Props in React Components

Topics covered:

- What are Props?
- How to use props?
    - Without Destructuring
        - Analyzing props
        - Extracting data
    - With Destructuring
    - Default Props

1. **Props**:
    - Props are similar to function arguments and are passed into the component as properties.
    - Props, an abbreviation for properties, transfer data between React components. The data flow within React's components is unidirectional (from parent to child only).
    - Props come in handy when you want the data flow in your app to be dynamic.

Example:

Add “taste” attribute in myElement:

```
const myElement = <Fruit taste='sweet'/>
```

The argument is given to the component as a props object:

Use the taste attribute in the component:

```
function Fruit(props) {
  return <h2>I am {props.taste} in taste!</h2>;
}
```

2. **Uses of Props:**
    - Props are another method for transferring data between components as parameters.
    - Props must be passed as an argument to your function to be used.
    - Comparable to calling your standard JavaScript functions with parameters

Without destructuring:

```
export default function FavColor(props) {
  const name = props.name;
  const color = props.color;
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
```

I'll now break down everything that just happened step by step:

Step 1: Introduce props as an argument.
This is what the function FavColor(props) in the code above does for us. As a result, you can use props in the component of your React app.

Step 2: Declare variables for props (s) **(Analyzing props)**

```
const name = props.name;
const color = props.color;
```

These variables, as you can see above, are distinct from conventional variables since their data relates to props.
If you do not wish to create variables for your props, you can just feed them into your template as follows:

```
      <h1>My name is {props.name}.</h1>
```

Step 3: In the JSX template, use a variable(s).
Now that you've defined your variables, you can put them wherever you want in your code.

```
return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
```

Step 4: Provide data on the App component's properties. **(Extracting Data)**
We've learned how to make our props, so the next step is to send data to them. We have previously imported the FavColor component, which is currently visible in the browser:

You can add default data to your properties so they don't appear empty when you declare them. You'll see how to do it in the final segment.

Recall that the current state of the App component:

```
function App() {
  return (
    <div className="App">
    <FavColor/>
    </div>
    );
  }

  export default App;
```

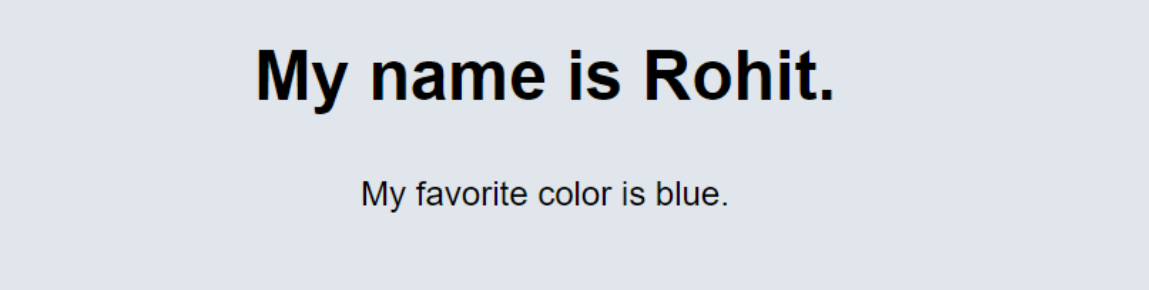
You must be wondering just to whom the data would be sent. You enter the data in the form of attributes to achieve this. It appears as follows:

```
function App() {
  return (
    <div className="App">
    <FavColor name='Rohit' color='blue'/>
    </div>
    );
  }

  export default App;
```

What has changed? From "FavColor" to <FavColor name="Rohit" color="Blue"/> in this example. Since those characteristics are linked to the properties defined in the FavColor component, this won't result in an error for you.

You should see the following in your browser:



NOTE: The variable name is not the prop itself. If I had made a variable like this: **const myPropName = props.name** and used it in my template like this: **<h1>My name is {myPropName}.</h1>**
And then if I did this: **<FavColor name="Rohit" color="Blue"/>**, the code will still work correctly. The name attribute is obtained from props.name rather than the variable name that contains the prop.
Using the logic given in the FavColor component, you can now create data dynamically for any component. You may declare an unlimited number of props.

With Destructuring:
Except for the method for declaring the props, the code for this section is identical to that of the previous section.
In the preceding part, we stated our props as follows:

```
const name = props.name;
const color = props.color;
```

However, we do not have to do this with destructuring. Simply follow these steps:

```
export function FavColor({ name, color }) {
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
```

The first line of code contains the difference. We destructured and supplied the variables as the function's argument rather than the props.
Other than that, nothing has changed.
You can also pass in functions and even data from objects, so don't think you're limited to just using single variables as your props data.

Default Props:
You can specify a default value if you don't want your props data to be empty when you create them. This is how to accomplish it:

```
function FavColor({ name, color }) {
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
FavColor.defaultProps={
    name: "User",
    color:"transparent"
}
export default FavColor;
```

We declared default values for our props right before the component was exported at the very end of the code. To start, we used the component's name and the built-in defaultProps that appear when you create a React app, separated by a dot and a period.

Now, instead of being blank whenever we import this component, those values will be the initial values. The default values are overridden when data is passed to the child component, as we did in the previous sections.

Output: