React props:

In React, "props" (short for properties) are a way to pass data from a parent component to a child component. Props are used to customize a component's behavior or appearance by providing it with input data.

When a component is created, it can receive props as an input, and these props can be accessed inside the component using the this.props object. For example, if a component receives a prop called "name", it can be accessed inside the component using this.props.name.

Props can be passed to a component using JSX syntax, like this:

```
<ComponentName propName={propValue} />
```

Here, propName is the name of the prop, and propValue is the value that will be passed to the component.

It's also possible to pass multiple props to a component like this:

```
<ComponentName prop1={value1} prop2={value2} />
```

Props are read-only and cannot be changed by the component that receives them, it is vital to remember this. Instead of using data, the component should utilize state if it needs to update the data.

To access the props passed to a child component, you can use the props object inside the component's render() method or the constructor() method.

For example, the following component receives a prop called name from its parent:

```
class ChildComponent extends React.Component {
  render() {
    return (
      <div>
        <p>Hello, {this.props.name}!</p>
      </div>
    );
  }
}
```

In this example, the child component uses the name prop to render a greeting.

It's important to note that, when consuming a prop, it's a good practice to define the type and default value of the prop using propTypes, this way you can be sure of the type of data you are working with and also make sure that if the prop is not passed, a default value is used.

When consuming a prop, it's also important to make sure that the component is designed to handle any possible value of the prop, so it can handle any possible edge case.

Validating Props in React:

In React, it's a good practice to validate props passed to a component to ensure that they are of the correct type and have the expected values. This can help to catch errors early and make the development process more efficient.

React provides a built-in way to validate props using the propTypes property. The propTypes property is an object that maps prop names to validation functions. For example:

```
class MyComponent extends React.Component {
  // ...
```

```
}

MyComponent.propTypes = {
  name: PropTypes.string,
  age: PropTypes.number,
  onClick: PropTypes.func
};
```

In this example, the MyComponent class has three props: name, age, and onClick. The propTypes object specifies that the name prop should be a string, the age prop should be a number, and the onClick prop should be a function.

You can also use isRequired to specify that a prop is required, and it will throw a warning if it's not provided.

```
MyComponent.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
  onClick: PropTypes.func.isRequired
};
```

You can also use custom validation functions for more complex validation. For example:

```
MyComponent.propTypes = {
  age: function(props, propName, componentName) {
    if (props[propName] < 0) {
      return new Error(
        “Invalid value entered”

      );
```

```
    }
  }
};
```

It's important to note that propTypes are only used in development mode, so it's safe to include them in production builds. React provides another property called defaultProps which can be used as fallback props when the props are not passed.

In summary, validating props in React is a good practice to ensure that the component receives the correct type and values of props. React provides a built-in way to validate props using the propTypes property. This can help to catch errors early and make the development process more efficient.

State in React:

- "State" in React refers to the data or variables that determine a component's behavior and render information to the user.
- In other words, state is what allows a component to "remember" certain information and change its behavior or appearance based on that information.
- State is managed by the component itself, and can be updated using the setState() method.
- State should be used with caution, as it can make a component difficult to reason about if it becomes too complex.

```
class MyComponent extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   message: "Hello World!"
  };
 }
```

```
  render() {
    return <h1>{this.state.message}</h1>;
  }
}
```

In this example, the MyComponent class has a state object with a single property message set to "Hello World!". In the render method, the state's message is displayed in an h1 element.

You can update the state by using setState method, it will re-render the component with updated state.

```
this.setState({ message: "New message"});
```

This will update the message in the state to "New message" and re-render the component with the updated message.

Combining State and Props in React:

In React, state and props are two ways to manage and pass data within a component and between components. State and props are used together to build dynamic and interactive user interfaces.

1. **State**: State is an object that holds the data specific to a component. State is managed by the component itself and can be updated by the component's methods. State is used to store data that changes within the component and can affect the component's behavior or appearance.

2. **Props**: Props are inputs passed from a parent component to a child component. Props are used to customize a component's behavior or

appearance by providing it with input data. Props are read-only and cannot be modified by the component that receives them.

When building a React application, state and props are often used together to create dynamic and interactive user interfaces. For example, a parent component might pass a prop to a child component, and the child component might use that prop along with its own state to render its UI.

Here's an example of a parent component passing a prop to a child component:

```
class ParentComponent extends React.Component {
  render() {
    return (
      <ChildComponent name={this.state.name} />
    );
  }
}

class ChildComponent extends React.Component {
  render() {
    return (
      <div>
        <p>Hello, {this.props.name}!</p>
      </div>
    );
  }
}
```

In this example, the parent component has a state variable called name and passes it as a prop to the child component. The child component receives the name prop and uses it to render a greeting.

It's also possible for a child component to use both props and state together. For example, a child component might receive a prop from its parent and use it to initialize its own state, like this:

```
class Component1 extends React.Component {
  constructor(props) {
    super(props);
    this.state = { value: props.initialValue };
  }

  render() {
    return (
      <div>
        <p>Value: {this.state.value}</p>
      </div>
    );
  }
}
```

In this example, the child component receives an initialValue prop and uses it to initialize its own value state variable. The child component then uses its value state variable to render its UI.

In summary, state and props are used together to build dynamic and interactive user interfaces in React. While state is used to store and manage data within a component, props are used to transmit data from parent to child components.

Handling and Changing State using 'useState':

useState is a Hook in React that helps us to add state to functional components. Prior to the introduction of Hooks, state could only be used in class components.

The current state and a function that can be used to update the state are the two items of the array that **useState** returns. The function will update the component's state and re-render it after receiving a new state value as an input.

Here's an example of how useState can be used in a functional component to create a simple counter:

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

function Counter() {
  const [countVar, setCountVar] = useState(0);

  return (
    <>
      <p>You clicked {countVar} times</p>
      <button onClick={() => setCountVar(countVar + 1)}>
        Click me
      </button>
    </>
  );
}
```

In this example, the useState Hook is called with an initial value of 0, and it returns an array with the current state count and a function setCountVar that can be used to update the state. The component's JSX includes a button that, when clicked, calls the setCountVar function with the new state value count + 1, which updates the component's state and re-renders the component with the new state value.

Note: useState Hook only works inside of the React functional components or your own Hooks.