Basic JavaScript, DOM, jQuery

Basic JavaScript

comments: // or /\* stuff \*/

data types:

*1. undefined*

*2. null*

*3. boolean*

*4. string*

*5. symbol*

*6. number*

*7. object.*

Declaring variable

**var keyword**

var ourName; // have variable *undefined* if not given initial variable.

var ourName = “Dog”;

**let keyword**

only defined in scope nearest enclosing block. Global if outside any block.

block scoping

let ourName = “scopedDog”;

**const keyword**

defined once, never redefined.

const ourName = “scopedDog”;

if assign ourName = “fish” => TypeError!

**Temporal Dead Zone:**

for **let** and **const**, both are hoisted but but cannot be accessed until after it has been declared. You will have ReferenceError if you call it before declared. While with **var**, you can call it before declared and get undefined.

Variables and functions are case sensitive and variables are typically in camelCase.

Numbers

*is a wrapper object so that you can work with numerical values*.

operations are same as other languages.

I++ = i+1;

numbers can be decimal points, no need to do anything different.

% remainder operator is not modulus, it does not work with negative numbers.

MyVar **+=** is myVar = myVar + num;

myVar **-=** is myVar = myVar – num;

myVar **\*=** is myVar = myVar \* num;

myVar **/=** is myVar/num;

When JavaScript variables are declared, they have an initial value of undefined. If you do a mathematical operation on an undefined variable your result will be NaN which means "Not a Number". If you concatenate a string with an undefined variable, you will get a literal string of "undefined".

Not all real numbers can accurately be represented in floating point. This can lead to rounding errors. **Loss of Significance Errors**

Strings

*global object.*

var stringName = “Strings can be added together.”;

if you want to use the quotations marks “ or ‘ inside your string, you need to use the **escape** from the quotes using an escape sequence”.

**Escape Sequences**

| Code | Output |
| --- | --- |
| \' | single quote |
| \" | double quote |
| \\ | backslash |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \b | backspace |
| \f | form feed |

**Length of String**

.length will get the length of string literal. Last character of string is

stringName[stringName.length-1];

**Indexing a String**

Indexing of string is through [x]; // starts from 0

**Immutability of String**

strings are *immutable* meaning that once defined, they cannot be changed.

E.g. var myStr = “Bob”;

myStr[0] = “J”; =>NO CHANGE OCCURS!

Need to reassign to get change…

myStr = “Job”;

**Copy A String**

String.slice();

Good way to copy a string, as it returns a new copied version of the string so that you can modify the values leaving the old string unaffected. This is done instead because when you assign a variable to a string, all you do is set a reference to the original, and your copy is immutable.

**Finding Occurrences of Words in Strings**

Regular expressions are used to find certain words or patterns inside of strings.

For example, if we wanted to find the word the in the string The dog chased the cat, we could use the following regular expression: /the/gi

Let's break this down a bit:

/ is the start of the regular expression.

the is the pattern we want to match.

/ is the end of the regular expression.

g means global, which causes the pattern to return all matches in the string, not just the first one.

i means that we want to ignore the case (uppercase or lowercase) when searching for the pattern.

andCount = testString.match(expression).length;

for **numbers** use expression: var expression = /\d+/g; // the + symbol means look for more than one.

For **whitespace** use expression: var expression = /\s+/g;

For any **non-whitespace** use expression: var expression = /\S/g;

Arrays

var name = [“stuff”, “stuff” ]

can contain different types of variables in same array.

also has zero-based indexing.

Nested arrays var name=[[“dog”, 1],[2,4]];

arrays are *mutable*.

**Push()** = Adding value to end of array

**Pop()** = removing value from end of an array;

**Shift()** = removing first value from array;

**unShift()** = adding first value to array

**Modify Array**

var deletedElements = array.splice(start, deleteCount, item1, item2...)

returns modified array, deleteCount is number of elements you want to remove, and item1, etc are elements that you want to add to the array.

**Selecting Parts of Array**

array.slice(beginningIndex, endIndex)

returns a new array from items that you selected.

If no arguments, shallow copy of entire array.

Conditional Statements

**if statements**

if (condition){

statements;

}

wasThatTrue and isItTrue are boolean conditions uses in if statements.

