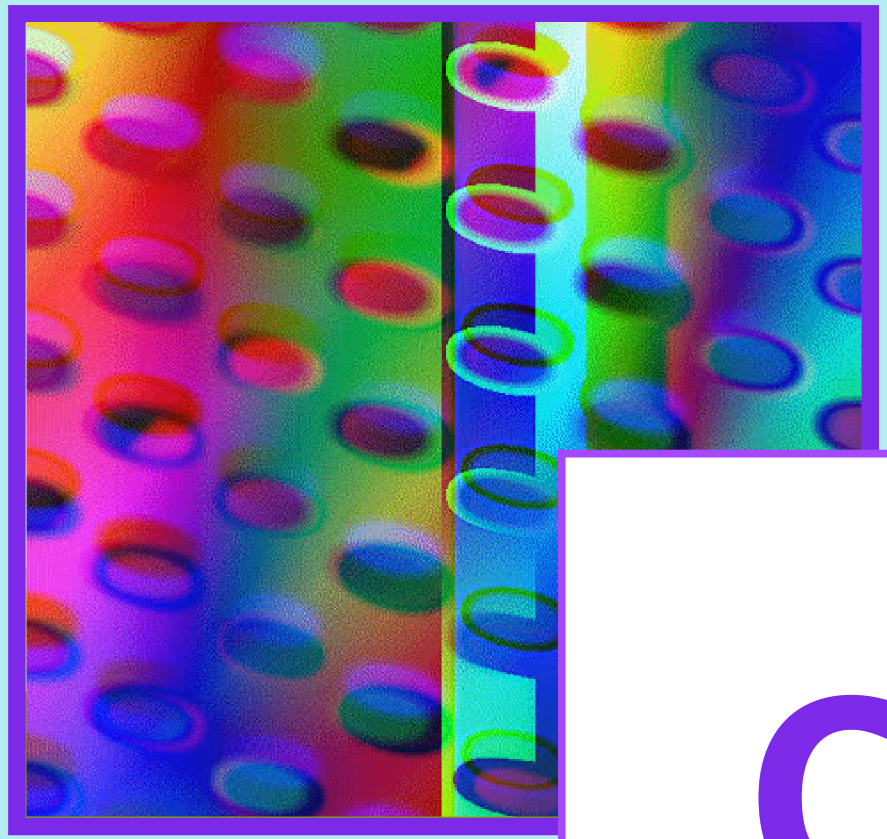
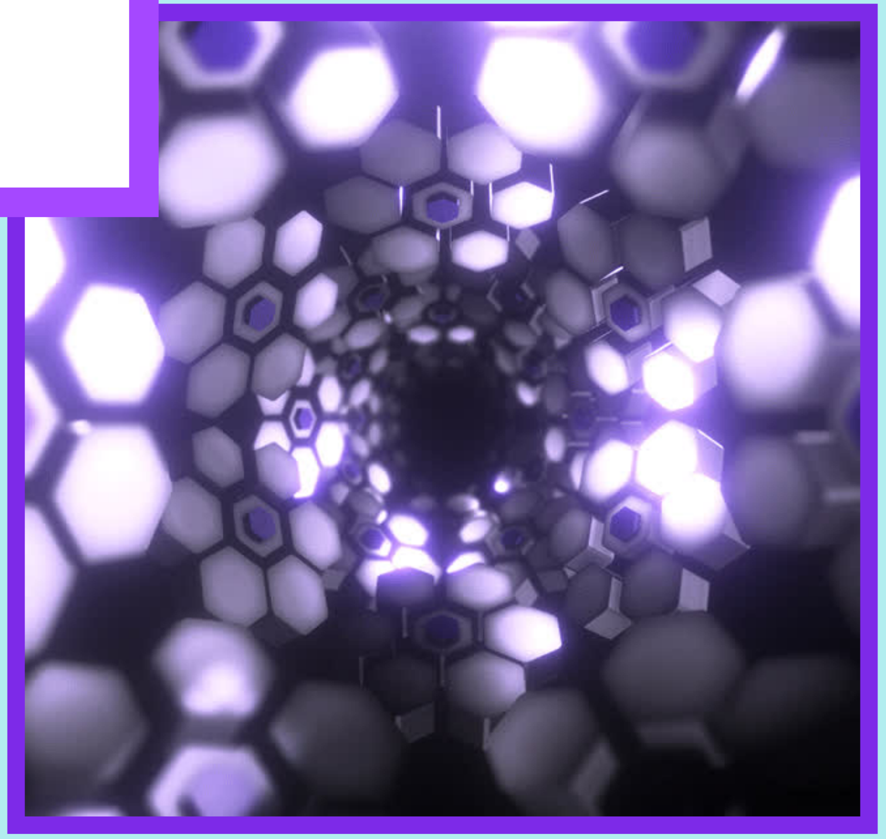
