Institute of Data

Software Engineering

Module 3 Advanced JavaScript

Advanced JavaScript

- Variable scope, closure
- Function object
- Scheduling: setTimeout and setInterval
- Decorators and forwarding, call/apply
- Function binding
- Prototypes
- Classes
- Error handling
- Promises, async/await

Variable scope

Lexical Environment

In JavaScript, every **function** and **code block** {...} has its own **scope**, and can define its own **lexical (local) environment** with its own **variables** (if declared with **let or const**). Internally, this lexical environment is stored as a hidden object.

The Lexical Environment object consists of two parts:

- **Environment Record**, an **object** that has **properties** to store all **local variables** (and some other information like the **value** of this).
- A **reference** to the **outer environment**, the one associated with the **outer code** from where the **inner** lexical environment was created.

Variable scope

Lexical Environment

A **variable** is just a **property** of the special **internal Environment Record object**.

The **outer environment** can be accessed from the **inner environment**, but not vice versa (unless we bundle them together).

```
let globalVariableCat = 'cat';
function myFunction() {
    let localVariableDog = 'dog'
    return 'global function with local scope variable '+localVariableDog;
}
console.log(localVariableDog) // ReferenceError: localVariableDog is not defined

// globalEnvironment = {
//     environmentRecord: {
//         globalVariableCat: 'cat', // has no access to localVariableDog
//         myFunction: <reference to function object>,
//     }
//     outer: null // there is no parent environment here
// }

// localMyFunctionEnvironment = {
//     environmentRecord: {
//          localVariableDog: 'dog'
//     },
//     outer: globalEnvironment // can still access everything in global
// }
```

Variable scope - Closures

A **closure** is the combination of a **function** and the **lexical environment** within which that function was declared. Closures allow a function to be **bundled** with variables that were in scope **at the time it was created**. This is most useful when we create functions that return other functions - a **function factory**:

```
function makeAdder(x) { // function factory: bundles value of x into the scope of adder
    return function adder(y) { // closure function 'adder' now has access to both x and y when created
        return x + y;
    };
}

const add5 = makeAdder(5); // sets x to 5, even when adder function is returned and called
console.log( add5(10) ) // x is still 5 and y is 10, result is 15
```

Variable scope - Closures

Closures are useful because we can associate the **current data from the lexical environment** with a **function that uses that data**. Closures are created every time a function is created, at function creation time (see MDN Closures).

```
function makeHeading(hTag) { // every call to makeHeading could have different values for hTag
    return function(title) { // unnamed closure function, can access value of hTag when created
        return `<${hTag}>${title}</${hTag}>`
    }
}

const getH1 = makeHeading('h1') // sets hTag to h1, creates new instance of closure in getH1
const getH2 = makeHeading('h2') // sets hTag to h2, creates separate instance of closure in getH2

console.log( getH1('Heading 1') ) // sets title to Heading 1 inside <h1>: <h1>Heading 1</h1>
console.log( getH2('Heading 2') ) // sets title to Heading 2 inside <h2>: <h2>Heading 2</h2>
```

Function object

In JavaScript, functions are objects (callable “**action objects**”). We can not only **call** them, but also treat them as **objects**, e.g. **add/remove** properties, **pass** by **reference**, etc.

name property

In the specification, this feature is called a **“contextual name”**. If the function does not provide one, then in an assignment it is figured out from the **context**.

```
function sayHiDefn() { console.log('Hi (function definition)'); } // named function
const sayHiExpn = function() { console.log('Hi (function expression)'); } // named variable
const sayHiArrow = () => console.log('Hi (arrow function)'); // named variable

//.name property exists as a built-in default property for all functions
console.log(sayHiDefn.name) // sayHiDefn - uses explicit function name
console.log(sayHiExpn.name) // sayHiExpn - figures out name from context
console.log(sayHiArrow.name) // sayHiArrow - figures out name from context
```

Function object

length property

Length is another built-in function property which returns the **number** of function **parameters**. The **rest parameters** **...** are not counted.

```
function oneParam(a) {
    console.log('This function has a single parameter: ' + a)
}
function twoParams(a, b) {
    console.log(`This function has two parameters: ${a} and ${b}`)
}
function manyParams(a, b, ...extras) {
    console.log(`This function has unlimited parameters: ${a} and ${b} and ${extras}`)
}
console.log(oneParam.length) // 1
console.log(twoParams.length) // 2
console.log(manyParams.length) // 2, because extras is 'the rest' and can't be counted
```

Function object

Custom properties

We can also add **properties of our own** to any function, with custom behaviour.

```
function sayHi() {
    console.log('Hi');
    sayHi.counter++; // increment the custom counter property every time the function is called
}
sayHi.counter = 0 // initialise the counter to 0 before calling it

sayHi() // Hi - calls the function and increments the counter
sayHi() // Hi - calls the function and increments the counter again
console.log( `Called ${sayHi.name} ${sayHi.counter} times` ) // Called sayHi 2 times
```

Scheduling - setTimeout

setTimeout allows us to delay execution of a function by a defined **interval of time**. It returns a unique reference to this timer so that it can be cancelled.

