CS5220 Advanced Topics in Web Programming

More Node.js

Chengyu Sun
California State University, Los Angeles

Overview

- More on object, class, and inheritance
- Using modules
- Asynchronous programming
  - Callback
  - Promise

Prototype Inheritance

```
let honda = { make: 'Honda' };

let civic = Object.create( honda );
civic.model = 'Civic';

console.log( civic );

console.log( civic.make );
console.log( civic.model );
console.log( honda.model );
```

- `honda` is `civic`'s prototype

Understand Properties in Prototype Inheritance

- When we try to *read* a property and it does not exist in the current object, JavaScript looks for it in the prototype
- When we try to *write* a property and it does not exist in the current object, a new property in the current object is created

Inherited Methods

```
honda.owner = 'Car Dealer';
honda.printOwner = function() {
  console.log(this.owner);
}

civic.printOwner(); //??

civic.owner = 'Chengyu';
console.log(civic.owner); //??
console.log(honda.owner); //??
```

More on `this`

- The value of `this` in a regular function is determined at runtime (i.e. *runtime binding*)
  - In a function call, `this` is bound to the global object in non-strict mode, and undefined in strict mode
  - In a method call, this is bound to the object the method is called on, not the object where the method is defined
  - See more on MDN
- An arrow function does not have its own `this` (i.e. it get `this` from outside)

Don't Use Arrow Functions for Methods ...

```
let car = {
  make: 'Honda',
  model: 'Civic'
};

car.print = () => console.log(
  `${this.make},${this.model}`);


car.print(); //??
```

… Don't Use Arrow Function for Methods

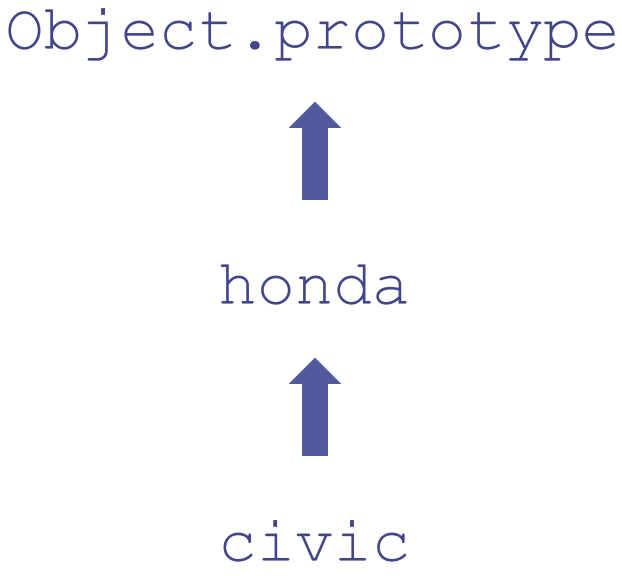
```
function Car(model, make) {
  this.model = model;
  this.make = make;
  this.print = () => console.log(
    `${this.make},${this.model}`);
}

let car1 = new Car("Honda", "Civic");
car1.print(); //??

let car2 = Object.create(car1);
car2.model = "Ford";
car2.make = "F150";
car2.print(); //??
```

[[Prototype]] and __proto__

- Each object has a hidden property `[[Prototype]]` that references its prototype
- `[[Prototype]]` can be accessed via the accessor property `__proto__`



What About `prototype`?

- Like `Object.prototype`, `Array.prototype`, `Date.prototype` ...?
- `Object`, `Array`, `Date` ... are *constructors,* i.e. functions
- `<constructor>.prototype` is used to set the prototype of any object created by the constructor

Create "Class" Using `prototype` ...

- Example: a Java class

```
class Cirlce {

    public static double PI = 3.1415;

    private double radius;

    public Circle(double radius) {
        this.radius = radius; }

    public double area() {
        return PI*radius*radius; }
}
```

… Create "Class" Using `prototype` …

```
function Circle(radius) {
  this._radius = radius;
}
```

- By convention, use PascalCasing for constructor name
- By convention, start the name of an internal property with _

... Create "Class" Using `prototype`

```
Circle.PI = 3.1415;

Circle.prototype.area =
  function() {
    return Circle._PI*
      this._radius*this._radius;
  };
```

- "Static" properties are properties of the constructor
- "Instance" properties are properties of the prototype

Compare Two "Class Patterns"

```
function Circle(radius) {
  this._radius = radius;
  this.diameter = function(){
    return this._radius*2;
  }
}
```

- Define `diameter()` on prototype or in constructor??

