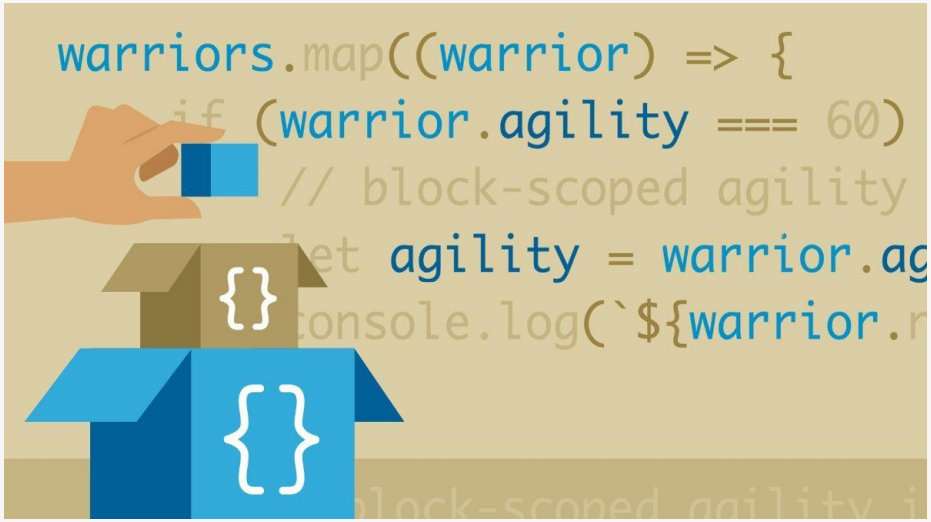
Front end development

Lesson 5 - Advanced Javascript

adapt

Scope

- Scope is a current execution context.
- The lexical context in which values and expressions can be accessed.
- If a variable is not in the current scope, then it's value is undefined, it is unavailable for us to use.
- Scopes could be layered in hierarchy so that child scopes have access to parent scopes but not vice versa.



adapt

Scope

- **Global variables** declared outside function has global scope and can be accessed everywhere.

```
var car = "Volvo";
console.log(car); // Volvo.
console.log(window.car); // Volvo.
```

- **Local (function scope) variables** have function scope, they can only be accessed from within the function.

```
function myCar() {
  var carName = "Audi";
  console.log(carName); // Audi.
}

console.log(typeof carName); //
Undefined.
```

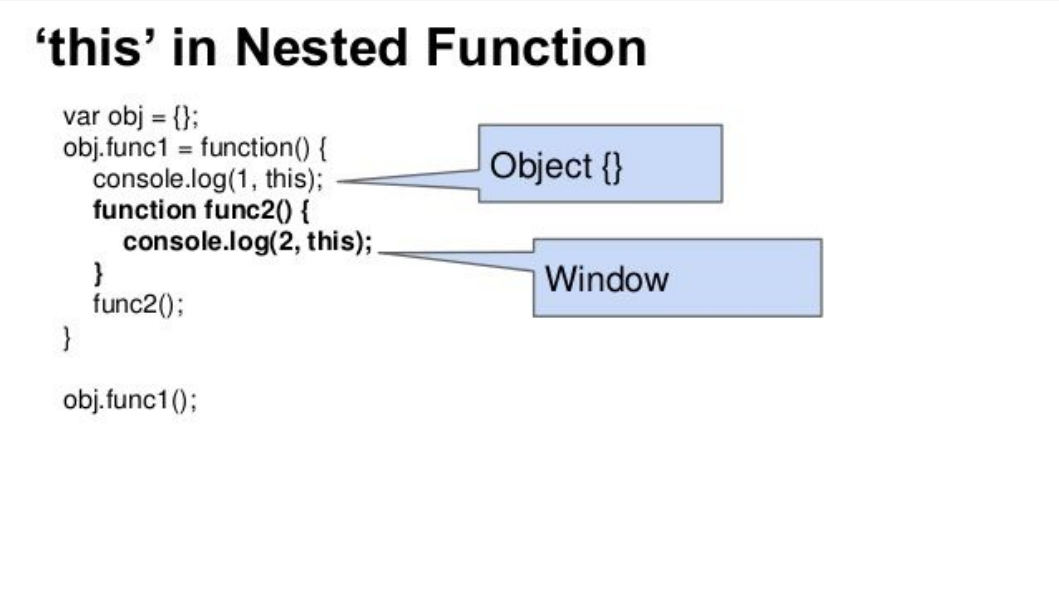
- **Block scope variables** are accessible only inside a single block statement where it's defined. variables defined with *let* and *const* are block scope.

```
function myCar() {
  var carName = "BMW";

  for (let i = 1; i <= 4; 1++) {
    console.log(carName, i) // BMW 1, 2, 3,
4
  }

  console.log(carName, i) // BMW Undefined.
}
```

