latex input: mmd-article-header Title: JavaScript Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: web programming, syntax, language, javascript CSS: http://fletcherpenney.net/css/document.css xhtml header: Copyright: 2014 Ethan Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

Array.prototype.splice()

Changes the contents of an existing array (in place!), adding and removing elements at the same time. This method is too confusing to use, in my opinion, but I've been seeing it quite a bit, so I'm guessing it's actually pretty common among JavaScript programmers.

```
array.splice(index , howMany[, element1[, ...[, elementN]]])
```

1. **index** is the starting index in the `array`
2. **howMany** says how many elements to ***remove*** from the `array` (starting at `index`)
3. the **element list** specifies which elements to add to the array (after removing `howMany`, starting at `index`)
4. **returns an array** containing all the removed elements (empty array if `0`, one-element array if `1`, etc.)

Use cases

1. **Remove an element from an array**

    ```
    array.splice(idxToRemove, 1);
    ```

ready callback

Called when the entire page has loaded (or something like that)

7/22/14

In `jQuery`, you can do

```
// A
$(document).ready(myHandler);

// B
$(myHandler);

// C
$(function() {
  myCode;
});

// C in Coffeescript
$ ->
  myCode
```

delete keyword

Removes a property from an object

4/15/14

MDN

Does not directly free memory

- There is a mark-and-sweep **garbage collector**, one of whose advantages over ARC is that cycles are still collectable because they are unreachable from roots in the stack.

Examples:

```
delete object.property
delete object['property']

var y = 23;
delete y; // returns false because we can't delete variable names

var x = 24;
delete x; // returns true because we can delete properties of the g
```

Deleting inherited properties has no effect, though returns `true`

this keyword

4/14/14

Nice Tutorial

By default, `this` refers to the object that invokes the function where `this` is used. Though see below for how to change this behavior using the `bind()`, `call()`, and `apply()` methods.

(When the code is executing in the browser...) **All global variables and functions are defined on the `window` object**. Therefore, `this` in a global function refers to the global `window` object (not in `strict` mode though).

bind(), call(), apply()

4/14/14

- Nice Tutorial
- MDN

Recall that **functions** are **objects**, and **objects** have **prototypes**, and **prototypes** provide **methods**. So the methods `bind()`, `call()`, and `apply()` belong to `Function.prototype`.

Allows you to easily set which specific object will be bound to *`this`* when a function or method is invoked.

The problem is that when we use the `this` keyword in methods, and we call said methods from a receiver object, sometimes this is not bound to the object that we intended. Said again, when we execute an object's method from within another object, the `this` keyword now refers to calling object, *not* the object where `this` was originally defined. This will cause errors, and requires us to use one of the methods in this § to specifically set the value of `this`.

.apply()

```
instance.method.apply(object2, array)
```

Call the `method()` method of `instance`, such that `this` (when used *inside* `method()`) refers to `object2`. We are passing `array` to `method()` as parameters. Now, if `method()` assigns any new attributes to `this`, they will become attributes of `object2`.

Borrowing Array methods

`Arrays` come with a number of useful methods for iterating and modifying arrays, `Objects` don't have useful native methods. If we have an "*array-like object*", we can borrow Array methods and use them on objects that are array-like.

An **array-like object** is an `object` that has its **keys defined as non-negative integers**. It is best to *specifically add a length property*.

For example:

```
var anArrayLikeObj = {
    0 : "Martin",
    1 : 78,
    2 : 67,
    3 : ["Letta", "Marieta", "Pauline"],
    length : 4
};
```

Gaze in astonishment as we utilize its *array-like-ness* to its fullest

```
// As usual, the first parameter sets the "this" value
var newArray = Array.prototype.slice.call (anArrayLikeObj, 0);
console.log (newArray); // ["Martin", 78, 67, Array[3]]
// notice the 'length' property got skipped because it
// is *not* a non-negative integer
```

Note that the `arguments` variable inside any function containing everything passed to the function as parameters is also an "*array-like object*" upon which we may call

```
Array.prototype.slice.call(arguments);
```

and all the other `Array` methods, like `indexOf()` and so on.

- We never made a full *copy* of the method, we only created some sort of *pointer* to it (conceptually at least).
- *Note:* If we change the original method, and the method has been set as a method on another object, the changes are reflected in the borrowed instances.

Variable-arity / Variadic functions

- It's like `printf(str, args...)` in Java
- E.g.`Math.max(1,2,3,5,6,7) // => 7` (look ma, it takes any number of arguments)
- We are ***not allowed*** to pass it a *list* instead, e.g. `Math.max([1,2,3,5,6,7])`
- But (the reason this JS feature is in this part of the document) we *can* `apply()` that list to it

```
Math.max.apply(null, [1,2,3,5,6,7])
```

- Recall that the first argument sets the `this`, which we don't need here

.call()

Quite similar to `.apply()`. In fact doing

```
instance.method.call(object2, param1, param2)
```

does exactly what it would do if we replaced `call()` with `apply()` (see above for what it does actually do), only we've replaced the `array` containing parameters, with the actual parameters passed in in a more (one might say) "normal" way.

.bind()

```
$('button').click(user.clickHandler);
```

At this point, if we use `this` from inside `var user`'s `clickHandler` function, `this` will be refering to the `<button>` element. But if we instead want it to refer to the `user` object, we must `bind` it.

So instead if we do

```
$('button').click(  user.clickHandler.bind(user)  );
```

and now it'll manipulate our `user` object as we wanted.

.bind() also facilitates currying

In this example, we *essentially* pass

```
takes3( this = null, a = true, b = 45, c = ¿unapplied? )
```

via `bind`. This returns us a *curried* function that is still waiting to receive a `c`, so that's why we pass `c` to that function and it gets used, along with the *values of variables that we had previously set in* `bind`.

```
function takes3(a, b, c) {
    var s = a ? 'g' : 'f';
    if (b > 25) return c + s;
    else return "yaya" + s;
}

var gAboveB = takes3.bind(null, true, 45);
gAboveB("asdf"); // => "asdfg" (fills in the last parameter from the
```

event.which

Indicates which key was pressed, via its *unicode* value.

Doesn't seem to do anything for non-alphanumeric-character keys.

4/13/14

To check what a character key's value is

How about entering the following in to the interpreter, then going back to the browser and hitting the key you want the value of? (This may ruin any existing `document.onkeypress` handlers.)

```
document.onkeypress = function(myEvent) { // note: event needn't be
    console.log(myEvent.which);
};
```

Navigator

contains information about the visitor's browser.

3/25/14

W3 Schools

Technically, it's the `window.navigator` object, but the `window.` prefix is not required.

E.g. (from `http://webaudiodemos.appspot.com/AudioRecorder/index.html`, `./Audio Recorder_files/main.js`, function initAudio, [currently] lines 162-173)

```
if (!navigator.getUserMedia)
    navigator.getUserMedia = navigator.webkitGetUserMedia
                           || navigator.mozGetUserMedia;
```

Using their example, you can get all sorts of other possibly-useful stuff as well:

```
Browser Code-Name:     navigator.appCodeName
Browser Name:          navigator.appName
Browser Version:       navigator.appVersion
Cookies Enabled:       navigator.cookieEnabled
Platform:              navigator.platform
User-agent header:     navigator.userAgent
User-agent language:   navigator.systemLanguage
```