

# day42-style

☰ 태그	
📅 날짜	@2022년 12월 2일

리액트 네이티브에서의 스타일링은 웹 프로그래밍에서 사용하는 css와 차이가 있다.

프로젝트를 만들자.

expo init react-native-style

```
Microsoft Windows [Version 10.0.19044.2251]
(c) Microsoft Corporation. All rights reserved.

C:\Users\wtjoeun>cd ..
C:\Users>cd ..
C:\>cd reactnative
C:\reactnative>expo init react-native-style
WARNING: expo-cli has not yet been tested against Node.js v18.12.0.
If you encounter any issues, please report them to https://github.com/expo/expo-cli/issues

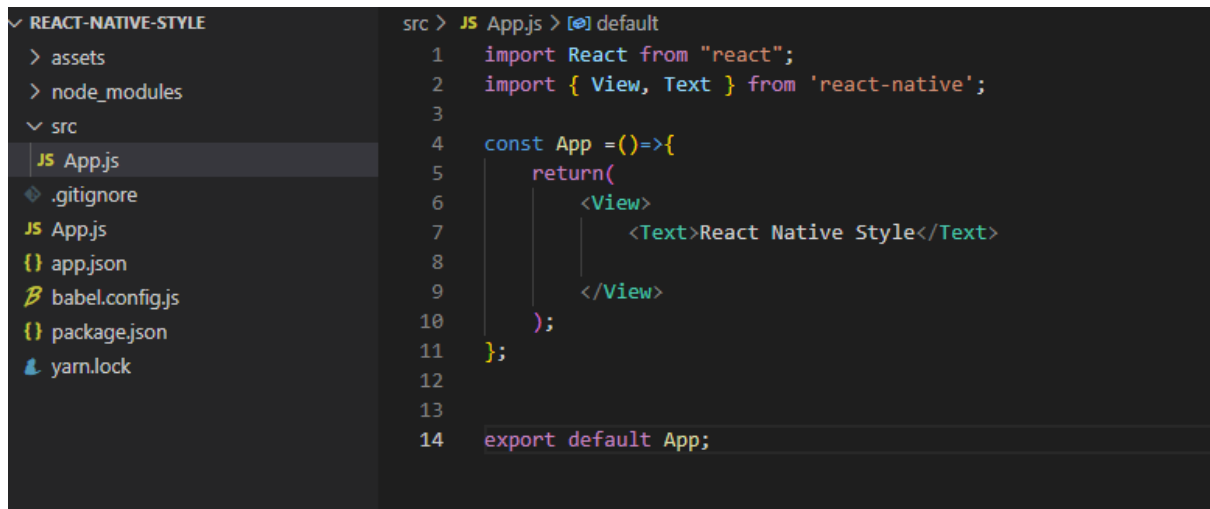
expo-cli supports following Node.js versions:
* >=12.13.0 <15.0.0 (Maintenance LTS)
* >=16.0.0 <17.0.0 (Active LTS)

Migrate to using:
> npx create-expo-app --template

? Choose a template: » - Use arrow-keys. Return to submit.
  ----- Managed workflow -----
> blank a minimal app as clean as an empty canvas
  blank (TypeScript) same as blank but with TypeScript configuration
  tabs (TypeScript)  several example screens and tabs using react-navigation and TypeScript
  ----- Bare workflow -----
  minimal            bare and minimal, just the essentials to get you started
```

blank로 설치

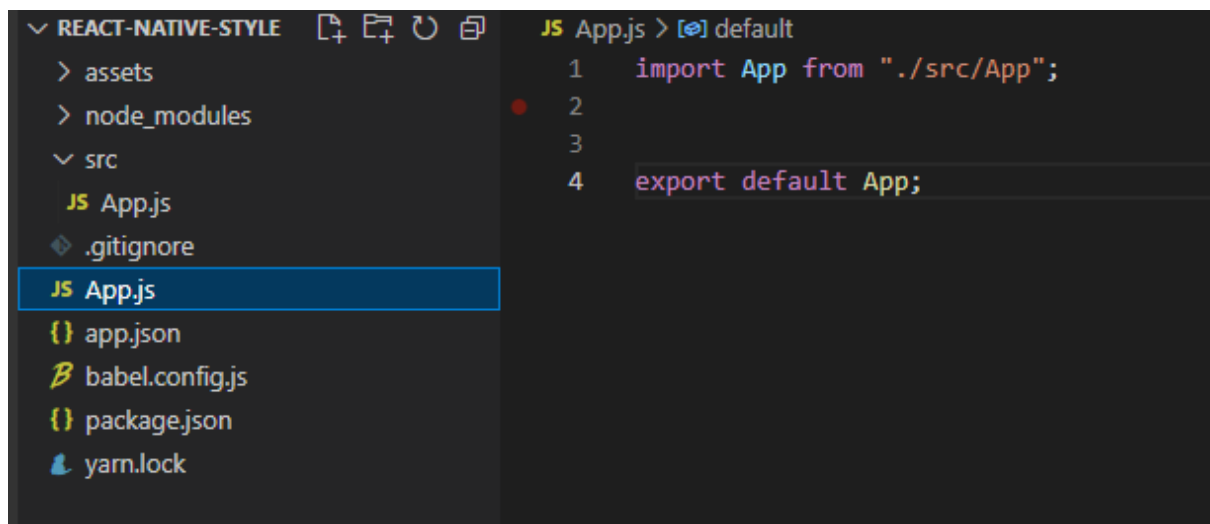
생성이 완료 되면 src폴더를 만들어서 그 밑에 app.js컴포넌트를 아래 처럼 작성하자



The screenshot shows a VS Code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project named 'REACT-NATIVE-STYLE' with a 'src' directory containing 'App.js'. The code editor shows the content of 'App.js' with the following code:

```
src > JS App.js > [🔍] default
1  import React from "react";
2  import { View, Text } from 'react-native';
3
4  const App = ()=>{
5    return(
6      <View>
7        <Text>React Native Style</Text>
8      </View>
9    );
10 };
11
12
13
14 export default App;
```

루트 디렉터리에는 아래처럼 코드를 작성한 app.js를 만들어주자.



