React Components, Elements, and Instances

December 18, 2015 by Dan Abramov

The difference between **components, their instances, and elements** confuses many React beginners. Why are there three different terms to refer to something that is painted on screen?

Managing the Instances

If you're new to React, you probably only worked with component classes and instances before. For example, you may declare a `Button` *component* by creating a class. When the app is running, you may have several *instances* of this component on screen, each with its own properties and local state. This is the traditional object-oriented UI programming. Why introduce *elements*?

In this traditional UI model, it is up to you to take care of creating and destroying child component instances. If a `Form` component wants to render a `Button` component, it needs to create its instance, and manually keep it up to date with any new information.

Code

```
class Form extends TraditionalObjectOrientedView {
  render() {
    // Read some data passed to the view
    const { isSubmitted, buttonText } = this.attrs;

    if (!isSubmitted && !this.button) {
      // Form is not yet submitted. Create the button!
      this.button = new Button({
        children: buttonText,
        color: 'blue'
      });
      this.el.appendChild(this.button.el);
    }

    if (this.button) {
      // The button is visible. Update its text!
      this.button.attrs.children = buttonText;
      this.button.render();
    }

    if (isSubmitted && this.button) {
      // Form was submitted. Destroy the button!
```

```
        this.el.removeChild(this.button.el);
        this.button.destroy();
      }

      if (isSubmitted && !this.message) {
        // Form was submitted. Show the success message!
        this.message = new Message({ text: 'Success!' });
        this.el.appendChild(this.message.el);
      }
    }
  }
```

This is pseudocode, but it is more or less what you end up with when you write composite UI code that behaves consistently in an object-oriented way using a library like Backbone.

Each component instance has to keep references to its DOM node and to the instances of the children components, and create, update, and destroy them when the time is right. The lines of code grow as the square of the number of possible states of the component, and the parents have direct access to their children component instances, making it hard to decouple them in the future.

So how is React different?

Elements Describe the Tree

In React, this is where the *elements* come to rescue. **An element is a plain object *describing* a component instance or DOM node and its desired properties.** It contains only information about the component type (for example, a `Button`), its properties (for example, its `color`), and any child elements inside it.

An element is not an actual instance. Rather, it is a way to tell React what you *want* to see on the screen. You can't call any methods on the element. It's just an immutable description object with two fields: `type: (string | ReactClass)` and `props: Object`[1].

DOM Elements

When an element's `type` is a string, it represents a DOM node with that tag name, and `props` correspond to its attributes. This is what React will render. For example:

Code

```
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

This element is just a way to represent the following HTML as a plain object:

Code

```
<button class='button button-blue'>
  <b>
    OK!
  </b>
</button>
```

Note how elements can be nested. By convention, when we want to create an element tree, we specify one or more child elements as the `children` prop of their containing element.

What's important is that both child and parent elements are *just descriptions and not the actual instances*. They don't refer to anything on the screen when you create them. You can create them and throw them away, and it won't matter much.

React elements are easy to traverse, don't need to be parsed, and of course they are much lighter than the actual DOM elements—they're just objects!

Component Elements

However, the `type` of an element can also be a function or a class corresponding to a React component:

Code

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}
```

This is the core idea of React.

An element describing a component is also an element, just like an element describing the DOM node. They can be nested and mixed with each other.

This feature lets you define a `DangerButton` component as a `Button` with a specific `color` property value without worrying about whether `Button` renders to a DOM `<button>`, a `<div>`, or something else entirely:

Code

```
const DangerButton = ({ children }) => ({
  type: Button,
  props: {
    color: 'red',
    children: children
```

```
    }
  });
```

You can mix and match DOM and component elements in a single element tree:

Code

```
const DeleteAccount = () => ({
  type: 'div',
  props: {
    children: [{
      type: 'p',
      props: {
        children: 'Are you sure?'
      }
    }, {
      type: DangerButton,
      props: {
        children: 'Yep'
      }
    }, {
      type: Button,
      props: {
        color: 'blue',
        children: 'Cancel'
      }
   }]
});
```

Or, if you prefer JSX:

Code

```
const DeleteAccount = () => (
  <div>
    <p>Are you sure?</p>
    <DangerButton>Yep</DangerButton>
    <Button color='blue'>Cancel</Button>
  </div>
);
```

This mix and matching helps keep components decoupled from each other, as they can express both *is-a* and *has-a* relationships exclusively through composition:

- `Button` is a DOM `<button>` with specific properties.
- `DangerButton` is a `Button` with specific properties.
- `DeleteAccount` contains a `Button` and a `DangerButton` inside a `<div>`.

Components Encapsulate Element Trees

When React sees an element with a function or class `type`, it knows to ask *that* component what element it renders to, given the corresponding `props`.

