

Ripasso di JavaScript

Agenda

Fondamenti di JavaScript

- Sintassi di base e tipi di dati
- Variabili e costanti (var, let, const)
- Operatori e espressioni
- Strutture di controllo (if/else, switch, cicli)
- Funzioni (dichiarazione, espressioni, arrow functions)

Strutture dati e oggetti

- Array e metodi degli array
- Oggetti e proprietà
- Destrutturazione
- JSON e manipolazione di dati

JavaScript moderno (ES6+)

- Template literals
- Spread e rest operators
- Default parameters
- Classi e ereditarietà
- Moduli (import/export)

Agenda

Funzioni asincrone

- Callbacks
- Promises
- Async/await
- Fetch API e richieste HTTP

Manipolazione del DOM

- Selettori e navigazione del DOM
- Creazione e modifica di elementi
- Eventi e gestione degli eventi
- Form e validazione

Sintassi di base e tipi di dati

JavaScript è un linguaggio a tipizzazione dinamica, il che significa che non è necessario dichiarare il tipo di una variabile quando la si crea.

JavaScript converte automaticamente i tipi quando necessario (coercizione), ma questa può portare a comportamenti inaspettati.



Sintassi di base e tipi di dati

```
1 // Numeri - interi e decimali
2 let intero = 42;
3 let decimale = 3.14;
4
5 // Stringhe - testo racchiuso tra apici
6 let saluto = "Ciao mondo!";
7 let nome = 'Mario';
8 let frase = `Il mio nome è ${nome}`; // Template literal (ES6)
9
10 // Booleani - vero o falso
11 let vero = true;
12 let falso = false;
13
14 // Null e Undefined
15 let vuoto = null;           // valore assegnato intenzionalmente
                               vuoto
16 let nonDefinito;          // valore non ancora assegnato
17
18 // Oggetti - collezioni di coppie chiave-valore
19 let persona = {
20     nome: "Laura",
21     età: 28,
22     studente: true
23 };
24
25 // Array - collezioni ordinate di valori
26 let numeri = [1, 2, 3, 4, 5];
27 let misto = [1, "due", true, null];
```



Variabili e costanti

In JavaScript moderno (ES6+), abbiamo tre modi per dichiarare variabili.

Differenze principali:

- `var` ha scope di funzione e può essere ridichiarata
- `let` ha scope di blocco e può essere riassegnata ma non ridichiarata
- `const` ha scope di blocco e non può essere né riassegnata né ridichiarata

```
1 // var - scoping a livello di funzione (vecchio stile)
2 var nome = "Mario";
3
4 // let - scoping a livello di blocco (uso raccomandato per
  variabili)
5 let età = 25;
6
7 // const - per valori che non cambieranno (uso raccomandato di
  default)
8 const PI = 3.14159;
```



```
1 // Esempio di scope di blocco
2 {
3   let bloccoVariabile = "Sono visibile solo in questo
  blocco";
4   var funzioneVariabile = "Sono visibile fuori dal blocco";
5 }
6 // console.log(bloccoVariabile); // Errore!
7 console.log(funzioneVariabile); // Funziona!
```

Operatori e espressioni

JavaScript supporta vari tipi di operatori:

```
1 let a = 10;
2 let b = 3;
3
4 console.log(a + b); // 13 (addizione)
5 console.log(a - b); // 7 (sottrazione)
6 console.log(a * b); // 30 (moltiplicazione)
7 console.log(a / b); // 3.33... (divisione)
8 console.log(a % b); // 1 (resto della divisione)
9 console.log(a ** b); // 1000 (elevamento a potenza)
10
11 // Incremento e decremento
12 let c = 5;
13 console.log(c++); // 5 (restituisce 5, poi incrementa a 6)
14 console.log(++c); // 7 (incrementa a 7, poi restituisce 7)
```

```
1 console.log(5 == "5"); // true (confronto con conversione di tipo)
2 console.log(5 === "5"); // false (confronto stretto, senza conversione)
3 console.log(5 != "6"); // true
4 console.log(5 !== "5"); // true
5 console.log(5 > 3); // true
6 console.log(5 >= 5); // true
```

```
1 console.log(true && false); // false (AND logico)
2 console.log(true || false); // true (OR logico)
3 console.log(!true); // false (NOT logico)
```



Strutture di controllo

Condizionali e Cicli:

```
1 // if/else
2 let età = 18;
3
4 if (età >= 18) {
5   console.log("Sei maggiorenne");
6 } else if (età >= 16) {
7   console.log("Quasi maggiorenne");
8 } else {
9   console.log("Sei minorenne");
10 }
11
12 // switch
13 let giorno = "Lunedì";
14
15 switch (giorno) {
16   case "Lunedì":
17     console.log("Inizio settimana");
18     break;
19   case "Venerdì":
20     console.log("Quasi weekend");
21     break;
22   case "Sabato":
23   case "Domenica":
24     console.log("Weekend!");
25     break;
26   default:
27     console.log("Giorno feriale");
28 }
29
30 // Operatore ternario
31 let status = età >= 18 ? "adulto" : "minorenne";
```



```
1 // for
2 for (let i = 0; i < 5; i++) {
3   console.log(`Iterazione ${i}`);
4 }
5
6 // while
7 let contatore = 0;
8 while (contatore < 5) {
9   console.log(`Contatore: ${contatore}`);
10  contatore++;
11 }
12
13 // do...while (esegue almeno una volta)
14 let j = 0;
15 do {
16   console.log(`j vale ${j}`);
17   j++;
18 } while (j < 3);
19
20 // for...of (per array)
21 let frutta = ["mela", "banana", "arancia"];
22 for (let frutto of frutta) {
23   console.log(frutto);
24 }
25
26 // for...in (per oggetti)
27 let persona = {nome: "Anna", età: 30};
28 for (let proprietà in persona) {
29   console.log(`${proprietà}: ${persona[proprietà]}`);
30 }
```

