Metodologie per la Programmazione per il Web - MF0437
Javascript - terza parte

Docente

Giancarlo **Ruffo**[giancarlo.ruffo@uniupo.it]

Informazioni, materiale e risorse su:

moodle [https://www.dir.uniupo.it/course/view.php?id=16455]

Slide adattate da una versione precedente a cura del Prof. Alessio Bottrighi

Asynchronous Programming

JavaScript: The Definitive Guide, 7th Edition
Chapter 11. Asynchronous JavaScript

Mozilla Developer Network

- Learn web development JavaScript » Dynamic client-side scripting » Asynchronous JavaScript
- Web technology for developers » JavaScript » Concurrency model and the event loop
- Web technology for developers » JavaScript » JavaScript Guide » Using Promises

Asyncronicity

- JavaScript is ***single-threaded*** and inherently synchronous
  - i.e., code cannot create threads and run in parallel in the JS engine
- **Callbacks** are the most fundamental way for writing asynchronous JS code
- How can they work asynchronously?
  - e.g., how can `setTimeout()` or other async callbacks work?
- Thanks to the **Execution Environment**
  - e.g., browsers and Node.js
- and the *Event Loop*

```
const deleteAfterTimeout = (task)
=> {
  // do something
}
// runs after 2 seconds
setTimeout(deleteAfterTimeout,
2000, task)
```

Non-Blocking Code!

- Asynchronous techniques are very useful, particularly for web development
- For instance: when a web app runs executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen
  - this is called **blocking**, and it should be the **exception!**
    - the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor
- This may happen ***outside*** browsers, as well
  - e.g., reading a long file from the disk/network, accessing a database and returning data, accessing a video stream from a web cam, etc.
- Most of the JS execution environments are, therefore, deeply ***asynchronous***
  - with ***non-blocking*** primitives
  - JavaScript programs are ***event-driven***, typically

Events

- JavaScript may handle ***hundreds*** of known events, e.g.,
  - a timer -> generates an event when the related timeout expires
  - a network request -> generates an event when the response from the network is received
  - a button -> generates an event when the user clicks on it
- Events can be handled by
  - **pre-defined** behaviors (by Node or the browser)
  - **user-defined** *event handlers* (in your code)
- or just **ignored**, if no event handler is defined

Event Loop

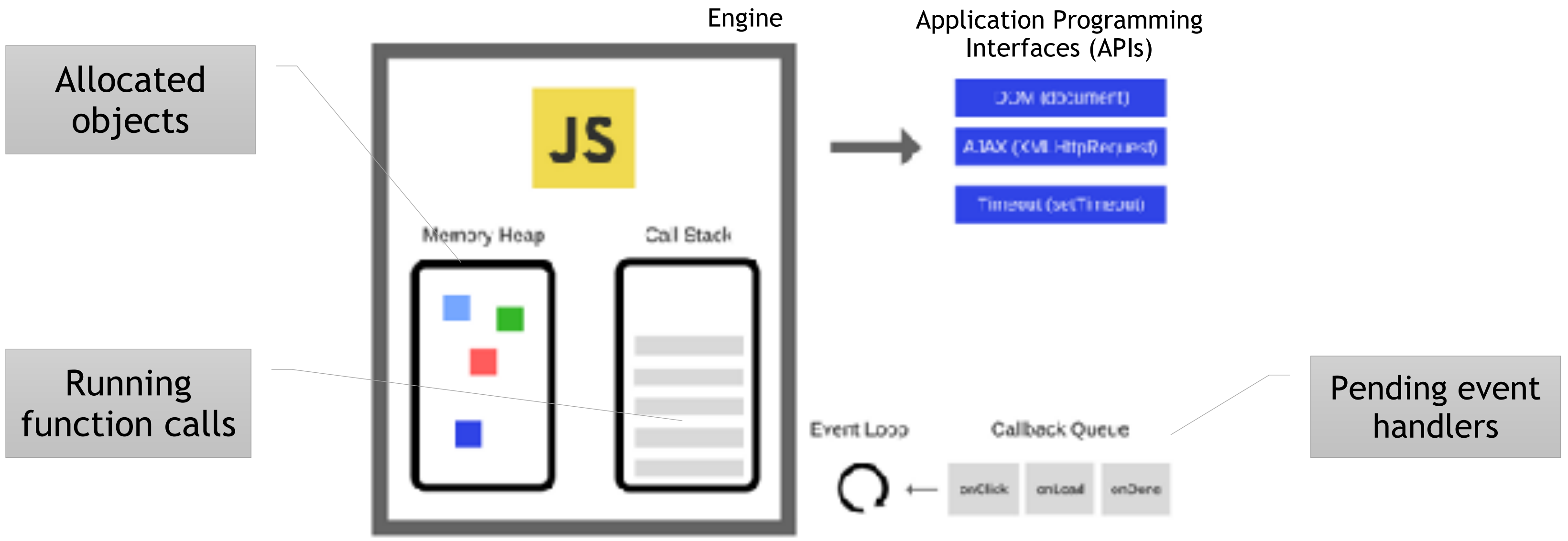
- But JavaScript is single-threaded!
  - event handling is synchronous and is based on an **event loop**
  - event handlers are queued on a ***Message Queue*** when the corresponding event happens
  - the Message Queue is ***polled*** when the main thread is idle
- How the event loop works is ***defined*** in the JavaScript specs
  - but it is ***implemented*** in the execution environments

"The event loop is what allows an execution environment to perform non-blocking operations - even though JavaScript is single-threaded - by offloading operations to the system kernel whenever possible"

Want to know more about the Event Loop?
https://www.youtube.com/watch?v=8aGhZQkoFbQ

Execution Environments: Runtime (from 10,000 feet)

The loop gives priority to the call stack, and it first processes everything in the call stack, and once there is nothing in there, it goes to pick up things in the Message Queue.



https://github.com/rohit120582sharma/Documentation/wiki/An-overview-of-JavaScript-Engine,-Runtime,-and-Call-Stack

Execution Environments: Runtime (from 10,000 feet)

- Let's try it: Philip Roberts' visualizer
- During code execution you may
  - **call** functions -> the function call is pushed to the call stack
  - **schedule** events -> the call to the event handler is put in the Message Queue
- Remind: events may be scheduled by external events, too
  - e.g., user actions, I/O, network, timers, …
- At any step, the JS interpreter:
  - if the call stack is not empty, pop the top of the stack and executes it
  - if the call stack is empty, pick the head of the Message Queue and executes it
- A function call or an event handler is **never** interrupted
  - again, avoid blocking code!