When it sees this element:

Code

```
{
  type: Button,
  props: {
    color: 'blue',
    children: 'OK!'
  }
}
```

React will ask `Button` what it renders to. The `Button` will return this element:

Code

```
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

React will repeat this process until it knows the underlying DOM tag elements for every component on the page.

React is like a child asking “what is Y” for every “X is Y” you explain to them until they figure out every little thing in the world.

Remember the `Form` example above? It can be written in React as follows[1]:

Code

```
const Form = ({ isSubmitted, buttonText }) => {
  if (isSubmitted) {
    // Form submitted! Return a message element.
    return {
      type: Message,
      props: {
        text: 'Success!'
      }
    };
  }

  // Form is still visible! Return a button element.
```

```
    return {
      type: Button,
      props: {
        children: buttonText,
        color: 'blue'
      }
    };
  };
```

That's it! For a React component, props are the input, and an element tree is the output.

The returned element tree can contain both elements describing DOM nodes, and elements describing other components. This lets you compose independent parts of UI without relying on their internal DOM structure.

We let React create, update, and destroy instances. We *describe* them with elements we return from the components, and React takes care of managing the instances.

Components Can Be Classes or Functions

In the code above, `Form`, `Message`, and `Button` are React components. They can either be written as functions, like above, or as classes descending from `React.Component`. These three ways to declare a component are mostly equivalent:

Code

```
// 1) As a function of props
const Button = ({ children, color }) => ({
  type: 'button',
  props: {
    className: 'button button-' + color,
    children: {
      type: 'b',
      props: {
        children: children
      }
    }
  }
});

// 2) Using the React.createClass() factory
const Button = React.createClass({
  render() {
    const { children, color } = this.props;
    return {
      type: 'button',
      props: {
        className: 'button button-' + color,
        children: {
          type: 'b',
          props: {
            children: children
```

```
            }
          }
        }
      };
    }
  });

  // 3) As an ES6 class descending from React.Component
  class Button extends React.Component {
    render() {
      const { children, color } = this.props;
      return {
        type: 'button',
        props: {
          className: 'button button-' + color,
          children: {
            type: 'b',
            props: {
              children: children
            }
          }
        }
      };
    }
  }
```

When a component is defined as a class, it is a little bit more powerful than a functional component. It can store some local state and perform custom logic when the corresponding DOM node is created or destroyed.

A functional component is less powerful but is simpler, and acts like a class component with just a single `render()` method. Unless you need features available only in a class, we encourage you to use functional components instead.

However, whether functions or classes, fundamentally they are all components to React. They take the props as their input, and return the elements as their output.

Top-Down Reconciliation

When you call:

Code

```
ReactDOM.render({
  type: Form,
  props: {
    isSubmitted: false,
    buttonText: 'OK!'
  }
}, document.getElementById('root'));
```

React will ask the `Form` component what element tree it returns, given those `props`. It will gradually "refine" its understanding of your component tree in terms of simpler primitives:

Code

```
// React: You told me this...
{
  type: Form,
  props: {
    isSubmitted: false,
    buttonText: 'OK!'
  }
}

// React: ...And Form told me this...
{
  type: Button,
  props: {
    children: 'OK!',
    color: 'blue'
  }
}

// React: ...and Button told me this! I guess I'm done.
{
  type: 'button',
  props: {
    className: 'button button-blue',
    children: {
      type: 'b',
      props: {
        children: 'OK!'
      }
    }
  }
}
```

This is a part of the process that React calls reconciliation which starts when you call `ReactDOM.render()` or `setState()`. By the end of the reconciliation, React knows the result DOM tree, and a renderer like `react-dom` or `react-native` applies the minimal set of changes necessary to update the DOM nodes (or the platform-specific views in case of React Native).

This gradual refining process is also the reason React apps are easy to optimize. If some parts of your component tree become too large for React to visit efficiently, you can tell it to skip this "refining" and diffing certain parts of the tree if the relevant props have not changed. It is very fast to calculate whether the props have changed if they are immutable, so React and immutability work great together, and can provide great optimizations with the minimal effort.

You might have noticed that this blog entry talks a lot about components and elements, and not so much about the instances. The truth is, instances have much less importance in React than in most object-oriented UI frameworks.

Only components declared as classes have instances, and you never create them directly: React does that for you. While mechanisms for a parent component instance to access a child component instance exist, they are only used for imperative actions (such as setting focus on a field), and should generally be avoided.

React takes care of creating an instance for every class component, so you can write components in an object-oriented way with methods and local state, but other than that, instances are not very important in the React's programming model and are managed by React itself.

Summary

An *element* is a plain object describing what you want to appear on the screen in terms of the DOM nodes or other components. Elements can contain other elements in their props. Creating a React element is cheap. Once an element is created, it is never mutated.

A *component* can be declared in several different ways. It can be a class with a `render()` method. Alternatively, in simple cases, it can be defined as a function. In either case, it takes props as an input, and returns an element tree as the output.

When a component receives some props as an input, it is because a particular parent component returned an element with its `type` and these props. This is why people say that the props flows one way in React: from parents to children.

An *instance* is what you refer to as `this` in the component class you write. It is useful for storing local state and reacting to the lifecycle events.

Functional components don't have instances at all. Class components have instances, but you never need to create a component instance directly—React takes care of this.

Finally, to create elements, use `React.createElement()`, JSX, or an element factory helper. Don't write elements as plain objects in the real code—just know that they are plain objects under the hood.

Further Reading

- Introducing React Elements
- Streamlining React Elements
- React (Virtual) DOM Terminology

---

1. All React elements require an additional `$$typeof: Symbol.for('react.element')` field declared on the object for security reasons. It is omitted in the examples above. This blog entry uses inline objects for elements to give you an idea of what's happening underneath but the code won't run as is unless you either add `$$typeof` to the elements, or change the code to use `React.createElement()` or JSX. ↩

RECENT POSTS

What's New in Create React App
React v15.5.0
React v15.4.0
Our First 50,000 Stars