Sommario

Sintassi e istruzioni JavaScript: informazioni preliminari

JavaScript è un linguaggio che si basa su una struttura semplice e flessibile, ma ci sono alcune regole fondamentali che devono essere rispettate per garantire che il codice funzioni correttamente. Di seguito vengono descritti concetti chiave come la sensibilità alle maiuscole, l'uso degli spazi, i commenti, il punto e virgola, le interruzioni di riga e il posizionamento del codice JavaScript.

1. Sensibilità alle maiuscole

JavaScript è un linguaggio **case-sensitive**, ovvero distingue tra maiuscole e minuscole. Questo vale per:

- **Identificatori**: nomi di variabili, funzioni e oggetti.
- **Parole chiave**: devono essere scritte esattamente come definite.

Esempio:

```
let nome = "Alice";
console.log(Nome); // Errore: "Nome" non è definito.
```

2. Spazi

JavaScript ignora gli spazi bianchi (spazi, tabulazioni e righe vuote), ma sono utili per rendere il codice leggibile. È buona pratica usare l'indentazione per evidenziare i blocchi di codice.

Esempio (codice leggibile):

```
if (condizione) {
  console.log("Condizione vera");
}
```

Esempio (codice confuso):

```
if(condizione){console.log("Condizione vera");}
```

3. Commenti

I commenti servono per spiegare il codice e non vengono eseguiti dal motore JavaScript. Esistono due tipi di commenti:

- **Commenti su una singola riga**:

```
// Questo è un commento singolo
```

- **Commenti su più righe**:

```
/*
  Questo è un commento
  su più righe
*/
```

4. Punto e virgola

In JavaScript, il punto e virgola (`;`) serve per terminare le istruzioni. Tuttavia, è **opzionale** in molti casi grazie all'**automatic semicolon insertion** (inserimento automatico del punto e virgola).

- **Buona pratica**: usare sempre il punto e virgola per evitare ambiguità.

Esempio con punto e virgola esplicito:

```
let a = 5;
let b = 10;
console.log(a + b);
```

Esempio senza punto e virgola:

```
let a = 5
let b = 10
console.log(a + b)
```

Tuttavia, ci sono situazioni dove la mancanza del punto e virgola può causare errori:

```
let a = 5
let b = 10
let somma
(a + b).toString(); // Errore: JavaScript interpreta la parentesi come
continuazione.
```

5. Interruzioni di riga

Le istruzioni possono essere suddivise su più righe, ma bisogna fare attenzione a evitare ambiguità. Le regole principali sono:

- Se l'istruzione è incompleta alla fine della riga, JavaScript la considera come continua.
- Usa parentesi o operatori per indicare chiaramente che il codice è legato.

Esempio corretto:

```
let somma =
  5 +
  10 +
  15;
console.log(somma);
```

Esempio ambiguo (da evitare):

```
let valore = 10
let risultato =
valore + 5 // Errore: potrebbe essere mal interpretato.
```

6. Posizionare correttamente il codice JavaScript

Il codice JavaScript può essere posizionato in tre modi principali in una pagina HTML:

- **Inline (direttamente nell'attributo `onclick`, `onload`, ecc.)**:

```
<button onclick="alert('Ciao!')">Clicca qui</button>
```

- **All'interno di un tag `<script>`**:

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      console.log("Questo è uno script inline.");
    </script>
  </head>
</html>
```

- **In un file esterno** (buona pratica per codice complesso o lungo):

```
<!DOCTYPE html>
<html>
  <head>
    <script src="script.js"></script>
  </head>
</html>
```

Il file `script.js` conterrà:

```
console.log("Questo è uno script in un file esterno.");
```

Posizionamento del tag `<script>`

Per evitare problemi durante il caricamento della pagina, è consigliabile:

- Posizionare il tag `<script>` **alla fine del corpo** della pagina HTML (prima di `</body>`).

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Hello, World!</h1>
    <script src="script.js"></script>
  </body>
</html>
```

- Oppure, aggiungere l'attributo `defer` al tag `<script>` per ritardare l'esecuzione fino al caricamento del DOM.

```
<script src="script.js" defer></script>
```

Seguendo queste regole e pratiche, il codice JavaScript sarà più leggibile, mantenibile e privo di errori accidentali. Se vuoi approfondire uno di questi punti, fammelo sapere!

Istruzioni JavaScript: una panoramica

In JavaScript, **un'istruzione** (in inglese "statement") è un'unità fondamentale di codice che esegue un'operazione specifica. Un programma JavaScript è costituito da una sequenza di istruzioni che vengono interpretate ed eseguite dal motore JavaScript in ordine sequenziale.

Cos'è un'istruzione?

Un'istruzione è un comando che il motore JavaScript deve eseguire. Ogni istruzione può:

- Dichiarare una variabile.
- Eseguire un calcolo.
- Controllare il flusso di esecuzione (es. condizionali o cicli).
- Effettuare un'operazione su dati.

Esempio di istruzione semplice:

```
let x = 5; // Assegna il valore 5 alla variabile x
console.log(x); // Stampa il valore di x sulla console
```

Ogni istruzione **può terminare con un punto e virgola** (`;`), anche se in molti casi JavaScript lo aggiunge automaticamente.

I due tipi principali di istruzioni JavaScript

Le istruzioni JavaScript si dividono in due categorie principali: **istruzioni di dichiarazione** e **istruzioni di controllo/esecuzione**.

1. Istruzioni di dichiarazione

Le istruzioni di dichiarazione servono per definire variabili, costanti o funzioni, che possono essere utilizzate successivamente nel programma.

a. Dichiarazione di variabili

Dichiarano e, opzionalmente, assegnano un valore a una variabile:

```
let x = 10; // Dichiarazione con inizializzazione
const PI = 3.14; // Dichiarazione di una costante
var y; // Dichiarazione senza inizializzazione
```

b. Dichiarazione di funzioni

Creano una funzione che può essere richiamata in seguito:

```
function saluta(nome) {
  console.log("Ciao, " + nome);
}
saluta("Alice"); // Richiama la funzione
```

c. Dichiarazione di classi (ES6+)

Definiscono una classe per implementare la programmazione a oggetti:

```
class Persona {
  constructor(nome, eta) {
    this.nome = nome;
    this.eta = eta;
  }
}
```

2. Istruzioni di controllo/esecuzione

Le istruzioni di controllo o esecuzione specificano come il programma deve comportarsi e quali operazioni eseguire.

a. Istruzioni di esecuzione semplice

Eseguono una singola operazione o espressione:

```
console.log("Hello, World!"); // Esegue un'operazione di output
x = x + 1; // Calcola un valore e aggiorna una variabile
```

b. Istruzioni condizionali

Controllano il flusso del programma in base a una condizione logica:

```
if (x > 10) {
  console.log("x è maggiore di 10");
} else {
  console.log("x è 10 o inferiore");
}
```

c. Istruzioni iterative (cicli)

Ripetono un blocco di codice più volte:

- **`for`**: per un numero definito di iterazioni.
- **`while`**: finché una condizione è vera.
- **`do...while`**: esegue almeno una volta, poi verifica la condizione.

Esempio:

```
for (let i = 0; i < 5; i++) {
  console.log(i); // Stampa i valori da 0 a 4
}
```

d. Istruzioni di salto

Interrompono o modificano il flusso di esecuzione:

- **`break`**: esce da un ciclo o da un blocco `switch`.
- **`continue`**: salta all'iterazione successiva di un ciclo.
- **`return`**: termina una funzione e restituisce un valore.

Esempio:

```
for (let i = 0; i < 5; i++) {
  if (i === 3) break; // Interrompe il ciclo quando i è 3
  console.log(i);
}
```

Sintesi dei due tipi di istruzioni

Tipo di Istruzione	Esempio
Dichiarazione	`let a = 10;` `function somma(a, b) { return a + b; }`
Controllo/Esecuzione	`if (a > b) { console.log("a è maggiore"); }` `for (let i=0; i<5; i++) { ... }`

In JavaScript, le **istruzioni** possono essere classificate in due forme base: **semplici** e **composte**. Questa distinzione si basa sulla complessità e sulla struttura dell'istruzione.

Istruzioni semplici

Le istruzioni semplici eseguono un'unica operazione o un'azione diretta. Sono auto-conclusive e generalmente terminano con un punto e virgola (`;`).

Esempi di istruzioni semplici:

a. Assegnazione

L'assegnazione di un valore a una variabile è una tipica istruzione semplice:

```
let x = 10; // Dichiarazione e assegnazione
x = x + 5;  // Aggiornamento del valore
```

b. Output

Utilizzo di una funzione per stampare un risultato o un messaggio:

```
console.log("Hello, World!"); // Stampa un messaggio nella console
```

c. Operazioni su dati

Effettuare calcoli o manipolazioni:

```
let somma = 5 + 10; // Esegue un calcolo e assegna il risultato
```

d. Chiamata a una funzione

Invocare una funzione per eseguire un'azione:

```
saluta(); // Chiama una funzione definita in precedenza
```

Istruzioni composte

Le istruzioni composte, o **blocchi di istruzioni**, contengono più istruzioni racchiuse tra parentesi graffe `{ }`. Sono utilizzate per raggruppare più operazioni da eseguire insieme, come nel caso di condizionali, cicli o funzioni.

Esempi di istruzioni composte:

a. Blocco generale

Un semplice blocco che raggruppa più istruzioni:

```
{
  let a = 10;
  let b = 20;
  console.log(a + b);
}
```

b. Condizionali

Un'istruzione `if` con un blocco di codice:

```
if (x > 0) {
  console.log("x è positivo");
  x = x - 1;
}
```

c. Cicli

Un ciclo che esegue un blocco di codice per ogni iterazione:

```
for (let i = 0; i < 5; i++) {
  console.log("Iterazione: " + i);
}
```

d. Funzioni

Un blocco di istruzioni racchiuso in una funzione:

```
function somma(a, b) {
  let risultato = a + b;
  return risultato;
}
```

Differenza principale

Istruzioni semplici	Istruzioni composte
Contengono un'unica operazione	Contengono più istruzioni
Terminano con un punto e virgola (`;`)	Racchiuse tra parentesi graffe `{ }`
Non hanno una struttura gerarchica	Possono contenere istruzioni semplici e altre composte

Esempio combinato

Ecco un esempio che utilizza entrambe le forme:

```
let x = 5;  // Istruzione semplice

if (x > 0) { // Inizio di un'istruzione composta
  console.log("x è positivo"); // Istruzione semplice all'interno di un blocco
  x--; // Istruzione semplice
}
```

Parole riservate in JavaScript

In JavaScript, le **parole riservate** sono termini che hanno un significato speciale all'interno del linguaggio. Non possono essere usate come identificatori (nomi di variabili, funzioni, classi, ecc.), perché sono già utilizzate o riservate dal linguaggio per scopi specifici.