adapt

This keyword in Javascript

- **This** in Javascript references to an object executing current function.
- **This** mostly is determined by how a function is called (using a runtime binding).
- In arrow functions **this** will behave differently.



This keyword in different contexts

- In **global execution contex**t when not inside any function *this* will always refers to a global object which is *window*.
- In **function context** when inside a function *this* will depend on how function was called.

```
console.log(this);
// Window {parent: Window, postMessage: ƒ, blur: ƒ, focus:
ƒ, close: ƒ, …}

const myDog = {
  name: 'Lokis',
  color: 'brown',
  age: 5,
  bark() {
    console.log(this.name, 'barks');
  },
};

myDog.bark() // Lokis barks.
```

adapt

This keyword in function context

- In **function context** (when inside a function) *this* will depend on how function was called:
  - **Simple call** - *this* will be a global object (window).
  - **Bind method** - you can bind any value as *this* to a function.
  - **Arrow function** - *this* will be the same as enclosing lexical context.
  - **Object method** - *this* will be pointing to a object the method is called on.
  - **Object prototype chain** - *this* refers to the object the method was called on, as if the method were on the object.
  - **With a getter or setter** - A function used as *getter* or *setter* has its this bound to the object from which the property is being set or gotten.
  - **As constructor** - when a function is called with a *new* keyword it's *this* is bound to the new object being constructed.
  - **Inline event handler** - *this* is bound to the DOM element the listener is placed on.

This keyword in function context example

- Functions *bark()* and *barkArrow()* were called on a *dog* object.
- Therefore **this** refers to a *dog* object inside those functions.
- Object properties could be accessed using **this** keyword.
- Note that **this** becomes a global *window* object inside *forEach* callback in *bark()* function, because this function was not called and bind itself a *window*.
- Inside *barkArrow() forEach* has an arrow function, so it is not bound to a *window* and still refers to a dog object.

```
const dog = {
  name: 'Lokis',
  color: 'brown',
  barks: ['au', 'auu', 'auuu', 'woof'],
  bark() {
    console.log(this.name); // Lokis
    this.barks.forEach(function(bark) {
      console.log(this.name, bark); // Undefined au ...
    });
  },
  barkArrow() {
    this.barks.forEach((bark) => {
      console.log(this.name, bark); // Lokis au ...
    });
  },
};

dog.bark();
dog.barkArrow();
```

Arrow functions

- Arrow functions are syntactically compact alternative to regular functions.
- They are without it's own bindings to **this** keyword.
- Therefore these functions are not suited to use as an object's methods and cannot be used as a constructor function.

```
// ES5
var add = function (num1, num2) {
    return num1 + num2;
}

// ES6
var add = (num1, num2) => num1 + num2
```

adapt

Arrow functions syntax

- Arrow function does automatic return when it's body is wrapped inside parenthesis ()
- Actually parenthesis could be skipped if return logic is in a single line. If there is a single parameter, parenthesis could be skipped also.
- Statements in curly brackets {} will act as in a regular function and *return* keyword should be supplied.

```
const sayHello = (name) => (
  'Hello ' + name
);

const sayHello2 = name => 'Hello ' + name;

const sayHello3 = (name) => {
  return 'Hello ' + name;
};
```