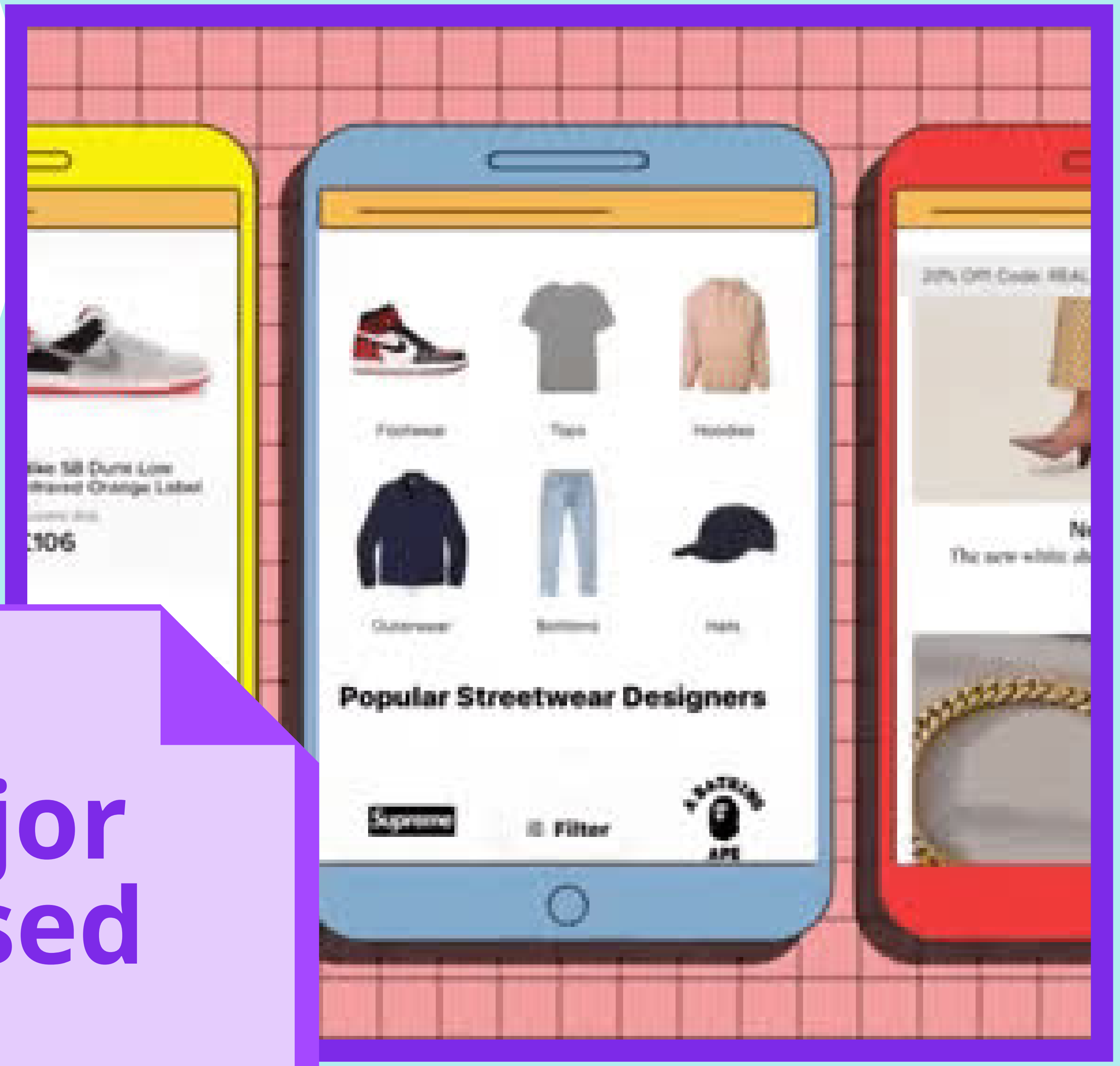


