Props and children

Previously, you learned that you could pass props to and within a component. But there is also a special prop known as **`props.children`**, which is automatically passed to every component. In this reading, you'll learn about **`props.children`** and what its purpose is.

To understand the concept of **`props.children`**, consider the following real-life situation: you have a couple of apples, and you have a couple of pears. You'd like to carry the apples some distance, so obviously, you'll use a bag.

It's not a "bag for apples". It's not a "bag for pairs". It's just a bag. Nothing about this bag makes it such that it needs to be referred to as a bag in which you'd only and always carry apples, nor a bag in which you'd only and always carry pears.

In a way, the bag "doesn't care" if it is used to carry apples or pears. Nothing about the bag changes. There are no changes in the bag's material, size, shape, or color - because it can handle apples or pears being carried inside of it, without issues.

Now, consider the following component:

```
function Apples(props) {
  return (
    <div className="promo-section">
        <div>
            <h2>These apples are: {props.color}</h2>
            </div>
            <div>
            <h3>There are {props.number} apples.</h3>
        </div>
    </div>
  )
}
export default Apples
```

There is also a **`Pears`** component:

```
function Pears(props) {
  return (
    <h2>I don't like pears, but my friend, {props.friend}, does</h2>
  )
}
```

Now, the question is this: Let's say you want to have a **`Bag`** component, which can be used to "carry" **`Apples`** or **`Pears`**. How would you do that?

This is where **`props.children`** comes in.

You can define a **`Bag`** component as follows:

```
function Bag(props) {
    const bag = {
        padding: "20px",
        border: "1px solid gray",
        background: "#fff",
        margin: "20px 0"
    }
    return (
        <div style={bag}>
            {props.children}
        </div>
    )
}
export default Bag
```

So, what this does in the `Bag` component is: it adds a wrapping `div` with a specific styling, and then gives it `props.children` as its content.

But what is this `props.children`?

Consider a very simple example:

```
<Example>
    Hello there
</Example>
```

The `Hello there` text is a child of the Example JSX element. The Example JSX Element above is an "invocation" of the **Example.js** file, which, in modern React, is usually a function component.

Now, did you know that this `Hello there` piece of text can be passed as a **named prop** when rendering the `Example` component?

Here's how that would look like:

```
<Example children="Hello there" />
```

Ok, so, there are two ways to do it. But this is just the beginning.

What if you, say, wanted to surround the `Hello there` text in an `h3` HTML element?

Obviously, in JSX, that is easily achievable:

```
<Example children={<h3>Hello there</h3>} />
```

What if the `<h3>Hello there</h3>` was a separate component, for example, named `Hello`?

In that case, you'd have to update the code like this:

```
<Example children={<Hello />} />
```

You could even make the `Hello` component more dynamic, by giving it its own prop:

```
<Example children={<Hello message="Hello there" />} />
```

So, given the **Bag**, **Apples**, and **Pears** examples from the beginning of this reading, armed with this new knowledge, how can you make it work?

Here's how you'd render the `Bag` component with the `Apples` component as its `props.children`:

```
<Bag children={<Apples color="yellow" number="5" />} />
```

And here's how you'd render the `Bag` component, wrapping the `Pears` component:

```
<Bag children={<Pears friend="Peter" />} />
```

While the above syntax might look a bit weird, it's important to understand what is happening "under the hood".

Effectively, the above syntax is the same as the two examples below.

```
<Bag>
    <Apples color="yellow" number="5" />
</Bag>

<Bag>
    <Pears friend="Peter" />
</Bag>
```

You can even have multiple levels of nested JSX elements, or a single JSX element having multiple children, such as, for example:

```
<Trunk>
    <Bag>
        <Apples color="yellow" number="5" />
        <Pears friend="Peter" />
    </Bag>
</Trunk>
```

So, in the above structure, there's a **`Trunk`** JSX element, inside of which is a single **`Bag`** JSX element, holding an **`Apples`** and a **`Pairs`** JSX element.

Before the end of this reading, consider this JSX element again:

```
<Bag>
    <Apples color="yellow" number="5" />
</Bag>
```

What is **Apples** to **Bag** in the above code?

In the above code, `Apples` is a prop of the `Bag` component. To explain further, the Bag component can wrap the Apples component, **or *any* other component**, because I used the **`{props.children}` syntax in the `Bag` component function declaration**. In other words, just like in the real world, when you take a bag to a grocery store, you can "wrap" a wide variety of groceries inside the bag, you can do the same thing in React: wrap a wide variety of components inside the **`Bag`** component, using the children prop to achieve this**.**

It's crucial to understand this when working with React.

Before the end of this reading, there's another important concept that you need to be aware of: *finding the right amount of modularization*. What does this mean? Imagine, for example, that you had a number of small bags, and that each bag could only carry a single apple or pear. You'd end up having to wrap each "apple" inside a "bag". That doesn't make much sense. You can think about components making your layouts modular in a similar way. You don't want to have an entire layout contained in a single component, because that would be very difficult to work with. On the flip side, if you made each HTML element in your layout a separate component, that would make it very hard to work with, although such layout would be modular. So it's all about moderation. You need to organize your layouts by splitting them into meaningful areas of the page, and then code those meaningful areas as separate components. that would constitute the right amount of modularity. To reinforce this point, It might help to think of it in terms of how a person would describe a website: there's a menu, a footer, the shopping cart, etc.

In conclusion, when you see a JSX element wrapping some other JSX element, you can easily understand that it's all just **`props.children`** in the background.