Sommario

In JavaScript, `NaN` sta per "Not-a-Number" ed è un valore speciale che rappresenta il risultato di operazioni matematiche o di conversione che non hanno un valore numerico valido. È una proprietà dell'oggetto globale `Number` e può essere ottenuta tramite `Number.NaN`, anche se è comunemente utilizzata come `NaN`.

Caratteristiche di `NaN`

1. **Tipo di dato**:
   - `NaN` è un valore del tipo `number`.
   - Esempio: `typeof NaN` restituisce `"number"`.
2. **Unicità di `NaN`**:
   - `NaN` è l'unico valore in JavaScript che non è uguale a se stesso.
   - Confrontare `NaN` con `NaN` (es. `NaN === NaN`) restituisce `false`.
3. **Origine di `NaN`**: `NaN` si genera in diversi scenari, tra cui:
   - Operazioni matematiche non valide:

     ```
     const result = 0 / 0; // NaN
     const invalidMath = Math.sqrt(-1); // NaN
     ```

   - Conversione di valori non numerici in numeri:

     ```
     const invalidConversion = Number("abc"); // NaN
     ```

```
const parseError = parseInt("hello"); // NaN
```

4. **Valore "infetto"**:
   - Una volta che un'operazione produce `NaN`, qualsiasi altra operazione che coinvolge `NaN` produrrà anch'essa `NaN`.
   - Esempio:

```
const x = NaN;
const y = x + 5; // NaN
```

Come controllare `NaN`

Poiché `NaN` non è uguale a se stesso, non è possibile utilizzare l'operatore di uguaglianza (`===` o `==`) per verificare la sua presenza. Esistono invece metodi e funzioni specifiche:

1. **`isNaN()`**:
   - La funzione globale `isNaN()` controlla se un valore è `NaN`.
   - Esempio:

```
isNaN(NaN); // true
isNaN("hello"); // true (comportamento confuso)
```

   - Nota: Questa funzione tenta di convertire il valore in un numero prima di verificare se è `NaN`. Ad esempio, `isNaN("hello")` restituisce `true` perché `"hello"` non può essere convertito in un numero.
2. **`Number.isNaN()`**:
   - Introdotta in ECMAScript 6, `Number.isNaN()` controlla strettamente se un valore è `NaN` senza tentare di convertirlo.
   - Esempio:

```
Number.isNaN(NaN); // true
Number.isNaN("hello"); // false
Number.isNaN(undefined); // false
```

3. **Confronto implicito**:
   - A causa del comportamento unico di `NaN`, il controllo esplicito è necessario; non ci sono altre proprietà implicite che lo rivelino direttamente.

Utilizzi pratici

1. **Validazione di input numerici**:
   - Si usa `Number.isNaN()` per verificare se un input è un numero valido:

```
function validateNumber(input) {
  if (Number.isNaN(input)) {
    return "Il valore non è un numero valido.";
  }
  return "Valore accettabile.";
}
```

2. **Errori in calcoli matematici**:
   - Se un calcolo matematico restituisce `NaN`, potrebbe essere necessario gestirlo:

```
const result = Math.log(-1);
if (Number.isNaN(result)) {
  console.error("Errore matematico: risultato non valido.");
}
```

3. **Conversioni da stringa a numero**:
   - Usando `parseFloat`, `parseInt` o `Number()`, è essenziale verificare l'output per evitare errori futuri:

```
const userInput = "abc";
const num = Number(userInput);
if (Number.isNaN(num)) {
  console.log("Input non valido.");
}
```

Punti da ricordare

- `NaN` indica l'impossibilità di rappresentare un numero.
- Non è uguale a se stesso (`NaN !== NaN`).
- Usa `Number.isNaN()` per controlli rigorosi.
- Non confondere `isNaN()` con `Number.isNaN()`; il primo può restituire `true` per input non numerici come stringhe.

