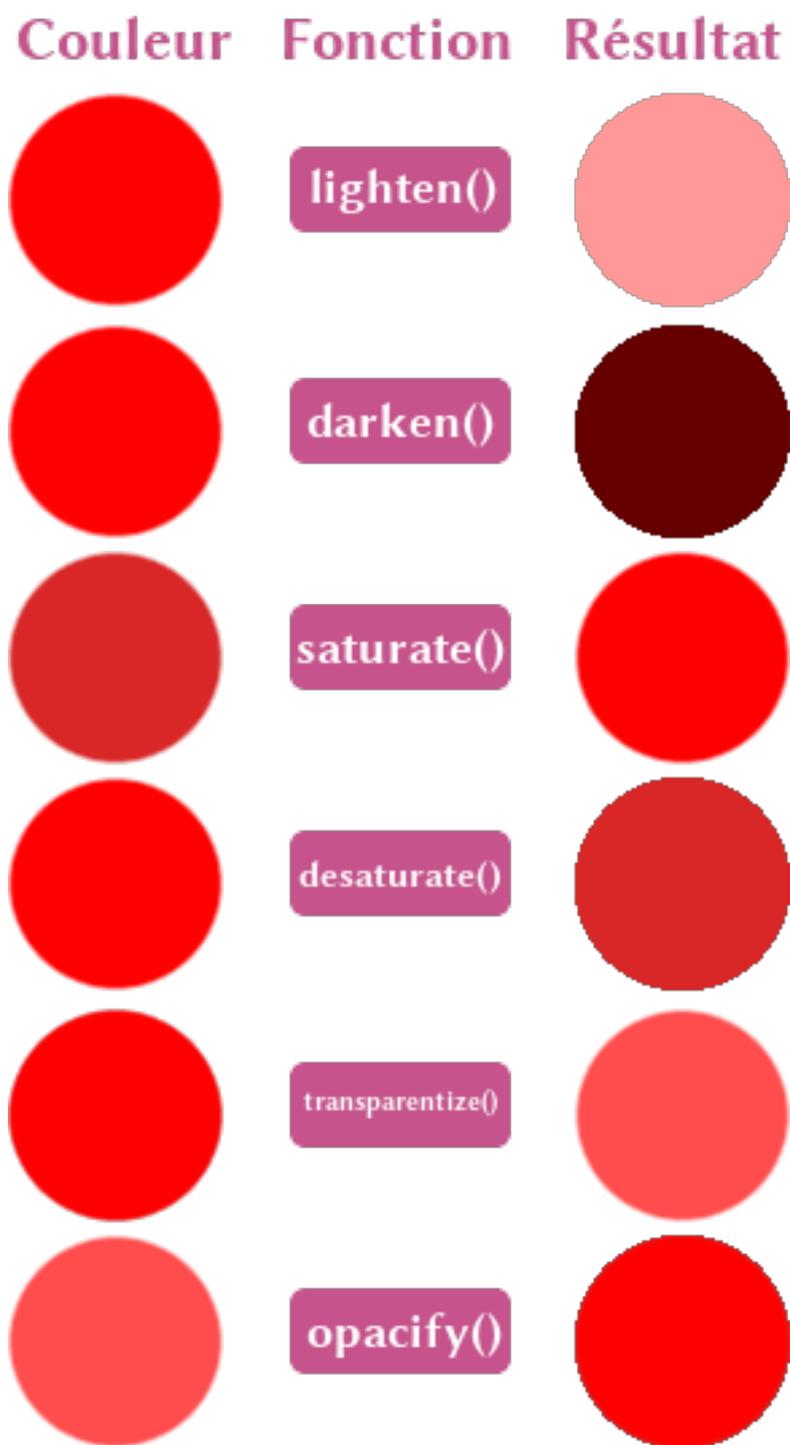


FIGURE 7.2. – Schéma-bilan des fonctions qui modifient une couleur (\$amount : 30%)



Les plus malins d'entre vous auront remarqué que le résultat de la fonction `grayscale()` correspond exactement à celui de `desaturate()` avec un `$amount` de 100%.

7.4. Jouons avec les listes

Avant de finir ce chapitre, je vous propose de regarder quelques fonctions qui permettent de modifier des listes.

La première est `append($list, $val)` et sert à ajouter un élément à la fin d'une liste :

```
1 $liste 1: "Source Sans Pro", Helvetica, Arial;
2 $liste2: append($font-list, sans-serif); // "Source Sans Pro",
   Helvetica, Arial, sans-serif
```

La deuxième est `join($list1, list2)` et assemble deux listes en une seule :

```
1 $liste1: 50px 2em;
2 $liste2: 10px 2em;
3 $liste3: join($liste1, $liste2); // 50px 2em 10px 2em
```



Sachez que `append()` et `join()` acceptent un troisième argument optionnel : `$separator`. Il permet de choisir le séparateur entre les éléments de la liste générée (entrez `comma` si vous voulez des virgules ou `space` si vous voulez des espaces). Par défaut, Sass choisit le séparateur de la/les listes passées en argument(s).

La troisième et dernière fonction est assez particulière mais très puissante. La fonction `zip($lists...)` accepte un ensemble de listes qu'elle va assembler en une unique *liste multidimensionnelle* (une liste de listes). Pour comprendre son fonctionnement, je vous propose d'essayer vous-même ce petit exemple :

```
1 $proprietes: color background-color width;
2 $durees: 1s 2s 10s;
3 $timing: linear ease-in-out ease;
4 div {
5   transition: zip($proprietes, $durees, $timing);
6 }
```

Le code généré par Sass sera le suivant :

```
1 div {
2   transition: color 1s linear, background-color 2s ease-in-out,
   width 10s ease;
3 }
```

II. Les bases de Sass

Comme vous pouvez le constater, `zip()` a renvoyé une liste contenant 3 listes, qui contiennent chacune un élément de chaque liste passée en argument.

?

Et dans la pratique, ça sert à quoi ?

Excellente et pertinente question, brave lecteur ! J'ai eu personnellement à utiliser `join()` dans un cas précis il y a peu, qui pourrait se présenter à vous dans le futur. Bourbon, que nous verrons plus tard, propose la variable suivante :

```
1 $font-stack-georgia: (  
2   "Georgia",  
3   "Times",  
4   "Times New Roman",  
5   serif,  
6 );
```

Elle fait partie de tout un lot de *font stacks*, c'est-à-dire des listes de polices prêtes à l'emploi. Personnellement, je cherchais à utiliser *Bitter*, qui est une très chouette police à empattement. Bourbon ne propose pas de *font stack* pour *Bitter*, mais comme elle se rapproche de *Georgia*, j'ai pu faire ceci, et le tour était joué :

```
1 $bitter: join(Bitter, $font-stack-georgia); // "Bitter", "Georgia",  
   "Times",...
```

À côté de cela, ces trois fonctions (et plus particulièrement `zip()`) se révéleront particulièrement utiles lorsqu'on utilisera des boucles, dans les prochains chapitres. Voyons justement trois autres fonctions (et après, c'est fini, promis) dont nous aurons besoin avec les boucles.

La première, qui n'est pas bien compliquée à appréhender, c'est la fonction `length($list)`. Elle renvoie, comme son nom l'indique, la longueur de la liste, c'est-à-dire le nombre d'éléments contenus dans celle-ci. Ainsi, `length($font-stack-georgia)` renverra 4, car il y a 4 éléments dans cette liste.

Avant d'évoquer les deux autres, il est important que vous compreniez bien à quoi correspond l'indice ou index d'un item. Dans une liste, chaque élément a un index unique. Il s'agit tout bêtement de sa position dans la liste : le premier a l'index 1, le deuxième a l'index 2, etc.