Categorie principali di parole riservate

1. Parole chiave utilizzate attivamente

Queste parole sono attivamente utilizzate nel linguaggio per dichiarare variabili, strutture di controllo, funzioni, oggetti, e altro.

a. Dichiarazioni

- `let`, `const`, `var`: per dichiarare variabili.
- `function`, `class`: per dichiarare funzioni e classi.
- `import`, `export`: per i moduli JavaScript.

Esempio:

```
let nome = "Alice";
function saluta() {
  console.log("Ciao, " + nome);
}
```

b. Controllo di flusso

- `if`, `else`, `switch`, `case`, `default`: per le istruzioni condizionali.
- `for`, `while`, `do`, `break`, `continue`: per i cicli.

Esempio:

```
if (x > 0) {
  console.log("Positivo");
} else {
  console.log("Non positivo");
}
```

c. Gestione di errori

- `try`, `catch`, `finally`, `throw`: per gestire errori ed eccezioni.

Esempio:

```
try {
  let x = 10 / 0;
} catch (err) {
  console.log("Errore:", err.message);
}
```

d. Operatori logici e tipi

- `typeof`: per ottenere il tipo di una variabile.
- `instanceof`: per verificare se un oggetto appartiene a una classe.

Esempio:

```
console.log(typeof "ciao"); // "string"
```

e. Costrutti asincroni

- `async`, `await`: per gestire operazioni asincrone.
- `yield`: usato con generatori.

Esempio:

```
async function esempio() {
  let risultato = await fetch("https://api.example.com");
}
```

2. Parole riservate per futuro utilizzo

Alcune parole non hanno un ruolo attivo, ma sono riservate per possibili sviluppi futuri o compatibilità con altre versioni di JavaScript. Non dovrebbero essere utilizzate.

Esempi di parole riservate per il futuro:

- `enum`, `implements`, `interface`, `package`, `private`, `protected`, `public`, `static`.

Elenco completo delle parole riservate

Categoria	Parole riservate
Dichiarazioni	`let`, `const`, `var`, `function`, `class`, `import`, `export`
Controllo di flusso	`if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`
Errori	`try`, `catch`, `finally`, `throw`
Operatori e tipi	`typeof`, `instanceof`, `new`, `delete`, `void`
Moduli e oggetti	`this`, `super`, `extends`, `constructor`, `static`, `get`, `set`
Asincroni	`async`, `await`, `yield`
Parole future	`enum`, `implements`, `interface`, `package`, `private`, `protected`, `public`, `static`, `abstract`

Categoria	Parole riservate
Valori predefiniti	`true`, `false`, `null`, `undefined`, `NaN`, `Infinity`

Note importanti

1. **Case-sensitive**: JavaScript è sensibile alle maiuscole e minuscole, quindi `Let` o `Function` non sono parole riservate valide.
2. **Uso vietato**: Non puoi usare le parole riservate come nomi di variabili, funzioni o classi.

Esempio di errore:

```
let if = 10; // Errore: "if" è una parola riservata
```

Uno sguardo rapido alle funzioni in JavaScript

Le **funzioni** in JavaScript sono uno dei concetti fondamentali del linguaggio. Una funzione è un blocco di codice riutilizzabile che esegue un'operazione specifica. Può accettare **parametri** in ingresso, eseguire calcoli o operazioni e restituire un **valore**.

1. Dichiarare una funzione

Le funzioni possono essere dichiarate utilizzando la parola chiave `function`. La struttura di base è:

```
function nomeFunzione(parametro1, parametro2) {
  // Corpo della funzione
  return parametro1 + parametro2; // Valore restituito
}
```

Esempio:

```
function somma(a, b) {
  return a + b;
}
console.log(somma(5, 3)); // Output: 8
```

2. Tipi di funzioni

a. Funzioni dichiarate

Le funzioni dichiarate sono definite con la parola chiave `function` e possono essere richiamate ovunque nel codice (grazie all'**hoisting**).

Esempio:

```
function saluta() {
  console.log("Ciao!");
}
saluta(); // Output: "Ciao!"
```

b. Funzioni anonime

Funzioni senza nome, spesso assegnate a una variabile o passate come argomento.

Esempio:

```
const saluta = function() {
  console.log("Ciao!");
};
saluta(); // Output: "Ciao!"
```

c. Funzioni freccia (Arrow Functions)

Introduzione con ES6, sono una sintassi concisa per definire funzioni.

Esempio:

```
const moltiplica = (a, b) => a * b;
```

```
console.log(moltiplica(2, 3)); // Output: 6
```

Se la funzione ha un solo parametro, le parentesi possono essere omesse:

```
const quadrato = x => x * x;
console.log(quadrato(4)); // Output: 16
```

d. Funzioni come oggetti

In JavaScript, le funzioni sono **oggetti di prima classe**:

- Possono essere assegnate a variabili.
- Passate come argomenti.
- Restituite da altre funzioni.

Esempio:

```
function generaFunzione() {
  return function(nome) {
    return "Ciao, " + nome;
  };
}
const saluta = generaFunzione();
console.log(saluta("Alice")); // Output: "Ciao, Alice"
```

3. Parametri e valori di ritorno

a. Parametri predefiniti

Puoi fornire valori predefiniti ai parametri, usati se non vengono specificati.

Esempio:

```
function saluto(nome = "Amico") {
  return "Ciao, " + nome;
}
console.log(saluto()); // Output: "Ciao, Amico"
```

b. Restituire valori

Le funzioni possono restituire valori utilizzando `return`.

Esempio:

```
function calcolaArea(base, altezza) {
  return base * altezza;
}
let area = calcolaArea(5, 10);
console.log(area); // Output: 50
```

4. Funzioni con numero variabile di argomenti

a. Oggetto `arguments`

L'oggetto `arguments` contiene tutti gli argomenti passati a una funzione, anche se non specificati come parametri.

Esempio:

```
function sommaTutti() {
  let somma = 0;
  for (let i = 0; i < arguments.length; i++) {
    somma += arguments[i];
  }
  return somma;
}
console.log(sommaTutti(1, 2, 3, 4)); // Output: 10
```

b. Operatore rest (`...`)

Con ES6, l'operatore `...` raccoglie tutti gli argomenti in un array.

Esempio:

```
function somma(...numeri) {
  return numeri.reduce((acc, numero) => acc + numero, 0);
}
console.log(somma(1, 2, 3, 4)); // Output: 10
```

5. Scope delle variabili

Le variabili definite all'interno di una funzione sono locali e non accessibili dall'esterno.

Esempio:

```
function esempio() {
  let locale = "Sono locale";
}
console.log(locale); // Errore: "locale" non è definita
```

Le funzioni possono accedere alle variabili definite all'esterno grazie al **closure**.

6. Funzioni asincrone

Le funzioni possono essere dichiarate come `async` per lavorare con operazioni asincrone.

Esempio:

```
async function esempioAsincrono() {
  let risultato = await fetch("https://api.example.com");
  console.log(risultato);
}
```

Funzioni come oggetti di prima classe in JavaScript

In JavaScript, le funzioni sono considerate **oggetti di prima classe** (**first-class objects**). Questo significa che:

1. Possono essere trattate come qualsiasi altro valore.
2. Possono essere assegnate a variabili, passate come argomenti ad altre funzioni, o restituite da funzioni.

Questa caratteristica rende JavaScript un linguaggio estremamente flessibile e potente, permettendo la programmazione funzionale e il trattamento delle funzioni come dati.

Caratteristiche delle funzioni come oggetti di prima classe

1. Le funzioni possono essere assegnate a variabili

Come qualsiasi altro valore, una funzione può essere assegnata a una variabile.

Esempio:

```
const saluta = function(nome) {
  return "Ciao, " + nome;
};
console.log(saluta("Alice")); // Output: "Ciao, Alice"
```

2. Le funzioni possono essere passate come argomenti

Puoi passare una funzione come argomento a un'altra funzione.

Esempio:

```
function esegui(funzione, valore) {
  return funzione(valore);
}

const quadrato = function(x) {
  return x * x;
};

console.log(esegui(quadrato, 5)); // Output: 25
```

3. Le funzioni possono essere restituite da altre funzioni

Le funzioni possono generare e restituire altre funzioni.

Esempio:

```
function creaMoltiplicatore(moltiplicatore) {
  return function(numero) {
    return numero * moltiplicatore;
  };
}

const raddoppia = creaMoltiplicatore(2);
console.log(raddoppia(5)); // Output: 10
```

```
const triplica = creaMoltiplicatore(3);
console.log(triplica(5)); // Output: 15
```

4. Le funzioni possono avere proprietà

Essendo oggetti, le funzioni possono avere proprietà aggiunte dinamicamente.

Esempio:

```
function saluta() {
  console.log("Ciao!");
}

saluta.messaggio = "Questo è un messaggio speciale";
console.log(saluta.messaggio); // Output: "Questo è un messaggio speciale"
```

5. Le funzioni possono essere anonime

In JavaScript, le funzioni possono essere create senza un nome e utilizzate immediatamente o assegnate a variabili.

Esempio:

```
const somma = function(a, b) {
  return a + b;
};

console.log(somma(3, 4)); // Output: 7
```

Esempio avanzato: callback e higher-order functions

Una funzione che accetta o restituisce altre funzioni è chiamata una **funzione di ordine superiore** (higher-order function). Questo approccio è comune nella programmazione funzionale.

Callback

Un callback è una funzione passata come argomento e chiamata all'interno della funzione.

Esempio:

```
function processo(numero, callback) {
  return callback(numero);
}

const raddoppia = (n) => n * 2;
console.log(processo(5, raddoppia)); // Output: 10
```

Higher-order functions

Funzioni come `map`, `filter`, e `reduce` sono esempi di funzioni di ordine superiore che lavorano con altre funzioni.

