#6 RN TodoApp 기능 부여하기

안녕하세요! 여러분

지난시간까지 우리의 **TodoApp Layout**을 만들어 보았습니당

슬슬 RN에 만들기에 재미가 생기시나요!!!!!! **~~저는 재밌습니다..~~**

오늘 이시간에는 우리가 만든 Layout에 활력을 불어넣어 줄겁니다 = **기능을 넣어줄꺼에요**

State라는 애를 이용할껀데요!!

State는 React개념 설명할때도 언급을 했던 내용이지만 변경가능한 값입니다.

(State의 자세한 개념은 React의 시계예제로 설명한 State를 참고해주세요)

▼ 잠깐 왜 State를 사용하나요?

> 우리는 **Props를 이용해 재사용을 해줄 수 있고 어떤 data를 component에 전달해 줄수 있었지만 데이터를 추가하거나 변경하는 작업은 불가능** 했습니다
>
> 따라서 **이와같은 작업을 해주기 위해서 변경 가능한 State라는 애를 가져와서 사용**할 것입니다.

그럼 이번시간에 해야할 학습목표(?)부터 살펴보고 갈께여

1. **할일 추가하기**
2. **완료한일은 완료 체크하기**
3. **삭제버튼 누르면 삭제 되기**

우선 작업에 앞서

state를 사용하기 위한 코드변경을 해보겠습니다.

우리는 지금까지 **App component**를 **function component**로 사용했습니다

하지만 이는 **props**에 특화되어있는거구
State를 사용하기 위해서는 **class component**로 바꿔줘야 합니다

component의 naming, 속성의 naming 등이 이전 자료와 조금 상이할 수 있습니다 ㅠㅠ
따라서 제 코드를 그대로 복붙하시면 에러가 발생할 확률이 높으니
전체적인 구조를 파악하며 함께 실습해보도록 해요

class component

```
//App.js
export default class App extends React.Component {
  constructor(props){
      super(props);  //상속받는 부모 class의 속성도 받아오는거에요
      this.state={
      todos:[
        {
          title : "나는 공부같은거 하지않아",
        },
        {
          title: "일찍 일어날래",
        },
      ]
    }
  }

  render() {
    return (
      <View style={styles.container}>
        <View style={styles.headercenter}>
          <Header />
        </View>
        <View style={styles.subtitleposition}>
            <Subtitle title="해야 할 일"/>
          <Input/>
        </View>
        <View style={styles.subtitleposition}>
          <Subtitle title="해야 할 일 목록" />
          <TodoItem text="코딩하기"/>
          <TodoItem text="운동하기" />

      </View>
    </View>
  );
  }
}
```

function component

```
//App.js
export default function App() {
  return (
    <View style={styles.container}>
      <View style={styles.headercenter}>
        <Header />
      </View>
      <View style={styles.subtitleposition}>
          <Subtitle title="해야 할 일"/>
        <Input/>
      </View>
      <View style={styles.subtitleposition}>
        <Subtitle title="해야 할 일 목록" />
        <TodoItem text="코딩하기"/>
        <TodoItem text="운동하기" />

      </View>
    </View>
  );
}
```

class component가
function component보다 복잡해보이죠?

기존의 function compo return을 **render() 메서드 안으로**
옮겨주는게 핵심이에요

저는 constructor를 이용해

초기 state값도 설정해 줬어요

할일 추가하기

이제 아래와 같이 state를 Props로 넘겨줄수 있습니다(state를 사용할수 있게 되었습니다!)

```
//App.js

//props만 사용한경우
<TodoItem text="코딩하기"/>


//초기 state값을 이용
<TodoItem text={this.state.todos[0].title}/>
```

위의 코드는 선언된 state에서 todos라는 list를 가져와 0번째 객체를 가져오는데
객체에서 title에 대응되는 value를 가져오는 code입니다


JS의 객체 모양에 대해 조금만 알고계시다면 그렇게 어렵지 않은 구조이니까 차근차근 따라가봅시다

위의 식은 state를 사용할수 있지만 생산성 측면에서는 아직도 안 좋은것 같지않나요.
todos가 3000개가 되어도
this.state.todos[0]... this.state.todos[1]... 이렇게 component를 생성할 수는 없잖아요???

FlatList

RN에서 제공하는 component로 List를 구현할때 사용합니다

```
//App.js
import { StyleSheet, Text, View,FlatList } from 'react-native';

  ...중략...

        <TodoItem text="코딩하기"/>
        <TodoItem text="운동하기" />
        <FlatList
        data={this.state.todos}
        renderItem={this._makeTodoItem}  //_makeTodoItem은 우리가 Item을 render하는 방법에 대해 나타내고 후에 작성해줍니다
        keyExtractor={(item, index) => { return `${index}`}}/>
```

이처럼 작성을 해주게 되면 우리가 App compo에서 가지고 있는
state의 내용들을 가져와 List의 형태로 보여주게 됩니다.

FlatList의 속성

- `data` : 랜더링 해줄 data를 의미, 해당 값은 배열을 받습니다
  (List의 요소들이니까 여러개의 애들을 받아와야겠져)
- `renderItem` : data의 원소 **하나하나**를 기준으로 그려주는 방식을 지정한다고 생각해줍시다

  마치 App component의 render함수가 component를 return받아 그려주는 것과 비슷하게 renderItem도 component를 그려주는역할을 하고 있습니다.

- `keyExtractor` : 각 Item들의 key를 생성하여 지정해줍니다
  (이때 unique한 값을 key로 사용하는 것이 좋습니다. **사실 index를 key로 하는 건 지양해야함**)

  ▼ key는 왜 사용해야 하나요?

  ```
  <ul>
    <li>Duke</li>
    <li>Villanova</li>
  </ul>

  <ul>
    <li>Connecticut</li>
    <li>Duke</li>
    <li>Villanova</li>
  </ul>
  ```

  위와 같은 code가 있을 때 React는 차이점이 있으면 변경사항을 생성해줍니다

  그런데 첫번째 부터 달라지면 비효율적으로 작동하게 되는데

  ```
  <ul>
    <li key="2015">Duke</li>
    <li key="2016">Villanova</li>
  </ul>

  <ul>
    <li key="2014">Connecticut</li>
    <li key="2015">Duke</li>
    <li key="2016">Villanova</li>
  </ul>
  ```

  이렇게 Key 값을 부여해서 작성을 해주게 되면

  key를 통해 기존 구조와 새로운 구조가 일치하는지 확인합니다

  key="2014"만 새로 생겼으므로 해당 element만 새로 추가해주면 되므로 보다 효율적으로 동작합니다.

  일반적으로 **index는 변할수 있기때문에 key값으로 사용하는 것을 지양**하지만
  우리는 편의를 위해서 index값을 key값으로 사용하도록 하겠습니다.

  ▼ **return `${index}` 설명**

```
var a = 5;
var b = 10;
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

위의 식을 보면 알수 있듯 ` 은 string안에 문법적인 요소를 사용하기 위해 사용했다고 이해해주면 되겠습니당

_makeTodoItem method 작성

위에서 우리는 `renderItem(this._makeTodoItem)`와 같이 작성해 주었습니다.

`renderItem` 은 함수를 인자로 받는데요!

```
renderItem={({item, index, separators}) => {
    //함수내용
    return //return값
}};
```

위와 같은 형태로 renderItem은 함수를 받아올 수 있는거에요.
그래서 우리는 저기 익명함수 자리에 method를 넣어주겠다고 한거죠

그리고 item,index,separators를 **renderItem이 받는 함수의 arguments로 사용**할수 있는데요

우리의 data를 render하기위해서 **item와 index를 가져와서 사용**해주도록 합시다

```
  //App.js
  //..중략..

