React Fundamentals

React Native runs on React, a popular open source library for building user interfaces with JavaScript. To make the most of React Native, it helps to understand React itself. This section can get you started or can serve as a refresher course.

We're going to cover the core concepts behind React:

- components
- JSX
- props
- state

Your first component

The rest of this introduction to React uses cats in its examples: friendly, approachable creatures that need names and a cafe to work in. Here is your very first Cat component:

Functional Component

```
1. import React from 'react';
2. import { Text } from 'react-native';
3.
4. const Myname = () => {
5.   return (
6.     <Text>Hello, My name is John Deo</Text>
7.   );
8. }
9.
10.  export default Myname;
11.
```

Class Component

Class components tend to be a bit more verbose than function components.

```
1. import React, { Component } from 'react';
2. import { Text } from 'react-native';
3.
4. class Myname extends Component {
5.   render() {
6.     return (
7.       <Text>Hello, I am your John Deo!</Text>
8.     );
9.   }
10.  }
11.
12.  export default Cat;
13.
```

You additionally import `Component` from React:

```
1. import React, { Component } from 'react';
```

Your component starts as a class extending `Component` instead of as a function:

```
class Myname extends Component {}
```

Class components have a `render()` function. Whatever is returned inside it is rendered as a React element:

```
class Myname extends Component {
  render() {
    return <Text>Hello, I am your John Deo!</Text>;
  }
}
```

And as with function components, you can export your class component:

```
class Myname extends Component {
  render() {
    return <Text>Hello, I am your Joh!</Text>;
  }
}

export default Myname;
```

Now take a closer look at that `return` statement. `<Text>Hello, I am your cat!</Text>` is using a kind of JavaScript syntax that makes writing elements convenient: JSX.

JSX

React and React Native use **JSX,** a syntax that lets you write elements inside JavaScript like so: `<Text>Hello, I am your cat!</Text>`. The React docs have [a comprehensive guide to JSX](#) you can reference to learn even more. Because JSX is JavaScript, you can use variables inside it. Here you are declaring a name for the cat, `name`, and embedding it with curly braces inside `<Text>`.

```
import React from 'react';
import { Text } from 'react-native';
const Myname = () => {
  const name = "John Deo";
  return (
    <Text>Hello, My name is {name}!</Text>

{/*
Here name is a string variable extract from
 const name = "John Deo"
 */}
   );
 }

 export default Myname;

```

```
Out put will be : Hello, My name is John Deo!
```

Any JavaScript expression will work between curly braces, including function calls like `{getFullName("John", "Ram", "Ismail")}`:

```
1. import React from 'react';
2. import { Text } from 'react-native';
3.
4. const getUserNames = (firstName, secondName, thirdName) => {
5.   return firstName + "," + secondName + "," + thirdName;
6. }
7.
8. const Usernames = () => {
9.   return (
10.      <Text>
11.        Hello, I am { getUserNames("John", "Ram", "Ismail")}!
12.      </Text>
13.  {/*Here we are running a function and returns a value that we written 'John Ram
   Ismail'*/}
14.    );
15.  }
16.
17.  export default Usernames;
18.
19.  //OUTPUT WILL BE : John,Ram,Ismail*/
20.
```

Custom Components

You've already met React Native's Core Components. React lets you nest these components inside each other to create new components. These nestable, reusable components are at the heart of the React paradigm.

For example, you can nest `Text` and `TextInput` inside a `View` below, and React Native will render them together:

```
1. import React from 'react';
2. import { Text, TextInput, View } from 'react-native';
3.
4. const Name = () => {
5.   return (
6.     <View>
7.       <Text>Hello, I am...</Text>
8.       <TextInput
9.         style= {{
10.            height: 40,
11.            borderColor: 'gray',
12.            borderWidth: 1
13.          }}
14.          defaultValue="Name me!"
15.        />
16.     </View>
17.   );
18.  }
19.
20.  export default Name;
21.
```

Output will be

Hello I am...

NAME ME! => TextInput

Developer notes

On Android, you usually put your views inside LinearLayout, FrameLayout, RelativeLayout, etc. to define how the view's children will be arranged on the screen. In React Native, View uses Flexbox for its children's layout. You can learn more in our guide to layout with Flexbox.