Funzioni

Dichiarazione di funzioni e scope:

```
1 // Dichiarazione di funzione standard
2 function saluta(nome) {
3     return `Ciao, ${nome}!`;
4 }
5
6 // Espressione di funzione
7 const salutaEspressione = function(nome) {
8     return `Ciao, ${nome}!`;
9 };
10
11 // Arrow function (ES6)
12 const salutaArrow = (nome) => {
13     return `Ciao, ${nome}!`;
14 };
15
16 // Arrow function con return implicito
17 const salutaBreve = nome => `Ciao, ${nome}!`;
18
19 // Funzione con parametri predefiniti
20 function salutaConDefault(nome = "amico") {
21     return `Ciao, ${nome}!`;
22 }
23
24 console.log(saluta("Mario"));           // Ciao, Mario!
25 console.log(salutaConDefault());        // Ciao, amico!
```



```
● ● ●
1 let globale = "Variabile globale";
2
3 function mostraScope() {
4     let locale = "Variabile locale";
5     console.log(globale); // Accessibile
6     console.log(locale); // Accessibile
7 }
8
9 mostraScope();
10 // console.log(locale); // Errore: locale non è definita
```

Esercizi pratici

Livello basilare

1. **Conversione di tipi:** Scrivi un programma che prende una stringa contenente un numero (es. "42") e la converta in un numero, poi aggiungi 10 e mostra il risultato.
2. **Controllo pari/dispari:** Crea una funzione che accetta un numero e restituisce "pari" se il numero è pari, altrimenti "dispari".
3. **Concatenazione stringhe:** Scrivi una funzione che prende nome e cognome come parametri e restituisce un saluto formale (es. "Buongiorno, Mario Rossi!").
4. **Contatore while:** Utilizza un ciclo while per contare da 10 a 1 (conto alla rovescia).
5. **Area rettangolo:** Crea una funzione che calcola l'area di un rettangolo date base e altezza come parametri.



Esercizi pratici

Livello intermedio

1. **Validazione email:** Scrivi una funzione che verifica se una stringa contiene il carattere "@" e almeno un punto dopo la @, restituendo un booleano.
2. **Numeri primi:** Crea una funzione che determina se un numero è primo (divisibile solo per 1 e per sé stesso).
3. **Ordinamento array:** Scrivi una funzione che accetta un array di numeri e lo restituisce ordinato dal più piccolo al più grande.
4. **Filtro array:** Crea una funzione che filtra un array di numeri, restituendo solo quelli maggiori di un valore soglia fornito come parametro.
5. **Contatore lettere:** Scrivi una funzione che conta quante volte una certa lettera appare in una stringa.



Esercizi pratici

Livello difficile

1. **Conversione numeri romani:** Scrivi una funzione che converte un numero (da 1 a 100) in numeri romani.
2. **Scrivi un programma che stampi i numeri da 1 a 100.** Per i multipli di 3, stampa "Fizz" invece del numero. Per i multipli di 5, stampa "Buzz". Per i numeri che sono sia multipli di 3 che di 5, stampa "FizzBuzz".



Agenda

Strutture dati e oggetti

- Array e metodi degli array
- Oggetti e proprietà
- Destrutturazione
- JSON e manipolazione di dati



Array e metodi degli array

Gli array in JavaScript sono collezioni ordinate di valori accessibili tramite indici numerici. Rappresentano uno dei modi più comuni per organizzare e manipolare dati.

Creazione e accesso agli array:

```
 1 // Creazione di array
 2 let numeri = [1, 2, 3, 4, 5];
 3 let frutta = ["mela", "banana", "arancia"];
 4 let misto = [42, "ciao", true, null, { nome: "Mario" }];
 5 let arrayVuoto = [];
 6
 7 // Accesso agli elementi (tramite indice a base 0)
 8 console.log(frutta[0]); // "mela"
 9 console.log(frutta[2]); // "arancia"
10
11 // Modifica di elementi
12 frutta[1] = "kiwi";
13 console.log(frutta); // ["mela", "kiwi", "arancia"]
14
15 // Proprietà length
16 console.log(numeri.length); // 5
```



