Numeric Conversion

* Implicitly occurs when division is applied to non-numbers

|  |  |
| --- | --- |
| Value | Becomes… |
| Undefined | NaN |
| null | 0 |
| true | 1 |
| False | 0 |
| String | Whitespaces from start and end are removed. If the remaining string is empty, the result is 0. Otherwise the number is “read” from the string. An error gives NaN. |

* “0” and space-only strings are true when converted to Booleans
* NaN returns false for all numeric conversions

String Concatenation

* If any of the operands in an addition operation is a string, the other one is converted to a string as well
* Any numeric operator that is not + always converts its operands to numbers

alert()

* Function that takes a single argument – a string that will then populate a popup alert

prompt()

* Function that takes two arguments – a title and an optional default value
* prompt() displays a popup modal with the title

Variable Naming

* The name can only have alphanumeric characters, or the symbols $ and \_
* The first character in a variable name cannot be a digit
* Common practice to use name constants with all-caps names, using underscores to separate words in the variable name
* Variable names starting with an underscore are considered internal by convention and shouldn’t be accessed by outside entities
  + It isn’t enforced by JavaScript, but rather is a convention
  + These are “protected” variables

Strict Mode

* Doesn’t allow variable declarations without the use of one of the scope-enforcing keywords

BigInt

* Represents integers of an arbitrary length
* Created by appending n to the end of an integer or by calling BigInt(), which creates a BigInt from the argument passed to it
* BigInts and regular numbers can’t be mixed in mathematical operations, but they can be used together in comparisons
* A mathematical operation on two BigInts returns a BigInt
* If a BigInt is converted to a number (a 64-bit data type), any extra bits will be cut off
* BigInt 0n is falsey, and all other BigInt values are truthy

Undefined

* A value is not assigned

Comparisons

* When comparing values of different types, JavaScript converts the values to numbers
* Strict equality operator
  + ===
  + Checks the equality of the two expressions without type conversion
* Mathematical comparisons convert null to a number, treating it as a zero
  + But the equality comparison for undefined and null is such that null == undefined and the two values don’t equal anything else

Conditional/Question Mark/Ternary Operator

* condition ? value1 : value2
  + the condition is evaluated, if it’s truthy, value1 is returned
  + otherwise, value2 is returned

|| and &&

* evaluate operands from left to right

??

* nullish coalescing operator
* a ?? b
  + if a is defined (i.e. not null or undefined), the result is a
  + if a isn’t defined (i.e. is null or defined), the result is b, regardless of b’s values
* common use case is to provide a default value for a potentially undefined variable (if a function is called without a value for a parameter, the parameter value will be undefined)
* || returns the first truthy value, while ?? returns the first defined value

Function Expressions

* Anonymous function
* Function expressions are executed when the execution reaches them and are only usable from that moment onward
* Function declarations can be called earlier than they’re defined

Arrow Functions

let func = (arg1, arg2, …., argN) => expression

let func = (arg1, arg2, …, argN) => {

}

Transpiler

* A piece of software that can parse modern code and rewrite it using older syntax constructs
* E.g. Babel

Polyfill

* A script that updates/adds new functions so that a method that isn’t yet supported by a particular JavaScript engine, but is in JavaScript specifications, can be used
* E.g. core js

Accessing Object Properties

* When using square brackets, the value of the key can be computed at runtime, but this is not the case when using dot notation

Computed Properties

* Used within objects
* Tells JS to take the value for the property from the variable with the name passed in the square brackets
* E.g.

let bag = {

[fruit] : 5

}

Property Shorthand

* When a property has the same name as a variable, the property/variable name only needs to be written once in the object literal during instantiation

in Operator

* Returns true if the specified property exists in the specified object
* E.g. “myProperty” in myObject

for-in Loop

* for(key in object) {}
* allows us to iterate through all keys in an object

Passing by Value vs. Passing by Reference

* objects are passed by reference, while primitives are passed by value

Method

* a function that is a property of an object
* method shorthand in an object literal

objectLiteral {

functionName() {}

}

this

* refers to the enclosing object
* can be used in any function, even if it’s not a method of an object
* value is evaluated at runtime
* this will be the global object (i.e. the window in a browser) if not called from within an object
* arrow functions do not have their own this (or super) – this will rather refer to the outer function/context
* in an object, the value of this will be the object before the period when calling the method through dot notation

Optional Chaining

* ?.
* Stops the evaluation if the value before ?. is undefined or null and returns undefined

Symbols

* Symbols in object literals are surrounded by square brackets
* Object.keys() and for-in loops do not include symbols
* Used for hidden object properties
* Created with a call to Symbol()

