

**B** Academy  
**RO**



**Manual**

**BEM**

## Attribute

# class

- ◆ In HTML, a **class** is a label (tag) for a group of elements.
- ◆ In CSS, a **class** is a tool for styling multiple elements at once.

### HTML:

```
<button class="button">Button</button>  
<a class="button">Link</a>
```

### CSS:

```
.button {  
    background-color: red;  
}
```

Attribute

# class

- ✓ Group elements and style them all at once
- ✓ Less code duplication
- ✓ Easier to update styles in one place

A good class name should answer the question:

**"What does this element do?"**

# What is naming methodologies

A **naming methodology** is a set of rules that help name classes in CSS so that the code is:

1. **Understandable.** You open the project a month later and immediately understand what does what.
2. **Consistent.** If each developer writes by their own rules, the code becomes a mess. The methodology saves us from “creative chaos.”
3. **Scalable.** The project has 10 buttons with different colors? Great, you immediately see which classes are responsible for each.

.this-is-catalog-card-item

.big-red-button-why-is-this-so-long

```
<div class="product-card product-card--highlighted">  
    
  <h2 class="product-card__title">Item name</h2>  
  <button class="product-card__button  
product-card__button--disabled">Buy</button>  
</div>
```



# Popular Naming Methodologies

- **OOCSS** — Separating container and content with CSS “objects”
- **SMACSS** — Style-guide to write your CSS with five categories for CSS rules
- **SUITCSS** — Structured class names and meaningful hyphens
- **Atomic** — Breaking down styles into atomic, or indivisible, pieces

# OOCSS (Object-Oriented CSS)

OOCSS says: "*Let's think of UI elements as objects.*" It follows two main principles:

1. **Separate structure from appearance.**

Instead of using a single class like `.red-button`, you break it down into:

- `.button` (defines the button's structure).
- `.button-red` (defines its color).
- **Structure** is the framework of an element: its size, positioning, and behavior.
- **Appearance** is the styling: colors, fonts, and borders.

2. **Reusability.**

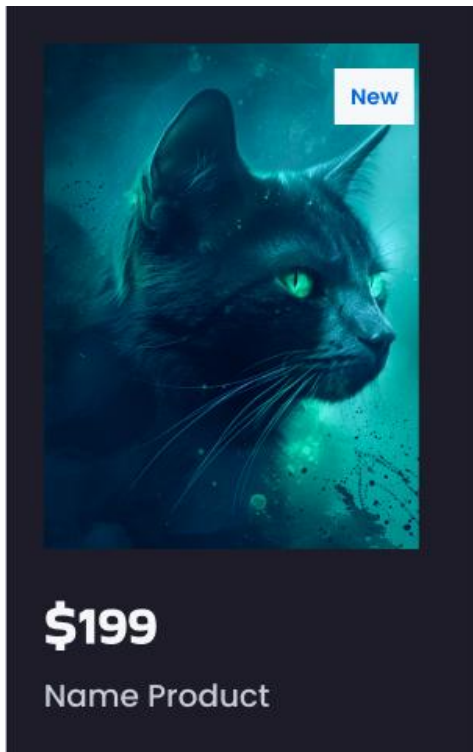
Components are designed to be used in different places without duplicating code.

It's like a construction set: you have basic building blocks, and how they are styled is determined by modifiers.

## Advantages of OOCSS

- ✓ **Reduces code duplication.**
- ✓ **Easy to make changes** – modify a class, and everything updates automatically.
- ✓ **Improves readability** – you can instantly see which class controls what.

# OOCSS (Object-Oriented CSS)



Separate structure from appearance

Reusability

```
<div class="card card--highlighted">
  
  <h2 class="text text--bold">Product name</h2>
  <p class="text text--small text--muted">New</p>
  <button class="button button--success">Buy</button>
</div>
```



