BRO Academy

CSS Preprocessors

Lesson 9

Lesson Plan

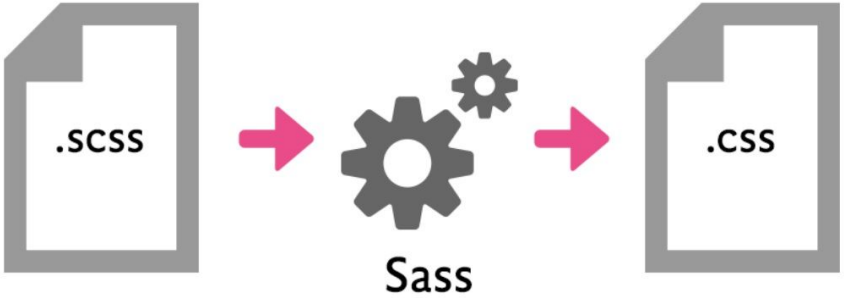
1
What is it

2
Popular preprocessors

3
Features

4
CSS vs Preprocessors

What is a CSS Preprocessor?



A **CSS preprocessor** is an extension of CSS that adds extra features to help you write cleaner and more powerful styles.

Why use one?

✅ Repeated code → hard to maintain
✅ Limited flexibility for large projects

Popular CSS Preprocessors

- Sass / SCSS – most popular and widely used (what we'll learn)
- Less – used in some older projects
- Stylus – flexible, but less common

💡 **SCSS = regular CSS + extra power** 💪

Variables

SCSS Variables

Variables store
a **reusable** value

```
$primary-color: #ff6600;
$base-font: 'Arial',
sans-serif;

.button {
    color: $primary-color;
    font-family: $base-font;
}
```

⚠️ Don’t overuse — only if value is reused or needs to be updated easily.

What do developers usually store in SCSS variables?

- Colors (color palette)
- Fonts (font settings)
- Spacing and sizes (e.g. margins, paddings, magic numbers)
- Breakpoints (for responsive design)

Any repeated or “magic” number

```
$primary-color: #ff6600;
$base-font: 'Arial',
sans-serif;
$spacing-sm: 8px;
$mobile: 480px;

.button {
    color: $primary-color;
    font-family: $base-font;
}
```

What are Magic Numbers? 🔮

💬 **A "magic number" is a hardcoded value with no explanation or meaning.**

These numbers are hard to understand, update, and maintain.

❌ **Problems:**
If someone else reads your code, they won’t understand **why** those specific values are used.

📌 **If you can't explain where a number comes from — it's probably a magic number!**

🔹 How to avoid magic numbers the right way?

✅ 1. Use variables ($variables)

✅ 2. Use relative units (em, rem, %, vh, vw)

✅ 3. Use math calculations (calc())

🔹 When are magic numbers acceptable?

Sometimes they're hard to avoid:

✅ When working with animations (e.g. `transition-delay: 0.3s;`)

✅ For complex visual effects (e.g. `clip-path: polygon(...)`)

✅ In experimental design where no variables exist yet

But whenever possible, it's better to avoid them.

Nesting

Nesting

SCSS allows you to **nest selectors and their styles** — but don't overdo it.

– classes

– pseudo-classes

– pseudo-elements

– user-actions

```
.card {

    &__title {
        font-size: 20px;
    }

    &__text {
        color: gray;
    }
}
```

Nesting order

– block styles

– element styles

– pseudo-elements

– pseudo-classes

– media

– combinator

– modifier

```
.card {
    padding: 20px;

    &:before {}
    &:hover {}
    &:first-child {}

    @media (min-width:768px) {}


    &.error {}

    &--modifier {}


    &__text {
        color: gray;
    }
}
```

&__parent selector

```
.card {

    &:hover {}
    &:before {}

    &__title {
        font-size: 20px;
    }

    &__text--active {
        color: gray;
    }
}

.card:hover {}
.card:before {}
.card__title {}
.card__text--active {}
```

SCSS Nesting & BEM Rules

📌 **Why is it better to write it separately?**

✅ Modifiers are written separately, as recommended by BEM

✅ Easier to read and maintain

✅ You can override styles more easily without issues

🚫
```
.card {
  &__title {
    font-size: 18px;

    &--big {
      font-size: 24px;
    }
  }
}
```

✅
```
.card {
  &__title {
    font-size: 18px;
  }

  &__title--big {
    font-size: 24px;
  }
}
```

Mixins

Mixins – What & When? (@mixin, @include)

Mixins help you **reuse styles with variations**.

✅ **Use mixins when:**

- Code is reused with different values
- You need logic (@if, @each)

⚠️ Don't use mixins for code that stays the same — use a class instead!

```
@mixin button($bg-color) {
  background: $bg-color;
  padding: 10px 20px;
  border-radius: 5px;
}

.btn-primary {
  @include button(blue);
}
```

What is usually placed in mixins?