You can render this component multiple times and in multiple places without repeating your code by using `<Name>`:

```
1. import React from 'react';
2. import { Text, TextInput, View } from 'react-native';
3.
4. const Name = () => {
5.   return (
6.     <View>
7.       <Text>I am also a John Deo!</Text>
8.     </View>
9.   );
10.  }
11.
12.  const Cafe = () => {
13.    return (
14.      <View>
15.        <Text>Welcome!</Text>
16.        <Name />
17.        <Name />
18.        <Name />
19.      </View>
20.    );
21.  }
22.
23.  export default Cafe;
24.
```

Output will be :

Welcome!
I am also a John Deo!
I am also a John Deo!
I am also a John Deo!

Any component that renders other components is a **parent component.** Here, `Cafe` is the parent component and each `Name` is a **child component.**

You can put as many cats in your cafe as you like. Each `<Name>` renders a unique element—which you can customize with props.

Also you can import component from other files (...)

In react-native we will create files with .js extension

So here I have two files

1. App.js
2. TextComp.js

Now I want to show TextComp.js in App.js , In React it's very easy as we think.

Let's Look.

App.js

```
1. import React from 'react';
2. import { View } from 'react-native';
3. import TextComp from './TextComp.js'
4. const App = () =>{
5. return (
6. <View>
7. <TextComp/>
8. </View>
9. );
10.  }
11.  export default App;
12.
```

TextComp.js

```
1. import React from 'react';
2. import { View, Text } from 'react-native';
3. const TextComp = () => {
4. return (
5. <View>
6. <Text>Hello Myname is John Deo<Text/>
7. </View>
8. );
9. }
10.  export default TextComp;
11.
```

Output will : Hello My name is John Deo

Props

Props is short for "properties." Props let you customize React components. For example, here you pass each `<Name >` a different `name` for `User's` to render:

```
1. import React from 'react';
2. import { Text, View } from 'react-native';
3.
4. const Name = (props) => {
5.   return (
6.     <View>
7.       <Text>Hello, I am {props.name}!</Text>
8.     </View>
9.   );
10.  }
11.
12.  const Users = () => {
13.    return (
```

```
      <View>
        <Name name="Maru" />
        <Name name="Jellylorum" />
        <Name name="Spot" />
      </View>
    );
  }

  export default Users;
```

Output will be :

Hello, I am Maru!
Hello, I am Jellylorum!
Hello, I am Spot!

Here take a look at

Here we are passing a props called name to the const Name
Name will accept the props and will display the props anywhere , we need to call like props.name

For example we can also pass a props called age
Code and out put will be

```
import React from 'react';
import { Text, View } from 'react-native';

const Name = (props) => {
   return (
     <View>
       <Text>Hello, I am {props.name} and my age is {props.age}!</Text>
     </View>
   );
  }

  const Users = () => {
    return (
      <View>
        <Name name="Maru" age=”12” />
        <Name name="Jellylorum" age=”18” />
        <Name name="Spot" age=”15” />
      </View>
    );
  }

  export default Users;
```

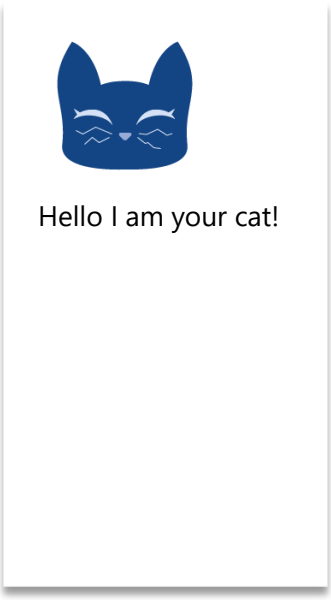
Output will be :

Hello, I am Maru and my age is 12!
Hello, I am and my age is 18!
Hello, I am and my age is 15!

Most of React Native's Core Components can be customized with props, too. For example, when using `Image`, you pass it a prop named `source` to define what image it shows:

```
1. import React from 'react';
2. import { Text, View, Image } from 'react-native';
3.
4. const CatApp = () => {
5.   return (
6.     <View>
7.       <Image
8.         source={{uri: "https://reactnative.dev/docs/assets/p_cat1.png"}}
9.         style={{width: 200, height: 200}}
10.        />
11.       <Text>Hello, I am your cat!</Text>
12.     </View>
13.   );
14. }
15.
16. export default CatApp;
17.
```

Output will be :



Demo phone

`Image` has many different props, including `style`, which accepts a JS object of design and layout related property-value pairs.

