JSX Cheat Sheet

How to write better JSX

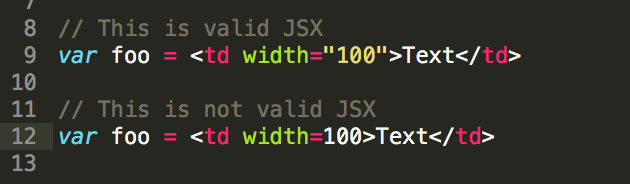
### What is JSX?

JSX is an XML-like syntax for writing legible ReactJS component code. It is *not* interpreted by the browser! Instead, Babel will interpret your JSX and convert it into valid Javascript.

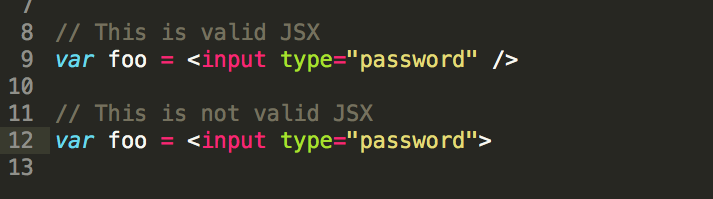
### First, some gotchas

While JSX looks almost identical to HTML, it has some minor differences that are easy to remember:

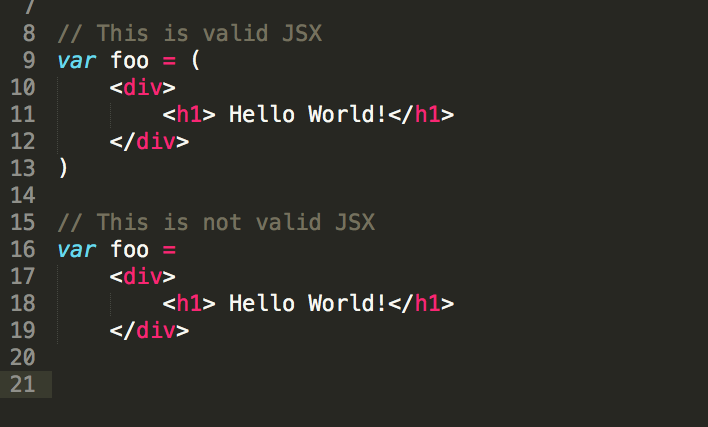
# You must use “” when defining attributes

****

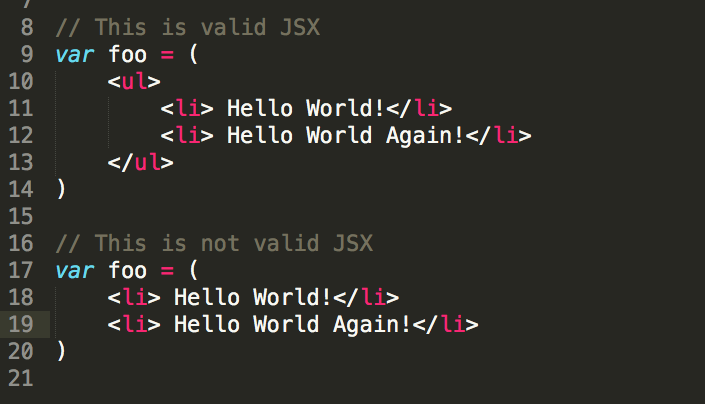
# Self-closing tags must be terminated with /

****

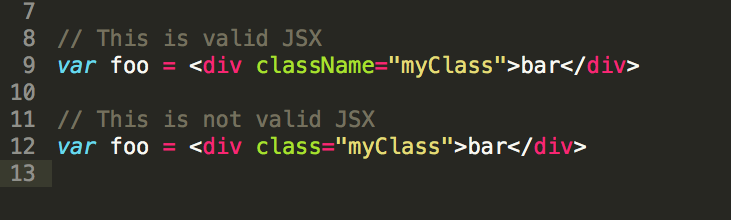
# Multi-line JSX must be wrapped with ( )



# A JSX statement must have only one outer element

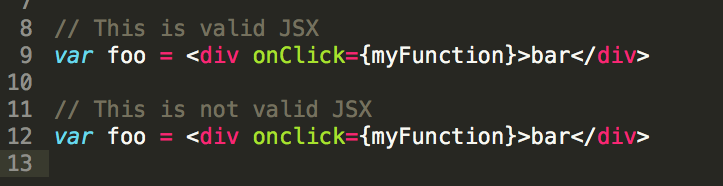
**

# You can’t use any Javascript reserved words

**

Note: To add a CSS class to your elements, use className instead of class. Chances are that this will trip you up a few times while you get used to JSX!