Object to Primitive Conversions

* hints determine the type of conversion

|  |  |
| --- | --- |
| Hint | For |
| string | Object-to-string conversion, when an operation on an object expects a string. |
| number | Object-to-number conversion, when a mathematical operation is being performed. |
| default | Used when the operator isn’t sure what type to expect. All built-in objects except for the Date object implement default hints as number hints. |

* To perform the conversin, JavaScript looks for the following methods (in order) and calls the first one that exists
  + obj[Symbol.toPrimitive](hint)
    - The method with the symbolic key Symbol.toPrimitive
  + If the hint is string, obj.toString() or obj.valueOf() if obj.toString() doesn’t exist
  + if the hint is number or default, obj.valueOf() or obj.toString() if obj.valueOf() doesn’t exist

toString()

* If the toString() method is not overridden in a custom object, it will return [object type], where type is the object’s type (e.g. [object Object])

valueOf()

* If the valueOf() method is not overridden in an object, it will return the object itself

Explicit Conversions

* Can use String(), Number(), or Boolean() functions (without the new keyword) to explicitly convert a value to the respective primitive type
* The double not operator, !!, can also be used to convert a value to its corresponding Boolean primitive

Backticks

* Allow expressions to be embedded in strings by wrapping the expressions in ${}
* Allow a string to span multiple lines

Strings

* Are immutable
* Are iterable

Deque

* A data structure that allows the addition/removal of elements both to/from its beginning/end

for-of Loop

* for(let a of myArray) {}
* iterates over the values of an iterable object

Iterator

* an object with the method next() that must return a value in the form {done: Boolean, value: any}, where done being true means that the iteration is finished, otherwise value will hold the subsequent value

Iterable

* an object with the Symbol.iterator() method that returns an iterator

Array-Like Objects

* objects that have indexes and length

NaN

* doesn’t equal anything, including itself
* unless being used with Object.is(), in which case Object.is(NaN, NaN), will return true

Accessing Characters in Strings

* if there is no character found, accessing a character at a specific index through square brackets (e.g. str[3]) will return undefined, while accessing a character at a specific index through the charAt() method will return an empty string

Substring Methods

|  |  |
| --- | --- |
| Method | Returns the substring |
| slice(startIndex [, endIndex] | From startIndex to endIndex, but not including endIndex |
| substring(startIndex [, endIndex] | Between startIndex and endIndex |
| substr(startIndex [, length]) | From startIndex with length characters |

Array Methods/Properties

|  |  |
| --- | --- |
| Method/Property | Explanation |
| shift() | Removes the first element of an array and returns it |
| unshift() | Adds the specified element(s) to the beginning of the array |
| length | Returns the greatest index in the array incremented by one |
| Array() | Constructor that creates an array with the specified length (if given a single numeric argument) or an array with the specified elements (if given multiple arguments or a single, non-numeric argument) |
| forEach() | Runs a function on each element of an array. Takes up to three parameters (item, index, and array, in that order). Mutates the existing array |
| filter() | Returns an array of all array elements that match the filter. Takes up to three parameters (item, index, and array, in that order) |
| map() | Runs a function on each element of an array and returns an array of results. Takes up to three parameters (item, index, and array, in that order) |
| reduce() | Runs a function on each element of an array, passing in the return value from the calculation on the preceding element to the next element. Takes up to four parameters (previousValue, currentValue, currentIndex, and array, in that order). |

Map

* different from objects in that keys can be any data type (including objects) and that it has additional methods and the size property
* if square bracket notation is used with a map, keys will only be able to be strings or symbols
* entries()
  + method that returns an iterable for entries as a [key, value] pair
  + used in for-of loops with maps by default
* get()
  + returns the value of the given key or undefined if the key doesn’t exist in the map

Set

* a set of values where each value may only occur once

WeakMap

* keys must be objects, not primitives
* doesn’t prevent garbage collection of objects that are serving as its keys
  + so if all references to an object serving as the key of a WeakMap are removed, the object will be garbage collected and removed from the WeakMap
* does not support iteration

Weak Set

* a set of values where each value must be an object and can only occur once
* does not support iteration
* doesn’t prevent garbage collection of objects that are serving as its values
  + so if all references to an object serving as a value of a WeakSet are removed, the object will be garbage collected and removed from the WeakSet

Rest Parameters

* indicated by ellipses (…)
* can be included in a function definition and followed by the name of an array that will contain them
* if rest parameters are used for a function, they must be the last parameters declared
* used to create functions that accept any number of arguments