CSS in JS

Presented by
Tom Isles

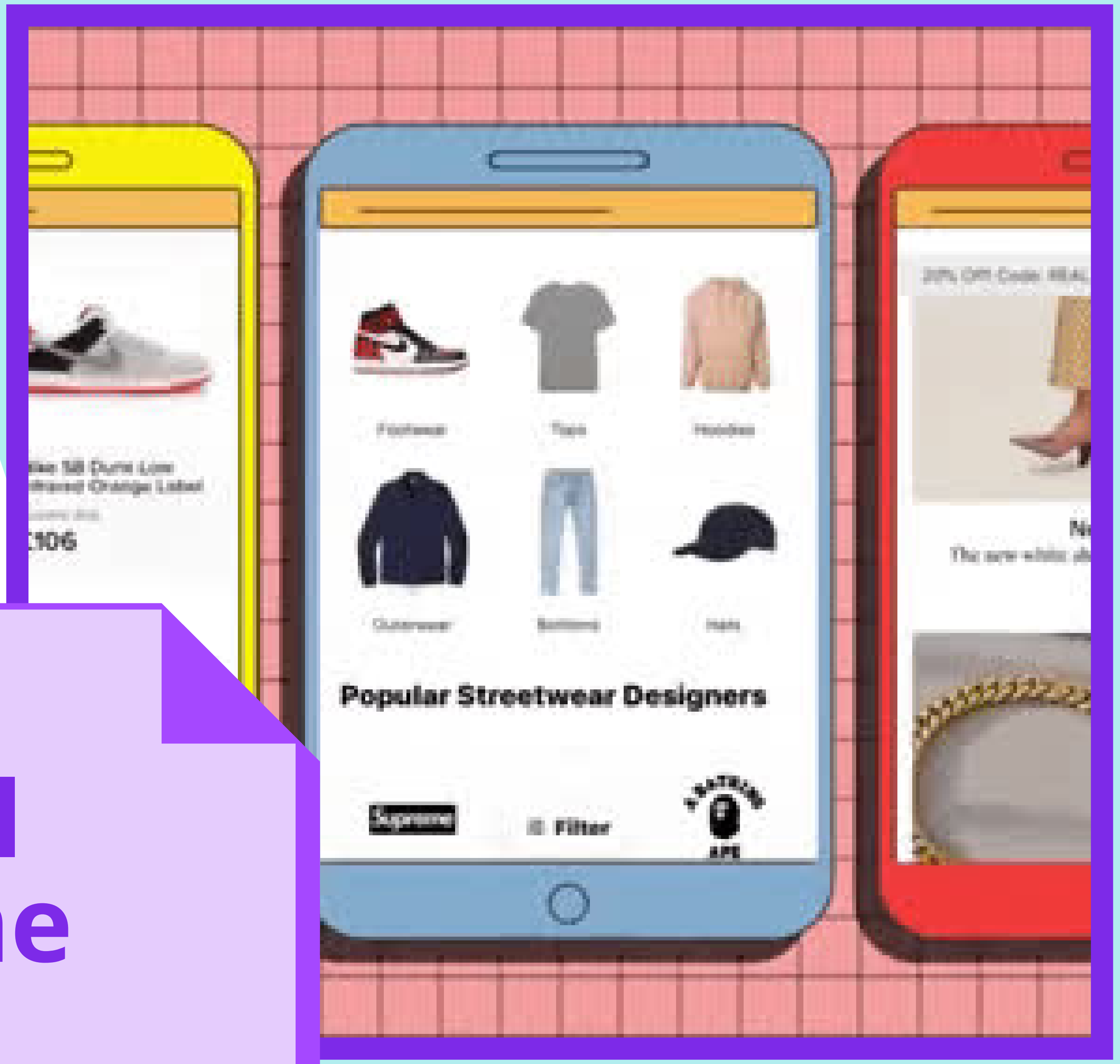


Q

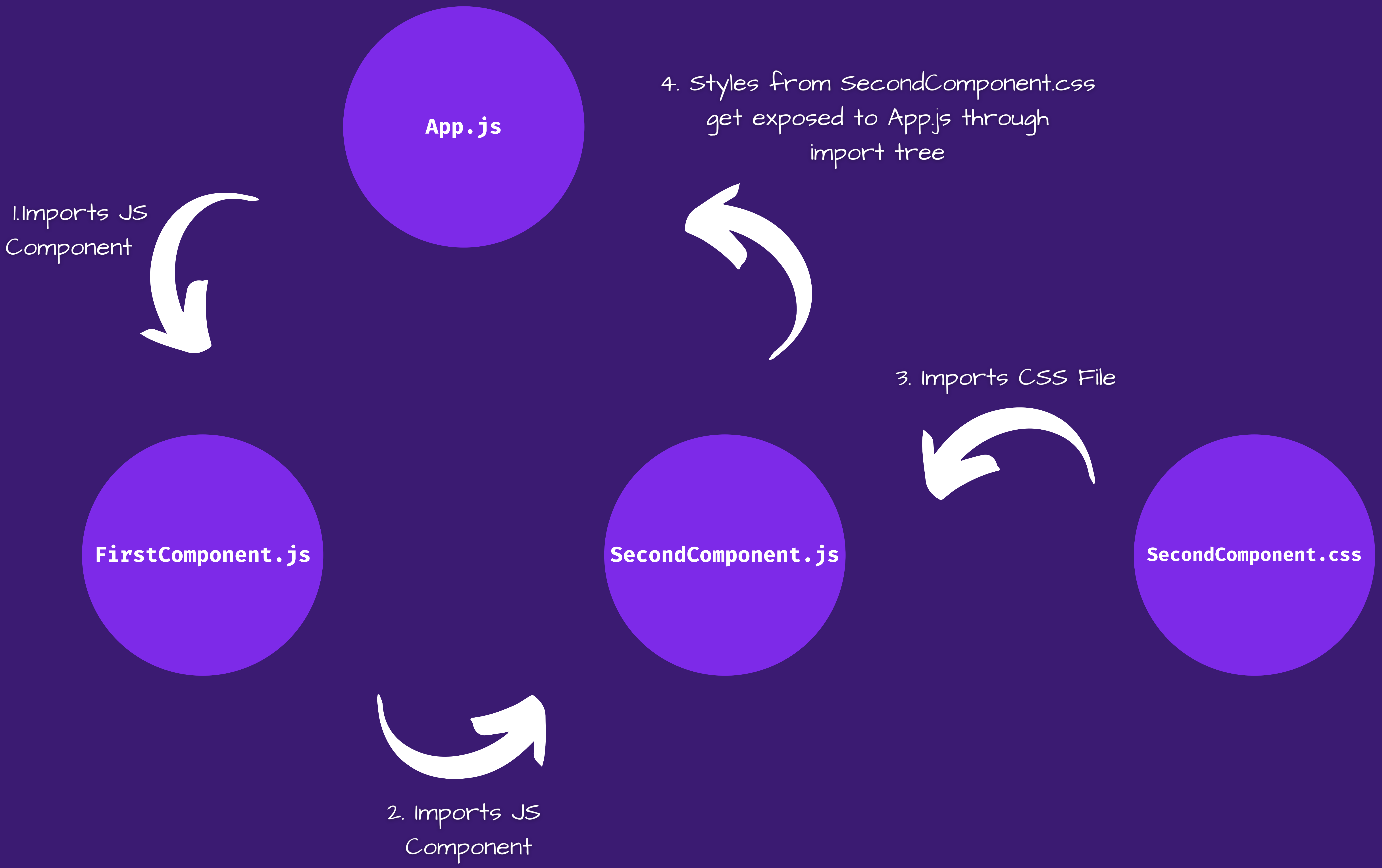


What is CSS's major problem when used with React?

A



CSS Selectors all exist in the same global scope.


App.js
4. Styles from SecondComponent.css get exposed to App.js through import tree
1. Imports JS Component
3. Imports CSS File
FirstComponent.js
SecondComponent.js
SecondComponent.css
2. Imports JS Component

Encapsulation

Encapsulation is an important property of maintainability in a codebase. Properly encapsulated components will not have any implicit dependencies on each other.

In other words, the component won't have access to things it shouldn't know about. If we do this right, our code is **decoupled**.

CSS imports are global, which breaks decoupling, because a component will have access to styles that are not within its own scope.

This means...

We can never truly have **confidence** that our components do what they say they do.

Every time I add a class name to a div, I have to worry that in some my class name has been defined in someone else's CSS file.

Imagine being in a group project, and two team members both write a component. For some reason whenever you add a class name to your component, it turns red, starts flashing and moves to the left side of the screen.

How did we get here?

CSS was invented for HTML files and single file web pages. It doesn't really translate to our component-based way of working.

BEM

CSS Modules

CSS in JS

BEM

Stands for **Block, Element, Modifier.** Predates React by a number of years.

Is a **Naming Convention.** No special technology here, just a way to name your components so that they don't tread on each other's toes.

The history of BEM began in 2005. Back then, from the frontender perspective, a typical project was a set of static HTML pages.

BEM

A **Block** is a standalone name, meaningful on its own, like *button*.

"button" is a Block

```
<button class="button">
   Normal button
</button>

<button class="button button--state-success">
   Success button
</button>

<button class="button button--state-danger">
   Danger button
</button>
```

BEM

A **Block** is a standalone name, meaningful on its own, like *button*.

An **Element** is part of a block that means nothing on its own, but denotes some characteristic of a block.

"button" is a Block

"state" is an Element

```
<button class="button">
   Normal button
</button>

<button class="button button--state-success">
   Success button
</button>

<button class="button button--state-danger">
   Danger button
</button>
```

BEM

A **Block** is a standalone name, meaningful on its own, like *button*.

An **Element** is part of a block, that means nothing on its own but denotes some characteristic of a block.

A **Modifier** is a flag to change the appearance or behaviour of a block or element.

"button" is a Block

"state" is an Element

"danger" is a modifier

```html
<button class="button">
   Normal button
</button>

<button class="button button--state-success">
   Success button
</button>

<button class="button button--state-danger">
   Danger button
</button>
```

BEM

A **Block** is a standalone name, meaningful on its own, like *button*.

An **Element** is part of a block that means nothing on its own, but denotes some characteristic of a block.

A **Modifier** is a flag to change the appearance or behaviour of a block or element.

BEM was once popular, particularly on static web pages, but it hasn't translated to React particularly well. It's **very verbose and annoying to type**. There are better solutions for our use case.

"button" is a Block

"state" is an Element

"danger" is a modifier.

```
<button class="button">
   Normal button
</button>

<button class="button button--state-success">
   Success button
</button>

<button class="button button--state-danger">
   Danger button
</button>
```

CSS Modules

</>

CSS Modules is a plugin that changes the way CSS scopes its declarations. It's commonly used with module bundlers like **Webpack** to compile your CSS files when you're building your application.

