<JavaScript - Interpreted />

The Fundamentals

Polyfilling

- making a function that doesn't exist due to modern updates

Script tag

- inline JavaScript
- obselete attributes:
  - `type` (e.g. text/javascript)
    - modern HTML redefined the definition
  - `language`: JavaScript is default now
- useful attributes:
  - `src = "path_to_js_file"`
    - useful because it can be cached
    - if `src` is set, the inline code will be ignored

Semicolons

- There is implicit semicolons (automatic semicolon insertion)
- based on lexical analysis and terminating token such as ')' or '}'
- **should always write semicolons - not dependable**

Comments

- cannot be nested 😛

Modern JS

- ES5 modified some of the existing functionality and are off by default
- `use strict` enables the ES5 changes
- always have it on top of the scope (globally or function body)
- cannot be cancelled part way

Variables

- `var`:
  - old
  - always processed at the beginning of script (regardless of actual declaration position or if conditions are met) (hoisting)
  - *assignments* are not hoisted (not pushed to the top): ran sequentially in the order of statement position
  - has no block scope or loop scope: only function-scoped or global

```
if (true) {
  var test = true; // use "var" instead of "let"
}
alert(test); // true, the variable lives after if
```

- `let`: has block scope(e.g. inside `if`)
- `const`: cannot be assigned after first assignment (usually in capital letters)
- naming: can only use numbers, letters, $, and _
- **tldr: `var` declarations are hoisted and minimum function scoped while `let` can be block scoped**

IIFE (obscelete)

- immediately invoked function expressions
- used for instant variable declarations
- must be wrapped in ()

```
(function() {
  let message = "x";
})
```

Numbers

- `NaN` takes precedence: propagates through all expressions if exists
- biggest number is 2^53
- if the number ends in 'n' it means "bigint"

Strings / Characters

- no such thing as a char type in JS
- it is immutable: `a+="asdfaDS"` actually makes a new string

Null

- just means empty or nothing or unknown

Undefined

- not assigned/defined
- can technically assign `undefined`, but should just use `null`

Objects/Symbols

- everything else is **primitive** since it's only 1 single thing
- symbols are used to create unique identifier for objects

typeof

- defined as operator or function
- ```
  typeof undefined // "undefined"
  typeof 0 // "number"
  typeof 10n // "bigint"
  typeof true // "boolean"
  typeof "foo" // "string"
  typeof Symbol("id") // "symbol"
  typeof Math // "object"  (1)
  typeof null // "object"  (2) - techically wrong(error within language)
  typeof alert // "function"  (3)
  ```

Boolean

- "0" is true
- " " also true(any non-empty string)

Conversion

- having "+" with a string always results a string(e.g. '3' + 2 = '32')
- it still runs left to right (2 + 2 + '3' = '43')
- all other operators convert to numbers

Increment / decrement

- count++ is the same as ++count unless the return value is used
- count++ returns count and then add
- ++count returns count + 1

Comma operator

- evaluate multiple expressions, but keep only the last one

String comparison

- lexicographical order(O(n) -> compares every string until one differs)
- it compares to the index of the js 'dictionary'(unicode)
- lowercase > uppercase

Special: "0" is true but 0 is false. but 0 == "0"

== (regular) vs === (strict)

- `==` uses type conversion
- `===` compares types first
- `null === undefined`: false
- `null == undefined`: true
- in math comparisons: null -> 0 and undefined -> NaN
- `null == 0` is false while `null >= 0` is true
  ```

Special: equality check does not convert to number while comparisons (>, >=, etc) does

Conditionals

- 0, empty string, null, undefined, NaN are all falsy
- conditional(ternary) operator should only be used with assignment, not function calls (designed this way, but it still works)
- ternary *cannot* contain break/continues even if it's in a loop, must use `if`

OR operator (find first truthy value)

- if used as assignment, it will return the first 'truthy' evaluation or the last operand (e.g. x will be undefined under `true || (x = 1)`)

AND operator (find first falsy value)

- if used as assignment, it will return the first 'falsy' evaluation or the last operand

Special: the precedence of && is higher than || so use brackets

Labels

- used for references outer loops in nested loops
- can directly break up to the labeled loop using `break <label>;`
- usage: `<label> : for(let...)...`
- must be of higher scoped to break/continue (not a "skip to line x")

Switch

- must include `break` or it will run all the cases under the truth one
    - can be used to group cases together
- case comparisons are strict (no object conversion)

Function scoping

- will use outer variables unless local one is defined (shadowing)
- if local one is defined, it is *not* linked to the outer variable
- functions will always get a *copy* when passed in parameter(only primitive types)

Default values

- can define defaults to be anything (not just string, can be function return value)

Return statement

- if a return value is not provided or no return statement, it returns `undefined`;
- multi-lined return statements must be enclosed in ()

Functions

- declarations: `function test() {}`
  - this is visible to everywhere inside the scope, before or after
  - JS looks for global func declarations and creates it first
  - in `use strict`, declarations are only available inside code block
- expressions: `let x = function() {};`
  - same as assignment; will not exist until the statement has been processed
- function is a "special value": when calling it without (), it shows the source code as a string
- can copy functions (e.g. `let y = x;` or `let y = test;` will give y the function of x)
- function expressions require ';' because it's an assignment, so it is used to terminate the statement

---

Objects: The Basics

Objects

- always stored/accessed by reference to memeory
- object comparisons are done by checking memory, not value
- `const` objects *properties* can be changed - memory doesn't change since reference is same
  - allowed: `const_object.key = "new value";`
  - not allowed: `const_object = { key: "same value" };`
- simple cloning (depth of 1):
  - option 1: iterate and assign to new object
  - option 2: `Object.assign(destination, [object1, object2, ...]);`
    - takes all properties of the array of objects and puts into destination object
    - duplicate keys will be *overwritten*
    - returns the new object
    - this operates the same as option 1, but looks nice
- keys do not require quotes unless it contains spaces
- accessing multi-word keys should be done with `object['key with spaces']`
- `delete object.property`
  - returns true if successful, false otherwise
  - nothing to do with freeing memory, only breaks the reference (otherwise properties would be floating randomly in memory)
- object property names have no restrictions (can use keywords)
- way to test if key has been created (regardless of value): `"property" in object`
- looping through objects:
  - `for (let key in object) {}`
  - only integral keys are sorted (means if we convert to number and back, it's still the same)
    - can be bypassed by appending "+" infront of the integers so that it's not the same after `f(f^-1(x))`
  - other keys are kepy in creation order
- **property value shorthand**
  - if input param is the same as keyname, you can ignore the value input
  - ```
    function makeUser(name, age) {
    return {
    ```
    ```

```
    name, // same as name: name
    age   // same as age: age
    // ...
  };
```

}

- **computed properties**
  - can use `[]` to use computed property as the key instead of the word itself

```
let fruit = prompt("Which fruit to buy?", "apple");

let bag = {
  [fruit]: 5, // the name of the property is taken from the variable fruit
};

alert( bag.apple ); // 5 if input to prompt was apple
```

Objects - `this`

- references the object it is scoped in
- `this` is evaluated at call time, not the position of declaration
- can be called inside any function
  - if the function is not assigned to an object, it will return `undefined`
- arrow functions have no `this`, it will reference the outer 'normal' function

Advanced `this`

- ```
  let user = {
  name: "John",
  hi() { alert(this.name); },
  bye() { alert("Bye"); }
  };

  user.hi(); // John (the simple call works)

  // now let's call user.hi or user.bye depending on the name
  (user.name == "John" ? user.hi : user.bye)(); // Error!
  ```

- object dot method works (`user.hi()`)
- evaluated method doesn't
- *tldr*: `obj.method()` uses the `.` to get the property and then `()` to execute it
- evaluated methods fail to get this because it becomes `let hi = user.hi` first, which no longer has scoping (global `this` returns undefined)
  - it assigns it to the external brackets as a temp variable
  ```

- *JavaScript returns using a special type called the 'reference type' which is how the dot and bracket operators work on to get the object reference*
    - `function.bind()` is a useful tool

Object to Primitive conversion

- happens when operations occur between objects (`obj1 + obj2`)
- all objects are true in boolean context
- operators usually cause numeric conversion
- outputs usually cause string conversion
- 3 types of conversions (called 'hints')
    - string
    - number
    - default - when not sure what to do
- *no boolean hint because all objects are default true*
- JS tries all 3 object methods through `obj[Symbol.toPrimitive](hint)`
    - this can be self defined within the object as well
    - hint = string: tries `obj.toString()` before `obj.valueOf()`
    - hint = number/default: tries `obj.valueOf()` before `obj.toString()`
- primitive conversion methods do not *necessarily* return the "hinted" primitive, but they *must* return some sort of primitive
- ```
  const user = {
    name: 'jhon',
    salary: 1000,