adapt

Lexical this binding in arrow functions

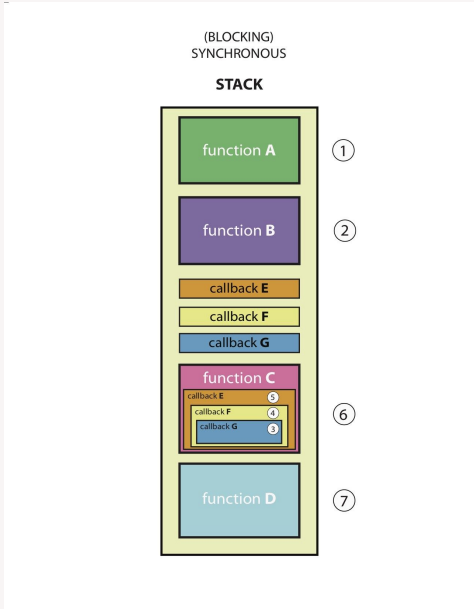
- Arrow functions do not provide their own binding to this value.
- It uses this value from the enclosing context that it was defined on.
- In Regular functions this keyword represents the object that called the function.

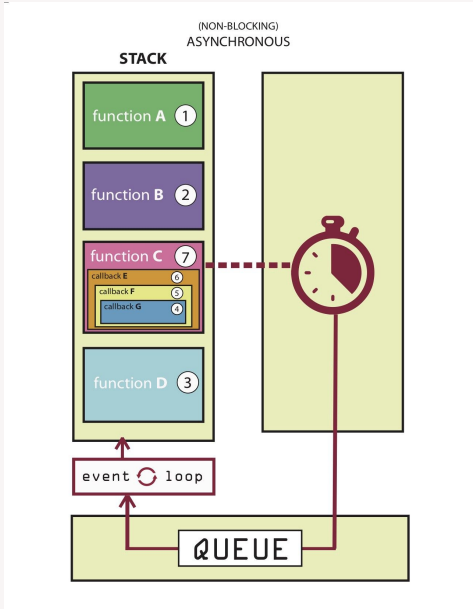
```
const obj1 = {
  name: 'Irmantas',
  age: 29,
  getInfoRegular: function() {
    console.log(this);
  },
  getInfoArrow: () => {
   console.log(this);
 },
};

// Outputs: {name: "Irmantas", age: 29, getInfoRegular: ƒ, getInfoArrow: ƒ}
obj1.getInfoRegular();

// Outputs: Window {parent: Window, postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, …}
obj1.getInfoArrow();
```

adapt

Asynchronous vs synchronous code execution

- Javascript is synchronous, blocking and single threaded language by default.
- But there are few ways to make it behave asynchronous non-blocking.
- These are **callbacks**, **promises** and **async/await** functions.





adapt

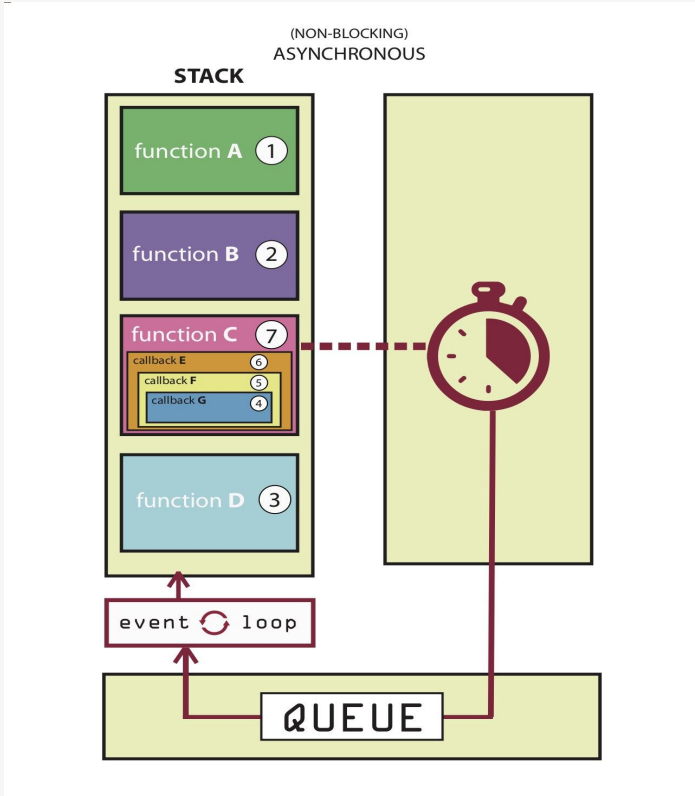
Callbacks

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
- Note, however, that callbacks are often used to continue code execution after an asynchronous operation has completed — these are called asynchronous callbacks.
- Since JavaScript is single threaded all processing blocks until one of the following occurs:
    - The current execution requests an external service such as an I/O or networking request, or a webworker request.
    - A function call is put on a timer to be executed at a later time.

