JS Course

### Type Conversion

<https://www.freecodecamp.org/news/js-type-coercion-explained-27ba3d9a2839/>

### null Vs undefined

* null and undefined are falsy values.
* null and undefined are both primitives. However an error shows that typeof null = object.

NULL : There are two features of null you should understand:

* null is an empty or non-existent value.
* null must be assigned.

*let* a = null;

console.log(a); *//null*

console.log(typeof a); *//object*

Undefined: Undefined most typically means a variable has been declared, but not defined.

*let* b;

console.log(b); // undefined

console.log(typeof b); *// undefined*

console.log(null !== undefined); *// true*

console.log(null == undefined); *// true*

console.log(null === undefined); *// false*

In JavaScript, a double equals tests for **loose equality** and preforms type coercion. This means we compare two values after converting them to a common type.

*var* test = {};

console.log(test); *// {}*

console.log(test.prop); *// undefined*

*const* doNothing = () => {};

console.log(doNothing()); *// undefined*

*const* someObj = {

  a: "ay",

  b: "bee",

  c: "si",

};

console.log(someObj.d); *//undefined*

### NaN

NaN : The number type in JavaScript is a set of all number values, including “Not A Number”, positive infinity and negative infinity.

“Not A Number” can be accessed using a special expression NaN, or as a property of the global object or Number function:

console.log(typeof NaN); *//number*

*var* a = NaN;

console.log(a); *// NaN*

console.log(typeof a); *// number*

The interesting property of NaN is that it doesn’t equal to any value, even with the NaN itself:

console.log(isNaN(NaN)); *// true*

console.log(NaN === NaN); *// false*

console.log(Number.NaN === NaN); *// false*

console.log(isNaN(Number.NaN)); *// true*

console.log(Number.isNaN(NaN)); *// true*

console.log(isNaN("hello world")); *// true*

console.log(Number.isNaN("hello world")); *// false*

<https://dmitripavlutin.com/nan-in-javascript/>

### == vs ===

Well in short: == inherently converts type and === does not convert type.

Double Equals (==) checks for value equality only. It inherently does type coercion. This means that before checking the values, it converts the types of the variables to match each other.

On the other hand, Triple Equals (===) does not perform type coercion. It will verify whether the variables being compared have both the same value **AND** the same type.

*let* Dhoni = "Cricketer";

*let* Bumrah = "Cricketer";

console.log(Dhoni == Bumrah); //true

console.log(Dhoni === Bumrah); //true

The value and the type of both Dhoni and Bumrah is same. Therefore the result is true for both.

*let* NumberOfDhoni = 7; *//NUmber*

*let* WicketOfBumrah = "7"; *// String*

console.log(NumberOfDhoni == WicketOfBumrah); *// true*

console.log(NumberOfDhoni === WicketOfBumrah); *// false*

The value of NumberOfDhoni and WicketOfBumrah looks similar here. However, the type of NumberOfDhoni is Number and type of  WicketOfBumrah is string. Even though the values are same, the type is not the same. Hence a == check returns true, but when checked for value **and** type, the value is false.

console.log(0 == false); *//true*

console.log(0 === false); *//false*

Reason: same value, different type. Type coercion

Object Comparison

*let* dhoni = {

  Profession: "cricketer",

};

reference point to : “Batsman”

*let* malinga = {

  Profession: "cricketer",

};

reference point to : “Bowler”

console.log(dhoni == malinga); // false

console.log(dhoni === malinga); // false

In this example dhoni and malinga object values are same but ans is false because only simple reason is that both have point to different different address for example dhoni Profession is crickter but type is batsman and malinga Profession is crickter but type is bowler both pointing to different area.

*let* dhoni = {

  Profession: "Wk/BatsMan",

};

*let* pant = dhoni;

console.log(dhoni == pant); // true

console.log(dhoni === pant); // true

### Objects

### Reference Vs Value

### Object Properties

### Object this keyword

It’s common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside Person.CalcAge() may need the age of the Person.

**To access the object, a method can use the this keyword.**

The value of this is the object “before dot”, the one used to call the method.

This work on the different different scenarrio

1] method <obj is calling the Method>

this refers to the owner of the object

*const* Person = {

  name: "Bhavesh",

  year: 2000,

  calcAge: *function* () {

    console.log(*this*); *//{ name: 'Bhavesh', year: 2000, calcAge: [Function: calcAge] }*

    console.log(2021 - *this*.year); *// 21*

  },

};

Person.calcAge();

Here during the execution of Person.CalcAge(), the value of this will be user.