# SMACSS (Scalable and Modular Architecture for CSS)

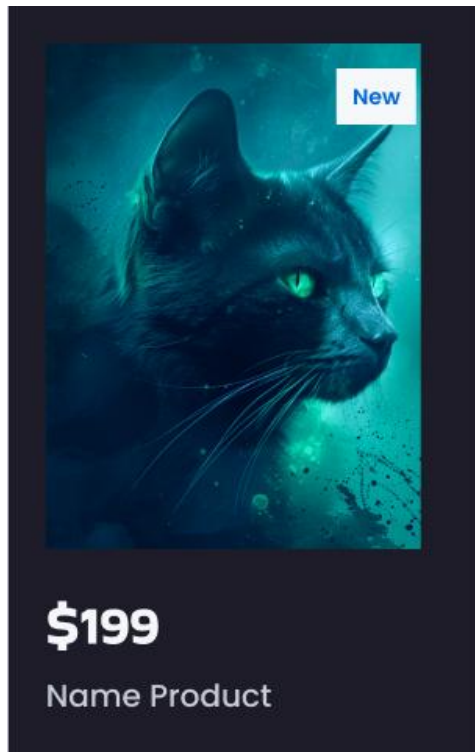
SMACSS is a methodology for those who love order. It organizes CSS into five categories:

1. **Base Styles.**
  - The foundation of the project, including margin and padding resets.
  - Example: Reset styles (`reset.css`) or default styles for HTML elements (`body`, `h1`, `a`).
2. **Layout Rules.**
  - Styles that define the **structure** of the page.
  - Used for positioning large sections like the **header**, **footer**, or **sidebars**.
3. **Modules.**
  - Reusable, independent components like cards or buttons.
  - Examples: Buttons, product cards, menus – elements that can be used multiple times across the site.
4. **States.**
  - Temporary changes in the UI (e.g., on hover or during loading).
  - Examples: Active elements, error states, loading indicators.
5. **Theme.**
  - Defines variations in the site's appearance, such as light or dark modes.

## Is it complicated?

Yes! But once implemented, your CSS will be as **organized as a pharmacy shelf**.  

# SMACSS (Scalable and Modular Architecture for CSS)



<b>Base Styles</b>	base.css
<b>Layout Rules</b>	layout.css
<b>Modules</b>	module.css
<b>States</b>	state.css
<b>Theme</b>	theme.css

```
<div class="product-card is-highlighted">
  
  <h2 class="product-card__title">Product name</h2>
  <p class="product-card__label">New</p>
  <button class="button">Buy</button>
</div>
```

# Atomic CSS

Atomic CSS is a **functional CSS methodology** that focuses on creating **single-purpose, utility-first classes**. Instead of writing component-specific styles, Atomic CSS encourages using pre-defined classes to **apply styles directly in HTML**, keeping the CSS minimal and highly reusable.

## Key Characteristics of Atomic CSS:

1. **Single-purpose classes:** Each class is responsible for **one specific style rule** (e.g., `.text-center`, `.bg-blue`).
2. **No component-based CSS:** Instead of writing styles for `.card` or `.button`, you apply multiple atomic classes to elements.
3. **Encourages inline styling via class names:** Styling is done in **HTML**, rather than writing separate CSS rules for components.
4. **Minimal CSS file size:** By reusing predefined classes, styles are not duplicated, reducing unnecessary CSS bloat.
5. **Faster rendering:** Since styles are defined globally and reused, the browser caches them efficiently, improving performance.

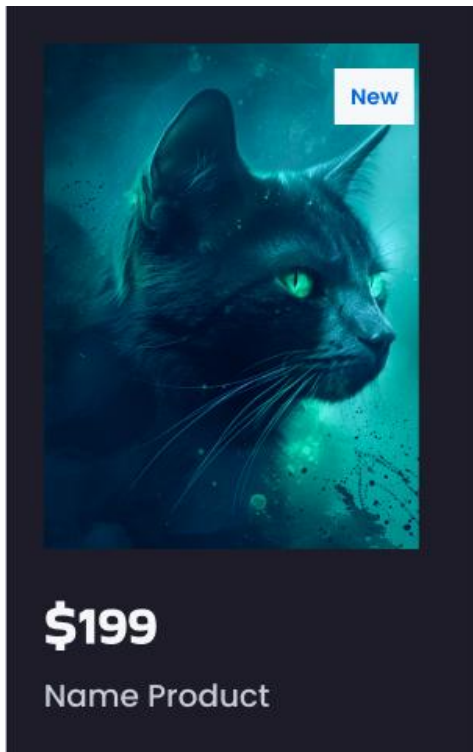
## Advantages of Atomic CSS

- ✓ **Faster development** – No need to write custom styles for each component.
- ✓ **Smaller CSS files** – Styles are reused, reducing redundancy.
- ✓ **Easier maintenance** – You never have to hunt down component styles in a massive CSS file.
- ✓ **Performance boost** – Global utility classes get cached, improving load times.