The screenshot shows the same VS Code editor, but the code in 'App.js' has been updated. The file explorer on the left now shows 'App.js' as the selected file. The code editor shows the following code:

```
JS App.js > [🔍] default
1  import App from "../src/App";
2
3
4  export default App;
```

## 스타일링

리액트 네이티브에서는 자바스크립트를 이용해 스타일링을 할 수 있다.

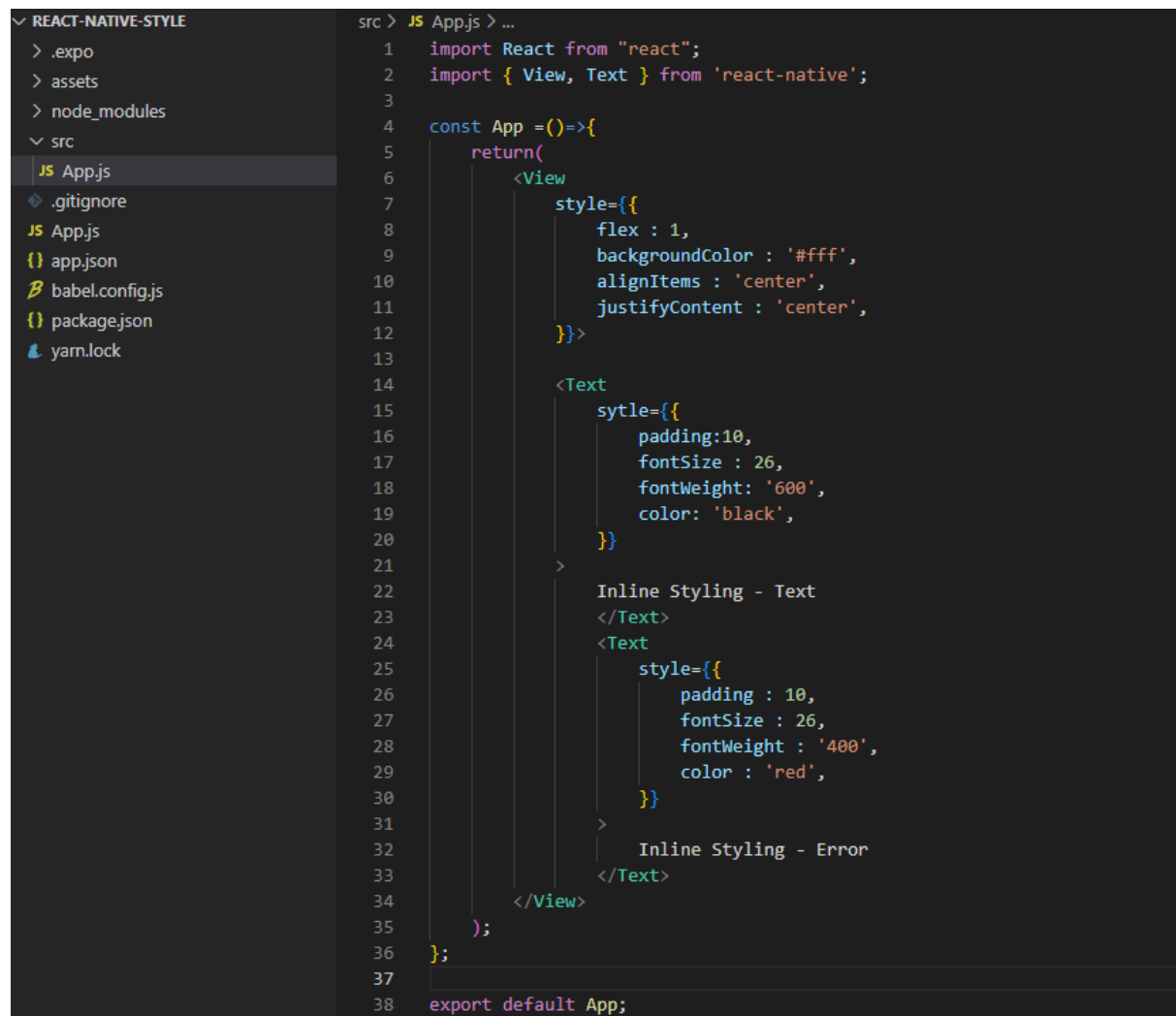
컴포넌트에는 style이라는 속성이 있고, 이 속성에 인라인 스타일을 적용하는 방법과 스타일 시트에 정의된 스타일을 사용하는 방법이 있다.

## 인라인 스타일링

인라인 스타일은 HTML의 인라인 스타일링처럼 컴포넌트에 직접 스타일을 입력하는 방식이다.

다만 HTML에서는 문자열 형태로 스타일을 입력하지만, 리액트 네이티브에서는 객체 형태로 전달해야한다는 차이점이 있다.