Array e metodi degli array

Aggiungere/rimuovere elementi e Ricerca e selezione:

```
1 let colori = ["rosso", "verde"];
2
3 // Aggiungere alla fine
4 colori.push("blu");           // Restituisce la nuova lunghezza:
5
6 console.log(colori);         // ["rosso", "verde", "blu"]
7
8 // Rimuovere dall'ultima posizione
9 let ultimo = colori.pop();   // Restituisce l'elemento rimosso:
10 "blu"
11 console.log(colori);         // ["rosso", "verde"]
12
13 // Aggiungere all'inizio
14 colori.unshift("giallo");    // Restituisce la nuova lunghezza:
15 3
16 console.log(colori);         // ["giallo", "rosso", "verde"]
17
18 // Rimuovere dalla prima posizione
19 let primo = colori.shift(); // Restituisce l'elemento rimosso:
20 "giallo"
21 console.log(colori);         // ["rosso", "verde"]
22
23 // Rimuovere/sostituire da qualsiasi posizione
24 // array.splice(indiceInizio, quantiElementi, nuoviElementi...)
25 colori = ["rosso", "verde", "blu", "giallo"];
26 colori.splice(1, 2);          // Rimuove 2 elementi dalla
27 posizione 1
28 console.log(colori);         // ["rosso", "giallo"]
29
30 colori = ["rosso", "verde", "blu", "giallo"];
31 colori.splice(1, 2, "arancione", "viola"); // Sostituisce 2
32 elementi
33 console.log(colori);         // ["rosso", "arancione", "viola",
34 "giallo"]
```



```
1 let numeri = [10, 20, 30, 40, 50];
2
3 // indexOf - trova la prima occorrenza di un elemento
4 console.log(numeri.indexOf(30));      // 2
5 console.log(numeri.indexOf(60));      // -1 (non trovato)
6
7 // includes - verifica se l'array contiene un elemento
8 console.log(numeri.includes(40));     // true
9 console.log(numeri.includes(60));     // false
10
11 // find - trova il primo elemento che soddisfa una condizione
12 let trovato = numeri.find(num => num > 25);
13 console.log(trovato);               // 30
14
15 // filter - trova tutti gli elementi che soddisfano una
16 // condizione
17 let filtrati = numeri.filter(num => num > 25);
18 console.log(filtrati);             // [30, 40, 50]
19
20 // slice - estrae una porzione dell'array
21 // array.slice(indiceInizio, indiceFine)
22 let porzione = numeri.slice(1, 4);
23 console.log(porzione);            // [20, 30, 40]
```

Array e metodi degli array

Trasformazione + Join e split:

```
1 let numeri = [1, 2, 3, 4, 5];
2
3 // map - crea un nuovo array applicando una funzione a ogni
elemento
4 let quadrati = numeri.map(num => num * num);
5 console.log(quadrati);           // [1, 4, 9, 16, 25]
6
7 // reduce - riduce l'array a un singolo valore
8 // array.reduce(callback(accumulatore, valoreCorrente),
valoreIniziale)
9 let somma = numeri.reduce((totale, numero) => totale + numero,
0);
10 console.log(somma);            // 15
11
12 // sort - ordina gli elementi dell'array
13 let frutta = ['banana', 'mela', 'arancia'];
14 frutta.sort();
15 console.log(frutta);           // ["arancia", "banana",
"mela"]
16
17 // ATTENZIONE: sort() converte tutto in stringhe per default
18 let cifre = [10, 2, 30, 4];
19 cifre.sort();
20 console.log(cifre);            // [10, 2, 30, 4] (ordine
lessicografico)
21
22 // Per ordinare numericamente, passare una funzione di
confronto
23 cifre.sort((a, b) => a - b);
24 console.log(cifre);            // [2, 4, 10, 30]
25
26 // reverse - inverte l'ordine degli elementi
27 numeri.reverse();
28 console.log(numeri);           // [5, 4, 3, 2, 1]
```



```
1 // join - unisce gli elementi in una stringa
2 let frutta = ["mela", "banana", "kiwi"];
3 let testo = frutta.join(", ");
4 console.log(testo);           // "mela, banana, kiwi"
5
6 // split (metodo delle stringhe) - divide una stringa in un
array
7 let elenco = "rosso,verde,blu";
8 let colori = elenco.split(",");
9 console.log(colori);           // ["rosso", "verde",
"blu"]
```

```
1 let numeri = [1, 2, 3, 4, 5];
2
3 // forEach - esegue una funzione su ogni elemento
4 numeri.forEach(numero => {
5   console.log(numero * 2);
6 });
7
8 // every - verifica se tutti gli elementi soddisfano una
condizione
9 let tuttiPositivi = numeri.every(num => num > 0);
10 console.log(tuttiPositivi);      // true
11
12 // some - verifica se almeno un elemento soddisfa una
condizione
13 let almeno3 = numeri.some(num => num > 3);
14 console.log(almeno3);           // true
```

Oggetti e proprietà

Gli oggetti in JavaScript sono collezioni di coppie chiave-valore e rappresentano un modo flessibile per modellare dati strutturati.

```
1 // Creazione di un oggetto
2 let persona = {
3   nome: "Mario",
4   cognome: "Rossi",
5   età: 30,
6   città: "Roma",
7   email: "mario.rossi@example.com"
8 };
9
10 // Accesso alle proprietà (due metodi)
11 console.log(persona.nome);           // "Mario"
12 console.log(persona["cognome"]);      // "Rossi"
13
14 // La notazione con parentesi quadre permette di usare
15 // variabili
15 let proprietà = "età";
16 console.log(persona[proprietà]);     // 30
17
18 // Modificare le proprietà
19 persona.età = 31;
20 persona["email"] = "nuovo.email@example.com";
21
22 // Aggiungere nuove proprietà
23 persona.telefono = "123456789";
24 persona["professione"] = "Sviluppatore";
25
26 // Eliminare proprietà
27 delete persona.città;
```



```
1 let persona = {
2   nome: "Mario",
3   cognome: "Rossi",
4   // Un metodo è una funzione all'interno di un oggetto
5   saluta: function() {
6     return `Ciao, sono ${this.nome} ${this.cognome}`;
7   },
8   // Sintassi abbreviata (ES6)
9   presentati() {
10     return `Mi chiamo ${this.nome} e ho ${this.età} anni`;
11   },
12   età: 30
13 };
14
15 console.log(persona.saluta());          // "Ciao, sono Mario
16 Rossi"
```