> Notice the double curly braces `{{ }}` surrounding `style`'s width and height. In JSX, JavaScript values are referenced with `{}`. This is handy if you are passing something other than a string as props, like an array or number: `<Name food={["rice", "kosun"]} age={2} />`. However, JS objects are ***also*** denoted with curly braces: `{width: 200, height: 200}`.

Therefore, to pass a JS object in JSX, you must wrap the object in **another pair** of curly braces: `{{width: 200, height: 200}}`

Image

```
<Image source={require('./assets/images/myimage.png')}/>
```

Here we are calling an image from asset with name myimage.png.
Place images anywhere you want , for good practice always place image in images folder [created folder]

if you have placed an image in a folder src/images/img.png then you should call image uri as
../images/img.png

You can use all style props…

For example:

Direct way …

```
<Image source{require...} style={{height :  100 , width : 100 , borderRadius : 10}}
```

Or with Stylesheet

```
1. import React from 'react'
2. import {Image, StyleSheet} from 'react-native'
3.
4. const MyApp = () => {
5. return (
6. <Image source={require('../images/myimage.png')} style={styles.myimage}/>
7. );
8. }
9. const styles = StyleSheet.create({
10.   myimage: {
11.  height: 100,
12.  width: 100,
13.  borderRadius: 10,
14.  padding: 20
15.  },
16.  Nextstyle: {
17.  ...
18.  }
19.  });
```

State

While you can think of props as arguments you use to configure how components render, **state** is like a component's personal data storage. State is useful for handling data that changes over time or that comes from user interaction. State gives your components memory!

As a general rule, use props to configure a component when it renders. Use state to keep track of any component data that you expect to change over time.

The following example takes place in a cat cafe where two hungry cats are waiting to be fed. Their hunger, which we expect to change over time (unlike their names), is stored as state. To feed the cats, press their buttons—which will update their state.

Class Component

```
1. import React, { Component } from "react";
2. import { Button, Text, View } from "react-native";
3.
4. class Cat extends Component {
5.   state = { isHungry: true };
6.
7.   render() {
8.     return (
9.       <View>
10.          <Text>
11.            I am {this.props.name}, and I am
12.            {this.state.isHungry ? " hungry" : " full"}!
13.          </Text>
14.          <Button
15.            onPress={() => {
16.              this.setState({ isHungry: false });
17.            }}
18.            disabled={!this.state.isHungry}
19.            title={
20.              this.state.isHungry ? "Pour me some milk, please!" : "Thank you!"
21.            }
22.          />
23.        </View>
24.      );
25.    }
26.  }
27.
28.  class Cafe extends Component {
29.    render() {
30.      return (
31.        <>
32.          <Cat name="Munkustrap" />
33.          <Cat name="Spot" />
34.        </>
35.      );
36.    }
37.  }
38.
39.  export default Cafe;
40.
```

As always with class components, you must import the `Component` class from React:

```
import React, { Component } from 'react';
```

In class components, state is stored in a state object:

```
export class Cat extends Component {
  state = { isHungry: true };
  //..
}
```

As with accessing props with `this.props`, you access this object inside your component with `this.state`:

```
<Text>
  I am {this.props.name}, and I am
  {this.state.isHungry ? ' hungry' : ' full'}!
</Text>
```

And you set individual values inside the state object by passing an object with the key value pair for state and its new value to `this.setState()`:

```
<Button
  onPress={() => {
    this.setState({ isHungry: false });
  }}
  // ..
/>
```

Do not change your component's state directly by assigning it a new value with `this.state.hunger = false`. Calling `this.setState()` allows React to track changes made to state that trigger rerendering. Setting state directly can break your app's reactivity!

When `this.state.isHungry` is false, the `Button`'s `disabled` prop is set to `false` and its `title` also changes:

```
<Button
  // ..
  disabled={!this.state.isHungry}
  title={
    this.state.isHungry
      ? 'Pour me some milk, please!'
      : 'Thank you!'
  }
/>
```

Finally, put your cats inside a `Cafe` component:

```
class Cafe extends Component {
  render() {
    return (
      <>
        <Cat name="Munkustrap" />
        <Cat name="Spot" />
      </>
    );
  }
}

export default Cafe;
```

See the `<>` and `</>` above? These bits of JSX are fragments. Adjacent JSX elements must be wrapped in an enclosing tag. Fragments let you do that without nesting an extra, unnecessary wrapping element like `View`.