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], . . .)
```

- func|code **Function** or a **string** of code (not recommended) to execute
- delay: the **delay** before function execution, in **milliseconds**, by default **0**
- arg1, arg2, …: **arguments** for the function
- timerId: (can be renamed) a unique reference to this timer

```
function printMessage(msg) {
    console.log(`Message: ${msg}`)
}
// function to be executed, then milliseconds to delay, then arguments for function
let timerId = setTimeout(printMessage, 1000, 'prints after 1 sec') // Message: prints after 1 sec
```

Scheduling - setTimeout

clearTimeout allows us to cancel a **timer**, using the identifier returned by **setTimeout**. This allows us to delay execution and then cancel during the delay.

```
let cancelledTimerId = setTimeout(printMessage, 1000, 'timeout cancelled so never prints')
clearTimeout(cancelledTimerId); // printMessage function is cancelled before delay of 1 second
```

Since the first argument to setTimeout is a function, we often use an **arrow function** to wrap simple statements that are delayed:

```
setTimeout( () => console.log("log statement inside arrow function"), 500 )
// prints 'log statement inside arrow function' after 0.5 seconds
```

Scheduling - setTimeout

setTimeout is an **asynchronous function**, meaning that it doesn't pause during the delay period. Instead, the delayed function is put on a timer and other **synchronous code** still executes until the timer expires and the delayed function executes.

```
setTimeout( () => console.log("first message"), 5000 ); //asynchronous code with 5s delay
setTimeout( () => console.log("second message"), 3000 ); //asynchronous code with 3s delay
setTimeout( () => console.log("third message"), 1000 ); //asynchronous code with 1s delay
setTimeout( () => console.log("fourth message"), 0 ); //asynchronous code with no delay
console.log("fifth message"); //standard synchronous code
//order of messages when running code:
// fifth message (first, synchronous therefore no delay)
// fourth message (second, even though comes before fifth message, still no delay)
// third message (prints after 1s)
// second message (prints after 3s)
// first message (prints after 5s). Timers do not stack, so total of 5s delay between first and
fifth
```

Scheduling - setInterval

setInterval is another **asynchronous** function, with the same arguments as **setTimeout**, that returns a unique reference to allow the interval to be **cancelled**.

```
let timerId = setInterval((func|code, [delay], [arg1], [arg2], . . .)
```

Unlike **setTimeout**, which runs the function only once, **setInterval** runs the function **repeatedly**, with a fixed **interval of time** (delay) between each call.

```
let tickId = setInterval( () => console.log('tick'), 2000 ) // 'tick' every 2s
setTimeout( () => clearInterval(tickId), 10*1000 ) // after 10s ticking stops
```

If we don't call **clearInterval**, the function passed to **setInterval** will **continue to execute** after each delay until the environment running the code is stopped.

Scheduling - setInterval

We can use **setInterval** to run code repeatedly, and stop once an **exit condition** is reached, such as a maximum number of calls, or when a certain amount of time has passed, or when a user triggers an event to stop the timer.

```
function repeatInterval(delay, limit)
{
    let counter = 1; // part of the outer environment record for repeatInterval
    // intervalTimer is a reference to interval, to allow it to be cancelled
    let intervalTimer = setInterval(function repeatThis() {
        console.log('repeatInterval: repeated '+counter+' of '+limit+' times');
        if (counter == limit) clearInterval(intervalTimer); // cancel interval after execution limit
        counter++; // keep track of how many times the interval has executed, in outer environment
    }, delay); // delay is milliseconds between intervals
}
repeatInterval(2000, 10); // make the interval repeat every 2 seconds for a max of 10 times
```

Scheduling - setTimeout

Nested setTimeout

There are two ways of running something **regularly**. **One** is setInterval. The **other** one is a **nested (or recursive) setTimeout**.

```
function repeatTimeout(delay, limit)
{
    let counter = 1;
    // setTimeout only happens once, so we don't need the reference to cancel it
    setTimeout(function repeatThis(current) { // named function, so we can refer to it recursively
        console.log('repeatTimeout: repeated ' + current + ' of ' + limit + ' times');
        // we do need to call setTimeout recursively so that it repeats executing the function
        if (current < limit) setTimeout(repeatThis, delay, current+1) // repeat if limit not reached
    }, delay, counter);
}
repeatTimeout(2000, 10); // make the timeout repeat every 2 seconds for a max of 10 times
```

Decorators and forwarding

Decorators

A **decorator** is a **wrapper** around a **function** that **alters** its behaviour. The main job is still **carried** out by the **function**. It can be seen as **“features”** or **“aspects”** that can be added to a function. We can add one or add many of these features **without changing** the original function code! Decorators are also sometimes called **higher order functions**, or **functional composition**.

Use cases:

We can use decorators for many things, but some typical examples include adding **logging**, **timing**, **validation** or **caching** features.

Decorators and forwarding

Use case: adding logging/timing information

```
function printGreeting(name) { // simple undecorated function
    console.log('Hello, ' + name);
}
printGreeting('Undecorated');

function loggingTimingDecorator(originalFunction) { // decorator takes a function as parameter
    return function (name) { // and returns that function with extra bits - timing/logging
        console.time('Function timer'); // start a timer
        console.log(`\nExecuting function ...`) // log a message
        const result = originalFunction(name); // execute the original function and store result
        console.timeEnd('Function timer'); // stop the timer
        return result; // return the result of running the original function
    }
}

// returns the original function WITH the timing/logging features included
const decoratedPrintGreeting = loggingTimingDecorator(printGreeting);
decoratedPrintGreeting('Decorated') // we can still call the decorated version in the same way
```

Decorators and forwarding - caching

