ES6 JavaScript - What You Need To Know

Destructuring assignment

- `let {a, b} = o` assigns Object `o`'s `a` and `b` properties to variables `a`, `b`
- `let [a, b] = arr` assigns first/second items of Array `arr` to variables `a` and `b`
- Assign defaults with `=`, e.g. `let {max = 5} = options`
- Destructuring can be performed on function arguments.
  `function fn({options = {}, flag = true}) { ... }`

`for .. of` loops

- Works on Iterables, including `Array`, `Map`, `Set` and generators.
- Does not work with objects.
- Use with destructuring assignment and `let`
  `for (let [key, value] of map) { ... }`

`let` / `const`

- Make variables scoped by block, not function
- Use in place of `var`
- `const` prevents re-assignment, but does not make assigned objects immutable

=> arrow functions

- *`argument => returned expression`*
- `this` inside function is equal to `this` where it was defined
  `function() { ... }.bind(this)`
- *returned expression* can be a block
  `x => { console.log('doubling'); return x*2 }`
- Use parentheses for more than one argument
  `(min, x, max) => Math.max(min, Math.min(x, max))`
- Use parentheses when argument is being destructured
  `({x, y}) => Math.sqrt(x*x, y*y)`

Backtick (``) Template Strings

- Interpolate with `${`*`expression`*`}`
  `` `Token token=${identity.get('accessToken')}` ``
- Can be split over multiple lines

... (spread operators / rest parameters)

- In functions parameters, creates an array of remaining arguments
  `function classes(...args) { return args.join(' ') }`
- In function arguments, expands array to actual parameters
  `console.log(...args)`
- Similar to `Function.prototype.apply`, but doesn't modify `this`

New `Array` Methods

- `arr.find(callback[, thisArg])`
  return the first item which when passed to `callback`, produces a truthy value
- `arr.findIndex(callback[, thisArg])`
  return the index of the first item which when passed to `callback` produces a truthy value
- `arr.fill(value[, start = 0[, end = this.length]])`
  fills all the elements of an array from a `start` index to an `end` index
- `arr.copyWithin(target, start[, end = this.length])`
  copies the sequence of items within the array to the position starting with `target`, taken from the position starting with `start`

New Built-in Classes

- `Map` - Map keys to values. Unlike objects, keys don't have to be strings
- `Set` - Store a set, where each stored value is unique
- `Symbol` - Use to make private object/class properties
- `Promise` - Manage callbacks for an event which will occur in the future

JavaScript Promises - What You Need To Know

The four functions you need to know

1. **`new Promise(fn)`**
   - `fn` takes two arguments: `resolve` and `reject`
   - `resolve` and `reject` are both functions which can be called with one argument
   - Returned promise will be rejected if an exception is thrown in the passed in function
2. **`promise.then(onResolve, onReject)`**
   - Returns a promise
   - Returned promise resolves to value returned from handler
   - Chain by returning a promise from `onResolve` or `onReject`
   - Returned promise will be rejected if an exception is thrown in a handler
   - Use `Promise.reject` to return a rejected promise from `onReject`
   - Make sure to follow by `promise.catch`
3. **`promise.catch(onReject)`**
   - Returns a promise
   - Equivalent to `promise.then(null, onReject)`
4. **`Promise.all([promise1, promise2, ...])`**
   - Returns a promise
   - When all arguments resolve, returned promise resolves to an array of all results
   - When any arguments are rejected, returned promise is immediately rejected with the same value
   - Useful for managing doing multiple things concurrently

Packages

- es6-promise - Polyfill older browsers
- bluebird - Get extra promise methods
- promisify-node - Promisify callback-accepting functions (npm)

Extra Reading

- Are JavaScript Promises swallowing your errors?
- Promises at MDN
- Promise browser support at Can I Use

The two functions you should know

- **`Promise.resolve(value)`**
  - Returns a promise which resolves to `value`
  - If `value` is a promise, just returns `value`
- **`Promise.reject(value)`**
  - Returns a rejected promise with the value `value`
  - Useful while processing errors with `promise.catch`

Patterns

- **Promisify a callback-accepting function `fn`**

  Assume callback passed to `fn` takes two arguments: `callback(error, data)`, where `error` is `null` if successful, and `data` is null if unsuccessful.

```
new Promise(function(resolve, reject) {
  fn(function(error, data) {
    if (error) {
      reject(error);
    }
    else {
      resolve(data);
    }
  });
});
```

- **Catch exceptions thrown in `then` handlers**

```
promise
  .then(function() { ... })
  .catch(function(err) {
    console.log(err.stack);
  });
```

THE ESSENTIALS

1. **`React.createElement(type, props, children)`**

   Create a `ReactElement` with the given component class, `props` and `children`.

   ```
   var link = React.createElement('a', {href: '#'}, "Save")
   var nav = React.createElement(MyNav, {flat: true}, link)
   ```

2. **`React.cloneElement(element, props, children)`**

   Create a new `ReactElement`, merging in new `props` and `children`.

3. **`ReactDOM.render(element, domNode)`**

   Take a `ReactElement`, and render it to a DOM node. E.g.

   ```
   ReactDOM.render(
     React.createElement('div'),
     document.getElementById('container')
   )
   ```

4. **`ReactDOM.findDOMNode(element)`**

   Return the DOM node corresponding to the given element (after `render`).