Technically, it’s also possible to access the object without this, by referencing it via the outer variable:

*const* Person = {

  name: "Bhavesh",

  year: 2000,

  calcAge: *function* () {

    console.log(Person.name); *// Bhavesh*

    console.log(2021 - Person.year); *// 21*

  },

};

Person.calcAge();

…But such code is unreliable. If we decide to copy Person to another variable, e.g. admin = Person and overwrite Person with something else, then it will access the wrong object.

That’s demonstrated below:

*const* Person = {

  name: "Bhavesh",

  year: 2000,

  calcAge() {

    console.log(Person.name); *// Bhavesh*

  },

};

*let* admin = Person;

Person = null;

admin.calcAge();

If we used this.name instead of Person.name inside the  console, then the code would work.

**2] Calling without an object: this == undefined, window, and strict mode**

*function* Person() {

  console.log(*this*);

}

Person();

If run: this pointing to window

*function* Person() {

  console.log(*this*.name);

}

Person();

If run: this pointing to undefined

"use strict";

*function* Person() {

  console.log(*this*.name);

}

Person();

If run: TypeError Cannot read property 'name' of undefined

In this case this is undefined in strict mode. If we try to access this.name, there will be an error.

In non-strict mode the value of this in such case will be the global object (window in a browser, we’ll get to it later in the chapter [Global object](https://javascript.info/global-object)). This is a historical behavior that "use strict" fixes.

3] Arrow function have no this()

Arrow functions are special: they don’t have their “own” this. If we reference this from such a function, it’s taken from the outer “normal” function.

For instance, here arrow() uses this from the outer Person.sayHi() method:

*const* ArrowFun = () => {

  console.log(*this*); //point to window

};

ArrowFun();

*const* call = {

  caller: "mom",

  says: () => {

    console.log(`Hey, ${*this*.caller} just called.`);

  },

};

call.says();

// Hey, undefined just called

*let* Person = {

  firstName: "Bhavesh",

  sayHi() {

*let* arrow = () => console.log(*this*.firstName);

    arrow();

  },

};

Person.sayHi(); // Bhavesh

That’s a special feature of arrow functions, it’s useful when we actually do not want to have a separate this, but rather to take it from the outer context

The value of this is defined at run-time.

* When a function is declared, it may use this, but that this has no value until the function is called.
* A function can be copied between objects.
* When a function is called in the “method” syntax: object.method(), the value of this during the call is object.

### Type

### Callback, Promises, async await

Callback Function: A callback is a function passed as an argument to another function.

This technique allows a function to call another function.

**Simply put:** A callback is a function that is to be executed **after** another function has finished executing — hence the name ‘call back’.

**More complexly put:** In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called **higher-order functions**. Any function that is passed as an argument is called a **callback function**.

*function* greet(*name*, *callback*) {

  console.log("Hello" + " " + name);

  callback();

}

*function* callMe() {

  console.log("I am a callback function");

}

greet("peter", callMe);

//Hello peter

//I am a callback function

In the above prog, there are two function while calling the greet() fun, two arguments are passed.The callme() fun is a callback fun

*function* CALLBACK() {

  setTimeout(() => {

    console.log("Waited 1 sec callback");

  }, 1000);

  console.log("Function executed first");

}

CALLBACK();

//Function executed first

//Waited 1 sec callback

CallBack hell (nested callback)

setTimeout(() => {

  console.log("callback1 run after 1 sec");

  setTimeout(() => {

    console.log("callback2 run after 2 sec");

    setTimeout(() => {

      console.log("callback3 run after 3 sec");

    }, 1000);

  }, 1000);

}, 1000);

//callback1 run after 1 sec

//callback2 run after 2 sec

//callback3 run after 3 sec

Promise():

