Workshop SASS for BEM development

by Vittorio Vittori

Hi, I’m Vittorio

I do frontend stuff @ **ideato**

Mainly **CSS** and **HTML**

vit.to/about

github.com/vitto

linkedin.com/in/vittoriovittori

Workshop contents

Theory

- What is **BEM**
- What to **DO** and **NOT** to do
- **Exercise:** Let's **draw** some BEM
- Some **CSS** and **SASS**
- What to **DO** and **NOT** to do

Practice

- Installation
- Let's make some **layout**
- Now it's **components** time
- Scalability conclusions

BEM

Block **E**lement **M**odifier

A brief history

BEM was created at **Yandex** by **Vitaly Harisov** as main author

en.bem.info/methodology

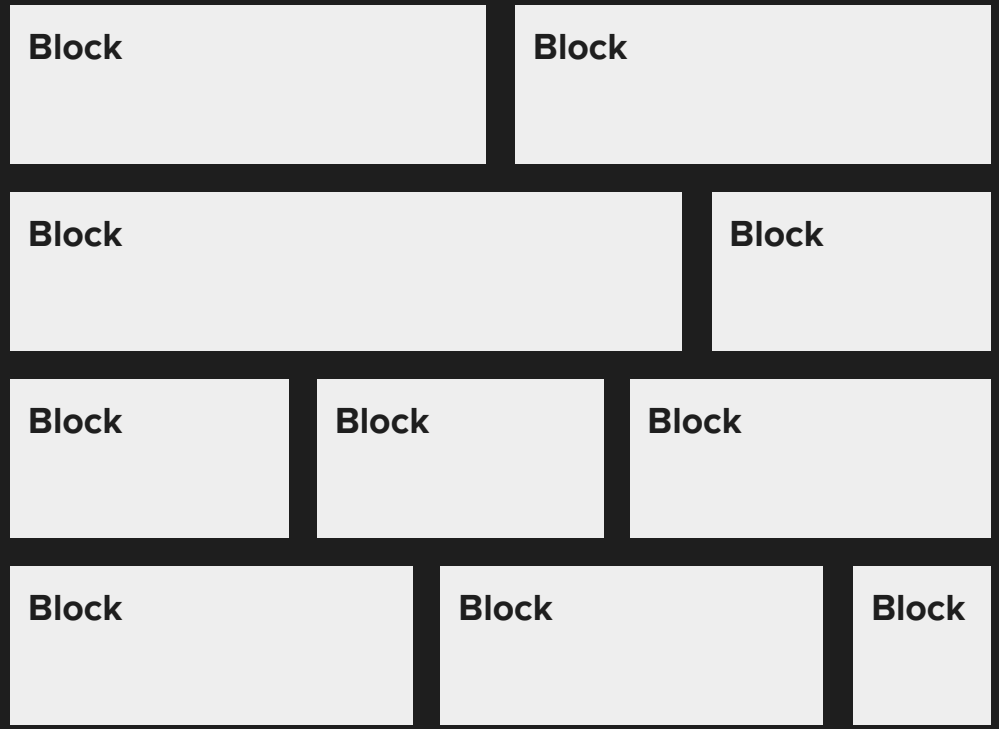
assortment.io/posts/introducing-bem-css-naming-convention

smashingmagazine.com/2016/06/battling-bem-extended-edition-common-problems-and-how-to-avoid-them/

Block

Encapsulates a standalone entity that is meaningful on its own. While blocks can be nested and interact with each other, semantically they remain equal; there is no precedence or hierarchy. Holistic entities without DOM representation (such as controllers or models) can be blocks as well.

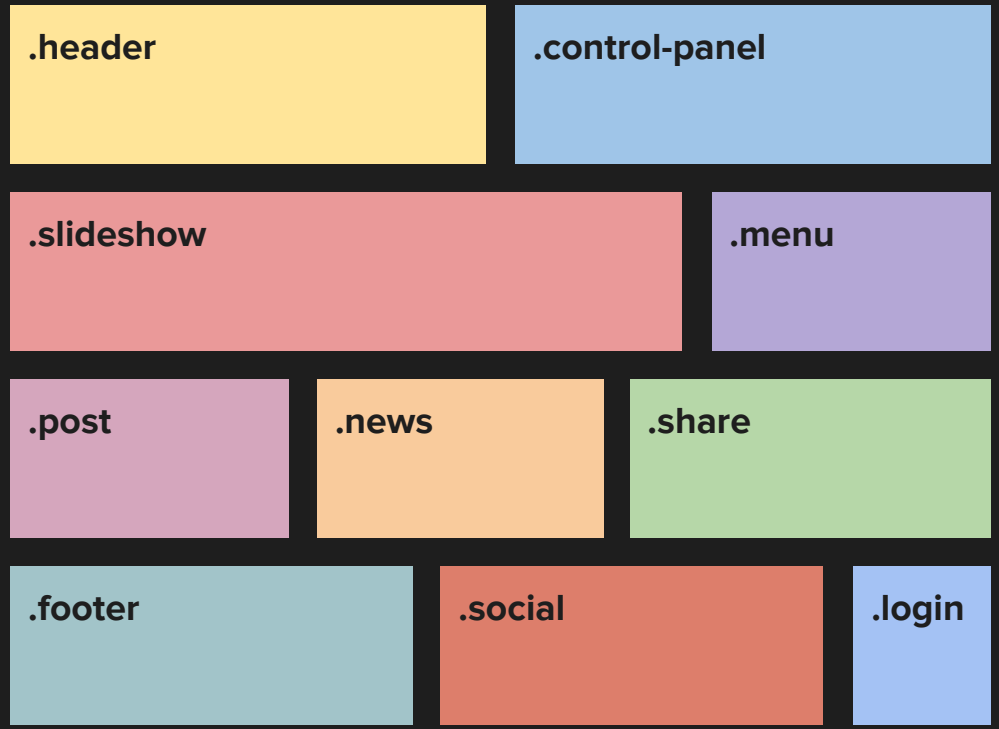
getbem.com/naming



Block

Encapsulates a standalone entity that is meaningful on its own. While blocks can be nested and interact with each other, semantically they remain equal; there is no precedence or hierarchy. Holistic entities without DOM representation (such as controllers or models) can be blocks as well.

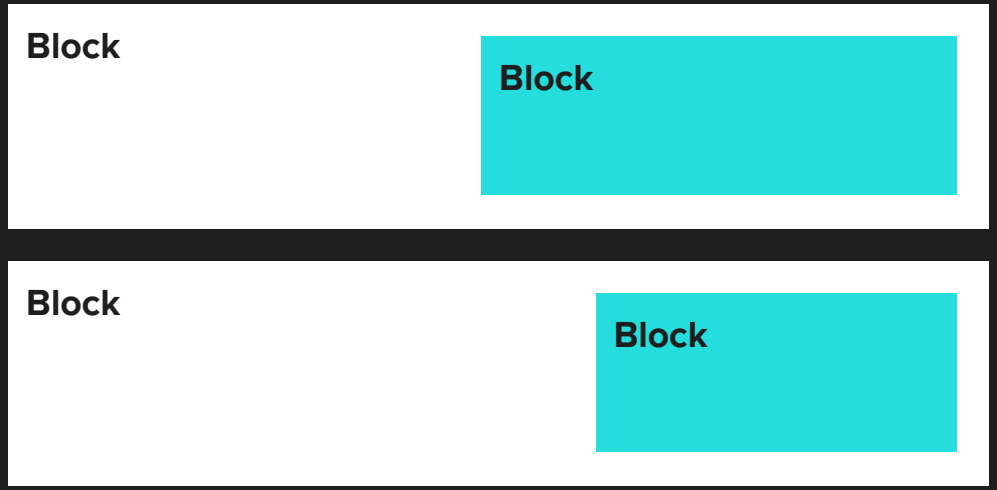
getbem.com/naming



Block

Encapsulates a standalone entity that is meaningful on its own. While blocks can be nested and interact with each other, semantically they remain equal; there is no precedence or hierarchy. Holistic entities without DOM representation (such as controllers or models) can be blocks as well.

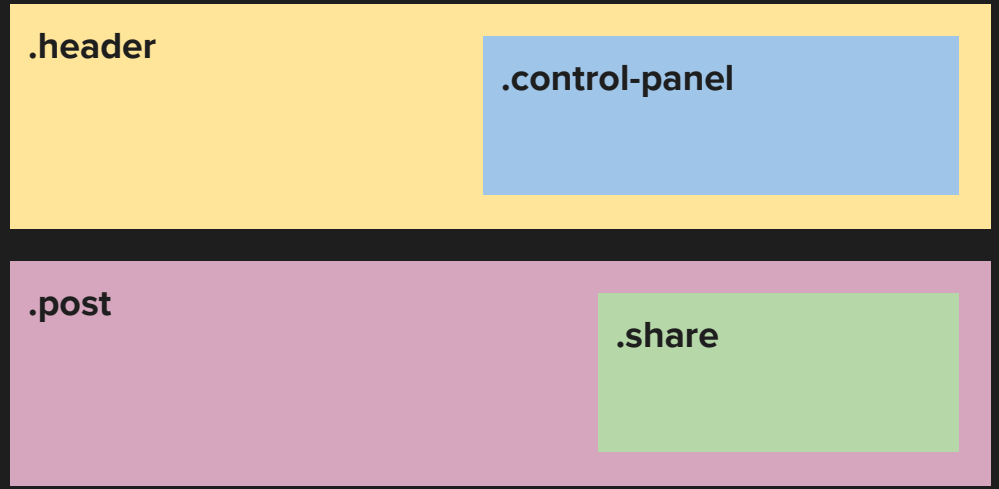
getbem.com/naming



Block

Encapsulates a standalone entity that is meaningful on its own. While blocks can be nested and interact with each other, semantically they remain equal; there is no precedence or hierarchy. Holistic entities without DOM representation (such as controllers or models) can be blocks as well.

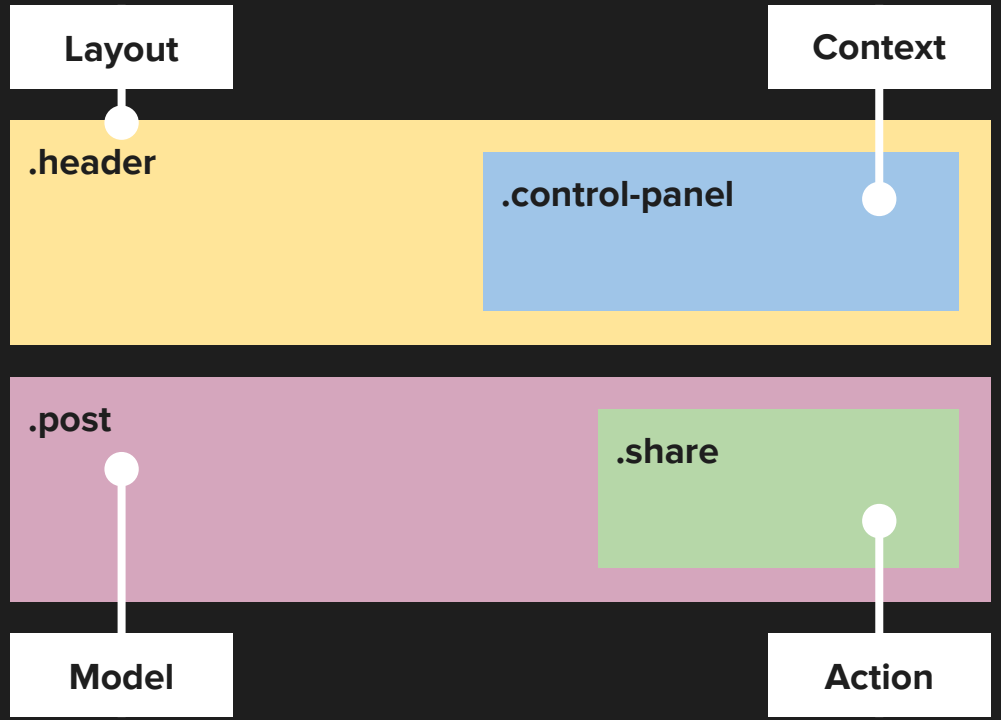
getbem.com/naming



Block

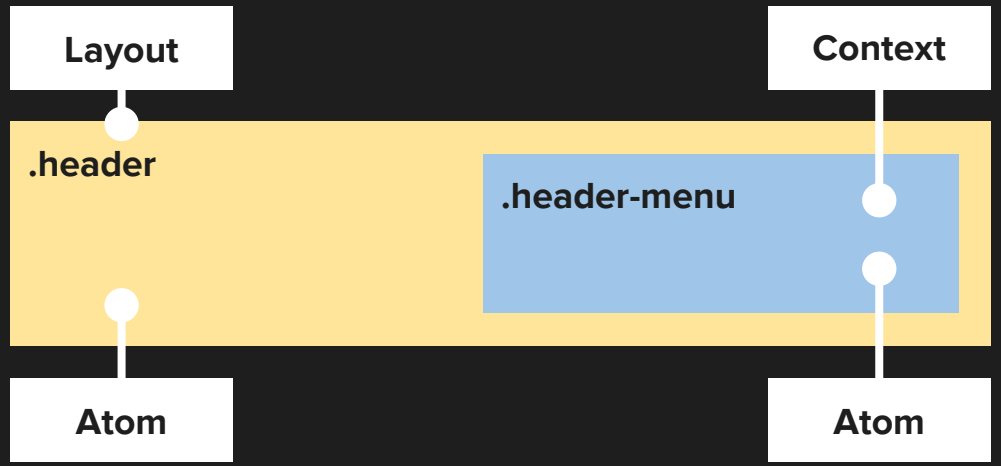
Encapsulates a standalone entity that is meaningful on its own. While blocks can be nested and interact with each other, semantically they remain equal; there is no precedence or hierarchy. Holistic entities without DOM representation (such as controllers or models) can be blocks as well.

getbem.com/naming



Block

Be atomic, if you find a context, probably this is a block

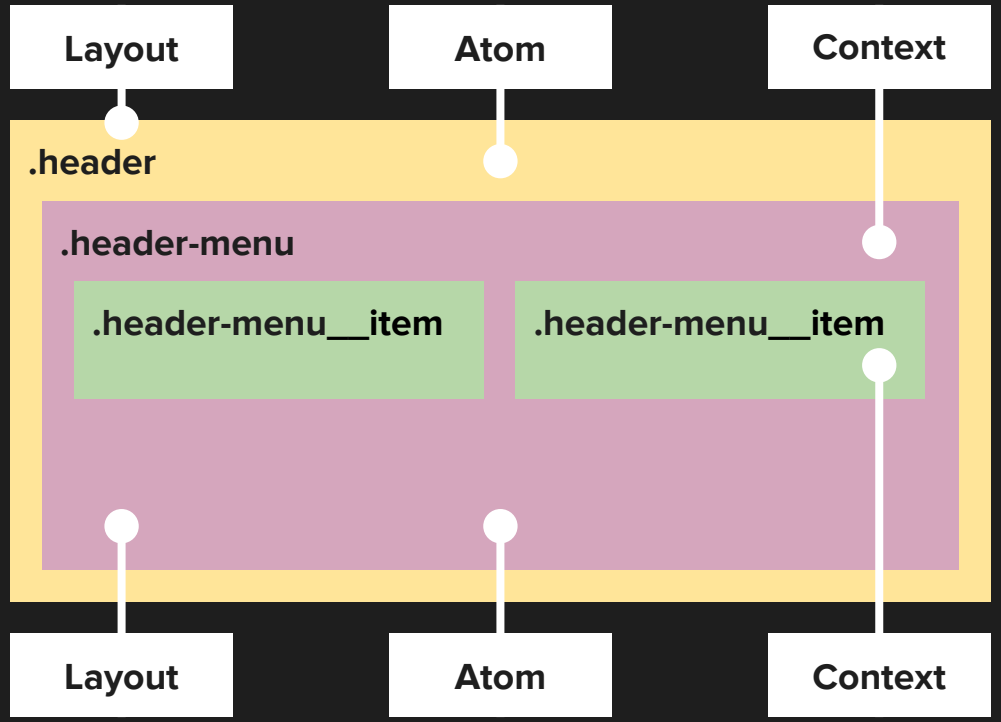


Naming convention

block-name	.control-panel
__element-name	.control-panel__button
--modifier-name	.control-panel__button--login
	.control-panel--not-logged

Block

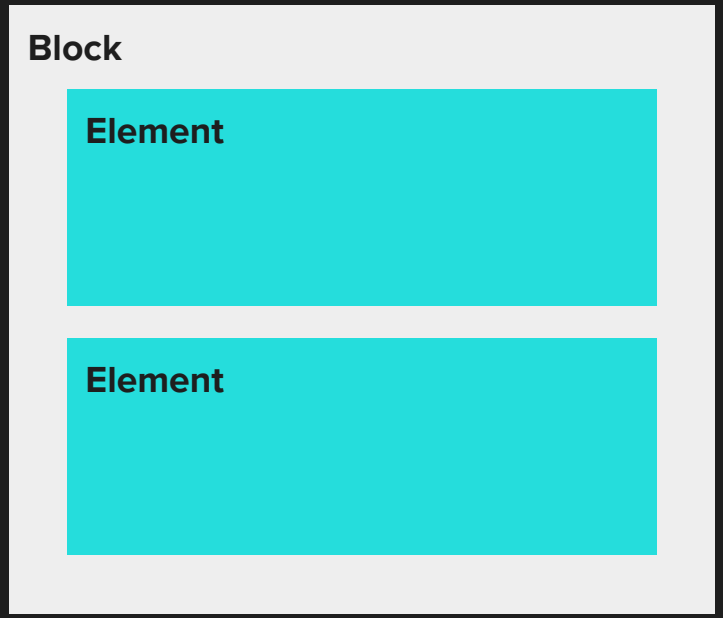
Be atomic, if you find a context, probably this is a block



Element

Parts of a block and have no standalone meaning. Any element is semantically tied to its block.

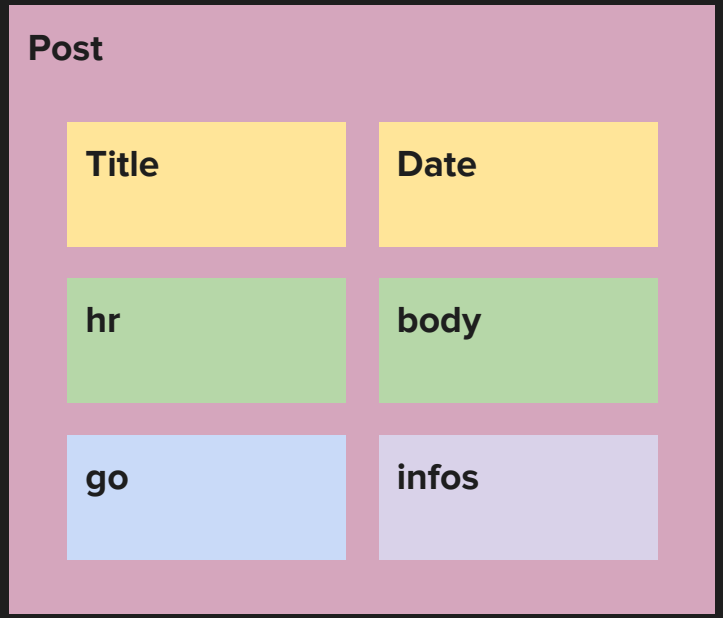
getbem.com/naming



Element

Parts of a block and have no standalone meaning. Any element is semantically tied to its block.

