This:

If the this is set to undefined or null, it gets substituted to globalThis.

If this is set to a primitive val, it get's substituted to that primitive value's wrapper.

JS

```
function getThisStrict() {
  "use strict"; // Enter strict mode
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThisStrict = getThisStrict;
console.log(typeof (1).getThisStrict()); // "number"
```

In use strict mode, this gives undefined if not set. While in normal mode it'll point to globalThis or window object.

Json.parse and stringify are binded to the obj they're called on.

In case of arrow functions, NO matter how it's called, this is bound to what it was at the time of creation. Bind, call, apply also don't work on arr fn.

In functions if we call some method from super class then the this STILL refers to the curr obj.

In classes, this refers to the Class for static methods.

Derived class constructors

Unlike base class constructors, derived constructors have no initial `this` binding. Calling `super()` creates a `this` binding within the constructor and essentially has the effect of evaluating the following line of code, where `Base` is the base class:

```
JS
this = new Base();
```

Warning: Referring to `this` before calling `super()` will throw an error.

Derived classes must not return before calling `super()`, unless the constructor returns an object (so the `this` value is overridden) or the class has no constructor at all.

Object fields does not create this scope, they inherits from the surrounding scope.

Prototype:

Prototype property of a class is not writable.

Arrow and async function do NOT have a prototype.

If we reassign func's prototype property to something Other

Note: Following the ECMAScript standard, the notation `someObject.[[Prototype]]` is used to designate the prototype of `someObject`. The `[[Prototype]]` internal slot can be accessed and modified with the `Object.getPrototypeOf()` and `Object.setPrototypeOf()` functions respectively. This is equivalent to the JavaScript accessor `__proto__` which is non-standard but de-facto implemented by many JavaScript engines. To prevent confusion while keeping it succinct, in our notation we will avoid using `obj.__proto__` but use `obj.[[Prototype]]` instead. This corresponds to `Object.getPrototypeOf(obj)`.

It should not be confused with the `func.prototype` property of functions, which instead specifies the `[[Prototype]]` to be assigned to all *instances* of objects created by the given function when used as a constructor. We will discuss the `prototype` property of constructor functions in a later section.

Every instance created from a constructor will have Constructor's prototype.

Object.create()

The `Object.create()` static method creates a new object, using an existing object as the prototype of the newly created object.

Try it

JavaScript Demo: Object.create()

```
const person = {
  isHuman: false,
  printIntroduction: function () {
    console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
  },
};

const me = Object.create(person);

me.name = 'Matthew'; // "name" is a property set on "me", but not on "person"
me.isHuman = true; // Inherited properties can be overwritten

me.printIntroduction();
// Expected output: "My name is Matthew. Am I human? true"

```

Arrow functions does not have [[Prototype]] obj.

event.stopImmediatePropagation()

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.

In other words, `event.stopPropagation()` stops the move upwards, but on the current element all other handlers will run.

To stop the bubbling and prevent handlers on the current element from running, there's a method `event.stopImmediatePropagation()`. After it no other handlers execute.

Use for...in to iterate over an obj

1. `Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any the given promises rejects, it becomes the error of `Promise.all`, and all other results are ignored.
2. `Promise.allSettled(promises)` (recently added method) – waits for all promises to settle and retur their results as an array of objects with:
   - `status`: `"fulfilled"` or `"rejected"`
   - `value` (if fulfilled) or `reason` (if rejected).
3. `Promise.race(promises)` – waits for the first promise to settle, and its result/error becomes the outcome.
4. `Promise.any(promises)` (recently added method) – waits for the first promise to fulfill, and its result becomes the outcome. If all of the given promises are rejected, `AggregateError` becomes the error of `Promise.any`.
5. `Promise.resolve(value)` – makes a resolved promise with the given value.
6. `Promise.reject(error)` – makes a rejected promise with the given error.

Of all these, `Promise.all` is probably the most common in practice.

JS converts num to str in (str + num) operation

Object.entries/fromEntries

You can avoid writing then for resolved promises but you HAVE to write catch for rejected promises

`Promise.all(iterable)` allows non-promise "regular" values in `iterable`

Normally, `Promise.all(...)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is".

For instance, here the results are `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to `Promise.all` where convenient.

Num.toFixed() returns a string

For...of and forEach are arr specific methods and does not work on obj.

In json.stringify() if the value of a key is undefined or a function, then it is not converted and Omitted. While NaN is converted to null

Classes does not return anything. So if you write, let c1 = Person('dev'), c1 will remain undefined.

Things to read:

data- dataset

Summary

- Methods to create new nodes:
  - `document.createElement(tag)` – creates an element with the given tag,
  - `document.createTextNode(value)` – creates a text node (rarely used),
  - `elem.cloneNode(deep)` – clones the element, if `deep==true` then with all descendants.
- Insertion and removal:
  - `node.append(...nodes or strings)` – insert into `node`, at the end,
  - `node.prepend(...nodes or strings)` – insert into `node`, at the beginning,
  - `node.before(...nodes or strings)` –- insert right before `node`,
  - `node.after(...nodes or strings)` –- insert right after `node`,
  - `node.replaceWith(...nodes or strings)` –- replace `node`.
  - `node.remove()` –- remove the `node`.

  Text strings are inserted "as text".

Elem.insertAdjacentHTML(where, html)

Elem.style.removeProperty('style-property')

Allow focusing on any element: tabindex

By default, many elements do not support focusing.

The list varies a bit between browsers, but one thing is always correct: `focus/blur` support is guaranteed for elements that a visitor can interact with: `<button>`, `<input>`, `<select>`, `<a>` and so on.

On the other hand, elements that exist to format something, such as `<div>`, `<span>`, `<table>` – are unfocusable by default. The method `elem.focus()` doesn't work on them, and `focus/blur` events are never triggered.

This can be changed using HTML-attribute `tabindex`.

Any element becomes focusable if it has `tabindex`. The value of the attribute is the order number of the element when `Tab` (or something like that) is used to switch between them.

That is: if we have two elements, the first has `tabindex="1"`, and the second has `tabindex="2"`, then pressing `Tab` while in the first element – moves the focus into the second one.

The switch order is: elements with `tabindex` from `1` and above go first (in the `tabindex` order), and then elements without `tabindex` (e.g. a regular `<input>`).

Elements without matching `tabindex` are switched in the document source order (the default order).

There are two special values:

- `tabindex="0"` puts an element among those without `tabindex`. That is, when we switch elements, elements with `tabindex=0` go after elements with `tabindex ≥ 1`.

  Usually it's used to make an element focusable, but keep the default switching order. To make an element a part of the form on par with `<input>`.
- `tabindex="-1"` allows only programmatic focusing on an element. The `Tab` key ignores such elements, but method `elem.focus()` works.

Elem.removeEventListener() how it works

Custom events and dispatcEvent()

Arr.find/findIndex/reduceRight/join/from

method	selects…	negatives
`slice(start, end)`	from `start` to `end` (not including `end`)	allows negatives
`substring(start, end)`	between `start` and `end` (not including `end`)	negative values mean `0`
`substr(start, length)`	from `start` get `length` characters	allows negative `start`

Str.indexOf/includes/startsWith/endsWith

Delete obj.property

Obj.keys/values/entries

Oninput and onchange

How to create promise

Window.addEven...('unhandledrejection')

IsNaN/isFinite

Number.isNaN/isFinite

Num.toFixed()

Math.trunc/round