    [Symbol.toPrimitive](hint) {
      return hint === 'string' ? this.name : this.salary;
    },
    toString() {
      return this[Symbol.toPrimitive]('string');
    },
    valueOf() {
      return this[Symbol.toPrimitive]('number');
    },
  };
  ```

toString / valueOf

- `toString()` returns a string `[object Object]` unless overwritten
- `valueOf()` returns the object itself unless overwritten
- to fully convert, we should implement these methods inside the object and JS will use these during the `toPrimitive` operation
- `toString()` will handle all conversions if other methods are absent

Constructor and `new`

- Constructor function
  ```

    - named capital letter
    - only executed with `new` keyword
- When `new` is called on constructor functions
    1. new empty object is created
    2. constructor function is executed and `this` will reference the new object
    3. returns `this`
- technically any function can be ran with `new`

Data Types

Primitive Methods

- recall 7 types: string, number, null, undefined, NaN, bigInt, bool
- recall: can store functions inside objects
- objects are 'heavier' than primitives but has many useful default methods
- Primitives are *wrapped* inside an "object wrapper" when it is called that provides extra functionality, but it is destroyed right after it returns the requested method
    - the wrapper works differently per primitive type
    - goal: provide methods but still lightweight
- `null` and `undefined` has wrapper objects hence no methods

Numbers

- stored as *64 bit double precision floating point numbers*
- `bigInt` used to represent numbers larger than 2^53
- scientific notation works in JS: `4e9` for 4 billion
- hex: `0x`
- binary: `0b`
- octal: `0o`
- `toString(base)`: converts a number to a base `(2 <= base <= 36)`
- calling number functions:
    - `123..toString(2)`: first dot is decimal so 2 is required
    - `(123).toString(2)`
- floating point loses precision since it is stored as binary: may see potential decimal errors
    - `number.toFixed(2)`: fix the decimal to remove the precision error
- just like in double precision floating, +0 and -0 exists in JS, but operators treat it as equal
- checking numbers:
    - `isNaN(i)`: the comparison of `NaN === NaN` actually returns false because each `NaN` is unique
    - `isFinite(value)`: checks if it is a *regular* number
        - empty / space-only strings are treated as `0` in all numeric methods
- number conversion: `parseInt()` and `parseFloat()`
    - using `+` or `Number()` is *strict*, if it is not exactly a number, it fails
    - `parseInt` and `parseFloat` read a number from string until they can't
        - if error, the gathered number is returned
        - returns `nan` when *no* digits could be read

    - `parseInt` takes a 2nd param of radix to change numeric system

Strings

- single quotes and double quotes are essential the same
- backticks:
    - allows multiple lines of string (spaces are kept as well)
    - **tagged templates**
        - allows a specified template function before first backtick
        - the function is called automatically, recieves the string and then processes it
            - *rarely used*
            - func`string`
- escaping special characters:
    - add `\` before
    - use backticks/different quotes
    - special characters do take up length of string!
- `str.length` is a property, not a function (O(1))
- `str[i]` is the same as `str.charAt(i)`; modern approach is all
    - `[]` returns undefined if out of bounds
    - `charAt` returns empty string if out of bounds
- character iteration `for (let char of "asdf")`
    - *object iteration is `let prop in obj`*!!!!
- **strings are immutable**
- searching for substring (O(n))
    - `str.indexOf(substr, start_pos)` returns first index found
- checking if substring exists: `str.includes(substr, pos)` returns boolean
    - starts with substring: `str.startsWith(substr)`
    - ends with substring: `str.endsWith(substr)`
- `str.slice(start, end?)`: returns part of the string between start to (but not including) end
    - does not modify the string
    - can use negative values to count from the end (starts at position -1 at the back)
- `str.substr(start, numberOfChars)` is legacy, but it returns u the substring
- `str.substring(start, end?)`: almost same as slice
    - start > end is allowed
    - negatives are not
- comparing strings
    - lowercase > uppercase always (ascii is like this too)
    - stored in UTF-16
- internationalization string compare `str.localeCompare(str2)`
    - str < str2: return negative
    - str > str2: return positive
    - str == str2: return 0