Callback example

- Non blocking function using a callback is a `setTimeout()` in the example given.
- Function together with a nested callback is sent to the queue and then processed through an event loop, so it won't block further execution of code.

```
// Blocking function 1.
console.log('1');

// Non blocking function using a callback.
setTimeout(() => console.log('2'), 0);

// Blocking function 2.
console.log('3');

// Result: 1 3 2.
```



adapt

Blocking callback

- In the example given, we have a *blockingFunction(name, callback)* which takes another function - *callback* as an argument.
- This function is a blocking synchronous function since it's callback is executed immediately.
- *callback(name)* is not wrapped inside *setTimeout(),* so that's why it's going to execute in a sequence.

```
function greeting(name) {
  console.log('Hello ' + name);
}

function blockingFunction(name, callback) {
  for (let i = 1; i <= 2; i++) {
    console.log(i);
  }
  callback(name);
}

console.log('started');
blockingFunction('Irmantas', greeting);
console.log('finished');

// Result: started 1 2 Hello Irmantas finished
```

adapt

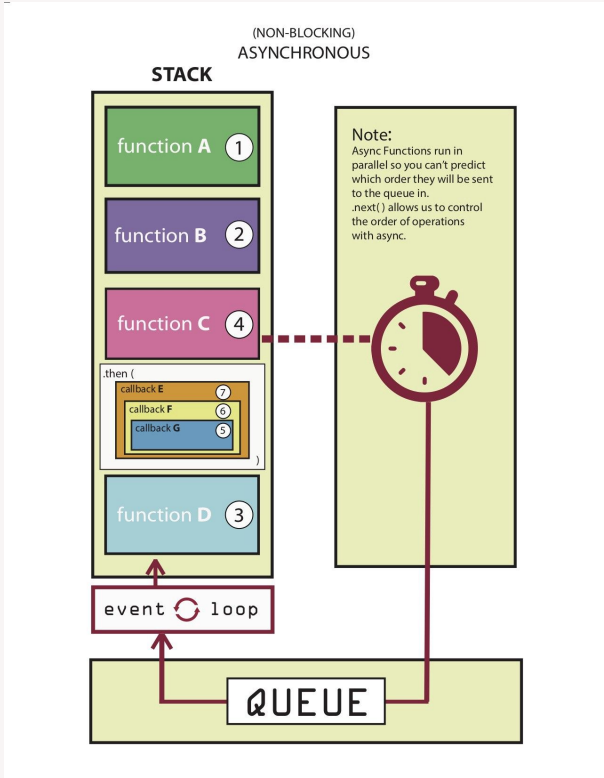
Non blocking callback

- In the example given the *nonBlockingFunction(name, callback)* is a non-blocking asynchronous function.
- It's callback is wrapped inside async *setTimeout()* function.
- So it will not block the execution of the following statements.
- Also there are more build in asynchronous functions in JS, but we need to read docs on them or experiment to make sure they are.

```
function greeting(name) {
  console.log('Hello ' + name);
}

function nonBlockingFunction(name, callback) {
  setTimeout(function() {
    for (let i = 1; i <= 2; i++) {
      console.log(i);
    }
    callback(name);
  }, 0);
}

console.log('started');
nonBlockingFunction('Irmantas', greeting);
console.log('finished');

// Result: started finished 1 2 Hello Irmantas
```

Promises

- In order to avoid so called callback hell (deep nesting of functions) which could be an issue in a more complex callback functions Promises could be useful.
- Promises allows us to code in a more modular, readable way while still maintaining asynchronous code execution.
- The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.