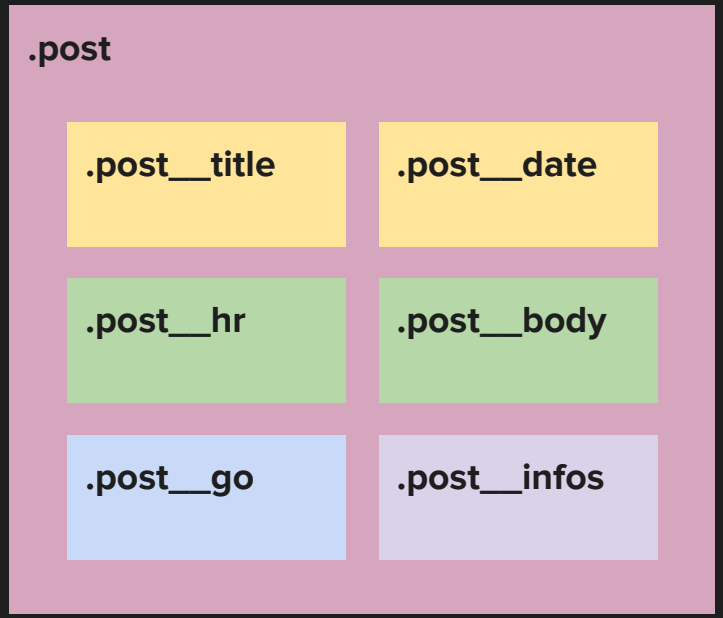
getbem.com/naming



Element

Parts of a block and have no standalone meaning. Any element is semantically tied to its block.

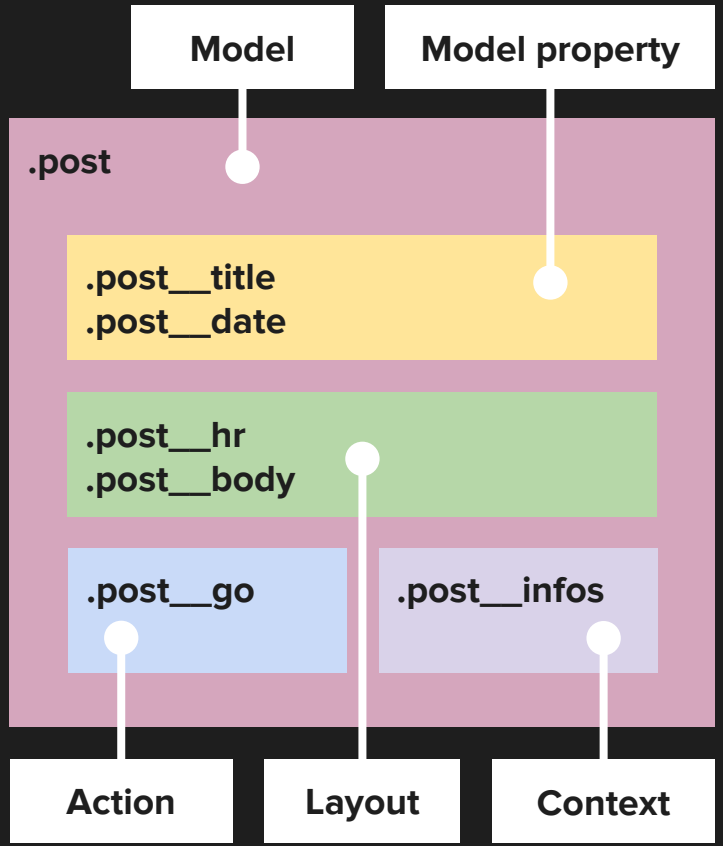
getbem.com/naming



Element

Parts of a block and have no standalone meaning. Any element is semantically tied to its block.