CSS Modules changes the scope of your class and animation names to local scope. It does so by taking in your CSS files, changing the class names, and spitting them out, which allows it to generate and manage global class names for you.

It basically compiles your CSS into more CSS.

CSS Modules

When importing a **CSS Module** into a react component, we don't just import the file. Instead, we can import a **styles** object from the file, or a **getStyle** function.

The CSS Module exports an object, which contains a set of all the mappings between your local class names and the global class names they get compiled to in the background.

```
import styles, { getStyle } from './App.css';

function App() {
   return (
      <div className={styles.app}>
         Hello, World!
      </div>
   )
}

export default App;
```

CSS Modules

The styles object might look something like the object on the right.

The keys of the object are the local names for your styles, and the values of the object are the global names that map to them.

```
const styles = {
   'app': 'bx748',
   'header': 'acjdn',
   getStyle: (styleString) =>
   styles[styleString],
}
```

CSS Modules

CSS Modules is not completely the same as CSS. It has a number of extra features.

A powerful feature of CSS module is its **composes** feature, which allows you to compose together styles. For example, a green success button could **compose** a basic button's style, and add its own style on top.

CSS Modules are an adequate solution to making sure our styles don't infect other components. We get isolation, and a few extra features on top, and we still get all the benefits of using plain CSS files.

```
.basicButton {
   cursor: pointer;
   width: 100%;
   height: 100%;
   borderRadius: 4px;
   padding: 4px;
}

.successButton {
   composes: basicButton;
   background-color: green;
   color: white;
}
```

CSS Modules

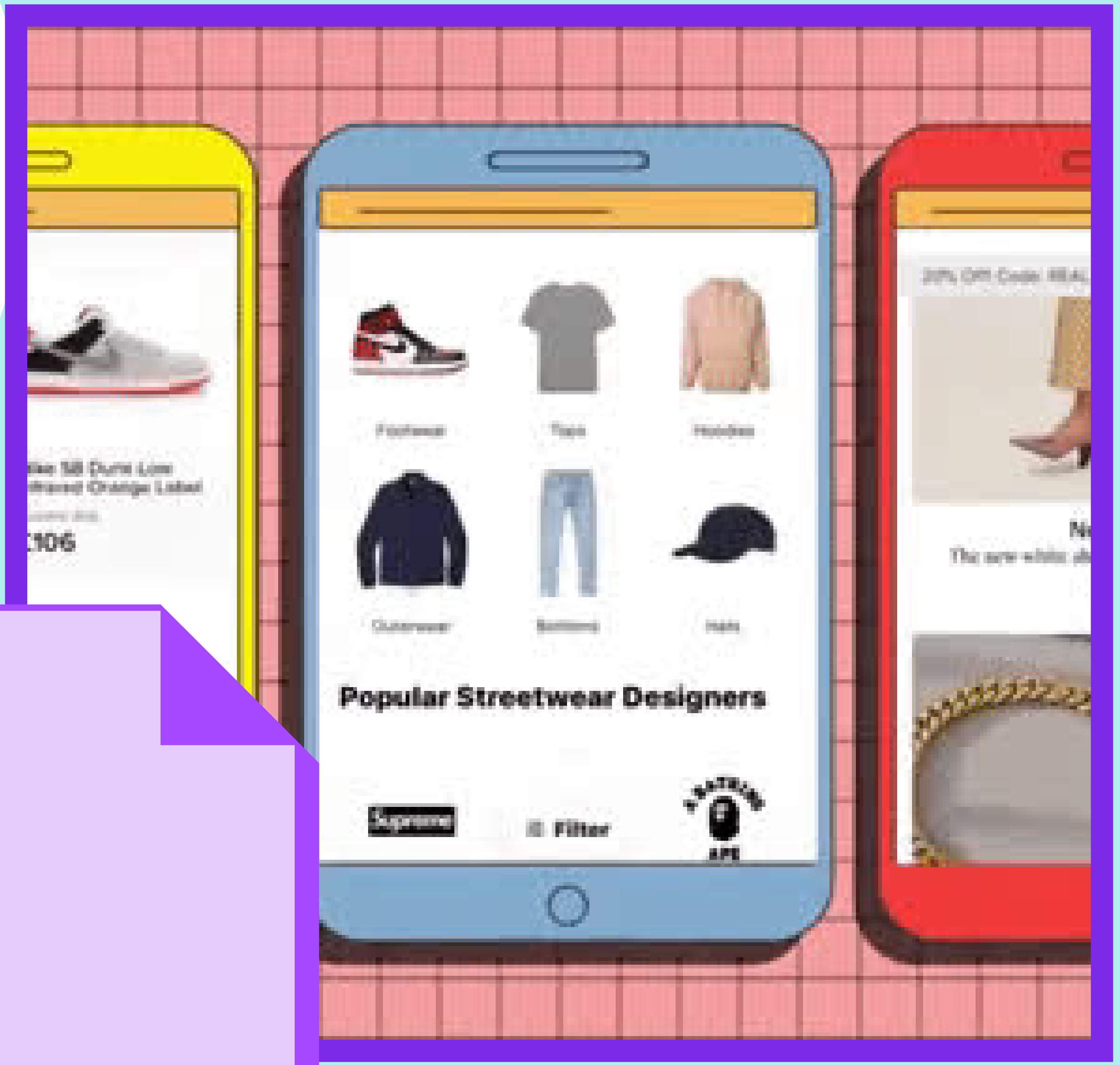
As of **Create-React-App V2, CSS modules comes bundled with React applications**.