      },
        {
          title: "일찍 일어날래",
        },
      ]
    }
  }

_makeTodoItem = ({ item, index }) => {
    return (
      <TodoItem text={item.title}/>
    )
}
```

이렇게 App component 안에 method로 `_makeTodoItem`을 만들어주고
`this.makeTodoItem`으로 가져와서 `renderItem`에서 사용해주는 겁니다

🙂 여기서 `_makeTodoItem이` method로 사용되었기에 앞에 접두사로 _가 붙었습니다

그림

이제 우리는 `TodoItem` Component를 하나하나 작성해 주지않고
`FlatList`를 이용해 간편하게 만들 수 있게 되었습니다
(우리가 작성했던 `TodoItem` component는 `FlatList` 때문에 필요없어졌으니 지우도록 해요)

하지만 초기 state에 의해 TodoList의 item을 생성하는 것은 구현했지만
우리가 입력한 **Input값에 대해 TodoList Item이 생성**되어야 진정한 TodoApp이겠죠?

그럼 입력한 **Input값이 우리의 state로 들어가**야하고
그 **state를 FlatList가 자동으로 compo로 생성**해주는 상상을 해볼수 있습니다.

Input으로 state 추가하기

```
//App.js
  constructor(props){
      super(props);


      this.state={
      inputValue: "",  //input창에 보여지는 Text입니다 초깃값은 공백! Placeholder만 보여줌
      todos:[
        {
          title : "나는 공부같은거 하지않아",
        },
        {
          title: "일찍 일어날래",
        },
      ]
    }
  }
  ... 중략...

  <SubTitle title="To-Do 입력"/>
        <Input
          value = {this.state.inputValue}
          changeText={this._changeText} //입력할때마다 글자가 별하니 _changeText를 실행시켜줌
          addTodo={this._addTodoItem}/> //입력이 완료되고 확인을 누르면(enter) 저장시키는 함수
 ...중략...
```

위에서 Input에 Props로 전달되고 있는 `_changeText` 와 `_addTodoITem` *method부터 만들어봐요*

```
//App.js

_makeTodoItem = ({ item, index }) => {
    return (
      <TodoItem text={item.title}/>
    )
};

_changeText = (value) => {
    this.setState({inputValue: value});
}

_addTodoItem = () => {
  if(this.state.inputValue != ''){
  const prevTodo = this.state.todos;  //현재의 todos를 prevTodo에 넣습니다.

  const newTodo = { title: this.state.inputValue}; //현재 input창에 있는 값을 새로운 할일로 등록

  this.setState({
    inputValue: '', //TodoItem이 추가되면 입력창은 비어야하므로
    todos: prevTodo.concat(newTodo) // 이전의 TodoItem에 새 Todo를 이어붙여 todos값으로 변경
  });
}
}
```

`_changeText` method는 어렵지 않죠?? **(입력창의 글자가 변할때마다 실행되는 method)**

value라는 애를 받아서 `this.setState()` 를 이용해 `inputValue` 라는
우리의 State내부의 값을 변경시켜준다는 의미입니다

(`setState` 의 역할이 궁금하신들께서는 #4 RN 들어가기전 React를 참고하세요)

`_addTodo` method를 분석해 봅시다 **(입력을 끝내고 enter를 누를때 실행되는 method)**

1. 지금까지 가지고있던 todos를 prevTodo라는 상수에 저장
2. 입력창에서 받아올 애를 newTodo라는 상수에 저장
3. setState를 이용해 입력창은 공백으로, prevTodo와 newTodo를 concat으로 이어서 todo로 set

그러면 method준비는 끝났고

우리가 Input 이라는 component에 지금 value와 `_changeText` 를 Props로 넘겨주고 있으니까

Input.js도 수정해서 Props를 사용할 수 있게 해줘야겠습니당