```
function slow(x) {
    // there can be a time-consuming job here, like adding up to a large number
    let random = 0, goal = Math.floor(Math.random() * x * 1_000_000); // random large number
    console.log(`slow(${x}): randomly generated goal for ${x * 1_000_000} is ${goal}`);
    for (let i = 0; i < goal; i++) random++;
    return random; // return large number after counting to it
}
function cachingDecorator(origFunction) { // decorator takes a function as parameter
    const cache = new Map(); // can also include outer environment variables via a closure

    return function(x) { // decorator returns same function with extra bits - caching
        if (cache.has(x)) { // if the key exists in the cache,
           console.log('returned cached value for ' + x); return cache.get(x); // read and return the result from it
        }
        let result = origFunction(x) // otherwise, call the original function and store the result
        cache.set(x, result); // then cache (remember) the result for next time
        return result;
    };
}
const fast = cachingDecorator(slow) // we can decorate the original slow function to use caching and make it fast
const fastTimed = loggingTimingDecorator(fast) // we can decorate the fast version to include timing for testing
fastTimed(8) // first time will still be slow because it's uncached
fastTimed(8) // but every time after this will be much faster because result is cached
```

Decorators and forwarding

Decorators

The previous decorator function examples work by directly executing the original function with a **fixed**, **known number of arguments**, eg:

```
let result = origFunction(x) // otherwise, call the original function and store the result
```

However, we usually don't know how many arguments the original function has, and need to make it **generic** so that it supports any amount. Our current examples also work only on functions with no reliance on context, or **this**. To truly make a reliable decorator function, we need a way to **forward** the execution of the original function, **regardless of its arguments and context**.

Decorators and forwarding - call/apply

func.apply and func.call

The previous decorator function examples work by directly executing the original function with a **fixed**, **known number of arguments**, eg:

```
func.call(context,  . . .args);
func.apply(context, args);
```

- Both **call** and **apply** are used to **bind** the **context** to a function **func** when executing it, and pass in a variable number of arguments **args**:
  - call accepts a comma-separated list of **args**, or use the spread syntax … to unpack and pass an **iterable variable args** as the list.
  - apply accepts only an **array** or **array-like** variable containing the args.
- For objects that are both **iterable and array-like**, such as a **real array**, we can use either. apply will probably be faster, because the JS **engine optimises** it.

Decorators and forwarding - call/apply

Forwarding with **call** or **apply** is a way to **pass all arguments** along with the **context** when calling a function. Now our decorator can add timing/logging for functions with any number of arguments. Try to fix the ***cachingDecorator*** as well.

```
function loggingTimingDecorator(originalFunction) { // same decorator function as before
    return function () { // BUT now the returned function doesn't name its arguments from here
        console.time('Function timer');
        console.log(`\nExecuting function ...`)
        //const result = originalFunction(name); // WON'T work as name is now undefined
        //const result = originalFunction.call(this, ...arguments) // WILL work, no matter how many args
        const result = originalFunction.apply(this, arguments) // and so does this - try out both
        console.log(arguments); // [Arguments] { '0': 8 }
        console.timeEnd('Function timer'); // stop the timer
        return result; // return the result of running the original function
    }
}
```

Decorators and forwarding - call/apply

Using **call** and **apply** also ensures the right **context** is used when forwarding function execution. This is needed when the function relies on **this**.

```
let worker = {
    getMultiplier() {
        return Math.floor(Math.random() * 1_000_000); // large random number
    },

    slow(x) {
        let random = 0, goal = x * this.getMultiplier(); // needs context to work
        for (let i = 0; i < goal; i++) random++;
        console.log(`worker.slow(${x}): randomly generated goal is ${goal}`);
        return random; // return large number
    }
};
worker.slow(5) // works, context comes from before the dot, ie. worker
worker.fast = cachingDecorator(worker.slow) // without call/apply, context is lost
worker.fast(3) // TypeError: Cannot read properties of undefined (reading 'getMultiplier')
```

Decorators and forwarding - call

call Use Case: Borrowing a method

Many utility functions such as **map**, **filter**, **slice**, etc. can be accessed through an instance of the object they are attached to. For example, **map()** is defined on the **Array** object, and can be called on any **array instance**. We can **borrow** these utility functions using **call** to supply a different **instance** as the **context**:

```
function isOdd(number) { return number % 2; } // returns true if number is odd, false otherwise

function getOddNumbers() {
    // arguments is not an array, but it 'borrows' the filter function from Array by using call
    return [].filter.call(arguments, isOdd); // arguments is context, isOdd is parameter for filter
}

let results = getOddNumbers(10, 1, 3, 4, 8, 9);
console.log(results); // [ 1, 3, 9 ] (array of all odd arguments)
```

Decorators and forwarding - call

call Use Case: Inheriting from a constructor function

We can use call inside a constructor function to **inherit from a parent**, by passing a custom **context** representing the child, as well as any **arguments**:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
    this.salePrice = price * .9; // 10% off
}

function Food(name, price) {
    Product.call(this, name, price); // inherits from Product with custom context
    this.category = 'food';
}

const cheese = new Food('cheese', 5);
console.log(`${cheese.name} is a ${cheese.category} and costs $${cheese.price} ($${cheese.salePrice} on sale)`);
```

Function binding

Issue of losing this

If a function relies on context (**this**) and is passed as a **reference** instead of being **called directly**, its context is lost.

```
const user = {
    name: 'John',
    sayHi() {
        console.log(`Hi, ${this.name}`)
    }
}
user.sayHi() // called directly, works! Hi, John
setTimeout(user.sayHi, 1000) // passed as reference, fails! Hi, undefined
```

Function binding

Solution 1 for missing this

We can **wrap** it **inside** a function. This allows the context to come from **before the dot** (ie. the **user** object) and references to **this** work as expected.

```
const user = {
    name: 'John',
    sayHi() {
        console.log(`Hi, ${this.name}`)
    }
}
setTimeout( function() { user.sayHi(); }, 1000 ) // works! Hi, John
setTimeout( () => user.sayHi(), 1000 ) // same as above, arrow function (more common)
```

Function binding

Solution 2 for missing this

We can explicitly bind the right "context" into the function reference.