## Criticism of Atomic CSS

- **Less semantic HTML** – The meaning of elements is **determined by class names**, which may reduce readability.
- **Long class names in HTML** – Since multiple classes are used per element, HTML can become cluttered.
- **Requires a mindset shift** – Developers used to component-based CSS may find this approach unconventional.

# Atomic



## Single-purpose classes

```
<div class="bg-black padding-20 margin-20">  
    
  <h2 class="fz-24 text-left">Product name</h2>  
  <p class="fz-16 absolute-top-right-corner">New</p>  
  <button class="fz-16 bg-blue text-centered">Buy</button>  
</div>
```

Paradigm	Description	Structure/Principles	Example	Key Benefits	Drawbacks	Common Use Cases
<b><u>BEM</u></b>	Block Element Modifier methodology for creating reusable components.	Block, Element, Modifier	<pre>.menu { } .menu__item { } .menu__item--active { }</pre>	Consistency, reusability, modularity	Can become verbose, naming conventions required	Component-based design, large projects
<b><u>OOCSS</u></b>	Object-Oriented CSS separates structure and skin for reusable visual objects.	Separation of Structure and Skin and Container and Content	<pre>.box {   width: 200px;   padding: 10px; } .box-blue {   background-color: blue; }</pre>	Reusability, modularity, maintainability	Can lead to more complex HTML structure	Large-scale applications
<b><u>SMACSS</u></b>	Scalable and Modular Architecture for CSS, focusing on consistent and modular styles.	Base, Layout, Module, State, Theme	<pre>.m-button { } .l-header { } .is-active { }</pre>	Scalability, modularity, ease of maintenance	Can be complex to implement initially	Large-scale applications, complex projects
<b><u>Atomic CSS</u></b>	Style sheet architecture using small, specific classes for styling.	Single responsibility, reusability, consistency	<pre>.text-center {   text-align: center; } .bg-blue {   background-color: blue; }</pre>	Reusability, maintainability, encapsulation	Increased HTML verbosity, learning curve	Large-scale applications, rapid prototyping

# BEM

**BLOCK / ELEMENT / MODIFIER**

# HISTORY of BEM

The methodology originated within the Russian company **Yandex**. Its history begins in 2009 when the development team was looking for a way to make the code more structured, readable, and maintainable for large projects.

## Why Did Yandex Need Such a Methodology?

Yandex is not only a search engine but also a multitude of services: Mail, Maps, Market, Music, etc. (essentially like Google).

Each service has its interface features, but many components are similar. For example, buttons, product cards, menus.

The BEM methodology helped Yandex:

1. **Write code consistently** across different projects.
2. **Avoid style conflicts**, especially when different teams worked on different parts of the same interface.
3. **Scale development**, as new developers could quickly understand the existing code.

## BEM Goes Global

In 2012, Yandex decided not only to use BEM within the company but also to share the methodology with the community.

Since then, BEM has become popular worldwide, especially in large projects where code support plays a key role.



**BLOCK**



**ELEMENT**



**MODIFIER**



# BLOCK

- **Definition:** an independent component that can be reused.
- **Example:** button (button), form (form).
- **What can be assigned to a block:** sizes, margins, color, background.

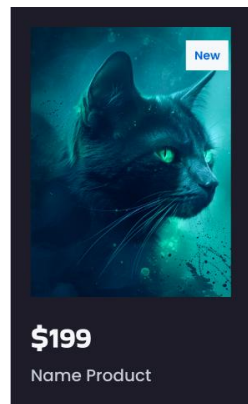
## What is a Block?

A block is an independent, self-sufficient component. It can exist without other parts of the layout, and all other elements are usually built around it.

