```
/*
*****************************************************************************************
 * REACT.JS CHEATSHEET
 * DOCUMENTATION: https://reactjs.org/docs/
 * FILE STRUCTURE: https://reactjs.org/docs/faq-structure.html
 *
*****************************************************************************************
**** */


```
npm install --save react       // declarative and flexible JavaScript library for
building UI
npm install --save react-dom   // serves as the entry point of the DOM-related
rendering paths
npm install --save prop-types  // runtime type checking for React props and similar
objects
```

// notes: don't forget the command lines

/*
****
 * REACT
 * https://reactjs.org/docs/react-api.html
 *
****
**** */

// Create and return a new React element of the given type.
// Code written with JSX will be converted to use React.createElement().
// You will not typically invoke React.createElement() directly if you are using JSX.
React.createElement(
  type,
  [props],
  [...children]
)

// Clone and return a new React element using element as the starting point.
// The resulting element will have the original element’s props with the new props
merged in shallowly.
React.cloneElement(
  element,
  [props],
  [...children]
)

// Verifies the object is a React element. Returns true or false.
React.isValidElement(object)
React.Children // provides utilities for dealing with the this.props.children opaque
data structure.

// Invokes a function on every immediate child contained within children with this set
to thisArg.
React.Children.map(children, function[(thisArg)])

// Like React.Children.map() but does not return an array.
React.Children.forEach(children, function[(thisArg)])

// Returns the total number of components in children,
// equal to the number of times that a callback passed to map or forEach would be
invoked.
React.Children.count(children)

// Verifies that children has only one child (a React element) and returns it.
// Otherwise this method throws an error.
React.Children.only(children)

// Returns the children opaque data structure as a flat array with keys assigned to
each child.
// Useful if you want to manipulate collections of children in your render methods,
// especially if you want to reorder or slice this.props.children before passing it
down.
React.Children.toArray(children)

// The React.Fragment component lets you return multiple elements in a render() method
without creating an additional DOM element
// You can also use it with the shorthand <></> syntax.
React.Fragment

/*
****
****
 * REACT.COMPONENT
 * React.Component is an abstract base class, so it rarely makes sense to refer to
React.Component
 * directly. Instead, you will typically subclass it, and define at least a render()
method.
 * https://reactjs.org/docs/react-component.html
 *
****
**** */

class Component extends React.Component {
  // Will be called before it is mounted
  constructor(props) {
    // Call this method before any other statement
    // or this.props will be undefined in the constructor
    super(props);

    // The constructor is also often used to bind event handlers to the class instance.
    // Binding makes sure the method has access to component attributes like this.props
and this.state
    this.method = this.method.bind(this);

    // The constructor is the right place to initialize state.
    this.state = {
      active: true,

      // In rare cases, it’s okay to initialize state based on props.
      // This effectively “forks” the props and sets the state with the initial props.
      // If you “fork” props by using them for state, you might also want to implement
componentWillReceiveProps(nextProps)
      // to keep the state up-to-date with them. But lifting state up is often easier
and less bug-prone.
      color: props.initialColor
    };
  }

  // Enqueues changes to the component state and
  // tells React that this component and its children need to be re-rendered with the
updated state.
  // setState() does not always immediately update the component. It may batch or defer
the update until later.
  // This makes reading this.state right after calling setState() a potential pitfall.
  // Instead, use componentDidUpdate or a setState callback.
  // You may optionally pass an object as the first argument to setState() instead of a
function.
  setState(updater[, callback]) { }

  // Invoked just before mounting occurs (before render())
  // This is the only lifecycle hook called on server rendering.
  componentWillMount() { }

  // Invoked immediately after a component is mounted.
  // Initialization that requires DOM nodes should go here.
  // If you need to load data from a remote endpoint, this is a good place to
instantiate the network request.
  // This method is a good place to set up any subscriptions. If you do that, don’t
forget to unsubscribe in componentWillUnmount().
  componentDidMount() { }

  // Invoked before a mounted component receives new props.
