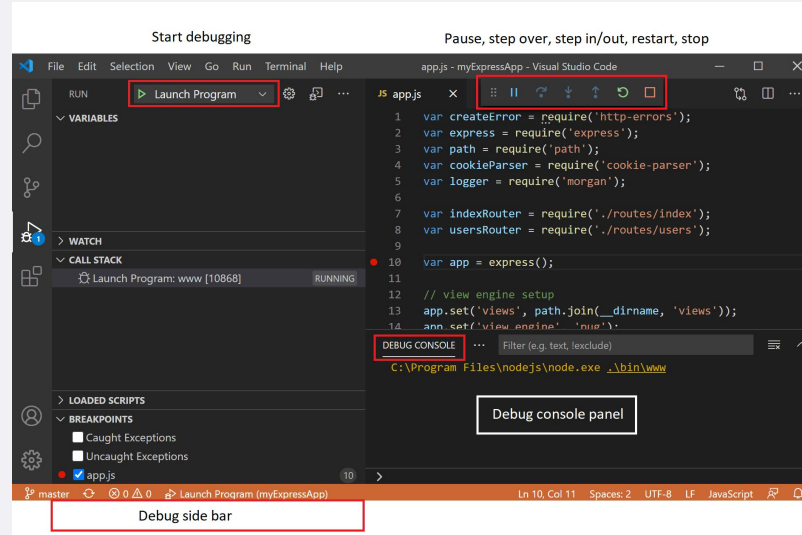




# **INTRODUZIONE AL DEBUG**

# Cos'è il Debug

- Processo di identificazione e risoluzione di errori (bug) nel codice
- Parte fondamentale del ciclo di sviluppo software
- Competenza essenziale per ogni sviluppatore
- Può occupare fino al 50% del tempo di sviluppo



# Tipi di Errori

1. **Errori di Sintassi:** impediscono l'esecuzione del codice
  - Python: `SyntaxError`, `IndentationError`
  - JavaScript: `SyntaxError`
2. **Errori Runtime:** si verificano durante l'esecuzione
  - Python: `TypeError`, `ValueError`, `IndexError`
  - JavaScript: `TypeError`, `ReferenceError`
3. **Errori Logici:** funzionamento non corretto ma senza messaggi di errore
  - I più difficili da rilevare
  - Risultati inaspettati o incorretti



# Strategie di Debug Generali



- Approccio scientifico: osservare, ipotizzare, testare
- Divide et impera: isolare le parti problematiche
- Logging strategico: aggiungere output in punti chiave
- Controllo di flusso: verificare i percorsi di esecuzione
- Minimizzazione: creare un esempio minimo che riproduce il problema



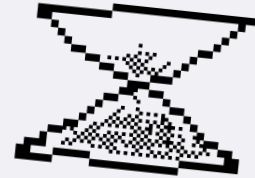
A hand-drawn decorative border in black ink, featuring various loops, swirls, and flourishes that frame the central text. The border is irregular and artistic, with some lines extending outwards from the corners and sides.

# **DEBUG IN JAVASCRIPT/NODE.JS**

# Strumenti di Debug in JavaScript/Node.js

- `console.log()`
- `console.error()`
- `console.table()` // Visualizza array o oggetti in formato tabella
- `console.time()`
  - a. `console.time("operazione");` // Inizia a misurare il tempo
  - b. `// ... operazioni da misurare ...`
  - c. `console.timeEnd("operazione");` // Termina e mostra il tempo trascorso
- Debugger di Chrome DevTools (per frontend e Node.js)
- Node.js Debugger integrato
- Debugging in VS Code

