**BASICS**

1. **isNaN**
   1. The isNaN() function determines whether a value is [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN), first converting the value to a number if necessary. Because coercion inside the isNaN() function can be [surprising](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN" \l "description), you may prefer to use [Number.isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN).
   2. The isNaN() function answers the question "is the input functionally equivalent to [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) when used in a number context". If isNaN(x) returns false, you can use x in an arithmetic expression as if it's a valid number that's not NaN. If isNaN(x) returns true, x will get coerced to NaN and make most arithmetic expressions return NaN (because NaN propagates).
   3. **isNaN(NaN); // true**

**isNaN(undefined); // true**

**isNaN({}); // true**

**isNaN(true); // false as true is 1**

**isNaN(null); // false as it is 0**

**isNaN(37); // false**

**// Strings**

**isNaN("37"); // false: "37" is converted to the number 37 which is not NaN**

**isNaN("37.37"); // false: "37.37" is converted to the number 37.37 which is not NaN**

**isNaN("37,5"); // true**

**isNaN("123ABC"); // true: Number("123ABC") is NaN**

**isNaN(""); // false: the empty string is converted to 0 which is not NaN**

**isNaN(" "); // false: a string with spaces is converted to 0 which is not NaN**

**// Dates**

**isNaN(new Date()); // false; Date objects can be converted to a number (timestamp)**

**isNaN(new Date().toString()); // true; the string representation of a Date object cannot be parsed as a number**

1. **Number.isNaN**
   1. The Number.isNaN() static method determines whether the passed value is the number value [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN), and returns false if the input is not of the Number type. It is a more robust version of the original, global [isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN) function.
   2. As opposed to the global [isNaN()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/isNaN) function, the Number.isNaN() method doesn't force-convert the parameter to a number. This makes it safe to pass values that would normally convert to [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) but aren't actually the same value as [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN). This also means that only values of the Number type that are also [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) return true.

Number.isNaN(NaN); // true

Number.isNaN(Number.NaN); // true

Number.isNaN(0 / 0); // true

Number.isNaN(37); // false

* 1. Number.isNaN() doesn't attempt to convert the parameter to a number, so non-numbers always return false. The following are all false:

Number.isNaN("NaN");

Number.isNaN(undefined);

Number.isNaN({});

Number.isNaN("blabla");

Number.isNaN(true);

Number.isNaN(null);

Number.isNaN("37");

Number.isNaN("37.37");

Number.isNaN("");

Number.isNaN(" ");

1. **JSON**
   1. **JSON.parse()**
      1. Parse a piece of string text as JSON, optionally transforming the produced value and its properties, and return the value.
      2. Example

const jsonText = `{

"browsers": {

"firefox": {

"name": "Firefox",

"pref\_url": "about:config",

"releases": {

"1": {

"release\_date": "2004-11-09",

"status": "retired",

"engine": "Gecko",

"engine\_version": "1.7"

}

}

}

}

}`;

console.log(JSON.parse(jsonText));

* 1. **JSON.stringify()**
     1. Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner.
     2. **Example**

const foo = {

foundation: "Mozilla",

model: "box",

week: 45,

transport: "car",

month: 7,};

JSON.stringify(foo);

* 1. **LocalStorage**

**// Creating an example of JSON**

**const session = {**

**screens: [],**

**state: true,**

**};**

**session.screens.push({ name: "screenA", width: 450, height: 250 });**

**session.screens.push({ name: "screenB", width: 650, height: 350 });**

**session.screens.push({ name: "screenC", width: 750, height: 120 });**

**session.screens.push({ name: "screenD", width: 250, height: 60 });**

**session.screens.push({ name: "screenE", width: 390, height: 120 });**

**session.screens.push({ name: "screenF", width: 1240, height: 650 });**

**// Converting the JSON string with JSON.stringify()**

**// then saving with localStorage in the name of session**

**localStorage.setItem("session", JSON.stringify(session));**

**// Example of how to transform the String generated through**

**// JSON.stringify() and saved in localStorage in JSON object again**

**const restoredSession = JSON.parse(localStorage.getItem("session"));**

**// Now restoredSession variable contains the object that was saved**

**// in localStorage**

**console.log(restoredSession);**

1. **Map**
   1. The Map object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)) may be used as either a key or a value.
   2. A key in the Map **may only occur once**; it is unique in the Map's collection. A Map object is iterated by key-value pairs — a [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) loop returns a 2-member array of [key, value] for each iteration. Iteration happens in insertion order, which corresponds to the order in which each key-value pair was first inserted into the map by the [set()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/set) method (that is, there wasn't a key with the same value already in the map when set() was called).
   3. Example

const contacts = new Map();