```
//Input.js

const Input = ({value,changeText,,addTodo}) => (
    <TextInput
        value={value}
        style={styles.input}
        placeholder={"오늘 어떤 일을 하실건가요?"}
        maxLength={30}
        onChangeText={changeText}   //입력창에 글자가 바뀌면 onChangeText가 활성화 된다
```

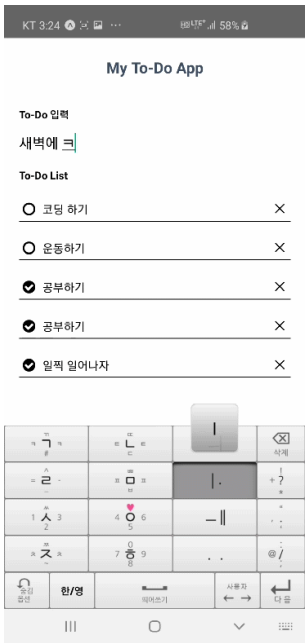
```
        onEndEditing={addTodo}      //입력이 끝나고 enter를 누르면 onEndEdition이 활성화
        returnKeyType="done"/>
);
```

위의 모양처럼 Props로 넘겨준 애들을 받아주는데

> **Input component의 동작 과정을 한번 차근차근 살펴봅시다**

1. 우리의 Input component는 실제로는 `<TextInput>`이라는 애를 만들어주는 역할입니다.
2. 입력이 이루어지면 `onChangeText`가 동작하며 `chageText`로 넘어온 함수를 실행시킵니다
3. App.js 에서 정의된 `_changeText`가 실행되며
   입력된 순간의 `value`를 인자로 받아 `setState`를 이용해 inputValue를 수정
4. InputValue가 수정되었으므로 `this.state.inputValue`도 수정되고 Input compo의 `value`가 다시 들어가고 **(rerendering)** 해당 value가 TextInput에 보여지고 입력창의 value로 자리잡는다
5. 완료(혹은 확인 혹은 Enter)버튼을 누르게되면 `onEndEditing`이 실행되게 되고 넘어온 `addTodo`함수를 실행해 주며 todos와 inputValue를 변경시켜준다
6. 2번에서 대기한다

Input이라는 component와 TextInput이라는 Component의 속성을 수정해주며 Props를 넘겨주는 과정이 있다 보니 헷갈리시겠지만 위의 번호를 차근차근 보시면 구조를 그리는데 도움이 될 것 같아요



완료한일은 완료 체크하기

그전에 우리는 체크버튼에 하나의 icon만을 해놨습니당 따라서 완료표시의 icon을 하나 더 추가해요

```
//TodoItem.js
const TodoItem = ({ text, isComplete}) =>(
<TouchableOpacity>
    <FontAwesome name={isComplete?"circle-o":"check-circle"} size={20} style={styles.check}/>
</TouchableOpacity>
```

위와 같이 **isComplete라는 Props를 추가**로 줘서 삼항연산자로 조건문을 달아줍시다

삼항연산자 ? 한줄로 적힌 If문이라고 생각해주시면 됩니다

isComplete가 **True이면 "circle-o" false이면 "check-circle"**이라는 이름의 icon이 되는겁니다.

그리고 현재 TodoItem을 만들때
(사실 정확하게는 TodoItem이 아니라 state를 추가해서 자동으로 TodoItem이 생성되게 하는거지만)
default로 isComplete 값을 같이 넘겨줘서 우리의 listitem들이 isComplete속성을 가지게 합시다

```
//App.js
_addTodoItem = () => {
   if(this.state.inputValue != ''){
  const prevTodo = this.state.todos;
  const newTodo = { title: this.state.inputValue, isComplete: false};
  this.setState({
    inputValue: '',
    todos: prevTodo.concat(newTodo)
  });
}
}
```

그리고 위처럼 추가된 state를 기반으로 List의 Item들을 만들어갈때
isComplete라는 속성을 제어할수 있게 `_makeTodoItem`도 건드려 볼께요

```
//App.js
_makeTodoItem = ({ item, index }) => {
    return (
      <TodoItem
        text={item.title}
        isComplete={item.isComplete}
        changeComplete={() => {
          const newTodo = [...this.state.todos];
          newTodo[index].isComplete = !newTodo[index].isComplete;
          this.setState({todos:newTodo});
        }} />
```