**== Equality operator**

JavaScript can compare over two different data types such as 1 == ‘1’ = TRUE.

**=== Strict Equality Operator**

JavaScript has an operator that compares data types as well. Here 1 === ‘1’ = FALSE.

**Typeof operator**

use typeof value to get the type of the value.

*!= inequality operator.*

*!== strict inequality operator.*

&& AND

|| OR

remember to string your if and else if statements from smallest to biggest to capture the best resolution!

**Switch statements**

switch (num){

case value1:

statement1;

break;

case value2:

statement2;

break;

…

case valueN:

statementN;

break;

default:

statement;

}

a cool trick using switch statements:

switch(val) {  
case 1:  
case 2:  
case 3:  
result = "1, 2, or 3";  
break;  
case 4:  
result = "4 alone";  
}

refactoring trick!

don’t need if or else statements if return [equality statement] does the same.

Loops

**for loops**

for([initialization];[condition]; [final-expression]){

statements;

}

**while loops**

while(condition){

statements;

}

**do...while loops**

do{

// statements

} while(condition);

will run through all the statements in the do loop before checking the while condition, and if true, will repeat do statements until while condition is false.

Functions and Scoping

**Scope**

local vs global:

global variables are just variables defined outside all functions and blocks.

Local variables are just within a block. Local variables with the same name as global variables will override the global variable.

**functions (nonES6)**

function functionName(input variables){

return variable;

}

// if no return, then undefined is returned.

**arguments object**

you can get the arguments input into a function by going arguments[x] within the function.

**Return Early Pattern (for debugging)**

function myFun() {  
  console.log("Hello");  
  return "World";  
  console.log("byebye")  
}  
myFun();

So that if there is an error in the function, and function does not return, you will have a console output.

Objects

similar to arrays but instead of using indexes to access and modify data, you access data through *properties*. Nearly everything in Javascript is an object, except for the primitives (String, Number, and Boolean) which are immutable, but even these have wrappers to make them behave like objects.

*object layout*

var objectName ={

“prop1”: “stuff”,

“prop2”: 2,

“prop3”: 1,

“prop4”: [“Water”, “Dogs”]

};

accessing properties: *objectName.prop1*;

or *objectName[“prop1”]*;

Here is an example of using a variable to access a property:

var someProp = "propName";  
var myObj = {  
propName: "Some Value"  
}  
myObj[someProp]; // "Some Value"

**overwrite properties**

by going *objectName.prop1 = “NEW THING”;*

**add properties**

by going *objectName.newprop = “NEW Thing!”*;

**delete properties**

by going *delete objectName.prop1*;

**check if property exist**

by.hasOwnProperty() returns true or false.

You can also **nest objects** together

**creating objects using constructor functions**

var Car = function() {  
  this.wheels = 4;  
  this.engines = 1;  
  this.seats = 5;  
};

As functions are actually just objects.

**making instances of objects with constructor function**

var myCar = new Car();

another cool example:

// Setup

function phoneticLookup(val) {

var result = "";

var lookup ={

alpha: "Adams",

bravo: "Boston",

charlie: "Chicago",

delta: "Denver",

echo: "Easy",

foxtrot: "Frank"

}

result = lookup[val];

return result;

}

phoneticLookup("charlie");

**map method**  is used to iterate through an array using multiple processes simultaneously.

OldArray.map(function(val)){ //multiplies all values in array by 4

return val\*4;

}

**reduce method** is used to operate cumulatively on each value of array.

var singleVal = array.reduce(function(previousVal, currentVal) {  
  return previousVal - currentVal;  
}, 0);

**filter method** is used to filter out unwanted variables.

array = array.filter(function(val) {  
  return val !== 5;  
});

**sort method** is used to sort array, it changes the original array and typically you should do a compare function inside of it too so that it knows how you want to sort it.

var array = [1, 12, 21, 2];  
array.sort(function(a, b) {  
  return a - b;  
});

**reverse method** is a method that just reverses an array, it alters the original array.

var myArray = [1, 2, 3];  
myArray.reverse();

**concat method** is a method that joins an array to the end of the first array.

newArray = oldArray.concat(otherArray);

**split method** is a method to split strings into many strings and put them into an array.

Var array=string.split(‘s’);

**join method** is a method that takes an array of strings and then joins them together to form one string. You can place an argument between each string.