- Buttons
- Responsiveness (media queries)
- Centering content (flexbox)
- Animations

```
@mixin button($bg-color) {
  background: $bg-color;
  padding: 10px 20px;
  border-radius: 5px;
}

.btn-primary {
  @include button(blue);
}
```

When to use a mixin and when to use a class?

👉 Use a **class** when the style is reused **without changes.**

✅ Good because you're reusing the same, consistent style.

👉 Use a **@mixin** when the style is **similar but needs variations**.

✅ Good because the structure repeats, but some values are different.

⚠️ **Mistakes to avoid:**

❌ Don't create a mixin for something that can be handled by a class

❌ If the styles are exactly the same, there's no point duplicating them with a mixin.

```
.btn {
  padding: 10px 20px;
  border-radius: 5px;
}

@mixin button($bg-color) {
  background: $bg-color;
  padding: 10px 20px;
  border-radius: 5px;
}

.btn-primary { @include
button(blue); }
.btn-secondary { @include
button(gray); }
```

When to use a mixin and when to use a class?

What are you doing?	`@mixin`	`.class`
Style is reused **without changes**	❌ No	✅ Yes
Style is reused with **variations**	✅ Yes	❌ No
Code is complex and should be **encapsulated**	✅ Yes	❌ No
You need to apply it to **different elements**	❌ No	✅ Yes

File system

Each BEM block = own file

💡 **Why do it this way?**

✅ **Clean code** → Each part lives in its own file
✅ **BEM-friendly** → Each block has its own file
✅ **Easier to maintain** → You can update files independently

```
📂 base/
    📄 global.scss
    📄 variables.scss
    📄 mixins.scss
📂 layout/
    📄 header.scss
    📄 footer.scss
    📄 grid.scss
📂 components/
    📄 button.scss
    📄 card.scss
📂 pages/
    📄 style.scss
    📄 index.scss
    📄 blog-page.scss
```

global.scss

for tags

```
body {}

a {}

img {}

ul {}
```

@import vs @use

SCSS allows you to split your code into multiple files.

💡 **Why use @use?**

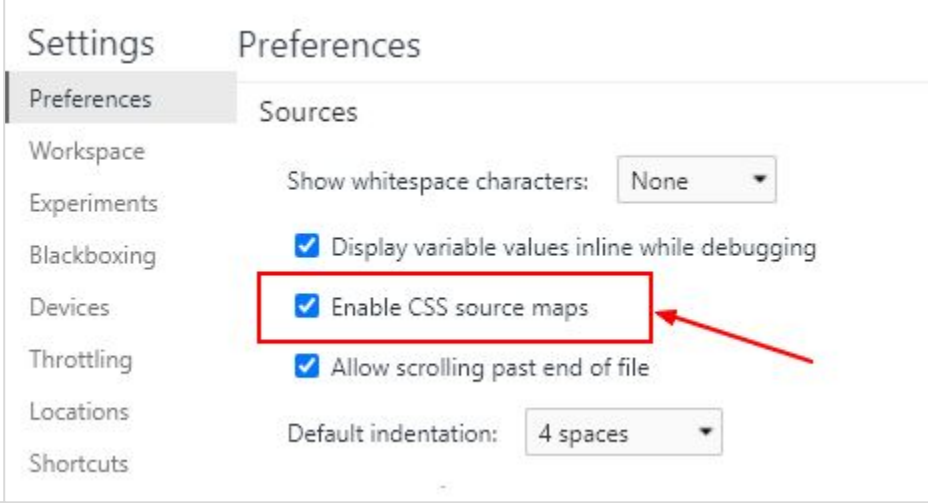
- No duplicate imports
- Better performance
- Scoped namespaces (e.g. buttons.$primary-color)

```
style.scss

@import 'buttons'; // old
@use 'buttons'; // new
```

Order is important

```
// GLOBAL
@import "global/variables.scss";
@import "global/mixins.scss";
@import "global/global.scss";

// BLOCKS
@import "components/header.scss";
@import "components/nav.scss";
@import "components/slider.scss";
```

Source map




```
▼<div class="width-wrapper">
  ▶<div class="title-header pens-title-header">…</div>
  ▼<div id="picks-grid" class="pen-grid view-grid">
    ▼<div class="single-pen

      " data-slug-hash="aygmoG">
      ▶<div class="iframe-wrap loaded">…</div>
      ▶<div class="meta">…</div> == $0
      </div>
    ▶<div class="single-pen

      " data-slug-hash="bryQGJ">…</div>
```

```
}

.meta {                                          page.css:528
  background: ▶ #32333b;
  position: relative;
  font-size: 0.85em;
  white-space: nowrap;
  padding: ▶ 1rem 1rem 0.5rem;
  z-index: 3;
}

* {                                              global.css:3
  margin: ▶ 0;
  ~~padding: ▶ 0;~~
  ~~-webkit-box-sizing: border-box;~~
  box-sizing: border-box;
}
```

What else is available in a preprocessor like SCSS?

