JS Review

RUJUAN XING

Variables and Types

```
var name = expression;
let name = expression; (ES6)
const name = expression; (ES6)

var age = 32;
let weight = 127.4;
const clientName = "Connie Client";
```

variables are declared with the `var/let/const` keyword (case sensitive)

types are not specified, but JS does have types ("loosely typed")

- Number, Boolean, String, Null, Undefined, Symbol, Object
- can find out a variable's type by calling typeof

Logical Operators

>, <, >=, <=, &&, ||, ! ==, !=, *===, !==*

most logical operators automatically convert types:

- 5 < "7" is true
- 42 == 42.0 is true
- "5.0" == 5 is true

=== and !== are **strict** equality tests; checks both type and value

- "5.0" === 5 is false

Always use **strict** equality

Boolean Type

```
let iLikeWebApps = true;
let ieIsGood = "IE6" > 0; // false
if ("web dev is great") { /* true */ }
if (0) { /* false */ }
```

any value can be used as a Boolean

- "falsey" values: false, 0, 0.0, NaN, empty String(""), null, and undefined
- "truthy" values: anything else, include objects

!! Idiom – gives boolean value of any variable

- const x=5;
- console.log(!x);
- console.log(x);
- console.log(!!x);

String Type

```
let s = "Connie Client";
let fName = s.substring(0, s.indexOf(" ")); // "Connie"
let len = s.length; // 13
let s2 = 'Melvin Merchant'; // can use "" or ' `
let s3 = `Melvin Merchant`;
```

methods: charAt, charCodeAt, fromCharCode, indexOf, lastIndexOf, replace, split, substring, toLowerCase, toUpperCase

- charAt returns a one-letter String (there is no char type)

length property (not a method as in Java)

concatenation with + : 1 + 1 is 2, but "1" + 1 is "11"

Array

```
let a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
```

array serves as many data structures: list, queue, stack, ...

methods: concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift

- push and pop add / remove from back
- unshift and shift add / remove from front
- shift and pop return the element that is removed

Array methods

```
let a = ["Stef", "Jason"]; // Stef, Jason
a.push("Brian"); // Stef, Jason, Brian
a.unshift("Kelly"); // Kelly, Stef, Jason, Brian
a.pop(); // Kelly, Stef, Jason
a.shift(); // Stef, Jason
a.sort(); // Jason, Stef
```

array serves as many data structures: list, queue, stack, ...

methods: concat, join, pop, push, reverse, shift, slice, sort, splice, toString, unshift, filter, map, reduce

- push and pop add / remove from back
- unshift and shift add / remove from front
- shift and pop return the element that is removed

Function Declaration

```
function name() {
 statement ;
 statement ;
 ...
 statement ;
}

function square(number) {
 return number*number;
}
```

declarations are "hoisted" (vs function expressions) – see Lecture07

- They can be declared anywhere in a file, and used before the declaration.

Function Expressions

Can be Anonymous function

- Widely used in JS with event handlers

```
const square = function(number) { return number * number };
const x = square(4) // x gets the value 16
```

Can also have a name to be used inside the function to refer to itself //NFE (Named Function Expression)

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) };
console.log(factorial(3));
```

Basically, a function expression is same syntax as a declaration, just used where an expression is expected

Anonymous functions

JavaScript allows you to declare anonymous functions

Can be stored as a variable, attached as an event handler, etc.

Keeping unnecessary names out of namespace for performance and safety

```
window.onload = function(){
 alert("Hello World!");
}
```

Arrow Function

An arrow function in JavaScript is a concise way to write anonymous functions, also known as lambda functions or fat arrow functions.

```
(parameter1, parameter2, ..., parameterN) => expression
```

```
const add = (a, b) => a + b;
console.log(add(3, 7)); // Outputs: 10
```

```
const square = (x) => x * x;
console.log(square(5)); // Outputs: 25
```

```
const greet = () => "Hello, world!";
console.log(greet()); // Outputs: Hello, world!
```

Spread Operator

The spread operator allows you to expand an iterable (e.g., an array or an object) into individual elements or properties.

```
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4, 5]; // Combines arrays
console.log(newNumbers); // Outputs: [1, 2, 3, 4, 5]
```

```
const person = { name: "John", age: 30 };
const additionalInfo = { city: "New York", job: "Engineer" };
const mergedPerson = { ...person, ...additionalInfo };
console.log(mergedPerson);
// Outputs: { name: "John", age: 30, city: "New York", job: "Engineer" }
```

What is destructuring assignment?

Special syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const numbers = [10, 20];
let [a, b] = numbers;
console.log(a);
console.log(b);
```

Benefits:

- 'Syntactic sugar' to replace the following:
  - `let a = numbers[0];`
  - `let b = numbers[1];`
- syntax sugar for calling for..of over the value to the right of = and assigning the values.

Destructuring assignment

Unwanted elements of the array can also be thrown away via an extra comma:

```
const [first, , third] = ["foo", "bar", "baz"];
console.log(first);
console.log(third);
```

Can use any “assignables” at the left side.

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');
console.log(user); //{ name: 'John', surname: 'Smith' }
```

Destructuring objects

Destructuring on objects lets you bind variables to different properties of an object.

- Order does not matter

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
let { title, width, height } = options;
alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

Destructure property to another name

to assign a property to a variable with another name, set it using a colon

```
// { sourceProperty: targetVariable }
let { width: w, height: h, title } = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

Creating objects via object literal

```
const name = {
  'fieldName': value,
  ...
  'fieldName': value
};
```

```
const pt = {
  'x': 4,
  'y': 3
};
alert(pt.x + ", " + pt.y);
```

- In JavaScript, you can create a new object without creating a class
- the above is like a Point object; it has fields named x and y
- the object does not belong to any class; it is the only one of its kind, a singleton
- typeof(pt) === "object"

JavaScript objects

- objects in JavaScript are like associative arrays
- the keys can be any string
- you do not need quotes if the key is a valid JavaScript identifier
- values can be anything, including functions
- you can add keys dynamically using associative array or the . syntax
- object properties that have functions as their value are called ‘methods’

```
const x = {
  'a': 97,
  'b': 98,
  'c': 99,
  'd': 199,
  'mult': function(a, b) {
                return a * b;
  }
};
```

Class syntax

```
class MyClass {
    // class methods
    constructor() { ... }
    method1() { ... }
    method2() { ... }
    method3() { ... }  ...
}  //no comma between methods (not an object literal)
```

Then use `new MyClass()` to create a new object with all the listed methods.

The `constructor()` method is called automatically by `new`, so we can initialize the object there.

Problem with ‘this’ inside timeout

- There is a problem if you call a function using ‘`this`’ inside a timeout

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
user.sayHi(); //works
setTimeout(user.sayHi, 2000); //problem!
```

- ‘`this`’ represents the object calling the function
  - `setTimeout` is a global function, which means it is actually a method of `window` (or `global` in Node.js)
  - `user.sayHi` is a reference to the `sayHi` function, it has now been passed as an argument (callback) to the `setTimeout` method, when it is called inside `setTimeout` the lexical context and value of ‘`this`’ will be `window`

Function binding

- When passing object methods as callbacks, for instance to `setTimeout`, there's a known problem: losing "`this`"
- The general rule: '`this`' refers to the object that calls a function
  - since functions can be passed to different objects in JavaScript, the same '`this`' can reference different objects at different times
  - Does not happen in languages like Java where functions always belong to the same object
- `setTimeout` can have issues with '`this`'
  - sets the call context to be `window`

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
setTimeout(user.sayHi, 1000); // Hello, undefined
```

this

In Java, every method has an implicit variable '`this`' which is a reference to the object that contains the method

- Java, in contrast to JavaScript, has no functions, only methods
- So, in Java, it is always obvious what '`this`' is referring to

In JavaScript, '`this`', usually follows the same principle

- Refers to the containing object
- If in a method, refers to the object that contains the method, just like Java
- If in a function, then the containing object is '`window`'
  - Not in "`use strict`" mode→ `undefined`
- Methods and functions can be passed to other objects!!
  - '`this`' is then a portable reference to an arbitrary object

‘this’ keyword inside vs outside object

*

```
function greeting() {
    console.log(this);
}

let  user  = {
    firstName: "John",
    sayHi() {
        console.log(this);
    }
};
console.log(this); // this is window object
greeting(); // greeting() is called by global window object, this is window
user.sayHi(); //sayHi() is called by the object user, this is user
```

Solution 1: a wrapper

```
let user = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};
setTimeout(function() { user.sayHi(); }, 2000); //wrapped versus just “user.sayHi”
//Or
setTimeout(() => user.sayHi(), 2000);
```

- Works because ‘`this`’ references the calling object and now the user object is calling the function
- Closure?
  - free variable?
- This anonymous function wrapper technique can be used whenever you want to pass a function as a callback along with arguments
  - In this case we are, in effect, passing the ‘this’ argument for the function call

.call() .apply() .bind()

There are many helper methods on the Function object in JavaScript

- `.bind()` when you want a function to be called back later with a certain context
- `.call()` or `.apply()` when you want to invoke the function immediately and modify the context.
- Can be used to manually change '`this`' context
- http://stackoverflow.com/questions/15455009/javascript-call-apply-vs-bind

```
var func2 = func.bind(anObject , arg1, arg2, …)// creates a copy
 of func using anObject as 'this' and its first 2 arguments bound
 to arg1 and arg2 values

func.call(anObject, arg1, arg2...);

func.apply(anObject, [arg1, arg2...]);
```

Solution 2~4: `call()` `.apply()` `.bind()`

- several techniques to set the '`this`' context parameter

```
let  user  = {
    firstName: "John",
    sayHi() {
        console.log(`Hello, ${this.firstName}!`);
    }
};

user.sayHi(); //works
setTimeout(user.sayHi, 2000); //problem! - this refers to window object
setTimeout(user.sayHi.bind(user), 2000); //works
setTimeout(() => user.sayHi.call(user), 2000); //works
setTimeout(() => user.sayHi.apply(user), 2000); //works
```

What is a Promise?

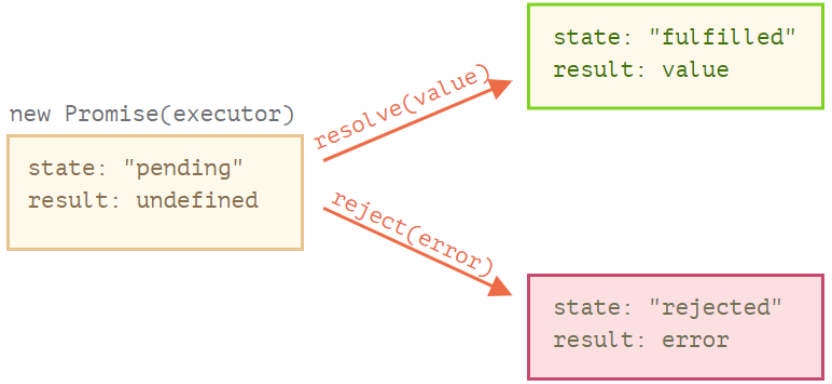
A promise is an object that represents something that will be available in the future. In programming, this "something" is values.

Promises propose that instead of waiting for the value we want (e.g. the image download), we receive something that represents the value in that instant so that we can "get on with our lives" and then at some point go back and use the value generated by this promise.

Promises are based on time events and have some states that classify these events:

- Pending: still working, the result is undefined;
- Fulfilled: when the promise returns the correct result, the result is a value.
- Rejected: when the promise does not return the correct result, the result is an error object.

Create a Promise Object




```
let promise = new Promise(function(resolve, reject) {
    // executor
});
```

The function passed to new Promise is called the **executor**. When new Promise is created, **the executor runs automatically**. Only the parts of resolve and reject are going to be asynchronous.

Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself.

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

- `resolve(value)` — if the job is finished successfully, with result `value`.
- `reject(error)` — if an error has occurred, `error` is the error object.

The promise object returned by the new Promise constructor has these internal properties:

- `state` — initially `"pending"`, then changes to either `"fulfilled"` when `resolve` is called or `"rejected"` when `reject` is called.
- `result` — initially `undefined`, then changes to `value` when `resolve(value)` called or `error` when `reject(error)` is called.

Consumers: `then, catch, finally`

A Promise object serves as a link between the executor and the consuming functions, which will receive the result or error. Consuming functions can be registered (subscribed) using methods `.then`, `.catch` and `.finally`.

```
let promise = new Promise(function(resolve, reject) {
    const random = Math.random();
    console.log('random: ', random);
    if (random > 0.5) {
        setTimeout(() => resolve("done!"), 1000);
    } else {
        setTimeout(() => reject(new Error("Whoops!")), 1000);
    }

});

promise.then(result => console.log(result))
    .catch(error => console.log(error))
    .finally(() => console.log("Promise ready!"));
```

Async/await

It’s a special syntax to work with promises in a more comfortable fashion

The `async` keyword: when you put `async` keyword in front of a function declaration, it turns the function into an `async` function.

The `await` keyword: `await` only works inside `async` functions. `await` can be put in front of any `async` promise-based function to pause your code on that line until the `promise` fulfills, then return the resulting `value`.

Async functions

`async` can be placed before a function. An `async` function always returns a `promise`:

- When no return statement defined, or return without a value. It turns a resolving a promise equivalent to `return Promise.Resolve()`
- When a return statement is defined with a value, it will return a resolving promise with the given return `value`, equivalent to `return Promise.Resolve(value)`
- When an error is thrown, a rejected promised will be returned with the thrown error, equivalent to `return Promise.Reject(error)`

```
console.log('start');
async function f() {
    return 1;
}

f().then(console.log);
console.log('end');
```

Await

The keyword `await` makes JavaScript wait until that `promise` settles and returns its result.

`await` literally suspends the function execution until the `promise` settles, and then resumes it with the `promise` result. That doesn't cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc.

It's just a more elegant syntax of getting the `promise` result than `promise.then`.

```
console.log('start');
async function foo() {
    return 'done!';
}
async function bar() {
    console.log('inside bar - start');
    let result = await foo();
    console.log(result); // "done!"
    console.log('inside bar - end');
}
bar();
console.log('end');
```

Await (cont.)

1. Can't use `await` in regular functions. If we try to use `await` in a non-async function, there would be a syntax error:

```
async function foo() {
    return 'done!';
}

function bar() {
    let result = await foo(); // Syntax error
    console.log(result);
}
bar();
```

2. `await` won't work in the top-level code

```
// syntax error in top-level code
async function baz() {
    return 'baz...';
}

let result = await baz(); //Syntax
Error
console.log(result);
```

Error Handling in Async functions

1. Use `await` inside async function
2. Chain async function call with a `.catch()` call.

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}

async function run() {
    try {
        await thisThrows();
    } catch (e) {
        console.log('Caught the error....');
        console.error(e);
    } finally {
        console.log('We do cleanup here');
    }
}

run();
```

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}

thisThrows()
    .catch(console.error)
    .finally(() => console.log('We do cleanup here'));
```

export

In JavaScript, the `import` and `export` statements are used to define and manage module dependencies, enabling modular code organization and reuse.

Named Exports:

- To export one or more variables, functions, or objects from a module, you can use the export keyword followed by the name of the item you want to export.

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

- You can also export multiple items in a single statement.

```
// math.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
export { add, subtract };
```

Default Exports

You can export a default value (typically a single function, class, or object) from a module using the `export default` syntax.

```
// utility.js
const greet = (name) => `Hello, ${name}!`;
export default greet;
```

Import

Named Imports:

- To import specific items (variables, functions, objects) from another module, use the import statement followed by the item names and the from keyword.

```
// app.js
import { add, subtract } from './math.js';

const result1 = add(5, 3);
const result2 = subtract(8, 2);
```

Default Imports:

- When importing a default export, you can use any name you prefer for the imported item.

```
// main.js
import greeting from './utility.js';

const message = greeting('Alice');
```