Oggetti e proprietà

Iterazione sulle proprietà:

```
1 let persona = {  
2     nome: "Mario",  
3     cognome: "Rossi",  
4     età: 30  
5 };  
6  
7 // Ottenere tutte le chiavi (proprietà)  
8 let chiavi = Object.keys(persona);  
9 console.log(chiavi);           // ["nome", "cognome",  
    "età"]  
10  
11 // Ottenere tutti i valori  
12 let valori = Object.values(persona);  
13 console.log(valori);          // ["Mario", "Rossi",  
    30]  
14  
15 // Ottenere coppie [chiave, valore]  
16 let entries = Object.entries(persona);  
17 console.log(entries);         // [[{"nome": "Mario"},  
    {"cognome": "Rossi"}, {"età": 30}]]  
18  
19 // Ciclo for...in  
20 for (let chiave in persona) {  
21     console.log(`${chiave}: ${persona[chiave]}`);  
22 }
```



Destruzione

La destrutturazione è una sintassi introdotta in ES6 che permette di estrarre valori da array e oggetti in modo conciso.

```
1 // Senza destrutturazione
2 let colori = ["rosso", "verde", "blu"];
3 let primo = colori[0];
4 let secondo = colori[1];
5
6 // Con destrutturazione
7 let [r, g, b] = colori;
8 console.log(r, g, b);           // "rosso" "verde" "blu"
9
10 // Saltare elementi
11 let [primo, , terzo] = colori;
12 console.log(primo, terzo);     // "rosso" "blu"
13
14 // Valori di default
15 let [a, b, c, d = "giallo"] = colori;
16 console.log(d);               // "giallo"
17
18 // Rest pattern
19 let [capo, ...squadra] = ["Mario", "Luigi", "Yoshi", "Toad"];
20 console.log(capo);            // "Mario"
21 console.log(squadra);         // ["Luigi", "Yoshi", "Toad"]
22
23 // Scambio di variabili
24 let x = 5, y = 10;
25 [x, y] = [y, x];
26 console.log(x, y);           // 10 5
```



```
1 let persona = {  
2     nome: "Mario",  
3     cognome: "Rossi",  
4     età: 30,  
5     città: "Roma"  
6 };  
7  
8 // Senza destrutturazione  
9 let nome = persona.nome;  
10 let età = persona.età;  
11  
12 // Con destrutturazione  
13 let { nome, età } = persona;  
14 console.log(nome, età);           // "Mario" 30  
15  
16 // Rinominare le variabili  
17 let { nome: firstName, cognome: lastName } = persona;  
18 console.log(firstName, lastName); // "Mario" "Rossi"  
19  
20 // Valori di default  
21 let { hobby = "Lettura" } = persona;  
22 console.log(hobby);             // "Lettura"  
23  
24 // Combinare rinomina e default  
25 let { città: city = "Sconosciuta" } = persona;  
26 console.log(city);              // "Roma"  
27  
28 // Rest pattern in oggetti  
29 let { nome, ...resto } = persona;  
30 console.log(resto);            // { cognome: "Rossi"  
    età: 30, città: "Roma" }
```

Destruzione

La destrutturazione è una sintassi introdotta in ES6 che permette di estrarre valori da array e oggetti in modo conciso.

```
1 // Senza destrutturazione
2 function saluta(persona) {
3     console.log(`Ciao ${persona.nome} ${persona.cognome}`);
4 }
5
6 // Con destrutturazione
7 function salutaConDest({ nome, cognome }) {
8     console.log(`Ciao ${nome} ${cognome}`);
9 }
10
11 salutaConDest(persona);           // "Ciao Mario Rossi"
12
13 // Con valori di default
14 function configura({ colore = "blu", dimensione = "medio",
15     bordo = false } = {}) {
16     console.log(`Colore: ${colore}, Dimensione: ${dimensione},
17     Bordo: ${bordo}`);
18 }
19
20 configura({ colore: "rosso" });    // "Colore: rosso,
21                                         Dimensione: medio, Bordo: false"
22 configura();                      // "Colore: blu,
23                                         Dimensione: medio, Bordo: false"
```

JSON e manipolazione di dati

La destrutturazione è una sintassi introdotta in ES6 che permette di estrarre valori da array e oggetti in modo conciso.

```
1 // Un oggetto JSON valido
2 let persona = {
3     "nome": "Mario",
4     "cognome": "Rossi",
5     "età": 30,
6     "indirizzo": {
7         "città": "Roma",
8         "cap": "00100"
9     },
10    "telefoni": [
11        "06123456",
12        "333123456"
13    ],
14    "attivo": true
15 };
16
17 // Differenze dalla sintassi degli oggetti JavaScript:
18 // - Le chiavi in JSON devono essere racchiuse tra virgolette
19 //   doppie
20 // - JSON non supporta funzioni o undefined come valori
21 // - JSON non supporta commenti
```



```
1 // Da oggetto JavaScript a stringa JSON
2 let persona = {
3     nome: "Mario",
4     età: 30,
5     hobbies: ["calcio", "lettura"]
6 };
7
8 let jsonString = JSON.stringify(persona);
9 console.log(jsonString);
10 // {"nome":"Mario","età":30,"hobbies":["calcio","lettura"]}
11
12 // Formattazione leggibile
13 let jsonFormatto = JSON.stringify(persona, null, 2);
14 console.log(jsonFormatto);
15 /*
16 {
17     "nome": "Mario",
18     "età": 30,
19     "hobbies": [
20         "calcio",
21         "lettura"
22     ]
23 }
24 */
25
26 // Da stringa JSON a oggetto JavaScript
27 let jsonDati = '{"nome":"Luigi","punti":150,"attivo":true}';
28 let oggetto = JSON.parse(jsonDati);
29
30 console.log(oggetto.nome);           // "Luigi"
31 console.log(oggetto.punti);          // 150
```