Arrays

- ordered collections
- common usages: stack/queue (known as dequeue (double ended queue))

    - queue: `push` and `shift`
    - stack: `push` and `pop`
- `unshift()`: add to start of array
- internals:
    - arrays extend object methods (they use accessors like objects as well `[]`)
    - copied by reference like object
    - to make arrays fast, JS engine stores it in a contiguous(right next to each other) area
    - JS provides optimizations to this "object" to make it fast, but there are ways to "turn it off" or misuse array intrepretations
        - `arr.prop` -> turns into real object
- performance:
    - `pop`/`push` are fast
    - `shift`/`unshift` are slow
        - needs to re-number all the other elements
        - update length
- `for(let key of arr)` loop through array/string
- `for(let key in obj)` loop through object
- `for (let i=0; i<arr.length; i++)` – works fastest, old-browser-compatible.
- `for (let item of arr)` – the modern syntax for items only,
- `for (let i in arr)` – never use.
- `arr.length`: returns the length
    - can change array length through this property
    - empty an array `arr.length = 0`
- `s = []` is the same as `s = new Array();`
    - `new Array(length)`: sets the array as undefined
- `toString()`: comma separated string
    - do not have their own `toPrimitive` or `valueOf()` conversion
- `array.concat(args)`: returns a new array with all args inside (does not modify the original)
    - way slower than `push()`, but since it returns a new array instead of modifying, it's better for state management (e.g. reducer for react)
- `array.sort()` (optimized quick sort - divide and conquer: splitting array into "more than" and "less than" until length of 1)
    - by default, the elements are sorted by strings
    - to do otherwise, pass in a comparator function
- `Array.from(input)`: converts `input` into an array
- Other array methods: https://javascript.info/array-methods

Iterables

- objects that can be used in `for..of` are iterables
- `Symbol.iterator` is called automatically by `for...of` but can also self-define

Map

- keyed data items
- `map.delete(key)`: removes key
- `map.set(key,val)`: creates a mapping

    - since `map.set` returns the map, you can chain it like `map.set().set()...`
- `map.get(key)`: gets the mapped value
- *can use `map[key]` but should not because it is treating the map as regular JS object, which has limitations*
- can use an object as key! (cannot in normal JS object as it will use `toString()` to convert into `[object Object]` before setting it as key)
- key comparison:
    - uses the algorithm `SameValueZero`, which is similar to `===` except `NaN === NaN` (means `NaN` can be a key)
- uses `for..of` or `forEach` to iterate
    - insertion order will be output order
- can change JS object to Map through `new Map(Object.entries(obj))`
    - will map key => value

Set

- unique values
- `set.add(val)`: adds value
- `set.delete(val)`: delete
- iterate using `for..of` or `forEach`
    - 2 params are the same value for `forEach` to match compatibility with `map`
    -
```
// the same with forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Destructuring Assignment

- *destorying* the structore and split into variables
- can throw unwanted elements in array using additional comma
- works with any *iterable* (strings)
- `...`: can use this to get *the rest* of the elements
-
```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar

// Note that type of `rest` is Array.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```
- object destructoring is more common
- order doesn't matter (will match keys)

- ```
  let {a,b,c} = this.state;
  ```

- can assign by destructoring as well (variable name = key name)
- `...`: can use this to get *the rest* of the props (same syntax as array - will create object)

Date Object

- `new Date()`: creates Date object for *current* time
- `new Date(num)`: num is the timestamp since Jan 1, 1970
  - num = 0: Jan 1, 1970
- `new Date(string)`: runs `Date.parse` to parse the string and return time
- `Date.now()`: grabs current time
- JS gets the time from proxy, UTC offset of the client's local env