Prenons l'exemple de notre liste `$bitter`. On obtient les couples index-valeur suivants :

Index	1	2	3	4	5
Valeur	Bitter	"Georgia"	"Times"	"Times New Roman"	serif

La fonction `nth($list, $n)` permet de récupérer une valeur de la liste `$list` en donnant son index `$n` :

```
1 font-family: nth($bitter, 2); // "Georgia"
2 font-family: nth($bitter, 9); // Provoque une erreur, il
  n'y a pas assez d'items
3 font-family: nth($bitter, length($bitter)); // serif
```

Comme vous le voyez ligne 3, on peut obtenir le dernier élément d'une liste en combinant `nth()` et `length()`.

La fonction `index($list, $value)` a le comportement exactement inverse. Elle renvoie l'index de la première apparition (en partant de la gauche) de la valeur `$value` dans la liste `$list`. Si la valeur donnée n'existe pas dans la liste, la fonction renvoie `null`. Ainsi, `index($bitter, Georgia)` renverra 2.



Si vous avez déjà fait un peu de programmation, vous remarquerez que, pour Sass, une liste commence à l'index 1 et non à l'index 0, au contraire de la plupart des langages de programmation.

7.5. Créer sa propre fonction



Je cherche une fonction très particulière que tu n'as pas présenté, comment faire ?

Il faut tout d'abord vérifier si cette fonction n'existe pas déjà. En effet, je n'ai pas la place (et ce serait sans intérêt) de décrire toutes les fonctions que Sass propose ici. Allez donc faire un tour sur [la page de la documentation consacrée aux fonctions](#) [↗](#). Il y en a pour tous les goûts.

Si vous pensez avoir besoin d'une fonction qui n'existe pas encore, il va falloir la créer. Prenons pour exemple une fonction chargée de calculer la largeur d'une grille de mise en page, constituée de colonnes séparées par des gouttières. Elle aura trois arguments :

- le nombre de colonnes `$n`
- la largeur d'une colonne `$colonne`
- la largeur d'un gouttière `$gouttiere`

Le nombre de gouttières est forcément égal à `$n - 1`. Voici donc le code de notre fonction :

```
1 @function largeur-grille($n, $colonne, $gouttiere){
2   @return $n * $colonne + ($n - 1) * $gouttiere;
3 }
```

Comme vous le voyez, la syntaxe est proche de celle pour créer un mixin, à ceci près que l'on utilise la directive `@function` et que l'on doit appeler `@return` pour renvoyer le résultat de la fonction.

II. Les bases de Sass

Maintenant, on peut utiliser notre fonction comme n'importe quelle autre :

```
1  div{
2    width: largeur-grille(12, 60px, 20px); //940px
3 }
```

Sachez enfin que, comme pour les mixins, on a la possibilité de donner des valeurs par défauts aux arguments :

```
1  @function largeur-grille($n: 12, $colonne: 60px, $gouttiere: 20px){
2    @return $n * $colonne + ($n - 1) * $gouttiere;
3 }
```

7.6. En résumé

- Une **fonction** demande des arguments, fait des calculs et renvoie un résultat.
- On peut créer ses propres fonctions avec la syntaxe `@function`.

Mémo des fonctions utiles		
Catégorie	Nom(Arguments)	Résultat
Nombres	<code>percentage(\$number)</code>	Le pourcentage correspondant au nombre
	<code>round(\$number)</code>	L'arrondi à l'unité
Couleurs	<code>rgb(\$red, \$green, \$blue)</code>	La couleur demandée
	<code>rgba(\$red, \$green, \$blue, \$alpha)</code>	La couleur demandée
	<code>hsl(\$hue, \$saturation, \$lightness)</code>	La couleur demandée
	<code>hsla(\$hue, \$saturation, \$lightness, \$alpha)</code>	La couleur demandée
	<code>invert(\$color)</code>	La couleur inverse (négatif)
	<code>grayscale(\$color)</code>	La nuance de gris correspondant à la couleur
	<code>mix(\$color1, \$color2)</code>	Le mélange des deux couleurs
	<code>lighten(\$color, \$amount)</code>	La couleur plus claire (selon le %)

II. Les bases de Sass

	<code>darken(\$color, \$amount)</code>	La couleur plus foncée (selon le %)
	<code>saturate(\$color, \$amount)</code>	La couleur plus saturée (selon le %)
	<code>desaturate(\$color, \$amount)</code>	La couleur moins saturée (selon le %)
	<code>opacity(\$color, \$amount)</code>	La couleur plus opaque (selon le %)
	<code>transparentize(\$color, \$amount)</code>	La couleur moins opaque (selon le %)
Listes	<code>nth(\$list, \$n)</code>	La valeur dans la liste à l'index \$n
	<code>index(\$list, \$value)</code>	L'index de la valeur dans la liste
	<code>length(\$list)</code>	Le nombre d'éléments dans la liste
	<code>append(\$list, \$val, \$separator: auto)</code>	La liste avec la valeur ajoutée à la fin
	<code>join(\$list1, \$list2, \$separator: auto)</code>	Les deux listes assemblées en une seule
	<code>zip(\$lists...)</code>	La liste multidimensionnelle tirée des listes

C'est la fin de ce dernier chapitre concernant les bases de Sass. Mais c'est pas fini ! Dans les prochains chapitres, il sera question d'héritage de classe, de boucles, de conditions et d'autres réjouissances.

C'est la fin de cette première partie consacrée aux bases de Sass. Mais il reste encore quelques concepts plus avancés à découvrir !

Troisième partie

Plus loin avec Sass

III. Plus loin avec Sass

Dans cette seconde partie, nous allons nous intéresser à quelques aspects un peu plus techniques de Sass. Nous parlerons d'héritage, des conditions, des boucles, du type map et nous essaierons la bibliothèque externe Neat.

8. L'héritage avec @extend

Pour inaugurer cette seconde partie consacrée à des sujets un peu plus avancés concernant Sass, j'ai décidé de vous parler de la directive @extend. L'héritage n'est pas une notion complexe en soit, mais je n'ai pas souhaité vous en parler avant car on l'utilise assez peu, et que son utilisation est de plus en plus déconseillée. Cependant, je m'en voudrait de ne pas l'évoquer pour autant. On va donc voir en quoi cela consiste, et on verra ensuite pourquoi il est recommandé de s'en passer.

8.1. Il est où le grisbi ?

Rassurez-vous, votre grande-tante est encore en vie ! Ici, ce sont des sélecteurs qui héritent. L'héritage est la possibilité pour un élément d'hériter des propriétés d'un autre élément. Prenons par exemple les *cartes* de notre page Web (représentant un produit, une étape de production ou un avis de client). Elles ont toutes des propriétés en commun, regroupées dans la classe `.card`, mais ont aussi un certain nombre de propriétés différentes, qui sont réparties entre les classes `.product`, `.step` et `.client`. On peut dire que, conceptuellement, une élément `.product`, `.step` ou `.client` hérite des propriétés de la classe `.card`.

Pour l'instant, cet héritage est écrit en dur dans le HTML : chaque élément a deux classes. Si jamais on oublie de donner la classe `card` à un élément `product`, il ne recevra pas tous les styles nécessaires pour son bon affichage. Sass nous permet de gérer cela autrement, à l'intérieur de la feuille de style, avec la directive `@extend`. Cette dernière permet de dire qu'un élément X hérite forcément des propriétés d'un élément Y :

```
1 .card {
2     ...
3     img {
4         ...
5     }
6 }
7 .product {
8     @extend .card;
9     min-height: 20rem;
10    background-color: #fff;
11    color: #222;
12    ...
13 }
14 .step {
15     @extend .card;
```

III. Plus loin avec Sass

```
16 }
17 .client {
18   @extend .card;
19   background-color: #222;
20   color: #fff;
21   ...
22 }
```

Qu'est-ce que cela va donner en CSS ? En fait Sass va, tout simplement, placer les sélecteurs qui héritent partout où il trouve le sélecteur `.card`. Ainsi, tous les blocs de propriétés concernant `.card` seront appliqués aussi à `.product`, `.step` et `.client` :

```
1 .card, .product, .step, .client {
2   ...
3 }
4 .card img, .product img, .step img, .client img {
5   ...
6 }
```

Et voilà, on n'a donc plus besoin de la classe `card` dans notre HTML, vu que l'héritage est déjà géré dans le CSS. Un nouvel élément a lui aussi besoin de ressembler à un carte ? Pas de souci, il suffit de le faire hériter lui aussi.