Back to Callbacks

- The most ***fundamental*** way for writing asynchronous JS code
- Great for "simple" things!

```
const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

rl.question('What is your age? ', (age) => {
    console.log('Your age is: ' + age);
    rl.close();
});
```

Handling Errors in Callbacks

- No "official" ways, only ***best practices***!
- Typically, the first parameter of the callback function is for storing any ***error***, while the second one is for the result of the operation
  - this is the strategy adopted by Node.js, for instance

```
fs.readFile('/file.json', (err, data) => {
  if (err !== null) {
    console.log(err);
    return;
  }
  //no errors, process data
  console.log(data);
})
```

Beware: Callback Hell!

- If you want to perform ***multiple asynchronous actions in a row*** using callbacks, you must keep passing new functions to handle the continuation of the computation after the previous action
  - every callback adds a level of nesting
  - when you have lots of callbacks, the code starts to be complicated very quickly

```
const readline = require('readline');
const rl = readline.createInterface(...);

rl.question('Task description: ', (answer) => {
  let description = answer;

  rl.question('Is the task important? (y/n)', (answer) => {
    let important = answer;

    rl.question('Is the task private? (y/n)', (answer) => {
      let private = answer;

      rl.question('Task deadline: ', (answer) => {
        let date = answer;

        ...

      }
    }
  }

  rl.close();

});
```

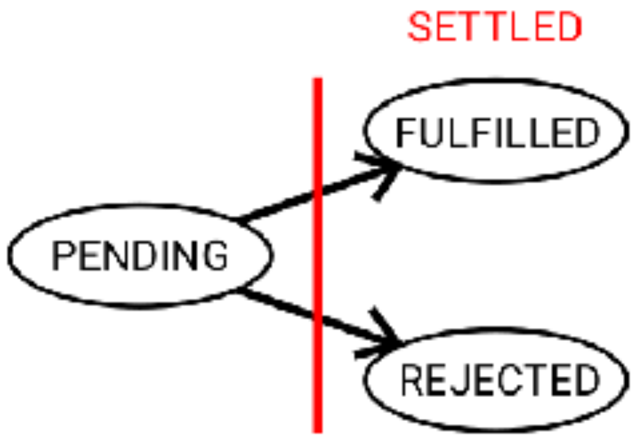
```
const readline = require('readline');
const rl = readline.createInterface(...);

rl.question('Task description: ', (answer) => {
  let description = answer;
  rl.question('Is the task important? (y/n)', (answer) => {
          let important = answer;
          rl.question('Is the task private? (y/n)', (answer) => {
                    let private = answer;
                    rl.question('Task deadline: ', (answer) => {
                    let date = answer;
                              ...
      }
    }
  }

  rl.close();
});
```

Promises

- A core language feature to **simplify** asynchronous programming
  - a possible solution to callback hell, too!
  - a fundamental building block for "newer" functions (async, ES2017)
- It is an **object** representing the eventual completion (or failure) of an asynchronous operation
  - i.e., an asynchronous function returns a ***promise*** to supply the value at some point in the future, instead of returning immediately a final value
- Promises standardize a way to **handle errors** and provide a way for errors to **propagate correctly** through a chain of promises

Promises

- Promises can be ***created*** or ***consumed***
  - lots of Web APIs expose Promises to be consumed!
- When consumed:
  - a Promise starts in a **pending** state
    - the caller function continues the execution, while it waits for the Promise to do its own processing, and give the caller function some "responses"
  - then, the caller function waits for it to either return the promise in a **fulfilled** state or in a **rejected** state



Promises

- Promises can be *created* or *consumed*
  - lots of Web APIs expose Promises to be consumed!
- When consumed:
  - a Promise starts in a **pending** state
    - the caller function continues the execution, while it waits for the Promise to do its own processing, and give the caller function some "responses"
  - then, the caller function waits for it to either return the promise in a **fulfilled** state or in a **rejected** state

```
let duration = 10;

const waitPromise = new Promise((resolve,
reject) => {
  if (duration >= 0) {
  // the promise can be fulfilled!
    resolve("It works!");
  } else {
    // time travel? we reject the promise
    reject(new Error("It doesn't work."));
  }
});

waitPromise.then((result) => {
  console.log("Success: ", result);
}).catch((error) => {
  console.log("Error: ", error);
});
```

Creating a Promise

- A *Promise* object is created using the `new` keyword and its constructor
- The constructor takes an "*executor function*", as its parameter
- This function takes two functions as parameters:
  - **resolve**, called when the asynchronous task completes successfully and returns the results of the task as a value
  - **reject**, called when the task fails and returns the reason for failure (an error object, typically)

```
const myPromise = new Promise((resolve, reject) => {

// do something asynchronous which eventually calls either:

// resolve(someValue); // fulfilled

// or

// reject("failure reason"); // rejected });
```

Creating a Promise

- You can also provide a function with "promise functionality"
- Simply have it **return a promise**!

```
function wait(duration) {
  // Create and return a new promise
  return new Promise((resolve, reject) => {

  // If the argument is invalid, reject the promise
  if (duration < 0) {
    reject(new Error('Time travel not yet implemented'));
  }

  // otherwise, wait asynchronously and then resolve the Promise
  // setTimeout will invoke resolve() with no arguments:
  // the Promise will fulfill with the undefined value
  setTimeout(resolve, duration);
  });
}
```

Consuming a Promise

- When a Promise is **fulfilled**, the `then()` callback is used
- If a Promise is **rejected**, instead, the `catch()` callback will handle the error
- `then()` and `catch()` are *instance methods* defined by the Promise object
  - each function registered with then() is invoked only once
- You can *omit* `catch()`, if you are interested in the result, only

```
waitPromise.then((result) => {
  console.log("Success: ", result);
}).catch((error) => {
  console.log("Error: ", error);
});

// if a function returns a Promise...
wait(1000).then(() => {
  console.log("Success!");
}).catch((error) => {
  console.log("Error: ", error);
});
```

Consuming a promise

- `p.then(onFulfilled[, onRejected]);`
  - callbacks are executed asynchronously (inserted in the event loop) when the promise is either fulfilled (success) or rejected (optional)
- `p.catch(onRejected);`
  - callback is executed asynchronously (inserted in the event loop) when the promise is rejected
- `p.finally(onFinally);`
  - callback is executed in any case, when the promise is either fulfilled or rejected.
  - useful to avoid code duplication in then and catch handlers
- All these methods return Promises, too!

