Props Basics

 (https://github.com/learn-co-curriculum/react-hooks-props-basics)  (https://github.com/learn-co-curriculum/react-hooks-props-basics/issues/new)

Learning Goals

- Use props to make reusable component templates
- Understand how props are passed to a component

Introduction

In this lesson, we'll learn how we can turn our React components into dynamic templates using **props**.

We can define a React component as follows:

A component is a *function* that takes *props* as an argument and returns *JSX*.

As the building blocks of React applications, components are *dynamic*, in that they can describe a **template** of JSX in which variable data can be populated. To illustrate dynamic components, we will build an example blogging application. Our application will include the following components:

- `BlogContent` - contains the content of the blog post
- `Comment` - contains one user's comment
- `BlogPost` - the 'top level' React component, which is responsible for rendering both `BlogContent` and `Comment`

Making Components Dynamic

Time to put the **dynamic** aspect of components to use! Let's start with the `BlogContent` component:

```
function BlogContent(props) {
  return <div>{props.articleText}</div>;
```

```
}
```

You should see something new in the above code. Our function has a parameter defined called `props`. Also, inside the return statement, we have this funky syntax: `{props.articleText}`.

This line is telling React to place the value that `props.articleText` represents within the `<div>`. Ok, so where does `props.articleText` come from?

Passing Information

React allows us to pass units of information from a *parent* component down to a *child* component. We call these **props**, which we will dig more into in a later lesson. Let's see how we can pass information from `BlogPost` down to its child `BlogContent`:

```
function BlogPost() {
  return (
    <div>
      <BlogContent articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration." />
    </div>
  );
}
```

Above we see that when we render the `BlogContent` component, we also create a prop called `articleText` that we assign a value of "Dear Reader: Bjarne Stroustrup has the perfect lecture oration." This value is accessible from within the `BlogContent` component as `props.articleText`!

The syntax for creating props for a React component is the same as the syntax for writing attributes for an HTML tag. For example, to assign a `<div>` an id, we give it an attribute:

```
<div id="card">Hello!</div>
```

To assign a **prop** to a **component**, we use the same syntax:

```
function ParentComponent() {
  // passing prop to ChildComponent
  return <ChildComponent text="Hello!" number={2} />;
}

function ChildComponent(props) {
  // using the values of the text and number props
  return (
    <div>
      {props.text} {props.number}
    </div>
  );
}
```

But remember, this is JSX and not HTML!

One more thing about props: they can be any data type! In our example, we pass a string and a number as props. But we can also pass booleans, objects, functions, etc. as props!

Props

Let's expand a bit on props here. Taking a look at both of our components will give us a better understanding of how data can be passed from one component to another:

```
// BlogPost.js
// PARENT COMPONENT
function BlogPost() {
  return (
    <div>
      {/* BlogContent is being returned from BlogPost */}
      {/* Therefore, BlogContent is a child of BlogPost */}
      <BlogContent articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration." />
```

```
      </div>
    );
  }

  // BlogContent.js
  // CHILD COMPONENT
  function BlogContent(props) {
    return <div>{props.articleText}</div>;
  }
```

When one component *returns* another component, this creates a special relationship between these two components. The component being returned is the *child* component, and the component returning that child is the *parent* component.

The only way for a *parent* component to pass data to its *child* components is via **props**.

On this line:

```
// BlogPost.js
<BlogContent articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration." />
```

We are adding a **prop** of `articleText` to our `BlogContent` component.

If we add a `console.log` in the `BlogContent` component to inspect the props:

```
// BlogContent.js
function BlogContent(props) {
  console.log(props);
  return <div>{props.articleText}</div>;
}
```

We'll see an **object** with **key-value pairs** related to the data we passed down from the parent component!

```
// BlogContent.js
console.log(props);
```

```
// => { articleText: "Dear Reader: Bjarne Stroustrup has the perfect lecture oration." }
```

We can add as many additional props as we want by assigning them in the parent component:

```
<BlogContent
  articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration."
  isPublished={true}
  minutesToRead={1}
/>
```

Note: for props that are strings, we don't need to place curly braces around the values; for other data types (numbers, booleans, objects, etc), we need curly braces.