# Debug con Console



```
1 function calcolaMedia(numeri) {
2   console.log("Input ricevuto:", numeri);
3
4   if (!Array.isArray(numeri)) {
5     console.error("Input non valido, deve essere un array");
6     return null;
7   }
8
9   if (numeri.length === 0) {
10    console.warn("Array vuoto, impossibile calcolare la media");
11    return null;
12  }
13
14  const somma = numeri.reduce((acc, val) => acc + val, 0);
15  console.log("Somma calcolata:", somma);
16
17  const media = somma / numeri.length;
18  console.log("Media calcolata:", media);
19
20  return media;
21 }
```


```
1 // Test con diversi input
2 console.log("Risultato:",
3   calcolaMedia([10, 20, 30]));
4 console.log("Risultato:",
5   calcolaMedia([]));
6 console.log("Risultato:",
7   calcolaMedia("non un array"));
```

# Debug Avanzato con Console

```
1 // Tabelle per dati strutturati
2 const dati = [
3   { nome: "Mario", età: 30, ruolo:
    "sviluppatore" },
4   { nome: "Luigi", età: 25, ruolo:
    "designer" },
5   { nome: "Giulia", età: 28, ruolo:
    "manager" }
6 ];
7
8 console.table(dati);
```

```
1 console.log("Lettura dati...");
2 console.log("Validazione...");
3 // altri codice (magari con un if)
4 console.error("Salvataggio...");
```





# Breakpoints nel browser

I browser moderni offrono potenti strumenti di debugging:

- Apri DevTools (F12 o Ctrl+Shift+I)
- Vai nella scheda "Sources" o "Debugger"
- Clicca sul numero di riga per impostare un breakpoint
- Usa i controlli per avanzare step-by-step

# Debugging con Node.js

```
1 // app.js
2 function elaboraData(dati) {
3   debugger; // Il debugger si fermerà qui quando attivato
4
5   if (!dati || !dati.length) {
6     throw new Error("Dati non validi");
7   }
8
9   const risultati = dati.map((item, index) => {
10     return {
11       id: index,
12       valore: item * 2,
13       categoria: item > 10 ? "alto" : "basso"
14     };
15   });
16
17   return risultati;
18 }
19
20 // Test
21 try {
22   const dati = [5, 15, 8, 20];
23   const risultati = elaboraData(dati);
24   console.log("Risultati:", risultati);
25
26   // Questo causerà un errore
27   const risultatiErrore = elaboraData(null);
28 } catch (errore) {
29   console.error("Si è verificato un errore:", errore.message);
30 }
```

## - Installa Node.js

Scarica e installa l'ultima versione di Node.js (necessaria la v6.3.0 o superiore).

## - Avvia Node.js con il flag `--inspect-brk`

Esegui il tuo script interrompendo all'inizio con:

```
node --inspect-brk index.js
```

## - Apri `about:inspect` in Chrome

Digita `about:inspect` nella barra degli indirizzi di Chrome e premi Invio.

## - Avvia gli strumenti di debug

Clicca su "Open dedicated DevTools for Node".

- Oppure Usa Visual Studio Code con l'estensione Node Debugger!!

# Debugging con Node.js

La differenza principale tra i due comandi è quando e come viene attivato il debugger:

1. `node --inspect index.js`
  - Avvia Node.js con il debugger attivo, ma **non interrompe l'esecuzione** all'inizio.
  - Devi impostare manualmente i breakpoint nel codice o nei DevTools di Chrome.
  - Utile per il debug di codice in esecuzione continua (es. server).
2. `node --inspect-brk index.js`
  - Avvia Node.js con il debugger attivo e **sospende immediatamente** l'esecuzione alla prima riga del codice.
  - Utile se vuoi fermarti all'inizio per analizzare lo stato dell'applicazione prima che parta.

Se devi attaccarti al debugger prima che qualsiasi codice venga eseguito, usa `--inspect-brk`.  
Se vuoi solo avere il debugging attivo senza fermare il codice all'inizio, usa `--inspect`.

A hand-drawn decorative border in black ink, featuring various loops, swirls, and lines that frame the central text. The border is irregular and artistic, with some lines extending outwards from the corners and sides.

