

Funzioni in javascript

Funzioni in JavaScript

1. Code Reusability e Funzioni

Le funzioni sono un concetto fondamentale della programmazione perché permettono di riutilizzare il codice, evitando di scrivere le stesse istruzioni più volte.

Dichiarazione di una funzione

Una funzione si dichiara con la parola chiave `function`, seguita dal nome della funzione e le parentesi tonde `()` che possono contenere dei parametri. Il corpo della funzione è racchiuso tra `{}`.

Esempio di funzione senza parametri:

```
function sayMyName() {  
    console.log("Hi Bob!");  
}  
sayMyName(); // Output: Hi Bob!
```

La funzione viene definita e poi chiamata con il suo nome.

Parametri e Argomenti

Le funzioni possono accettare parametri, che sono variabili definite tra le parentesi tonde `()`. Quando la funzione viene chiamata, si passano i valori reali (argomenti).

Esempio di funzione con parametri:

```
function sayMyName(name) {  
    console.log("Hi, " + name);  
}  
  
sayMyName("James"); // Output: Hi, James  
sayMyName("Adam");  // Output: Hi, Adam
```

Funzioni con valore restituito (`return`)

Alcune funzioni eseguono calcoli e restituiscono un valore con la parola chiave `return`.

Esempio di funzione con `return`:

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

Il valore restituito dalla funzione può essere usato in altre operazioni.

2. Concetti Avanzati sulle Funzioni

Dipendenze Circolari

Quando due funzioni si chiamano a vicenda in modo infinito, si crea una dipendenza circolare, che porta a un loop infinito e può bloccare il programma.

Esempio di dipendenza circolare (da evitare!):

```
function chicken() {  
  egg();  
}  
function egg() {  
  chicken();  
}  
egg(); // Chiamata infinita tra chicken() ed egg()
```

Ricorsione

La ricorsione è una tecnica in cui una funzione chiama sé stessa fino a raggiungere una condizione di uscita.

Esempio: Calcolo della serie di Fibonacci

```
function fibonacci(n) {  
  if (n < 2) {  
    return n;  
  }  
}
```

```
    return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
console.log(fibonacci(5)); // Output: 5
```

⚠ **Attenzione:** la ricorsione può essere molto lenta per numeri elevati!

3. Scope delle Variabili (Visibilità delle Variabili)

Lo **scope** determina dove una variabile è accessibile nel codice.

Scope Locale

Una variabile dichiarata dentro una funzione è accessibile solo all'interno della funzione.

Esempio:

```
function addNumbers(num1, num2) {  
    let localResult = num1 + num2;  
    console.log("Il risultato locale è: " + localResult);  
}  
addNumbers(5, 7);  
// console.log(localResult); // ERRORE: localResult non è visibile fuori dalla  
funzione
```

Scope Globale

Una variabile dichiarata fuori da qualsiasi funzione è accessibile ovunque nel codice.

Esempio:

```
let globalResult;  
  
function addNumbers(num1, num2) {  
    globalResult = num1 + num2;  
}  
addNumbers(5, 7);  
console.log(globalResult); // Output: 12
```

⚠ **Attenzione:** le variabili globali possono causare conflitti nel codice se non gestite correttamente!

4. Convenzioni di Codifica (Best Practices)

Per scrivere codice leggibile e manutenibile, seguiamo alcune regole:

Indentazione e Formattazione

Esempi di codice sbagliato e corretto:

✗ Codice difficile da leggere

```
function addNumbers(num1,num2) {return num1 + num2;}
```

✓ Codice leggibile

```
function addNumbers(num1, num2) {  
    return num1 + num2;  
}
```

Commenti e Documentazione

I commenti spiegano cosa fa il codice.

✗ Commento poco utile

```
// Somma due numeri  
function addNumbers(num1, num2) {  
    return num1 + num2;  
}
```

✓ Commento dettagliato con JSDoc

```
/**  
 * Somma due numeri e restituisce il risultato  
 * @param {number} num1 - Primo numero  
 * @param {number} num2 - Secondo numero  
 * @returns {number} - Somma dei due numeri  
 */  
function addNumbers(num1, num2) {  
    return num1 + num2;  
}
```

5. Differenza tra `let` e `var`

JavaScript ha due modi principali per dichiarare variabili:

- `let` (preferito nel codice moderno)
- `var` (usato nel vecchio codice)

`let` : Block Scope

Le variabili dichiarate con `let` sono visibili solo dentro il blocco `{}` in cui sono dichiarate.

Esempio:

```
function worker() {  
  let x = 88;  
  for (let i = 0; i < 4; i++) {  
    console.log('i block =', i); // OK  
  }  
  console.log('x func =', x); // OK  
  // console.log('i !block =', i); // ERRORE: i non è definito fuori dal  
  blocco  
}  
worker();
```

`var` : Function Scope


Le variabili dichiarate con `var` sono visibili nell'intera funzione, anche fuori dai blocchi `{}`.