And all of these props will be added as **keys on the props object** in the child component:

```
// BlogContent.js
console.log(props);
/*
  {
    articleText: "Dear Reader: Bjarne Stroustrup has the perfect lecture oration.",
    isPublished: true,
    minutesToRead: 1
  }
*/
```

Having the ability to share this data between components by passing *props* down to a child component from a parent makes our components incredibly flexible! For example, here's how we could expand our `BlogContent` component based on those additional props:

```
function BlogContent(props) {
  console.log(props);

  if (!props.isPublished) {
    // hide unpublished content
```

```
    // return null means "don't display any DOM elements here"
    return null;
  } else {
    // show published content
    return (
      <div>
        <h1>{props.articleText}</h1>
        <p>{props.minutesToRead} minutes to read</p>
      </div>
    );
  }
}
```

Above, we are using ***conditional rendering*** (https://reactjs.org/docs/conditional-rendering.html) to only display the blog content if it is published, based on the `isPublished` prop.

Expanding our Application

We still need a `Comment` component that we can use for each comment in a `BlogPost`. The `Comment` component would look something like this:

```
function Comment(props) {
  return <div>{props.commentText}</div>;
}
```

This component, when used, will display content that is passed down to it, allowing us to pass different content to multiple `Comment` components. Let's add them in. Of course, with components being re-usable, we can make as many as we want:

```
function BlogPost() {
  return (
    <div>
      <BlogContent articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration." />
```

```
        <Comment />
        <Comment />
        <Comment />
      </div>
    );
  }
```

...and just as before, we can pass content data down to them:

```
function BlogPost() {
  return (
    <div>
      <BlogContent articleText="Dear Reader: Bjarne Stroustrup has the perfect lecture oration." />
      <Comment commentText="I agree with this statement. - Angela Merkel" />
      <Comment commentText="A universal truth. - Noam Chomsky" />
      <Comment commentText="Truth is singular. Its ‘versions’ are mistruths. - Sonmi-451" />
    </div>
  );
}
```

There is quite a bit going on here. Most notably, we are passing information from a parent component to many child components. Specifically, we are doing this by creating a prop called `commentText` to pass to each `Comment` component, which is then accessible in each instance of `Comment` as `props.commentText`. Let's expand the HTML this would ultimately render:

```
<div>
  <div>Dear Reader: Bjarne Stroustrup has the perfect lecture oration.</div>

  <div>I agree with this statement. - Angela Merkel</div>

  <div>A universal truth. - Noam Chomsky</div>
```

```
    <div>Truth is singular. Its 'versions' are mistruths - Sonmi-451</div>
  </div>
```

...but seeing is believing so let's look at this in technicolor! Following is an inspection of the *real live DOM elements* that React rendered when we blasted this code into a new application (classes, IDs, and minor CSS have been added for a better visual display):

Alright now! Take a moment. Stretch your limbs, make a sandwich, let the glorious paradigm sink in. Dynamic components are a core facet of React programming, and most of what we do as React programmers builds upon them.

Conclusion

While HTML elements are the basic building blocks of a website, (for example, a `<div>`), a React application usually consists of several React *components* combined together. Unlike simple HTML elements, React components are smarter and bigger. They allow you to do much more and incorporate logic into how content displays.

Components:

- are modular, reusable, and enable a 'templating' functionality
- help us organize our user interface's *logic* and *presentation*
- enable us to think about each piece in isolation, and apply structure to complex programs

Props:

- are passed from a *parent component* to a *child component*
- can be accessed in the *child components* via an *object* that is passed into our component function as an argument
- can hold any kind of data (strings, numbers, booleans, objects, even functions!)

Going forward we will expand on what we can do with components, how they fit into the larger React landscape, and what built-in functionality they come with.

Resources

- **Components and Props (https://reactjs.org/docs/components-and-props.html)**