Chaining Promises

- One of the most important benefits of Promises
- They provide a natural way to express a sequence of asynchronous operations as a **linear** chain of `then()` invocations
  - without having to nest each operation within the callback of the previous one
    - the "callback hell" seen before
- **Important:** always `return` results, otherwise callbacks won't get the result of a previous promise

```
getRepoInfo()
  .then(repo => getIssue(repo))
  .then(issue =>
getOwner(issue.ownerId))
  .then(owner =>
sendEmail(owner.email, 'Some text'))
  .catch(e => {
    // just log the error
    console.error(e);
});
```

Promises… in Parallel

```
Promise.all(promises)
  .then(results => {
    console.log(results);
  })
  .catch(e => {console.error(e)});
```

- What if we want to execute several asynchronous operations in **parallel**?
- `Promise.all()`
  - takes an ***array*** of Promise objects as its input and returns a Promise
  - the returned Promise will be ***rejected*** if any of the input Promises are rejected
  - otherwise, it will be ***fulfilled*** with an array of the fulfillment values of each of the input promises
  - the input array can contain ***non-Promise values***, too: if an element of the array is not a Promise, it is simply copied unchanged into the output array
- `Promise.race()`
  - returns a Promise that is fulfilled or rejected when the first of the Promises in the input array is fulfilled or rejected
  - if there are any non-Promise values in the input array, it simply returns the first of those

async/await (ES8)

- Two new keywords: **`async`** and **`await`**
  - they simplify the ***use*** of Promises and allow us to write Promise-based asynchronous code that **looks like** synchronous (blocking) code
  - code is easier to debug, too
- **`async`**
  - prepending the keyword to any function means that it will return a promise

```
const sayHello = async () => "hello";
sayHello().then(console.log); // this will log "hello"
```

async/await (ES8)

- `await`
  - takes a Promise and turns it back into a return value or a thrown exception
  - it can be used with any asynchronous function that returns a Promise
- One **critical** rule: you can **only** use the `await` keyword within functions that have been declared with the `async` keyword
  - i.e., it cannot be used in top-level code

```
async ...
  const greetings = await sayHello();
```

Example: async / await

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}
async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();
```

Return a promise

async is needed to use `await`

Looks like sequential code

```
> "calling"
//... 2 seconds
> "resolved"
```

Example: async / await

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}
async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  return 'end';
}

asyncCall().then(console.log);
```

Implicitly returns a `Promise`

Can use Promise methods

```
> "calling"
//... 2 seconds
> "end"
```

Example... Before and After

```
const makeRequest = () => {
  return getAPIData()
    .then(data => {
      console.log(data);
      return "done";
    }
  );
}

let res = makeRequest();
```

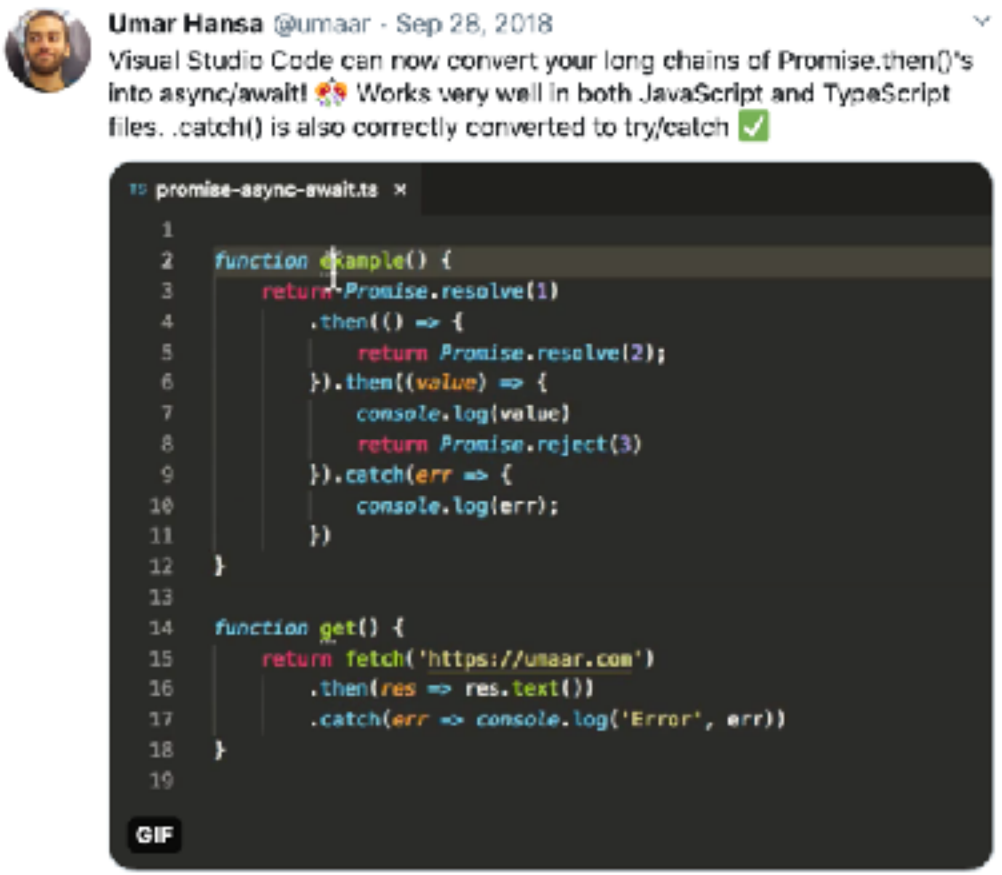
```
const makeRequest = async () => {
  console.log(await getAPIData());
  return "done";
};

let res = makeRequest();
```

Chaining Example... Before and After

```
function getData() {
  return getIssue()
    .then(issue =>
getOwner(issue.ownerId))
    .then(owner => sendEmail(owner.email,
'Some text'));
}

// assuming that all the 3 functions above
return a Promise
```

```
async function getData {
  const issue = await getIssue();
  const owner = await
getOwner(issue.ownerId);
  await sendEmail(owner.email, 'Some
text');
}
```

Converting Promise-based Function to async/await with Visual Studio Code



Umar Hansa @umaar · Sep 28, 2018

Visual Studio Code can now convert your long chains of Promise.then()'s into async/await! 🎊 Works very well in both JavaScript and TypeScript files. .catch() is also correctly converted to try/catch ✅

```

function example() {
    return Promise.resolve(1)
        .then(() => {
            return Promise.resolve(2);
        }).then((value) => {
            console.log(value)
            return Promise.reject(3)
        }).catch(err => {
            console.log(err);
        })
}

function get() {
    return fetch('https://umaar.com')
        .then(res => res.text())
        .catch(err => console.log('Error', err))
}

```

https://twitter.com/i/status/1045655069478334464

Promises or async/await? Both!