JSON e manipolazione di dati

Riepilogo

- Gli array sono collezioni ordinate accessibili con indici numerici
- I metodi più importanti degli array includono push/pop, unshift/shift, splice, map, filter, reduce
- Gli oggetti sono collezioni di coppie chiave-valore
- La destrutturazione permette di estrarre valori da array e oggetti in modo conciso
- JSON è un formato standard per scambiare dati tra client e server



Esercizi pratici

Livello basilare

1. **Modifica array:** Scrivi una funzione che prende un array di numeri e restituisce un nuovo array con ogni elemento raddoppiato.
2. **Unione di array:** Crea una funzione che unisce due array in un unico array, alternando gli elementi del primo e del secondo.
3. **Filtra numeri pari:** Crea una funzione che filtra un array, restituendo solo i numeri pari.
4. **Recupero dati da JSON:** Scrivi una funzione che analizza una stringa JSON contenente informazioni su libri e restituisce un array con i titoli.



Esercizi pratici

Livello intermedio

1. **Media voti:** Scrivi una funzione che accetta un array di oggetti studente (con nome e voto) e calcola la media dei voti.
2. **Fusione oggetti:** Crea una funzione deepMerge che unisce due oggetti, anche con proprietà annidate.
3. **Trasformazione JSON:** Scrivi una funzione che prende un oggetto JavaScript complesso e lo converte in JSON, ma esclude le proprietà con valori null o undefined.



Agenda

JavaScript moderno (ES6+)

- Template literals
- Spread e rest operators
- Default parameters
- Classi in ES6
- Moduli (import/export)

Spread e rest operators

Gli operatori spread (...) e rest sono versatili e potenti, utilizzati in diversi contesti per lavorare con array e oggetti.

```
1 // 1. Combinare array
2 const frutta = ["mela", "banana"];
3 const verdura = ["carota", "pomodoro"];
4 const alimenti = [...frutta, ...verdura];
5 console.log(alimenti); // ["mela", "banana", "carota",
6                           "pomodoro"]
7
8 // 2. Creare copie di array (shallow copy)
9 const originale = [1, 2, 3];
10 const copia = [...originale];
11 copia.push(4);
12 console.log(originale); // [1, 2, 3] (non modificato)
13 console.log(copia);     // [1, 2, 3, 4]
14
15 // 3. Passare elementi di array come argomenti a funzioni
16 function somma(a, b, c) {
17   return a + b + c;
18 }
19 const numeri = [1, 2, 3];
20 console.log(somma(...numeri)); // 6
21
22 // 4. Con oggetti (ES2018+)
23 const personal = { nome: "Mario", età: 30 };
24 const persona2 = { ...personal, città: "Roma", età: 31 };
25 console.log(persona2); // { nome: "Mario", età: 31, città:
                           "Roma" }
26
27 // Nota: le proprietà successive sovrascrivono quelle
28 // precedenti (età)
```



```
1 // 1. Raccogliere argomenti rimanenti in una funzione
2 function mostraArgomenti(primo, ...resto) {
3   console.log("Primo argomento:", primo);
4   console.log("Altri argomenti:", resto);
5 }
6
7 mostraArgomenti("a", "b", "c", "d");
8 // "Primo argomento: a"
9 // "Altri argomenti: ["b", "c", "d"]"
10
11 // 2. Destrutturazione con rest
12 const [primoFrutto, secondoFrutto, ...altriFrutti] = ["mela",
13                                         "banana", "arancia", "kiwi"];
14 console.log(primoFrutto); // "mela"
15 console.log(secondoFrutto); // "banana"
16 console.log(altriFrutti); // ["arancia", "kiwi"]
17
18 // 3. Destrutturazione di oggetti con rest
19 const { nome, ...altreProprietà } = { nome: "Mario", età: 30,
20                                         città: "Roma" };
21 console.log(nome); // "Mario"
22 console.log(altreProprietà); // { età: 30, città: "Roma" }
```

Template literals

I template literals (o template strings) sono un modo avanzato per lavorare con le stringhe, permettendo di incorporare espressioni e creare stringhe multilinea con facilità.

```
1 // Sintassi base: backtick (`) invece di apici singoli o doppi
2 const saluto = `Ciao mondo`;
3
4 // Interpolazione di variabili ed espressioni
5 const nome = "Mario";
6 const età = 30;
7 const messaggio = `${nome} ha ${età} anni.`;
8 console.log(messaggio); // "Mario ha 30 anni."
9
10 // Espressioni nelle stringhe
11 console.log(`2 + 2 = ${2 + 2}`); // "2 + 2 = 4"
12 console.log(`${nome.toUpperCase()} ha ${2023 - 1993} anni.`);
13 // "MARIO ha 30 anni."
14 // Stringhe multilinea (preserva interruzioni di linea e
15 // indentazione)
15 const poesia = `
16 Questa è una poesia
17 su più righe
18 senza necessità di escape
19 o concatenazioni.
20 `;
21 console.log(poiesia);
22
23 // Tagged templates (avanzato)
24 function evidenzia(stringhe, ...valori) {
25   return stringhe.reduce((risultato, stringa, i) => {
26     return `${risultato}${stringa}<strong>${valori[i]} ||
27       ''</strong>`;
28   }, '');
29 }
30 const prezzo = 19.99;
31 const prodotto = "libro";
32 console.log(evidenzia`Il ${prodotto} costa ${prezzo}€.`);
33 // "Il <strong>libro</strong> costa <strong>19.99</strong>€."
```