  // If you need to update the state in response to prop changes (for example, to reset
it),
  // you may compare this.props and nextProps and perform state transitions using
this.setState() in this method.
  componentWillReceiveProps(nextProps) { }

  // Let React know if a component’s output is not affected by the current change in
state or props.
  // The default behavior is to re-render on every state change, and in the vast
majority of cases you should rely on the default behavior.
  // shouldComponentUpdate() is invoked before rendering when new props or state are
being received. Defaults to true.
```

```
  // This method is not called for the initial render or when forceUpdate() is used.
  // Returning false does not prevent child components from re-rendering when their
state changes.
  shouldComponentUpdate(nextProps, nextState) { }

  // Invoked just before rendering when new props or state are being received.
  // Use this as an opportunity to perform preparation before an update occurs. This
method is not called for the initial render.
  // Note that you cannot call this.setState() here; nor should you do anything else
  // (e.g. dispatch a Redux action) that would trigger an update to a React component
before componentWillUpdate() returns.
  // If you need to update state in response to props changes, use
componentWillReceiveProps() instead.
  componentWillUpdate(nextProps, nextState) { }

  // Invoked immediately after updating occurs. This method is not called for the
initial render.
  // Use this as an opportunity to operate on the DOM when the component has been
updated.
  // This is also a good place to do network requests as long as you compare the
current props to previous props (e.g. a network request may not be necessary if the
props have not changed).
  componentDidUpdate(prevProps, prevState) { }

  // Invoked immediately before a component is unmounted and destroyed.
  // Perform any necessary cleanup in this method, such as invalidating timers,
canceling network requests,
  // or cleaning up any subscriptions that were created in componentDidMount().
  componentWillUnmount() { }

  // Error boundaries are React components that catch JavaScript errors anywhere in
their child component tree,
  // log those errors, and display a fallback UI instead of the component tree that
crashed.
  // Error boundaries catch errors during rendering, in lifecycle methods, and in
constructors of the whole tree below them.
  componentDidCatch() { }

  // This method is required.
  // It should be pure, meaning that it does not modify component state,
  // it returns the same result each time it's invoked, and
  // it does not directly interact with the browser (use lifecycle methods for this)
  // It must return one of the following types: react elements, string and numbers,
portals, null or booleans.
  render() {
    // Contains the props that were defined by the caller of this component.
    console.log(this.props);

    // Contains data specific to this component that may change over time.
    // The state is user-defined, and it should be a plain JavaScript object.
    // If you don't use it in render(), it shouldn't be in the state.
    // For example, you can put timer IDs directly on the instance.
    // Never mutate this.state directly, as calling setState() afterwards may replace
the mutation you made.
```

```
    // Treat this.state as if it were immutable.
    console.log(this.state);

    return (
      <div>
        {/* Comment goes here */}
        Hello, {this.props.name}!
      </div>
    );
  }
}

// Can be defined as a property on the component class itself, to set the default props
for the class.
// This is used for undefined props, but not for null props.
Component.defaultProps = {
  color: 'blue'
};

component = new Component();

// By default, when your component’s state or props change, your component will re-
render.
// If your render() method depends on some other data, you can tell React that the
component needs re-rendering by calling forceUpdate().
// Normally you should try to avoid all uses of forceUpdate() and only read from
this.props and this.state in render().
component.forceUpdate(callback)


/*
*****************************************************************************************
****
 * REACT.DOM
 * The react-dom package provides DOM-specific methods that can be used at the top
level of
 * your app and as an escape hatch to get outside of the React model if you need to.
 * Most of your components should not need to use this module.
 * https://reactjs.org/docs/react-dom.html
 *
*****************************************************************************************
**** */


// Render a React element into the DOM in the supplied container and return a reference
// to the component (or returns null for stateless components).
ReactDOM.render(element, container[, callback])

// Same as render(), but is used to hydrate a container whose HTML contents were
rendered
// by ReactDOMServer. React will attempt to attach event listeners to the existing
markup.
ReactDOM.hydrate(element, container[, callback])
```

```
// Remove a mounted React component from the DOM and clean up its event handlers and
state.
// If no component was mounted in the container, calling this function does nothing.
// Returns true if a component was unmounted and false if there was no component to
unmount.