contacts.set("Jessie", { phone: "213-555-1234", address: "123 N 1st Ave" });

contacts.has("Jessie"); // true

contacts.get("Hilary"); // undefined

contacts.set("Hilary", { phone: "617-555-4321", address: "321 S 2nd St" });

contacts.get("Jessie"); // {phone: "213-555-1234", address: "123 N 1st Ave"}

contacts.delete("Raymond"); // false

contacts.delete("Jessie"); // true

console.log(contacts.size); // 1

* 1. **Iterating map with for-of**

**const myMap = new Map();**

**myMap.set(0, "zero");**

**myMap.set(1, "one");**

**for (const [key, value] of myMap) {**

**console.log(`${key} = ${value}`);**

**}**

**// 0 = zero**

**// 1 = one**

**for (const key of myMap.keys()) {**

**console.log(key);**

**}**

**// 0**

**// 1**

**for (const value of myMap.values()) {**

**console.log(value);**

**}**

**// zero**

**// one**

**for (const [key, value] of myMap.entries()) {**

**console.log(`${key} = ${value}`);**

**}**

**// 0 = zero**

**// 1 = one**

* 1. **Iterating map with forEach**

**myMap.forEach((value, key) => {**

**console.log(`${key} = ${value}`);**

**});// 0 = zero// 1 = one**

* 1. **new Map(iterable)**
     1. An [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) or other [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) object whose elements are key-value pairs. (For example, arrays with two elements, such as [[ 1, 'one' ],[ 2, 'two' ]].) Each key-value pair is added to the new Map.

1. **parseFloat()**
   1. The parseFloat() function parses a string argument and returns a floating point number.
   2. Return:A floating point number parsed from the given string, or [NaN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN) when the first non-whitespace character cannot be converted to a number.
   3. Note-:If string starts with a digit then it will convert all the digits into float value and return whereas if string starts with some other alphabet then it will give NaN
2. **parseInt()**
   1. **Same as parseFloat(),the only change is it parses a string argument and return integer or NaN**
3. **Set**
   1. The Set object lets you store unique values of any type, whether [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive) or object references.
   2. . You can iterate through the elements of a set in insertion order. The insertion order corresponds to the order in which each element was inserted into the set by the [add()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/add) method successfully (that is, there wasn't an identical element already in the set when add() was called).
   3. Example

**const mySet1 = new Set();**

**mySet1.add(1); // Set(1) { 1 }**

**mySet1.add(5); // Set(2) { 1, 5 }**

**mySet1.add(5); // Set(2) { 1, 5 }**

**mySet1.add("some text"); // Set(3) { 1, 5, 'some text' }**

**const o = { a: 1, b: 2 };**

**mySet1.add(o);**

**mySet1.add({ a: 1, b: 2 }); // o is referencing a different object, so this is okay**

**mySet1.has(1); // true**

**mySet1.has(3); // false, since 3 has not been added to the set**

**mySet1.has(5); // true**

**mySet1.has(Math.sqrt(25)); // true**

**mySet1.has("Some Text".toLowerCase()); // true**

**mySet1.has(o); // true**

**mySet1.size; // 5**

**mySet1.delete(5); // removes 5 from the set**

**mySet1.has(5); // false, 5 has been removed**

**mySet1.size; // 4, since we just removed one value**

**mySet1.add(5); // Set(5) { 1, 'some text', {...}, {...}, 5 } - a previously deleted item will be added as a new item, it will not retain its original position before deletion**

**console.log(mySet1); // Set(5) { 1, "some text", {…}, {…}, 5 }**

* 1. **Iterating sets**

for (const item of mySet1) {

console.log(item);}// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

for (const item of mySet1.keys()) {

console.log(item);}// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

for (const item of mySet1.values()) {

console.log(item);}// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

// key and value are the same herefor (const [key, value] of mySet1.entries()) {

console.log(key);}// 1, "some text", { "a": 1, "b": 2 }, { "a": 1, "b": 2 }, 5

* 1. **.Set object to Array object**

const myArr = Array.from(mySet1); // [1, "some text", {"a": 1, "b": 2}, {"a": 1, "b": 2}, 5]

* 1. **Array object to set object**

const numbers = [2, 3, 4, 4, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 5, 32, 3, 4, 5];

console.log([...new Set(numbers)]);

1. **Undefined**
   1. The undefined global property represents the primitive value [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures" \l "undefined_type). It is one of JavaScript's [primitive types](https://developer.mozilla.org/en-US/docs/Glossary/Primitive).
   2. A variable that has not been assigned a value is of type undefined. A method or statement also returns undefined if the variable that is being evaluated does not have an assigned value. A function returns undefined if a value was not [returned](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/return)