CSC309H1S

Programming on the Web

Winter 2023

Lecture 9: jQuery and Advanced JavaScript

Instructor: Kuei (Jack) Sun

Department of Computer Science
University of Toronto

v1.01

jQuery

- One of the most popular JavaScript libraries
- Simplifies HTML DOM tree traversal and manipulation
- Helps with event handling and AJAX requests
- Installation
  - https://jquery.com/download/
  - `<script src="jquery-3.6.3.min.js"></script>`
  - Choose between compressed (smaller, faster) vs. uncompressed (readable)
  - Alternatively, can use directly from a CDN
    - E.g., https://developers.google.com/speed/libraries#jquery
- Slowly being replaced by React

jQuery Basics

- Syntax
  - Everything is done through the **$** function, based on query selectors
  - Example
    - `$("p").hide()`
    - `$(document).ready(function() { /* initialization code */ });`
- Effectively a wrapper around plain JavaScript
- jQuery objects have different methods/properties

```
// Plain JavaScript
document.querySelector("#title").innerHTML = "<h1>Hello</h1>";

// jQuery
$("#title").html("<h1>Hello</h1>");
```

- Designed to support *chaining*

Common jQuery Methods

- `val([value])`
  - Get or set input value

```
username.val()
```

- `attr(k [, value])`
  - Get or set attribute with name *k*

```
$("a").attr("href");
```

- `css(p [, value])`
  - Get or set CSS property with name *p*

```
input.css("color", "blue");
```

- `html([value])`
  - Get or set arbitrary HTML

- `click(function)`
  - Register onclick event

```
$("input").click(function() {
    $(this).css("background-color",
                "lightcyan");
});
```

- `parent(), children()`
  - Get parent or children
- `next(), prev()`
  - Get next or previous sibling
- `addClass(), removeClass()`
  - Add or remove class(es)

Quercus Exercise Q1

- Create a form with the following fields:
  - Username
  - Email
  - Password
  - Repeat password
  - Security question: "What's 8 + 16/4?"
- Implement client-side validation *using jQuery*, with these checks:
  - Checks if the security question is answered correctly.
  - Checks password and repeat password are the same.
  - Checks if domain name of email address ends with "utoronto.ca".
- Errors should appear when “Sign me up!” is clicked

Ajax with jQuery

- $.ajax(url [, settings])
- Can specify URL, method, etc.
  - All optional
  - https://api.jquery.com/jquery.ajax/
- Accepts handler for success or error
- On success
  - data parameter contains JSON result

```
var jqxhr = $.ajax("example.php")
  .done(function(data) {
    alert("success: " + data);
  })
  .fail(function() {
    alert("error");
  });
```

```
$.ajax("/user/", {
    method : 'PATCH',
    data : {
        username : $('#username-input').val(),
    },
    headers : {
        'X-CSRFToken' :
        $('input[name=csrfmiddlewaretoken]').val(),
    },
    success : function() {
        $('.show-modal').hide();
    },
    error : function(xhr) {
        if (xhr.status === 400) {
            var resp = xhr.responseJSON;
            if (resp['username']) {
                var message = resp['username'][0];
                $error.html(message).show();
            }
        }
    }
});
```

More JavaScript

Built-in Functions/Methods

- `parseInt(x [, base])`
  - Attempts to convert string to integer
  - Returns NaN on failure
- `isNaN(x)`
  - Checks if value is NaN
  - Note: NaN === NaN is false
- `parseFloat(x)`
  - Attempts to convert string to float
  - Returns NaN on failure
- `String.padStart(n, c)`
  - Pad *n* characters with character *c*
- `setTimeout(code, time)`
  - Execute *code* after *time* millisec
- `setInterval(code, time)`
  - Execute *code* every *time* millisec
- `String.trim()`
  - Remove leading/trailing spaces
- `escape(x)`
  - Convert to URL-encoded string
- `unescape(x)`
- Quercus Exercise Q2

Sessions

- Session-based authentication
  - Browser already stores/sends cookies header
- Token-based authentication
  - You are responsible for storing/using the token
  - Use localStorage global variable

```
localStorage.setItem('access_token', access_token);
localStorage.getItem('access_token');
```