JSON

- `JSON.stringify()`: the resulting `json` string is called *JSON-encoded* or *serialized*
  - only double quotes - need to work for all languages through API
  - object prop names are double quotes as well
  - works for Primitives + object + array
- since it needs to work for all languages
  - will skil function properties, symbolic properties, and properties that are *undefined*
- *there cannot be circular references*
- `JSON.parse()`: decode the json string
  - `new` not allowed
  - allowed custom parse function as 2nd param incase it's a special object (like `Date`)

Rest parameters

- `...lastVar`: "gather the remaining params into an array"
- rest parameter *must* be at the end

- ```
  function sumAll(...args) { // args is the name for the array
    let sum = 0;

    for (let arg of args) sum += arg;

    return sum;
  }

  alert( sumAll(1) ); // 1
  alert( sumAll(1, 2) ); // 3
  alert( sumAll(1, 2, 3) ); // 6
  ```

Spread

- opposite of `Rest`: trying to define inputs to a function, not the function's paramters
- breaks array into separate variables
- will break any iterable (has internal interator to gather elements)
    - e.g. `[..."Hello"] -> [H,e,l,l,o]`
    - only works with iterables
        - `Array.from()` works with array-like and iterables

Argument variable

- deprecated: used to capture inputs if there are no input params (just log `arguments` in a function without params)
- arrow functions have no `arguments` variable
    - similar to `this`, will reference outer "normal" function

Variable Scope

- if a variable is declared inside a code block (`{...}`) it is only visible inside that block
- `for(let i =...)`: `i` is declared inside the block

Higher order functions (closures!)

-
```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter(); // closure happens here

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

- Explanation: Lexical Environment!
    - 1. variables
        - every running function (code block) and the script as a whole have an associated object called the *Lexical Environment* which consists of
            - Environment record - stores all **local variables and properties**
            - reference to outer *lexical environment*
        - global lexical environment has no *outer* scope so the `outer` is null
        - the LE is populated as the code gets interpreted, starting with declared variables as `uninitialized` (but will not allow usage until it runs the `let` statement)
    - 2. function declaration (not expression!)
        - same as variables, but becomes instantly usable from the start
    - 3. inner & outer LE

        - when a function is *ran*, a LE is created for that function to store the properties and the `outer` pointer will point to the scope that the function is called on
        - when a variable is required, inner LE is searched all the way back up to global LE (if `strict mode` is on, will return error. otherwise it's created)
        - every function call creates a new LE, but *the reference to the outer LE is only created once*
        - in this case, the LE is created only at the line `return count++`
- All functions are naturally closures in JS except `new Function`
- if a new reference is made to the same function, a **new LE reference** is made so all the properties are reset for the new reference
- *Definition: function that remembers its outer variables, and can access them*
- *Another definition: functions that preserve data*

Double function calls

- ```
  function sum(a) {

    return function(b) {
      return a + b; // takes "a" from the outer lexical environment
    };

  }

  alert( sum(1)(2) ); // 3
  alert( sum(5)(-1) ); // 4
  ```

Global Object

- built into language/environment
- browser: `window`
- global variables can only be defined with `var` (not recommended)
    - should write it as `window.importantVar = "asdf"`
        - can directly access as `importantVar` without `window`
- can be used to test browser compatibility
    - `if (!window.Promise) {console.log('browser old!)}`

Function Objects

- all functions are objects in JS: can access object methods
    - e.g. `function.name` returns function name
    - `function.length` returns # of params (rest params don't count)
- named function expression: `let x = function func() {}`
    - externally cannot access `func()`
    - internally allowed `func()` (e.g. recursion)
    - good for when `x` is reassigned to something else, can still access `func`
    - does not work for pure function declaration

new functions
  ```

- will convert the passed in "string" and convert into functional code
- has no outer lexical environment as it will use the global LE (cannot access outside variables)

