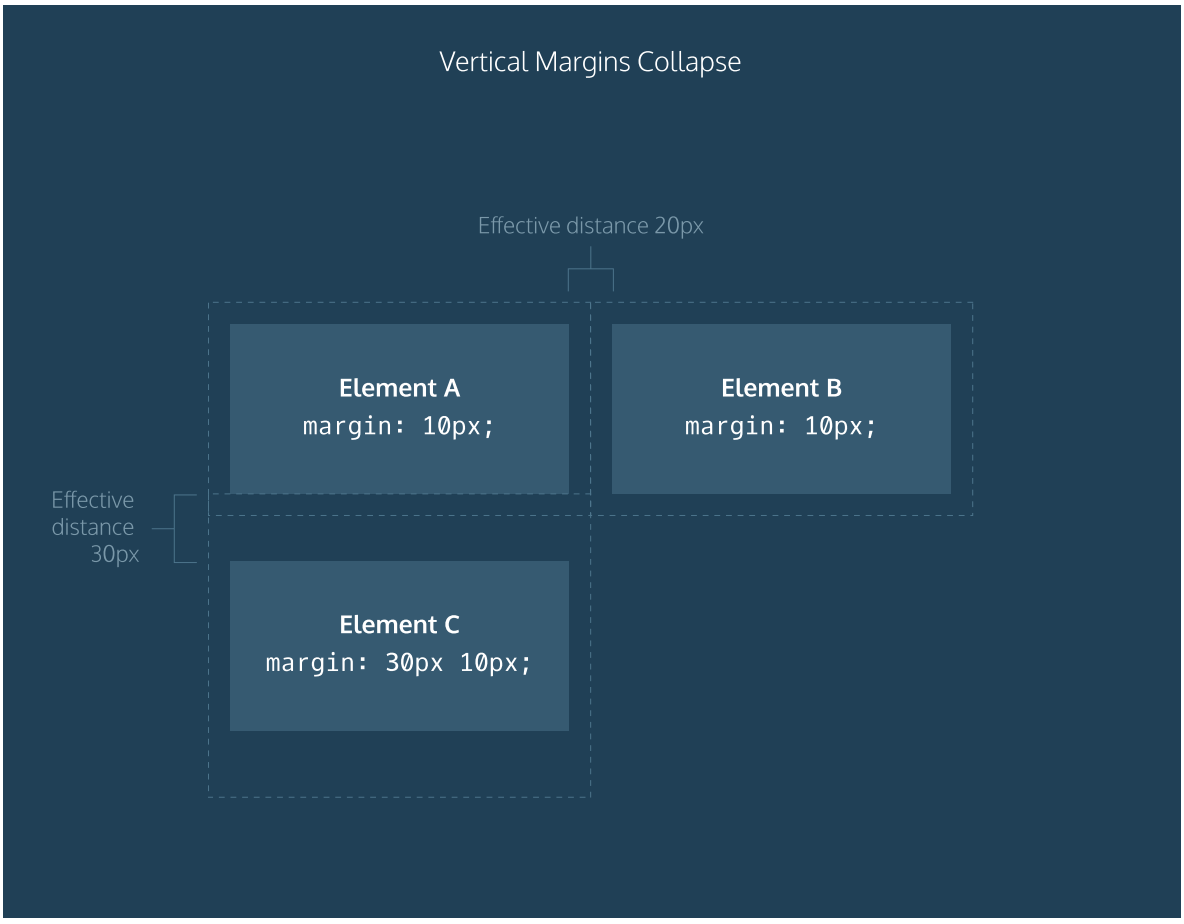