# Javascript event listeners should be camel-cased (instead of lowercase)



Note: If those curly braces confuse you, don’t worry! They’re explained below.

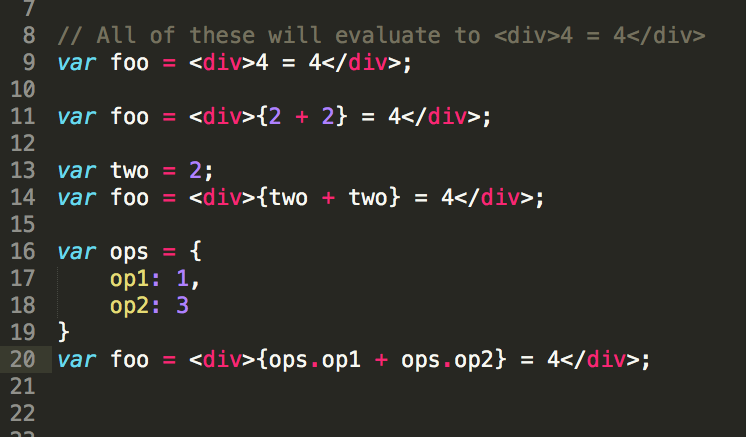
### 

### JS in your JSX

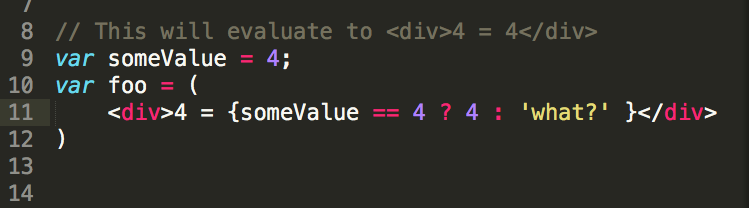
Now that the gotchas are out of the way, we can dive into the cool parts of JSX:

# You can type regular Javascript anywhere in your JSX

If you need to inject javascript into your JSX, all you need to do is include curly braces. The javascript expression will be evaluated, and the result will be injected into the virtual DOM:

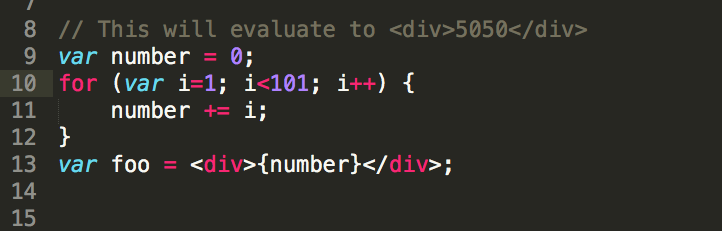


# Unfortunately, some Javascript isn’t allowed inside your JSX

You can’t use if-else statements, while loops, or for loops inside your JSX. The closest you can get is by using a ternary operator for if-else statements:

# If you need to use a for loop, while loop, or if-else block, do it before your JSX

The best practice for using complex Javascript in your components is to handle it before the JSX. The result of the loop can be saved to a variable, which can later be injected into your JSX:



### 

### 

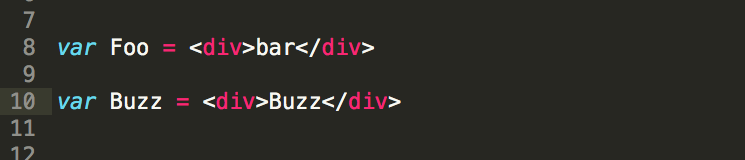
### 

### 

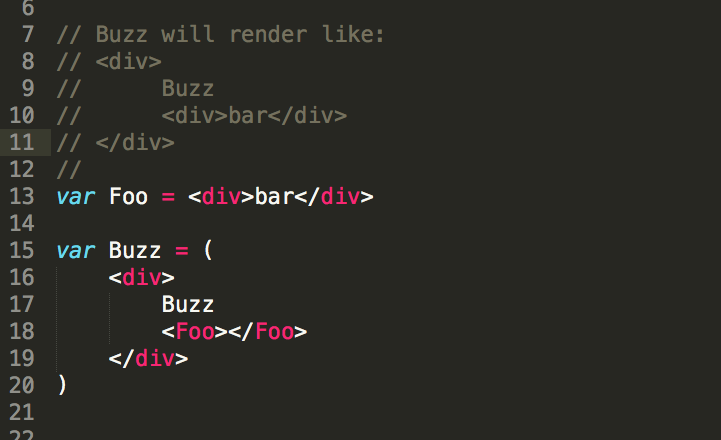
### Nesting Components

# Simple nesting

Let’s say you’ve created a component called Foo and a component called Buzz:



If you want Foo to be rendered *inside* of Buzz, you can treat it like it’s a custom HTML tag:



In fact, you can use Foo as many times as you want within Buzz:



Note: In the above example, I chose to make Foo into a self-closing tag. You’re allowed to do that!

### 

### 

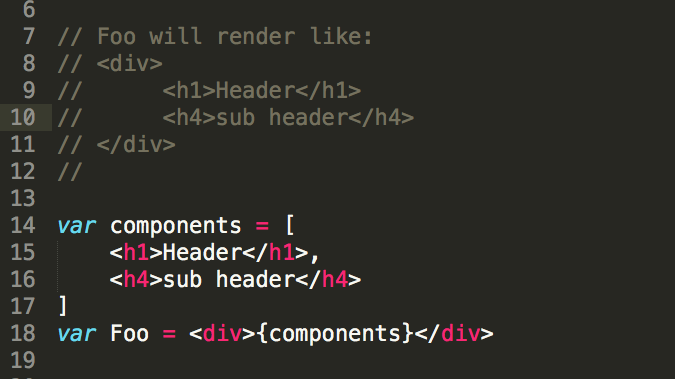
### 

### 

### 

# Nesting Components with Arrays

You can use an Array to inject nest multiple components at once:



# Siblings with same tags need the “key” attribute!!!

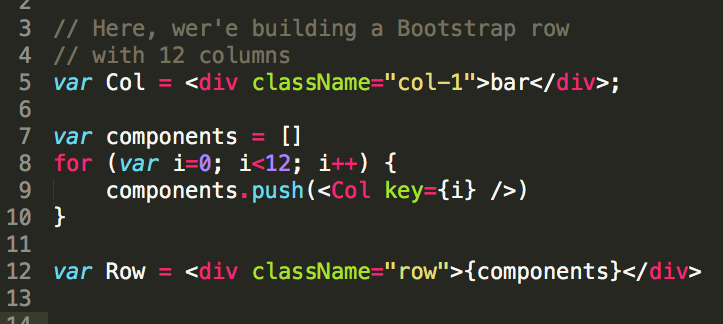
Let’s say that the components array in the example above had two h1 tags instead of an h1 and an h4…

In this case, React-DOM would have a hard time distinguishing the the h1 tags from one another! To make it easier on React-DOM, we should distinguish the two tags with the “key” attribute:



It doesn’t matter what values we set the key attribute to, as long as they are different than their siblings. **Including unique keys in identical siblings is the only way React can tell the two apart**. You can leave out the keys if you want, but you’ll get a console warning, and you might notice your application slow down a bit.

Most often, you’ll use keys when you build an array of similar elements:



### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

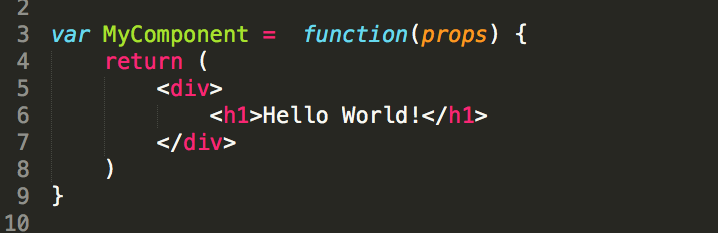
### 

### 

### Components as variables, functions, and classes

All throughout this document you’ve seen JSX saved to variables. **These variables are not full components!** Sure, they represent pseudo-html, but for them to actually exist in the virtual DOM, they need to be wrapped with either a function or a class. More often than not, you’ll be defining your components as classes so that you can gain access to props, state, and the Component lifecycle methods (These concepts are complicated enough to deserve their own document… )

As a function:



As a class:

