La **formattazione delle stringhe in JavaScript** permette di manipolare e combinare stringhe per produrre output dinamici e ben formattati. Esistono vari metodi e tecniche per ottenere questa formattazione, utilizzando strumenti come operatori, funzioni predefinite e template literals.

1. Concatenazione con l'operatore `+`

L'operatore `+` è stato uno dei primi modi per unire le stringhe in JavaScript.

Esempio:

```
const nome = "Mario";
const messaggio = "Ciao, " + nome + "!";
console.log(messaggio); // Output: "Ciao, Mario!"
```

Questo approccio può diventare complesso da leggere con stringhe lunghe o dinamiche.

2. Template Literals (`template strings`)

I template literals sono introdotti in ES6 (ECMAScript 2015) e utilizzano il simbolo **backtick** (`` ` ``) (Alt + 096) per creare stringhe dinamiche e multilinea, permettendo l'inclusione di variabili ed espressioni tramite la sintassi `${...}`.

Vantaggi:

- Miglior leggibilità rispetto alla concatenazione.
- Supporto per stringhe multilinea senza necessità di caratteri di escape.

Esempio:

```
const nome = "Mario";
const eta = 30;
const messaggio = `Ciao, ${nome}! Hai ${eta} anni.`;
console.log(messaggio); // Output: "Ciao, Mario! Hai 30 anni."
```

Stringhe multilinea:

```
const messaggio = `Questa è
una stringa su
più righe.`;
console.log(messaggio);
// Output:
// Questa è
// una stringa su
// più righe.
```

3. Metodi delle stringhe

JavaScript offre molti metodi utili per manipolare e formattare le stringhe. Ecco una panoramica dei più comuni:

3.1 `toUpperCase()` e `toLowerCase()`

Convertire l'intera stringa in maiuscolo o minuscolo:

```
const testo = "Ciao, Mondo!";
console.log(testo.toUpperCase()); // Output: "CIAO, MONDO!"
console.log(testo.toLowerCase()); // Output: "ciao, mondo!"
```

3.2 `trim()`

Rimuove gli spazi bianchi all'inizio e alla fine della stringa:

```
const testo = "   Ciao!   ";
console.log(testo.trim()); // Output: "Ciao!"
```

3.3 `padStart()` e `padEnd()`

Aggiungono caratteri all'inizio o alla fine della stringa fino a raggiungere una certa lunghezza:

```
const numero = "5";
console.log(numero.padStart(3, "0")); // Output: "005"
console.log(numero.padEnd(3, "0"));   // Output: "500"
```

3.4 `replace()` e `replaceAll()`

Sostituiscono parte della stringa.

- **`replace()`** sostituisce solo la prima occorrenza.
- **`replaceAll()`** sostituisce tutte le occorrenze.

```
const testo = "banana e banana";
console.log(testo.replace("banana", "mela")); // Output: "mela e banana"
console.log(testo.replaceAll("banana", "mela")); // Output: "mela e mela"
```

3.5 `split()`

Divide una stringa in un array basandosi su un delimitatore:

```
const frase = "Ciao, come stai?";
const parole = frase.split(" ");
console.log(parole); // Output: ["Ciao,", "come", "stai?"]
```

3.6 `slice()` e `substring()`

Estraggono una parte della stringa:

```
const testo = "Ciao, Mondo!";
console.log(testo.slice(0, 4)); // Output: "Ciao"
console.log(testo.substring(6, 11)); // Output: "Mondo"
```

3.7 `concat()`

Concatena stringhe (equivalente all'operatore +):

```
const parte1 = "Ciao, ";
const parte2 = "Mondo!";
```

```
console.log(parte1.concat(parte2)); // Output: "Ciao, Mondo!"
```

4. Interpolazione di espressioni

Con i template literals, puoi includere **espressioni JavaScript** oltre alle variabili.

Esempio:

```
const a = 5;
const b = 10;
console.log(`La somma di ${a} e ${b} è ${a + b}.`);
// Output: "La somma di 5 e 10 è 15."
```

5. Stringhe dinamiche avanzate

Costruzione condizionale:

Puoi costruire stringhe dinamiche usando condizioni:

```
const loggato = true;
const messaggio = `Benvenuto, ${loggato ? "utente registrato" : "ospite"}!`;
console.log(messaggio); // Output: "Benvenuto, utente registrato!"
```

Funzioni per formattazione complessa:

Se necessario, puoi definire funzioni per creare stringhe dinamiche:

```
function formattaNome(nome, cognome) {
  return `${nome} ${cognome.toUpperCase()}`;
}
console.log(formattaNome("Mario", "Rossi")); // Output: "Mario ROSSI"
```

6. Stringhe internazionali con `Intl`

Per formattare stringhe con numeri, date o valute, puoi usare l'API `Intl`.