adapt

Callback vs Promise Example

```
function loadImageCallbacked(url, callback) {
  let image = new Image();

  image.onload = function() {
    callback(null, image);
  }

  image.onerror = function() {
    let message = 'Could not load image at ' + url;
    callback(new Error(message));
  }

  image.src = url;
}
```

```
function loadImagePromised(url) {
  return new Promise((resolve, reject) => {
    let image = new Image();

    image.onload = function() {
      resolve(image);
    }

    image.onerror = function() {
      let message = 'Could not load image at ' + url;
      reject(new Error(message));
    }

    image.src = url;
  });
}
```

adapt

Callback vs Promise Example

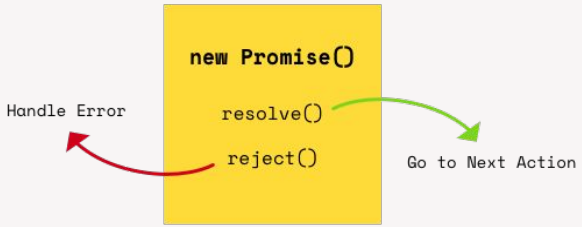
```
loadImageCallbacked('assets/images/cat1.jpeg', (error, img) => {
  if (error) throw(error);
  addImg(img.src);

  loadImageCallbacked('assets/images/cat2.jpeg', (error, img) => {
    if (error) throw(error);
    addImg(img.src);

    loadImageCallbacked('assets/images/cat3.jpeg', (error, img) => {
      if (error) throw(error);
      addImg(img.src);
    });
  });
});
```

```
Promise.all([
  loadImagePromised('assets/images/cat1.jpeg'),
  loadImagePromised('assets/images/cat2.jpeg'),
  loadImagePromised('assets/images/cat3.jpeg'),
]).then((images) => {
  images.forEach(image => addImg(image.src));
}).catch(error => {
  throw(error);
});
```

adapt

Callback vs Promise conclusion

- As you can see there is a deep nesting of functions when trying to maintain the order of execution in async callbacks.
- Promises compose, so it's much more readable and error handling is more straightforward compared to callback based approach.



adapt

Async, await

- Inside a function marked as an async you are allowed to place an await keyword in front of an expression that returns a promise.
- Then the execution of async function is paused until the promise is resolved.
- The idea behind async / await is to be able to write asynchronous code that looks like synchronous code.
- Async functions return a promise.

Async () => { Await }

adapt

Promise vs async/await Example

- In the example given function will receive data from API using *Promise* based *fetch* method.
- As you can see each step is chained inside *.then()* method.

```
function fetchCatImagesPromise(userId) {
  return fetch(`http://catappapi.herokuapp.com/users/${userId}`)
    .then(response => response.json())
    .then(user => {
      const promises = user.cats.map(catId =>
        fetch(`http://catappapi.herokuapp.com/cats/${catId}`)
          .then(response => response.json())
          .then(catData => catData.imageUrl)
      )

      return Promise.all(promises);
    });
}

fetchCatImagesPromise(123)
  .then(result => console.log('promise 1', result));
/**
 * Result:
 * [
 *   "http://images.somecdn.com/cat-21.jpg",
 *   "http://images.somecdn.com/cat-33.jpg",
 *   "http://images.somecdn.com/cat-45.jpg",
 * ]
 */
```