```
    );
  }
```

- `...this.state.todos` : ...(three dot) 은 `this.state.todos`의 요소들을 하나하나 펼쳐놓습니다
  따라서 `const newTodo` 에 새로운 todos list가 들어가게 되는 것입니다

- `!newTodo[index].isCompletes` : `_makeTodoItem`에서 받아오는 index로 newTodo에서
  index를 찾아 changeComplete가 실행되면
  해당 index의 isComplete값을 !를 통해 반전이 되게 했습니다.

> 😀 renderItem이라는 속성이 data라는 속성의 배열값을 하나하나 돌면서
> renderItem으로 받은 함수를 실행시키기때문에
> 우리는 index값을 가져와서 사용할수 있었습니다.

- `this.setState`를 통해 newTodo를 todos에 적용해주었습니다

그럼 이제 위에서 설정한 changeComplete라는 Props로 넘겨주는 함수가 실행되도록
TodoItem.js에 들어가서 Prop의 실행조건을 달아봅시다

```
//TodoItem.js
const TodoItem = ({ text, isComplete,changeComplete}) =>(
<TouchableOpacity
    onPress={changeComplete}>
<FontAwesome name={isComplete?"circle-o":"check-circle"} size={20} style={styles.check}/>
</TouchableOpacity>
```

Press라는 동작이 발생하면 Props로 넘어온 changeComplete가 실행되고
이는 우리의 Todos에서 해당하는 **index의 isComplete**요소를 반전시켜줍니다

그럼 다시한번 완료버튼을 추가한 구조를 간단히 그리고 넘어가면

- Input Text창에서 입력된 값을 addTodoItem이 우리의 state로 저장해준다
  (이때 complete의 값은 default로 false이다)

- 그럼 state의 todos가 변경되고 FlatList에 의해서 data들이
  _makeTodoItem을 통해 component로 리턴된다

- 그 안에는 우리가 추가해준 changeComplete와 같은 속성이 있기때문에 우리는 이제 isComplete속성도 다룰수 있게 되었다.

이렇게 완료버튼을 체크하는 기능까지 넣었습니다

삭제버튼 누르면 삭제되기

```
//App.js

    <TodoItem
        text={item.title}
        isComplete={item.isComplete}
        changeComplete={() => {
          const newTodo = [...this.state.todos];
          newTodo[index].isComplete = !newTodo[index].isComplete;
          this.setState({todos:newTodo});
        }}
        deleteItem={() => {
          const newTodo = [...this.state.todos];
          newTodo.splice(index,1);
          this.setState({todos:newTodo});
        } />
```

```
    );
  }
```

changeComplete 익명함수를 작성한것과 유사하게

deleteItem은 newTodo라는 새로운 객체 list를 만들어서

newTodo에서 splice를 통해 객체를 삭제해줍니다. 그리구 setState로 저장해주면 끝입니다!!!

▼ **splice**?

`splice( start, deleteCount,item... )`

- start : 배열 시작위치
- deleteCount start에서부터 삭제할 갯수
- item : 삭제한 위치에 추가할 요소

