Il **teorema di Böhm-Jacopini**

Il **teorema di Böhm-Jacopini** è un risultato fondamentale nella teoria della programmazione strutturata. Questo teorema afferma che ogni programma può essere realizzato utilizzando solo tre strutture di controllo:

1. **Sequenza**: istruzioni eseguite una dopo l'altra.
2. **Selezione**: una scelta condizionale tra due o più percorsi (usando, ad esempio, `if-else`).
3. **Iterazione**: ripetizione di un blocco di istruzioni (usando, ad esempio, `while` o `for`).

Secondo Böhm e Jacopini, qualsiasi programma (compreso uno con istruzioni di salto come `goto`) può essere riscritto usando solo queste tre strutture. Questo teorema ha gettato le basi per la programmazione strutturata, che favorisce programmi più leggibili, mantenibili e privi di salti non strutturati.

Sommario

Esempi specifici applicati al codice JavaScript.

1. Sequenza

Nella sequenza, le istruzioni vengono eseguite una dopo l'altra, nell'ordine in cui sono scritte.

Esempio in JavaScript:

```
let x = 5;
x = x + 2;
console.log(x);
```

In questo esempio, abbiamo tre istruzioni:

- `let x = 5;` (assegnazione)
- `x = x + 2;` (calcolo e riassegnazione)
- `console.log(x);` (stampa del valore)

Queste istruzioni vengono eseguite una dopo l'altra senza deviazioni di flusso.

2. Selezione (condizione `if-else`)

La selezione consente al programma di scegliere un percorso di esecuzione in base a una condizione. In JavaScript, le strutture `if`, `if-else`, e `switch` implementano la selezione.

Esempio di `if-else` in JavaScript:

```
let numero = 10;

if (numero > 5) {
    console.log("Il numero è maggiore di 5");
} else {
    console.log("Il numero è 5 o minore");
}
```

In questo caso, il programma controlla se `numero` è maggiore di 5:

- Se la condizione è vera, esegue il primo blocco (`console.log("Il numero è maggiore di 5")`).
- Altrimenti, esegue il secondo blocco (`console.log("Il numero è 5 o minore")`).

Questo tipo di selezione permette di definire un comportamento differente in base alla condizione.

3. Iterazione (loop con `while` o `for`)

L'iterazione consente al programma di ripetere un blocco di istruzioni finché una certa condizione rimane vera. In JavaScript, le iterazioni possono essere implementate con `for`, `while`, o `do...while`.

Esempio di ciclo `for` in JavaScript:

```
for (let i = 0; i < 5; i++) {
```

```
    console.log("Iterazione numero:", i);
}
```

Qui, il ciclo `for` ripete il blocco `console.log("Iterazione numero:", i);` cinque volte, con `i` che parte da `0` e arriva a `4`.

Esempio di ciclo `while` in JavaScript:

```
let i = 0;

while (i < 5) {
    console.log("Iterazione numero:", i);
    i++;
}
```

In questo caso, il ciclo `while` esegue lo stesso blocco di codice finché `i` è minore di `5`.

Applicazione del Teorema di Böhm-Jacopini: eliminare il `goto`

Immaginiamo un codice che utilizza un salto non strutturato (come il vecchio `goto` di linguaggi storici). JavaScript non ha `goto`, ma possiamo simulare una situazione simile usando `break` o `continue` impropriamente.

Codice con `goto` simulato (non strutturato):

```
for (let i = 0; i < 10; i++) {
    console.log("Inizio ciclo:", i);
    if (i === 5) {
        console.log("Salto a una nuova sezione!");
        break; // qui usiamo break per simulare un salto
    }
    console.log("Fine ciclo:", i);
}
```

In questo esempio, il `break` interrompe il ciclo non appena `i` è uguale a `5`, simulando un "salto" non strutturato. Con il teorema di Böhm-Jacopini, possiamo riscriverlo in una struttura senza `break`:

Riscrittura del codice secondo Böhm-Jacopini (strutturato):

```
let i = 0;
let esecuzione = true;

while (esecuzione && i < 10) {
    console.log("Inizio ciclo:", i);

    if (i === 5) {
        console.log("Salto a una nuova sezione!");
        esecuzione = false; // Fermiamo il ciclo impostando la condizione
    } else {
        console.log("Fine ciclo:", i);
    }

    i++;
}
```

In questo modo, abbiamo eliminato l'uso di `break`, creando una variabile (`esecuzione`) che controlla l'andamento del ciclo in modo strutturato.