## How to Find a Block on a Layout?

Look at the major interface parts that do not depend on others.

# BLOCK examples



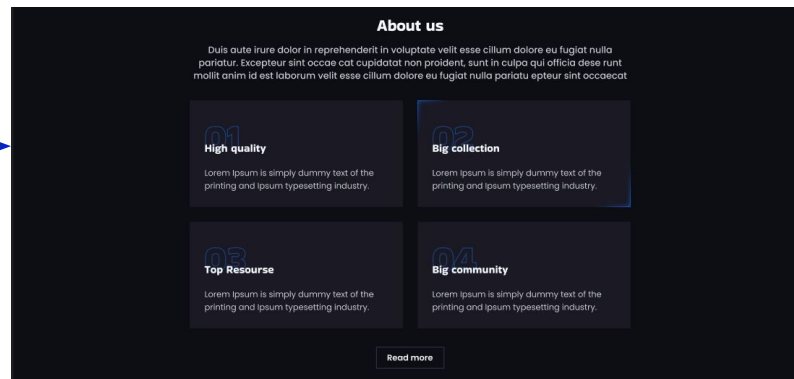
← using in different pages

↑ using on all pages

page section →



← using in many blocks





class="logo"

class="nav"

class="search-form"

class="button"

class="header"

# ELEMENT

- Definition: part of a block that cannot exist independently.
- Example: icon in a button (`button__icon`), form title (`form__title`).
- What can be assigned to an element: internal margins, font color, styling.

## What is an Element?

An element is a part of a block, without which it doesn't make sense. It's like a partner: good, but only paired with a block.

## How to Find an Element on a Layout?

These are usually internal parts of a large component.

class="nav\_\_list"

class="nav\_\_list-item"

class="nav\_\_link"

class="search-form\_\_input"

# MIX

A **mix** in BEM is a technique that allows an element or block to have multiple BEM classes at the same time. This helps when an element needs to inherit styles or behavior from multiple sources without violating the **block-element-modifier** (BEM) structure.

## Why Use Mixes in BEM?

**To combine styles from different blocks.**

- Sometimes an element belongs to multiple blocks or needs to behave as part of different components.

**To reuse styles without duplication.**

- Instead of redefining styles, you can mix in a class from another component.

**To create variations without relying too much on modifiers.**

- Modifiers should only change the appearance or behavior of a block/element, but when the same element logically belongs to two different structures, a mix makes more sense.

## When Not to Use a Mix?

✗ **Instead of a modifier:** If an element only needs a simple change in appearance, a **modifier** (**block--modifier**) is a better choice.

✗ **For nesting inside another block:** Elements should always be part of a **single block**, so avoid overusing mixes to "nest" elements into unrelated structures.



class="header\_\_search search-form"

class="header\_\_nav nav"

class="header\_\_logo logo"

class="header"

# MODIFIER

- Definition: change in the appearance or behavior of a block/element.
- Example: button with size modification (`button--large`), active menu item (`menu__item--active`).
- What can be assigned to a modifier: states (colors on hover), specific sizes.

## What is a Modifier?

A modifier is something that slightly changes the appearance or behavior of a block or element. It doesn't create something new; it just changes its "mood."

## How to Find a Modifier on a Layout?

Look at component variations. If there is something that differs from the standard block or element, it's a modifier.





class="nav\_\_link nav\_\_link--active"

class="btn btn--primary"

class="header"

# Different Ways to Define a Modifier

Yandex Approach	Gallagher Approach
_ (one underscore)	-- (two dashes)
<code>block_modifier</code>	<code>block--modifier</code>

## Why this way:

- **Convenience of working with CSS and JS.** Modifiers look like standalone classes, making it easier to add/remove them through scripts.
- **Readability.** — visually highlights the modifier, easing code perception.

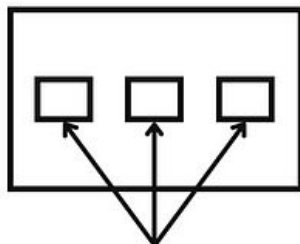
`card--highlighted` — a modifier for the `card` block that highlights the card.