- Basic syntax

```
const boundFunc = func.bind(context);
```

- The fix with bind

```
const user = {
    name: 'John',
    sayHi() {
        console.log(`Hi, ${this.name}`)
    }
}
const boundSayHi = user.sayHi.bind(user) // new function reference with user context explicitly bound
setTimeout( boundSayHi, 1000 ) // works! Hi, John
```

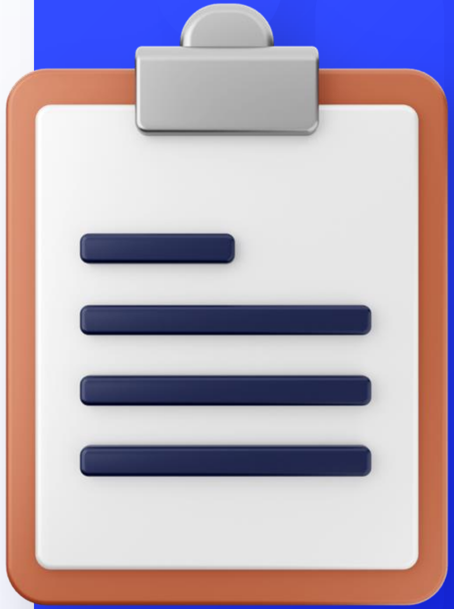
Apply, Call, Bind - Summary

Call, **apply** & **bind** are special methods that can be called on any **function**. They allow us to access and override the **context** and **arguments** for that function.

Normally, **this** in a function refers to the object it was called on. With apply(), call() and bind(), we can set **this** to any value when calling a function, customising the **context**. This allows us to use methods of one object as generic utility functions.

Apply and **call** also allow us to customise the **arguments** for the function. **Apply** is slightly more efficient, but they are very similar, with the main difference being:

- call() - function arguments are passed individually as a list
- apply() - function arguments are combined in one iterable object, typically an array



Prototypes

Prototypes in JavaScript enable objects to **inherit** features from each other in **hierarchical (parent-child)** relationships.

Every JavaScript object has a built-in property called its **prototype**. Any **parent** objects also have a prototype, forming a **chain** of prototypes.

When you attempt to access a **property** of an object, JavaScript looks for it in the **object first**. If not found, it searches in the object's **prototype** and continues up the **chain**. If the property is not found, the search stops, and undefined is returned.

Call, **apply** & **bind** are properties that belong to the **Function** prototype - there are many others on the **Object**, **String**, **Date**, **Number**, **Array** etc. prototypes.

See Object Prototypes on the MDN for more information.

Prototypal inheritance

[[Prototype]] property

In JavaScript, objects **inherit** properties and functions from a **prototype** stored in a special hidden property [[Prototype]] (as named in the specification), that is either null or **references another object** (such as Object).

It can be accessed in a few ways. Object.getPrototypeOf is the most reliable:

```
let animal = { eats: true, sleeps: true, legs: 4, mammal: true }; // inherits from Object prototype

let animalPrototype = Object.getPrototypeOf(animal); // recommended way to get prototype

console.log(animalPrototype); // BUT printing it via console.log is incomplete

console.log(Object.getOwnPropertyNames(animalPrototype)); // prints all prototype (Object) properties
```

Prototypal inheritance

- Object.setPrototypeOf (simplest) or Object.create (most control over property descriptors) are the **recommended ways** to **set** a prototype

```
let rabbit1 = { jumps: true };
Object.setPrototypeOf(rabbit1, animal); // NEW recommended way, uses default property descriptor settings

let rabbit2 = Object.create(animal, { // creates a new object from prototype, with custom properties
    jumps: { // name of custom 'own' property for rabbit object
        value: true, // property descriptor to set the property value
        enumerable: true // property descriptor to make this enumerable - otherwise jumps won’t be in for...in
    }
});
console.log(rabbit1, rabbit2); // { jumps: true } - only prints 'own' properties, not inherited ones
console.log(rabbit1.legs, rabbit2.legs); // 4 - inherited properties do exist
for (let prop in rabbit1) console.log(`${prop} is ${rabbit1[prop]}`) // own properties, then inherited ones
for (let prop in rabbit2) console.log(`${prop} is ${rabbit2[prop]}`) // own properties, then inherited ones
```

Constructor Function Prototypes

Constructor functions can also use prototypal inheritance, using a syntax called F.prototype. This is a regular **property** named prototype on F, which is a constructor function with the **first letter capitalised**.

```
function Rabbit(name) { // constructor function, first letter capitalised by convention
    this.jumps = true;
    this.name = name;
}
Rabbit.prototype = animal; // sets the prototype to inherit from (same animal object as previous)

let whiteRabbit = new Rabbit('White Rabbit');
console.log(whiteRabbit); // { jumps: true, name: 'White Rabbit' } - own properties
for (let prop in whiteRabbit) console.log(`${prop} is ${whiteRabbit[prop]}`); // all properties
```

Native Prototypes

The prototype **property** is widely used by the core of JavaScript itself. All **built-in constructor functions** (eg. Array, Object, Date, String, Number) use it.