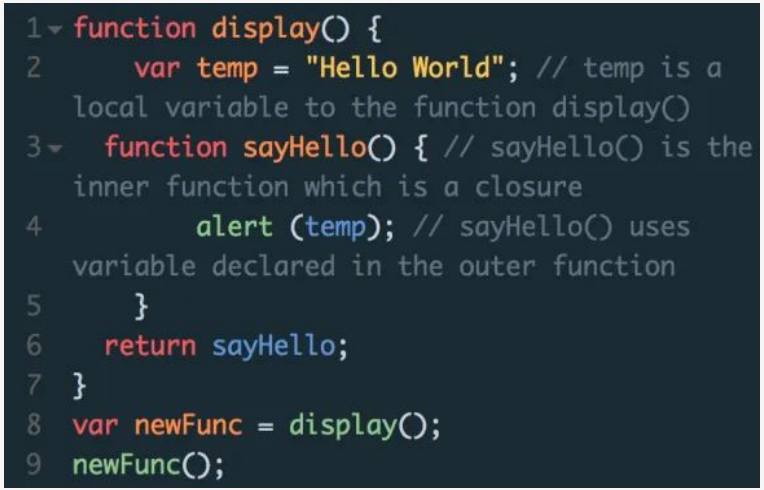
Promise vs async/await Example

- Fetching data from API using async / await methods.
- There is an async function passed inside a map.
- Async function returns a promise for each iteration, so we have an array of promises in *catImageUrls*.
- We need to combine these promises into one using Promise.all() and return it.

```
async function fetchCatImagesAsync2(userId) {
  const response = await fetch(`http://catappapi.herokuapp.com/users/${userId}`);
  const user = await response.json();
  const catImageUrls = user.cats.map(async (catId) => {
    const response = await fetch(`http://catappapi.herokuapp.com/cats/${catId}`);
    const catData = await response.json();

    return catData.imageUrl;
  });

  return await Promise.all(catImageUrls);
}

fetchCatImagesAsync2(123)
  .then(result => console.log('Promise 3', result));

/**
 * Result:
 * [
 *   "http://images.somecdn.com/cat-21.jpg",
 *   "http://images.somecdn.com/cat-33.jpg",
 *   "http://images.somecdn.com/cat-45.jpg",
 * ]
 */
```

Closures

- A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment).
- In other words, a closure gives you access to an outer function's scope from an inner function.
- In JavaScript, closures are created every time a function is created, at function creation time.



```
function display() {
    var temp = "Hello World"; // temp is a local variable to the function display()
  function sayHello() { // sayHello() is the inner function which is a closure
        alert (temp); // sayHello() uses variable declared in the outer function
    }
  return sayHello;
}
var newFunc = display();
newFunc();
```

Closure example

- The following code illustrates how to use closures to define public functions that can access private functions and variables.
- Using closures in this way is known as the module pattern.
- This is a powerful way to encapsulate these properties from global namespace.
- Here we create a single lexical environment that is shared by three functions: *counter.increment*, *counter.decrement*, and *counter.value*.

```
const counter = (function() {
  let privateCounter = 0;

  const changeBy = (val) => {
    privateCounter += val;
  };

  return {
    increment: () => changeBy(1),
    decrement: () => changeBy(-1),
    value: () => privateCounter,
  };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```

Object Oriented Javascript

- **Encapsulation** - we group related functions(methods) and variables into single object. Pseudo encapsulation.
- **Abstraction** - complexity is hidden from you. Isolate impact of changes.
- **Inheritance** - avoid redundancy.
- **Polymorphism** - refactor ugly switch/case statements.

```
const circle = {
  radius: 1, // property.
  location: {
    x: 1,
    y: 1,
  },
  draw: function() { // method.
    console.log(this);
  },
}

circle.draw(); // Logs {radius: 1, location: {…}, draw: f}
```

adapt

Factory functions

- Object literal syntax is not a good way to create an object and duplicate it if it has a method.
- Since the method will be the same.
- Factory functions are functions used to create an object.

```
// Factory function.
function createCircle(radius) {
  return {
    radius,
    draw: function() {
      console.log(this);
    },
  };
}

const circle = createCircle(1);
circle.draw(); // {radius: 1, draw: ƒ}
```

Constructor functions

- **New** keyword creates an empty object and makes **this** keyword point to that object instead of global (node) or window (browser) object.
- Created function returns **this** automatically.

```
// Constructor function
function Circle(radius) {
  this.radius = radius;
  this.draw = function() {
    console.log(this);
  };
}

const circleObj = new Circle(1);
circleObj.draw(); // Circle {radius: 1, draw: ƒ}
```

adapt

Prototypes based object creation

- All objects in Javascript are instances of **Object** constructor.
- Typical object inherits methods from **Object.prototype**.
- This is so called a prototype chain.
- We can also create our own methods using *object.prototype.myMethod* syntax.
- These methods have access to **this** value of object created with new keyword using a constructor function.