Spread Syntax

* indicated by ellipses (…)
* expands an iterable object into a list of arguments
* used to pass an array to functions that normally require a list of arguments

JSON

* different from JavaScript object literals in that strings and object property names are always surrounded by double quotes
* JSON.stringify()
  + converts the object passed as an argument to a JSON string (a.k.a a JSON-encoded, serialized, stringified, or marshalled object)
* JSON.parse()
  + converts the JSON passed as an argument to an object
* JSON can support object literals, arrays, and the string, number, Boolean, and null primitives, but it doesn’t support functions as properties, symbols, and undefined properties
* JSON also doesn’t support circular references between objects

Lexical Environment

* An internal hidden object that consists of two parts
  + Environment Record – an object that stores all local variables as its properties
  + A reference to the outer lexical environment
* Variables declared with let can’t be referenced before they have been declared
* Functions that are declared (i.e. not assigned to variables) can be invoked before they are declared
* If a variable isn’t found anywhere in the lexical environment in strict mode, there’s an error
  + If not in strict mode, a new global variable with the variable name will be created
* All functions remember the lexical environment in which they’re declared

Closure

* A function that remembers its outer variables and can access them

var

* Variables declared with var are either function-scoped or globally-scoped
* Variables originally declared with var can be redeclared with var
* Variables declared with var are processed when the function starts execution (if they are function-scoped) or when the script starts execution (if they are globally-scoped)
  + This is variable hoisting
  + Therefore, var variables can therefore be referred to before they’re defined
  + But they won’t have a value until the code line containing the initial assignment

Global Object

* window is the global object in a browser
* global is the global object in Node.js
* globalThis is a standard property representing the global object that’s available in all JavaScript environments
* functions and variables declared with var in a browser become properties of the global object

Functions

* because functions are objects in JavaScript, they can have custom properties assigned to them

Object Properties

* have four descriptors
  + value, which holds the value associated with the property
  + writable, which holds a true or false value indicating if the value of the property can be changed
  + enumerable, which holds a true or false indicating if the property is listed in loops and if it’s returned from a call to Object.keys()
  + configurable, which holds a true or false indicating if the property can be deleted and its flags can be modified
    - a non-configurable property can still have its writable flag changed from true to false

Object.assign()

* method that copies all enumerable own properties from one or more source objects to a target object
* properties in the target object are overwritten by properties in the sources and later sources’ properties overwrite earlier ones

Object.getOwnPropertyDescriptor()

* takes two arguments
  + the object to get information from
  + the name of the property
* returns the flags of a property in a property descriptor object

Object.defineProperty()

* takes three arguments
  + the object to work with
  + the property to set the flags of
  + the property descriptor object to assign to the property
* if a property exists, the method updates the flags of the property
* if the property doesn’t exist, the method creates the property
  + any non-specified flags have false assigned to them
* when not in strict mode, performing an operation that violates a property’s flags won’t thrown an exception, but won’t succeed
  + performing an operation that violates a property’s flags in strict mode will throw an exception

Object.defineProperties()

* method to define flags for multiple properties
* takes two arguments
  + the object to update the properties of
  + an object containing property-property descriptor object pairs

Object.getOwnPropertyDescriptors()

* method to retrieve all property descriptor objects for an object
* a flag-aware way to clone an object (along with Object.defineProperties())

Accessor Properties

* functions that execute when getting and setting a value, but look like regular (i.e. data) properties to external code
* a property can either be an accessor property or a data property, not both
* defined through getter and setter methods
* accessor properties can be used for protected variables
* e.g.

let obj = {

get propName() {

}

set propName(value) {

}

}

Accessor Descriptors

* accessor descriptors have four flags
  + get – the getter function
  + set – the setter function
  + enumerable, which holds a true or false indicating if the property is listed in loops and if it’s returned from a call to Object.keys()
  + configurable, which holds a true or false indicating if the property can be deleted and its flags can be modified

Prototype

* an object referenced by the hidden property [[Prototype]] of another object
* used when a nonexistent object property is read from or a nonexistent method is called
* JavaScript will take the property from the prototype (if it exists) through prototypal inheritance
* can work with the prototype for an object with the \_\_proto\_\_ getter/setter
* all built-in prototypes eventually inherit from a prototype with the Object class as its [[Prototype]]
  + the Object class has a null [[Prototype]]
* Object.create(proto, [descriptors])
  + Creates an empty object with the given proto as its prototype and optional property descriptors
* Object.getPrototypeOf(obj)
  + Returns the prototype of obj
* Object.setPrototypeOf(obj, proto)
  + Assigns proto as the prototype of obj