- Object.prototype

```
const obj = {} // simple empty object
console.log( Object.getPrototypeOf(obj) === Object.prototype ) // true: its prototype is Object prototype
console.log( Object.getOwnPropertyNames(Object.getPrototypeOf(obj)) ) // inherited properties from Object prototype
```

- Other **built-in** prototypes: Array, Date, Function etc. also keep **methods** in **prototypes**. *Try the below for Date and String!*

```
const arr = [] // simple empty array
console.log( Object.getPrototypeOf(arr) === Array.prototype ) // true: its prototype is Array prototype
console.log( Object.getOwnPropertyNames(Object.getPrototypeOf(arr)) ) //inherited properties from prototype
```

Native Prototypes

Changing native prototypes

Native prototypes can be **modified**. **BUT**, **BUT**, **BUT** it is **not** recommended to do so unless it is **polyfilling**. Polyfilling is a piece of code used to provide modern JS functionality on older browsers that do not natively support it.

```
String.prototype.show = function() { // creates new 'show' function on built-in String prototype
    console.log(this);
};
"BOOM!".show(); // BOOM! - we can now call .show() on any string, since they all inherit from the
prototype

// polyfilling for String.prototype
if (!String.prototype.repeat) { // if there's no such function in the prototype already
    String.prototype.repeat = function(n) { // define a repeat function to repeat the string n times
        return new Array(n).join(this); // uses the string ('this') as glue to join n empty array items
    };
}
console.log( "La".repeat(3) ); // LaLaLa
```

Native Prototypes

Borrowing from prototypes

Some methods of native prototypes are often **borrowed**, to provide the same functionality on a different type of variable.

```
// define our own join() function for objects
const obj = {
    0: "Hello",
    1: "world",
    length: 2, // needed for join to work
};

obj.join = Array.prototype.join; // adds a join function to THIS object that uses Array.join()
Object.prototype.join = Array.prototype.join; // adds a join function to ALL objects

console.log(obj.join(',')); // Hello,world
```

Classes

In modern JavaScript, we can use **classes** as a template for creating objects and encapsulating data and functions. These are built on prototypes but with extra syntax and features which are useful for **object-oriented** programming.

```javascript
// a class is like a template or blueprint
class ExampleClass {

  // each instance of the class will have any properties
  prop1 = 'value1';
  prop2 = 'value2';

  constructor() {
    // constructor function creates a new instance of this class
  }

  method1() {
    // methods are functions of the class
  }
}
```

Object Oriented vs. Functional

- **Object-Oriented Programming (OOP)** is a programming paradigm that organises code into **objects**, each representing an instance of a **class** with its own data and behavior. Objects can interact with each other through defined interfaces.
- **Functional Programming (FP)** is a programming paradigm that organises code into **functions**. It emphasizes immutability, avoiding side effects, and the use of functions as first-class citizens.
- In simple terms, OOP is centered around objects with specific roles and behaviors, while FP focuses on functions and their interactions with data.

Historically FP was the main way of programming. Around the 90s OOP became popular with languages such as C++ and Java. Today FP has gained favour again and both are used.

Classes: Inheritance

Classes can **inherit** from a parent class (prototype) via the extends keyword.

```
class Animal {
    constructor(name) {
        this.speed = 0; this.name = name;
    }
    run(speed) {
        this.speed = speed;
        console.log(`${this.name} runs with speed ${this.speed} kph.`);
    }
    stop() {
        this.speed = 0;
        console.log(`${this.name} stands still.`);
    }
}
class Rabbit extends Animal {
    hide() { // custom function, also inherits from Animal
        console.log(`${this.name} hides!`);
    }
}
let bunny = new Rabbit('bunny'); // bunny contains properties and methods from Animal and Rabbit
bunny.run(9); // bunny runs with speed 9 kph.
bunny.hide(); // bunny hides!
```

Classes: Inheritance

Methods and **properties** can be **inherited** from a parent and then **overridden** in child classes. The super keyword refers to the **parent** class.

```
class Rabbit extends Animal {

    stop() { // overrides stop method in parent class
        super.stop(); // call parent stop
        this.hide(); // and then hide
    }

    hide() { // custom function, also inherits this.name from Animal
        console.log(`${this.name} hides!`);
    }
}

let bunny = new Rabbit('bunny'); // bunny contains properties and methods from Animal and Rabbit
bunny.run(9); // bunny runs with speed 9 kph.
bunny.stop(); // bunny stands still. bunny hides!
```

Classes: Inheritance

Constructors can also **inherit** from a parent, with extra child-specific class code. The super keyword in a child constructor calls the parent class constructor.

```
class Rabbit extends Animal {

    constructor(name, earLength)
    {
        super(name); // call the constructor function of the parent, inherited Animal class
        this.earLength = earLength; // adds custom properties only for instances of Rabbit
    }
    stop() { // function overridden from parent class
        super.stop(); // call parent stop
        this.hide(); // and then hide
    }
    hide() { // custom child class function
        console.log(`${this.name} hides!`);
    }
}

let bunny = new Rabbit('bunny', 8); // bunny contains properties and methods from Animal and Rabbit
console.log( bunny.earLength ) // 8
```