- Set Authorization header with the token value
  - In jQuery ajax request settings:

```
beforeSend: function (xhr) {
    xhr.setRequestHeader("Authorization", "Bearer " + access_token);
},
```

Closures

- Recall functions are *first class citizens* in JavaScript
  - Functions can be defined inside a function and be returned
- Closure
  - Nesting of functions where inner function has access to local variables in the outer function(s)
- Questions
  - What's stored in X and Y?
  - What's the scope of a, b, c?
  - What should happen to b at the end of function call?

```
function outer() {
    var b = 10;
    var c = 100;

    function inner() {
        var a = 20;
        console.log("a = " + a + ", b = " + b);
        a++;
        b++;
    }

    return inner;
}

var X = outer(); var Y = outer();
```

Capture

- Inner function *captures* local variable(s) from outer function
- Captured variables can be referenced by inner function
  - Each invocation of outer function creates new copies of outer variables
- Can capture function arguments as well

```
function foo(i) {
    var x = { count : 0 };
    return function() {
        x.count += i;
        console.log("x.count = " + x.count);
    };
}

m = foo(5);
n = foo(7);
m(); m(); n();
```

Output:

```
x.count = 5
x.count = 10
x.count = 7
x.count = 14
```

For Loop and Closures

- Recall that:
  - `var` declares function scope variable
  - `let` declares block scope variable
- In a for loop, var and let also behaves differently
  - `var` declares a variable once and updates its value
  - `let` redeclares the variable multiple times with different values

```
function outer() {
    let a = [];
    for (var i=1; i<=5; i++)
        a.push(function() {
            return i;
        });
    return a;
}
```

What's the output of the following code execution?

```
for (fun of outer()) {
    console.log("i = " + fun());
}
```

For Loop and Closures

- Problem
  - var only creates one variable
  - All closures created in the invocation of function references same variable
    - This causes aliasing among different closures
- Solution 1:
  - Force a copy by using immediately invoked function expression

```
function outer() {
    let a = [];
    for (var i=1; i<=5; i++)
        a.push((function(i) {
            return function() { return i; };
        })(i));
    return a;
}
```

Output:

```
i = 1
i = 2
i = 3
i = 4
i = 5
```

For Loop and Closures

- Problem
  - `var` only creates one variable
  - All closures created in the invocation of function references same variable
    - This causes aliasing among different closures
- Solution 2:
  - Use `let` to declare loop variable, which creates one variable per iteration

```
function outer() {
    let a = [];
    for (let i=1; i<=5; i++)
        a.push(function() {
            return i;
        });
    return a;
}
```

using `let` here

Arrow Function

- Similar to lambda function in Python
  - More powerful because it allows for a code block on left side of arrow
- Syntax:
  - `(param1, param2, …) => expression | body`
- Before:

```
function regular(a, b) { return a + b; }
```

- After:

```
const arrow = (a, b) => { return a + b; };
```

- Simplified:

```
const concise = (a, b) => a + b;
```

For 1 parameter, can simplify to:

```
const f = x => x + 1;
```

Functional Programming

- Arrow function used often in functional programming paradigm
- JavaScript arrays have higher order functions
- `forEach`
  - Given each element and its index, do something

```
var names = ["ali", "hassan", "mohammad"];
names.forEach((item, index) => console.log(index + ": " + item));
```

- `map`
  - For each element, modify it in some way and return a new array

```
upper = names.map(item => item.toUpperCase());
// ['ALI', 'HASSAN', 'MOHAMMAD']
```

Higher Order Functions

- `filter`
  - Returns a new array, keeping elements that satisfies the condition

```
var students = [{name: "Jay", id: 1}, {name: "Ali", id: 2}, {name: "Jay", id: 3}]
let jays = students.filter(item => item.name === "Jay");
// [{name: "Jay", id: 1}, {name: "Jay", id: 3}]
```

- `reduce`
  - Returns an aggregate value after processing the array
  - Accumulator takes an initial value

```
var names = ["ali", "hassan", "mohammad"];
let longest = names.reduce((acc, cur) => Math.max(cur.length, acc),
                           Number.NEGATIVE_INFINITY);
// 8
```