Esempio:

```
const numeri = [1, 2, 3, 4];
const doppi = numeri.map((n) => n * 2); // Applica la funzione a ogni elemento
console.log(doppi); // Output: [2, 4, 6, 8]
```

Conclusione

Le funzioni come oggetti di prima classe sono una caratteristica potente che permette di scrivere codice:

- **Modulare**: Funzioni riutilizzabili e combinabili.
- **Espressivo**: Lavorare con funzioni come dati.
- **Funzionale**: Adottare uno stile di programmazione basato su composizione e trasformazione.

Lavorare con variabili e dati in JavaScript

In JavaScript, i dati possono essere rappresentati in vari tipi di formati. Esaminiamo i principali **tipi di dati** e le funzionalità associate.

Numeri

I numeri in JavaScript comprendono sia numeri interi che numeri decimali.

Esempi:

```
let numeroIntero = 10;
let numeroDecimale = 3.14;
```

Funzioni numeriche

JavaScript offre diverse funzioni per manipolare i numeri:

- `Number.parseInt()` - Converte una stringa in un numero intero.
- `Number.parseFloat()` - Converte una stringa in un numero decimale.
- `Number.isFinite()` - Controlla se il numero è finito.
- `Number.isInteger()` - Controlla se il valore è un intero.

Esempio:

```
let numero = "42";
console.log(Number.parseInt(numero)); // Output: 42
console.log(Number.isInteger(42));    // Output: true
```

Testare la funzione `isNaN()`

`isNaN()` verifica se un valore è **"Not a Number" (NaN)**.

Esempio:

```
console.log(isNaN("ciao")); // Output: true
console.log(isNaN(42));     // Output: false
```

Costanti numeriche

Alcune costanti numeriche disponibili in JavaScript includono:

- `Number.MAX_VALUE`: Il valore numerico massimo rappresentabile.
- `Number.MIN_VALUE`: Il valore numerico minimo rappresentabile.
- `Number.POSITIVE_INFINITY`: Rappresenta infinito positivo.
- `Number.NEGATIVE_INFINITY`: Rappresenta infinito negativo.
- `Number.NaN`: Rappresenta "Not a Number".

Esempio:

```
console.log(Number.MAX_VALUE); // Output: Valore massimo rappresentabile
console.log(Number.NaN);       // Output: NaN
```

Oggetto Math

L'oggetto `Math` fornisce una serie di funzioni matematiche.

Funzioni comuni:

- `Math.round(x)`: Arrotonda al numero intero più vicino.
- `Math.ceil(x)`: Arrotonda per eccesso.
- `Math.floor(x)`: Arrotonda per difetto.
- `Math.random()`: Genera un numero casuale tra 0 e 1.
- `Math.sqrt(x)`: Calcola la radice quadrata.

Esempio:

```
console.log(Math.round(3.7));   // Output: 4
console.log(Math.random());    // Output: Un numero casuale
```

Stringhe

Le stringhe rappresentano sequenze di caratteri e possono essere delimitate da virgolette singole (`'`), doppie (`"`) o backtick (``).

Esempio:

```
let saluto = "Ciao, mondo!";
let frase = 'JavaScript è fantastico';
```

Escape delle virgolette

Per includere virgolette all'interno di una stringa, utilizziamo il carattere di escape (`\`).

Esempio:

```
let frase = "Si dice \"JavaScript\"!";
console.log(frase); // Output: Si dice "JavaScript"!
```

Metodi e proprietà delle stringhe

Le stringhe in JavaScript dispongono di molti metodi utili:

- `length`: Restituisce la lunghezza della stringa.
- `toUpperCase()`, `toLowerCase()`: Converte la stringa in maiuscolo o minuscolo.
- `indexOf()`: Trova la posizione di una sottostringa.
- `slice()`: Estrae una parte della stringa.
- `replace()`: Sostituisce una sottostringa con un'altra.

Esempio:

```
let saluto = "Ciao, JavaScript!";
console.log(saluto.length);            // Output: 18
console.log(saluto.toUpperCase());     // Output: CIAO, JAVASCRIPT!
console.log(saluto.indexOf("Java"));   // Output: 6
```

```
console.log(saluto.slice(6, 16));      // Output: JavaScript
```

Boolean

I valori booleani possono essere `true` o `false` e sono spesso utilizzati per il controllo di flusso.

Esempio:

```
let maggiore = 10 > 5;
console.log(maggiore); // Output: true
```

Null

`null` rappresenta l'assenza intenzionale di un valore.

Esempio:

```
let vuoto = null;
console.log(vuoto); // Output: null
```

Undefined

`undefined` indica che una variabile è stata dichiarata ma non ha ancora un valore assegnato.

Esempio:

```
let variabileNonDefinita;
console.log(variabileNonDefinita); // Output: undefined
```

Oggetti

Gli oggetti sono strutture dati che rappresentano insiemi di coppie **chiave-valore**.

Esempio:

```
let persona = {
  nome: "Alice",
  eta: 25,
  saluta: function() {
    return "Ciao, sono " + this.nome;
  }
};

console.log(persona.nome);        // Output: Alice
console.log(persona.saluta());    // Output: Ciao, sono Alice
```

Matrici (Array)

Le matrici sono contenitori per memorizzare insiemi ordinati di valori.

Esempio:

```
let numeri = [1, 2, 3, 4];
console.log(numeri[2]); // Output: 3
```

Metodi utili per gli array:

- `push()`: Aggiunge un elemento alla fine.
- `pop()`: Rimuove l'ultimo elemento.
- `shift()`: Rimuove il primo elemento.
- `unshift()`: Aggiunge un elemento all'inizio.