# **Debugging in Python**

# Strumenti di Debug in Python

- `print()`: semplice ma efficace
- Logging con il modulo `logging`
- Debugger integrato: `pdb`
- IDE con funzionalità di debug: PyCharm, VS Code

```
1 # Debugging semplice con print
2 print(f"Valore di x: {x}")
3
4 # Utilizzare il modulo pdb (Python Debugger)
5 import pdb; pdb.set_trace() # Python < 3.7
6 breakpoint() # Python >= 3.7
7
8 # Comandi principali di pdb:
9 # n (next): esegui la linea corrente
10 # s (step): entra nella funzione chiamata
11 # c (continue): continua fino al prossimo breakpoint
12 # p variabile: stampa il valore di una variabile
13 # q (quit): esci dal debugger
```

# Debug con print

```
1 def calcola_media(neri):
2     print(f"Input ricevuto: {neri}")
3     somma = sum(neri)
4     print(f"Somma calcolata: {somma}")
5     media = somma / len(neri)
6     print(f"Media calcolata: {media}")
7     return media
8
9 # Esempio di utilizzo
10 try:
11     risultato = calcola_media([10, 20, 30])
12     print(f"Risultato: {risultato}")
13     risultato_problematico = calcola_media([]) # Causerà un errore
14 except Exception as e:
15     print(f"Errore rilevato: {e}")
```

# Debug con PDB

```
1 def funzione_complessa(dati):
2     # Inserire breakpoint
3     import pdb; pdb.set_trace() # In Python 3.7+: breakpoint()
4
5     risultato = []
6     for item in dati:
7         # Logica complessa
8         valore = item * 2
9         if valore > 10:
10             risultato.append(valore)
11
12     return risultato
13
14 # Comandi PDB utili:
15 # n (next): esegue la linea corrente
16 # s (step): entra nelle funzioni chiamate
17 # c (continue): continua fino al prossimo breakpoint
18 # p variabile: stampa il valore di variabile
19 # l (list): mostra il codice intorno alla posizione corrente
20 # q (quit): esce dal debugger
```

# Logging in Python

```
1 import logging
2
3 # Configura il logger
4 logging.basicConfig(
5     level=logging.DEBUG,
6     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
7     filename='app.log'
8 )
9
10 # Usa diversi livelli di logging
11 logging.debug("Informazione dettagliata per debugging")
12 logging.info("Informazione generale sul funzionamento")
13 logging.warning("Avviso: potenziale problema")
14 logging.error("Si è verificato un errore")
15 logging.critical("Errore critico che blocca l'applicazione")
```

```
1 import logging
2
3 # Configurazione base del logging
4 logging.basicConfig(
5     level=logging.DEBUG,
6     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
7     filename='app.log'
8 )
9
10 def calcola_media(neri):
11     logging.debug(f"Funzione chiamata con: {neri}")
12
13     if not nerri:
14         logging.error("Lista vuota, impossibile calcolare la media")
15         return None
16
17     somma = sum(nerri)
18     logging.debug(f"Somma calcolata: {somma}")
19
20     media = somma / len(nerri)
21     logging.info(f"Media calcolata: {media}")
22
23     return media
24
25 # Esempio di utilizzo
26 risultato = calcola_media([10, 20, 30])
27 print(f"Risultato: {risultato}")
```



# Gestione delle Eccezioni in Python

```
1 def funzione_rischiosa(valore):
2     try:
3         risultato = 100 / valore
4         return risultato
5     except ZeroDivisionError:
6         print("Errore: Divisione per zero")
7         return None
8     except TypeError as e:
9         print(f"Errore di tipo: {e}")
10        return None
11    finally:
12        print("Operazione completata (con o senza successo)")
13
14 # Test con valori diversi
15 print(funzione_rischiosa(5))    # Funziona
16 print(funzione_rischiosa(0))    # ZeroDivisionError
17 print(funzione_rischiosa("10")) # TypeError
```



FINE



— WWW.DRACO.ME —