Il est important de noter qu'on peut spécifier n'importe quel sélecteur **simple** après `@extend`. Il ne peut donc pas s'agir d'un *sélecteur imbriqué* du type `X Y`, `X>Y`, `X+Y` et `X~Y` (les deux derniers étant respectivement les sélecteurs d'adjacence directe et indirecte). Bon, d'un autre côté, si vous souhaitez que `.bidule` hérite de `.machin>p+div~.truc`, c'est que vous avez l'esprit légèrement tordu.



Encore plus fort : on peut faire des héritages en chaîne, «en cascade» :

```
1 .message {
2   border: 2px solid black;
3 }
4 .alerte {
5   @extend .message;
6   color: red;
7 }
8 .danger {
9   @extend .alerte;
10  font-size: xx-large;
11 }
```

La classe `.danger` hérite de la classe `.alerte` qui hérite de la classe `.message` !

8.2. Les placeholders (un nouvel eldorado ?)

Maintenant que nos classes héritent de `.card`, on a vu qu'on n'avait plus besoin de mentionner `card` dans notre HTML. À vrai dire, la classe `.card` n'est utile que pour Sass, pour faire notre héritage. Depuis sa version 3.2, Sass inclut donc un nouveau type de sélecteurs, les *placeholders*, ou « *@extend-only selectors* ». Un placeholder est une classe qui ne sert qu'à l'héritage, et qui disparaîtra durant la compilation. Pour transformer une classe en placeholder, ce n'est pas bien compliqué : il suffit de remplacer le `.` par un `%`. Ainsi, dans notre exemple :

```
1 %card {
2     ...
3     img {
4         ...
5     }
6 }
7 .product {
8     @extend %card;
9     min-height: 20rem;
10    background-color: #fff;
11    color: #222;
12    ...
13 }
14 .step {
15     @extend %card;
16 }
17 .client {
18     @extend %card;
19     background-color: #222;
20     color: #fff;
21     ...
22 }
```

Et le résultat en CSS ? La même chose qu'auparavant, mais sans `.card` :

```
1 .product, .step, .client {
2     ...
3 }
4 .product img, .step img, .client img {
5     ...
6 }
```

On se retrouve donc avec un placeholder, qui n'apparaît que dans le fichier SCSS, dont héritent les classes que l'on utilise réellement dans le HTML.

8.3. Bonne pratique : Faut-il oublier @extend ?

?

Mais, n'aurait-on pas pu utiliser un mixin à la place ?

Héhé, excellente question, chers lecteurs, que de nombreuses personnes se sont posées avant vous. Il est vrai qu'à chaque fois qu'on utilise l'héritage (ici avec le placeholder `%card`), on pourrait aussi faire appel à un mixin (le mixin `card`). À l'inverse, on aurait aussi pu gérer nos boutons avec l'héritage de placeholders plutôt qu'avec des mixins : on n'a pas accès aux arguments, mais on aurait pu créer un placeholder `%button`, dont héritent deux placeholders `%light-button` et `dark-button`, dont héritent les différentes classes de boutons de notre design. Le résultat est en apparence le même, cependant, on a quelques différences entre les 2 techniques :

- Il y a des répétitions dans le fichier CSS généré avec des mixins, pas avec l'héritage.
- On obtient des sélecteurs plus longs avec l'héritage.
- On ne peut évidemment pas utiliser d'arguments avec l'héritage.
- L'héritage pose problème à l'intérieur de media-queries, pas les mixins.
- On obtient parfois des comportements assez inattendus avec l'héritage.

Concernant le premier point, on a vu dans le chapitre sur les mixins que ce n'était pas un réel problème, vu que l'algorithme de *gzip* gère très bien les répétitions. Concernant le deuxième point, il peut, à grande échelle, influencer sur le temps de chargement de la page, mais c'est probablement peu visible². Le troisième point peut poser problème concernant la ré-utilisabilité du code : un placeholder n'est pas paramétrable, il n'a donc pas vraiment vocation à être réutilisé dans plusieurs projets. Les deux derniers points sont plus complexes, et carrément problématiques. En fait, pour les comprendre, il faut garder en mémoire qu'un placeholder, ce n'est techniquement pas la même chose qu'un mixin, leur fonctionnement étant différent.

8.3.1. Héritage & Media-queries

En effet, lorsqu'une classe `.x` hérite d'une classe `.y`, Sass va ajouter le sélecteur de `.x` partout où il trouvera `.y`, et il ne fera rien d'autre. Ainsi, imaginons le code suivant :

```
1 %agrumes {
2     color:red;
3 }
4 .mandarine {
5     @media (min-width: 720px) {
6         @extend %agrumes;
7     }
8 }
```

2. Encore que... il faudrait vérifier, je suis loin d'être spécialiste du domaine.

III. Plus loin avec Sass

Il est important que vous compreniez que ce code ne peut pas fonctionner. On pourrait s'attendre à obtenir ceci :

```
1 @media (min-width: 720px) {
2   .mandarine {
3     color:red;
4   }
5 }
```

Mais ça, c'est ce qui se passerait si on avait utilisé un mixin. Avec un placeholder, c'est très différent : Sass cherche où il peut trouver le sélecteur `%agrume` pour le remplacer par le sélecteur `.mandarine`. Sauf qu'il voit bien que l'élément `.mandarine` est à l'intérieur d'une media-query, alors que `%agrume` est à l'extérieur de celle-ci. Il ne va pas déplacer l'élément `%agrume` à l'intérieur de la media-query, parce que ce n'est pas comme cela que fonctionne l'héritage. On a donc droit à une erreur de compilation. En bref, l'héritage et les media-queries ne font pas bon ménage.

8.3.2. Comportements imprévus

Concernant le dernier point, il y a plusieurs choses à savoir. En effet, on pense, à tort, que l'héritage est équivalent à l'utilisation des mixins. Ce qui, régulièrement, peut générer du code superflu et imprévu.

Tout d'abord, rappelez-vous : je vous ai dit dans la première partie qu'on pouvait donner n'importe quel sélecteur simple à `@extend`. Cela veut dire que, théoriquement, un élément peut hériter d'une balise HTML comme `a`, `strong`, `h1`, etc. Mais ce serait plutôt une mauvaise idée. En effet, il est probable que ces balises existent dans différents contextes sur votre page : dans votre bannière, dans les différentes sections, dans une barre latérale, dans le pied de page... Si la classe `.bidon` hérite d'une de ces balises, Sass insèrera son nom partout où apparaît la balise en question, dans tous les contextes différents où la balise est utilisée. Mais, est-ce vraiment ce que l'on souhaite : la classe `.bidon` se trouve-t-elle réellement dans chacun de ces contextes ? Parce que, si ce n'est pas le cas, on aura créé des sélecteurs qui n'ont aucune raison d'exister.

Par ailleurs, n'oubliez pas que `@extend` ne déplace pas le code. Petit exemple :

```
1 .mandarine {
2   @extend %mere;
3   color : red;
4 }
5 %mandarine {
6   color: blue;
7 }
```

De quelle couleur est le texte de l'élément `.mandarine` ? Rouge ? Loupé, il est bleu, parce que la propriété du placeholder `%agrume` est située en-dessous dans la feuille de style et écrase donc la précédente.