Caso Pratico: Sommare Numeri Positivi Fino a Trovare Uno Zero

Vediamo un caso pratico in cui possiamo applicare il teorema di Böhm-Jacopini per riscrivere un codice non strutturato.

```
let somma = 0;

while (true) {
    let numero = prompt("Inserisci un numero (0 per fermare):");
    numero = parseInt(numero);

    if (numero === 0) break;

    if (numero > 0) {
        somma += numero;
    }
}

console.log("Somma dei numeri positivi:", somma);
```

In questo codice, `break` interrompe il ciclo non appena l'utente inserisce `0`. Vediamo come riscriverlo usando solo strutture di controllo.

Codice Strutturato senza `break`:

```
let somma = 0;
let numero = parseInt(prompt("Inserisci un numero (0 per fermare):"));

while (numero !== 0) {
    if (numero > 0) {
        somma += numero;
    }

    numero = parseInt(prompt("Inserisci un numero (0 per fermare):"));
}

console.log("Somma dei numeri positivi:", somma);
```

In questo caso, la condizione `numero !== 0` viene controllata all'inizio del ciclo `while`, quindi non abbiamo bisogno di `break`.

Conclusioni

Il teorema di Böhm-Jacopini ci insegna che possiamo eliminare le istruzioni non strutturate (come `goto` o `break`) utilizzando le tre strutture fondamentali. Questo porta a codice più chiaro e facile da mantenere, riducendo il rischio di errori.

La pseudocodifica del teorema di Böhm-Jacopini

La pseudocodifica del **teorema di Böhm-Jacopini** è utile per comprendere come strutturare un programma utilizzando esclusivamente le tre strutture di controllo fondamentali: sequenza, selezione e iterazione. Di seguito vediamo come queste tre strutture possono essere usate per costruire qualsiasi logica di programmazione.

Struttura generale che segue i principi del teorema.

Pseudocodifica del Teorema di Böhm-Jacopini

```
INIZIO

    // 1. SEQUENZA - Eseguiamo le istruzioni in ordine
    Dichiara variabili necessarie
    Inizializza le variabili
    Esegui calcoli preliminari

    // 2. SELEZIONE - Definisce scelte condizionali
    SE (condizione1) ALLORA
        // Blocco di codice da eseguire se condizione1 è vera
        Esegui operazioni specifiche per condizione1
    ALTRIMENTI SE (condizione2) ALLORA
        // Blocco di codice da eseguire se condizione2 è vera
        Esegui operazioni specifiche per condizione2
    ALTRIMENTI
        // Blocco di codice da eseguire se nessuna condizione precedente è vera
        Esegui operazioni alternative
    FINE SE

    // 3. ITERAZIONE - Ripete un blocco di codice fino a che una condizione è
soddisfatta
    MENTRE (condizioneIterativa)
        // Blocco di codice che si ripete finché condizioneIterativa è vera
        Esegui operazioni iterative
        Aggiorna variabili per evitare loop infiniti
    FINE MENTRE

FINE
```

Descrizione delle Componenti della Pseudocodifica

1. **Sequenza**:
   - Inizia con la dichiarazione e inizializzazione delle variabili.
   - Esegue i calcoli preliminari e altre istruzioni in modo sequenziale.
2. **Selezione (condizionale)**:
   - Utilizza la struttura `SE ... ALTRIMENTI SE ... ALTRIMENTI` per eseguire blocchi di codice in base alle condizioni.
   - Ogni blocco viene eseguito solo se la sua condizione è soddisfatta, scegliendo quindi un percorso di esecuzione specifico.
3. **Iterazione**:
   - Usa il ciclo `MENTRE (condizioneIterativa)` per ripetere un blocco di codice finché una condizione è vera.
   - Include un aggiornamento delle variabili per assicurarsi che il ciclo possa terminare

Esempio Pratico in Pseudocodifica

Immaginiamo di voler sommare i numeri positivi inseriti dall'utente fino a che l'utente non inserisce `0`.

```
INIZIO
    Dichiara variabile somma = 0
    Dichiara variabile numero

    // Ciclo di iterazione per raccogliere input dell'utente
    MENTRE (numero ≠ 0)
        Stampa "Inserisci un numero (0 per terminare):"
        numero = LeggiInputUtente()

        SE (numero > 0) ALLORA
            somma = somma + numero
        FINE SE
    FINE MENTRE

    Stampa "La somma dei numeri positivi è:" + somma
FINE
```

In questa pseudocodifica:

- **Sequenza**: le variabili sono dichiarate e inizializzate all'inizio.
- **Selezione**: il `SE (numero > 0) ALLORA` verifica se il numero è positivo prima di aggiungerlo alla somma.
- **Iterazione**: il ciclo `MENTRE (numero ≠ 0)` continua a raccogliere input fino a che l'utente inserisce `0`.