- Functions
- Loops
- @if and @else statements – Control directives
- @for
- @while
- Inheritance (@extend)
- SASS Interpolation
- SASS Placeholders (%placeholder)

Custom properties - CSS variables

Custom properties

a.k.a. CSS variables

are values you define in CSS using `--name` syntax and reuse with `var()`

💡 **Key features:**

- Defined in the browser, not at compile time
- Can be updated dynamically with JavaScript
- Can be overridden in different parts of the page
- Inherit through the DOM tree

```
:root {
  --primary-color: #ff6600;
  --font-size-base: 16px;
}

.button {
  background-color:
var(--primary-color);
  font-size:
var(--font-size-base);
}
```

SCSS vs CSS Custom Properties

Feature	SCSS Variables ($)	CSS Custom Properties (--var)
Defined at	Compile time	Runtime (in browser)
Can be changed with JS?	❌ No	✅ Yes
Inherit values?	❌ No	✅ Yes

✅ Use SCSS variables for development & structure.

✅ Use CSS custom properties for theming and dynamic styling.

✅ **SCSS variables**:

- For consistent values during development
- Colors, spacing, breakpoints

✅ **CSS Custom Properties**:

- When value should change in browser
- Theming (light/dark)
- Dynamic styles with JS

CSS nesting

CSS Nesting vs SCSS Nesting

🔥 CSS now supports nesting, but it's less powerful than SCSS

✅ It works — but it's not as flexible as SCSS.

💡 For now, it's still better to use SCSS!

```
.card {
  & .title {
    color: red;
  }
}
```

CSS Nesting – Limitations

1️⃣ Only 1 level deep without &

2️⃣ No nested @media or @keyframes like SCSS

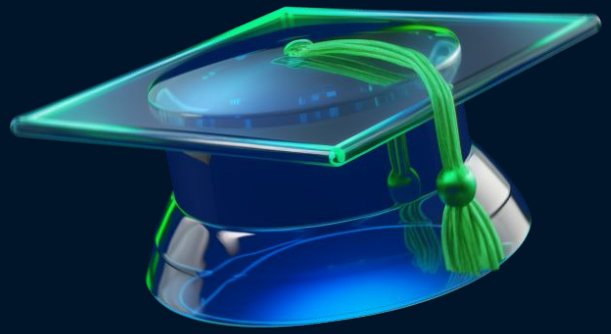
3️⃣ Lacks logic (no @if, @each, etc.)

```
.card {
  & .title {
    color: red;
  }
}
```

✅ Good for small, flat components
❌ Not powerful enough for large projects → SCSS is still better

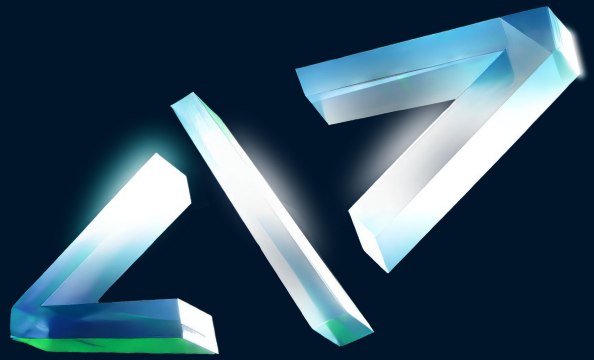
Summary

1. What is SCSS
2. Why use SCSS
3. What SCSS gives you:
    - variables
    - Nesting
    - mixin
    - use
4. New CSS features:
    - CSS Custom Properties
    - CSS Nesting



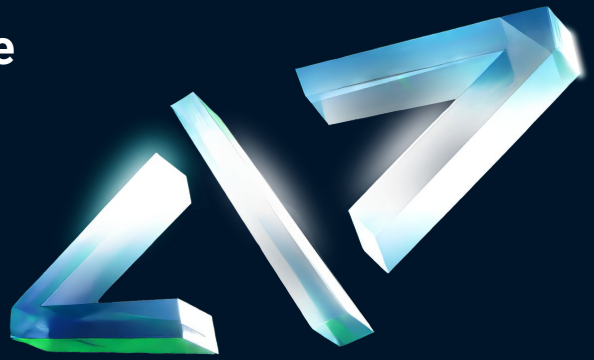
Homework

1. Split your styles into multiple SCSS files
   - One BEM block = one SCSS file
   - Use separate files for:
     - variables
     - mixins
     - base styles
     - components
2. Move repeated values into SCSS variables
3. Use SCSS nesting