III. Plus loin avec Sass

Parlons aussi un peu de la *fusion des sélecteurs*. Regardez cet exemple :

```
1 #####div1 #div2 em {
2   @extend strong;
3 }
4 #####div3 #div4 strong {
5   font-weight: bold;
6 }
```

Comment Sass va-t-il traduire ceci ? J'ai mis en évidence les deux séquences à fusionner. Comme elles n'ont pas d'éléments en commun, Sass créera deux nouveaux sélecteurs : le premier avec les parents de `strong` avant les parents de `em` et le second avec les parents de `em` avant les parents de `strong`. Voici donc le résultat :

```
1 #####div3 #div4 strong,
2 #div3 #div4 #div1 #div2 em,
3 #####div1 #div2 #div3 #div4 em {
4   font-weight: bold;
5 }
```

Maintenant, que se passe-t-il lorsque les éléments parents ont au moins un élément en commun ? Faisons le test avec cet exemple :

```
1 #####commun #div1 em {
2   @extend strong;
3 }
4 #####commun #div2 strong {
5   font-weight: bold;
6 }
```

Comme vous le voyez, les sélecteurs ont l'élément `#commun` en commun. Sass, durant la traduction en CSS, fusionnera les deux `#commun` :

```
1 #####commun #div2 strong,
2 #commun #div2 #div1 em,
3 #####commun #div1 #div2 em {
4   font-weight: bold;
5 }
```

Là encore, on peut se demander si tous ces sélecteurs générés sont bien utiles, s'ils ciblent bien tous des éléments qui existent réellement dans la page, parce que si ce n'est pas le cas, cela ralentira inutilement l'affichage de la page, ce qui, sur des terminaux peu puissants (GSMs d'entrée de gamme par exemple), se fera sentir.

8.3.3. On oublie l'héritage ?

Je pense que cette section assez théorique devrait vous avoir fait comprendre que l'héritage a de nombreux effets secondaires, alors que les mixins n'ont que des avantages, et sont paramétrables. Plusieurs articles publiés sur des sites spécialisés par des personnes autrement plus qualifiées que moi-même recommandent de proscrire `@extend`, tout simplement. Après, vous faites ce que vous voulez, vous êtes des grandes personnes.

8.4. En résumé

- L'héritage est la possibilité pour un élément d'**hériter des propriétés CSS d'un autre élément**.
- Les **placeholders** sont des sélecteurs dont le nom commence par le symbole `%` et dont peuvent hériter d'autres éléments. Ils ne servent d'ailleurs qu'à cela car ils n'apparaissent pas ensuite dans le code CSS généré.
- L'héritage amène parfois à **des résultats inattendus** et présente de sérieuses limitations. On lui préfère souvent les **mixins**.

Dans le prochain chapitre, on se lance dans un sujet passionnant, les conditions !

9. Les conditions

Entrons maintenant dans le royaume des *conditions*. Comme c'est un concept assez complexe à expliquer, je vous propose d'étudier d'abord le principe dans un premier temps, puis de voir son utilité dans un deuxième temps.

9.1. Les bases

Tout d'abord, il faut savoir que Sass nous permet d'effectuer des tests sur nos variables. Par exemple, on peut vérifier que la variable `$truc` vaut 42. Pour cela on utilise l'opérateur `==`, comme ceci :

```
1 $truc == 42
```

Ce test a pour résultat un booléen. Ce booléen vaudra `true` si le test est positif (si `$truc` vaut bien 42) ou `false` si le test est négatif (si `$truc` ne vaut pas 42). Il existe six opérateurs différents pour faire des tests :

Opérateur	Exemple	Test positif si...
<code>==</code>	<code>\$truc == 42</code>	<code>\$truc</code> égal 42
<code>!=</code>	<code>\$truc != 42</code>	<code>\$truc</code> différent de 42
<code><</code>	<code>\$truc < 42</code>	<code>\$truc</code> inférieur strict à 42
<code>></code>	<code>\$truc > 42</code>	<code>\$truc</code> supérieur strict à 42
<code><=</code>	<code>\$truc <= 42</code>	<code>\$truc</code> inférieur ou égal à 42
<code>>=</code>	<code>\$truc >= 42</code>	<code>\$truc</code> supérieur ou égal à 42

J'ai mis les deux premiers en gras car ce sont les seuls que j'ai déjà utilisé dans la pratique. Les autres sont, à mon humble avis, plus anecdotiques.

Intéressons-nous maintenant à la directive `@if`. Elle attend une condition à évaluer (le plus souvent un booléen) et un bloc de code entre accolades. Elle s'utilise comme ceci :

```
1 p{
2   @if $condition {
```

III. Plus loin avec Sass

```
3     color: red;
4   }
5 }
```

La condition à évaluer est ici `$condition` :

- si `$condition` vaut `true` ou n'importe quoi autre que `false` ou `null`, alors le bloc de code sera inséré :

```
1 p {
2   color: red;
3 }
```

- si `$condition` vaut `false` ou `null` alors Sass ignorera le bloc de code.

Comme vous devriez le deviner, on peut remplacer `$condition` par l'un des tests vus précédemment :

```
1 $var: 'bidule';
2 p {
3   @if $var == 'bidule'{
4     color: red;
5   }
6   @if $var == 'machin'{
7     color: blue;
8   }
9 }
```

Le code généré sera :

```
1 p {
2   color: red;
3 }
```

9.2. @else, @else if...

Passons à des choses plus complexes... On veut dire ceci à Sass :

```
1 SI $var == 'bidule'
2   Appliquer color: red;
3 SINON
```

III. Plus loin avec Sass

```
4 Appliquer color: blue;
```

La directive Sass pour dire « SINON » est `@else`. Elle s'utilise ainsi :

```
1 p{
2   @if $var == 'bidule'{
3     color: red;
4   }
5   @else{
6     color: blue;
7   }
8 }
```

Donc, si on a `$var: 'chose'`; alors le code généré sera :

```
1 p {
2   color: blue;
3 }
```

Encore plus compliqué. On veut que Sass comprenne ceci :

```
1 SI $var == 'bidule'
2   Appliquer color: red;
3 SINON SI $var == 'machin'
4   Appliquer color: green;
5 SINON
6   Appliquer color: blue;
```

Pour dire « SINON SI » à Sass, on va employer `@else if`, comme cela :

```
1 p{
2   @if $var == 'bidule'{
3     color: red;
4   }
5   @else if $var == 'machin'{
6     color: green;
7   }
8   @else{
9     color: blue;
10  }
11 }
```

III. Plus loin avec Sass

Deux trois choses à savoir (qui peuvent sembler évidentes à certains d'entre vous) :

- Les directives `@else` et `@else if` sont optionnelles.
- On peut utiliser autant de `@else if` que l'on veut après un `@if`, il ne peut y avoir qu'un `@else`.
- Le `@else` se place toujours en dernier, après le/les `@else if` s'il y en a.
- Il y a une grande différence entre plusieurs `@if` d'affilée et un `@if` suivi de plusieurs `@else if`. Dans le premier cas, Sass évalue chaque condition individuellement. Dans le second, si l'une des conditions est remplie, alors Sass ne regardera pas les suivantes.

Voilà pour la théorie, voyons maintenant à quoi ça sert dans la pratique..

9.3. Mini-TP : un mixin bien compliqué

9.3.1. Prérequis

Passons maintenant à la pratique, et j'ai décidé de vous laisser un peu libres. Vous vous souvenez des mixins « *media-queries* » que nous avons codés il y a quelques chapitres ? Je vous avais suggéré de faire un mixin par appareil (mobile, ordinateur de bureau, etc.). Il y a mieux. Nous allons créer un unique mixin `breakpoint` qui accepte un argument (le nom de l'appareil) et qui génère la *media-query* correspondante.³ Voici par exemple ce que j'attends :

```
1  \\SCSS :
2  p{
3    @include breakpoint(mobile){
4      font-size: 16px;
5    }
6  }
7  \\Ce qui donne en CSS :
8  @media screen and (max-width: 640px) {
9    p {
10     font-size: 16px;
11  }}
```

Je veux que votre mixin accepte au moins `mobile`, `tablette` et `ordi` en argument. Si on donne une valeur différente, Sass doit afficher un message d'erreur et ne pas générer de code CSS. Pour cela, vous devez utiliser la directive `@error` dont nous n'avons pas encore parlé, qui s'emploie ainsi :

```
1  @error
    "L'argument $appareil ne peut valoir que mobile, tablette ou ordi.";
```

Voilà, vous savez tout, à vous de jouer !