Function Prototypes

* if a property named prototype on a function is assigned an object obj, the [[Prototype]] of new objects instantiated with that function’s constructor will be the same object obj
  + the default prototype property for a function is an object with a single property named constructor, with a value of the function
  + the constructor property of an object can be used to create new objects using the same constructor

Classes

* declared with the class keyword followed by the name of the class
* under-the-hood, a class is a special kind of function
  + when a class is declared, a function with the name of the class is created
  + this function is populated with the code in the constructor method (named constructor), if it exists
  + methods declared within the class are stored in the prototype of the class
* there are differences between declaring a class and manually declaring functions with prototypes
  + classes need to be instantiated with the new keyword
  + class methods are non-enumerable
  + classes always use strict mode
* classes can be defined as expressions and assigned to variables
  + e.g.

let myClassVar = class {

}

* when extending a class using the extends keyword, JavaScript sets the [[Prototype]] of the child class prototype to be the prototype of the parent class (i.e. Child.prototype.\_\_proto\_\_ is Parent.prototype
* static methods and static properties can be inherited because the function that represents the parent class is the [[Prototype]] of the function that represents the child class
* i.e. inheriting from a class sets up two prototypes – one between the prototype of the constructor functions (for methods and properties) and one between the constructor functions themselves (for static methods and properties)

Constructors in Child Classes

* constructors in child classes must call super() and do so before using this
* this is because a derived constructor (a constructor in a child class) doesn’t set the value of this, but rather relies on the parent constructor to do so

Static Methods

* static methods are those in a class prepended with the static keyword
* they’re attached to the class, rather than the prototype

Static Properties

* those class properties prepended by the static keyword

class ClassName {

static property = value;

}

* equivalent to

ClassName.property = value;

Destructuring Assignment

* a syntax that allows unpacking of values from objects and arrays into different parameters
* default values can be placed after an assignment operator on the left side of the destructuring assignment for the case that the element (in array destructuring) or key (in object destructuring) doesn’t exist
* array destructuring

let [variableToHoldElement0, , …, variableToHoldElementN] = arr

* object destructuring

let {prop : varName = default } = obj

* + in the basic syntax, values are assigned to the variables on the left side of the destructuring assignment based on matching variable and key names
  + this can be overridden by preceding a variable name with a colon and the key whose value it should be given (i.e. what : goesWhere)
  + if using existing variables, the entire expression should be surrounded by parentheses so it isn’t interpreted as a code block
  + if destructuring a nested object, any inner key variables will be assigned corresponding values, but corresponding outer key variables will not be declared
* destructuring assignment can also be used for the parameters of a function
  + in fact, a default value of an empty object can be assigned through destructuring assignment in the parameter parentheses to allow a function to be called without arguments
* rest parameters can also be used in destructuring assignment

Private and Protected Fields

* protected fields are inherited
* private fields are those prepended with a #
  + they cannot be accessed from outside of the class that declares them (even if attempting to access from inheriting classes)
  + this is enforced by JavaScript and not just a convention

Mixin

* a class containing methods that can be used by other classes without a need to inherit from it
* can use a mixin by assigning it to the prototype of a class with a call to Object.assign()
  + the method takes two parameters (the class prototype and the mixin)

Callback

* a function that executes after an asynchronous function has finished
* a function that’s passed as a value to another function and will only be executed when an event happens
* able to be worked with in JavaScript because functions are first-class in JavaScript, meaning that they can be assigned to variables and passed to other functions
* Error-first callbacks are a convention for callback functions where the first parameter of the callback function is reserved for an error (if it occurs) and the following arguments are for successful results

Promise

* A proxy for a value that will eventually become available
* Once a promise is called, it will start in a pending state, meaning that the calling function continues execution
* The promise eventually moves to a resolved or rejected state, invoking either the callback function(s) passed to then or catch, respectively
* the promise constructor takes an executor function with two parameters – resolve and reject callbacks (in that order)
  + e.g. new Promise((resolve, reject) => {})
* The promise serves as a link between the executor and consuming functions
* the resolve and reject callbacks are provided by JavaScript
* the executor runs automatically and attempts to perform a job
* when the job is finished, the executor calls the resolve callback if it was successful or the reject callback if there was an error (whether explicit rejections or other errors, so long as those other errors occurred while the executor was running)
* any errors that occur in an asynchronous executor without a reject call will not cause the promise to be rejected
* the promise constructor returns a new promise object that has state and result properties
  + state initially has a pending value that changes to fulfilled when the resolve callback is invoked or rejected when the reject callback is resolved
  + result is initially undefined, then changes to the value returned from the promise when the resolve callback is invoked or the error when the reject callback is invoked