APP 컴포넌트에 인라인 스타일을 적용해서 아래처럼 수정해보자



```
src > JS App.js > ...
1  import React from "react";
2  import { View, Text } from 'react-native';
3
4  const App = ()=>{
5    return(
6      <View
7        style={{
8          flex : 1,
9          backgroundColor : '#fff',
10         alignItems : 'center',
11         justifyContent : 'center',
12       }}>
13
14        <Text
15          style={{
16            padding:10,
17            fontSize : 26,
18            fontWeight: '600',
19            color: 'black',
20          }}
21        >
22          Inline Styling - Text
23        </Text>
24        <Text
25          style={{
26            padding : 10,
27            fontSize : 26,
28            fontWeight : '400',
29            color : 'red',
30          }}
31        >
32          Inline Styling - Error
33        </Text>
34      </View>
35    );
36  };
37
38  export default App;
```

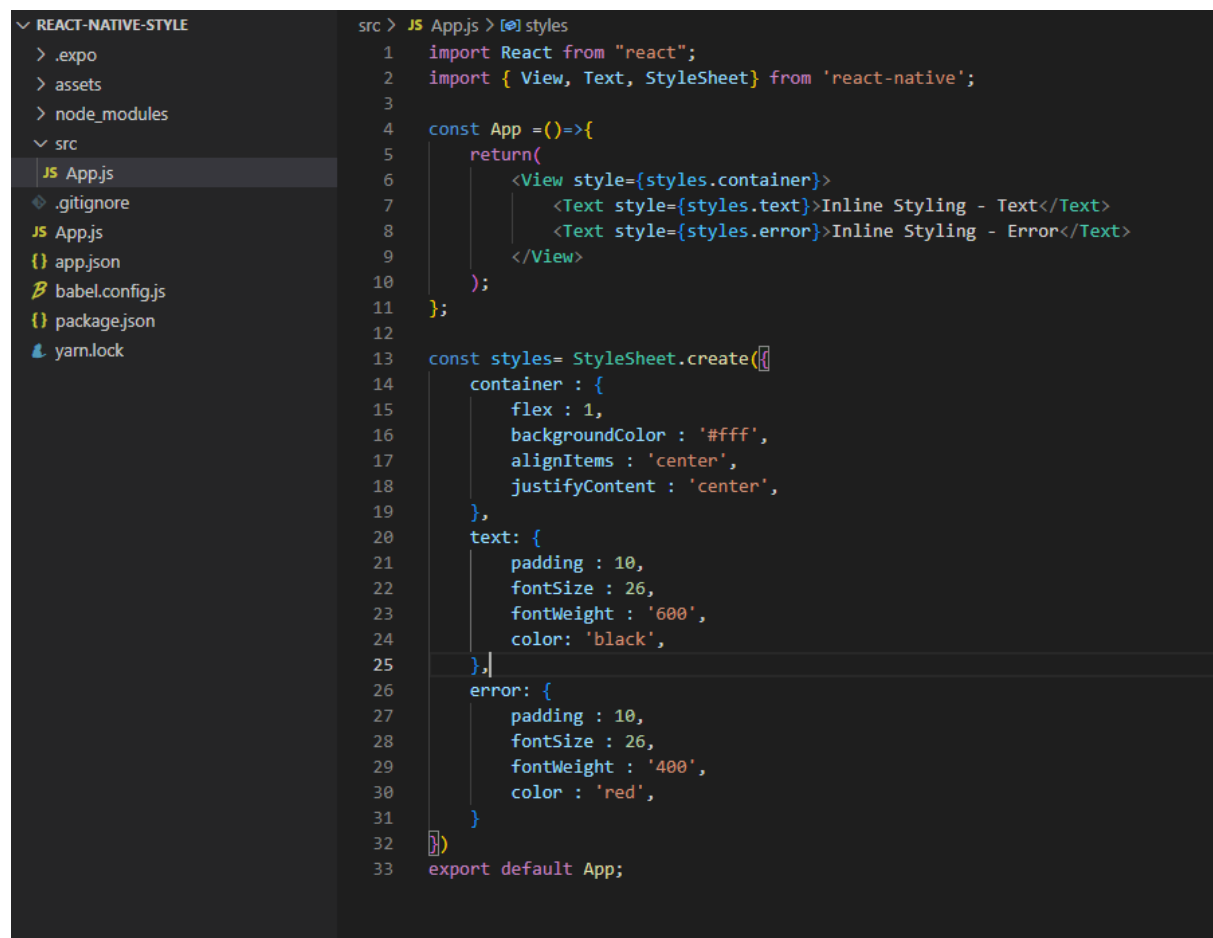


인라인 스타일링은 어떤 스타일이 적용되는지 잘 보인다는 장점이 있다. 하지만 두 Text 컴포넌트처럼 비슷한 역할을 하는 컴포넌트의 동일한 코드가 반복된다는 점과, 어떤 이유로 해당 스타일이 되었는지 코드만으로는 명확하게 이해하기 어렵다는 단점이 있다.

## 클래스 스타일링

클래스 스타일링은 컴포넌트의 태그에 직접 입력하는 방식이 아니라 스타일 시트에 정의된 스타일을 사용하는 방법이다. 스타일시트에 스타일을 정의하고 컴포넌트에서는 정의된 스타일의 이름으로 적용하는 클래스 스타일링 방법은 웹 프로그래밍에서 css클래스를 이용하는 방법과 유사하다. 프로젝트를 생성하면 함께 생성되는 app.js파일에서 클래스 스타일링이 적용된 모습을 확인할 수 있다.