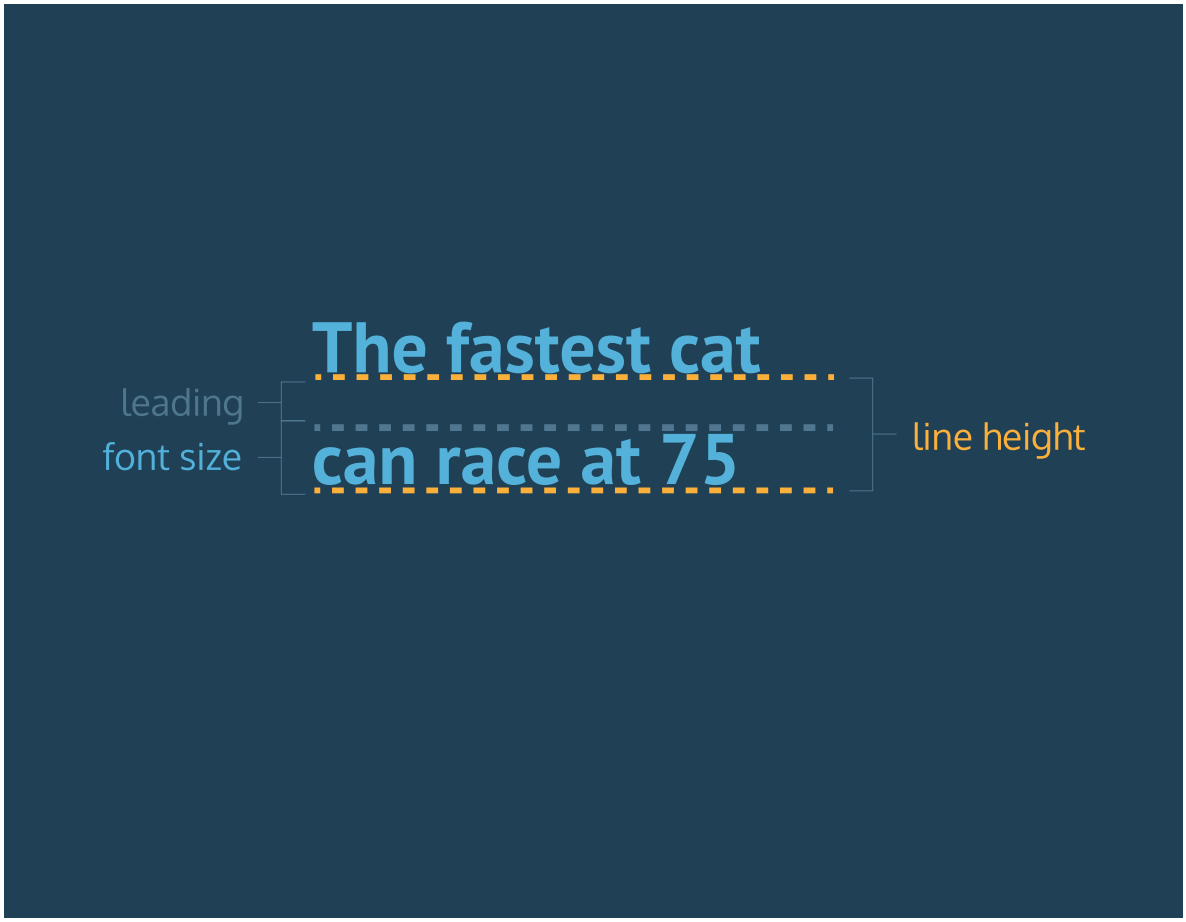
CSS