Diagram

Description automatically generated

* a promise that is resolved or rejected is referred to as settled

Consuming Promises

* consuming functions for promises can be registered using the then, catch, and finally methods
* then methods take two arguments
  + a function to invoke when the promise is resolved, which takes a single parameter (the result of the promise; oftentimes named result)
  + a function to invoke when the promise is rejected, which takes a single parameter (the error that occurred; oftentimes named error)
  + the second argument does not need to be provided
* catch methods take a single argument
  + a function to invoke when the promise is rejected, which takes a single parameter (the error that occurred)
* finally methods take a single argument
  + a function to invoke when the promise is settled, whether it’s been resolved or rejected
  + this function takes no arguments

Promise Chaining

* then methods can be chained because a call to the then method returns a promise itself
* when a then method returns a value, it becomes the value for the next promise (e.g. then methods pass their results/errors to the next then/catch)
* e.g.

promise.then(function(result) {

}).then(function(result) {

});

* multiple then methods can also be added to a single promise, but this isn’t chaining
  + in this case, the then methods don’t pass results to each other, but rather process it independently
* when anything in a chain of promises fails and raises an error or rejects the promise, execution goes to the nearest catch() function further down the chain

\*\* As a good practice, an asynchronous action should always return a promise. \*\*

Promise API

* the Promise class has six static methods
* Promise.all()
  + Takes a single argument (an iterable of promises)
  + Returns a new promise that resolves when all promises in the array parameter are resolved
  + The array of the results corresponding to the iterable of promises passed as the argument becomes the result of the Promise.all() call
  + If any of the promises passed to the method’s parameter are rejected, the promise returned by the method rejects with the corresponding error
  + If any of the objects in the iterable passed to Promise.all() is not a promise, it’s passed to the returned array as is
* Promise.allSettled()
  + Takes a single argument (an iterable of promises)
  + Waits for all promises to settle before returning an array of the results
  + If an individual promise was resolved, the corresponding element in the returned array will be the following object {status: “fulfilled”, value: result}
  + If an individual promise was rejected, the corresponding element in the returned array will be the following object {status: “rejected”, reason: error}
* Promise.race()
  + Takes a single argument (an iterable of promises)
  + Returns the result or error of the first settled promise
* Promise.any()
  + Takes a single argument (an iterable of promises)
  + Returns the result of the first fulfilled promise
  + If all promises are rejected, the returned promise is rejected with AggregateError (an error that stores all promise errors in its errors property)
* Promise.resolve()
  + Takes a single argument (the value to return as the result of the promise)
  + Creates a resolved promise with the given argument as its result
  + Used when a function is expected to return a promise
  + Equivalent to

new Promise(resolve => resolve(value))

* Promise.reject()
  + Takes a single argument (the value to return as the error from the promise)
  + Creates a rejected promise with the given argument as its error
  + Equivalent to

new Promise((resolve, reject) => reject(error))

Arguments

* The values passed to a function when it’s called

Microtask Queue

* An internal queue specified by ECMAScript (also known as PromiseJobs)
* The queue is FIFO and execution of a task is started only when nothing else is running
* When a promise is settled, the promise handlers are placed in the microtask queue because promise handlers are always asynchronous actions
* So any code in the current context that hasn’t executed will still run before the promise handlers of a settled promise (even if that promise was settled immediately)
* The unhandledrejection event is fired when a promise error is not handled at the end of the microtask queue
* If we need to guarantee that a piece of code is executed after a promise handler, we can add it to a chained then method

Async/Await

* Built on promises
* Functions that begin with the async keyword (after the function keyword) always return a promise
* Any return statements that don’t explicitly return a promise will have their values wrapped in a resolved promise
* The await keyword makes the JavaScript engine suspend function execution until a promise settles and returns a result
* It can only be used inside async functions
* To implement error handling, an await statement can be wrapped in a try/catch block
* If an exception thrown by an async function isn’t caught, the promise returned by the method will become rejected, so a catch() method can be appended to the invocation of the async function to be invoked in the case that the function results in an error