- If the output of `function2` is dependent on the output of `function1`, use `await`.
- If two functions can be run in parallel, create two different `async` functions and then run them in parallel with `Promise.all(promisesArray)`
- Instead of creating huge `async` functions with many `await asyncFunction()` in it, it is better to create **smaller** `async` functions
- If your code contains blocking code, it is better to make it an `async` function. The callers can decide on the level of asynchronicity they want

https://medium.com/better-programming/should-i-use-promises-or-async-await-126ab5c98789

Functional Programming

JavaScript: The Definitive Guide, 7th Edition
Chapter 6. Array
Chapter 7.8 Functional Programming

Functional Programming: A Brief Overview

- A programming paradigm where the developer mostly construct and structure code using ***function***
  - not JavaScript's main oriented paradigm, but JavaScript is well suited
- Three notable features of the paradigm:
  - functions are ***first-class*** citizen
    - functions can be used as if they were variables or constants, combined with other functions and generate new functions in the process, chained with other functions, etc.
  - ***higher-order functions***
    - a function that operates on functions, taking one or more functions as arguments and typically returning a new function
  - function ***composition***
    - composing/creating functions to simplify and compress your functions by taking functions as an argument and return an output

Functional Programming in JavaScript

- JavaScript supports the features of the paradigm "out of the box"
- Even with a "traditional" approach, using functional techniques can increase the readability of your code
  - e.g., replacing procedural loops with their functional version
- Functional programming is about ***avoiding mutability***
  - i.e., do not change objects in place!
  - e.g., if you need to perform a change in an array, return a new array
- We are going to have a look at some of the most used and useful techniques and methods for functional programming in JavaScript

Iterating over Arrays

- Iterators: `for ... of`, `for (..;..;..)`

Functional style

- Iterators: `forEach(f)`
  - f is a function that processes the element
- Iterators: `every(f)`, `some(f)`
  - f is a function that returns true or false
- Iterators that return a new array: `map(f), filter(f)`
  - `f` **works on the element of the array passed as parameter**
- Reduce: exec a callback function on all items to progressively compute a result.

.forEach()

- `forEach()` invokes your (synchronous) function once on each element of an array
  - it always returns ***undefined*** and is <u>not</u> chainable
  - there is no way to stop or break a `forEach()` loop other than by throwing an exception
- `forEach()` does not mutate the array on which it is called
  - however, its callback ***may*** do so

```
const letters = [..."Hello world"];
let uppercase = ""
letters.forEach(letter => {
  uppercase += letter.toUpperCase();
});
console.log(uppercase); // HELLO WORLD
```

.every()

- `every()` tests whether all elements in the array pass the test implemented by the provided function
  - it returns a Boolean value
  - it executes its callback once for each element present in the array until it finds the one where the callback returns a falsy value
    - if such an element is found, immediately returns false

```
let a = [1, 2, 3, 4, 5];

a.every(x => x < 10) // => true: all values are < 10

a.every(x => x % 2 === 0) // false
```

.some()

- `some()` tests whether at least one element in the array passes the test implemented by the provided function
  - it returns a Boolean value
  - it executes its callback once for each element present in the array until it finds the one where the callback returns a truthy value
    - if such an element is found, immediately returns true

```
let a = [1, 2, 3, 4, 5];

a.some(x => x%2===0) // => true; a has some even numbers

a.some(isNaN)
```

.map()

- `map()` passes each element of the array on which it is invoked to the function you specify
  - it always returns a ***new*** array containing the values returned by the callback
  - the function you pass should return a value

```
const a = [1, 2, 3];

b = a.map(x => x*x);

console.log(b); // [1, 4, 9]
```

```
const letters = [..."Hello world"];

uppercase = letters.map(letter =>
letter.toUpperCase());

console.log(uppercase.join(''));
```

.filter()

- `filter()` creates a *new* array with all elements that pass the test implemented by the provided function
  - if no element pass the test, an empty array is returned
  - the callback it should be predicate: a function that returns true or false

```
const a = [5, 4, 3, 2, 1];

a.filter(x => x < 3); // generates [2, 1], values less than 3

a.filter((element, index) => index%2 == 0); // [5, 3, 1]
```

.reduce()

- `reduce()` combines the elements of an array, using the specified function, to produce a *single* value
  - this is a common operation in functional programming and goes by the names "inject" and "fold"
- `reduce()` takes two arguments:

1. the function that performs the reduction/combination operation (combine or reduce 2 values into 1)
2. an (optional) initial value to pass to the function; if not specified, it uses the first element of the array as initial value

.reduce()

- Callbacks used with `reduce()` are different than the ones used with `forEach()` and `map()`
  - the *first* argument of the callback (*reducer function*) is the accumulated result of the reduction so far
  - on the first call to this function, its first argument is the initial value
  - on subsequent calls, it is the value returned by the previous invocation of the reducer function

.reduce()

.reduce(callback(accumulator, currentValue, [, index[, array]])[, initialValue])

- ***accumulator*** accumulates callback's return values. It is the accumulated value previously returned in the last invocation of the callback—or initialValue, if it was supplied (see below).
- ***currentValue***: the current element being processed in the array.
- ***index*** (Optional) of the current element being processed in the array. Starts from index 0 if an initialValue is provided. Otherwise, it starts from index 1.
- ***array*** (Optional) was called upon.
- ***initialValue*** (Optional): a value to use as the first argument to the first call of the callback. If no *initialValue* is supplied, the first element in the array will be used as the initial accumulator value and skipped as currentValue. Calling reduce() on an empty array without an *initialValue* will throw a TypeError.

```
const a = [5, 4, 3, 2, 1];

a.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
// => 15; the sum of the values

a.reduce((acc, val) => acc*val, 1);
// => 120; the product of the values

a.reduce((acc, val) => (acc > val) ? acc : val);
// => 5; the largest of the values
```

Chaining Functions: Example

```
let shoppingCart = [{ productTitle: "Functional Programming", type:
"books", amount: 10 }, { productTitle: "Kindle", type: "electronics",
amount: 30 },{ productTitle: "Shoes", type: "fashion", amount: 20 },
{ productTitle: "Clean Code", type: "books", amount: 60 }];

const byBooks = (order) => order.type == "books";

const getAmount = (order) => order.amount;

const sumAmount = (acc, amount) => acc + amount;

function getTotalAmount(shoppingCart) {

  return shoppingCart.filter(byBooks)

    .map(getAmount)

    .reduce(sumAmount, 0); }

getTotalAmount(shoppingCart); // 70
```

source: https://www.freecodecamp.org/news/functional-programming-principles-in-javascript-1b8fc6c3563f/