ES6 Class Syntax

```
const PI = 3.1415;
class Circle {
  constructor(radius) {
    this._radius = radius;
  }
  area() {
    return PI*
    this._radius*this._radius;
  }
}
```

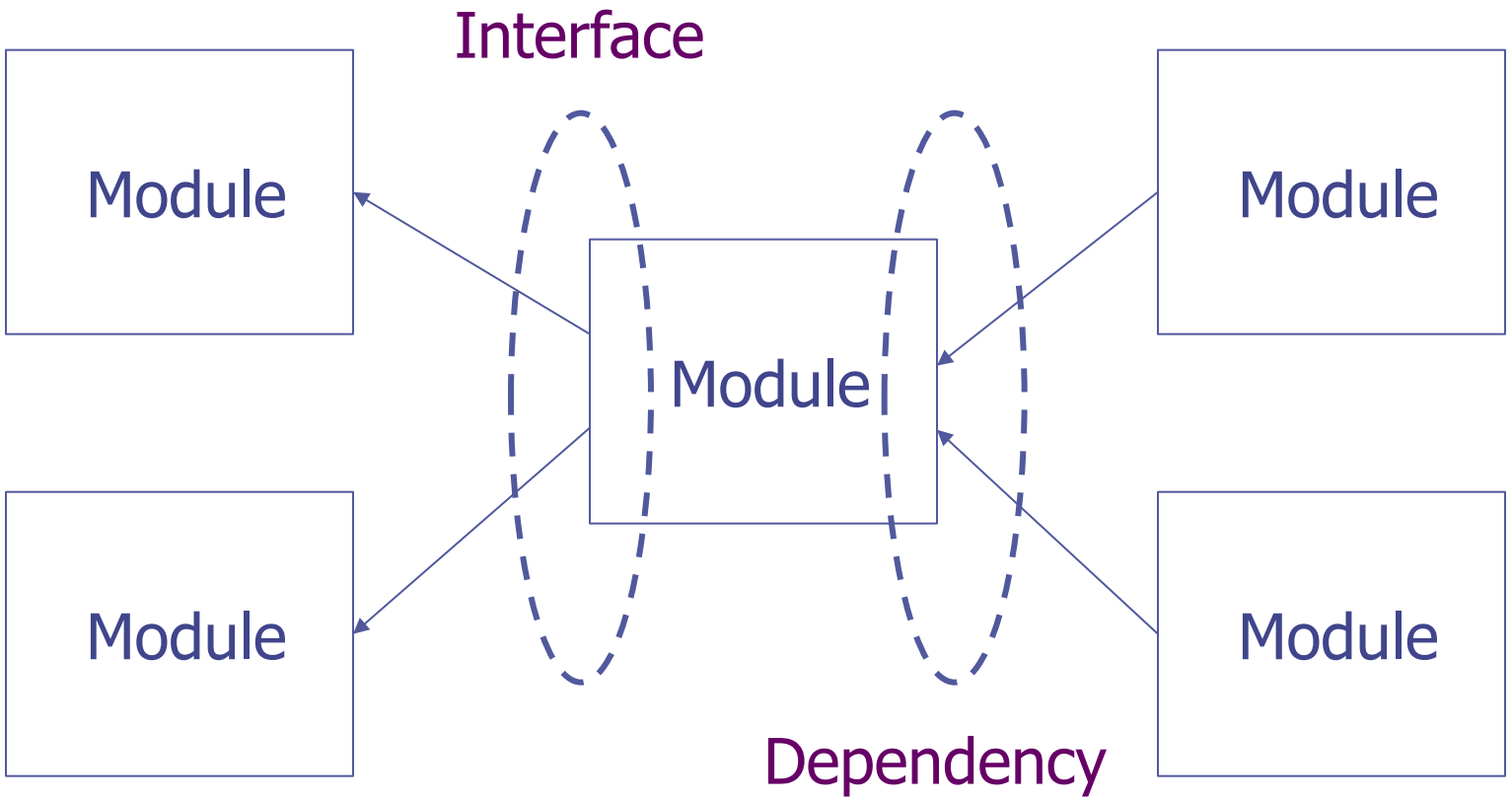
About ES6 Class Syntax

- A constructor followed by methods
  - Why only one constructor??
  - How about overloading methods??
- Can have `static` methods but not static fields
  - Later specifications introduce more features
- More than *syntax sugar*
  - Make class inheritance easier (among other things)

ES6 Class Inheritance

```
class BigCircle extends Circle {
  constructor(radius, factor) {
    super(radius);
    this._factor = factor;
  }
  area() {
    return super.area() *
    this._factor;
  }
}
```

Modules



CommonJS Module System

- Each file is a module
- Dependencies are loaded by calling the function `require()`
- Interface is exposed through the `module.exports` object

CommonJS Module Example

a.js

```
const foo = "foo";

function bar() {
  console.log(foo);
}

module.exports = {
  foo,
  bar
};
```

b.js

```
const a =
  require("./a");

a.bar();
```

More About `require()`

- Loads the module and returns the module object
- A module is only loaded once with multiple `require()` calls
- Core modules and NPM-installed modules are loaded by name, e.g. `require('http')`
- File and folder modules are loaded by path
  - Path can either absolute or relative (starting with `./` or `../`); `.js` extension can be omitted

ES6 Module Examples ...