Var salad = veggies.join(“ and “);

**JSON**

[JavaScript Object Notation](http://www.json.org/) or JSON is a related data interchange format used to store data.

{  
"artist": "Daft Punk",  
"title": "Homework",  
"release\_year": 1997,  
"formats": [   
"CD",  
"Cassette",  
"LP"  
],  
"gold": true  
}

**Unique Objects**

var Car = function(wheels, seats, engines) {  
  this.wheels = wheels;  
  this.seats = seats;  
  this.engines = engines;  
};

Objects can have their ownfunctions called **methods**.

var Car = function() {

// this is a **private** variables  
 var speed = 10;

// these are **public** methods  
 this.accelerate = function(change) {  
 speed += change;  
 };

this.decelerate = function() {  
 speed -= 5;  
 };

this.getSpeed = function() {  
 return speed;  
 };  
};

Math Object

**Random Numbers**

between 0 to 1 *Math.random()*

**floor fractions**

*Math.floor()*

console.log("Hi World"); // prints to console

**Random Numbers Within a Range**

Math.floor(Math.random() \* (max - min + 1)) + min

**parseInt()**converts numbers in string to integers. Returns NaN if first character of string cannot be converted to number.

parseInt(str, radix); where radix converts it into radix number system. e.g. 2 = binary.

**Ternary operator**condition ? Statement-if-true: statement-if-false;

Can be used to chain together multiple conditions e.g.

function findGreaterOrEqual(a, b) {  
  return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";  
}

**Document ready in vanilla Javascript**

document.addEventListener(“DOMContentLoader”, function(){//do stuff});

Closures:

Closures are functions that refer to independent (free) variables (variables used locally, but defined in an enclosing scope). These variables remember the environment in which they were created.

JSON APIs and Ajax

APIs mean Applicaiton Programming Interfaces

Ajax is a technology that updates HTML with data

// code for getMessage button

$(“#getMessage”).on(“click”, function(){

$.getJSON("/json/cats.json", function(json){

$(".message").html(JSON.stringify(json));

});

#(“.message”).html(“message”); // changes the html message in .message

});

JSON is a data format for transferring API data.

{key :value, key:value}

adding JSON data to html variable.

json.forEach(function(){

var keys = Object.keys(val);

html += "<div class = 'cat'>";

keys.forEach(function(key){

html+= "<strong>" + key + "</strong>: " + val[key] + "<br>";

});

html += "</div><br>"

});

rednering images from JSON

html += "<img src = '" + val.imageLink + "' " + "alt='" + val.altText + "'>";

getting location:

if (navigator.geolocation){

navigator.geolocation.getCurrentPosition(function(position){

$("#data").html("latitude: " + position.coords.latitude + "<br>longitude: " + position.coords.longitude);

});

}

**Calling an api**

url = “something.com”

$.ajax({

Type: “GET”,

[url:url](../../../../../../C:/Users/JackyRecelis/AppData/Roaming/Microsoft/Word/url),

success:function(data){

//do press

}

**NodeJS**

This is a package that allows you to turn your computer into a server and access server side data using Javascript, rather than using PHP or any server side language.

NodeJS has a set of built-in modules, similar to javascript libraries that you can just use.

In windows, use the command node filename.js to run any nodeJS file.

In linux/ubuntu, use the command nodejs to run the files.

**Require(‘modulename’)**  keyword: To use a module

**Creating a module**

Use the **exports**  keyword to make properties and methods available outsid of module file. Kind of like public keyword in java amd C++.

e.g.

exports.myDateTime = function(){

return Date();

};

var http = require('http');  
http.createServer(function (req, res) {  
    res.writeHead(200, {'Content-Type': 'text/html'});  
    res.write(req.url);  
    res.end();  
}).listen(8080);

To log into server as client, go to browser and type in [http://localhost:8080/](http://localhost:8080/summer).

the module **http** allows Node.js to transfer data using the hyper-text transfer protocol, and can create servers to do this bys using the createServer() function in the http module.

Http.createServer(function(req, res){

// can do things on server creation

});

The res argument is the response you get from the server. So when you’re building your own server, you can write strings or do stuff with your response.

you can write to the screen using the function: res.**write()**.

you can add a HTTP header using the function:

res.**writeHead(200, {response headers})**;

the first argument is the status code, and the second is the response headers

The end of your response is given using the res**.end()** function.