Arrow Function and this

- Regular functions can have their own this value
  - Refers to the object that called the function (method)
  - Event listeners
    - this refers to the element that triggered the event
- Arrow function *does not* have their own this value
  - Do **not** use as event listeners or object methods
  - But, arrow function can capture this in a closure (unlike regular functions)

```
const person = {
    name : 'King Bob',
    greet() {
        setTimeout(() => console.log(this.name + " says hi!"), 500);
    }
};
```

Destructuring

- Unpack values from arrays or objects into local variable
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- Destructuring an object
  - Remember the variable names has to be same as property names

```
const hero = { alias : "Batman", name : "Bruce Wayne" };
const {alias, name} = hero;
console.log(name + " is " + alias);
```

  - Can place the rest into a sub-object

```
const {alias, ...rest} = hero;
// rest is { name : 'Bruce Wayne'}
```

- Destructuring an array

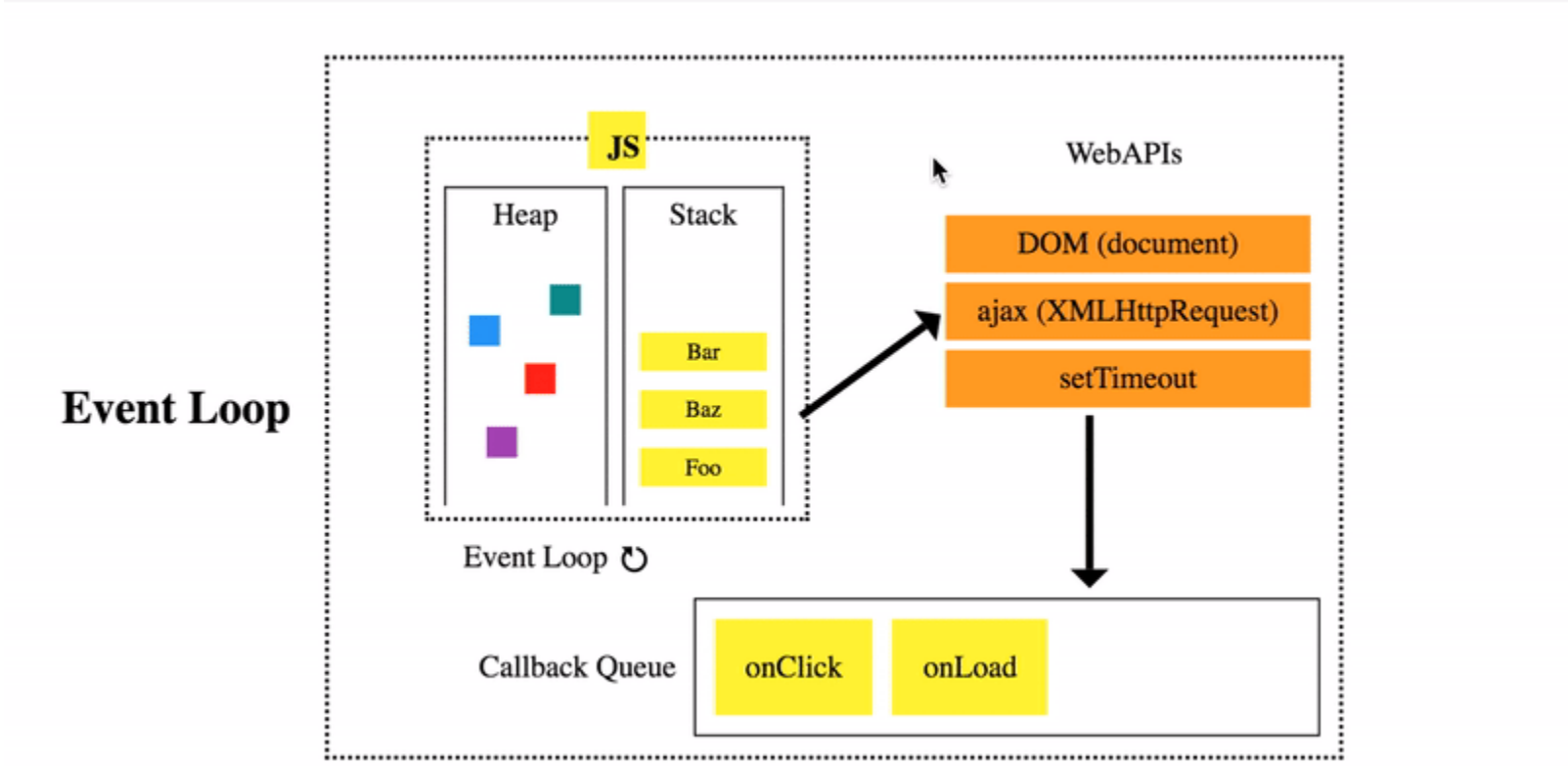
```
let a, b;
[a, b] = [3, 7]; // a = 3 and b = 7
```

Event Loop and Promises

Event Loop

- JavaScript supports event-driven paradigm
- JavaScript code is run in a single thread
  - Scripts are executed at load time; the rest are all *events.*
  - Reduces overhead of managing threads and shared variables
- Event loop provides the illusion of multiple threads
- Events are pushed to the event queue
  - Example: ready, click, ajax, setTimeout
- Event loop constantly checks for new events and execute their callback
  - This happens synchronously

Visualization of Event Loop



https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd

Callback Hell

```
fs.readdir(source, function (err, files) {
    if (err) {
        console.log('Error finding files: ' + err)
    } else {
        files.forEach(function (filename, fileIndex) {
            console.log(filename)
            gm(source + filename).size(function (err, values) {
                if (err) {
                    console.log('Error identifying file size: ' + err)
                } else {
                    console.log(filename + ' : ' + values)
                    aspect = (values.width / values.height)
                    widths.forEach(function (width, widthIndex) {
                        height = Math.round(width / aspect)
                        console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                        this.resize(width, height).write(dest + 'w' + width + '_' + filename, function (err) {
                            if (err) console.log('Error writing file: ' + err)
                        })
                    }.bind(this))
                }
            })
        })
    }
})
```

Code written in a way where execution happens from top to bottom.

Don’t write JavaScript like this

Promises

- Problem
  - Callbacks can make code hard to understand due to nesting
    - Example: jQuery ajax has at least two callbacks for success and error
- Promise
  - An alternative to using callbacks

```
let test = new Promise(function(resolve, reject) {
    resolve("resolved!");
});
test.then(msg => console.log(msg));
```

  - Code inside of promise is executed immediately
  - Calling resolve or reject pushes events to event queue (asynchronous)
  - Can later handled by the methods then or catch

Fetch API

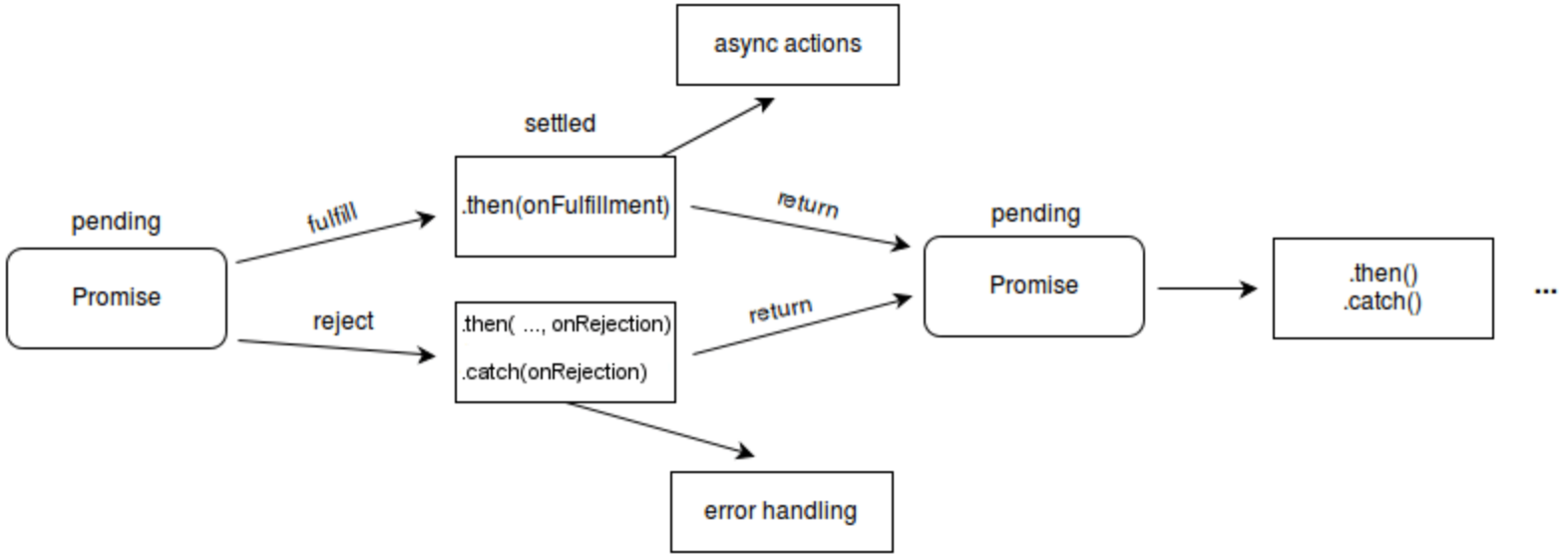
- Returns a Promise object

```
let request = fetch('/account/login/', {
    method: 'POST',
    data : {username: 'Kia', password: '123'},
});

request.then(response => response.text())
       .then(text => console.log(text));
```