9.3.2. La solution

Fini ? Je vous laisse comparer votre solution avec la mienne. Si vous bloquez, vous pouvez toujours regarder un petit extrait puis ré-essayer par vous-même.

☞ Afficher le contenu masqué

9.3.3. Un peu mieux...

Comme tout TP, ce mixin n'est pas parfait. En effet, il est généralement déconseillé de choisir des *breakpoints* (points d'arrêts) spécifiques à un appareil. De nouveaux écrans apparaissent chaque année, plus petits (smartwatches) ou plus grands (phablettes, télévisions connectées). Il est donc recommandé de définir ses *breakpoints* en fonction du contenu.

Adaptons notre mixin pour qu'il utilise des variables plutôt que des largeurs d'écran fixes :

☞ Afficher le contenu masqué

Ainsi, on ne pense plus en terme d'appareils, mais de présentation de la page, de la plus étroite à la plus large. Les variables ont des valeurs par défaut, mais qui sont faites pour être modifiées selon le projet (il faut faire des tests pour déterminer les bons *breakpoints*, par exemple en utilisant [ish](#) [↗](#)).

Si vous enregistrez ce code dans une feuille partielle à part, vous pourrez ensuite l'importer dans chacune de vos feuilles de styles, en changeant auparavant les valeurs par défaut (c'est tout l'intérêt de `!default`) :

```
1 $bp-S: 500px;  
2 $bp-M: 980px;  
3 $bp-L: 1250px;  
4 @import 'partials/breakpoint';
```

C'est tout pour ce TP, il y a sans doute d'autres améliorations possibles. Pour approfondir, vous pouvez regarder du côté de [Breakpoint Sass \(en\)](#) [↗](#) , qui propose un mixin comme le nôtre (mais en plus complet).

9.4. En résumé

- La directive `@if` permet d'indiquer qu'un bloc de propriétés ne doit être appliqué que si une certaine condition est remplie. On l'utilise ainsi :

```
1 @if $une_variable == "quelque chose"{  
2   color: red;  
3 }
```

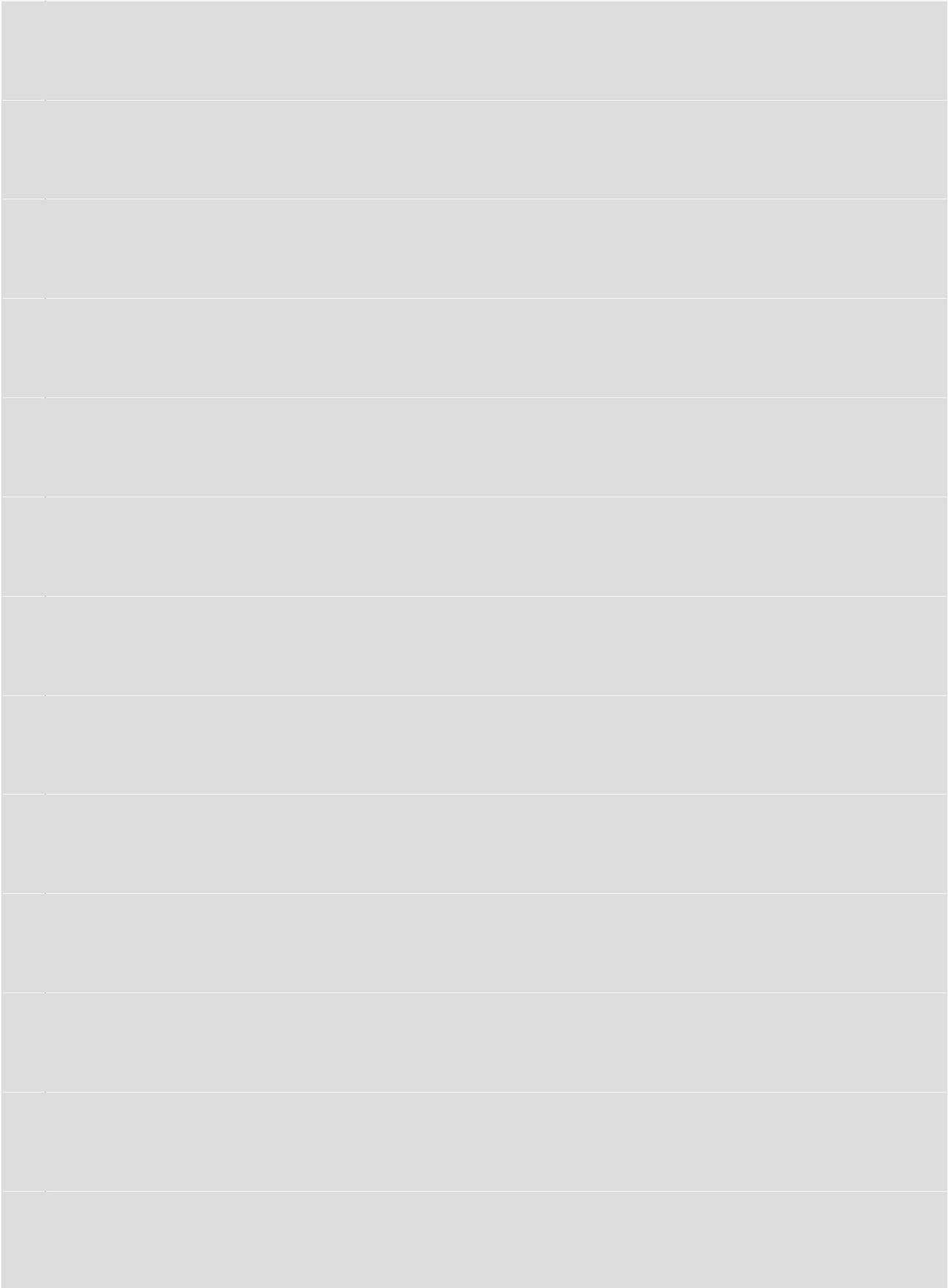
- Les directives `@else if` et `@else` permettent de créer des structures conditionnelles plus complexes. Elles sont facultatives et signifient respectivement « Sinon Si » et « Sinon ».

9.5. Pour s'entraîner

Pourquoi ne pas modifier un peu notre mixin `button`? Imaginons un instant un mixin qui n'accepte qu'un seul argument `$style`? Si `$style` vaut `"light"`, alors on obtient un bouton avec un fond clair, s'il vaut `$dark` on obtient un bouton au fond sombre.

Contenu masqué

Contenu 1



Contenu 2

```
1 $bp-S: 640px !default;
2 $bp-M: 800px !default;
3 $bp-L: 1024px !default;
4
5 @mixin breakpoint($bp){
6   @if $bp == xsmall{
7     @media screen and (max-width: $bp-S){
8       @content;
9     }
10  }
11  @else if $bp == small{
12    @media screen and (min-width: $bp-S){
13      @content;
14    }
15  }
16  @else if $bp == medium{
17    @media screen and (min-width: $bp-M){
18      @content;
19    }
20  }
21  @else if $bp == large{
22    @media screen and (min-width: $bp-L){
23      @content;
24    }
25  }
26  @else{
27    @error
28      "L'argument $bp ne peut valoir que xsmall, small, medium et
29  }
```

[Retourner au texte.](#)

10. Les boucles

Une fois n'est pas coutume, on va partir d'un exemple issu de notre fil rouge. Regardez le code suivant :

```
1 ###presentation {
2     background-image: url('img/bg-presentation.jpg');
3 }
4 ###production {
5     background-image: url('img/bg-production.jpg');
6 }
7 ###contact {
8     background-image: url('img/bg-contact.jpg');
9 }
```

Vous remarquez qu'on a répété trois fois le même code en ne changeant qu'un mot à chaque fois.

Stockons les différentes variantes dans une liste :

```
1 $sections: "presentation", "production", "contact";
```

Et maintenant, comment fait-on ? On utilise une boucle, évidemment !