```
function Book(author, title, year) {
  this.author = author;
  this.title = title;
  this.year = year;
}

Book.prototype.getDescription = function() {
  return `Book ${this.title} was written by ${this.author} in
${this.year}`;
}

const book1 = new Book('Book one', 'John Lil', '2019');

console.log(book1.getDescription());
```

adapt

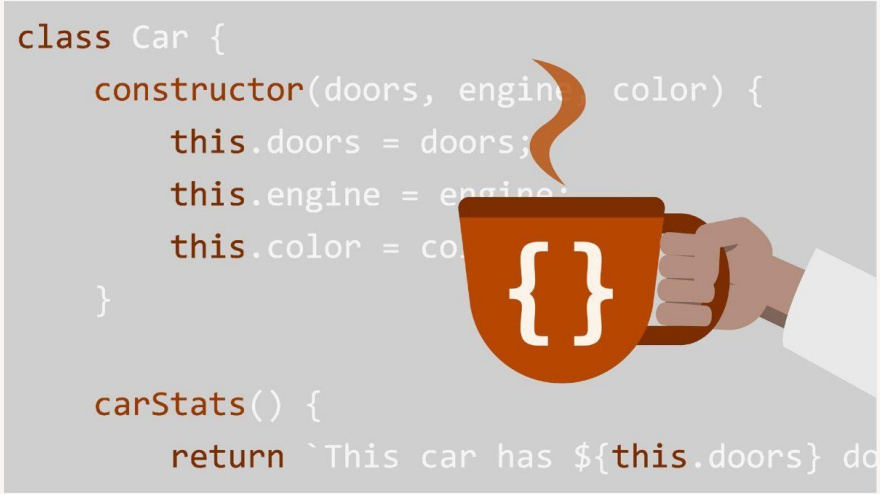
Inheritance using prototypes

- In order to inherit properties and methods from other object we need to call a parent constructor inside a child constructor using a *call()* method providing current *this* value and a properties required by a parent constructor.
- Then we should create a new object using *object.create()* method using a parent prototype and assign it to child prototype. This will allow prototype methods to be inherited.
- Also we should assign child prototype constructor to a child constructor function.

```
function Book(author, title, year) {
  this.author = author;
  this.title = title;
  this.year = year;
}

Book.prototype.getDescription = function() {
  return `Book ${this.title} was written by
    ${this.author} in ${this.year}`;
}

function Magazine(author, title, year, month) {
  Book.call(this, author, title, year,);
  this.month = month;
}

Magazine.prototype = Object.create(Book.prototype);
Magazine.prototype.constuctor = Magazine;

const mag1 = new Magazine('Mag one', 'John Lil', '2019', 'Dec');

console.log(mag1.getDescription());
// Book Mag 1 was written by John Lil in 2019
```

Class keyword

- Syntactic sugar for prototype inheritance, but makes it easier to read, understand and is more similar to traditional object oriented languages syntax.
- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactic sugar over JavaScript's existing prototype-based inheritance.
- The class syntax does not introduce a new object-oriented inheritance model to JavaScript.



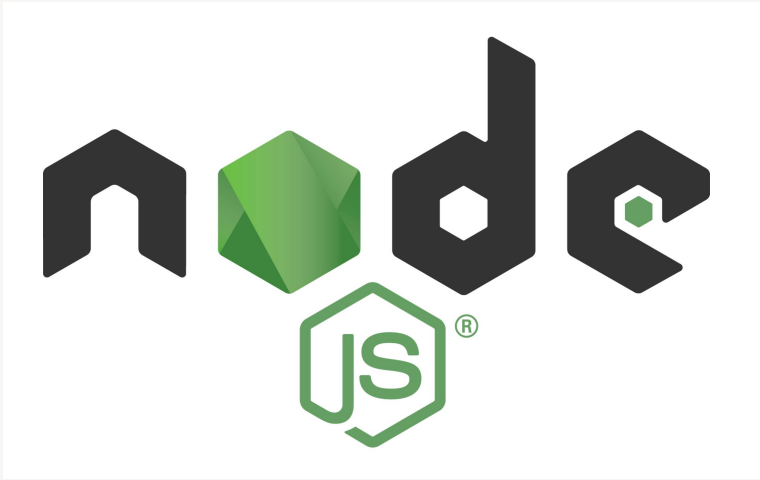
Class example

- The constructor method is a special method for creating and initializing an object created with a class.
- A constructor can use the super keyword to call the constructor of the super class.
- The extends keyword is used in class declarations or class expressions to create a class as a child of another class.
- The super keyword is used to call corresponding methods of super class