## 클래스 스타일 방식으로 변경해보자



```
src > JS App.js > [0] styles
1  import React from "react";
2  import { View, Text, StyleSheet } from 'react-native';
3
4  const App = () => {
5    return (
6      <View style={styles.container}>
7        <Text style={styles.text}>Inline Styling - Text</Text>
8        <Text style={styles.error}>Inline Styling - Error</Text>
9      </View>
10    );
11  };
12
13  const styles = StyleSheet.create({
14    container: {
15      flex: 1,
16      backgroundColor: '#fff',
17      alignItems: 'center',
18      justifyContent: 'center',
19    },
20    text: {
21      padding: 10,
22      fontSize: 26,
23      fontWeight: 'bold',
24      color: 'black',
25    },
26    error: {
27      padding: 10,
28      fontSize: 26,
29      fontWeight: 'bold',
30      color: 'red',
31    },
32  });
33  export default App;
```



인라인 스타일을 적용했을 때보다 조금 더 깔끔해진 것 같다.

또한 전체적인 스타일을 관리하는 데도 클래스 스타일링이 인라인 스타일링보다 더 쉽다. 만약 오류가 있는 상황에서 글자 색을 빨간색이 아니라 주황색으로 변경해야 한다면, 클래스 스타일링 방식에서는 `error` 객체에서 `color: 'orange'`로만 변경하면 되지만, 인라인 스타일링의 경우 모든 파일을 찾아다니며 변경해야 하는 단점이 있다.

간략하게 화면을 확인하는 상황에서는 인라인 스타일을 사용하는 것이 편할 수 있지만, 장기적으로 생각하면 클래스 스타일을 사용하는 것이 관리 측면에서 유리하다.

## 여러 개의 스타일 적용

작성한 app 컴포넌트의 스타일 코드에서 `text` 스타일과 `error` 스타일은 중복된 스타일이 많으며, 두 스타일 모두 `text` 컴포넌트에 적용되는 스타일이라는 공통점도 있다. 이렇게 중복된 코드를 제거하다 보면 스타일을 덮어쓰거나 하나의 컴포넌트에 여러 개의 스타일을 적용해야 할 때가 있다. 이렇게 여러 개의 스타일을 적용해야 할 경우는 배열을 이용하여 `style` 속성에 여러개의 스타일을 적용하면 된다.

```
REACT-NATIVE-STYLE
> .expo
> assets
> node_modules
src
  JS App.js
  .gitignore
  App.js
  app.json
  babel.config.js
  package.json
  yarn.lock

src > JS App.js > [0] styles
1 import React from "react";
2 import { View, Text, StyleSheet } from 'react-native';
3
4 const App = ()=>{
5   return(
6     <View style={styles.container}>
7       <Text style={styles.text}>Inline Styling - Text</Text>
8       <Text style={[styles.text, styles.error]}>Inline Styling - Error</Text>
9     </View>
10   );
11 };
12
13 const styles= StyleSheet.create({
14   container : {
15     flex : 1,
16     backgroundColor : '#fff',
17     alignItems : 'center',
18     justifyContent : 'center',
19   },
20   text: {
21     padding : 10,
22     fontSize : 26,
23     fontWeight : '600',
24     color: 'black',
25   },
26   error: {
27     fontWeight : '400',
28     color : 'red',
29   }
30 },
31 ),
32
33
34
35 export default App;
```

중복된 코드가 사라지면서 코드가 깔끔해지고 공통된 부분의 관리가 편해진다.

여러개의 스타일을 적용할 때 주의할 점은 적용하는 스타일의 순서이다. 뒤에 오는 스타일이 앞에 있는 스타일을 덮는다는 것을 기억해야한다. 예를들어 앞의 코드에서 적용된 스타일의 순서를 변경해 [styles.error, styles.text]로 작성하면 글자 색이 빨간색 대신 검은색으로 된다.

여러 개의 스타일을 적용할 때 반드시 클래스 스타일만 적용해야 하는 것은 아니다.

아래처럼 인라인과 클래스를 혼용해서 사용하는 방법도 있다.

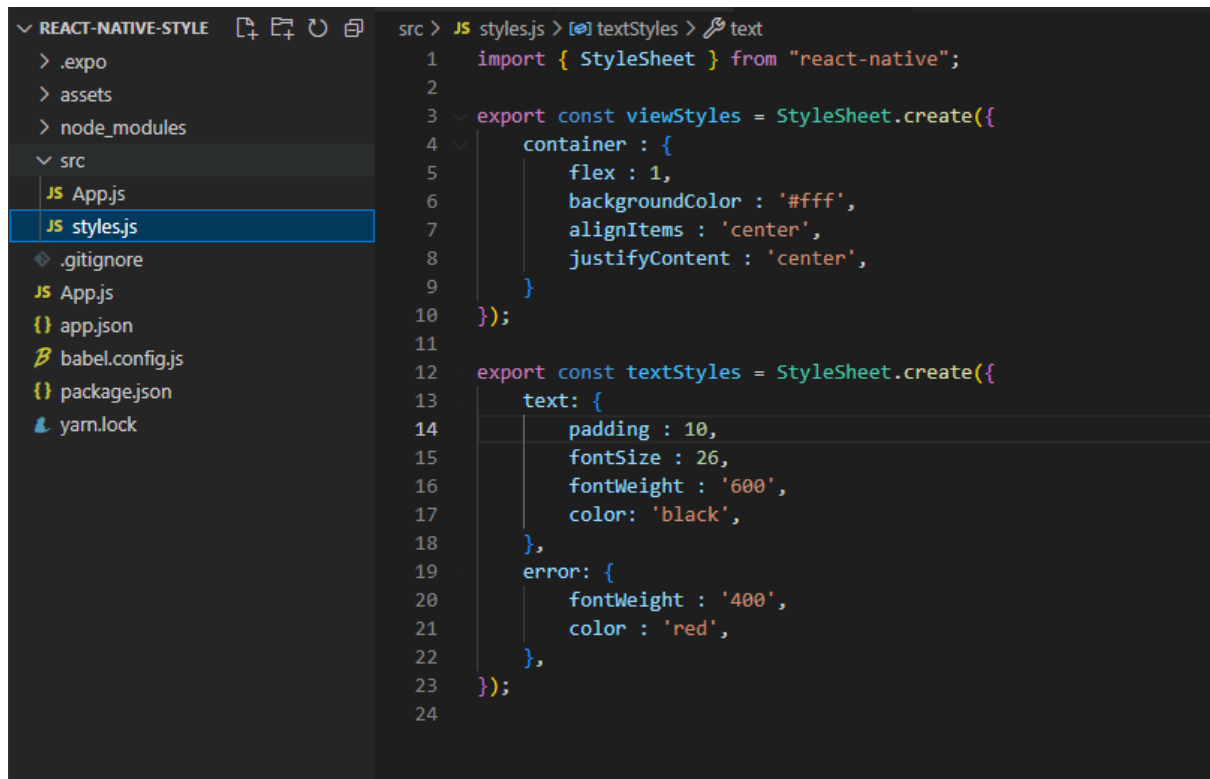
```
const App = ()=>{
  return(
    <View style={styles.container}>
      <Text style={[styles.text, {color : 'green'}]}>Inline Styling - Text</Text>
      <Text style={[styles.text, styles.error]}>Inline Styling - Error</Text>
    </View>
  );
};
```