Esempio:

```
numeri.push(5);
console.log(numeri); // Output: [1, 2, 3, 4, 5]
```

Definire e utilizzare variabili in JavaScript

Le variabili in JavaScript vengono utilizzate per memorizzare e manipolare valori. Definire e utilizzare correttamente le variabili è uno degli aspetti fondamentali della programmazione.

Dichiarare variabili

In JavaScript, le variabili possono essere dichiarate usando le parole chiave `var`, `let`, e `const`.

a. `var` (Obsoleto ma ancora usato)

La parola chiave `var` è stata la modalità principale per dichiarare variabili nelle versioni precedenti di JavaScript. Le variabili dichiarate con `var` hanno **scope** globale o di funzione e sono soggette all'**hoisting** (cioè vengono "sollevate" all'inizio del loro scope).

Esempio:

```
var x = 10;
console.log(x); // Output: 10
```

Tuttavia, `var` è stato sostituito in gran parte da `let` e `const` per motivi di scoping e chiarezza.

b. `let` (Preferito per variabili modificate)

`let` è la parola chiave moderna per dichiarare variabili che potrebbero essere **modificate** durante l'esecuzione del programma. Le variabili dichiarate con `let` hanno **scope** di blocco (cioè sono visibili solo all'interno del blocco di codice in cui sono state dichiarate).

Esempio:

```
let numero = 5;
numero = 10; // La variabile può essere modificata
console.log(numero); // Output: 10
```

c. `const` (Preferito per variabili costanti)

`const` viene usato per dichiarare variabili il cui **valore non può essere modificato** dopo l'assegnazione iniziale. Anche le variabili dichiarate con `const` hanno **scope** di blocco.

Esempio:

```
const pi = 3.14159;
console.log(pi); // Output: 3.14159
// pi = 3.14; // Errore: "pi" è una costante e non può essere modificata
```

Tipi di variabili

Le variabili in JavaScript possono contenere diversi tipi di dati. Ecco i principali:

a. Numeri

I numeri in JavaScript possono essere interi o decimali (floating point).

Esempio:

```
let intero = 10;
let decimale = 3.14;
```

b. Stringhe

Le stringhe rappresentano sequenze di caratteri e vengono definite con virgolette singole (`'`) o doppie (`"`).

Esempio:

```
let saluto = "Ciao, mondo!";
let nome = 'Alice';
```

c. Booleani

Le variabili booleane possono assumere solo due valori: `true` o `false`.

Esempio:

```
let isActive = true;
let isCompleted = false;
```

d. Oggetti

Gli oggetti sono collezioni di coppie chiave-valore.

Esempio:

```
let persona = {
  nome: "Alice",
  eta: 25
};
```

e. Array (Matrici)

Le matrici sono elenchi ordinati di valori.

Esempio:

```
let numeri = [1, 2, 3, 4, 5];
```

f. `null` e `undefined`

- `null` rappresenta un valore vuoto o "intenzionalmente assente".
- `undefined` indica che una variabile è stata dichiarata ma non è stata ancora inizializzata con un valore.

Esempio:

```
let valoreNull = null;
let valoreUndefined;
```

3. Validità delle variabili

La **validità di una variabile** si riferisce al contesto in cui la variabile è visibile e può essere utilizzata. La validità è determinata dal **scope** (ambito) e dal **lifetime** (durata della vita) della variabile.

a. Scope di variabili

- **Scope globale**: Se una variabile è dichiarata all'esterno di qualsiasi funzione o blocco, è visibile in tutto il programma.
- **Scope di funzione**: Se una variabile è dichiarata all'interno di una funzione, è visibile solo all'interno di quella funzione.
- **Scope di blocco**: Variabili dichiarate con `let` o `const` sono visibili solo all'interno del blocco di codice in cui sono dichiarate (come un ciclo o una struttura condizionale).

Esempio di scope globale:

```
let globale = "sono globale";
function mostra() {
  console.log(globale); // Accesso alla variabile globale
}
mostra(); // Output: "sono globale"
```

Esempio di scope di funzione:

```
function esempio() {
  let locale = "sono locale";
  console.log(locale); // Accesso alla variabile locale
}
esempio(); // Output: "sono locale"
// console.log(locale); // Errore: "locale" non è definita fuori dalla funzione
```

Esempio di scope di blocco:

```
if (true) {
  let dentroBlocco = "sono dentro il blocco";
  console.log(dentroBlocco); // Output: "sono dentro il blocco"
}
// console.log(dentroBlocco); // Errore: "dentroBlocco" non è definita fuori dal
blocco
```

b. Hoisting (Sollevamento)

L'**hoisting** è un comportamento in JavaScript dove le dichiarazioni di variabili (dichiarate con `var`) vengono "sollevate" all'inizio del loro contesto (funzione o globale). Tuttavia, solo la dichiarazione viene sollevata, non l'inizializzazione.

Esempio con `var`:

```
console.log(a); // Output: undefined
var a = 5;
console.log(a); // Output: 5
```

Con `let` e `const`, l'**hoisting** non funziona come con `var`, e le variabili non sono accessibili prima della dichiarazione.