The **listen** method at the end of createServer tells the server to listen to port 8080.

The **req** argument is the request from the client and is an object.

You can get the input **url** of the client logging into your server. This is the url that follows the localhost.

NodeJS as a file server. **Require(‘fs’)** so that you can access files on your computer.

You can read files. create files, update files, and delete files using nodeJS.

e.g.

inside the createServer function →

fs.**readFile(‘fileinsamedirectory’, function(err,data){**

**// do stuff**

**// data is the data, if in html, you can specify in writeHead what content it is using**

res.writeHead(200, {'Content-Type': 'text/html'});

**})**;

Other functions for file system

fs.appendFile()

appends to end of file. If file does not exist, creates that file.

Etc.

fs.open()

fs.writeFile()

to delete files use: fs.unlink()

Javascript Browser Object Model (BOM)

not all same in difference browsers

window.innerHeight

window.innerWidth

[doesn’t incluse scrollbars]

window.open()

window.close()

window.moveTo()

window.resizeTo()

Window.Screen Object

information on user’s screen, don’t need to use window. Prefix.

Props

screen.width

* screen.height
* screen.availWidth, in pixels minus things like Windows Taskbar
* screen.availHeight
* screen.colorDepth, amount of different colours computer resolution, based on hardware
* screen.pixelDepth, same as colorDepth for modern computers

Window.location object

can be written without window prefix

* window.location.href returns the href (URL) of the current page
* window.location.hostname returns the domain name of the web host
* window.location.pathname returns the path and filename of the current page
* window.location.protocol returns the web protocol used (http: or https:)
* window.location.assign(“URL”) loads a new document
* window.location.port returns port number of page, if default 80 for http and 443 for https, browsers will display 0 or nothing.

Window.history

can be written without window prefix.

* history.back returns previous URL
* history.forward loads next URL on history list

Window Navigator

can be written without window prefix

* navigator.cookieEnabled returns true if cookies enabled, else false
* navigator.appName returns application name of browser, Netscape is application name of IE11, Chrome, Firefox and Safari. So doesn’t give that much info.
* Navigator.appCodeName returns app code name which apparently is Mozilla for pretty much everything.
* Navigator.product returns produce name of browser engine. For Mozilla this is Gecko.
* Navigator.appVersion returns version of browser
* Navigator.userAgent returns browser type, OS and 32bit or 64bit.

Warning, don’t use this navigator object to get browser type because people can change it.

* Navigator.platform returns operating system.
* Navigator.language returns browser’s language.
* Navigator.onLine returns true is browser is online.
* Navigator.javaEnabled() returns true if Java is enabled.

Javascript Popup Boxes

all can be written without window prefix

* window.alert(“string”) can be written without window prefix
* window.confirm(“sometext”) will have OK returns true and Cancel returns false,
* window.prompt(“sometext”, “defaultText”) gives a box with some text and a form.
* Line breaks are done using \n characters.

Timing Events

all can be written without window prefix

* window.setTimeout(function, milliseconds) runs function after waiting for milliseconds
* window.clearTimeout(setTimeoutHandle) stops setTimeout if function has not begin running yet.
* Window.setInterval(function,milliseconds) function is executed between time intervals set by second argument.
* Window.clearInterval(setIntervalHandle) stops setInterval

Cookies

* cookies saved in name-value pairs e.g. username = someone someone
* document.cookie property
* create cookie: document.cookie= “username=John Smith”;
* expiry date of cookie: document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
* can tell browser which path cookie belongs to. document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";
* reading cookie by: var x= document.cookie
* change cookie same way as creating it.
* Delete cookie by setting expires parameter to passed date.

**ReactJS**

It’s a Javascript library that is used to create user interfaces. It’s good because it allows you to render parts of a website from the server which makes it faster on your own computer because you don’t need to do that much on your own computer. The basic building blocks of stuff in React are components.

You’ll need to call the ReactJS headers and add this line to initiate your fundamental class. Here it is element. The header files are:

import React from 'react';

import ReactDOM from 'react-dom';

If you writing in a different file, you’ll need to import that class using:

import Game from './Game';

You’ll need to export everything from that other file using this command:

export default Game;

To actually render everything, you’ll need to call this to your fundamental component.