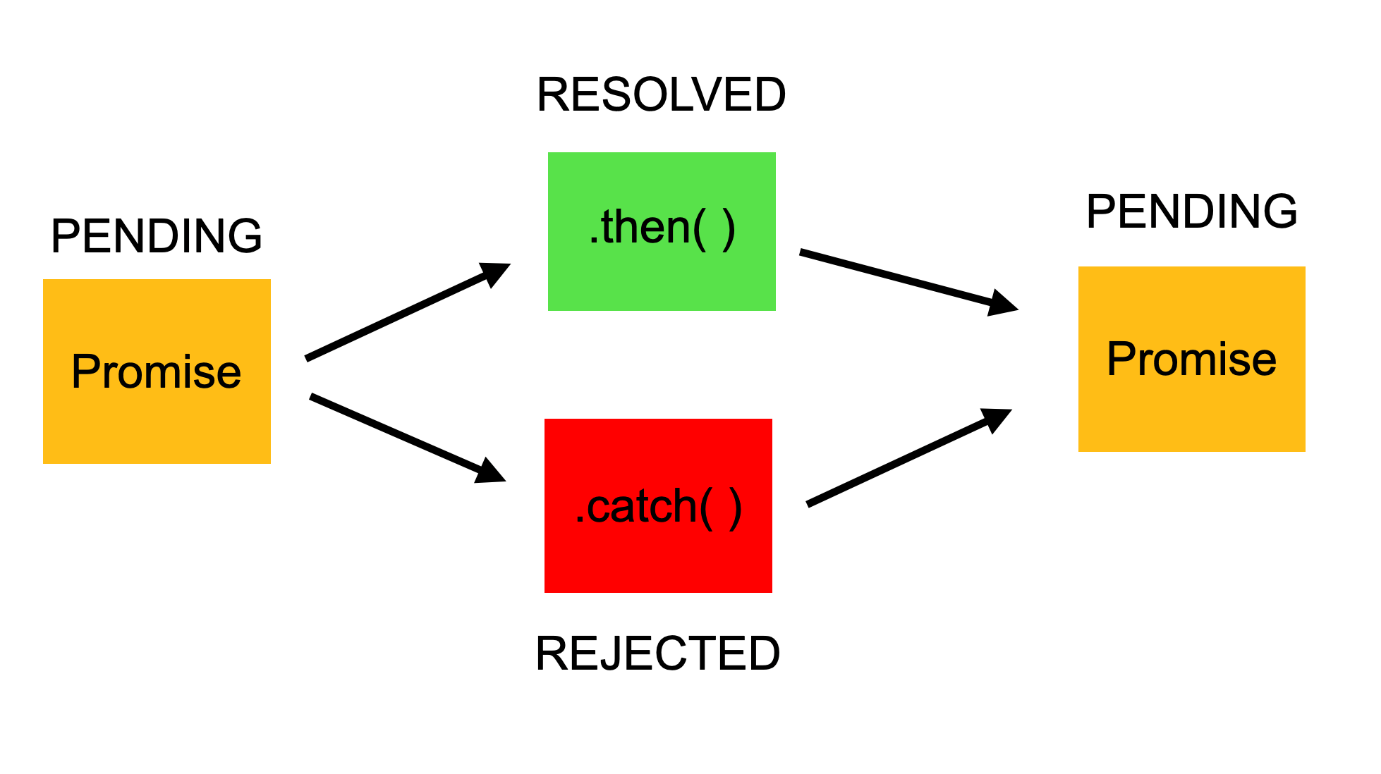
A promise in JavaScript is similar to a promise in real life. When we make a promise in real life, it is a guarantee that we are going to do something in the future. Because promises can only be made for the future.

A promise has 2 possible outcomes: it will either be kept when the time comes, or it won’t.

This is also the same for promises in JavaScript. When we define a promise in JavaScript, it will be resolved when the time comes, or it will get rejected.

First of all, a Promise is an object. There are 3 states of the Promise object:

* **Pending:** Initial State, before the Promise succeeds or fails
* **Resolved:** Completed Promise
* **Rejected:** Failed Promise



For example, when we request data from the server by using a Promise, it will be in pending mode until we receive our data.

If we achieve to get the information from the server, the Promise will be resolved successfully. But if we don’t get the information, then the Promise will be in the rejected state.

//Create a promise

*const* myPromise = new Promise((*resolve*, *reject*) => {

*const* rand = Math.floor(Math.random() \* 2);

  if (rand === 0) {

    resolve();

  } else {

    reject();

  }

});

myPromise

  .then(() => console.log("Success"))

  .catch(() => console.log("Error occured"));

In the above example if value is 0 then my promise resolved and print success if not the reject will throw an error.

Additionally, if there are multiple requests, then after the first Promise is resolved (or rejected), a new process will start to which we can attach it directly by a method called chaining.

fetch("https://pokeapi.co/api/v2/pokemon/ditto")

  .then((*res*) => res.json())

  .then((*data*) => console.log(data))

  .catch((*err*) => console.error(err));

Async awit():*async and await make promises easier to write"*

**async** makes a function return a Promise

**await** makes a function wait for a Promise

*const* fetchPokemon = *async* (*id*) => {

  try {

*const* res = await fetch(`https://pokeapi.co/api/v2/pokemon/${id}`);

*const* data = await res.json();

    console.log(data);

  } catch (error) {

    console.error(err);

  }

};

fetchPokemon(2);

### Type

### Type