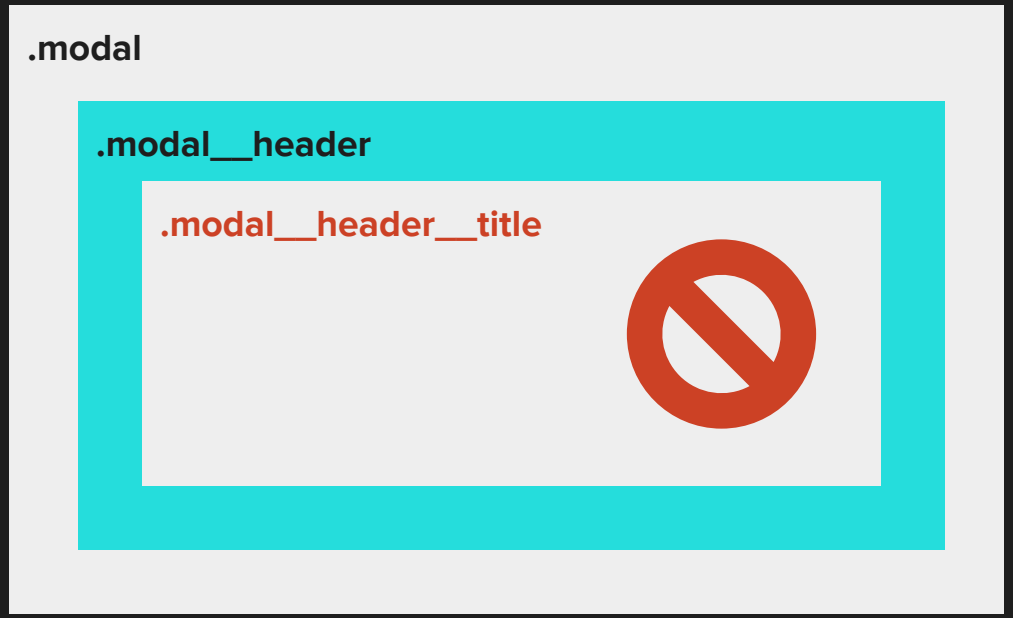
getbem.com/naming



Don’t

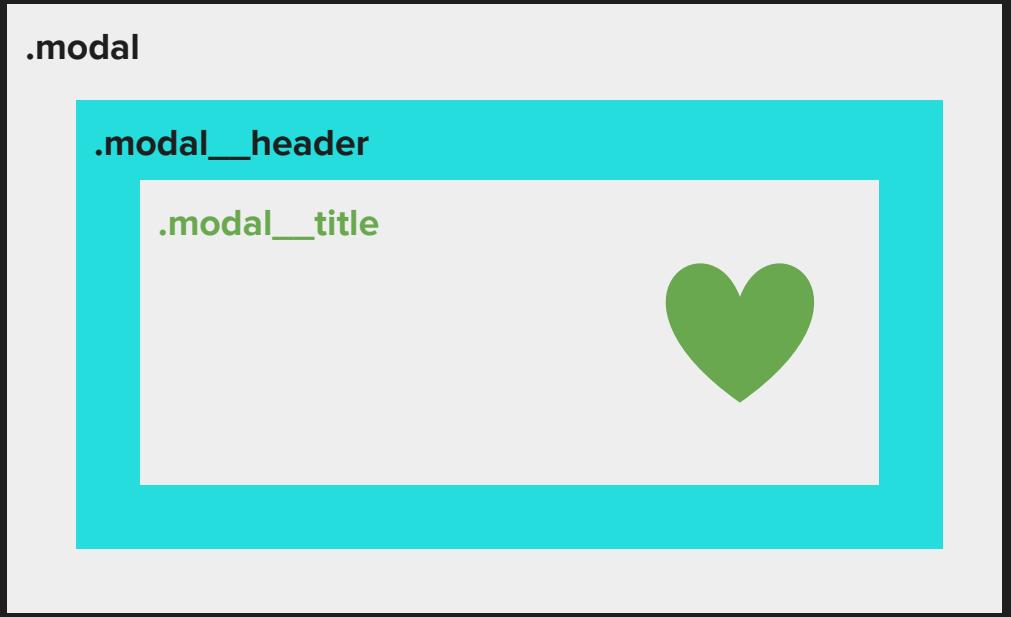
Don’t use deeper nesting. BEM is designer to keep stuff simple and clean.

Only blocks has elements, there aren’t element’s elements.



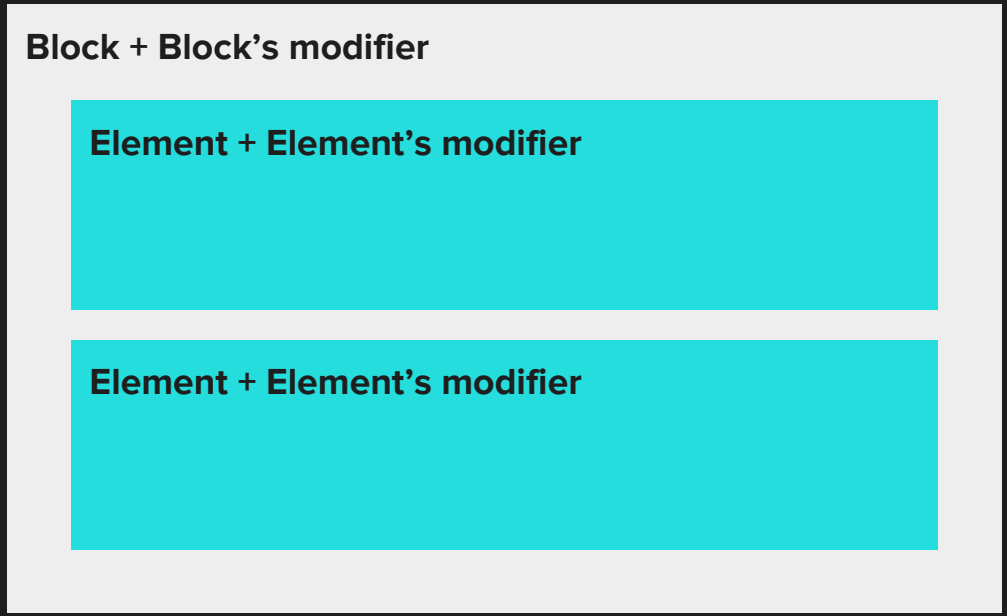
Do

If elements are nested, just nest them without use naming concatenation.



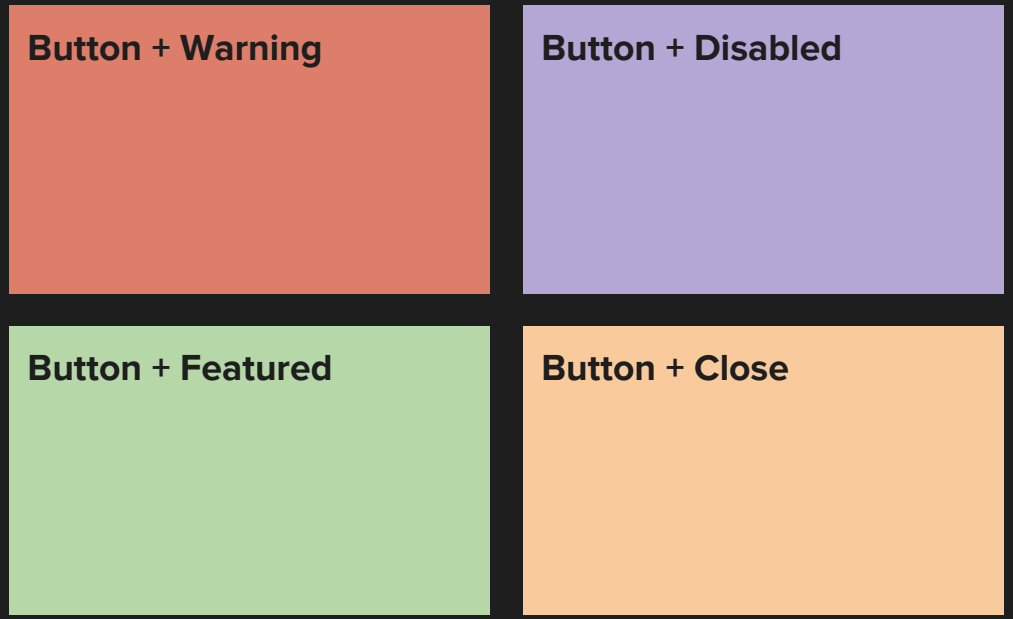
Modifier

Flags on blocks or elements. Use them to change appearance, behavior or state.



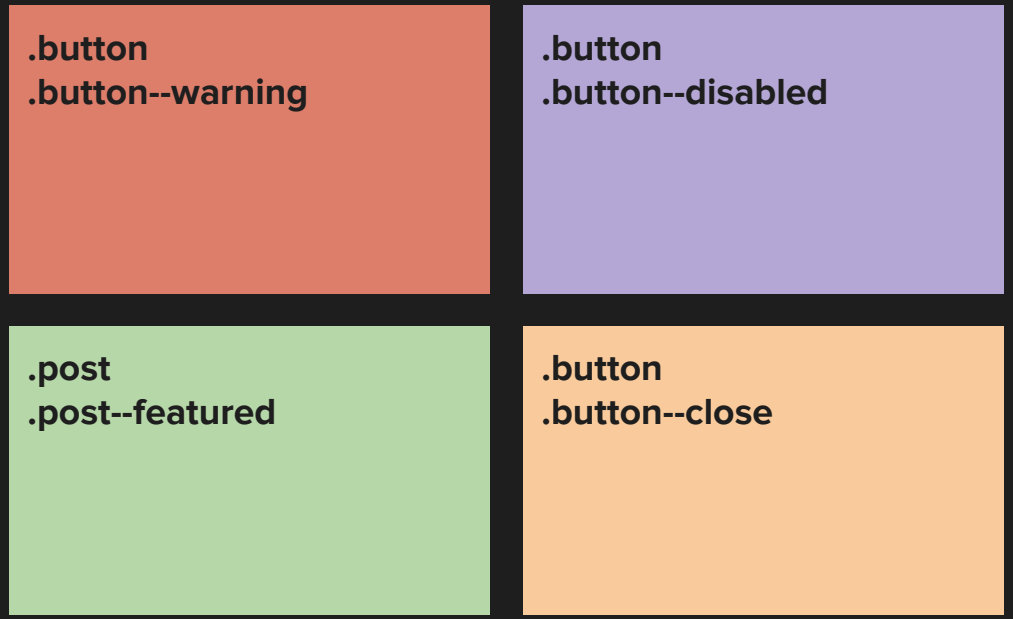
Modifier

Flags on blocks or elements. Use them to change appearance, behavior or state.