Esempio:

```
function worker() {  
  var x = 88;  
  for (var i = 0; i < 4; i++) {  
    console.log('i block =', i);  
  }  
  console.log('i fuori dal blocco =', i); // OK  
}  
worker();
```

⚠ **Problema di `var`** : può causare bug perché è visibile anche fuori dai blocchi.

✓ **Conclusione:**

-  `let` è più sicuro e prevedibile, quindi è la scelta consigliata per dichiarare variabili.

Approfondimento sulle Funzioni Avanzate e Arrow Functions in JavaScript

Le funzioni in JavaScript possono essere scritte in modi diversi per migliorare leggibilità, prestazioni e manutenibilità. Qui approfondiamo **funzioni avanzate**, tra cui **funzioni di prima classe**, **funzioni di ordine superiore**, **chiusure (closures)** e **Arrow Functions**.

1. Funzioni di Prima Classe (First-Class Functions)

In JavaScript, le funzioni sono **oggetti di prima classe**, il che significa che possono essere:

- ✓ **Assegnate a variabili**
- ✓ **Passate come argomenti ad altre funzioni**
- ✓ **Restituite da altre funzioni**

Assegnare una funzione a una variabile

```
const greet = function(name) {  
    return "Hello, " + name;  
};  
console.log(greet("Alice")); // Output: Hello, Alice
```

2. Funzioni di Ordine Superiore (Higher-Order Functions)

Le funzioni che accettano **altre funzioni come parametri** o **restituiscono funzioni** sono chiamate **funzioni di ordine superiore**.

Esempio: Funzione che accetta un'altra funzione

```
function applyOperation(a, b, operation) {  
    return operation(a, b);  
}
```

```
}

function add(x, y) {
  return x + y;
}

function multiply(x, y) {
  return x * y;
}

console.log(applyOperation(5, 3, add)); // Output: 8
console.log(applyOperation(5, 3, multiply)); // Output: 15
```

Esempio: Funzione che restituisce un'altra funzione

```
function createMultiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = createMultiplier(2);
console.log(double(5)); // Output: 10
```

3. Closures (Chiusure)

Una **closure** è una funzione che **ricorda il contesto in cui è stata creata**, anche dopo che la funzione esterna ha terminato la sua esecuzione.

Esempio di Closure

```
function createCounter() {
  let count = 0; // Variabile privata

  return function() {
    count++;
    return count;
  };
}

const counter = createCounter();
```

```
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

4. Arrow Functions (=>)

Le **Arrow Functions** sono una sintassi più compatta per definire funzioni in JavaScript.

Sintassi Base

```
const add = (a, b) => a + b;
console.log(add(3, 4)); // Output: 7
```

5. Differenze tra Funzioni Normali e Arrow Functions

A. Arrow Function senza `return` (Implicito)

```
const square = x => x * x;
console.log(square(5)); // Output: 25
```

B. Arrow Function con più istruzioni (Serve il `return`)

```
const sum = (a, b) => {
  console.log("Adding:", a, "+", b);
  return a + b;
};
console.log(sum(2, 3)); // Output: Adding: 2 + 3 \n 5
```

C. Arrow Function senza Parametri

```
const greet = () => "Hello, World!";
console.log(greet()); // Output: Hello, World!
```

D. Arrow Function con un Solo Parametro


```
const double = x => x * 2;
console.log(double(4)); // Output: 8
```

6. Arrow Functions e `this`

Le Arrow Functions NON hanno il proprio `this`

Con una funzione normale, il valore di `this` dipende da come la funzione viene chiamata.
Con un'Arrow Function, `this` viene preso dal **contesto in cui la funzione è stata definita**.

Esempio di Differenza

Funzione Normale (Il valore di `this` dipende dal contesto)

```
const person = {
  name: "Alice",
  greet: function() {
    console.log("Hi, I'm " + this.name);
  }
};

person.greet(); // Output: Hi, I'm Alice
```

Arrow Function (`this` NON cambia)

```
const person = {
  name: "Alice",
  greet: () => {
    console.log("Hi, I'm " + this.name);
  }
};

person.greet(); // Output: Hi, I'm undefined
```

👉 **Soluzione:** Usare una funzione normale o salvare `this` in una variabile.

```
const person = {
  name: "Alice",
  greet: function() {
    const self = this;
```

```
        setTimeout(function() {  
            console.log("Hi, I'm " + self.name);  
        }, 1000);  
    }  
};  
  
person.greet(); // Output dopo 1 secondo: Hi, I'm Alice
```

Conclusione

- ◆ **Funzioni avanzate** in JavaScript includono funzioni di ordine superiore, closure e funzioni di prima classe.
- ◆ **Arrow Functions (=>)** sono più concise ma hanno **differenze nel comportamento di `this`** rispetto alle funzioni normali.
- ◆ Le Arrow Functions sono perfette per **callback e funzioni brevi**, ma le funzioni normali sono più adatte quando abbiamo bisogno di gestire `this`.

🚀 **Usa le Arrow Functions quando possibile, ma scegli la sintassi giusta in base al contesto!**