Classes: Inheritance

- Child classes also **inherit**, and can **override**, **class properties**, such as type below:

```
class Animal {
    type = 'animal';
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }
    describe() {
        console.log(`${this.name} is a ${this.type}`)
    }
    //... previous Animal methods go here
}
class Rabbit extends Animal {
    type = 'rabbit';
    //... previous Rabbit methods go here
}

new Rabbit('bunny').describe() // bunny is a rabbit
new Animal('fuzzy wuzzy').describe() // fuzzy wuzzy is a animal
```

Classes: Static methods & properties

- We can assign **methods** and **properties** to the **class** itself, **using** the static keyword. These static methods/properties belong to the **class**, and are **inherited** by any child classes, but **don't belong** to any particular **instance**.

```javascript
class Person {
    static latin = 'persona'; // static (class) property, belongs to class not any instance
    constructor(name) {
        this.name = name; // standard property, is unique to each instance of the class
    }
    getName() { // standard method, belongs to each instance of the class
        return this.name;
    }
    static createAnonymous() { // static (class) method, belongs to class not any instance
        return new Person("Unnamed Person");
    }
}
let jonas = new Person('Jonas')
console.log( jonas.getName() ) // Jonas - name and getName() belong to an instance of Person
console.log( jonas.latin ) // undefined - latin property doesn't belong to jonas
console.log( Person.latin ) // persona - latin property belongs to Person class
let anon = Person.createAnonymous() // use static (factory) method to create generic Person instance
```

Classes: Protected methods & properties

So far our classes have only used **public** properties & methods, so all instances can use these as needed. However, an important principle of **object-oriented programming** involves **delimiting** the **internal** interface from the **external** one. We can do this in a few ways: one is to signify properties or methods as **protected** by using an underscore (convention only, not enforced by JS):

```
class Laptop {
    _hardDiskType = 'HDD'; // protected property, meant to be internal

    constructor(brand) {
        this.brand = brand; // public property, can be used externally by instances
    }
    getHDiskType() { return this._hardDiskType; } // public method to access protected property
}
const macbook = new Laptop('Macbook Pro');
console.log(macbook.brand) // public property, accessed externally from any instance
console.log(macbook._hardDiskType) // works, not recommended as it violates encapsulation principles
console.log(macbook.getHDiskType()) // recommended way to access protected property
```

Classes: Private methods & properties

Private properties & methods are enforced by JS, and **cannot be accessed outside of the class itself**. This lets us maintain certain data as internal only.

```
class Laptop {
    _hardDiskType = 'HDD'; // protected property, SHOULD only be used by inheriting classes
    #numCPUFans = 1; // private property, CAN only be used internally by class methods

    constructor(brand) { // constructors are always public
        this.brand = brand; // public property
    }
    isGaming() { return false; } // public method
    getHDiskType() { return this._hardDiskType; } // public method to access protected property
    _increaseCPUFans() { // protected method
        if (this.isGaming()) this.#numCPUFans++ // can access private properties internally
    }
}
const macbook = new Laptop('Macbook Pro');
console.log(macbook.#numCPUFans) // error: private property is not accessible
```

Institute of Data

Classes: Private methods & properties

Classes that inherit **protected** properties and methods can access and modify them, but **private** properties and methods cannot even be accessed by children.

```
class GamingLaptop extends Laptop {

    constructor(brand) {
        super(brand); // public property, externally available to instances
        this._hardDiskType = 'SSD'; // protected props should be accessed by children, not instances
        this.#numCPUFans = 2; // error: private property is not accessible
        this._increaseCPUFans(); // use protected method to change #numCPUFans as it's internal
    }
    isGaming() { return true; } // public method
}
const alienware = new GamingLaptop('Alienware');
//console.log(alienware.#numCPUFans) // error: private property is not accessible
console.log(alienware._hardDiskType) // no error: but protected property SHOULD NOT be accessed
console.log(alienware.getHDiskType()) // better: public method for accessing protected property
```

Classes: Summary

Classes are a **template** for creating **objects**. They encapsulate **data** with **methods** to work on that data. Classes in JS are built on **prototypes** but also have some syntax and semantics that are unique to classes.

Key features of classes:

- **Constructor** (called when creating an object as an instance of the class with **new**)
- **Instance methods** and **fields** (defined in the class, belong to an object instance)
- **Static methods** and **fields** (defined in the class, belong to the class itself)
- **Inheritance** (via the **extends** keyword - children inherit/override methods and fields)
- **Encapsulation** (via **_protected** and **#private** methods/fields)

See MDN: Using Classes for more information and examples.