## 외부 스타일 이용하기

만든 스타일을 다양한 곳에서 사용하고 싶은 경우 어떻게 해야할까 상황에 따라 외부 파일레 스타일을 정의하고 여러 개이— 파일에서 스타일을 공통으로 사용하는 경우가 있다.

외부 파일에 스타일을 작성하고 컴포넌트에서 외부 파일에 정의된 스타일을 이용하는 방법 에 대해 알아보자. src 폴더 밑에 styles.js파일을 생성하고 아래처럼 작성하자.



```
src > JS styles.js > [0] textStyles > text
1  import { StyleSheet } from "react-native";
2
3  export const viewStyles = StyleSheet.create({
4    container: {
5      flex: 1,
6      backgroundColor: '#fff',
7      alignItems: 'center',
8      justifyContent: 'center',
9    }
10  });
11
12  export const textStyles = StyleSheet.create({
13    text: {
14      padding: 10,
15      fontSize: 26,
16      fontWeight: '600',
17      color: 'black',
18    },
19    error: {
20      fontWeight: '400',
21      color: 'red',
22    },
23  });
24
```

View 컴포넌트에 적용되어 있던 스타일과 text컴포넌트에 적용했던 스타일을 나누고 둘 모두 외부에서 사용할 수 있도록 했다. 이제 app컴포넌트에서 styles.js 파일에 정의된 스타일을 이용하도록 수정하자.

```

src > JS App.js > ...
1  import React from "react";
2  import { View, Text } from 'react-native';
3  import { viewStyles, textStyles } from "../styles";
4
5  const App = ()=>{
6    return(
7      <View style={viewStyles.container}>
8        <Text style={textStyles.text, {color : 'green'}}>Inline Styling - Text</Text>
9        <Text style={textStyles.text, textStyles.error}>Inline Styling - Error</Text>
10      </View>
11    );
12  };
13
14
15  |
16
17  export default App;
```



---

## 리액트 네이티브 스타일

리액트 네이티브에는 많은 종류의 스타일 속성들이 있다. 그 중에는 특정 플랫폼에서만 적용되는 스타일도 있고 웹 프로그래밍에서 사용해본 익숙한 속성들도 있다. 이 책에서는 모든 스타일 속성을 다루지는 않으며, 자주 사용되는 중요한 스타일 속성들에 대해 알아보자.

### flex와 범위

화면의 범위를 정하는 속성에는 폭과 높이를 나타내는 width와 height가 있다. 리액트 네이티브에서도 width와 height를 설정할 수 있다.

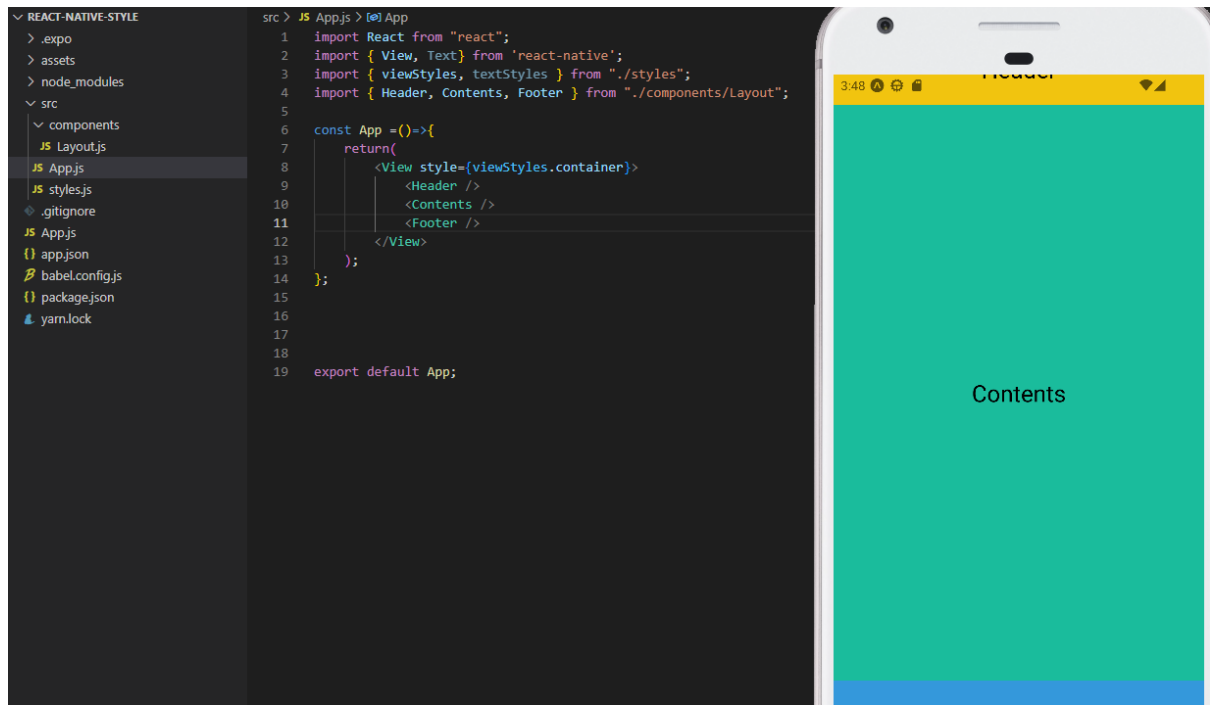
src폴더 밑에 components 폴더를 생성하고 Layout.js파일을 components 폴더 안에 생성하자.