Numeri:

```
const numero = 1234567.89;
console.log(new Intl.NumberFormat('it-IT').format(numero)); // Output:
"1.234.567,89"
```

Valute:

```
const prezzo = 19.99;
console.log(new Intl.NumberFormat('it-IT', { style: 'currency', currency: 'EUR'
}).format(prezzo));
// Output: "19,99 €"
```

Date:

```
const oggi = new Date();
console.log(new Intl.DateTimeFormat('it-IT', { dateStyle: 'long'
}).format(oggi));
// Output: "25 novembre 2024"
```

7. Stringhe HTML e sicurezza

Quando si costruiscono stringhe HTML dinamicamente, bisogna evitare **iniezioni di codice**. Esempio:

```
const nomeUtente = "<script>alert('XSS');</script>";
const messaggio = `<div>Benvenuto, ${nomeUtente}!</div>`;
console.log(messaggio);
// Potenziale vulnerabilità XSS se nomeUtente contiene codice malevolo.
```

Soluzione:

Usa funzioni di escape (es. librerie come DOMPurify) per neutralizzare contenuti non sicuri.

In JavaScript, il simbolo **`$`** è utilizzato principalmente all'interno dei **template literals** (o **template strings**) per l'interpolazione di variabili ed espressioni. Questo rende le stringhe più dinamiche e leggibili, consentendo di combinare testo statico con dati o calcoli dinamici senza dover usare concatenazioni complesse.

Template Literals e Interpolazione

Cos'è un template literal?

Un **template literal** è una stringa racchiusa tra backtick (` `) anziché tra virgolette singole (') o doppie ("). Al suo interno, puoi utilizzare il simbolo **$** con le parentesi graffe (${...}`) per includere variabili o espressioni JavaScript.

Uso del simbolo `$`

Il simbolo `$` in combinazione con le parentesi graffe `${}` indica un'**interpolazione**: il contenuto tra `${}` viene valutato come codice JavaScript, e il risultato viene inserito nella stringa.

Sintassi:

```
`testo ${espressione} testo`
```

Esempi di base

1. **Interpolare variabili**:

```
const nome = "Luca";
const messaggio = `Ciao, ${nome}! Come stai?`;
console.log(messaggio); // Output: "Ciao, Luca! Come stai?"
```

2. **Interpolare espressioni**:

```
const a = 10;
const b = 20;
const risultato = `La somma di ${a} e ${b} è ${a + b}.`;
console.log(risultato); // Output: "La somma di 10 e 20 è 30."
```

3. **Funzioni e chiamate inline**: Puoi usare funzioni o chiamate dirette:

```
function saluto(nome) {
  return `Ciao, ${nome}`;
}
console.log(`${saluto("Maria")}, benvenuta!`);
// Output: "Ciao, Maria, benvenuta!"
```

Caratteristiche avanzate

1. Uso in stringhe multilinea

I template literals supportano le stringhe multilinea senza bisogno di concatenare manualmente o utilizzare caratteri di escape:

```
const messaggio = `Questa è una stringa
che si estende
su più righe.`;
console.log(messaggio);
// Output:
// Questa è una stringa
// che si estende
// su più righe.
```

2. Valutazioni dinamiche

L'uso di `${}` consente di incorporare qualsiasi espressione JavaScript valida:

- **Operazioni matematiche**:

  ```
  const prezzo = 100;
  const sconto = 20;
  const totale = `Il prezzo scontato è ${prezzo - sconto} euro.`;
  console.log(totale); // Output: "Il prezzo scontato è 80 euro."
  ```

- **Condizioni**:

  ```
  const loggato = true;
  const messaggio = `Benvenuto, ${loggato ? "utente registrato" :
  "ospite"}!`;
  console.log(messaggio); // Output: "Benvenuto, utente registrato!"
  ```

- **Esecuzione di metodi o proprietà**:

```
const oggi = new Date();
const dataFormattata = `Oggi è il ${oggi.toLocaleDateString()}.`;
console.log(dataFormattata); // Output: "Oggi è il 25/11/2024."
```

Esempi complessi

1. Generare codice HTML dinamico

Puoi usare i template literals per generare stringhe HTML:

```
const nome = "Luca";
const eta = 25;
const html = `
  <div>
    <h1>Benvenuto, ${nome}!</h1>
    <p>Hai ${eta} anni.</p>
  </div>
`;
console.log(html);
// Output:
// <div>
//   <h1>Benvenuto, Luca!</h1>
//   <p>Hai 25 anni.</p>
// </div>
```