Esempio con `let` e `const`:

```
// console.log(b); // Errore: Cannot access 'b' before initialization
let b = 10;
```

Garbage Collection in JavaScript

La **garbage collection** (raccolta dei rifiuti) è un processo automatico che gestisce la memoria in JavaScript, liberando la memoria inutilizzata da variabili, oggetti e risorse che non sono più raggiungibili o utilizzabili dal programma. Questo permette al programmatore di concentrarsi sulla logica del programma senza doversi preoccupare esplicitamente di gestire la memoria.

Come funziona la Garbage Collection in JavaScript

JavaScript, essendo un linguaggio con **gestione automatica della memoria**, non richiede che l'utente allocchi o rilasci manualmente la memoria. La **garbage collection** si occupa di identificare e liberare la memoria associata a variabili e oggetti che non sono più necessari.

Concetti principali

1. Oggetti non più raggiungibili

Quando un oggetto o una variabile non è più raggiungibile (ad esempio, se non è più referenziato da nessuna parte del codice), il garbage collector può liberare la memoria associata ad esso. Gli oggetti diventano "non raggiungibili" quando non ci sono più riferimenti ad essi.

Esempio:

```
let persona = {
  nome: "Alice",
  eta: 30
};

// Successivamente...
persona = null; // L'oggetto precedente è ora non raggiungibile
```

In questo caso, l'oggetto `persona` è ora "non raggiungibile" e, quando il garbage collector eseguirà il suo ciclo, la memoria associata a quell'oggetto verrà liberata.

2. Ciclo di Garbage Collection

JavaScript utilizza un meccanismo di **garbage collection basato su "tracciamento delle radici"** (mark-and-sweep). Questo processo si svolge in due fasi:

- **Fase di marcatura (marking)**: Il garbage collector attraversa tutte le variabili e gli oggetti accessibili (denominati "radici"), come variabili globali e variabili locali attive nelle funzioni, e marca gli oggetti ancora "vivi" (cioè quelli che possono essere raggiunti).
- **Fase di pulizia (sweeping)**: Una volta marcati gli oggetti viventi, il garbage collector elimina gli oggetti che non sono stati marcati, liberando la memoria da risorse non più necessarie.

3. Algoritmo di Mark-and-Sweep

Il più comune algoritmo di garbage collection è il **Mark-and-Sweep**:

1. Il **garbage collector** inizia da una serie di **oggetti di partenza** (come variabili globali e oggetti locali).

2. Segna (mark) tutti gli oggetti che possono essere raggiunti partendo da queste radici.
3. Gli oggetti che non sono stati segnati come raggiungibili vengono poi **rimossi** (swept) e la memoria che occupano viene liberata.

Esempio:

```
function creaOggetti() {
  let oggetto1 = { id: 1 };
  let oggetto2 = { id: 2 };

  // Oggetti oggetto1 e oggetto2 sono creati qui
}

creaOggetti(); // Oggetti oggetto1 e oggetto2 non sono più accessibili fuori da questa funzione
// Dopo che la funzione è terminata, questi oggetti diventano non raggiungibili e il GC può liberarli.
```

4. Referenze Circolari

Un aspetto particolare che il garbage collector può gestire è la **referenza circolare**. Due o più oggetti possono fare riferimento l'uno all'altro creando un ciclo. Sebbene questi oggetti siano raggiungibili tra di loro, se non sono accessibili da alcuna parte esterna al ciclo, JavaScript sarà comunque in grado di rilevare che non sono più utilizzati e rimuoverli dalla memoria.

Esempio di referenza circolare:

```
let oggettoA = {};
let oggettoB = {};

// Crea una referenza circolare
oggettoA.b = oggettoB;
oggettoB.a = oggettoA;

// Se non ci sono altre referenze a oggettoA o oggettoB fuori dal ciclo,
// il garbage collector li rimuoverà quando diventano non più raggiungibili.
oggettoA = null;
oggettoB = null;
```

Gestione della memoria e ottimizzazione

Anche se il garbage collection è automatico, è importante ricordare che un uso inefficiente della memoria può causare dei **problemi di performance**, come la **fragmentazione della memoria** o **pause nella raccolta dei rifiuti**. Ecco alcuni suggerimenti per gestire meglio la memoria:

1. **Liberare esplicitamente oggetti inutilizzati**: Anche se il garbage collector si occupa automaticamente della pulizia, è buona pratica impostare a `null` o `undefined` le variabili che non sono più necessarie, specialmente per oggetti grandi o complessi.

   ```
   let oggettoGrande = { /* dati */ };
   oggettoGrande = null; // La memoria associata può essere liberata
   ```

2. **Limitare le referenze inutili**: Se un oggetto è temporaneo o non necessario, cerca di ridurre il numero di riferimenti a esso. Più riferimenti ci sono a un oggetto, più tempo ci vorrà per rimuoverlo dal heap.
3. **Evita i cicli di oggetti inutili**: Anche se JavaScript gestisce le referenze circolari, è meglio evitare cicli complessi che possono aumentare il carico sulla garbage collection.

Conclusione

La **garbage collection** in JavaScript è una parte fondamentale per la gestione automatica della memoria. Il processo di **mark-and-sweep** permette al motore JavaScript di liberare la memoria non più utilizzata, migliorando l'efficienza e prevenendo i **memory leaks** (perdite di memoria). Tuttavia, anche se automatico, è importante progettare il codice in modo che la memoria venga gestita in modo ottimale, evitando riferimenti inutili e migliorando le prestazioni generali dell'applicazione.