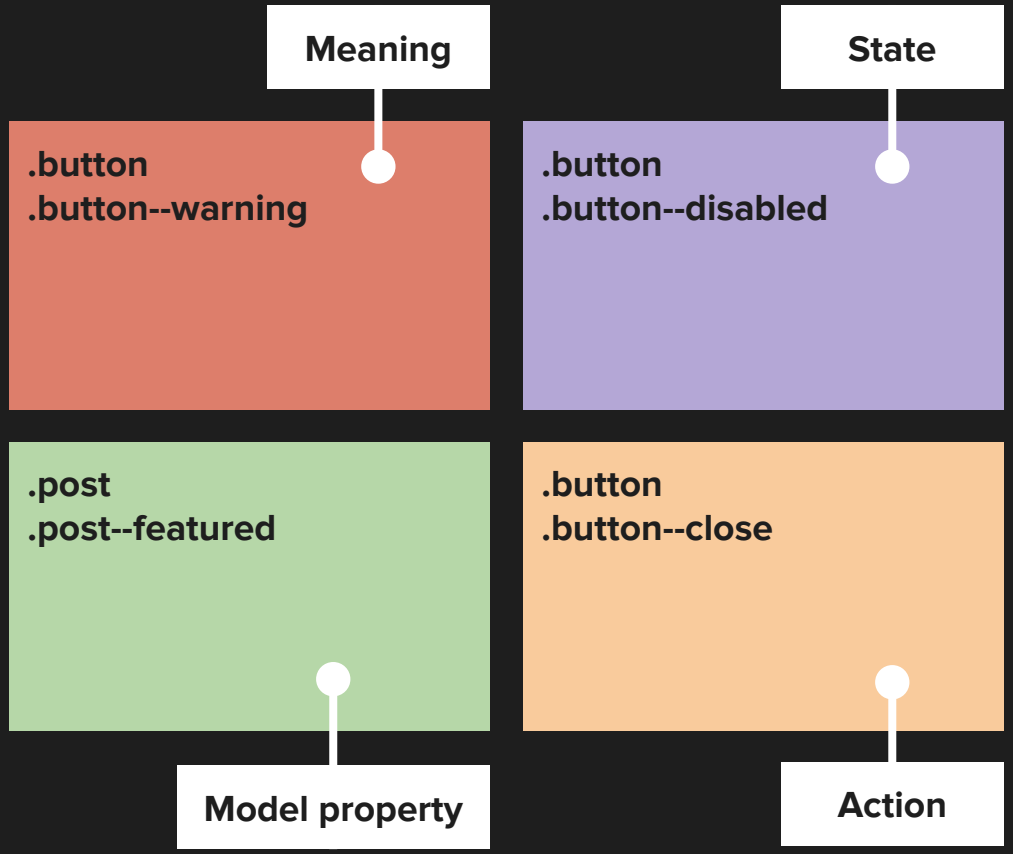
Modifier

Flags on blocks or elements. Use them to change appearance, behavior or state.



Modifier

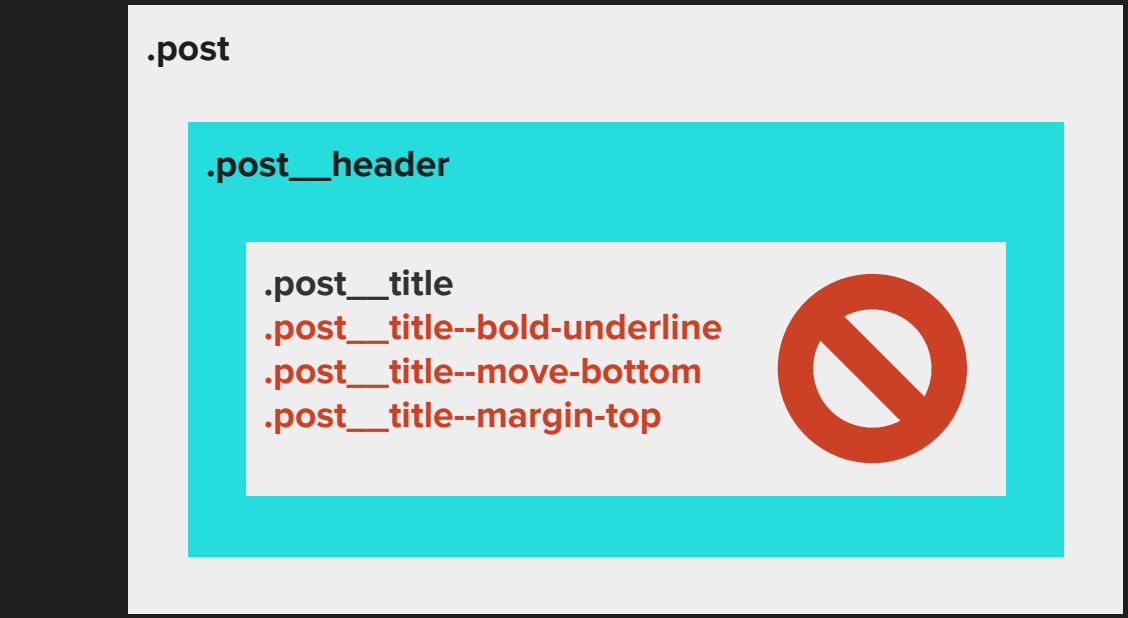
Flags on blocks or elements. Use them to change appearance, behavior or state.



Don’t

Don’t use style infos as modifiers, doesn’t help devs to understand how layout element works.

If this happens, it means you need more infos about the project.



Do

Use meaning by state, model property or state and reduce style naming on modifiers when you can.