Box margin



Line Height



Grid

```
.grid {
  display: grid;
  border: 2px blue solid;
  width: 400px;
  height: 500px;
  grid-template: repeat(3, 1fr) / 3fr 50% minmax(100px, 500px);
  grid-gap : 20px 5px;
}
.item {
  grid-row: 5/7;
  grid-column: 2/ span 6;
}
.item2 {
  grid-area: 6 / 8 / span 3 / span 1;
}
```

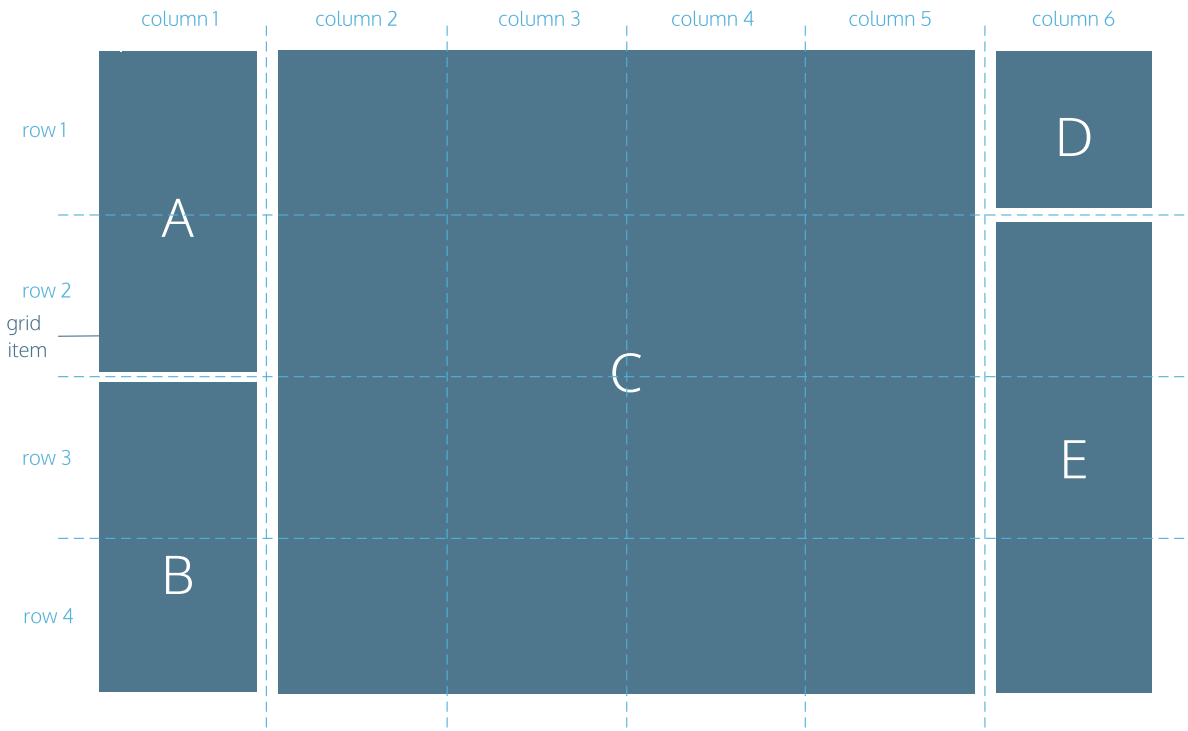
Dans la propriété `grid-template`, la partie avant le `/` correspond aux lignes, après ce sont les colonnes.

`fr` = Fraction de la hauteur ou l'espace disponible (colonnes ou lignes).

`repeat` = fonction CSS spécifique à la propriété `grid`. On peut placer plusieurs valeurs après le nombre de répétitions afin de répeter un pattern (ex: repeat(3, 1fr 2fr))

`span` = Longueur de cellules à fusionner. Permet d'éviter de se tromper la **fin d'une ligne ou colonne est non incluse** (+1)

`grid-area` = début ligne / début colonne / fin ligne / fin colonne.



Media Queries -> Responsive

Throughout this lesson, you learned:

- When a website responds to the size of the screen it's viewed on, it's called a *responsive* website.
- You can write *media queries* to help with different screen sizes.
- Media queries require *media features*. Media features are the conditions that must be met to render the CSS within a media query.
- Media features can detect many aspects of a user's browser, including the screen's width, height, resolution, orientation, and more.
- The `and` operator requires multiple media features to be true at once.
- A comma separated list of media features only requires one media feature to be true for the code within to be applied.
- The best practice for identifying where media queries should be set is by resizing the browser to determine where the content naturally breaks. Natural breakpoints are found by resizing the browser.

Sass

```
@import url(https://fonts.googleapis.com/css?family=Pacifico);
//Add variables here:
$translucent-white: rgba(255,255,255,0.3);
$icon-square-length: 300px;
$standard-border: 4px solid black;
```

```
h1 {
  font-family: Roboto, sans-serif;
  text-align: center;
}

.banner {
  font-family: 'Pacifico', cursive;
  height: 400px;
  background-image: url("lemonade.jpg");
  .slogan {
  position: absolute;
  border: 4px solid black;
  top: 200px;
  left: 25%;
  width: 50%;
  height: 200px;
  text-align: center;
  background-color: $translucent-white;
    span {
  font-size: 24px;
  line-height: 200px;
}
}
  border : {
    top: 4px solid black;
    bottom: 4px solid black;
}
}
```

1. **Nesting** is the process of placing child selectors and properties in the scope of a parent selector. This allows a programmer to draw DOM relationships and avoid repetition.
2. **Variables** make it easy to update code and reference values by allowing you to assign an identifier to a value.
3. Sass **Data Types** include:

- Numbers
- Strings
- Booleans
- null
- Lists (Séparés avec espaces ou virgules)
- Maps => (key1: value1, key2: value2);

Fonctions

```
@for $i from $begin through $end {
   //some rules and or conditions
}
background: adjust-hue(blue, $i * $step);

width: if( $condition, $value-if-true, $value-if-false);
```

- **Functions** in Sass allow for an easier way to style pages, work with colors, and iterate on DOM elements.
- Having both **for loops** and **each loops** gives the programmer different formats to iterate on both lists and maps.
- The introduction of **conditional statements** allows you to create logic-based styling rules using SCSS.