Chaining Functions: Example

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

const averageSUVPrice = vehicles
  .filter(v => v.type === 'suv')
  .map(v => v.price)
  .reduce((sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice); // 33399
```

source: https://opensource.com/article/17/6/functional-javascript

Where To Go From Here…

- Going deeper in (or "enforcing") functional programming in JavaScript is out of scope for this course
- "JavaScript: The Definitive Guide, 7th Edition", chapter 7.8 provides some additional pointers
- Other interesting links:
  - https://www.freecodecamp.org/news/functional-programming-principles-in-javascript-1b8fc6c3563f/
  - https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0

Classes and Modules

JavaScript: The Definitive Guide, 7th Edition
Chapter 8. Classes

Mozilla Developer Network
- Learn web development JavaScript » Dynamic client-side scripting » Introducing JavaScript objects
- Web technology for developers » JavaScript » JavaScript reference » Classes
- Web technology for developers » JavaScript » JavaScript Guide » JavaScript Modules

A Prototype-based Language

- JavaScript is an object-based language based on **prototypes**, rather than being class-based
  - classes ***exist*** but they are "syntactical sugar", primarily
- Every JavaScript object has a hidden property (`[[Prototype]]`) that <u>points</u> to a ***second*** object associated with it (or it is null)
- This second object is known as an **object prototype**, and the first object inherits all the properties (including functions) from the prototype
  - this prototype may also have another prototype object, which it inherits methods and properties from, and so on
  - properties are ***not*** copied between objects, but directly accessed

Prototypes Inheritance

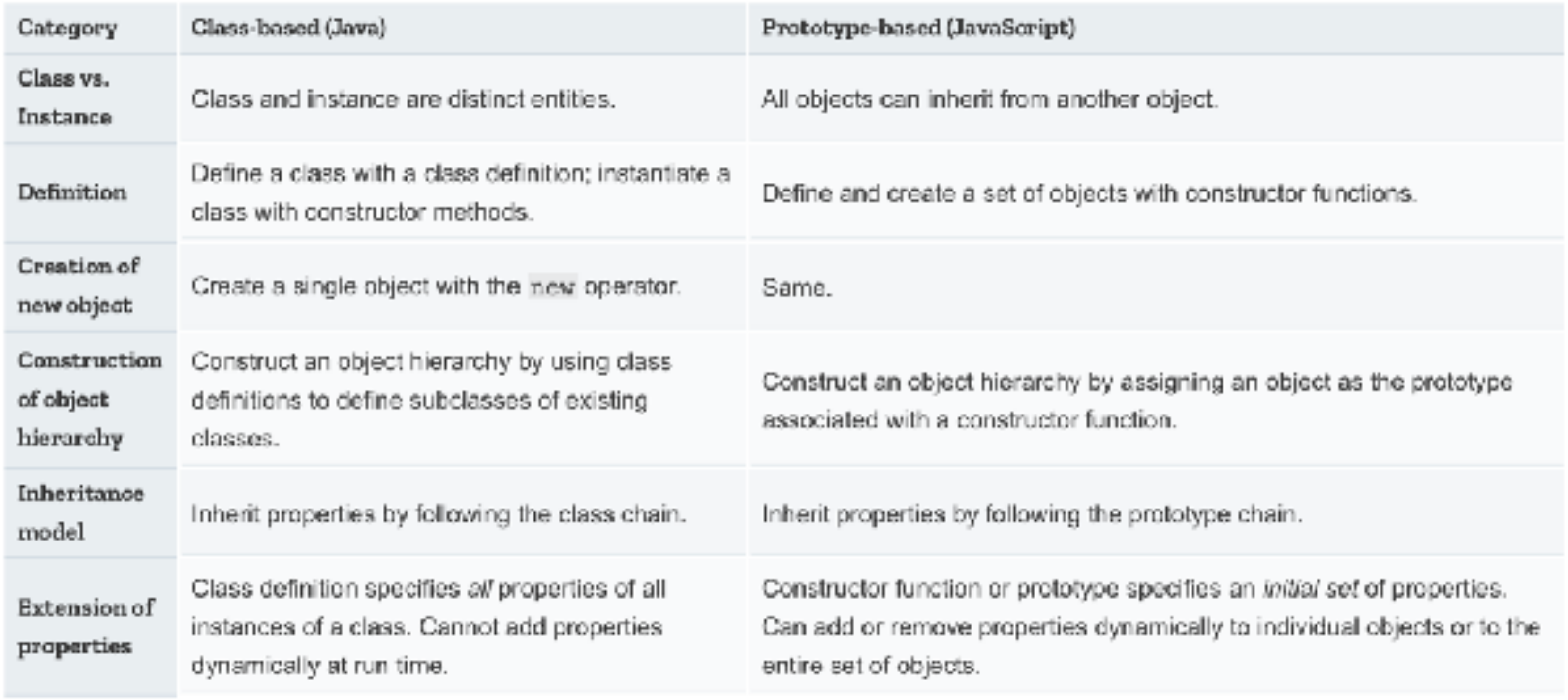
- All objects created by object literals (i.e., {}) have the *same* prototype object: `Object.prototype`
- **`Object.prototype` is one of the rare objects that has no prototype**
  - it does not inherit any properties
- Most built-in constructors have a prototype that inherits from `Object.prototype`
  - for instance, `Date.prototype` inherits properties from `Object.prototype`
  - a `Date` object created by `new Date()` inherits properties from both `Date.prototype` and `Object.prototype`
  - this is called **prototype chain**

Prototypes Inheritance

- To know which prototype(s) an object has, you can use `Object.getPrototypeOf()`
- Given an object, the properties that will be inherited from it are specified by the `prototype` property of the object constructor
  - that is why, e.g., Arrays inherits `toString()` but not `entries()` from Objects

```
let date = new Date();
let list = [];
let obj = {};
Object.getPrototypeOf(obj);
// {}
Object.getPrototypeOf(list);
// []
Object.getPrototypeOf(date);
// Date {}
Object.getPrototypeOf(Array.prototype);
// {}
```

Class-based vs. Prototype-based Languages



Category	Class-based (Java)	Prototype-based (JavaScript)
Class vs. Instance	Class and instance are distinct entities.	All objects can inherit from another object.
Definition	Define a class with a class definition; instantiate a class with constructor methods.	Define and create a set of objects with constructor functions.
Creation of new object	Create a single object with the `new` operator.	Same.
Construction of object hierarchy	Construct an object hierarchy by using class definitions to define subclasses of existing classes.	Construct an object hierarchy by assigning an object as the prototype associated with a constructor function.
Inheritance model	Inherit properties by following the class chain.	Inherit properties by following the prototype chain.
Extension of properties	Class definition specifies *all* properties of all instances of a class. Cannot add properties dynamically at run time.	Constructor function or prototype specifies an *initial set* of properties. Can add or remove properties dynamically to individual objects or to the entire set of objects.

Prototype Inheritance: Step-by-Step Example (0.a)

- Let's build an example!
- We would like to describe a generic **person**, her interests and some of her "basic" activities…
- … we are going to use *construction functions* to do this!