2. Combinazione di cicli e interpolazione

Puoi costruire contenuti dinamici in modo più leggibile rispetto alla concatenazione:

```
const frutta = ["Mela", "Banana", "Arancia"];
const lista = `
  <ul>
    ${frutta.map(item => `<li>${item}</li>`).join("")}
  </ul>
`;
console.log(lista);
// Output:
// <ul>
//   <li>Mela</li>
//   <li>Banana</li>
//   <li>Arancia</li>
// </ul>
```

Differenze tra Template Literals e Concatenazione Tradizionale

Template Literals	Concatenazione Tradizionale
Usa backtick (`` ` ` ``) e `${}`	Usa il simbolo `+` per unire stringhe
Leggibile e conciso	Complesso per stringhe lunghe o dinamiche
Supporta stringhe multilinea direttamente	Richiede `\n` o concatenazione manuale
Valutazioni dinamiche con `${}`	Bisogna chiudere e riaprire virgolette

Esempio di confronto:

- **Concatenazione tradizionale**:

```
const nome = "Luca";
const eta = 25;
const messaggio = "Ciao, " + nome + "! Hai " + eta + " anni.";
console.log(messaggio); // Output: "Ciao, Luca! Hai 25 anni."
```

- **Template Literal**:

```
const nome = "Luca";
const eta = 25;
const messaggio = `Ciao, ${nome}! Hai ${eta} anni.`;
console.log(messaggio); // Output: "Ciao, Luca! Hai 25 anni."
```

Uso del `$` come carattere normale

Il simbolo `$` può essere usato come un carattere normale in una stringa. Non viene interpretato in modo speciale al di fuori del contesto dei template literals:

```
const messaggio = "Il costo è di 50$.";
console.log(messaggio); // Output: "Il costo è di 50$."
```

Limitazioni dei template literals

1. **Escape di caratteri speciali**: Per usare il simbolo **backtick** (`` ` ``) all'interno di un template literal, devi fare l'escape con una barra rovesciata (`\`):

```
const messaggio = `Usa il simbolo \` per delimitare un template literal.`;
console.log(messaggio);
// Output: "Usa il simbolo ` per delimitare un template literal."
```

2. **Performance in contesti complessi**: Sebbene i template literals siano leggibili, in contesti altamente dinamici (es. grandi cicli) potrebbero essere meno performanti rispetto a tecniche più ottimizzate come librerie di template rendering.

In JavaScript, **l'escape dei caratteri speciali** consente di rappresentare caratteri che altrimenti avrebbero un significato speciale nella sintassi delle stringhe o che non possono essere inseriti direttamente. Questo si ottiene usando il carattere di escape **barra rovesciata (\)**.

Di seguito viene fornita una spiegazione dettagliata di come funziona l'escape dei caratteri speciali, con esempi pratici.

1. Caratteri speciali di escape

Tabella dei principali caratteri di escape:

Sequenza di escape	Descrizione	Esempio	Output
`\'`	Apostrofo singolo	`'L\'esempio'`	`L'esempio`
`\"`	Doppi apici	`"Un \"testo\" valido"`	`Un "testo" valido`
`\\`	Barra rovesciata	`"Percorso:\\cartella"`	`Percorso:\cartella`
`\n`	Nuova linea	`"Prima linea\nSeconda linea"`	Prima linea Seconda linea
`\r`	Ritorno a capo (carriage return)	`"Line1\rLine2"`	**Comportamento variabile**
`\t`	Tabulazione	`"Colonna1\tColonna2"`	`Colonna1 Colonna2`
`\b`	Backspace (cancella precedente)	`"AB\bC"`	`AC`
`\f`	Form feed	`"Prima pagina\fSeconda pagina"`	Comportamento storico, poco usato
`\v`	Tabulazione verticale	`"Testo\vTabulato"`	**Non visibile** su molti sistemi
`\0`	Carattere NULL	`"Hello\0World"`	`Hello World` (NULL invisibile)

Esempio pratico:

```
const esempio = "Questa è una stringa con \"doppi apici\" e un\ninterruzione di linea.";
console.log(esempio);
// Output:
// Questa è una stringa con "doppi apici" e un
// interruzione di linea.
```

2. Escape dei caratteri Unicode

Unicode semplice (`\u`)

È possibile rappresentare un carattere Unicode specificando il suo valore esadecimale in formato `\uXXXX`, dove `XXXX` è un numero a 4 cifre.