ReactDOM.render(<Game />, document.getElementById('root'));

**JSX**

ReactJS is typicaly used with JSX, a syntax extension to Javascript which allows HTML to be embedded into your Javascript code. You need to use {} braces for Babel to use Javascript expressions into JSX HTML code, this allows you to use Javascript to change different components elsewhere.

If you have an empty tag, <img></img>, you write instead as <img /> which is also how you write components.

You can embed user input in JSX without a malignant user putting malware into your code because JSX converts everything into Strings before it used.

Elements in JSX

These are the objects that can contain headings, methods to run these objects, and values.

Creating an element is as simple as:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

Or using the createElement function:

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

This is what it actually looks like:

// Note: this structure is simplified

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world'

}

};

As I said before: elements are rendered onto the DOM using the render function within the reactDOM node.

const element = <h1>Hello, world</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

Elements are immutable, they can’t be changed at all after they’ve been made, so the only way to change an element is by creating a new element everytime and then calling the render function in reactDOM.

But most of the time, you only call the reactDOM.render() function only once in your app, and the way you do this is by using stateful components.

The really cool thing about React is that it only renders the changes you’ve made, so even if you call the whole element that contain heaps of children, React will look at your current element and compare it to what it was, and then only make the changes that you made.

Functional Components and Props

Are things that accept at least one property (**props**) and return an element. This element is defined by the user, using a props. Props is an object and is usually defined and sent to the functional component as another element. Functional components are named as such because they are actually functions, and differ from class components.

*Always start component names with a capital letter.*

*For example, <div /> represents a DOM tag, but <Welcome /> represents a component and requires Welcome to be in scope.*

Components are usually created to represent an entity like a button, screen, or form. Their output is often important, returning the element behind the component itself, and so components are often called from other components. E.g. A component Board calls multiple Button components, which render these multiple buttons on the board.

*Components must return a single root element. This is why we added a <div> to contain all the <Welcome /> elements.*

Components that have a lot of nesting can be written so it has components within the component of it extracted out of it. Kind of like taking cumbersome function, and making parts of the function into small functions.

*We recommend naming props from the component's own point of view rather than the context in which it is being used.*

The big rule behind React is that props must **NEVER change** inside a component (otherwise called ‘pure’). But of course websites are dynamic, so the way around this is by using states.

State and Lifecycle

Make a constructor function within your class component.

Class ClassName extends Component{

Constructor(props){ // props if you want to pass values into component from another component

Super(props);

this.state = {

stateName:stateValue

}

}

}

**Lifecycle Hook Methods:**

Methods of a class component that change based off component event.

componentDidMount() {

}

componentWillUnmount() {

}

The componentDidMount() hook runs after the component output has been rendered to the DOM.

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output.

**If you don't use something in render(), it shouldn't be in the state.**

**Using State Correctly: 3 Things**

1. Do not modify State directly, use setState();
2. State may be set asynchronously, do not rely on their values to set values for other states.

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

1. State updates are merged, so you can do multiple state updates within one setState.

Data flows down.

So parent or child do not know if have states or not but states can be passed down to children as props.

**Handling Events**

onClick = {doSomethingFunction}

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly.

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('The link was clicked.');

}

return (

<a href="#" onClick={handleClick}>

Click me

</a>

);

}

Here, e is a synthetic event.

If you want to use *this* in your event function: add this to constructor class

// This binding is necessary to make `this` work in the callback

this.handleClick = this.handleClick.bind(this);

Can use ES6 functions if you don’t want to use bind().

// This syntax ensures `this` is bound within handleClick.

// Warning: this is \*experimental\* syntax.

handleClick = () => {

console.log('this is:', this);

}

Or this:

// This syntax ensures `this` is bound within handleClick

return (

<button onClick={(e) => this.handleClick(e)}>

But if you pass this callback down to lower components, it may trigger double renderings.

**Conditional Rendering**

You can use if statements to render different things.

if (isLoggedIn) {

return <UserGreeting />;

}

return <GuestGreeting />;

}

You can do this within an expression in JSX by doing something like this: (equivalent to an if statement)

{unreadMessages.length > 0 &&

<h2>

You have {unreadMessages.length} unread messages.