```
//TodoItem.js
<TouchableOpacity
    onPress={changeComplete}>
<FontAwesome name={isComplete?"circle-o":"check-circle"} size={20} style={styles.check}/>
</TouchableOpacity>
<Text style={styles.todos}>{text}</Text>
</View>
<TouchableOpacity
        onPress={deleteItem}>
    <AntDesign name="close" size={20}/>
</TouchableOpacity>
```

 이제 삭제까지 구현해서 우리는 기본적인 App의 기능을 모두 만들었습니다

그런데 뭔가 좀 허전하지 않나요

개발하면서 **새로고침을 할때마다 우리가 입력하거나 변경했던 state들이 날아가고 있습니다.**

따라서 지금부터느 RN이 제공해주는 간단한 저장소를 통해 해당 내용들을 저장해보도록 할께요

AsyncStorage

- AsyncStorage는 RN에 내장된 key-value방식의 저장소입니다(DB와는 다릅니다!)
- String만 저장가능합니다. Object를 저장할수 없습니다
- sqlite나 다른 DB를 적용하는것보다 data를 저장함에 있어 성능이 아주 좋으므로
  간단한 데이터를 저장하고 가져올때는 AsyncStorage로 구현하는 것이 좋습니다.

▼ **AsyncStorage이 없어진다구요?(Deprecated)**

지금 보조자료를 만들고 있는 시점에서는 RN 의 버전이 0.60이라
기본 compo로 AsyncStorage를 사용할수 있습니다. 하지만 공홈에 Deprecated라고 명시해놔서
아마도 다음 버전부터 core에서 AsyncStorage가 빠질텐데요.

그래서 공홈에서는

> react-native-community/async-storage
> https://github.com/react-native-community/async-storage

이곳을 이용해서 사용하라고 권장하고 있습니다.

하지만 우리는 expo라는 RN을 편리하게 개발할수 있게 해주는 tool을 사용하고 있기때문에
저 방법을 사용할 수가 없어요
(제가 강의영상에서 초기에 expo라는 툴을 이용하면 native code를 붙이는데 제한이 있다고 했는데 지금 이 이슈가 그 이슈에요)

따라서

> Documentation
> https://docs.expo.io/versions/v34.0.0/sdk/webview/

여기 expo를 이용하시면 되는데요. 이전에 webview도 asyncStorage처럼 RN에 있다가 빠진 애라서 expo가 저런방식으로 제공해주고 있습니다.
따라서 asyncStorage도 빠지게 되면 expo에서 제공을 해줄 거라는 의견이 많아서

만약 asyncStorage가 빠지게 되면 expo sdk의 latest version을 한번 확인해보도록 해요

어느 저장소나 그렇듯 data를 저장하는 부분과 불러오는 부분으로 구성되어야 할 것 같습니다.

우선 저장하는 부분

```
//App.js
import { StyleSheet, Text, View, FlatList,AsyncStorage } from 'react-native';
... 중략 ...

      {
        title: "일찍 일어나자",
        isComplete: false,
      },
    ]
  }
}

storeData =() => {
     AsyncStorage.setItem('@todo:state', JSON.stringify(this.state));
  }
```

`storeData`

AsyncStorage는 **key와 value방식의 저장소**라 했죠?

위에서 `'@todo:state'` 부분이 key `JSON.stringify(this.state)` 가 value부분이 되는 것입니다

일반적으로 **@앱이름:키이름** 으로 key를 작성해주게됩니다.(특히 문자열로 묶여있는게 보이시죠!)

또한 value부분에는 우리의 this.state가 현재 object모양으로 되어있기때문에 이를 `JSON.stringify()` 를 이용해서 JSON형태의 문자로 변경해 저장해줍니다.

▼ JSON은 무엇인가요?

Java**S**cript **O**bject **N**otation으로 쉽게 JS의 객체 모양을 빌려온 **data format**이라고 생각해줍시다

그리구 위에서 작성해준 storeData가 state의 변경시점마다 적용될수있게 배치해줘야합니다.

```
_makeTodoItem= ({item,index}) =>{
      return (
        <TodoItem

        changeComplete={()=>{
          const newTodo=[...this.state.todos];
          newTodo[index].isComplete = !newTodo[index].isComplete;
          this.setState({todos:newTodo},this.storeData);
        }}
        deleteItem={()=>{
          const newTodo = [...this.state.todos];
          newTodo.splice(index,1);
          this.setState({ todos: newTodo }, this.storeData);
        }}/>
      );
  }

_addTodoItem = () => {
    this.setState({
      inputValue: '',
      todos: prevTodo.concat(newTodo)
    }, this.storeData);
}
}
```

`setState`가 실행되고 변경사항을 저장소에 반영해줘야 할 위치에
callback함수의 자리에 `storeData`함수를 넣어주도록 합시다
(setState함수는 두번째인자로 callback함수를 받을수 있어요)

그럼 setState가 실행되면서 storeData가 실행되고 rerendering이 될꺼에요

▼ callback함수란?

다른 함수의 argument로 들어간 function을 callback함수라고 생각해주세여!

가져오는 부분

우리는 `storeData`로 data저장까지 완료했으니
`getData`라는 애도 선언해서 Data를 가져올수 있게 해줍시다.