ReactDOM.unmountComponentAtNode(container)

// If this component has been mounted into the DOM, this returns the corresponding
native browser
// DOM element. This method is useful for reading values out of the DOM, such as form
field values
// and performing DOM measurements. In most cases, you can attach a ref to the DOM node
and avoid
// using findDOMNode at all.
ReactDOM.findDOMNode(component)

// Creates a portal. Portals provide a way to render children into a DOM node that
exists outside
// the hierarchy of the DOM component.
ReactDOM.createPortal(child, container)


/*
*****************************************************************************************
****
 * REACTDOMSERVER
 * The ReactDOMServer object enables you to render components to static markup.
 * https://reactjs.org/docs/react-dom.html
 *
*****************************************************************************************
**** */


// Render a React element to its initial HTML. React will return an HTML string.
// You can use this method to generate HTML on the server and send the markup down on
the initial
// request for faster page loads and to allow search engines to crawl your pages for
SEO purposes.
ReactDOMServer.renderToString(element)

// Similar to renderToString, except this doesn’t create extra DOM attributes that
React uses
// internally, such as data-reactroot. This is useful if you want to use React as a
simple static
// page generator, as stripping away the extra attributes can save some bytes.
ReactDOMServer.renderToStaticMarkup(element)

// Render a React element to its initial HTML. Returns a Readable stream that outputs
an HTML string.
// The HTML output by this stream is exactly equal to what
ReactDOMServer.renderToString would return.
// You can use this method to generate HTML on the server and send the markup down on
the initial
// request for faster page loads and to allow search engines to crawl your pages for
```

```
SEO purposes.
ReactDOMServer.renderToNodeStream(element)

// Similar to renderToNodeStream, except this doesn’t create extra DOM attributes that
React uses
// internally, such as data-reactroot. This is useful if you want to use React as a
simple static
// page generator, as stripping away the extra attributes can save some bytes.
ReactDOMServer.renderToStaticNodeStream(element)


/*
*****************************************************************************************
****
 * TYPECHECKING WITH PROPTYPES
 * https://reactjs.org/docs/typechecking-with-proptypes.html
 *
*****************************************************************************************
**** */


import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,

  // Anything that can be rendered: numbers, strings, elements or an array
  // (or fragment) containing these types.
  optionalNode: PropTypes.node,

  // A React element.
  optionalElement: PropTypes.element,

  // You can also declare that a prop is an instance of a class. This uses
  // JS's instanceof operator.
  optionalMessage: PropTypes.instanceOf(Message),

  // You can ensure that your prop is limited to specific values by treating
  // it as an enum.
  optionalEnum: PropTypes.oneOf(['News', 'Photos']),

  // An object that could be one of many types
  optionalUnion: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
    PropTypes.instanceOf(Message)
  ]),

  // An array of a certain type
  optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

  // An object with property values of a certain type
  optionalObjectOf: PropTypes.objectOf(PropTypes.number),

  // An object taking on a particular shape
  optionalObjectWithShape: PropTypes.shape({
    color: PropTypes.string,
    fontSize: PropTypes.number
  }),

  // You can chain any of the above with `isRequired` to make sure a warning
  // is shown if the prop isn't provided.
  requiredFunc: PropTypes.func.isRequired,

  // A value of any data type
  requiredAny: PropTypes.any.isRequired,

  // You can also specify a custom validator. It should return an Error
  // object if the validation fails. Don't `console.warn` or throw, as this
  // won't work inside `oneOfType`.
  customProp: function(props, propName, componentName) {
    if (!/matchme/.test(props[propName])) {
      return new Error(
        'Invalid prop `' + propName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  },

  // You can also supply a custom validator to `arrayOf` and `objectOf`.
  // It should return an Error object if the validation fails. The validator
  // will be called for each key in the array or object. The first two
  // arguments of the validator are the array or object itself, and the
  // current item's key.
  customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location,
propFullName) {
    if (!/matchme/.test(propValue[key])) {
      return new Error(
        'Invalid prop `' + propFullName + '` supplied to' +
        ' `' + componentName + '`. Validation failed.'
      );
    }
  })
};
```