Layout.js에는 Header 컴포넌트, Contents 컴포넌트, Footer 컴포넌트를 정의하자.

p.116 그림

```
src > components > JS Layout.js > ...
1  import React from "react";
2  import { StyleSheet, View, Text } from "react-native";
3
4  export const Header = () => {
5    return(
6      <View style={[styles.container, styles.header]}>
7        <Text style={styles.text}>Header</Text>
8      </View>
9    );
10 };
11 export const Contents = () =>{
12   return(
13     <View style={[styles.container, styles.contents]}>
14       <Text style={styles.text}>Contents</Text>
15     </View>
16   );
17 };
18 export const Footer = ()=>{
19   return(
20     <View style={[styles.container, styles.footer]}>
21       <Text style={styles.text}>Footer</Text>
22     </View>
23   );
24 };
25 const styles = StyleSheet.create({
26   container : {
27     width : '100%',
28     alignItems : 'center',
29     justifyContent : 'center',
30     height : 80,
31   },
32   header : {
33     backgroundColor : '#f1c40f',
34   },
35   contents : {
36     backgroundColor : '#1abc9c',
37     height : 640,
38   },
39   footer : {
40     backgroundColor : '#3498db',
41   },
42   text : {
43     fontSize : 26,
44   },
45 });
```

이제 app.js 에 컴포넌트를 추가하여 사용해보자.



기종마다 조금 다른 모습으로 출력된다.

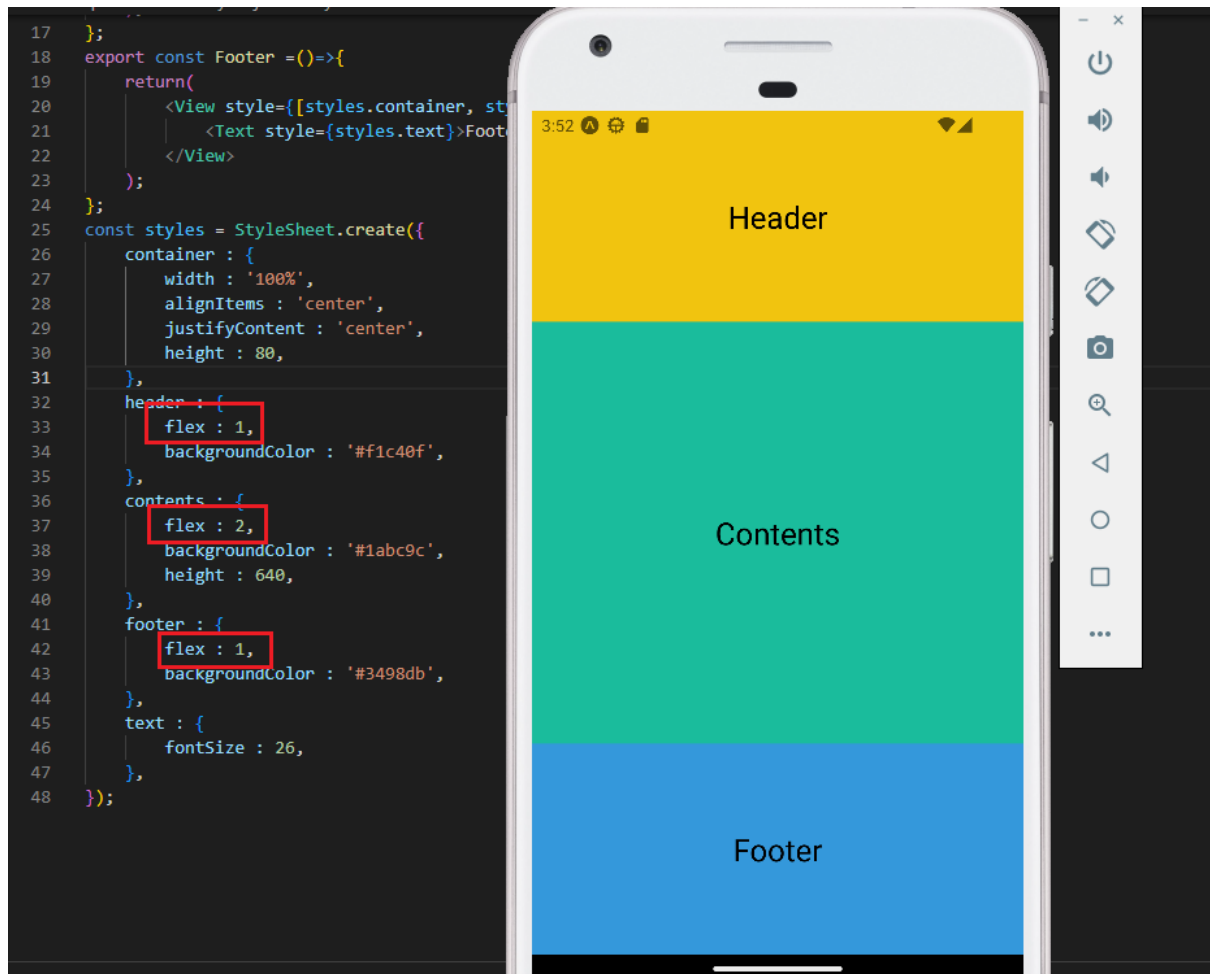
고정값을 이용하면 기기마다 화면 크기의 차이 때문에 서로 다른 모습으로 나타나고 이런 다양한 크기의 기기에 대응하기 어렵다.