Module

* A JavaScript file that contains methods that can be imported and used by another JavaScript file
* The export keyword is used to denote variables, methods, and classes that can be used from outside the module in which they’re declared
* The import keyword is used to import variables and methods from another module
* Modules always use strict mode
* A module has its own top-level scope
* Modules are only evaluated the first time they’re imported
  + Therefore, top-level module code should only be used for initialization
  + If something needs to be invoked multiple times, it should be placed in an exported function
  + This also means that exported variables are shared between all importers
* Module scripts in browsers are deferred, meaning that they wait to execute until the HTML has been fully rendered

Exporting from Modules

* Entities can either be exported from a module by
  + prepending their declaration with the export keyword
  + using the export keyword after declaration and surrounding the exported entities with curly braces
    - e.g.

export {x, y}

* Using the as keyword allows exporting as an alias
  + E.g.

export {x as y}

* The default export syntax allows us to denote the default export of a module
  + The default keyword follows the export keyword in the export statement
  + If exporting after declaration, the export statement will use the following format

export {x as default}

* + There can only be one default export per module
  + The default export is not required to have a name
* Exported entities that aren’t the default export are referred to as named exports because they must have names when declared

Importing from Modules

* Everything from a module can be imported as an object using the asterisk
  + When importing everything as an object, the default property of the object holds the default export of the module
  + E.g.

import \* as x from “x.js”;

* Like with exporting, using the as keyword allows importing as an alias
* If importing named exports, the entities to import are listed in curly braces
* If importing a default export, the curly braces can be omitted
* If importing both the default export and named exports from the same module, the import statement will use the following format

import {default as x, y} from “x.js”;

* Imported variables cannot have their values changed, but their properties can still be modified

Dynamic Imports

* import()
  + pass the module to import in parentheses
  + returns a promise that resolves into a module object that contains all exports of the module
  + allows modules to be imported conditionally or at runtime

Proxy Objects

* wrap other objects and intercept operations

let proxy = new Proxy(target, handler)

* + where target is the object to wrap and handler is the proxy configuration
  + the handler has traps (i.e. methods that intercept operations)
* when an operation is performed on a proxy object, the engine checks for a corresponding trap in the handler
  + if the trap exists, its invoked and the operation is executed on the proxy object
  + otherwise, the operation is performed on the target
* without a handler, a proxy object will transparently forward operations to its target
* proxy specification

|  |  |  |
| --- | --- | --- |
| Internal Method | Handler Method | Invoked When… |
| [[Get]] | get | Reading a property |
| [[Set]] | Set | Writing to a property |
| [[HasProperty]] | has | Using the in operator |
| [[Delete]] | deleteProperty | Using the delete oper[[Sator |
| [[Call]] | apply | Function call |
| [[Construct]] | construct | Using the new operator |
| [[GetPrototypeOf]] | getPrototypeOf | Invoking Object.getPrototypeOf() |
| [[SetPrototypeOf]] | setPrototypeOf | Invoking Object.setPrototypeOf() |
| [[IsExtensible]] | isExtensible | Invoking Object.isExtensible() |
| [[PreventExtensions]] | preventExtensions | Invoking Object.preventExtensions() |
| [[DefineOwnProperty]] | defineProperty | Invoking Object.defineProperty() or Object.defineProperties() |
| [[GetOwnProperty]] | getOwnPropertyDescriptor | Invoking Object.getOwnPropertyDescriptor, Object.keys(), Object.values(), Object.entries(), or using a for-in loop |
| [[OwnPropertyKeys]] | ownKeys | Invoking Object.getOwnPropertyNames(), Object.getOwnPropertySymbols(), Object.keys(), Object.values(), Object.entries(), or using a for-in loop |

eval()

* a built-in function that allows the execution of a string of JavaScript code
* the result of an eval() call is the result of the last statement
* in strict mode, eval() has its own lexical environment
* eval() can access outer local variables

Reference Type

* when a method on an object is invoked, two operations occur
  + the dot notation retrieves the property (i.e. method) from the object
    - rather than returning a function, the dot notation retrieves a value of Reference Type
  + the parentheses execute the retrieved method
* Reference Type is a specification type, meaning that it can’t be explicitly used, rather JavaScript uses it internally
* Reference Type values take the form (base, name, strict) where
  + base is the object
  + name is the name of the property
  + strict is true if strict mode is being used
* Reference Types are used to set the correct value of this when a method is called
* Any operation outside of direct method invocation ignores the Reference Type, so, e.g., assigning a method from an object to another variable will lose the value of this
* This problem can be solved by function binding

Event Loop

* While a task from the event loop is executing, rerendering doesn’t occur
* Immediately after every microtask, the engine executes all tasks from the microtask queue (e.g. promise handlers), prior to running any other macrotasks or rerendering

dispatchEvent()