Default parameters

I parametri predefiniti permettono di specificare valori di default per i parametri di una funzione, semplificando la gestione di argomenti mancanti o undefined.

```
● ● ●  
1 // Sintassi base  
2 function saluta(nome = "Ospite") {  
3     return `Ciao, ${nome}!`;  
4 }  
5  
6 console.log(saluta("Mario")); // "Ciao, Mario!"  
7 console.log(saluta());         // "Ciao, Ospite!"  
8  
9 // Parametri predefiniti più complessi  
10 function creaOggetto(id, { nome = "Predefinito", valore = 0 } =  
11     {}) {  
12     return { id, nome, valore };  
13 }  
14 console.log(creaOggetto(1, { nome: "Prodotto", valore: 100 }));  
15 // { id: 1, nome: "Prodotto", valore: 100 }  
16 console.log(creaOggetto(2, { nome: "Servizio" }));  
17 // { id: 2, nome: "Servizio", valore: 0 }  
18 console.log(creaOggetto(3));  
19 // { id: 3, nome: "Predefinito", valore: 0 }  
20  
21 // I parametri predefiniti possono usare valori precedenti  
22 function incrementa(base, incremento = 1, risultato = base +  
23     incremento) {  
24     return risultato;  
25 }  
26 console.log(incrementa(5));      // 6  
27 console.log(incrementa(5, 2));    // 7
```



Classi ES6

Le classi in JavaScript, introdotte in ES6, forniscono una sintassi più pulita e familiare per la programmazione orientata agli oggetti, semplificando la creazione di costruttori, l'ereditarietà e la gestione dei metodi.

```
1 // Definizione di classe base
2 class Persona {
3     // Il costruttore viene chiamato quando si istanzia la classe
4     constructor(nome, età) {
5         this.nome = nome;
6         this.età = età;
7     }
8
9     // Metodi della classe
10    saluta() {
11        return `Ciao, sono ${this.nome}!`;
12    }
13
14    // Getter e setter
15    get informazioni() {
16        return `${this.nome}, ${this.età} anni`;
17    }
18
19    set nuovaEtà(età) {
20        if (età >= 0) {
21            this.età = età;
22        }
23    }
24
25    // Metodi statici (chiamati sulla classe, non sull'istanza)
26    static creaPersonaAnonima() {
27        return new Persona("Anonimo", 30);
28    }
29 }
```



```
1 // Creazione di istanze
2 const personal = new Persona("Mario", 30);
3 console.log(personal.saluta());           // "Ciao, sono Mario!"
4 console.log(personal.informazioni);       // "Mario, 30 anni"
5 personal.nuovaEtà = 31;
6 console.log(personal.età);                // 31
7
8 const anonimo = Persona.creaPersonaAnonima();
9 console.log(anonimo.nome);               // "Anonimo"
```

Moduli (import/export)

I moduli JavaScript permettono di organizzare il codice in file separati, facilitando la manutenzione e il riutilizzo del codice. ES6 ha introdotto una sintassi standardizzata per i moduli.

```
1 // File: matematica.js
2
3 // Export nominati (singoli)
4 export const PI = 3.14159;
5
6 export function somma(a, b) {
7     return a + b;
8 }
9
10 // Export multipli
11 const sottrazione = (a, b) => a - b;
12 const moltiplicazione = (a, b) => a * b;
13
14 export { sottrazione, moltiplicazione };
15
16 // Export con rinomina
17 function divisione(a, b) {
18     return a / b;
19 }
20
21 export { divisione as dividi };
22
23 // Export default (uno solo per modulo)
24 export default function(x) {
25     return x * x;
26 }
```



```
1 // File: app.js
2
3 // Import singoli
4 import { somma, PI } from './matematica.js';
5 console.log(somma(2, 3));      // 5
6 console.log(PI);              // 3.14159
7
8 // Import multipli
9 import { sottrazione, moltiplicazione } from './matematica.js';
10
11 // Import con rinomina
12 import { dividi as divisione } from './matematica.js';
13 console.log(divisione(10, 2)); // 5
14
15 // Import default
16 import quadrato from './matematica.js';
17 console.log(quadrato(4));    // 16
18
19 // Import di tutto in un oggetto
20 import * as mat from './matematica.js';
21 console.log(mat.PI);         // 3.14159
22 console.log(mat.somma(5, 3)); // 8
23 console.log(mat.default(3)); // 9 (il default viene
                             // importato come 'default')
```

Esercizi pratici

Livello base

1. **Template literals avanzati:** Crea una funzione che formatta le informazioni di un prodotto (nome, prezzo, disponibilità) usando template literals e logica condizionale.
2. **Operatori spread e rest:** Scrivi una funzione che unisce due oggetti utente mantenendo tutte le proprietà, ma in caso di conflitto dà priorità al secondo oggetto. Poi, aggiungi una lista variabile di ruoli.



Esercizi pratici

Livello intermedio

1. **Classe con metodi avanzati:** Crea una classe `Carrello` che gestisce un carrello della spesa con metodi per aggiungere prodotti, rimuoverli, calcolare il totale e applicare sconti.
2. **Gestione moduli:** Crea un mini-sistema di gestione di una librerie con più moduli. Un modulo per la gestione dei libri e un modulo principale che lo utilizza.



Esercizi pratici

Livello difficile

1. **Sistema completo con classi e moduli:** Implementa un semplice sistema di gestione di un blog con classi per utenti, post e commenti, utilizzando tutte le funzionalità ES6+ viste nel modulo.