Questa organizzazione rende chiaro il flusso del programma e segue i principi del teorema di Böhm-Jacopini.

Per provare direttamente gli esempi di codice JavaScript, puoi usare diversi strumenti e ambienti accessibili. Di seguito, ti elenco alcuni metodi semplici e pratici:

1. Console del Browser

I browser moderni come **Chrome**, **Firefox**, **Edge** e **Safari** hanno una console JavaScript integrata, ideale per testare frammenti di codice.

Come fare:

- Apri il browser e vai su qualsiasi pagina web.
- Premi `F12` o `Ctrl+Shift+I` (su Windows) oppure `Cmd+Opt+I` (su Mac) per aprire gli strumenti per sviluppatori.
- Vai alla scheda **Console**.
- Incolla il codice JavaScript e premi **Invio** per eseguirlo.

Questo è un modo rapido per testare piccoli script o esempi di codice JavaScript.

2. Editor Online (CodePen, JSFiddle, o Repl.it)

Piattaforme come **CodePen**, **JSFiddle**, e **Repl.it** permettono di eseguire codice JavaScript direttamente online, con interfacce più complete rispetto alla console.

Esempi di piattaforme:

- CodePen: crea un nuovo progetto, seleziona la scheda **JavaScript** e incolla il codice.
- JSFiddle: incolla il codice nella sezione **JavaScript** e premi **Run** per eseguirlo.
- Repl.it: crea un nuovo progetto di tipo JavaScript e incolla il codice, quindi premi **Run**.

Queste piattaforme sono particolarmente utili per progetti più complessi e ti permettono di condividere facilmente i tuoi esperimenti con altri.

3. Editor di Testo con Browser (Visual Studio Code con Live Server)

Per lavorare in locale, puoi usare **Visual Studio Code** insieme all'estensione **Live Server**.

Passaggi:

1. Crea un file `index.html` con un blocco `<script>` all'interno del corpo.
2. Inserisci il codice JavaScript tra i tag `<script></script>`.
3. Apri `index.html` con l'estensione Live Server, che eseguirà il file in un browser.

Esempio di file `index.html`:

```
<!DOCTYPE html>
<html lang="it">
<head>
    <meta charset="UTF-8">
    <title>Test JavaScript</title>
</head>
<body>
    <h1>Esecuzione di codice JavaScript</h1>
    <script>
        // Qui puoi incollare il codice di esempio JavaScript
        let somma = 0;
        for (let i = 1; i <= 5; i++) {
            somma += i;
        }
        console.log("Somma dei numeri da 1 a 5:", somma);
    </script>
</body>
</html>
```

4. Node.js per Eseguire JavaScript Fuori dal Browser

Se hai installato **Node.js**, puoi eseguire codice JavaScript direttamente dal terminale. Questo è utile per testare codice che non richiede funzionalità specifiche del browser (come `alert` o `document`).

Come fare:

1. Installa Node.js (disponibile su nodejs.org).
2. Crea un file con estensione `.js` (es. `test.js`).

3. Incolla il codice JavaScript nel file.
4. Esegui il comando `node test.js` nel terminale per vedere l'output.

Esempio di `test.js`:

```
let somma = 0;
for (let i = 1; i <= 5; i++) {
    somma += i;
}
console.log("Somma dei numeri da 1 a 5:", somma);
```

Queste sono le opzioni principali per provare il codice JavaScript direttamente.

Cos'è un Blocco di Codice?

Un blocco di codice viene rappresentato con le parentesi graffe `{ }`, ed è utilizzato per raggruppare una serie di istruzioni che devono essere eseguite insieme. Possiamo trovarlo, ad esempio, all'interno di un ciclo `for`, di una struttura condizionale `if`, oppure come corpo di una funzione.

Sintassi:

```
{
    // Inizia il blocco di codice
    // Qui si scrivono le istruzioni
    let a = 5;
    console.log(a);
    // Finisce il blocco di codice
}
```

Questo blocco può contenere una o più istruzioni, che vengono eseguite nell'ordine in cui sono scritte.