Conversione dei tipi in JavaScript

In JavaScript, la conversione tra diversi tipi di dati (numeri, stringhe, booleani) è un'operazione comune e può essere eseguita in vari modi. Esistono principalmente due tipi di conversione: **implicita** (o automatica) e **esplicita** (o forzata).

1. Conversione di numeri

a. Conversione Implicita a Numero

JavaScript esegue la conversione automatica a numero in determinate situazioni, come ad esempio quando un'operazione richiede un numero, ma viene passato un altro tipo (ad esempio una stringa).

Esempio di conversione implicita a numero:

```
let str = "123";
let num = str * 1;  // La stringa viene convertita implicitamente in numero
console.log(num);    // Output: 123 (come numero)
```

Un altro esempio di conversione implicita si verifica quando si somma un numero con una stringa, ma in tal caso si ottiene una concatenazione (invece di somma numerica) perché JavaScript converte il numero in stringa:

```
let numero = 5;
let testo = "5";
let risultato = numero + testo;  // Concatenazione
console.log(risultato);  // Output: "55"
```

b. Conversione Esplicita a Numero

Per effettuare una conversione esplicita da una stringa o un altro tipo a numero, possiamo usare diverse funzioni:

- **`Number()`**: Converte un valore in un numero.
- **`parseInt()`**: Converte una stringa in un numero intero.
- **`parseFloat()`**: Converte una stringa in un numero decimale.

Esempio con `Number()`:

```
let valore = "42";
let numero = Number(valore);
console.log(numero);  // Output: 42 (numero)
```

Esempio con `parseInt()` e `parseFloat()`:

```
let intero = parseInt("42px");
console.log(intero);  // Output: 42 (solo la parte numerica)

let decimale = parseFloat("3.14abc");
console.log(decimale);  // Output: 3.14 (solo la parte numerica)
```

Trattamento degli errori: Se la stringa non può essere convertita in un numero valido, la funzione `Number()` restituirà `NaN` (Not a Number). Ad esempio:

```
let invalido = Number("ciao");
console.log(invalido);  // Output: NaN
```

2. Conversione di stringhe

a. Conversione Implicita a Stringa

JavaScript esegue la conversione implicita di un valore in stringa in diverse situazioni, come ad esempio quando un numero viene concatenato a una stringa.

Esempio di conversione implicita a stringa:

```
let numero = 10;
let risultato = "Il numero è " + numero;  // Implicitamente il numero viene
convertito in stringa
console.log(risultato);  // Output: "Il numero è 10"
```

b. Conversione Esplicita a Stringa

Per forzare una conversione di un valore in stringa, si possono usare i seguenti metodi:

- **`String()`**: Converte qualsiasi tipo di valore in stringa.
- **`toString()`**: Metodo di oggetti (compresi numeri, booleani, array, etc.) per ottenere una rappresentazione in stringa.

Esempio con `String()`:

```
let numero = 123;
let stringa = String(numero);
console.log(stringa);  // Output: "123" (come stringa)
```

Esempio con `toString()`:

```
let array = [1, 2, 3];
let stringaArray = array.toString();
console.log(stringaArray);  // Output: "1,2,3" (array convertito in stringa)
```

Nel caso di un valore `null` o `undefined`, la conversione in stringa restituirà rispettivamente "null" e "undefined":

```
let nullo = null;
console.log(String(nullo));  // Output: "null"

let indefinito;
console.log(String(indefinito));  // Output: "undefined"
```

3. Conversione di booleani

a. Conversione Implicita a Booleano

JavaScript converte automaticamente un valore in booleano in situazioni in cui è richiesto un valore di tipo booleano, come nelle espressioni condizionali. In JavaScript, i seguenti valori sono considerati "falsy" (falsi) quando convertiti in booleano:

- `false`
- `0`
- `""` (stringa vuota)
- `null`
- `undefined`
- `NaN`

Tutti gli altri valori sono considerati "truthy" (veri).

Esempio di conversione implicita a boolean:

```
let valore = 0;
if (valore) {
  console.log("Valore vero");
} else {
  console.log("Valore falso");  // Output: "Valore falso"
}
```

b. Conversione Esplicita a Boolean

Per forzare una conversione esplicita a boolean, si può usare il costruttore `Boolean()`:

Esempio con `Boolean()`:

```
let valore1 = 0;
let booleano1 = Boolean(valore1);
console.log(booleano1);  // Output: false

let valore2 = "ciao";
let booleano2 = Boolean(valore2);
console.log(booleano2);  // Output: true
```

Questa funzione converte i valori "falsy" in `false` e tutti gli altri valori in `true`.

Sintesi dei metodi di conversione

- **Conversione a numero**:
  - `Number()` – Converte a numero.
  - `parseInt()` – Converte a intero (parsa una stringa).
  - `parseFloat()` – Converte a numero decimale.
- **Conversione a stringa**:
  - `String()` – Converte qualsiasi tipo in stringa.
  - `toString()` – Converte a stringa (metodo su oggetti).
- **Conversione a booleano**:
  - `Boolean()` – Converte a booleano.