이때 flex를 이용하면 문제를 해결할 수 있다.

flex는 width나 height와 달리 항상 비율로 크기가 설정된다. flex는 값으로 숫자를 받으며 값이 0일 때는 설정된 width와 height값에 따라 크기가 결정되고, 양수인 경우 flex값에 비례하여 크기가 조정된다.

예로 flex가 1로 설정된 경우 자신이 차지할 수 있는 영역을 모두 차지한다. 만약 동일한 부모 컴포넌트에 있는 컴포넌트 두 개의 flex값이 각각 1과 2로 연결되어 있다면, 두 개의 컴포넌트는 차지할 수 있는 영역을 1:2의 비율로 나누어 가진다.

만약 컴포넌트가 화면을 1:2:1의 비율로 나누어 채우게 하려면 아래 처럼 만들면 된다.



컴포넌트의 부모인 app컴포넌트는 화면 전체를 차지한다.

flex로 비율을 설정하면 나누어 채우게 된다.

다양한 크기의 기기에서 항상 그림 p.116처럼 동일하게 화면이 구성되도록 하려면 Header 컴포넌트와 Footer 컴포넌트의 높이를 80으로 고정하고 Contents 컴포넌트가 나머지 부분을 차지하도록 설정하면 된다.

```
JS App.js \ JS App.js src JS Layout.js X JS styles.js
src > components > JS Layout.js > styles > contents

17   };
18   export const Footer = () => {
19     return (
20       <View style={ [styles.container, styles.footer]} >
21         <Text style={styles.text}>Footer</Text>
22       </View>
23     );
24   };
25   const styles = StyleSheet.create({
26     container : {
27       width : '100%',
28       alignItems : 'center',
29       justifyContent : 'center',
30       height : 80,
31     },
32     header : {
33       backgroundColor : '#f1c40f',
34     },
35     contents : {
36       flex : 1,
37       backgroundColor : '#1abc9c',
38       height : 640,
39     },
40     footer : {
41       backgroundColor : '#3498db',
42     },
43     text : {
44       fontSize : 26,
45     },
46   });
```

정렬

flex Direction



지금까지는 항상 위에서 아래로 컴포넌트가 쌓였다. 화면을 구성하다 보면 컴포넌트가 쌓이는 방향을 변경하고 싶을 때가 있는데, 이때 `flexDirection`을 이용하면 컴포넌트가 쌓이는 방향을 변경할 수 있다.

그림 p.122

`flexDirection`으로 설정할 수 있는 값으로는 4가지가 있다.

- `column` : 세로 방향으로 정렬(기본값)
- `column-reverse` : 세로 방향 역순 정렬
- `row` : 가로 방향으로 정렬
- `row-reverse` : 가로 방향 역순 정렬

`flexDirection`은 자기 자신이 쌓이는 방향이 아니라 자식 컴포넌트가 쌓이는 방향이라는 것을 기억하자.

`justifyContent`

컴포넌트를 배치할 방향을 결정한 후 방향에 따라 정렬하는 방식을 결정하는 속성이 `justifyContent`와 `alignItems`이다. `justifyContent`는 `flexDirection`에서 결정한 방향과 동일한 방향으로 정렬하는 속성이고, `alignItems`는 `flexDirection`에서 결정한 방향과 수직인 방향으로 정렬하는 속성이다.

그림 p.123

- `flex-start` : 시작점에서부터 정렬( 기본값)
- `flex-end` : 끝에서부터 정렬
- `center` : 중앙정렬
- `space-between` : 컴포넌트 사이의 공간을 동일하게 만들어서 정렬
- `space-around` : 컴포넌트 각각의 주변 공간을 동일하게 만들어서 정렬
- `space-evenly` : 컴포넌트 사이와 양 끝에 동일한 공간을 만들어서 정렬

`alignItems`

그림 p.124

- flex-start :
- flex-start : 시작점에서부터 정렬( 기본값)
- flex-end : 끝에서부터 정렬
- center : 중앙정렬
- stretch : alignItems의 방향으로 컴포넌트 확장
- baseline : 컴포넌트 내부의 텍스트 (text) 베이스라인 (baseline)을 기준으로 정렬

justifyContent와 반대 방향이라고 생각하면 이해하기 쉽다.

---

## 그림자

그림자는 리액트 네이티브에서 플랫폼마다 다르게 적용되는 스타일 속성이다

리액트 네이티브에는 그림자와 관련해 설정할 수 있는 스타일 속성이 4가지 있다.