a.js

```
export const foo
  = "foo";

export function
  bar() {
  console.log(foo);
}
```

b.js

```
import * as a
  from "./a";

a.bar();
```

… ES6 Module Examples

```
import {foo, bar} from "./a";

import {bar} from "./a";

import {bar as myBar} from "./a";
```

Default Export in ES6 Module

a.js

```
default export const foo = "foo";

export function bar() {
  console.log(foo);
}
```

```
import bar from "./a";

import foo, * as a from "./a";

import foo, {bar} from "./a";
```

Synchronous Programming

- Some tasks may take a very long time, e.g. read/write a file on disk, request data from a server, query a database …
- *Why waiting is bad??*

```
some_task(); /*
  wait for task to
  complete
*/

// process the result
// of task
… …

// do other things
… …
```

Multi-processing and Multi-threading

One process/thread

```
some_task();

// process the result
// of task
… …
```

Another process/thread

```
// do other things
… …
```

- What's the different between a process and a thread??

Problems of Multi-processing and Multi-threading

- OS must allocate some resources for each process/ thread
- Switching between processes and threads (a.k.a. *context switch*) takes time
- Communicating among processes and synchronizing multiple threads are difficult

Big problems for busy web servers

One of the reasons why Node.js became popular in server-side development

Asynchronous Programming …

```
callback(result) {
  // process the result
  // of task
  … …
}

some_task( callback ); /*
  calls to some_task()
  returns immediately
*/

// do other things
… …
```

… Asynchronous Programming

- Everything runs in one thread
- Asynchronous calls return immediately (a.k.a. *non-blocking*)
- A callback function is called when the result is ready
- A.K.A. Event-driven Programming
  - A callback function is basically an event handler that handles the "result is ready" event

Asynchronous Programming Example 1

- `get_page.js`: request and print a web page
  - Use of the request package
  - Error-first callback style

Asynchronous Programming Example 2

- `write_file.js`: open a file, add one line, then close the file
  - Use of the File System package
    - open()
    - write()
    - close()
  - "Callback Hell" (a.k.a. "Pyramid of Doom")

Promise

- A Promise is a JavaScript object
  - `executor`: a function that may take some time to complete. After it's finished, it sets the values of `state` and `result` based on whether the operation is successful
  - `state`: "pending" ➔ "fulfilled"/"rejected"
  - `result`: undefined ➔ value/error

Use A Promise

```
promise.then(
   function(result) {/* handle result */},
   function(err) {/* handle error */ }
);
```

- After `executor` finishes, either the success handler or the error handler will be called and `result` will be passed as the argument to the handler

Other Common Usge of Promise

```
promise.then( success_handler );

promise.then( null, error_handler );

promise.catch( error_handler );

promise
   .then( success_handler )
   .catch( error_handler )
```

Promise Example 1

- `get_page_promise.js`: request and print a web page
  - Use of the request-promise-native package
  - `request()` returns a *promise*
- Two commonly used Promise implementations
  - ES6 (i.e. native)
  - bluebird has more features (e.g. `finally`) and better performance in some situations

About Promise

- There can only be one result or an error
- Once a promise is settled, the result (or error) never changes
- `then()` can be called multiple times to register multiple handlers

Promises Chaining

- Suppose we have three functions `f1`, `f2`, `f3`
  - `f1` returns a Promise
  - `f2` relies on the result produced by `f1`
  - `f3` relies on the result produced by `f2`

```
f1.then(f2).then(f3)
```

Understand Promise Chaining ...

```
f1.then(f2)
```

- then() returns a Promise based on the return value of the handler function
  - If `f2` return a regular value, the value becomes the result of the Promise
  - If `f2` return a promise, the result of that Promise becomes the result of the Promise returned by `then()`

… Understand Promising Chaining

```
f1.then(f2).then(f3)
```

- The result of `f1` is passed to `f2`
- The result of `f2` is passed to `f3`

Promise Example 2

- `write_file_promise.js`: open a file, add one line, then close the file
  - Use of promisify() in the Utilities package

Original function:

`func(args…, callback(err,result))`

Promisified:

`func(args…)` returns a Promise

Parallel Execution

```
Promise.all([promise1, promise2 …]).then(
  function(results) {
    // results is an array of values, one
    // by each promise
  }
)

Promise.race([promise1, promise2 …]).then(
  function(result) {
    // result is the result of the promise
    // that settles first
  }
)
```

Create a Promise

```
new Promise(function(resolve,reject){…})
```

- `resolve` and `reject` are two functions passed by JavaScript engine
  - Call `resolve(value)` to set result to value and state to "fulfilled"
  - Call `reject(error)` to set result to error and state to "rejected"
- More details at Promise Basics

async and await

- `async` declares a function to be asynchronous
  - The return value of the function will be wrapped inside a Promise
- `await` waits until a Promise settles and returns its result
  - *`await` can only be used in an `async` function*

Promise Example 3

- `write_file_async.js`: open a file, add one line, then close the file
  - Use of `async` and `await`
  - What about error??