10.1. La boucle @for

Un boucle permet de répéter un certain nombre de fois un même bout de code, en changeant une valeur à chaque fois. Il existe 3 familles de boucles dans le langage SCSS (nous n'en verrons que deux). La première, c'est la boucle @for. Observons le code suivant :

```
1 @for $i from 1 through 3 {
2     #section-#{$i} {
3         background-image: url('img/bg-' + $i + '.png');
4     }
5 }
```

On obtient le CSS suivant :

```
1 #####section-1 {
2     background-image: url("img/bg-1.png");
3 }
4 #####section-2 {
5     background-image: url("img/bg-2.png");
6 }
7 #####section-3 {
8     background-image: url("img/bg-3.png");
9 }
```

Comme vous pouvez le voir, la boucle `@for` a répété trois fois le bloc de code, en changeant la valeur de `$i` à chaque itération (ou répétition de la boucle). Cette variable (appelée indice de la boucle) a pris tour à tour les valeurs 1, 2 puis 3 (tous les entiers entre 1 et 3, `from 1 through 3`). Remarquez que ici, l'interpolation et la concaténation m'ont été très utiles.

Bon, c'est très bien tout cela, mais on n'a pas obtenu le code du départ. Alors oui, on pourrait changer les ids de nos sections dans le HTML et les noms de nos images, mais ce serait de la triche. Non, on va plutôt utiliser notre liste `$sections`, et une fonction que nous avons vu il y a peu : `nth()`. Et il faut bien dire qu'elle ne révèle toute son utilité qu'avec les boucles :

```
1 $sections: "presentation", "production", "contact";
2 @for $i from 1 through length($sections) {
3     #section-#{nth($section, $i)} {
4         background-image: url('img/bg-' + nth($section, $i) +
5             '.png');
6     }
```

Et voilà ! On obtient à nouveau un code qui se sert des noms de images et de nos ids. Remarquez que, pour plus de flexibilité, j'ai utilisé `length()` pour indiquer la fin de la boucle. Ainsi, si on a une nouvelle section à ajouter, on aura juste à modifier la liste et la boucle sera toujours valable.

i

Deux petites choses à savoir aussi sur la boucle `@for` (mais dont l'utilité est limitée) :

- on peut compter à l'envers : `@for $i from 4 through 1` (`$i` vaudra 4, 3, 2, puis 1).
- on peut remplacer `through` par `to` : dans ce cas, Sass n'inclut pas la dernière valeur (dans notre exemple, `$i` vaudra 1 puis 2 mais pas 3).

10.2. La boucle `@each`

La boucle `@for` est formidable. Mais il y a encore mieux. Les boucles sont la plupart du temps utilisées pour parcourir des listes (c'est d'ailleurs ce qu'on a fait). Les concepteurs de Sass ont

III. Plus loin avec Sass

donc ajouté à notre attirail une autre boucle, dédiée spécifiquement à cette utilisation. La boucle `@each`, comme son nom l'indique, se répète pour CHAQUE item d'une liste. Elle s'utilise comme ceci :

```
1 $sections: "presentation", "production", "contact";
2 @each $section in $sections {
3     #section-#{ $section } {
4         background-image: url('img/bg-' + nth($section, $i) +
5             '.png');
6     }
```

La première ligne se lit ainsi : la variable `$section` vaudra tour à tour chaque valeur contenue dans la liste `$sections`. On obtient donc le même code que précédemment.

Vous vous rappelez des listes multidimensionnelles? Elles sont aussi parcourables avec `@each`, c'est ce qu'on appelle *l'affectation multiple*. À chaque itération, on peut accéder à tous les éléments d'une des « sous-listes », comme ceci :

```
1 // Ce bout de code permet de gérer en un rien de temps le rythme
   // vertical de vos titres.
2 @each $titre, $fonte in (h1, 2.5rem), (h2, 2rem), (h3, 1.5rem) {
3     #{$titre} {
4         font-size: $fonte;
5     }
6 }
```

Le code CSS généré est le suivant :

```
1 h1 {
2     font-size: 2.5rem;
3 }
4 h2 {
5     font-size: 2rem;
6 }
7 h3 {
8     font-size: 1.5rem;
9 }
```

Personnellement, je trouve plus simple et plus lisible d'utiliser la fonction `zip()` :

```
1 $titres: h1, h2, h3;
2 $fontes: 2.5rem, 2rem, 1.5rem;
```

```
3 @each $titre, $fonte in zip($titres, $fontes) {  
4   #{$titre} {  
5     font-size: $fonte;  
6   }  
7 }
```

Voilà, c'est tout pour `@each`, même si nous en reparlerons lorsqu'il sera question des *maps*.

10.3. Mini-TP : La gestion d'une sprite

10.3.1. Énoncé

Je vous propose de finir ce chapitre avec un petit exercice où les boucles nous seront particulièrement utiles : l'utilisation d'une **sprite d'images**. Une sprite, si vous ne savez pas ce que c'est, c'est une grande image sur laquelle on regroupe toutes les icônes, tous les smileys, tous les pictogrammes utilisés dans un design. Comme souvent, le but recherché est de réduire la charge pour le serveur et le temps de chargement pour les visiteurs (plutôt que d'effectuer une trentaine de requêtes sur les images, on n'en a plus qu'une seule à effectuer). Ensuite, on se débrouille avec des `background-position` pour sélectionner la bonne image. Comme c'est assez fatigant à effectuer manuellement, il existe des outils clef-en-main qui génèrent directement la *sprite* et le code CSS correspondant. Mais je vous propose de partir sur quelque chose de simple et de tout faire avec une boucle. Le but est d'obtenir ceci :



FIGURE 10.1. – Résultat de l'exercice

Voici le code HTML (il n'a rien d'exceptionnel) :

```
1 <a class="button icon-settings">Settings</a>  
2 <a class="button icon-watch">Watch</a>  
3 <a class="button icon-power">Power</a>  
4 <a class="button icon-fullscreen">Full Screen</a>  
5 <a class="button icon-reduce">Reduce</a>
```

Voici notre sprite :

Et voici les styles de base de nos boutons (ce n'est pas franchement le sujet de l'exercice) :

```
1 .button {  
2   font-family: sans-serif;  
3   border: none;  
4   padding: 10px 10px 10px 52px; // On laisse de la place à gauche  
5   background-color: #fe0;  
6   position: relative; // Pour pouvoir placer  
   l'icône en absolute à l'intérieur  
7   display: inline-block;  
8   height: 32px;  
9   line-height: 32px;  
10  
11  &:hover {  
12    background-color: lighten(#fe0, 20%); //  
   C'est plus sympa, non ?  
13  }  
14 }
```

Maintenant, à vous de jouer ! Il suffit de chipoter un peu avec `.button::before`, une boucle et `background-position` et le tour est joué.

10.3.2. Correction

Voici ma correction de l'exercice :

☞ Afficher le contenu masqué

J'ai surligné la partie qui nous intéresse vraiment. On a donc une liste qui contient les noms des icônes dans le bon ordre et une boucle `@for` qui parcourt la liste (avec la fonction `nth()`), insère le nom dans le sélecteur (avec une interpolation qui va bien) et la valeur de l'indice `$i` sert à calculer la position de la sprite. On aurait aussi pu parcourir la liste directement avec une boucle `@each` et se servir de la fonction (`index()`) pour retrouver la valeur de `$i`, mais ça me semblait moins évident.

Voilà pour ce petit exercice. Comme dit précédemment, on préférera souvent utiliser un outil externe³ qui nous génère la sprite directement et le code (S)CSS qui va avec. En effet, on avait ici des icônes de la même taille alignées, ce qui est rarement le cas en conditions réelles, où il s'agit de gérer des sprites plus complexes. Voici par exemple celle utilisée par votre site préféré :



FIGURE 10.2. – La sprite de Zeste de Savoir

-
- La boucle `@for` est utile pour **répéter plusieurs fois un même code en changeant uniquement un nombre** et s'utilise ainsi : `@for $i from 1 through 4 {...}`.
 - La boucle `@each` permet de **répéter un même bout de code en parcourant une liste d'items**, on s'en sert comme ceci : `@each $aliment in frites, tomate, yaourt, raviolis {...}`.