```
class Book {
  constructor(author, title, year) {
    this.author = author;
    this.title = title;
    this.year = year;
  }

  getDescription() {
    return `Book ${this.title} was written by
      ${this.author} in ${this.year}`;
  }
}

class Magazine extends Book {
  constructor(author, title, year, month) {
    super(author, title, year);
    this.month = month;
  }
}

const mag1 = new Magazine('John Lil', 'Mag 1', '2019', 'Dec');
console.log(mag1.getDescription());
// Book Mag 1 was written by John Lil in 2019
```

Server side Javascript

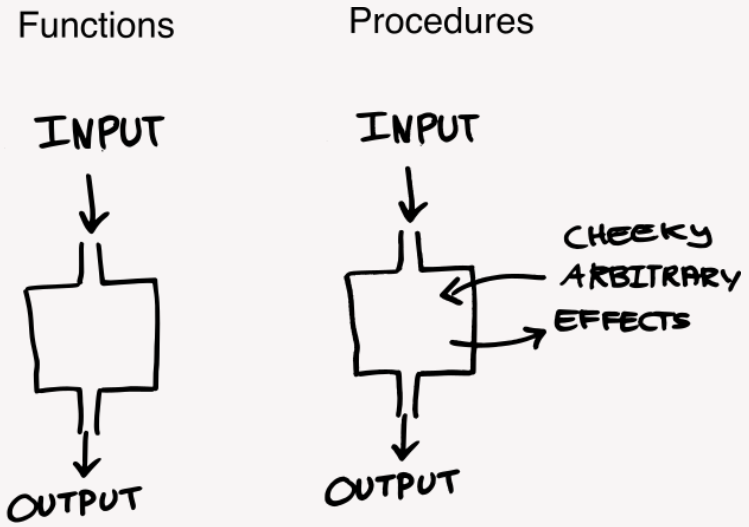
- Node vs Browser.
- In 2009, Ryan Dahl took Chromes v8 javascript engine and embed it into a c++ application.
- As a result now we have a node.js to run javascript outside of browser.



adapt

Functional programming

- is a programming paradigm — a style of building the structure and elements of computer programs that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.
- In functional code, the output value of a function depends ONLY on the arguments that are passed to the function
- Immutable means state cannot be modified after it is created. Mutable means state can be modified after it is created



adapt

Map

- The map() method creates a new array with the results of calling a provided function on every element in the calling array.
- map calls a provided callback function once for each element in an array, in order, and constructs a new array from the results.
- map does not mutate the array on which it is called.

```
const array1 = [1, 2, 3, 4, 5];

// pass a function to map
const map1 = array1.map((item, index, array) => {
  return item * 2;
});

console.log(map1);
// expected output: Array [2, 4, 6, 8, 10]
```

adapt

Reduce

- The *reduce()* method executes a provided reducer function on each element of the array, resulting in a single output value.
- Your reducer function's returned value is assigned to the accumulator (*acc*), whose value is remembered in each iteration throughout the array and becomes the final, single resulting value.
- If no initial value is supplied for reducer function, the first element in the array will be used as *acc* and skipped.

```
const arr = [1, 2, 3, 4];

const arrMultiplied = arr.reduce((acc, value,
index, array) => {
  acc[index] = value * 2;

  return acc;
}, []);

console.log(arrMultiplied);
// (4) [2, 4, 6, 8]
```

adapt

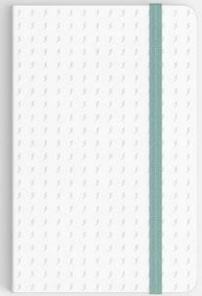
Filter

- The *filter()* method creates a new array with all elements that pass the test implemented by the provided function.
- *filter()* calls a provided callback function for each element in an array and constructs a new array of all the values for which callback returns a value that coerces to true

```
const arr1 = ['Adapt', 'Academy', 'Frontend',
'Lecture', 'Javascript', 'Html', 'Css'];

const filter1 = arr1.filter((word, index, array) => {
  return word.length > 5;
});

console.log(filter1);
// (4) ["Academy", "Frontend", "Lecture", "Javascript"]
```

adapt

Questions?
Answers?



Or if you'll have questions later:

irmantas.tamasauskas@adaptagency.com