Maintenabilité

```
@mixin no-variable {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

%placeholder {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

span {
  @extend %placeholder;
}

div {
  @extend %placeholder;
}

p {
  @include no-variable;
}

h1 {
  @include no-variable;
}```

would compile to:

```scss
span, div{
  font-size: 12px;
  color: #FFF;
  opacity: .9;
}

p {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
  //rules specific to ps
}
```
```

```
h1 {
  font-size: 12px;
  color: #FFF;
  opacity: .9;
  //rules specific to ps
}
```

As a general rule of thumb, you should

- Try to **only create mixins that take in an argument**, otherwise you should extend.
- Always look at your CSS output to make sure your extend is behaving as you intended.

Flex box

Contrairement à `Grid` qui est plutôt destiné à faire du “layout” de pages entières (axe x et y) `Flexbox` est plutôt destiné à organiser des ensembles d’élements dans un contenaire.

Flex direction

```
.container {
  display: flex;
  flex-direction: column;
  width: 1000px;
}
```

Up to this point, we’ve only covered flex items that stretch and shrink horizontally and wrap vertically. As previously stated, flex containers have two axes: a *major axis* and a *cross axis*. By default, the major axis is horizontal and the cross axis is vertical.

The major axis is used to position flex items with the following properties:

1. `justify-content`
2. `flex-wrap`
3. `flex-grow`
4. `flex-shrink`

The cross axis is used to position flex items with the following properties:

1. `align-items`
2. `align-content`

The major axis and cross axis are interchangeable. We can switch them using the `flex-direction` property. If we add the `flex-direction` property and give it a value of `column`, the flex items will be ordered vertically, not horizontally.

Flex flow

```css
.container {
  display: flex;
  flex-wrap: wrap;
  flex-direction: column;
}
/* VERSION RAPIDE */
.container {
  display: flex;
  flex-flow: column wrap;
}
```

Resumé

1. `display: flex` changes an element to a block-level container with flex items inside of it.
2. `display: inline-flex` allows multiple flex containers to appear inline with each other.
3. `justify-content` is used to space items along the major axis.
4. `align-items` is used to space items along the cross axis.
5. `flex-grow` is used to specify how much space (and in what proportions) flex items absorb along the major axis.
6. `flex-shrink` is used to specify how much flex items shrink and in what proportions along the major axis.
7. `flex-basis` is used to specify the initial size of an element styled with `flex-grow` and/or `flex-shrink`.
8. `flex` is used to specify `flex-grow`, `flex-shrink`, and `flex-basis` in one declaration.
9. `flex-wrap` specifies that elements should shift along the cross axis if the flex container is not large enough.
10. `align-content` is used to space rows along the cross axis.
11. `flex-direction` is used to specify the major and cross axes.
12. `flex-flow` is used to specify `flex-wrap` and `flex-direction` in one declaration.
13. Flex containers can be nested inside of each other by declaring `display: flex` or `display: inline-flex` for children of flex containers.

CSS transition

```css
transition-property: width;
transition-duration: 750ms;
transition-timing-function: ease-out;
transition-delay: 250ms;
/* SHORTHAND */
transition: width 750ms ease-out 250ms;
```

- At least `transition-property` and `transition-duration` have to be set.
- `ease-in` — starts slow, accelerates quickly, stops abruptly
- `ease-out` — begins abruptly, slows down, and ends slowly
- `ease-in-out` — starts slow, gets fast in the middle, and ends slowly
- `linear` — constant speed throughout
- `ease` (default) is like `ease-in-out`, except it starts slightly faster than it ends.

Combination

```
transition: color 1s linear,
font-size 750ms ease-in 100ms;
```

The **shorthand** transition rule has one advantage over the set of separate `transition-<property>` rules: you can describe **unique transitions for multiple properties**, and combine them.

To combine transitions, add a comma (`,`) before the semicolon (`;`) in your rule.

All

```
transition: all 1.5s linear 0.5s;
/*OR in separate properties*/
transition-property: all;
```

`all` means every value that changes will be transitioned in the same way. It can also be given to the `transition-property`.