</h2>

}

Because anything false that is && will equal false, and will not be rendered.

If else inline can easily be done using the **ternary operator.**

The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.

If you want to hide a component, return null.

Lists and Keys

**Basic Lists**

Best to do this inside a component using a map() function. You’ll need a key as well, because React needs to be able to reference each item in your array to know which one has been changed. Most often you would use IDs from your data as keys:

function NumberList(props) {

const numbers = props.numbers;

const listItems = numbers.map((number) =>

<li key={number.id}>{number}</li>

);

return (

<ul>{listItems}</ul>

);

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

<NumberList numbers={numbers} />,

document.getElementById('root')

);

Keys serve as a hint to React but they don't get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name.

Forms

**Controlled Components**

Forms typically have own internal state. You want to be able to have a submission function that has access to your data. So you want to combine internal state with your component states.

In your form element:

<form onSubmit={this.handleSubmit}>

<label>

Name:

<input type="text" value={this.state.value} onChange={this.handleChange} />

</label>

<input type="submit" value="Submit" />

</form>

With the handleChange function using the target value of the event. (What is event? What is the target value?)

handleChange(event) {

this.setState({value: event.target.value});

}

**TextArea**

In React, a <textarea> uses a value attribute instead. And everything else is the same.

**Select Tag**

In HTML, <select> creates a drop-down list. For example, this HTML creates a drop-down list of flavors.

Put this into your form, and you’ll be able to record your values.

<select value={this.state.value} onChange={this.handleChange}>

<option value="grapefruit">Grapefruit</option>

<option value="lime">Lime</option>

<option value="coconut">Coconut</option>

<option value="mango">Mango</option>

</select>

Lifting State Up

If you want to change state of parent in props, you can make event function ‘controlled’. E.g. pass the event function down as well.

There should be a single "source of truth" for any data that changes in a React application.

If something can be derived from either props or state, it probably shouldn't be in the state.

Composition and Inheritance

Sometimes, you don’t know what’s going to be inside a component, based on your parent component, so you can pass in your props like this.

function FancyBorder(props) {

return (

<div className={'FancyBorder FancyBorder-' + props.color}>

{props.children}

</div>

);

}

Where props.children fills in everything you need into FancyBorder.

function WelcomeDialog() {

return (

<FancyBorder color="blue">

<h1 className="Dialog-title">

Welcome

</h1>

<p className="Dialog-message">

Thank you for visiting our spacecraft!

</p>

</FancyBorder>

);

}

If you want more specialisation, you can call it another name other than children.

function SplitPane(props) {

return (

<div className="SplitPane">

<div className="SplitPane-left">

{props.left}

</div>

<div className="SplitPane-right">

{props.right}

</div>

</div>

);

}

function App() {

return (

<SplitPane

left={

<Contacts />

}

right={

<Chat />

} />

);

}

Specialisation

You can make a component do different things by having a component within another component.

this gives functionality to webpages and allows you to run JavaScript

give elements class=”**target**”

<script>

$(document).ready(function(){

// code here runs as soon as page is open

$("button").addClass("animated bounce"); // can target all **elements** of a type

$(".well").addClass("animated shake"); // can target elements by **class**

$("#target3").addClass("animated fadeOut"); // can target elements by **id**

$("#target1").addClass("btn-primary"); // can target same element by different handle

$("#target1").("color", "red"); // can change **CSS** of an element

$("#target1").**prop**("disabled", true); // can access property of element

$("#target4").**html**("<em>#target4</em>"); // can affect text in html, here *emphasis*

$("#target4").**remove**(); // can **remove** element completely

$("#target2").**appendTo**("#right-well"); // can move elements to different wells

$("#target5").**clone().appendTo**("#left-well"); // can **copy** element to different wells

$("#target1").**parent**().css("background-color", "red"); // cantarget element of **parent**

$("#right-well").**children**().css("color","orange"); // can target **children** of element

$(".target:nth-child(2)").addClass("animated bounce"); // target **specific children** of element

$(".target:even").addClass("animated shake"); // can target **odd** or **even** elements

$("body").addClass("animated hinge"); // can handle entire page

});

</script>