Je ne vous ai volontairement pas parlé de la boucle `@while`, qui me semble moins utile que les deux autres. Sachez que c'est une boucle qui se répète tant qu'une condition est vraie. Si vous pensez en avoir besoin un jour, je vous invite à consulter [la doc Sass](#) .

3. Si vous êtes intéressé, je vous invite à regarder du côté de [Stitches](#) .

Contenu masqué

Contenu 1

```
1 .button {
2   font-family: sans-serif;
3   border: none;
4   padding: 10px 10px 10px 52px; // On laisse de la place à gauche
5   background-color: #fe0;
6   position: relative; // Pour pouvoir placer
   l'icône en absolue à l'intérieur
7   display: inline-block;
8   height: 32px;
9   line-height: 32px;
10
11  &:hover {
12    background-color: lighten(#fe0, 20%); //
   C'est plus sympa, non ?
13  }
14
15  &::before {
16    content: '';
17    display: block;
18    position: absolute;
19    top: 10px;
20    left: 10px;
21
22    width: 32px;
23    height: 32px;
24    background-image: url(sprite.png);
25  }
26 }
27 $icons: 'settings', 'watch', 'power', 'fullscreen', 'reduce';
28
29 @for $i from 1 through 5 {
30   .icon-#{nth($icons, $i)}::before {
31     $pos: -32px * ($i - 1); // On commence à 0 et on finit à
   32*4px
32     background-position-x: $pos;
33   }
34 }
```

[Retourner au texte.](#)

11. Et maintenant ?

Nous arrivons tout doucement à la fin de ce tutoriel. Dans ce dernier chapitre (un peu fourre-tout, je dois bien l'admettre), il sera question du type `map`, de bibliothèques externes et des différentes implémentations de Sass.

11.1. Le type `map`

Avant de vous lâcher dans la nature, je m'en voudrais de ne pas évoquer le type `map`. Je n'ai pas eu l'occasion de vous en parler jusqu'à présent, mais il s'agit du dernier arrivé (depuis Sass 3.3) parmi les types de données. Une `map` correspond à ce que l'on retrouve dans certains langages de programmation sous le nom de *tableau associatif*, de *hash*, ou encore de *dictionnaire*. On peut voir une `map` comme une liste dont chaque item serait une paire clef-valeur. Comme c'est tout de suite plus simple avec un exemple, je vous propose de voir comment on pourrait s'en servir pour les couleurs de notre feuille de styles :

```
1 $colors: (  
2   base: #ff0,  
3   base-hover: #ff4,  
4   dark: #222,  
5   dark-hover: #444,  
6   light: #eee,  
7   ultra-light: #fff  
8 );
```

i

Notez que j'ai écrit toutes les valeurs en dur pour cet exemple, mais que j'aurais clairement pu utiliser les fonctions `darken()` et `lighten()` pour en calculer certaines.

Comme vous pouvez le voir, une `map` se définit entre parenthèses, les paires étant séparées par des virgules.

Une fois notre `map` ainsi constituée, on peut s'en servir avec la fonction `map-get()` ainsi :

```
1 .client {  
2   ...  
3   background-color: map-get($colors, dark);  
4   color: map-get($colors, ultra-light);
```

III. Plus loin avec Sass

```
5   ...
6   }
```

En terme d'organisation, on se retrouve avec une seule variable pour regrouper toutes nos couleurs, au lieu de la demi-douzaine auparavant, c'est donc plutôt une bonne chose. Cependant, la syntaxe `map-get()` est sacrément lourde à utiliser. C'est pourquoi, certains recommandent d'utiliser une fonction *helper* pour raccourcir celle-ci :

```
1  @function color($name) {
2    @return($colors, $name);
3  }
4
5  .client {
6    ...
7    background-color: color(dark);
8    color: color(ultra-light);
9    ...
10 }
```

Simple, clair, organisé. On peut d'ailleurs faire des choses assez cools avec un tel système, pour supporter plusieurs « variantes » d'une même couleur via des sous-map. Si vous voulez en savoir plus, je vous invite à consulter [cet article de Erskine Design](#) .

En dehors des couleurs, le type map peut aussi se révéler utile pour toute variable de configurations devant contenir des valeurs qui sont utilisées par plusieurs fonctions d'une même bibliothèque, par exemple. Mais c'est quoi une bibliothèque, au juste ?

11.2. Utiliser des bibliothèques externes

Depuis le début de ce tutoriel, je n'arrête pas de vous dire que Sass évite de se répéter. C'est tellement vrai que des gens ont créé pour nous des **bibliothèques**, regroupant *mixins* et *fonctions* prêts à l'emploi. Je propose de vous en présenter deux-trois ici, sans entrer dans les détails, car leur documentation est souvent très accueillante.

11.2.1. Neat



FIGURE 11.1. – Neat

III. Plus loin avec Sass

Neat nous facilite la vie lorsqu'on doit créer des grilles fluides. Comme il s'agit d'une tâche récurrente, et qui nécessite des calculs parfois complexes, c'est l'exemple parfait de ce qu'une bibliothèque Sass peut nous offrir. Pour l'installer, la première chose à faire est d'utiliser **gem** :

```
1 gem install neat
```

Ensuite, déplacez-vous dans le dossier qui accueille les fichiers SCSS de votre projet et entrez la commande :

```
1 neat install
```

Cela devrait créer un sous-dossier "neat". Pour utiliser Neat, il ne vous reste plus qu'à l'importer, comme n'importe quel *partial* :

```
1 @import "neat/neat";
```

Neat propose principalement deux mixins : **grid-container** et **grid-column**. Voici un petit exemple pour mieux comprendre :

```
1 <div class="conteneur">
2   <div class="simple">Lorem ipsum...</div>
3   <div class="simple">Lorem ipsum...</div>
4   <div class="double">Lorem ipsum...</div>
5 </div>
```

```
1 @import "neat/neat";
2
3 .conteneur {
4   @include grid-container;
5 }
6 .simple {
7   @include grid-column(3);
8 }
9 .double {
10  @include grid-column(6);
11 }
```

On obtient ceci :

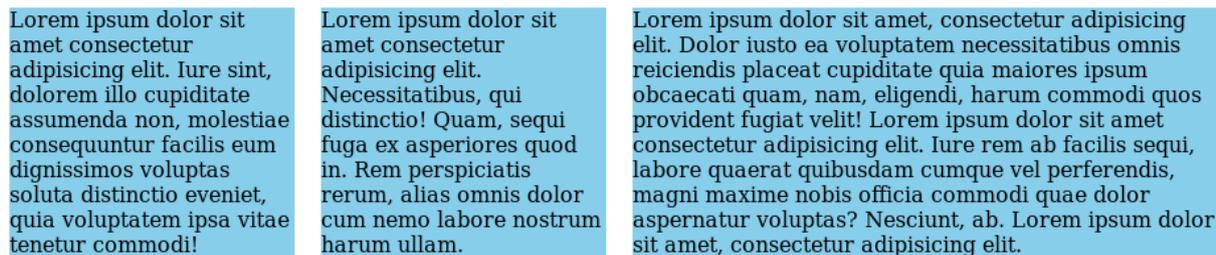


FIGURE 11.2. – Un exemple de grille utilisant Neat

Par défaut, Neat utilise une grille de douze colonnes, séparées par des gouttières de 20 pixels. On peut changer cela, en utilisant une map :

```
1 grid: (  
2   columns: 4,  
3   gutter: 20px);  
4  
5 .conteneur {  
6   @include grid-container;  
7 }  
8 .simple {  
9   @include grid-column(1, $grid);  
10 }  
11 .double {  
12   @include grid-column(2, $grid);  
13 }
```

Je ne développe pas plus au sujet de Neat, mais je vous invite à [lire la documentation qui est très bien faite](#) [↗](#) . Sachez qu'on peut aussi se servir de Neat pour créer des grilles *responsive* avec des *breakpoints*.