Agenda

Funzioni asincrone

In JavaScript, le operazioni asincrone sono essenziali per gestire attività che richiedono tempo, come il recupero di dati da un server, la lettura di file o l'interazione con API esterne, senza bloccare l'esecuzione del resto del codice.

- Callbacks
- Promises
- Async/await
- Fetch API e richieste HTTP

Callbacks

Le callback sono la forma più basilare di gestione dell'asincronicità in JavaScript. Una callback è una funzione che viene passata come argomento ad un'altra funzione e viene eseguita dopo che un'operazione asincrona è stata completata.

```
1 // Simulazione di un'operazione asincrona
2 function recuperaDati(id, callback) {
3     console.log(`Inizio recupero dati per ID: ${id}`);
4
5     // setTimeout simula un'operazione che richiede tempo (come
6     // una chiamata a un server)
6     setTimeout(() => {
7         const dati = { id: id, nome: "Prodotto " + id, prezzo:
8             Math.random() * 100 };
9
10        // Esegui la callback passando i dati recuperati
11        callback(dati);
11        }, 1500); // Simula un ritardo di 1.5 secondi
12
13        console.log("La funzione recuperaDati è tornata, ma i dati
13        non sono ancora pronti");
14    }
15
16 // Utilizzo della callback
17 recuperaDati(42, (risultato) => {
18     console.log("Dati ricevuti:", risultato);
19     console.log(`Il prodotto ${risultato.nome} costa
19     ${risultato.prezzo.toFixed(2)}€`);
20 });
21
22 // Questo codice viene eseguito immediatamente, non aspetta il
22 // completamento di recuperaDati
23 console.log("Continuazione del programma principale mentre i
23 dati vengono recuperati...");
```



```
1 Inizio recupero dati per ID: 42
2 La funzione recuperaDati è tornata, ma i dati non sono ancora
2 pronti
3 Continuazione del programma principale mentre i dati vengono
3 recuperati...
4 (Dopo 1.5 secondi)
5 Dati ricevuti: {id: 42, nome: "Prodotto 42", prezzo: 45.23}
6 Il prodotto Prodotto 42 costa 45.23€
```



Callbacks

Callback in funzioni standard della libreria JavaScript:

```
1 // Esempio con setTimeout
2 setTimeout(() => {
3     console.log("Questo messaggio apparirà dopo 2 secondi");
4 }, 2000);
5
6 // Esempio con gestione eventi
7 document.getElementById("mioBottone").addEventListener("click",
8     (evento) => {
9         console.log("Il bottone è stato cliccato!", evento);
10    });
11 // Esempio con metodi di array
12 [1, 2, 3].forEach(numero => {
13     console.log(`Elaborazione del numero: ${numero}`);
14 });


```

Il problema del "Callback Hell" (o "Pyramid of Doom"):

Quando abbiamo bisogno di eseguire più operazioni asincrone in sequenza, l'utilizzo di callback può portare a una struttura di codice profondamente annidata e difficile da leggere/mantenere:

```
1 recuperadati(1, (prodotto) => {
2     console.log("Prodotto trovato:", prodotto);
3
4     recuperadettagli(prodotto.id, (dettagli) => {
5         console.log("Dettagli recuperati:", dettagli);
6
7         recuperacommenti(prodotto.id, (commenti) => {
8             console.log("Commenti recuperati:", commenti);
9
10            recuperasuggerimenti(prodotto.categoria,
11                (suggerimenti) => {
12                    console.log("Suggerimenti trovati:",
13                        suggerimenti);
14
15                    // Il codice continua a innestarsi...
16                    }, (errore) => {
17                        console.error("Errore nei suggerimenti:",
18                            errore);
19                    });
20                }, (errore) => {
21                    console.error("Errore nei commenti:", errore);
22                });
23            }, (errore) => {
24                console.error("Errore nel recupero prodotto:", errore);
25            });

```

Questo codice diventa rapidamente difficile da leggere, debuggare e mantenere. Per risolvere questo problema, sono state introdotte le Promises.

Promises

Le Promises sono oggetti che rappresentano il risultato futuro di un'operazione asincrona.
Una Promise può trovarsi in uno dei seguenti stati:

- **pending**: stato iniziale, né completata né fallita
- **fulfilled**: operazione completata con successo
- **rejected**: operazione fallita

```
1 // Riscriviamo la funzione recuperaDati utilizzando una Promise
2 function recuperaDati(id) {
3     return new Promise((resolve, reject) => {
4         console.log(`Inizio recupero dati per ID: ${id}`);
5
6         // Simuliamo un'operazione asincrona
7         setTimeout(() => {
8             // Simulazione: se l'ID è negativo, fallisce
9             if (id < 0) {
10                 reject(new Error("ID non valido"));
11             } else {
12                 const dati = { id: id, nome: "Prodotto " + id,
13                               prezzo: Math.random() * 100 };
14                 resolve(dati); // Completamento con successo
15             }
16         }, 1500);
17     });
}
```