Don’t

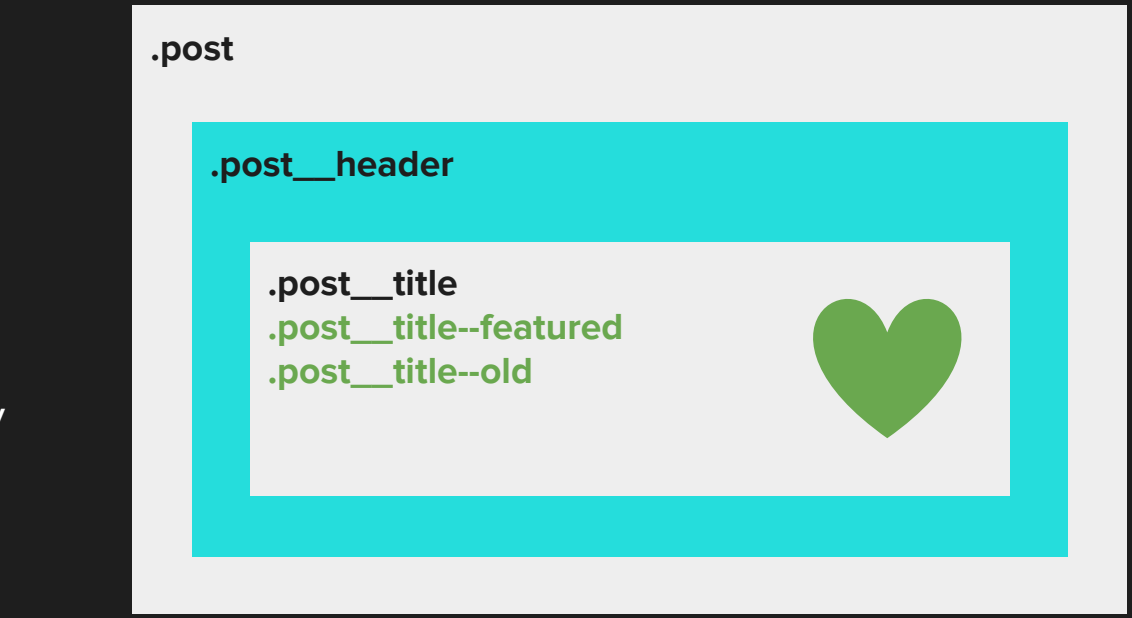
Don’t use alone modifiers. They suffers of abandonment syndrome.

Modifiers are meant to modify, not to be.

Do

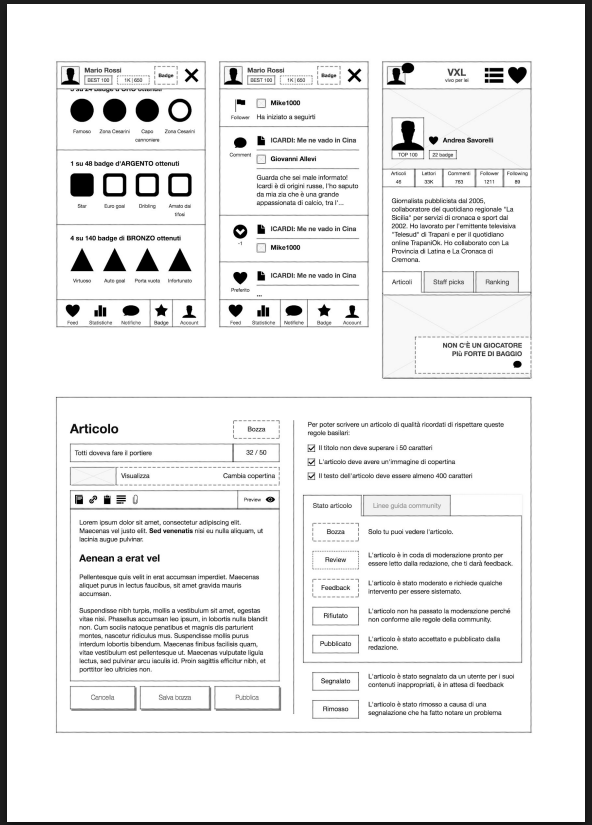
Modifiers are meant to modify things, not to be things.

Every modifier has it’s block or element and they can be concatenated.



Exercise

Let’s find blocks on this paper.



CSS

Cascading Style Sheet

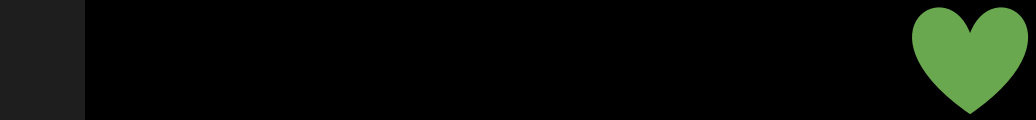
Don’t

Don’t nest selectors in CSS, doesn’t give anything more of what we need, just makes overriding more difficult.

```
// CSS

.modal {
  background-color: rgba(0, 0, 0, 0.85);
}

.modal .modal__window {
  background-color: white;
}
```

Do

Keep selectors specificity as low as you can, when you can.



```
// CSS

.modal {
  background-color: rgba(0, 0, 0, 0.85);
}

.modal__window {
  background-color: white;
}
```

Don’t

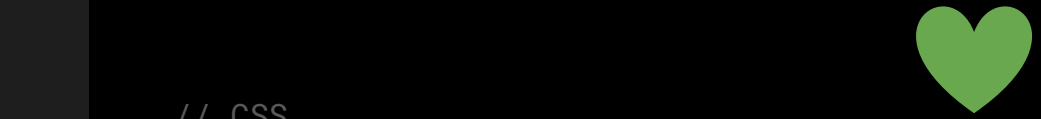
Every day, when someone mixes coding styles and country languages, somewere in the world, a good front-end developer dies.

```
// CSS

.modal {
  background-color: rgba(0, 0, 0, 0.85);
}

.modal__finestra {
  background-color: white;
}

.modal__Pulsante--Close {
  background-color: black;
  color: white;
}
```

Do

Keep selectors specificity as low as you can, when you can.