Esempio:

```
const cuore = "\u2764";
console.log(cuore); // Output: ❤
```

Unicode esteso (`\u{...}`)

Per caratteri Unicode oltre le 4 cifre (come quelli al di fuori del Piano Multilingue Base), si usa la sintassi `\u{...}`.

Esempio:

```
const emoji = "\u{1F600}";
console.log(emoji); // Output: 
```

3. Escape in stringhe Template Literals

Anche nei **template literals** (racchiusi da backtick, `), i caratteri speciali richiedono escape:

- Per rappresentare un backtick: `\` `.
- Per la barra rovesciata: `\\`.

Esempio:

```
const template = `Usa il carattere \` per delimitare un template literal.`;
console.log(template);
// Output: Usa il carattere ` per delimitare un template literal.
```

4. Caratteri di escape in espressioni regolari

Nelle **espressioni regolari** (RegEx), alcuni caratteri hanno significati speciali e devono essere preceduti da una barra rovesciata (`\`) per essere trattati come caratteri letterali.

Caratteri che richiedono escape:

Caratteri speciali	Significato nella RegEx
`.`	Qualsiasi carattere
`*`	Zero o più occorrenze
`+`	Una o più occorrenze
`?`	Zero o una occorrenza
`^`	Inizio stringa o negazione
`$`	Fine stringa
`()`	Gruppi di cattura
`[]`	Intervalli di caratteri
`{}`	Quantificatori
`` ` ``	`` ` ``
`\`	Usata per l'escape dei caratteri

Esempio:

Per cercare un punto letterale, devi fare l'escape:

```
const regex = /\./;
console.log("a.b".match(regex)); // Output: ["."]
```

5. Escape di caratteri in URL

JavaScript fornisce funzioni per gestire i caratteri speciali in URL:

- **`encodeURIComponent`**: Converte tutti i caratteri speciali in una stringa valida per un URL.
- **`encodeURI`**: Simile, ma lascia intatti caratteri come `/`, `:`, ecc.

Esempio:

```
const query = "parametro=valore con spazi";
const url = "http://example.com?" + encodeURIComponent(query);
console.log(url);
// Output: http://example.com?parametro%3Dvalore%20con%20spazi
```

6. Caratteri speciali non immediatamente visibili

6.1 Spazi non rompenti (`\u00A0`)

Uno spazio non rompente è un carattere che impedisce la divisione di una linea:

```
const testo = "Parola\u00A0non divisibile.";
console.log(testo); // Output: "Parola non divisibile."
```

6.2 Line separator (`\u2028`) e Paragraph separator (`\u2029`)

Questi caratteri Unicode rappresentano separatori di linea o paragrafo.

7. Escape di caratteri HTML

Quando lavori con stringhe HTML in JavaScript, è importante "escapare" caratteri come `<`, `>`, `&` per evitare vulnerabilità di tipo XSS.

Soluzione manuale:

```
function escapeHTML(testo) {
  return testo
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#039;");
}
console.log(escapeHTML('<script>alert("XSS")</script>'));
// Output: &lt;script&gt;alert(&quot;XSS&quot;)&lt;/script&gt;
```

8. Escape di sequenze esadecimali e ottali

8.1 Esadecimale (`\xNN`)

Permette di rappresentare un carattere tramite il suo codice esadecimale a 2 cifre:

```
const simbolo = "\xA9"; // Codice per ©
console.log(simbolo); // Output: ©
```

8.2 Ottale (`\NNN`)

Questa notazione è meno comune, rappresenta caratteri con valori ottali:

```
const carattere = "\101"; // Ottale per 'A'
console.log(carattere); // Output: A
```

Errori comuni e limitazioni

1. **Escape incompleto**: Dimenticare di fare l'escape di caratteri speciali può portare a errori di sintassi:

   ```
   const stringa = "Ciao, "Mario""; // Errore di sintassi
   ```

2. **Uso di `\` senza un carattere valido**: Una barra rovesciata senza un carattere di escape valido genera un errore:

   ```
   const stringa = "Ciao, \Mario"; // Errore: carattere non valido
   ```

3. **Limitazioni nei template literals**: Anche nei template literals, certi caratteri speciali richiedono escape (es. backtick).