Additional Homework

1. Use mixins
   - For example, a button mixin or responsive typography
2. Use CSS custom properties
   - For things like color themes or dynamic text color



Quality Criteria for HTML Course

❤️ Mandatory for passing the course

💛 Required for the highest grade

💚 Optional

❤️ SCSS preprocessor is used

❤️ Styles are split into components

❤️ SCSS variables are used for colors and fonts

💛 Mixins are used where appropriate

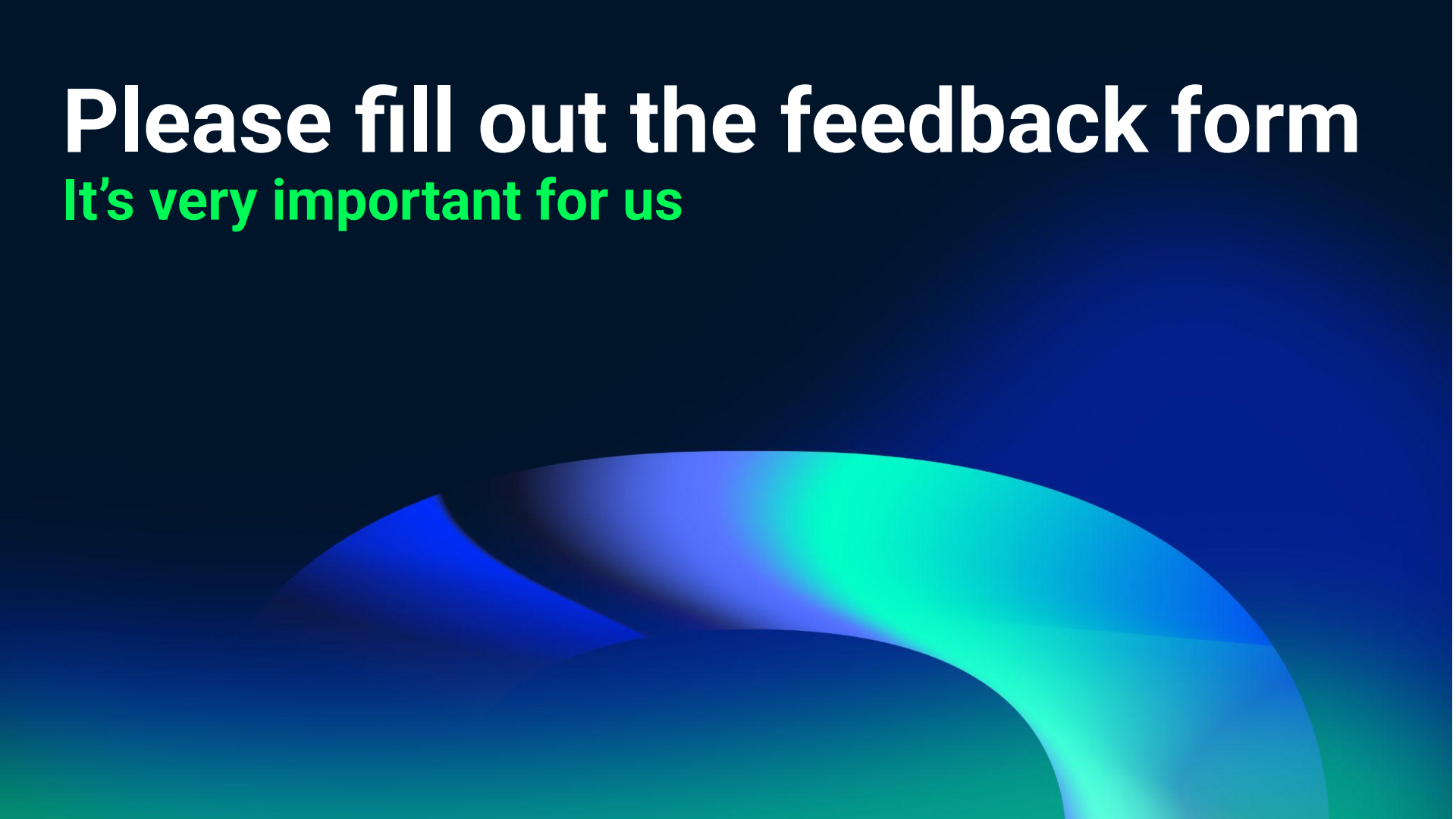
💛 No magic numbers (all values are explained or reusable)

BRO Academy

QUESTIONS?

Please fill out the feedback form

It’s very important for us



THANK YOU!

Have a good evening!