# BEM IN CSS



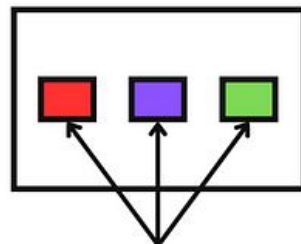
**Block**

A navigation bar with `.nav`



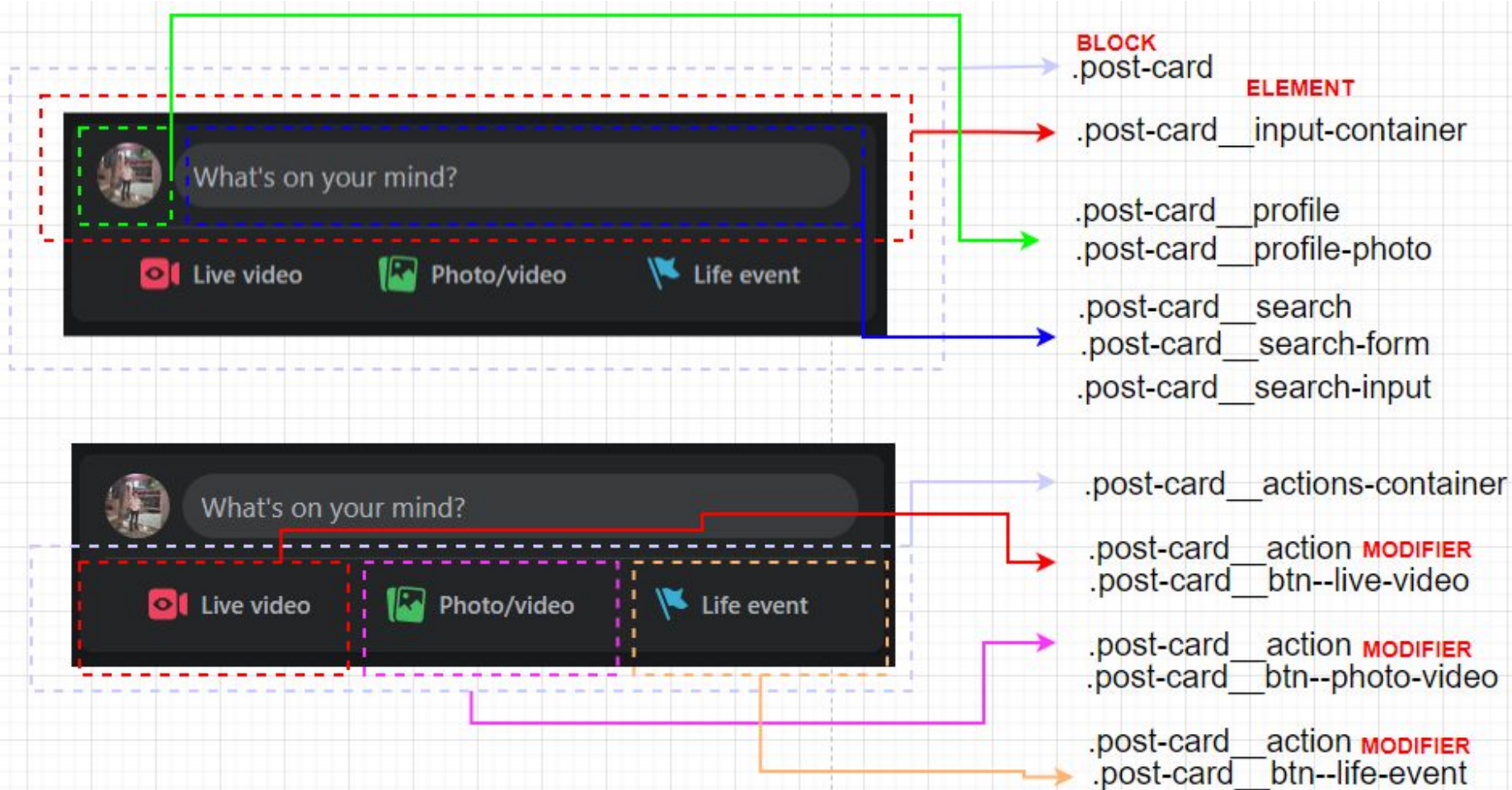
**Element(s)**

Each item in navigation bar with `.nav__item`



**Modifier**

Add property to each item by adding `.nav__item--red`, `.nav__item--purple`, `.nav__item--green`