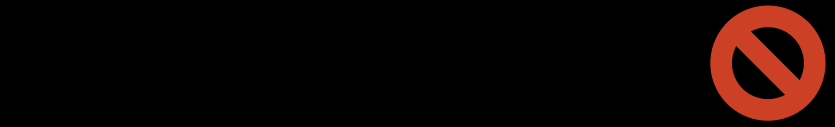
Consistency is the key to work inside a team.



```
// CSS

.modal {
  background-color: rgba(0, 0, 0, 0.85);
}

.modal__window {
  background-color: white;
}

.modal__button--close {
  background-color: black;
  color: white;
}
```

Don’t

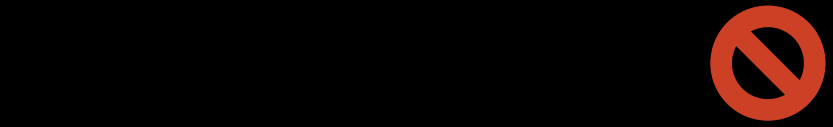
Avoid **!important**, it should be used only on modifiers, if needed.



```
// CSS

.modal {
  background-color: black !important;
}

.modal--problem {
  background-color: red;
}
```

Don’t

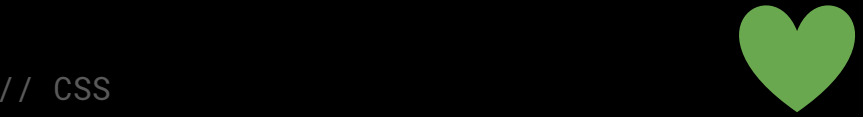
Avoid **!important** on modifiers, if they don’t give anything more to your code.



```css
// CSS

.modal {
  background-color: black;
}

.modal--problem {
  background-color: red !important;
}
```

Do

The **!important** is not the enemy, it should be used just when needed.



```
// CSS

.modal {
  background-color: black;
}

.modal--problem {
  background-color: red !important;
}

@media (max-width: 768px) {
  .modal {
    background-color: blue;
  }
}
```

SASS

Syntactically Awesome Style Sheets

BEM coding

Write code as simple as you can.

sassmeister.com

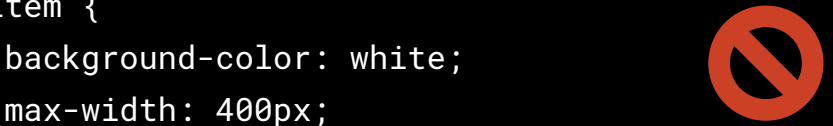
```
.item {
  background-color: white;
  max-width: 400px;
  width: 100%;

  &--wide {
    max-width: 600px;
  }

  &__title {
    color: black;
    font-size: 24px;
    font-weight: 700;

    &--warning {
      color: red;
    }
  }
}
```

Don’t

Nesting is your enemy.
This outputs a monster.

sassmeister.com

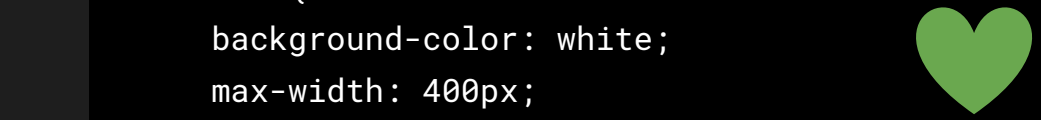
```
.item {
  background-color: white;
  max-width: 400px;
  width: 100%;

  &__title {
    color: black;
    font-size: 24px;
    font-weight: 700;

    &--warning {
      color: red;
    }

    &__icon {
      margin-right: 10px;
    }
  }
}
```



Do

BEM naming convention is your friend.

```
.item {
  background-color: white;
  max-width: 400px;
  width: 100%;

  &__title {
    color: black;
    font-size: 24px;
    font-weight: 700;

    &--warning {
      color: red;
    }
  }

  &__icon {
    margin-right: 10px;
  }
}
```



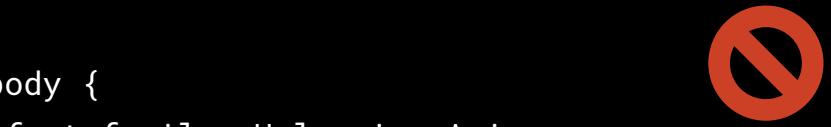
How decrease nesting errors?

Write **SASS mixins** that helps you to keep code clean.

Go to GitHub and start play:
github.com/vitto/workshop-bem-and-sass#play-examples

Don’t

Do not duplicate code, it can became an hell when you need to scale it.

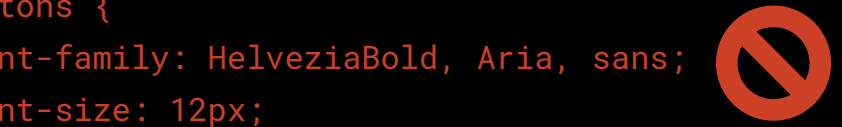


```
.body {
  font-family: Helvezia, Aria, sans;
  font-weight: 400;
}

.button {
  font-family: HelveziaBold, Aria, sans;
  font-size: 12px;
  font-weight: 700;
}

.title {
  font-family: HelveziaBold, Aria, sans;
  font-size: 18px;
  font-weight: 700;
}
```

Don’t

Do not exagerate with refactoring or you’ll just write code twice.

```
%buttons {
  font-family: HelveziaBold, Aria, sans;
  font-size: 12px;
  font-weight: 700;
}

%titles {
  font-family: HelveziaBold, Aria, sans;
  font-size: 18px;
  font-weight: 700;
}

.button {
  @extend %buttons;
}

.title {
  @extend %titles;
}
```



Don’t

Do not exagerate with refactoring or you’ll just write code twice.

Be forced to override properties is a bad smell of bad scaling.

```
%buttons {
  border-radius: 8px; ←
  font-family: HelveziaBold, Aria, sans;
  font-size: 12px; ←
  font-weight: 700;
}

.button {
  @extend %buttons;
}

.button-nice {
  @extend %buttons;
  border-radius: 0; ←
  font-size: 16px; ←
}
```

Do

When you find duplicates, it’s time to make some refactoring.

```
%bold {
  font-family: HelveziaBold, Aria, sans;
  font-weight: 700;
}

.body {
  font-family: Helvezia, Aria, sans;
  font-weight: 400;
}

.button {
  @extend %bold;
  font-size: 12px;
}

.title {
  @extend %bold;
  font-size: 18px;
}
```

Question	Answer
Have I done something wrong due to the fact I need to do some refactoring to my code?	No, it’s ok, it’s the right way a developer does, organize code when needed.
Should I write my code scalable proof by the beginning of the project?	Only if your project needs to be scalable, this doesn’t mean write bad code, but write the code you need.

Question

Should I be generic when I write a component? So using something like .grid for the layout and use it everywhere in the project?

Answer

The more you are generic, the more you need to make it flexibile. This is a bad smell. A component with a lot of purposes is a bad component. If you notice you are adding a lot of code, maybe it’s time to split it on two different grid components.

Pros

- Your project is scalable
- Devs will understand how the HTML template works very fast
- Just one level class selectors
- Conflict proof selectors
- Flexible end reusable components
- Nice for teams

Cons

- It’s time consuming
- It’s hard to naming things
- Verbose selectors naming

Now go to the project

And let’s make some code

Thank you to be here

Have a nice  2018!