Error Handling

We can define our own way of handling errors in JS before they crash our code.

try…catch syntax

- The **try…catch** construct has two main blocks: try, and then catch.

```
try {
    // code . . .
} catch (err) {
   // code . . .

}
```

- Only works for **runtime** errors, the code must be **runnable** as valid

```
try {
    const error = "mismatched quotes'
} catch (error) {
    console.log('will not catch above error')
}
// SyntaxError: Invalid or unexpected token - doesn't go to catch block
```

Error Handling

Non-syntactical errors can be **caught** and handled, allowing other code to run:

```
try {
    noSuchVariable;
} catch (error) { // error is just a variable name. 'error', 'err' or 'e' are all commonly used
    console.log('caught an error: '+ error.message) // all errors have a message property
}
// caught an error: noSuchVariable is not defined
console.log('even though an error occurred above, it was caught so code continues');
```

Only **synchronous** errors can be caught; **asynchronous** errors still cause a crash:

```
try {
    setTimeout( () => noSuchVariable, 1000 );
} catch (error) { // error is just a variable name. 'error', 'err' or 'e' are all commonly used
    console.log('only synchronous errors! ' + error.message) // all errors have a message property
}
console.log('prints synchronous code then throws uncaught asynchronous error after 1 sec');
```

Error Handling

The throw operator can be used in the **try** or **catch** block to intentionally cause a new or existing error to crash the application for a specific reason:

```
function checkJson(json) { // checks json argument for validity and ensures a name property
    try {
        const user = JSON.parse(json); // parse string into object
        if (!user.name) {
            throw new SyntaxError("Incomplete data: no name"); // we can throw our own custom errors
        }
        return true; // returns true (valid json) if no error was thrown above

    } catch (err) {
        if (err instanceof SyntaxError) { // once caught, we can do specific things based on error type
            console.log( "JSON Error: " + err.message );
        } else {
            throw err; // rethrow other non-syntax errors; invalid json will still cause a crash
        }
    }
    return false; // returns false if any error occurred
}
```

Error Handling

The finally clause is used when something in the **try…catch** block needs **finalising** in any **case of outcome**, regardless of any thrown **errors** or **return** statements.

```
function checkJson(json) {
    try {
        // ... as above
        return true;
    } catch (err) {
        //... as above
    }
    finally {
        console.log('at the end'); // always prints, even if returning true or throwing an error
        // used to complete operations and perform cleanup regardless if we hit errors or not,
        // eg. closing db connections, removing interval timers, cancelling event listeners, etc
    }
    return false; // returns false if any error occurred
}
```

Institute of Data

Promises

A Promise represents the **eventual** completion (or failure) of an **asynchronous** operation. It provides a way to **wait** for an **unknown** period of time and **then** execute certain code once a **result** (or an **error**) is returned.

We can **produce** (create) promises explicitly, but most often we will **consume** (use) promises from other asynchronous code, such as **database manipulation** or **HTTP requests**.

- **Producer Syntax**: Promise constructor takes a **single argument** which is a **function**, with **resolve** and **reject** callback functions as arguments.

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

Promises

Consumer Syntax: uses the **then**, **catch** and **finally** functions of an existing promise to define what should happen when the asynchronous operation succeeds or fails.

- then - function that executes when promise **resolves** (success)

```
promise.then( (result) => console.log(result), // prints if/when promise resolves successfully
    (error) => console.error(error) ) // optional, prints if/when promises completes with error
```

- catch - function that executes when promise **rejects** (failure)

```
promise.then( (result) => console.log(result) ) // prints if/when promise resolves successfully
    .catch( (error) => console.error(error) ) // prints if/when promises completes with error
```

- finally - function that executes when promise **settles** (either success or failure)

```
promise
    .finally( () => console.log('promise is settled') ) // prints when promise settles
    .then( (result) => console.log(result) ) // prints if/when promise resolves successfully
    .catch( (error) => console.error(error) ) // prints if/when promises completes with error
```

Promises

Simple example using **arrow functions**, which illustrates how we can **create** (produce) a simple promise returning a delayed, **random success/failure outcome**, and then **respond to** (consume) the outcome scenarios.

```
// example promise. settles after 250ms with success or failure depending on random number
const promise = new Promise((resolve, reject) => { // resolve/reject are callback functions
    if (Math.random() > 0.5) setTimeout( () => resolve('Random number ok'), 250 ) // success
    else setTimeout( () => reject('Random number too low'), 250 ) // failure
})

promise // consume the promise by responding to outcomes when they happen
    .finally( () => console.log('Wait is over, promise has settled.') ) // always prints
    .then( (result) => console.log('Success! ' + result ) ) // prints resolve msg
    .catch( (error) => console.log('Error! ' + error ) ) // prints reject msg
```

Promises: fetch

fetch is a browser-based function that **sends HTTP requests** to retrieve data from other servers. Since this type of operation is **asynchronous**, it returns a **promise** we can use to process the results **once it completes**.

```
<html> <!-- very basic fetch example demonstrating real-world promises -->
<body>
    <h2>Check the Dev Inspector Console</h2>
    <script>
        fetch('https://reqres.in/api/users') // request data from this server
                    // when it completes, access the JSON from the HTTP response sent by resolved
promise
            .then(response => response.json()) // .json() also returns a promise
            .then(json => console.log(json)) // log the returned JSON to the browser console
            .catch(error => console.error(error)) // if there was an error, log that too
    </script>
</body>
</html>
```

Promises: chaining

- If we have a **sequence of asynchronous** tasks to be performed **one after another** (as in the previous **fetch** example), we can **chain** the .then() calls one after the other. Any value returned from .then() is itself a **promise**:

```
let start = 10;
new Promise((resolve, reject) => {
    resolve(start); // resolve promise successfully with value of 10
}).then((result) => { // when resolve is called, it triggers .then()
    console.log(result); return result + start; // values returned from .then() are also promises
}).then((result) => { // so we can chain them together
    console.log(result); return result + start; // increasing result by 10 each time
}).then((result) => { // we can continue to chain them together
    console.log(result); return result + start; // increasing result by 10 each time
});
// prints 10, 20, 30
```