* Takes a single argument (the event to fire)
* Fires the event from the specified element

Custom Events

* The CustomEvent class differs from the Event class in that the second argument in the constructor can take a detail property to hold information to propagate through the event

Cookies

* Small strings of data stored directly in the browser
* Commonly used for authentication
* document.cookie stores cookies from the current website
* writing to document.cookie doesn’t overwrite all cookies, only the cookie(s) provided in the write operation
* if a cookie has the httpOnly attribute, it can’t be accessed from document.cookie
* third-party cookies are those placed by a domain other than the one the user is visiting
  + traditionally used for tracking and ads services
  + because they’re bound to the domain that created them, they can be used to track the same user between different sites if each site uses a resource from the third-party domain

XSRF Attacks

* cross-site request forgery attacks
* forces an end user to execute unwanted operations on a web application in which they’re currently authenticated

Clickjacking Attack

* tricks a user into clicking a webpage element which is invisible (e.g. through a transparent iframe) or disguised as another element
* used to, e.g., download malware, visit malicious pages, or provide sensitive information

Decorator

* a function that takes another function and alters its behavior
* used to enhance a function without modifying the underlying function
* e.g.

decoratorFunction(functionToDecorate)

* or

@decoratorFunction

functionToDecorate

Class Member Decorators

* have three parameters
  + target – the class that the member is a part of
  + name – the name of the member in the class
  + descriptor – the member descriptor
* applied to individual members of a class

Class Decorators

* applied to the entire class
* have one parameter – the constructor function being decorated

Call Forwarding

* passing all arguments and the context provided to one function to another
* used to preserve context in decorators
* call(context, …args)
  + takes two parameters – the context and a list of arguments to pass to the function
  + used to pass iterables as the list of arguments
* apply(context, args)
  + takes two parameters – the context and an array-like object to pass to the function

bind()

* returns a function that, when called has its this set to the argument provided to bind()

Method Borrowing

* taking a method from an object and calling it in the context of another object

Losing Context

* context (i.e. this) is lost when a method is passed separately from the object

Partial Function

* a function that fixes parameters of an existing function

fetch()

* takes two parameters
  + the URL to send a request to
  + an object with optional parameters including method, headers, and body
  + if the second parameter is omitted, a GET request will be made to the URL
* the engine starts the request and returns a promise
* this promise resolves with an object of the Response class
  + can use the status and ok properties (the latter of which returns true if the HTTP status code is between 200 and 299) to check the result of the call
* to get the response body, one of the following methods can be invoked on the response

|  |  |
| --- | --- |
| text() | Reads the response and returns it as text |
| json() | Parses the response as JSON |
| formData() | Returns the response as a FormData object |
| blob() | Returns the response as a Blob |
| arrayBuffer() | Returns the response as an ArrayBuffer |

* if the request body is a string, the Content-Type header is set to text/plain;charset=UTF-8
  + if sending JSON, this should be changed to application/json
* await statements are used both when invoking the fetch method and processing the response body

Cross-Origin Request

* a request sent to another domain, protocol, or part
* safe requests
  + those requests that use safe methods and safe headers
  + safe methods are GET, POST, and HEAD
  + safe headers are Accept, Accept-Language, Content-Language, and Content-Type (if the value of Content-Type is application/x-www-form-urlencoded, multipart-form-data, or text/plain)
  + can be made with a <form> or <script> element without any special methods
* when a safe cross-origin request is made, the browser adds the Origin header to it with the URL of the requestor
  + the server inspects the Origin, and if it agrees to accept the request, adds the Access-Control-Allow-Origin header to the response
    - the header will contain the allowed origin or \*
    - JavaScript will then be able to access the response
  + If the server rejects the request, the JavaScript execution will throw an error

CORS

* cross-origin resource sharing
* When an unsafe cross-origin request is made, the browser makes a preflight request before issuing the request made from JavaScript
  + the preflight request is an OPTIONS request with Access-Control-Request-Method and Access-Control-Request-Headers headers holding the requested method and unsafe request headers, respectively
  + if the server agrees to accept the unsafe request, it will respond with a 200 status and the headers Access-Control-Allow-Methods (holding a list of allowed methods), Access-Control-Allow-Headers (holding a list of allowed headers), and Access-Control-Max-Age (holding the number of seconds to cache permissions) headers
  + then the request made from JavaScript will be sent, and the process described for safe requests will be followed

XMLHttpRequest

* used in modern web development for
  + historical reasons to support existing scripts
  + support for old browsers without the use of polyfills
  + functionality that fetch can’t accomplishment