To use them, your css files should be suffixed with **.module.css**.

Demo

Q

Why use CSS at all?



CSS in JS

</>

A note before we start: CSS in JS is a relatively new idea in web development and so you may see differing opinions about it online!

Inline Styles

2014: A facebook engineer, Christopher Chedeau, details all the insanity involved in getting CSS to work at scale in Facebook and asks one simple question.

Why not just do away with CSS files and use inline styles?

Inline Style

```
const style = {
   color: red;
}

<div style={style} />
```

Inline Styles

Inline Styles:

- Are local to components. No more weird class name errors!
- Treat them the way you would any javascript object.
- Better experience - no need to switch files.
- Use javascript to change values easily.

Inline Styles

</>

But...

- Hard to do responsive design. No Media Queries!
- Hard to handle behaviours like hover, focus, and so on.
- Hard to reuse like you can with class names.

CSS in JS

It would be great if we could get all the benefits of using JS:

- Isolation.
- Better developer experience.
- Manipulating values using JS.

With all the benefits of CSS:

- Reusability.
- Responsive design using media queries.
- Handling behaviours like hover, focus.

We can. It's called **CSS in JS.**

CSS in JS

CSS in JS allows us to use standard CSS syntax inside javascript files. It supports:

- CSS selectors
- Dynamic behaviours (:hover, :focus, etc)
- Media queries
- Composition, extensible classes
- Animations (keyframes, transitions)

But it's **Javascript**.

Libraries

CSS in JS is implemented using external libraries.
Popular examples include:

- Styled Components
- Emotion*
- Linaria

**Developed here in Sydney by ThinkMill!*

Styled Components

Today we'll be creating a project that uses the **styled-components** library. Let's start by creating a new react project.

```
npx create-react-app styled-components-example
```

The first thing we need to do is install styled components into our project.

```
cd styled-components-example && yarn add styled-components
```

StyledComponent.js

We'll create a component that uses styled components to make a custom button component.

We'll only go through the basics here, but feel free to continue experimenting in your own time.

Before we start: Tagged Template Literals

Styled components uses an ES6 language concept called *tagged template literals.*

Tagged template literals allow you to prefix a template string with a function name, which will call the function and pass in the template string as the first index of an array parameter.