- Callback is specified in the then method(s) instead of ajax object
- Promise states
  - Pending: the initial state
  - Resolved: happens when the resolve function is called
  - Rejected: happens when the reject function is called

Promise State Transition



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Example Promise

- Delayed division

```
const slow_divide = (a, b) => new Promise((resolve, reject) => {
    if (b != 0) {
        setTimeout(() => resolve(a / b), 1000);
    }
    else {
        reject("Error: attempting division by zero");
    }
});

slow_divide(6, 2)
    .then((res) => {
        console.log("result is " + res);
    })
    .catch((msg) => {
        console.log(msg);
    });
```

Both resolve and reject takes only *one* argument. If you need to pass more than one values, use an object.

Chaining Promises

- then/catch will get called even if promise is already settled
- Multiple callbacks can be added by calling then several times
- What's the output?

```
const add = (num1, num2) => new Promise((resolve) => resolve(num1 + num2));

add(2, 4)
    .then((result) => {
        console.log(result); return result + 10;
    })
    .then((result) => {
        console.log(result); return add(result, 2);
    })
    .then((result) => {
        console.log(result);
    })
```

Return value wrapped in Promise

Explicitly returning a Promise

Promise vs. Callback

- Promise
  - Still a bit of callback hell

```
slow_divide(36, 2)
    .then((res) => {
        console.log("result is " + res);
        return slow_divide(res, 3);
    })
    .then((res) => {
        console.log("result is " + res);
        return slow_divide(res, 6);
    })
    .then((res) => {
        console.log("result is " + res);
    })
    .catch((msg) => console.log(msg));
```

- Callback
  - Code difficult to read and maintain

```
function slow_divide(a, b, c, d) {
    const TIMEOUT = 1000;
    setTimeout(() => {
        let res = a / b;
        console.log("result is " + res);
        setTimeout(() => {
            res = res / c;
            console.log("result is " + res);
            setTimeout(() => {
                res = res / d;
                console.log("result is " + res);
            }, TIMEOUT);
        }, TIMEOUT);
    }, TIMEOUT);
}

slow_divide(36, 2, 3, 6);
```

async and await

- Promises still breaks program logic
- async function
  - await operator: waits for a Promise to be fulfilled before continuing code
  - Error handling naturally done through try/catch

```
async function divide_thrice(a, b, c, d) {
    try {
        let res = await slow_divide(a, b);
        console.log("result is " + res);
        res = await slow_divide(res, c);
        console.log("result is " + res);
        res = await slow_divide(res, d);
        console.log("result is " + res);
    } catch(err) {
        console.log(err);
    }
}
```

Simplifies code that consumes result from Promise objects.

No more callback hell.

Note: await can only be used inside async functions.