**General  
rules**

**for class  
names**

## General rules

# class

Not recommended	Better
<ul style="list-style-type: none"><li>Use only <b>English words</b>, <b>no transliteration</b></li></ul>	
<b>krasnaya-knopka</b>	<b>error-button</b>
<ul style="list-style-type: none"><li>Written in <b>lowercase</b></li></ul>	
<b>PageHeader</b>	<b>page-header</b>

- Avoid using **numbers** in class names, the order of blocks may change

**first-element**

**product-item**

Numbers in class names can create a rigid attachment to the order of elements. For example, `.section1`, `.section2` imply that the first section will always be first, but the page structure may change. If the order changes or new blocks appear, you'll have to edit class names, complicating code maintenance.

- The Name Should Be Understandable

**btn-01**

**button--primary**

A class name should clearly reflect its purpose so that other developers (or you yourself in a few months) can quickly understand what this class is responsible for. If the name is uninformative (`.abc`, `.style1`), you'll spend time parsing the code, reducing readability and ease of maintenance.

- No spaces in class names; use **hyphens** for compound class names

**product item**

**product-item**

HTML does not support spaces in class names, and if used, the browser will perceive it as several different classes. Instead of spaces, it is customary to use dashes (-), as this makes names readable and convenient for work.

- Do not name classes **by content, style, or tag**, as they may change

**text-about-product**

**product-item\_\_text**

**red-button**

**button button--danger**

**p1**

**promo\_\_text**

If a class is named `.red-button` and the designer decides to change the text color to blue, you'll either have to change the class name (which is inconvenient) or use styles that don't match its name. Similarly, `.p1` implies that the element will always be a `<p>`, but if it needs to be replaced with a `<span>`, the name loses its meaning. It's better to use neutral and meaningful names like `.error-message` or `.primary-button`, reflecting the element's role, not its current characteristics.



**BEM**

**rules**

**for class  
names**

- Each Element Belongs Only to Its Block

`class="block1__element block2__element"`

`class="block__element"`

If one element is used in multiple blocks, it can lead to confusion. For example, if `.block1__element` also has `.block2__element`, it's unclear how these elements are related. It's better to create independent elements to avoid tight links between blocks, making code more modular and maintainable.

- Do Not Nest Elements Within Each Other Using Double Underscore

`block__element__subelement`

`block__subelement`

Deep nesting complicates code reading and maintenance. If we write `block__element__subelement`, it creates a rigid structure that's hard to change. It's better to present nested elements as independent parts of the block (`block__subelement`), making the code more flexible and understandable.

- Do Not Create Overly Long Names

`card__button--password-large-disabled`

`card__button`  
`card__button--large`  
`card__button--disabled`

Long classes make code reading and maintenance difficult. If a class name describes element nesting too thoroughly, it may indicate an overly complex structure. Instead of long and detailed names, it's better to separate styles into base classes and modifiers, simplifying reuse.

- A Modifier Cannot Be Used Without the Main Class

`class="button--large"`

`class="button button--large"`

Modifiers should be an addition to the base class, not exist separately. If you write `.button--large` without `.button`, it makes the code unstructured and confusing. It's better to always apply modifiers only with their base classes, so they logically inherit styles.

- Use Mixes, Not Modifiers, to Define Block Styles Depending on Their Location

```
class="button button--card"
```

```
class="button card_button"
```

Sometimes, the same element (e.g., button) is used in different blocks, and we want to change its styles depending on context.

Here, `button` is responsible for the basic button style, and `card__button` defines styles specific to the card. This approach makes the code cleaner because:

- `button` remains universal and reusable.
- `card__button` clearly indicates it's a button in the card without changing the `button` meaning.
- If you need to change button styles in another block, you can simply add `modal__button`, `form__button`, etc., without changing the base `button` class.

Thus, mixes allow using elements in different contexts without creating tight ties through modifiers, making the code more flexible and maintainable.