Promises: returning promises

- A handler, used in .then(handler) may explicitly create and return a **promise**, which can also be used for promise chaining:

```
let start = 10;
new Promise( resolve => setTimeout(() => resolve(start), start * 10)
).then(result => { // promise handler function inside .then()
    console.log(result); let next = result + start;
    return new Promise( resolve => setTimeout(() => resolve(next), next * 10) );
}).then(result => { // can explicitly return new promises
    console.log(result); let next = result + start;
    return new Promise( resolve => setTimeout(() => resolve(next), next * 10) );
}).then(result => { // which use the results of previously resolved promises in the chain
    console.log(result); let next = result + start;
    return new Promise( resolve => setTimeout(() => resolve(next), next * 10) );
});
// prints 10, 20, 30, but with 100, 200 and 300ms delays in between
```

Promises: static methods

- Promise.all(promises): accepts an iterable of promises, waits for all promises to resolve and returns an array of their results. If **any of the given promises rejects**, it becomes the **error of Promise.all**, and all other results are **ignored**.
- Promise.allSettled(promises): accepts an iterable of promises, waits for all promises to **settle** (either success or failure) and returns their **results as an array of objects** with:
  - status: "fulfilled" or "rejected"
  - **value** (if fulfilled) or **reason** (if rejected)
- Promise.race(promises): accepts an iterable of promises, waits for the **first promise to settle**, and its result/error becomes the outcome.
- Promise.any(promises): waits for the first promise to fulfil, and its result becomes the outcome. If all of the given promises are rejected, AggregateError becomes the error of Promise.any.
- Promise.resolve(value): makes a **resolved promise with the given value**.
- Promise.reject(error): makes a **rejected promise** with the given error.

Promises: async/await

async and await are two important keywords that go hand in hand, and can be used to force promises to behave **synchronously** - ie to **wait** until the promise resolves before executing the rest of the code body. They replace the .then() and .catch() syntax of asynchronously processed promises.

```
const promise = new Promise((resolve) => {
    setTimeout( () => resolve('Simple successful promise'), 250 )
});

// using .then to process asynchronously:
promise.then(msg => console.log(msg));

// using await to process synchronously (if using await in a function it needs to be async):
let msg = await promise;
console.log(msg);
```

Promises: async/await

async function

The word **async** before a function means a **function** always returns **a promise**. It is **required** when the function body includes an **await** statement. The **async** and **await** keywords enable **asynchronous**, **promise-based behavior** to be written more **concisely** using **synchronous style code**.

```
async function asyncFunctionDeclaration() { ... } // function declaration syntax

const asyncFunctionExpression = async function() { ... } // function expression syntax

const asyncFunctionArrow = async () => { ... } // arrow function syntax
```

Promises: async/await

await keyword

Must be used inside an **async** function. It makes JavaScript **wait** until that promise **settles** and returns its **result**. Any errors can be caught with a **try…catch**.

```
async function waitForPromise() { // async function allows synchronous promise handling internally
    // since we have synchronous code and no .catch(), we use try ... catch for errors
    try {
        let promiseResult = await promise; // waits here as long as promise needs to resolve
        console.log(`Success: ${promiseResult}`) // then continues executing other code
        return true;

    } catch(error) {
        console.error(`Failure: ${error.message}`)
    }
    //only gets here if return true above did NOT happen, ie. there was an error
    return false;
}
```

Promises: Summary

Promises are a way to handle asynchronous operations in JavaScript, allowing you to work with code that will complete in the future. Promises can be in one of three states: **pending**, **fulfilled**, or **rejected**.

Key features of promises:

- **Creation**: A promise is created using the **Promise** constructor, which takes a function with **resolve** and **reject** parameters, which themselves are functions.
- **Then** and **Catch**: `.then()` handles the result when a promise is **fulfilled** (resolved), and `.catch()` handles errors when a promise is **rejected**.
- **Async/Await**: Keywords to handle promises in a cleaner, more intuitive way that resembles synchronous code
- **Error Handling**: Errors are propagated down the chain, making it easy to manage failures.
- **Parallel Execution**: Multiple promises can run in parallel using `Promise.all()` or `Promise.race()`.

For more details and examples, see MDN: Using Promises.

Class activity

Work through the following exercises to practice some of these advanced JS concepts:

Mimic a pizza making procedure, by writing code that prints statements in the below order:

```
• Started preparing pizza ...
• Made the base
• Added the sauce and cheese
• Added the pizza toppings
• Cooked the pizza
• Pizza is ready
```

- **Task 1**: Create 6 JS functions which print the pizza processing statements and call those functions in sequence. Use a mix of function declarations, expressions and arrow functions.
- **Task 2**: Make the functions asynchronous by using setTimeout with different time durations, maintaining the right order.
- **Task 3**: Modify the asynchronous functions to use Promises and achieve the required result.
- **Task 4**: Modify the functions to use async/await and achieve the required result.

Revision

- How does the concept of ‘scope’ affect variables in JS?
- What is a closure? When would it be useful?
- What are some built-in function properties? Can we add custom ones?
- What is the difference between **setTimeout** and **setInterval**? What do they do?
- What is a decorator function? When would it be useful?
- How can we pass an unknown number of arguments to a generic function?
- How can an object inherit properties and methods?
- What is a class? When would it be useful?
- How can we handle errors and prevent them crashing our code?
- What is a promise? How can we use it?