Cache decorators

- used for "slow" functions that doesn't change - can cache the result using a wrapper object and a `Map()`

setTimeout and setInterval

- cannot pass a function execution into setTimeout, it expects a reference (the string)
- `setTimeout`: returns a timerId, can cancel using `clearTimeout(timerId)`
- `setInterval`: runs indefinitely by the interval passed in
  - cancel using `clearInterval(timerid)`
  - less accurate than *nested setTimeout* because timer starts when the function starts
  - nested setTimeout: starts the timer after function is finished since it's synchronous
  - ```
    let i = 1;
    setInterval(incI(i++), 2000); // timer goes right when incI() runs

    let id = setTimeout(function incI(i) {
      func(i); // function that increases i
      setTimeout(incI(i), 2000); // runs after func() is done - synchronous
    }, 2000);
    ```
- internal references are created when functions are passed into timer functions so even when no reference exist, it remains in memory (they also reference outer lexical envs as well)
  - good to clear the timer function when done, so garbage collector can take it away

Function Binding

- recall: passing function with `this` referencing outer object to another value will lose the scoping
  - same with `setTimeout(object.function, 200)` as it separated the function and passed it into timeout
    - `setTimeout` sets the `this` reference to global `window`
    - solution 1: to fix for `setTimeout`, run a wrapper function `setTimout(() => user.sayHi(), 100); // sayHi uses this reference`
    - solution 2: `bind(object)`
      - creates a bounded variant of the function call to the `object` passed in
      - binding only happens once per object

Arrow functions

- has no `this` - access outer lexical environment
- `forEach` sets `this` to undefined by default for `function(){}`, but since `() => {}` has no `this`, it is uneffected (will reference outer lexical env)
- not having `this` also means cannot create new with it - cannot be constructors
    ```

- calling `bind` will not work on arrow functions
- has no `arguments` variable
- has no `super`

Prototype Inheritance

- if reading a property from `object` and **it's missing**, JS takes it from the prototype (prototype inheritance)
- set an object's prototype to another to use their functions
  - `object1.__proto__ = object2`
- can set directly within object: `object = {prop: true, __proto__: otherObject}`
- can chain any number of objects (no cycle) and will inherit all objects chained
- the value of `__proto__` can only be object or `null`
- prototypes do not effect `this` - `this` only binds to the object before the dot operator

Setters and Getters

- inside the object, you can directly set `set` or `get` functions
- ```
  let user = {
    name : "asdf",
    set fullName(value) {
      this.name = value;
    }
    get fullName() {
      return this.name;
    }
  }
  ```

Try...catch

- runs the `try` code block
- if no errors, `catch` is ignored
- if error occurs, the control flows to `catch` and the error object is available
  - will ignore the rest of `try`
- only works for runtime errors
  - parse time: when engine is reading the code
  - runtime: code is syntactically correct, but something else went wrong
- only works synchronously (`setTimeout` errors cannot be tested)
  - must put `try..catch` inside the `setTimeout` so it runs together
- errors are split into 3 properties
  - name: error name
  - message: error details
  - stack: where it happened
  - logging errors will show all 3 in a string
- can throw new errors `return new Error(message)`
  ```

Callbacks

- async functions will run by itself (e.g. script loading, `setTimeout`, module loading)
- callbacks are used to ensure the previous async method finishes running
  - `documentElement.onload` and `documentElement.onerror` are callbacks for when the element is rendered
- should not use callbacks on deep-nested async chains (use *promises*)

Promises

- `let promise = new Promise((resolve, reject) => {})`
  - the function passed into the `Promise` is called *executor* and it is executed automatically
  - `resolve` and `reject` are callbacks provided by JS
    - run these appropriately with respective to your executor
- initial state of promise = **pending**
- if it is resolve/rejected, it is in a **settled** state
- state of "resolved" = **fulfilled**
  - result: value passed into `resolve(value)`
- state of "rejected" = **error**
  - result: error passed into `reject(error)`
    - recommended to use the `new Error()` object
- Comsumers (functions that call the promise)
  - access the result using `promise.then(res, rej)` where res, rej are receiving functions
- if only interested in errors: `promise.catch(f)` (same as `.then(null, f)`)
- `.finally()` runs regardless of the settled state
  - will pass the resolve/reject onto the next `.then` call
  - ```
    new Promise((res, rej) => {
      res("done!");
    })
    .finally() {
      // will always run here - pass state to next .then
    }
    .then(res, rej) => {
      // value is done
    }
    ```
- can call many handlers on the same promise

Promise Chaining

- flow:
  - initial promise resolves
  - first `.then` called
  - the returned value of the `.then` is passed onto the next `.then`
- the returned value of `.then` handlers always become the next result of the promise
- can return a `new Promise()` inside a `.then` call - same effect as a normal promise
    ```

- `.then` can also return a **thenable** object, which acts the same way as a promise inside classes

Promise - Error handling

- promises have an invisible `try..catch` in the sense that if error is thrown/rejected, the `catch` will treat it as exception
  - not only the executor, but in every `.then` statement, the `.catch` after will catch it
- can chain `.then` after a `.catch` *if* the error is handled properly (returning inside a `.catch` will be a resolved)
- can also throw another error if it cannot handle (will skip all the `.then` calls until it hits the next `.catch`)
- if the promise handler has **no error handlers**, it will generate a global error
  - can be read using