```
function Person(first, last, age, gender,
interests) {
  this.name = { 'first': first, 'last' : last};
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}
Person.play = function() {
  console.log(`${this.name.first} is having
fun.`)
}
let person = new Person('Marco', 'Rossi', 35,
'male', ['music', 'painting']);
```

Prototype Inheritance: Step-by-Step Example (0.b)

- Now, we would like to represent a **student**...
- ... notice that our `Student` is almost equivalent to our `Person, but`
  - the student has a "study" function
  - the student has an "id"
- How can we use the inheritance to **link** students with persons?
  - and remove some duplicate code?

```
function Student(first, last, age, gender, interests, id) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


Student.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

Prototype Inheritance: Step-by-Step Example (1.a)

- First, let's put everything together and remove duplicate code!

Notably, this code does not work !!!

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last': last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}
Person.play = function() {
  console.log(`${this.name.first} is having fun.`)
}


function Student(id) {
  this.id = id;
  this.study = function() {
    // we will re-add the console.log() later on
  }
}
```

Prototype Inheritance: Step-by-Step Example (1.b)

- Then, we need to define a `Student` with all the properties of `Person` (name, age, etc.)
  - we can call the `Person`'s constructor from the `Student`'s one, to chain constructors
- We use the `call()` function
  - it allows for a function belonging to one object to be assigned and called for a different object
  - it provides a new value of `this` to the function/method
  - so you can write a method once and then inherit it in another object, with the proper context, without having to rewrite the method

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}
Person.play = function() {
  console.log(`${this.name.first} is having fun.`)
}


function Student(first, last, age, gender, interests, id) {
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

Prototype Inheritance: Step-by-Step Example (1.b)

- Done?
- What happens if we execute `student.sleep()`?
  - `TypeError: student.sleep is not a function`
- We need to tell `Student` that its prototype is `Person`, to inherit `Person`'s methods!

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}
Person.play = function() {
  console.log(`${this.name.first} is having fun.`)
}


function Student(first, last, age, gender, interests, id) {
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

Prototype Inheritance: Step-by-Step Example (1.c)

- Currently, every instance of `Student` has the properties of `Person` (first, last, etc.), but not its *sleep()* method
- We know that inherited properties should go in the `prototype` property: let's check this!

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.prototype.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}
Person.play = function() {
  console.log(`${this.name.first} is having fun.`)
}


function Student(first, last, age, gender, interests, id) {
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

Prototype Inheritance: Step-by-Step Example (1.d)

- Now, we want to make sure that all instances of Student will have access to the methods on `Person.prototype`
- We can use `Object.create()`
  - it allows you to create an object which will delegate to another object on failed lookups
  - in other terms, it returns a newly created object that inherits from a specified prototype object
- In our case, the object we want to create is `Student`'s prototype and the object we want to delegate to on failed lookups is `Person.prototype`

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.prototype.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}


function Student(first, last, age, gender, interests, id)
{
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


Student.prototype = Object.create(Person.prototype);


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

Prototype Inheritance: Step-by-Step Example (1.d)

- One last thing, not always needed
- After replacing the object on `Student`'s prototype property, we have that the `Student`'s prototype constructor inherits from `Person.prototype`
  - which has `constructor = Person`
- This is *misleading* and *may* generate issues, e.g., when copying objects or with functions that read that value
  - so, we need to "restore" the original `Student`'s constructor

```
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.prototype.sleep = function() {
  console.log(`${this.name.first} is sleeping.`)
}


function Student(first, last, age, gender, interests, id) {
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() {
    console.log(`${this.name.first} is studying.`)
  }
}


Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;


let student = new Student('Marta', 'Verdi', 22, 'female',
['music', 'skiing'], 12345);
```

```javascript
function Person(first, last, age, gender, interests) {
  this.name = { 'first': first, 'last' : last };
  this.age = age;
  this.gender = gender;
  this.interests = interests;
}
Person.prototype.sleep = function() { console.log(`${this.name.first} is sleeping.`)}
Person.play = function() { console.log(`${this.name.first} is having fun.`) }


function Student(first, last, age, gender, interests, id) {
  Person.call(this, first, last, age, gender, interests);
  this.id = id;
  this.study = function() { console.log(`${this.name.first} is studying.`) }
}
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;


let student = new Student('Marta', 'Verdi', 22, 'female', ['music', 'skiing'], 12345);


student.sleep(); // "Marta is sleeping."
Student.play(); // "ERROR, a student does not play!!!"
```

Classes

- Classes are primarily ***syntactical*** sugar over JavaScript's existing prototype-based inheritance
  - included from ES6
- They are special functions, based on the **`class`** keyword
- Two ways to define a class:
  - **class declaration**
  - **class expression**
- A class can be instantiated with the **`new`** keyword

Class Declaration

- One way to define a class:
  - `class` + chosen name of the class
- Class declarations are **<u>not</u>** hoisted
  - you cannot instantiate a class before declaring it
    - you should not, in any case!

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Class Expression

- Another way to define a class, with two variants:
  - *named*
  - *unnamed*
- The name given to a (named) class expression is local to the class's body
  - and accessed through the class' `name` property
  - it is "myRectangle" and "Rectangle" for the example
- As class declarations, they are **<u>not</u>** hoisted

```
// named
let Rectangle = class myRectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Class Body

- The class body is always executed in **strict mode**
- Each class can have only one `constructor()`
  - a constructor can use the `super` keyword to call the constructor of the super class
- Classes can have
  - prototype methods
  - static methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Prototype Methods

- Several types of prototype methods exist
- The syntax for a method is:

  -
    ```
    methodName() {
       /* method body */
    }
    ```
  - it adds a property named `methodName` to the class and sets the value of that property to the specified function
  - you use this with ***objects***, too

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
const square = new Rectangle(10, 10);
console.log(square.calcArea());
```