- shadowColor : 그림자 색 설정
- shadowOffset : width와 height값을 지정하여 그림자 거리 설정
- shadowOpacity : 그림자의 불투명도 설정
- shadowRadius : 그림자의 흐림 반경 설정

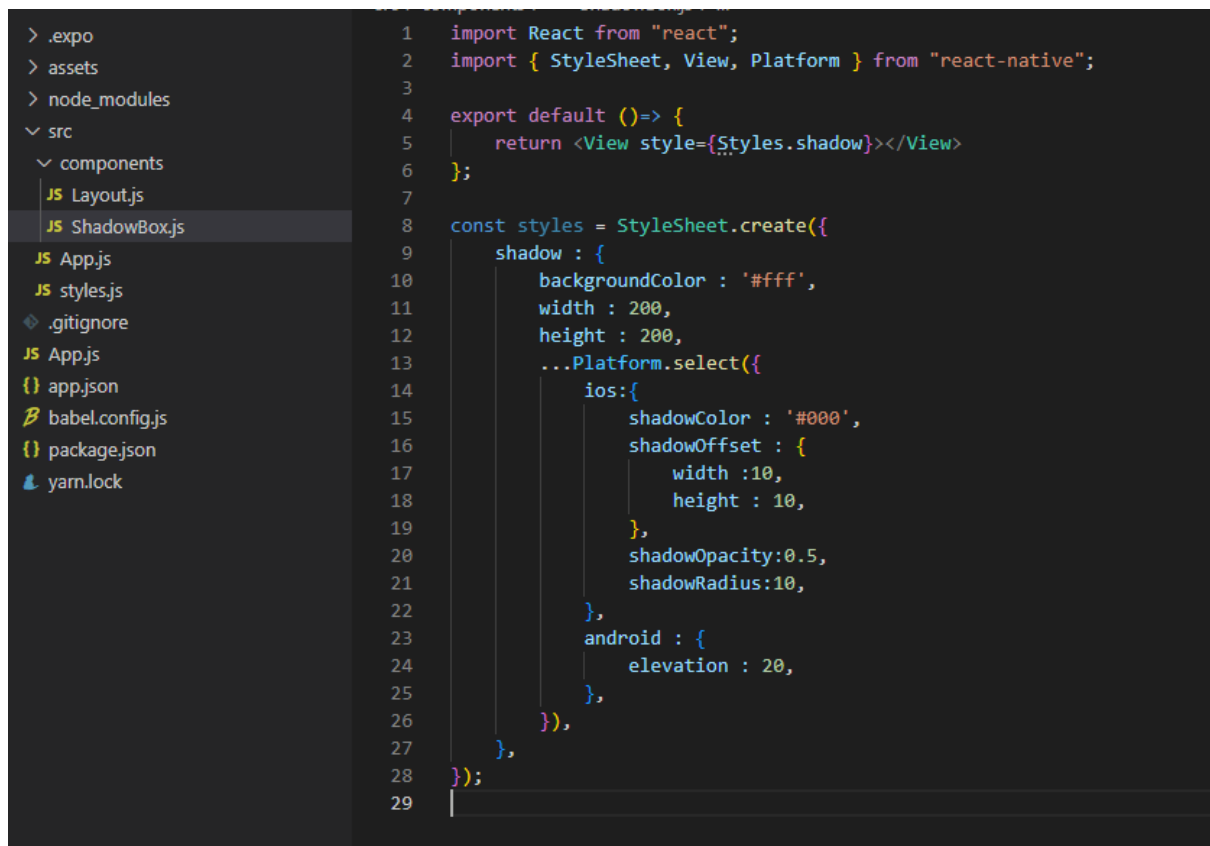
위 속성들로 그림자를 표현할 수 있지만, 이것들은 ios에서만 적용되는 속성들이다.

안드로이드에서 그림자를 표현하려면 elevation이라는 속성을 사용해야한다.

이렇게 각 플랫폼마다 적용 여부가 다른 속성이었다. 이 경우 리액트 네이티브에서 제공하는 platform 모듈을 이용해 각 플랫폼마다 다른 코드가 적용되도록 코드를 작성할 수 있다.

- Platform : <https://bit.ly/react-native-platform>

components 폴더 밑에 ShadowBox.js를 생성하고 Platform 을 이용해서 그림자가 있는 박스를 만들어 보자



```
1 import React from "react";
2 import { StyleSheet, View, Platform } from "react-native";
3
4 export default () => {
5   return <View style={Styles.shadow}></View>
6 };
7
8 const styles = StyleSheet.create({
9   shadow : {
10     backgroundColor : '#fff',
11     width : 200,
12     height : 200,
13     ...Platform.select({
14       ios:{
15         shadowColor : '#000',
16         shadowOffset : {
17           width :10,
18           height : 10,
19         },
20         shadowOpacity:0.5,
21         shadowRadius:10,
22       },
23       android : {
24         elevation : 20,
25       },
26     }),
27   },
28 });
29
```

Platform을 이용해서 ios와 안드로이드에서 스타일 코드가 다르게 적용되도록 했다.

이제 app.js에 컴포넌트를 적용시켜보자

```
src > JS App.js > [⌘] default
1  import React from "react";
2  import { View, Text } from 'react-native';
3  import { viewStyles, textStyles } from "../styles";
4  import { Header, Contents, Footer } from "../components/Layout";
5  import ShadowBox from "../components/ShadowBox";
6
7  const App =()=>{
8    return(
9      <View style={viewStyles.container}>
10        <ShadowBox />
11      </View>
12    );
13  };
14
15
16
17
18  export default App;
```