In JavaScript, i **tipi di dato** (o "data types") definiscono il tipo di valore che una variabile può contenere. I tipi di dato si dividono in due categorie principali: **tipi primitivi** e **tipi complessi**. Ecco una spiegazione dettagliata:

1. Tipi Primitivi

I tipi primitivi rappresentano valori immutabili e semplici. JavaScript offre sette tipi primitivi:

a) `undefined`

- Rappresenta un valore non definito.
- Una variabile ha il valore `undefined` se non le è stato assegnato alcun valore.
- Esempio:

```
let x;
console.log(x); // undefined
```

b) `null`

- Rappresenta l'assenza intenzionale di un valore.
- Viene spesso utilizzato per indicare che una variabile non ha un valore.
- Differisce da `undefined` in quanto `null` viene assegnato esplicitamente.
- Esempio:

```
let y = null;
console.log(y); // null
```

c) `boolean`

- Rappresenta un valore logico: `true` o `false`.
- Utilizzato per controllare condizioni e cicli.
- Esempio:

```
const isActive = true;
console.log(isActive); // true
```

d) `number`

- Rappresenta sia numeri interi che numeri a virgola mobile (decimali).
- Supporta anche valori speciali come:
  - `Infinity`: risultato di una divisione per 0.
  - `-Infinity`: risultato di una divisione negativa per 0.
  - `NaN` (Not-a-Number): risultato di operazioni matematiche non valide.
- Esempio:

```
const num1 = 42;        // Intero
const num2 = 3.14;      // Decimale
const result = 0 / 0;   // NaN
```

e) `bigint` (introdotto in ES2020)

- Rappresenta numeri interi di dimensioni arbitrariamente grandi.
- È utile quando i numeri superano il limite di precisione di `number` (2^{53} - 1).
- I valori `bigint` si definiscono aggiungendo `n` alla fine del numero.
- Esempio:

```
const bigNum = 1234567890123456789012345678901234567890n;
```

f) `string`

- Rappresenta una sequenza di caratteri, delimitata da apici singoli (`'`), doppi (`"`) o backtick (`` ` ``).
- Supporta stringhe multilinea e interpolazione con template literals (`` ` ``).
- Esempio:

```
const greeting = "Hello";
const name = 'John';
const message = `Hello, ${name}!`; // Interpolazione
```

g) `symbol` (introdotto in ES6)

- Rappresenta un identificatore univoco e immutabile.
- Viene spesso utilizzato come chiave unica per proprietà negli oggetti.
- Esempio:

```
const sym1 = Symbol("description");
const sym2 = Symbol("description");
console.log(sym1 === sym2); // false
```

2. Tipi Complessi

I tipi complessi (o "reference types") rappresentano strutture dati più elaborate. Questi includono:

a) `object`

- Gli oggetti sono collezioni di coppie chiave-valore.
- Le chiavi possono essere stringhe o simboli; i valori possono essere qualsiasi tipo di dato.
- Esempio:

```
const person = {
  name: "Alice",
  age: 25,
  isStudent: true,
};
```

Tipologie di oggetti specifici:

1. **Array**:
   - Collezione ordinata di elementi.
   - Accesso tramite indice (a partire da 0).
   - Esempio:

     ```
     const fruits = ["apple", "banana", "cherry"];
     console.log(fruits[1]); // "banana"
     ```

2. **Function**:
   - Oggetti invocabili che eseguono un'operazione.
   - Esempio:

     ```
     function greet(name) {
       return `Hello, ${name}`;
     }
     ```

3. **Date**:
   - Oggetto per la gestione di date e orari.
   - Esempio:

     ```
     const today = new Date();
     console.log(today.toISOString());
     ```

4. **RegExp**:
   - Oggetto per la manipolazione di espressioni regolari.
   - Esempio:

     ```
     const regex = /hello/i;
     console.log(regex.test("Hello")); // true
     ```

5. **Map e Set**:
   - **Map**: Collezione di coppie chiave-valore con chiavi di qualsiasi tipo.

     ```
     const map = new Map();
     map.set("key1", "value1");
     console.log(map.get("key1")); // "value1"
     ```

- **Set**: Collezione di valori unici.

```
const set = new Set([1, 2, 3, 3]);
console.log(set.size); // 3
```

3. Operatore `typeof`

Per determinare il tipo di dato di una variabile, si utilizza l'operatore `typeof`:

```
console.log(typeof 42); // "number"
console.log(typeof "hello"); // "string"
console.log(typeof true); // "boolean"
console.log(typeof undefined); // "undefined"
console.log(typeof null); // "object" (bug storico di JavaScript)
console.log(typeof {}); // "object"
console.log(typeof Symbol("id")); // "symbol"
```

4. Differenza tra Tipi Primitivi e Complessi

Caratteristica	Tipi Primitivi	Tipi Complessi
Archiviazione	Archiviati nello stack	Archiviati nell'heap
Mutabilità	Immutabili	Mutabili
Copia	Copia del valore	Copia del riferimento