- Modifiers Should Not Change the Meaning of a Block or Element

class="button--form-submit"

class="button--large button--disabled"

A modifier is just a change in the state or appearance of an element, not its logical role. If a modifier radically changes the block's purpose, it disrupts its semantic integrity. For instance, `.button--form-submit` turns a regular button into a form submit button, but that's already a different element in meaning. Instead, modifiers should only describe the appearance (`.button-large`, `.button--disabled`), leaving the main logic unchanged.



- Avoid Global Names for Modifiers

menu\_\_item active  
card highlighted

menu\_\_item--active  
card--highlighted

Using global names like `.active` or `.highlighted` can conflict with other styles on the page. For example, `.active` can apply to menus, cards, and buttons, leading to unexpected effects. Therefore, modifiers should be tied to their block to avoid style overlaps.



 Tony Nguyen |  20 Jun 2022

## Sonic live

One September 22, 2021 A volutpat nulla eros, nulla ultrices nec nulla imperdiet...

Read more >

```
<div class="product-card">
  <img class="
    product-card__image
  ">
  <div class="
    product-card__label
  ">NFT</div>
  <div class="
    product-card__title
  ">Sonic live</div>
  <div class="
    product-card__button
    button
    button--large
    button--icon
  ">Read more</div>
</div>
```

# USE BEM FOR UI KIT

**UI Kit (User Interface Kit)** is a collection of components that can be reused in a project.

Using BEM to organize these components will make them:

1. **Universal** — easily adaptable to different tasks.
2. **Scalable** — adding new components will be simple.
3. **Maintainable** — the code will be understandable for the whole team.

## 1. Each Component is a Block

- In BEM, each independent component (e.g., button, card, input field) is considered a block.
- Blocks in a UI Kit should be isolated so they can be used in different parts of the interface.

### Example:

- Button: `button`
- Card: `card`
- Input: `input`

## 2. Elements Are Component Parts of a Block

- Elements represent subcomponents of a block that cannot exist on their own.
- In a UI Kit, elements are often used to describe the internal parts of components, such as icons, text, or titles.

### Example:

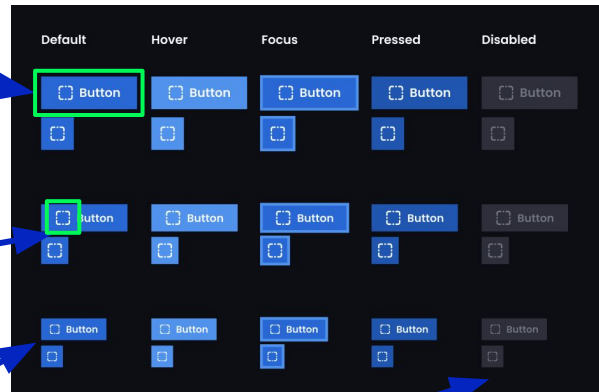
- Icon inside a button (`button__icon`)

## 3. Modifiers for Component Variations

Modifiers are used to create different states or styles of components.

These can be:

- Different sizes (`button-large`, `card--small`).
- States (`button--disabled`, `card--highlighted`).
- Themes (`button--primary`, `card--dark`).





# EXAMPLES COMPONENTS IN UI KIT

**button--**

primary / secondary / basic

large / small / medium

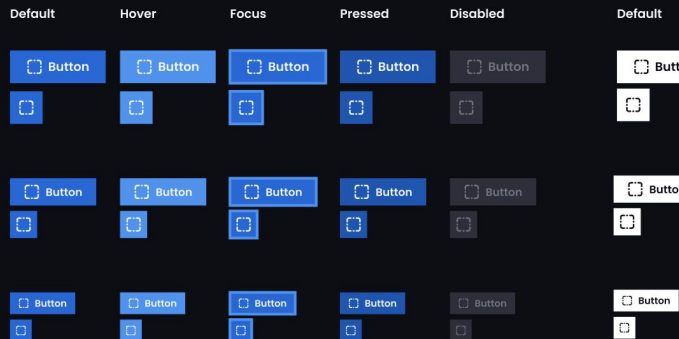
icon

active / disabled

**input--**checked / invalid / valid

**card--**sold / new / popular

## Buttons



## Checkbox