Prototype Methods: Getters and Setters

- JavaScript defines two methods to create a *pseudo-property*
- **Getters** allow access to a property that returns a dynamically computed or internal value
  - get + function()
- **Setters** are used to execute a function whenever a specified property is attempted to be changed
  - set + function()
- It is **not possible** to simultaneously have a
  - getter bound to a property and have that property hold a value
  - setter on a property that holds an actual value

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get perimeter() {
    return this.calcPerimeter();
  }
  // Setter
  set perimeter(perimeter) {
    this.height = perimeter/2 - this.width;
  }
  // Method
  calcPerimeter() {
    return 2*(this.height + this.width);
  }
}
const square = new Rectangle(10, 10);
square.perimeter = 100;
console.log(square.perimeter);
```

Static Methods

- The `static` keyword defines a static method for a class
- Static methods are called without instantiating their class and cannot be called through a class instance

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Static method
  static isWider(a, b) {
    return (a.width > b.width)? a: b;
  }
}
const s = new Rectangle(10, 10);
const r = new Rectangle(20, 10);
console.log(Rectangle.isWider(s, r));
```

Subclassing and Super Class Calls

- The `extends` keyword is used to create a class as a child of another class
  - it works with "super classes" defined as construction functions, too
- The `super` keyword is used to call corresponding methods of super class
  - *not* only the constructor!
  - not *only* from the constructor!

```
class Person {
  constructor(first, last, age, gender, interests) {
    this.name = { 'first': first, 'last' : last };
    this.age = age;
    this.gender = gender;
    this.interests = interests;
  }
  sleep() {
    console.log(`${this.name.first} is sleeping.`)
  }
  play() {
    console.log(`${this.name.first} is having fun.`)
  }
}


class Student extends Person {
  constructor(first, last, age, gender, interests, id)
{
    super(first, last, age, gender, interests);
    this.id = id;
  }
}
```

Modules

- Mechanisms for splitting JavaScript programs into separate files that can be imported when needed
- A module is a JavaScript file that **exports** one or more values (objects, functions or variables), using the `export` keyword
  - each module is a piece of code that is executed once it is loaded
- Any other JavaScript module can **import** the functionality offered by another module by importing it, with the `import` keyword
- Imports and exports must be at the ***top level***
- Two main kinds of exports:
  - **named** exports (several per module)
  - **default** exports (one per module)

Default Export

- Modules that only export **single values**
  - one per module
- Syntax
  - `export default <value>`

```
export default str =>
str.toUpperCase();


// OTHER examples
export default {x: 5, y: 6};


export default "name";


function grades(student) {...};
export default grades;
```

Named Exports

- Modules that export **one or more values**
  - several for modules
- Syntax
  - `export <value>`
  - `export {<value>, <...>}`

```
export const name = 'Luigi';


function grades(student) {...};

export grades;


const name = 'Luigi';

const anotherName = 'Fulvio';

export { name, anotherName }

// we can also rename them...

// export {name, anotherName as teacher}
```

Imports

- To import something exported by another module
- Syntax(es)
  - **ES6:**
    `import package from 'module-name'`
  - **CommonJS:**
    `const package = require('module-name')`
- Imports are:
  - hoisted
  - read-only views on exports

Import From a Default Export

```
--- module1.js ---
export default str =>
str.toUpperCase();
```

```
--- module2.js ---
import toUpperCase from './
module1.js';
// you choose the name!

// another example
import uppercase from '/home/appweb/
module1.js';

// usage of the imported function
uppercase('test');
```

Import From a Named Export

```
--- module1.js ---
const name = 'Luigi';
const anotherName = 'Fulvio';


export { name, anotherName };
```

```
--- module2.js ---
import { name, anotherName } from
'./module1.js';


// you can rename imported values,
// if you want
import { name as first, anotherName
as second} from './module1.js';
// usage
console.log(first);
```

Other Imports Options

- You can import everything a module exports
  - `import * from 'module'`
- You can import a few of the exports (e.g., if `exports {a, b, c}`):
  - `import {a} from 'module'`
- You can import the default export with any named exports:
  - `import default, { name } from 'module'`

this

JavaScript: The Definitive Guide, 7th Edition
Chapter 8. Classes

You Don't Know JS: this & Object Prototypes

'`this`' in JavaScript

- Given the peculiar treatment of Objects and Classes in JS, the '`this`' keyword behaves differently than on other OO languages
  - '`this`' does not refer to the function in which it appears
  - '`this`' does not (always) refer to the current object (functions are not always bound as object methods)
  - '`this`' does not refer to the context (i.e., external functions) in which the function is defined
  - '`this`' does not refer to the object that generated the call (e.g., the object generating an event)
- Nevertheless, '`this`' is extremely useful in callbacks and object methods
  - We must learn its rules…

The golden rule

- Within each function, the '`this`' keyword is always *bound* to some specific objects
- The binding of '`this`' depends (almost) **exclusively** on the ***call site*** of the function (i.e., how the function is called)
  - Does not depend on *how* the function is declared (function expression, function statement, passed references, …)
  - Does not depend on *where* the function is declared (global, object property, nested, …)

Notable exception: Arrow Functions (see at the end)

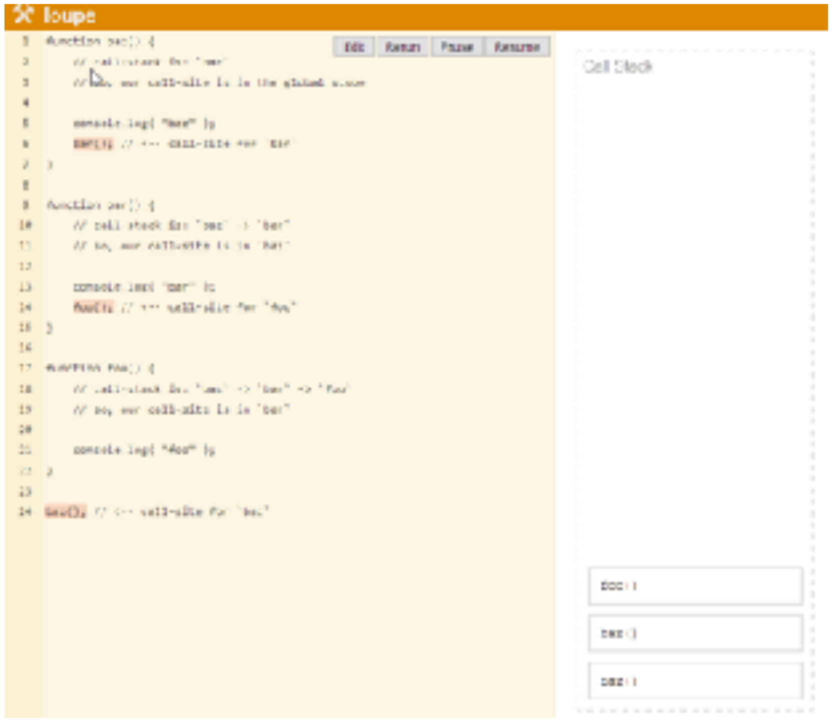
The *call site* of a function

- Locate where the function is called from
  - imagine being in a function, just called
  - go back one step in the ***call stack***, and check where you were just before being called
  - that location is the ***true*** call site
- The same function might be called from different places, in different times
  - each time, the call site for ***that invocation*** is the **only** important information