```
///App.js
storeData =() => {
     AsyncStorage.setItem('@todo:state', JSON.stringify(this.state));
  }

getData =() => {
     AsyncStorage.getItem('@todo:state').then((state)=>{
      if (state !== null) {
        this.setState(JSON.parse(state));
    }
  }
```

`getData`

storeData에서 저장한 내용을 들고오는 부분입니다.

`getItem` method를 이용해서 key값으로 우리의 data를 불러온 다음
`then`을 이용해서 **getItem이 가져온 `state` 값을 인자로 받아 함수를 실행**시켜주는 겁니다

그리고 `getItem`이 저장소를 뒤져서 가져온 state값이 null이 아니면
우리의 State를 다시금 state로 `setState`해주게 되는데요

이때 우리가 store로 저장소에 저장한 값은 **JSON형식의 string data**이므로
해당 **string data를 JSON.parse()를 이용해 다시 object로 바꿔서 우리의 state로 사용**해줍시다

▼ then 이 뭔가요?

then은 javascript에서 비동기방식으로 처리를 하기위해
promise객체를 넘겨받아 처리해주는 방식이에요

**쉽게 풀어 설명드리자면 getItem이 꼭! 실행되고 난 뒤에
state가 null인지 검사하는 익명함수를 실행시켜줘라는 의미입니다**

▼ 혹시라도 then을 쓰고 싶지 않은 분들을 위해

then을 쓰고 싶지 않다고 해서 Javascript에서 비동기코드를 핸들링 하기위해 callback함수를 쓴다면 callback 지옥에 빠지게 될수 도 있고 가독성도 좋지 않게 될꺼에요

따라서 **async & await** 라는 키워드를 찾아보시면

```
_storeData= async ()=>{
    try {
      await AsyncStorage.setItem('@todo:state',JSON.stringify(this.state));
    }catch(e){

    }
  }


  _getData = async() =>{

    try{
      const mystate = await AsyncStorage.getItem('@todo:state');
      if (mystate !== null) {
        this.setState(JSON.parse(mystate));

      }
    }catch(e){

    }

  };
```

이렇게 코드가 직관적이고 뭔가(?) 깔끔해진것을 볼수 있습니다

~~물론 예외처리 때문에 길어보이는것은 착각입니다~~

그리고 `getData` 도 적절한 위치에 배치해봅시다

```
componentWillMount(){
    this.getData()
  }

  storeData=()=>{
    AsyncStorage.setItem('@todo:state',JSON.stringify(this.state));
  }
```

data를 가져오는 부분은 **우리의 앱이 화면에 나타나기전에 해줘**야하므로

`componentWillMount()` 라는 method를 가져와서 `getData`를 실행시켜주겠습니다

`componentWillMount()` 는 **React 의 life cycle**이라는 개념을 알고 계시다면 쉽게 이해가 가실거에요

제가 RN 들어오기전 React 의 개념을 간단하게 설명드릴때
state를 설명하면서 설명을 드린적이 있는데요
Mount 라는 것은 HTML 코드에 react element들이 rendering되는 것을 M**ount** 라고 합니다

따라서 `willMount`라고하면 미래에 마운트 될거니까 ~~미래형... 예~~
아직 rendering 이 되지 않은 상태잖아요?

그래서 우리는 getData를 이용해서 data를 가져오고 Mount(rendering)을 진행하게 되는 것입니다.

지금까지 잘 따라하셨다면 저장까지 되는 기본적인 TodoApp이 만들어졌을꺼에요