```
window.addEventListener('unhandledrejection', (ev) => {
  console.log(ev.promise); // promise that generated the error
  console.log(ev.reason); // most likely unhandled error
});
```

Promise API

- wait for `n` promises to resolve: `let promise = Promise.all([...promises])`
  - if one fails, it will go to error
  - common trick: use array.map to return a list of promises
  - *for network request, can transform into object through `response.json()`*
- wait for `n` promises to settle: `allSettled`

```
Promise.allSettled(/* iterable */)
.then(res => {
  // res.status: fulfilled or rejected
  // res.value: the value or error
})
```

- wait for first promise to settle and returns the status, value: `Promise.race([...])`

Promisification (promisfy)

- process of turning a regular function into a promise
- creating a wrapper and returning a new promise

Async- background

- Event loop checks if call stack is empty or not
  - if empty: looks for callbacks in the message queue waiting to be executed
- message queue holds all "async" function executions
- the event loop will put it on top of the call stack and thus execute it after

Macrotask and microtask

- macrotasks: setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI rendering
- microtasks: process.nextTick, Promises, Object.observe, MutationObserver
- after **every** macrotask, all microtasks should be ran before the next execution of anything else

Promises - background

- ECMA specifies an internal queue `PromiseJobs` or *microtask queue*
- all `.then`/`.catch`/`.finally` is placed inside the queue
- execution of the task inside the queue is initiated only when nothing else in the current node is running
    - ```
      let promise = Promise.resolve();
      promise.then(() => alert("promise done!"));
      alert("code finished"); // this alert shows first
      ```

Async/wait

- easy way to work with promises
- keyword `async` infront of function declaration
    - function now returns a promise!
- keyword `await`: makes JS wait until promise has settled
    - can only use **inside async** functions
    - allows other things to run while this function waits
- cannot use async/await in global scope
- to do error checking inside `async function`, run `try{}catch{}` inside
- do not need to use `.then` for `async` functions since the `wait` should do the job for you, and use `try..catch` inside

---

Extras

Console.log()

- not the same as `alert()`
    - it does not "expect" a certain type
    - will print it to the console (object -> prints object tree)
    - `alert()` expects a string

Garbage Collection

- *reachability*: any values that can be accessed (in any way) will be stored in memeory (e.g. references, local variables, etc)
    - e.g. if an (and only) object reference is overwritten, then garbage collector will clear the old object since it's unreachable
- the outgoing links do *not* matter, if the object cannot be accessed it's garbage
      ```

- the *mark-and-sweep* algorithm
    - garbage collector (gc) will mark all roots
    - traverse through the graph and marks all references
        - repeats until all reachable references are visited
    - all objects that are *unmarked* will be freed
- additional optimizations to MS algo
    - generational collection: split objects into "new" and "old"
        - the old ones are checked less often
    - incremental collection:
        - break apart massive objects so that engine isn't delayed for a long time but instead `n * short periods`
    - idle-time collection: only run the MS algorithm when CPU is idle so that user isn't effected

Cache:

- files will be cached when attached as a source using the `script` tag
- allows a response to be re-used without calling it again
    - can be controlled by passing a `max-age` header
- any other pages that references the same script will not need to download again
- **reduces traffic and faster renders**

ECMA2020