Example Call Site Analysis

http://latentflip.com/loupe/

```
function baz() {
    // call-stack is: `baz`
    // so, our call-site is in the global scope

    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}

function bar() {
    // call-stack is: `baz` -> `bar`
    // so, our call-site is in `baz`

    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}

function foo() {
    // call-stack is: `baz` -> `bar` -> `foo`
    // so, our call-site is in `bar`

    console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```



Rule #1: default binding

- Standalone function invocation

  ```
  let a = foo();
  ```

  - normal function call
  - default rule, applies if other special cases do not apply
- When in strict mode, '`this`' inside '`foo`' is undefined
- When not in strict mode, '`this`' inside '`foo`' is the global object
  - global or window, according to the execution environment
- It is useless, no reason to use it
  - never use '`this`' inside functions called in standalone mode

Rule #2: Implicit binding

- Called in the context of an object (method)

  `let a = obj.foo() ;`
- `foo()` is a property of `obj`
  - defined inline with a function expression
  - defined elsewhere but assigned to a property
- Inside `foo`, `this` refers to `obj`
  - the specific object instance on which the function is called
  - `this.a` refers to property `a` of `obj`

```
function foo() {
  console.log( this.a );
}

let obj = {
  a: 2,
  foo: foo
};

obj.foo(); // 2
```

Beware: losing the object reference

```
function foo() {

  console.log( this.a );

}



let obj = {

  a: 2,

  foo: foo

};



let bar = obj.foo; // function reference/alias!

Call Site

bar(); // "oops, global"
```

```
function foo() {
  console.log( this.a );
}



function doFoo(fn) {
  // `fn` is just a reference to `foo`
  fn();
}
Call Site


let obj = {
  a: 2,
  foo: foo

};



doFoo( obj.foo ); // "oops, global"
```

Beware: losing the object reference

```
function foo() {

  console.log( this.a );

}



let obj = {

  a: 2,

  foo: foo

};



let bar = obj.foo; // fur

Call Site

bar(); // "oops, global"
```

```
function foo() {
  console.log( this.a );
}


function doFoo(fn) {
  // `fn` is just a reference to `foo`
  fn();
}
Call Site







); // "oops, global"
```

Must be careful, if we pass the function reference around, we lose the object reference, and the "default binding" will be applied.

Always pass objects, never functions, if you want ‘this’ to work in the passed object

Rule #3: Explicit binding

- You may call a function indirectly, with a ***calling method*** (natively defined for all JS functions)

```
let y = foo.call(object, param, param, param)
let y = foo.apply(object, [param, param, param])
```

- In this case the call to `foo` is ***explicitly bound*** to the `object` (1st parameter)
  - inside the function, `this` is bound to `object`
  - it basically behaves like `object.foo()`, even if `foo` is not a property of `object`.

Hard Binding

- Even the explicit binding may be "lost", if you pass the function around
  - instead of passing the object
- You may force a binding to a function using its **`.bind()`** method to construct a new 'bound' function

```
let newfoo = foo.bind(object) // newfoo is a
bound function
let y = newfoo(params)
```

- The `newfoo` function will always be bound to `object`

Rule #4: `new` binding

- When an object is created with a constructor function call, the function is bound to the newly created object

```
let obj = new Foo();
```

  - within Foo, this refers to the new object (later assigned to obj)

Aside: how ‘`new`’ works

- JS constructor call
  - when a function is invoked with `new` in front of it

```
let object = new Func() ;
```

1. a brand-new object is created (aka, constructed) out of thin air
2. the newly constructed object is [[Prototype]]-linked *(not relevant now)*
3. the newly constructed object is set as the '`this`' binding for that function call
4. unless the function returns its own alternate object, the `new`-invoked function call will automatically return the newly constructed object

Summary of rules

- Is the function called with `new` (**new binding**)?
  - if so, `this` is the newly constructed object

```
let bar = new Foo();
```

- Is the function called with `call` or `apply` (**explicit binding**), even hidden inside a `bind` *hard binding*?
  - if so, `this` is the explicitly specified object

```
let bar = foo.call( obj2 );
```

Summary of rules

- Is the function called with a context (**implicit binding**), otherwise known as an owning or containing object?
  - if so, `this` is *that* context object.

```
let bar = obj1.foo() ;
```

- Otherwise (**default binding**):
  - if in *strict mode*, `this` is `undefined`, otherwise `this` is the global object (`global` in node, `window` in browsers)

```
let bar = foo()
```

Exception : Arrow Functions =>

- The above rules do **not** apply to Arrow Functions

```
let fun = (n) =>
{ this.a=n; }
```

- Arrow functions adopt the `'this'` binding **from the enclosing function scope** (or global scope)
  - check the call site of the enclosing function!
- Extremely handy in event handlers and callbacks

```
function foo() {
     setTimeout(() => {
     // `this` here is lexically
     // adopted from `foo()`
          console.log( this.a );
     },100);
}

let obj = {
          a: 2
};

foo.call( obj ); // 2
```

In practice...

Rule	Example at call site	Suggestion
	`let foo = function(n) { this.a = n ; }`	
4. New binding	`let y = new Foo(3) ;`	**Normal usage for object constructors**

3. Explicit binding	`let y = foo.call(obj, n) ;`
	`let newfoo = foo.bind(obj) ;`
		Seldom used in user code, mostly in libraries

2. Implicit binding	`let y = obj.foo() ;`	**Normal usage for object methods**
1. Default binding	`let y = foo() ;`	Never use. Does not work in Strict mode.

Exception:
Arrow Functions
	`let foo = (n)=>{ this.a = n ; }`
	`Uses surrounding scope (closure over this)`
		Useful in callbacks (event handlers, async functions, …)

References for 'this'

- You Don't Know JS: this & Object Prototypes - 1st Edition, Kyle Simpson, https://github.com/getify/You-Dont-Know-JS/tree/1st-ed/this%20%26%20object%20prototypes , Chapter 1 and Chapter 2