Uso dei Blocchi di Codice nelle Strutture di Controllo

Con `if` e `else`

Il blocco di codice è usato per raggruppare le istruzioni da eseguire in base al risultato di una condizione.

Esempio:

```
let numero = 10;

if (numero > 5) {
    // Blocco di codice eseguito se la condizione è vera
    console.log("Il numero è maggiore di 5");
} else {
    // Blocco di codice eseguito se la condizione è falsa
    console.log("Il numero è 5 o minore");
}
```

Qui, ogni blocco (uno per l'`if` e uno per l'`else`) è separato, e il codice dentro un blocco verrà eseguito solo se la rispettiva condizione è soddisfatta.

Con i Cicli `for` e `while`

Nei cicli, il blocco di codice rappresenta le istruzioni che verranno ripetute a ogni iterazione.

Esempio con `for`:

```
for (let i = 0; i < 3; i++) {
    // Blocco di codice eseguito ad ogni iterazione del ciclo
    console.log("Iterazione numero:", i);
}
```

In questo caso, il blocco di codice dentro le parentesi graffe verrà eseguito tre volte, una per ogni valore di `i` da `0` a `2`.

Blocchi di Codice nelle Funzioni

Le funzioni in JavaScript utilizzano un blocco di codice per definire il corpo della funzione, che contiene tutte le istruzioni che la funzione deve eseguire quando viene chiamata.

Esempio di Funzione:

```
function saluta() {
    // Blocco di codice della funzione
    let nome = "Mario";
    console.log("Ciao, " + nome);
}
```

Qui, il blocco di codice `{ ... }` contiene tutte le istruzioni della funzione `saluta`. Ogni volta che `saluta()` viene chiamata, viene eseguito il blocco di codice.

Ambito delle Variabili nei Blocchi di Codice (Block Scope)

L'uso delle variabili nei blocchi di codice è strettamente legato all'**ambito** (scope) delle variabili. In JavaScript, le variabili dichiarate con `let` e `const` all'interno di un blocco sono limitate a quel blocco. Questo è noto come **block scope**.

Esempio:

```
    let x = 10;
    console.log(x); // Funziona, stampa: 10
}
console.log(x); // Errore: x non è definita al di fuori del blocco
```

In questo caso, `x` esiste solo all'interno del blocco di codice delimitato da `{ }`. Una volta che il blocco termina, `x` non è più accessibile.

Blocco di Codice Vuoto

In JavaScript, è anche possibile utilizzare blocchi di codice vuoti, cioè blocchi che non contengono istruzioni. Sebbene questo sia raro, a volte può essere utile per scopi di debug o per creare una struttura di codice provvisoria.

Esempio:

```
{
    // Blocco di codice vuoto
}
```

Questo non ha alcun effetto pratico ma è sintatticamente valido.

Nesting (Nidificazione) di Blocchi di Codice

I blocchi di codice possono essere annidati, cioè possono contenere altri blocchi al loro interno. Questo è comune nelle strutture di controllo complesse.

Esempio:

```
if (true) {
    console.log("Condizione esterna vera");

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

In questo esempio, ci sono due blocchi di codice, uno dentro l'altro. Il blocco interno verrà eseguito solo se anche il blocco esterno è eseguito (ossia se la prima condizione è vera).

Best Practices per l'Uso dei Blocchi di Codice

- **Indentazione**: indentare correttamente il codice all'interno dei blocchi migliora la leggibilità.
- **Nomi delle variabili univoci nei blocchi**: evitare di usare nomi di variabili uguali nei blocchi nidificati può prevenire errori.
- **Uso di `let` e `const` invece di `var`**: le variabili dichiarate con `let` e `const` rispettano il block scope, mentre `var` è function-scoped, e quindi è visibile all'interno dell'intera funzione in cui viene dichiarata, non solo nel blocco.

Riassunto

In JavaScript, un blocco di codice:

- è delimitato da `{ }` e contiene una o più istruzioni;
- viene utilizzato per raggruppare codice in strutture di controllo (`if`, `for`, `while`, funzioni, ecc.);
- definisce un contesto di ambito per variabili dichiarate con `let` e `const` (block scope);
- può essere annidato, permettendo strutture complesse;
- migliora la leggibilità e la struttura logica del programma.