11.2.2. Bourbon



FIGURE 11.3. – Bourbon

Les auteurs de Neat se sont avant tout fait connaître avec **Bourbon**, une autre bibliothèque Sass. Au départ, il s'agissait d'un bibliothèque de *mixins* gérant les préfixes propriétaires (préfixes requis par les navigateurs sur certaines propriétés CSS3). Par exemple, un mixin `animation` permettait d'obtenir toutes les variantes préfixées de la propriété `animation`. **Ce n'est plus le cas**. Tout d'abord, parce que l'évolution des navigateurs rend l'utilisation de tels préfixes

III. Plus loin avec Sass

de moins en moins nécessaire, et ensuite parce des outils comme [Autoprefixer](#) [↗], intervenant après Sass, on remplacé les bibliothèques.

Aujourd'hui, Bourbon propose néanmoins plusieurs mixins, fonctions et variables prédéfinis qui peuvent vous permettre de gagner du temps. Il faut le voir comme un couteau-suisse pour Sass.

L'installation se déroule de la même manière que Neat : d'abord `gem install bourbon` puis `bourbon install` afin de pouvoir l'importer avec `@import bourbon/bourbon;`.

Ça ne sert pas à grand chose que je vous liste tout ce que Bourbon propose, mais sachez qu'on y trouve un mixin pour générer des triangles, différentes courbes de timing pour les animations, des listes de polices de caractères fréquentes, etc. La documentation est de toute façon très bien faite, et [je vous invite à vous y référer](#) [↗].

11.2.3. Susy3



FIGURE 11.4. – Susy3

Susy est un autre système de grille fluide, très chouette à utiliser. L'approche est un peu différente, Susy ne proposant pas de mixins tous prêts mais des fonctions qui effectuent les calculs de largeur à notre place. L'idée est de ne pas imposer une méthode (Neat utilise des flottants dans ces mixins), afin de pouvoir utiliser, si on le souhaite, les dernières nouveautés de CSS à ce sujet, *flexbox* et le module de grille de CSS⁴. La [page de présentation](#) [↗] est un bon endroit pour commencer.

11.3. RubySass, LibSass, DartSass, et les autres...

Avant de se quitter, j'aimerais vous parler un peu des différentes implémentations de Sass. Jusqu'à présent, nous avons utilisé RubySass, c'est-à-dire l'implémentation de Sass écrite en Ruby. Il s'agit de l'édition originale et officielle de Sass (au moment où j'écris ces lignes nous en sommes à la version 3.5). Elle a l'avantage d'être assez facile à utiliser via la commande `sass --watch` qui accepte pas mal de paramètres. D'ailleurs, à ce sujet, saviez-vous que `sass --watch` accepte un paramètre `-t` qui permet de décider quel style d'indentation doit s'appliquer aux fichiers CSS compilés ? `t` peut prendre la valeur *nested* (sa valeur par défaut), *compressed*, *compact* ou *expanded* (les noms parlent d'eux-même) :

4. Comment ! Vous ne connaissez pas le [CSS Grid module](#) [↗] ? Mais c'est le futur, les amis !

```
1 sass --watch sass:stylesheets -t compressed
```

Cependant, RubySass présente l'inconvénient d'être assez lent. À notre échelle ce n'est pas un véritable problème, mais ça peut le devenir sur de gros projets. C'est pour cela qu'est apparue LibSass, un portage de Sass en C/C++. LibSass est pensée pour être extrêmement rapide mais ne propose pas, de base, une interface en ligne de commande, comme RubySass. À la place, on a la possibilité de faire appel à LibSass à partir d'un grand nombre de langages de programmation.

Cela peut sembler un peu compliqué, mais c'est ce qui va permettre par exemple d'appeler Sass depuis Javascript. Je ne choisis évidemment pas cet exemple par hasard. Vous avez peut-être déjà entendu parler de Node.js, qui permet d'exécuter du code Javascript côté serveur.

Pour plusieurs raisons, la première étant que les développeurs front-end connaissent bien Javascript, Node est aussi devenu l'outil standard pour scripter toutes les actions répétitives du front-end comme la minification du JS et du CSS, l'utilisation d'Autoprefixer, la génération des sprites et, vous l'aurez compris, la compilation Sass. C'est pourquoi l'implémentation la plus populaire de LibSass reste node-sass.

Je ne vais pas développer ici l'automatisation des tâches du front-end avec Node, déjà parce que je ne connais pas bien ce sujet, et ensuite parce qu'il est suffisamment vaste pour qu'un cours entier lui soit destiné.

?

Mais, n'est-ce pas problématique pour les développeurs de Sass de devoir maintenir RubySass et LibSass en même temps ?

C'est une excellente question, à se demander si on ne vous l'a pas soufflée.

En effet, le fait de devoir maintenir deux versions n'est pas sans inconvénients. Les nouvelles fonctionnalités apparaissent d'abord dans RubySass, et il faut attendre pour qu'elles soient disponibles dans LibSass. En plus, il existe, ou il a existé, des comportements légèrement différents entre les deux implémentations.

C'est ainsi qu'à la mi-2016, la développeuse Natalie Weizenbaum a annoncé [le lancement de DartSass](#) [↗](#), destiné à être la nouvelle implémentation officielle de Sass, remplaçant à le long terme RubySass. Le choix du langage Dart, soutenu par Google, n'est pas anodin. Dart permet une exécution rapide, proche de la vitesse de LibSass, tout en restant pratique pour implémenter de nouvelles fonctionnalités pour les développeurs de Sass, comme l'était Ruby. Surtout, Dart peut être exporté vers Javascript, facilitant l'intégration avec Node. À l'heure où j'écris ces lignes, DartSass est encore en bêta, mais il semble bien que ce soit là que se trouve le futur de ce qui est désormais, je l'espère, votre préprocesseur favori.

Chapitre assez chargé, n'est-ce pas ? On a vu comment se servir du type map, on a regardé du côté des bibliothèques externes et on a même évoqué (rapidement) LibSass. Avant de se quitter, je vous laisse avec quelques bonnes adresses pour se tenir au courant des nouveautés autour de Sass.

III. Plus loin avec Sass

En plus du [blog officiel de l'équipe qui développe Sass](#) , je vous conseille de jeter régulièrement un œil du côté de sites spécialisés comme [SitePoint](#) , [The Treehouse](#) ou [CSS Tricks](#) . Ces trois là sont de vraies mines d'or.

À l'époque où j'ai débuté l'écriture de ce tutoriel, [The Sass Way](#) était la référence en termes d'articles. Malheureusement, il ne semble pas avoir été mis à jour depuis 2015 (oui, ce tutoriel a mis pas mal de temps à voir le jour :-°). Ce qui me permet de bien insister sur le fait que l'écosystème Sass est en constant changement. Si Sass reste à peu près stable, les bibliothèques qui l'entourent ne cessent de changer et d'évoluer. Un article qui a 2 ans peut donc être déjà obsolète.

Ah, et pour finir, [n'oubliez pas la doc!](#)

Et voilà, vous en savez d'avantage sur les fonctionnalités plus avancées de Sass !

Quatrième partie

Conclusion

IV. Conclusion

Et voilà, vous êtes arrivés à la fin de ce tuto ! Merci de l'avoir lu jusqu'au bout ! Je n'exclus pas quelques changements par la suite, si j'ai le temps d'améliorer certains exemples, ou si certaines infos deviennent obsolètes (et cela risque d'arriver assez vite). S'il vous reste des interrogations, n'hésitez pas à les poster sur le forum Web.

Pour finir, je remercie évidemment toutes les personnes qui ont suivi ce tuto en bêta et m'ont apporté leurs conseils (et tout particulièrement Vayel) ainsi qu'Abdelazer et artragis pour leur relecture minutieuse.

Liste des abréviations

HSL Hue Saturation Light (Teinte Saturation Lumière). [23](#), [45](#)

HSLA Hue Saturation Light Alpha (Teinte Saturation Lumière Alpha). [23](#)