SPECIAL PROPS

`children` is automatically added to `this.props` by `React.createElement`.
`className` corresponds to the HTML `class` attribute.
`htmlFor` corresponds to the HTML `for` attribute.
`key` uniquely identifies a `ReactElement`. Used with elements in arrays.
`ref` accepts a callback function which will be called:
1. with the component instance or DOM node on mount.
2. with `null` on unmount and when the passed in function changes.

`style` accepts an *object* of styles, instead of a string.

PROPTYPES

Available under `React.PropTypes`. Optionally append `.isRequired`.

`any` `array` `bool` `element` `func`
`node` `number` `object` `string`

`instanceOf(constructor)`

`oneOf(['News', 'Photos'])`

`oneOfType([propType, propType])`

CLASS COMPONENTS

```
var MyComponent = React.createClass({
  displayName: 'MyComponent',

  /* ... options and lifecycle methods ... */

  render: function() {
    return React.createElement( /* ... */ )
  },
})
```

Options

`propTypes`	`object` mapping prop names to types
`getDefaultProps`	`function()` returning `object`
`getInitialState`	`function()` returning `object`

Lifecycle Methods

`componentWillMount`	`function()`
`componentDidMount`	`function()`
`componentWillReceiveProps`	`function(nextProps)`
`shouldComponentUpdate`	`function(nextProps, nextState)` -> `bool`
`componentWillUpdate`	`function(nextProps, nextState)`
`componentDidUpdate`	`function(prevProps, prevState)`
`componentWillUnmount`	`function()`

COMPONENT INSTANCES

- Accessible as `this` within class components
- Stateless functional components do not have component instances.
- Serve as the object passed to `ref` callbacks
- One component instance may persist over multiple equivalent `ReactElement`s.

Properties

`props` contains any props passed to `React.createElement`
`state` contains state set by `setState` and `getInitialState`

Methods

1. `setState(changes)` applies the given changes to `this.state` and re-renders
2. `forceUpdate()` immediately re-renders the component to the DOM

COMPONENTS ARE CLASSES

```
export class MyComponent extends Component {
  componentWillMount() {
    // ...
  }

  render() {
    return <div>Hello World</div>
  }
}
```

STATIC PROPERTIES

```
export class MyComponent extends Component {
  static propTypes = {
    // ...
  }
  static defaultProps = {
    // ...
  }
}
```

INITIAL STATE

```
export class MyComponent extends Component {
  state = {
    disabled: this.props.disabled,
  }
}
```

CONSTRUCTORS

Replace `componentWillMount`:

```
export class MyComponent extends Component {
  constructor(props) {
    super(props)
    // Do stuff
  }
}
```


BOUND METHODS

Bind methods to your component instance when used as handlers:

```
export class MyComponent extends Component {
  onMouseEnter = event => {
    this.setState({hovering: true})
  }
}
```

IMPORT

JSX assumes a `React` object is available, so make sure to import it:

```
import React, { Component, PropTypes } from 'react'
```

DESTRUCTURING

Especially useful with stateless function components:

```
export const Commas = ({items, ...otherProps}) =>
  <div {...otherProps}>{items.join(', ')}</div>
```

TEMPLATE LITERALS

Use to make dynamic class:

```
<input className={`Control-${this.state}`} />
```

And to sweeten your object literals with dynamic property names:

```
this.setState({
  [`${inputName}Value`]: e.target.value,
});
```

CLASS DECORATORS

Use in place of mixins:

```
@something(options)
class MyComponent extends Component {}

// Desugars to

let MyComponent = something(options)(
  class MyComponent extends Components {}
)
```

JSX

`<tags>` become `React.createElement`

Use `<lowercase />` tags for DOM elements:

```jsx
<div />
```

```js
React.createElement('div')
```

And use `<Capitalized />` tags for custom elements:

```jsx
<Modal />
```

```js
React.createElement(Modal)
```

`attributes` are `props`

Use `""` quotes when your props are strings:

```jsx
<Modal title="Edit" />
```

```js
React.createElement(Modal, {
  title: "Edit"
})
```

And use `{}` braces when your props are literals or variables:

```jsx
<Modal
  title={`Edit ${name}`}
  onClose={this.handleClose}
/>
```

```js
React.createElement(Modal, {
  title: `Edit ${name}`,
  onClose: this.handleClose,
})
```

`{...object}` becomes `Object.assign`

Use it in place of `Object.assign`

```jsx
<div
  className='default'
  {...this.props}
/>
```

```js
React.createElement('div',
  Object.assign(
    { className: 'default' },
    this.props
  )
)
```

`<tag> children` become `props.children`.

They can be text:

```jsx
<p>
  What good is a phone call...
</p>
```

```js
React.createElement('p', {},
  "What good is a phone call..."
)
```

They can be elements:

```jsx
<Modal>
  <h1>
    And then there will be cake
  </h1>
</Modal>
```

```js
React.createElement(Modal, {},
  React.createElement('h1', {},
    "And then there will be cake"
  )
)
```

Or they can be a mix of both:

```jsx
<p>
  I'm sorry <em>Dave</em>
</p>
```

```js
React.createElement('p', {},
  "I'm sorry ",
  React.createElement('em', {}, "Dave")
)
```

Interpolate `children` using `{}`

You can interpolate text:

```jsx
<p>
  I'm sorry {name}
</p>
```

```js
React.createElement('p', {},
  "I'm sorry ", name
)
```

Or even arrays:

```jsx
<ol>
  {steps.map(step =>
    <li key={step.name}>
      {step.content}
    </li>
  )}
</ol>
```

```js
React.createElement('ol', {},
  steps.map((step, i) =>
    React.createElement(
      'li', {key: i}, step
    )
  )
)
```