Functional Component

You can add state to a component by calling React's `useState` Hook. A Hook is a kind of function that lets you "hook into" React features. For example, `useState` is a Hook that lets you add state to function components. You can learn more about other kinds of Hooks in the React documentation.

```
1. import React, { useState } from "react";
2. import { Button, Text, View } from "react-native";
3.
4. const Cat = (props) => {
5.   const [isHungry, setIsHungry] = useState(true);
6.
7.   return (
8.     <View>
9.       <Text>
10.          I am {props.name}, and I am {isHungry ? "hungry" : "full"}!
11.        </Text>
12.        <Button
13.          onPress={() => {
14.            setIsHungry(false);
15.          }}
16.          disabled={!isHungry}
17.          title={isHungry ? "Pour me some milk, please!" : "Thank you!"}
18.        />
19.      </View>
20.    );
21.  }
22.
23.  const Cafe = () => {
24.    return (
25.      <>
26.        <Cat name="Munkustrap" />
27.        <Cat name="Spot" />
28.      </>
29.    );
30.  }
31.
32.  export default Cafe;
33.
```

First, you will want to import `useState` from React like so:

```
import React, { useState } from 'react';
```

Then you declare the component's state by calling `useState` inside its function. In this example, `useState` creates an `isHungry` state variable:

```
const Cat = (props) => {
```

```
  const [isHungry, setIsHungry] = useState(true);
  // ...
};
```

You can use `useState` to track any kind of data: strings, numbers, Booleans, arrays, objects. For example, you can track the number of times a cat has been petted with `const [timesPetted, setTimesPetted] = useState(0)`!

Calling `useState` does two things:

- it creates a "state variable" with an initial value—in this case the state variable is `isHungry` and its initial value is `true`
- it creates a function to set that state variable's value—`setIsHungry`

It doesn't matter what names you use. But it can be handy to think of the pattern as `[<getter>, <setter>] = useState(<initialValue>)`.

Next you add the `Button` Core Component and give it an `onPress` prop:

```
<Button
  onPress={() => {
    setIsHungry(false);
  }}
  //..
/>
```

Now, when someone presses the button, `onPress` will fire, calling the `setIsHungry(false)`. This sets the state variable `isHungry` to `false`. When `isHungry` is false, the `Button`'s `disabled` prop is set to `true` and its `title` also changes:

```
<Button
  //..
  disabled={!isHungry}
  title={isHungry ? 'Pour me some milk, please!' : 'Thank you!'}
/>
```

You might've noticed that although `isHungry` is a const, it is seemingly reassignable! What is happening is when a state-setting function like `setIsHungry` is called, its component will re-render. In this case the `Cat` function will run again—and this time, `useState` will give us the next value of `isHungry`.

Finally, put your cats inside a `Cafe` component:

```
const Cafe = () => {
  return (
    <>
      <Cat name="Munkustrap" />
      <Cat name="Spot" />
    </>
  );
```

```
};
```

See the `<>` and `</>` above? These bits of JSX are fragments. Adjacent JSX elements must be wrapped in an enclosing tag. Fragments let you do that without nesting an extra, unnecessary wrapping element like `View`.

TextInput

`TextInput` is a Core Component that allows the user to enter text. It has an `onChangeText` prop that takes a function to be called every time the text changed, and an `onSubmitEditing` prop that takes a function to be called when the text is submitted.

For example, let's say that as the user types, you're translating their words into a different language. In this new language, every single word is written the same way: 🍕. So the sentence "Hello there Bob" would be translated as "🍕🍕🍕".

```
1. import React, { useState } from 'react';
2. import { Text, TextInput, View } from 'react-native';
3.
4. const PizzaTranslator = () => {
5.   const [text, setText] = useState('');
6.   return (
7.     <View style={{padding: 10}}>
8.       <TextInput
9.         style={{height: 40}}
10.          placeholder="Type here to translate!"
11.          onChangeText={text => setText(text)}
12.          defaultValue={text}
13.        />
14.        <Text style={{padding: 10, fontSize: 42}}>
15.          {text.split(' ').map((word) => word && '🍕').join(' ')}
16.        </Text>
17.      </View>
18.    );
19.  }
20.
21.  export default PizzaTranslator;
22.
```

run on expo

In this example, we store text in the state, because it changes over time.

There are a lot more things you might want to do with a text input. For example, you could validate the text inside while the user types. For more detailed examples, see the React docs on controlled components, or the reference docs for TextInput.

Text input is one of the ways the user interacts with the app. Next, let's look at another type of input and learn how to handle touches.

END OF PART ONE