In other words, its syntactic sugar for calling a function.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
const capitalize = (str) => {
    return str[0].toUpperCase()
}

const capitalizedHello = capitalize`Hello, World!`

// HELLO, WORLD!
```

StyledComponent.js

We'll start by creating a basic button style.

We define a *styled.button* using the styled-components library.

We use a tagged template literal to input css directly.

Styled components are actually React components, so remember to name them correctly.

```
import styled from 'styled-components';


const StyledButton = styled.button`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   border-radius: 4px;
   padding: 16px;
`;

return (
   <StyledButton />
)
```

Dynamic Behaviours

Styled-components extends the standard CSS syntax, allowing us to define dynamic behaviours such as focus and hover.

A special symbol "&" lets a styled element reference itself.

We can use **&** to specify what happens on hover, or even to specify appearance behaviours to an element's children.

```
import styled from 'styled-components';


const StyledButton = styled.button`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   border-radius: 4px;
   padding: 16px;

   &:hover {
      transform: scale(1.05);
   }

   & > p {
      // Any <p> tag under a styled button will be
      // affected
   }
`;
```

Media Queries

We can also define media queries inside a styled component. By default, the media query will only affect the styled component that defines it.

```
import styled from 'styled-components';


const StyledButton = styled.button`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   border-radius: 4px;
   padding: 16px;

   @media (max-width: 768px) {
      /* responsive styles go here */
   }
`;
```

Props

We can pass props in to our styled components.

This allows us to have easily customisable styled components.

Each style accepts a callback function which will have props passed to it.

Remember, styled components are just react components, so we pass in props in the same way.

```
import styled from 'styled-components';


const ColoredButton = styled.button`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   border-radius: 4px;
   padding: 16px;
   background-color: ${props => props.backgroundColor}
   color: ${props => props.color}
`

return (
   <ColoredButton backgroundColor="black" color="white" />
)
```

Extending

We can also use the *styled* function to extend styles that we've already defined.

This allows us to compose styles together, or extend off base styles to create new variants.

```
import styled from 'styled-components';


const StyledButton = styled.button`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   border-radius: 4px;
   padding: 16px;
`;

const ColoredButton = styled(StyledButton)`
   background-color: ${props => props.backgroundColor}
   color: ${props => props.color}
`
```

We can pass in a component to our styled function for inheritance

Extending with Mixins

The styled function automatically creates a React component.

However, we don't need to create a react component to use styled components.

styled-components exports a css function which allows us to define arbitrary CSS using tagged template literals.

The CSS keyword is used to provide mixins. A mixin is an object
that can be used by other objects without any inheritance relationship, ie it can be "mixed in" easily.

```
import { css } from 'styled-components';


const removeButtonStyles = css`
   font-size: 100%;
   font-family: inherit;
   border: 0;
   padding: 0;
`;

const StyledButton = styled.button`
   ${removeButtonStyles};
   padding: 16px;
   border-radius: 4px;
   background-color: ${props => props.backgroundColor}
   color: ${props => props.color}
`
```

We interpolate the CSS mixin into our styled component

Animations

Styled-components provides a *keyframes* function which lets us define CSS animations in JS.

You can then interpolate your animation into a styled component.

```
import { keyframes } from 'styled-components';


const rotate = keyframes`
   from {
      transform: rotate(0deg)
   }

   to {
      transform: rotate(360deg)
   }
`;

const Rotate = styled.div`
   animation: ${rotate} 2s linear infinite;
`
```

Problems

It's not all good news:

- Many libraries have performance tradeoffs and are not as fast as vanilla CSS.
- Can increase the amount of time to first render a page.
- The browser can usually download CSS and JS independently of each other. Not so with many CSS in JS libraries - they're bundled together*

**Linaria is an exception to this rule*

Demo

Summary

- What are the problems with importing raw CSS files?
- We somewhat solved this in olden days using naming schemes like BEM, but these don't hold up in React.
- CSS modules
- CSS in JS
- Styled Components

Good work!