Long Polling

* an implementation of persistent connection with a server where
  + a request is sent to the server
  + the server doesn’t close the connection until it has a message to send
  + when a message appears, the server responds to the request with it
  + the browser makes a new request immediately
* best used when messages are rare

WebSocket

* a protocol that provides a way to exchange data between a browser and a server through a persistent connection
* the data can be passed in both directions without breaking the connection and needing additional HTTP requests
* can create a WebSocket connection by instantiating the WebSocket class
  + the constructor takes a single parameter – the URL to open a connection with
  + in the URL, wss (the encrypted WebSocket protocol) replaces https
* there are four events in a socket
  + open (the connection was established)
  + message (data was received)
  + error (there was an error)
  + close (the connection was closed)
* can call the send() method on the WebSocket instance to send data
* data can be sent and received as a string or binary data

Breakpoint Types

|  |  |
| --- | --- |
| Breakpoint Type | Pauses execution |
| Line-of-Code | On an exact line of code |
| Conditional Line-of-Code | On an exact line of code when some other condition is true |
| DOM | On the code that changes or removes a specific DOM node, or its children |
| XHR | When an XHR URL contains a string pattern |
| Event Listener | On the code that runs after an event is fired |
| Exception | On the line of code that is throwing a caught or uncaught exception |
| Function | Whenever a specific function is called |

Sources Panes in Chrome DevTools

* Scope
  + Shows currently defined local and global variables and their values
* Watch
  + Allows storing of values and expressions to monitor over time

Console API

* console.assert()
  + writes an error message to the console if the assertion is false
  + takes an assertion as a first parameter
  + can take following parameters as objects or strings to output
* console.info()
  + outputs an informational message to the console
  + takes parameters as objects or strings to output
* console.error()
  + outputs an error message to the console
  + takes parameters as objects or strings to output
* console.log()
  + outputs a message to the console
  + takes parameters as objects or strings to output

URL API

* components of a URL object
  + href – the full URL
  + protocol – at the beginning of the URL, ends with a colon
  + search – a string of parameters, starts with a question mark
  + hash – starts with the # character
  + port – the port of the URL
  + hostname – the URL domain
  + host – the hostname and port
  + origin – the protocol, hostname, and port
  + protocol//hostname:port/pathname?search#hash

Black-Box Testing

* functional testing
* treats the software being tested as a black-box without knowing its internals
* traditionally done by separate quality engineers
* advantages
  + efficient for large segments of code
  + code access is not required
  + separation between user and developer perspectives
* disadvantages
  + limited coverage because only a fraction of test scenarios are performed
  + inefficient testing due to the tester’s lack of knowledge about the software internals
  + blind coverage because the tester has limited knowledge about the application

White-Box Testing

* structural testing
* knows the internals of the software being tested
* advantages
  + efficient in finding errors and problems
  + thorough testing due to knowledge of the internals of the software being tested
  + helps optimize the code
  + maximum coverage because the tester has internal knowledge of the application
* disadvantages
  + requires code access
  + requires high level knowledge of the internals of the software under test
* usually automated through unit tests

Generator Functions

* return generator objects to manage execution rather than executing when called
* e.g.

function\* generator() {

yield x;

}

* next()
  + returns an object with two properties – value (holding the yielded value) and done (a Boolean indicating if the generator function has finished execution)
  + will execute the generator function until the next yield statement, at which point the generator function execution pauses and the value is returned to the calling code
  + once the generator function has finished execution, subsequent calls to its next() method will return {done: true}
* generators can be iterated through using the for-of loop
  + but this will ignore the final returned value of the generator function if it is returned with a return statement, rather than a yield statement
* way to implement iterables more concisely
* yield\*
  + directive that delegates execution to another generator
  + allows for generator composition, where an outer generator transparently iterates over an inner generator and forwards its yielded values

Function\*

throw

* statement used to throw a user-defined exception
* the throw keyword is followed by the expression containing the error message to use as the value of the exception

Error Object Properties

|  |  |
| --- | --- |
| Property | Meaning |
| message | Error message |
| name | Error name |

Shadow DOM

* a DOM element can have two types of DOM subtrees
  + a light tree, made of HTML children
  + a shadow tree, not reflected in HTML
* a shadow tree can be used in custom elements to hide component internals and apply component-specific styles
* the shadow root contains the shadow DOM
  + there can only be one shadow root per element
* shadow DOM elements aren’t visible to querySelector() calls from the light DOM
* shadow DOM elements may have ids that conflict with those in the light DOM, they only need to be unique within the shadow tree