Una **stringa ben formata** in JavaScript è una stringa che rispetta tutte le regole sintattiche e semantiche previste dal linguaggio. È quindi una stringa che può essere interpretata correttamente dal motore JavaScript senza generare errori o comportamenti imprevisti. Ecco cosa si intende nello specifico:

1. Corretto utilizzo dei delimitatori

Una stringa deve essere racchiusa tra coppie di delimitatori validi:

- Virgole singole: `'stringa'`
- Doppi apici: `"stringa"`
- Backtick (per template literals): `` `stringa` ``

Esempi:

```
// Ben formate
const stringa1 = "Questa è una stringa";
const stringa2 = 'Anche questa è valida';
const stringa3 = `Ecco un template literal`;

// Mal formate
// const stringa4 = "Questa non è valida'; // Errore: delimitatori misti
```

```
// const stringa5 = "Questa non è valida; // Errore: delimitatore mancante
```

2. Escape corretto dei caratteri speciali

Se nella stringa devono essere presenti caratteri speciali (come apici, backslash, o nuove righe), devono essere **correttamente escape**.

Esempi:

```
// Ben formate
const stringa1 = "Ciao, \"Mondo\"!";
const stringa2 = 'Ciao, \'Mondo\'!';
const stringa3 = "Percorso: C:\\Documenti\\File";

// Mal formate
// const stringa4 = "Ciao, "Mondo"!"; // Errore: il doppio apice interrompe la stringa
// const stringa5 = "C:\\Percorso\\Non valido"; // Errore: il backslash non è escape correttamente
```

3. Corretto uso dei template literals

Quando si utilizzano i **template literals**, l'interpolazione deve avvenire nel modo corretto con `${}` per inserire espressioni o variabili. Inoltre, devono rispettare il delimitatore backtick (``).

Esempi:

```
// Ben formata
const nome = "Mario";
const saluto = `Ciao, ${nome}!`;

// Mal formata
// const saluto2 = `Ciao, ${nome!}`; // Errore: manca una parentesi graffa
// const saluto3 = "Ciao, ${nome}!"; // Errore: interpolazione non valida nei doppi apici
```

4. Non interrompere accidentalmente la stringa

Una stringa non può contenere caratteri che interrompano il suo contesto senza essere escape. Ad esempio:

- Non può contenere nuovi apici o virgolette senza escape.
- Non può contenere una nuova linea (a meno che si usino i template literals o l'escape `\n`).

Esempi:

```
// Ben formata
const stringa1 = "Questa è una stringa su una sola riga.";
const stringa2 = "Questa è una stringa che termina con una\nnuova linea.";
const stringa3 = `Questa è una stringa multilinea
```

```
che non genera errori.`;

// Mal formata
// const stringa4 = "Stringa non valida
// su più righe"; // Errore: JavaScript non consente nuove linee nelle stringhe
con " o '
```

5. Codifiche valide

La stringa deve utilizzare caratteri validi o codificati correttamente (esadecimali, Unicode, ecc.) e rispettare i requisiti del contesto.

Esempi:

```
// Ben formata
const stringa1 = "\u0048\u0065\u006C\u006C\u006F"; // "Hello" in Unicode
const stringa2 = "Codice esadecimale: \x41";      // "A" in esadecimale

// Mal formata
// const stringa3 = "\u00G1"; // Errore: sequenza Unicode non valida
```

6. Rispettare il contesto di utilizzo

In certi contesti (come JSON), una stringa è considerata ben formata solo se segue regole specifiche, come l'uso obbligatorio dei doppi apici.

Esempio in JSON:

```
// Ben formata
const jsonString = '{"nome": "Mario", "eta": 30}';

// Mal formata
// const jsonString = "{'nome': 'Mario', 'eta': 30}"; // Errore: JSON richiede
doppi apici
```

7. Assenza di caratteri invisibili o ambigui

Caratteri non visibili (es. spazi non rompenti `\u00A0`, caratteri di controllo) potrebbero rendere una stringa **tecnicamente valida**, ma non sempre ben formata in termini di leggibilità o utilizzo.

Esempio:

```
// Stringa ben formata
const stringa = "Questa è una stringa leggibile";

// Stringa tecnicamente valida ma non ben formata
const stringa2 = "Questa\u00A0è una stringa\u200B con caratteri strani";
// Contiene uno spazio non rompente e un carattere zero-width
```

Conclusione

Una stringa è **ben formata** in JavaScript se:

1. Utilizza correttamente i delimitatori (`'`, `"`, `` ` ``).
2. I caratteri speciali sono correttamente escape.
3. Rispetta il contesto di utilizzo (es. JSON).
4. Non contiene interruzioni di linea non consentite (salvo escape o template literals).
5. Non contiene codifiche o caratteri non validi.

Scrivere stringhe ben formate è fondamentale per evitare errori di sintassi e garantire il corretto funzionamento del codice.