Promises

Utilizzo di una Promise con then/catch:

```
1 // Consumo della Promise
2 recuperati(42)
3     .then(resultato => {
4         console.log("Dati ricevuti:", risultato);
5         return risultato; // Possiamo passare dati alla
6             prossima then
7     })
8     .then(dati => {
9         console.log(`Il prodotto ${dati.nome} costa
10            ${dati.prezzo.toFixed(2)}€`);
11    })
12    .catch(error => {
13        console.error("Si è verificato un errore:",
14            errore.message);
15    })
16    .finally(() => {
17        console.log("Operazione completata (con successo o con
18            errore)");
19    });
20
21 console.log("Il programma principale continua
22 l'esecuzione...");
```



```
1 // Promise.all - attende che tutte le Promise siano risolte
2 const promiseArray = [
3     recuperati(1),
4     recuperati(2),
5     recuperati(3)
6 ];
7
8 Promise.all(promiseArray)
9     .then(resultati => {
10         console.log("Tutti i prodotti sono stati recuperati:",
11             risultati);
12         // risultati è un array con i risultati di ciascuna
13         // Promise, nello stesso ordine
14     })
15     .catch(error => {
16         console.error("Almeno una delle Promise è fallita:",
17             errore);
18         // Se anche una sola Promise fallisce, l'intera
19         // operazione fallisce
20     });
21
22 // Promise.race - restituisce la prima Promise che si risolve
23 // (o fallisce)
24 Promise.race([
25     new Promise(resolve => setTimeout(() => resolve("Prima"),
26         1000)),
27     new Promise(resolve => setTimeout(() => resolve("Seconda"),
28         500)),
29     new Promise(resolve => setTimeout(() => resolve("Terza"),
30         1500))
31 ])
32 .then(vincitore => {
33     console.log("La Promise più veloce è stata:", vincitore);
34     // Output: "Seconda"
35 });
```

Async/Await

Introdotto in ES2017, `async/await` è costruito sopra le Promises che rende il codice asincrono più leggibile e simile al codice sincrono.

```
1 // Una funzione async restituisce sempre una Promise
2 async function salutaDopoUnSecondo(nome) {
3     // Simuliamo un'operazione asincrona
4     await new Promise(resolve => setTimeout(resolve, 1000));
5
6     return `Ciao, ${nome}!`;
7 }
8
9 // Uso della funzione async
10 console.log("Prima della chiamata");
11
12 salutaDopoUnSecondo("Mario")
13     .then(messaggio => {
14         console.log(messaggio); // "Ciao, Mario!" (dopo 1
15             secondo)
16     });
17 console.log("Dopo la chiamata");
```



Async/Await

Introdotto in ES2017, async/await è costruito sopra le Promises che rende il codice asincrono più leggibile e simile al codice sincrono.

```
1 // La parola chiave await può essere usata solo all'interno di una funzione async
2 async function recuperaEProcessaDati() {
3     try {
4         console.log("Inizio recupero dati...");
5
6         // await "sospende" l'esecuzione della funzione fino alla risoluzione della Promise
7         const prodotto = await recuperaDati(42);
8         console.log("Prodotto ricevuto:", prodotto);
9
10        const dettagli = await recuperaDettagli(prodotto.id);
11        console.log("Dettagli ricevuti:", dettagli);
12
13        const commenti = await recuperaCommenti(prodotto.id);
14        console.log("Commenti ricevuti:", commenti);
15
16        console.log("Tutti i dati sono stati recuperati e processati!");
17        return { prodotto, dettagli, commenti };
18    } catch (errore) {
19        // Gestisce qualsiasi errore che si verifica nelle operazioni await
20        console.error("Si è verificato un errore:",
21                     errore.message);
22        throw errore; // Possiamo rilanciare l'errore o gestirlo diversamente
23    }
}
```



```
1 // Chiamiamo la funzione
2 recuperaEProcessaDati()
3     .then(risultati => {
4         console.log("Elaborazione completata:", risultati);
5     })
6     .catch(errore => {
7         console.error("La funzione ha generato un errore:",
8                     errore);
8 });

```

Async/Await

Elaborazione parallela con await:

```
1 async function recuperaDatiInParallelo() {
2     try {
3         console.log("Inizio recupero dati in parallelo...");
4
5         // Avviamo tutte le operazioni asincrone
5         contemporaneamente
6         const promiseProdotto = recuperaDati(1);
7         const promiseDettagli = recuperaDettagli(1);
8         const promiseCommenti = recuperaCommenti(1);
9
10        // Attendiamo che tutte siano complete
11        const prodotto = await promiseProdotto;
12        const dettagli = await promiseDettagli;
13        const commenti = await promiseCommenti;
14
15        // Alternativa usando Promise.all
16        // const [prodotto, dettagli, commenti] = await
17        //     Promise.all([
18        //         recuperaDati(1),
19        //         recuperaDettagli(1),
20        //         recuperaCommenti(1)
21        // ]);
22
23        return { prodotto, dettagli, commenti };
24    } catch (errore) {
25        console.error("Errore nel recupero parallelo:",
26        errore);
27        throw errore;
28    }
29}
```

Fetch API e richieste HTTP

La Fetch API è un'interfaccia JavaScript moderna per effettuare richieste HTTP. Restituisce Promises, quindi si integra perfettamente con il resto del codice asincrono.

```
1 fetch(url, [options])
2     .then(response => {
3         // response è un oggetto Response, non sono ancora i
4         dati
5         if (!response.ok) {
6             throw new Error(`Errore HTTP: ${response.status}`);
7         }
8         return response.json(); // Converte il corpo della
9         risposta in JSON
10    })
11    .then(data => {
12        console.log("Dati ricevuti:", data);
13    })
14    .catch(error => {
15        console.error("Errore nella richiesta:", error);
16    });

```



Fetch API e richieste HTTP

Esempio completo di una GET request:

```
1 fetch('https://jsonplaceholder.typicode.com/posts/1')
2   .then(response => {
3     if (!response.ok) {
4       throw new Error(`Errore HTTP: ${response.status}`);
5     }
6     return response.json();
7   })
8   .then(post => {
9     console.log("Post ricevuto:", post);
10  })
11  .catch(error => {
12    console.error("Errore nel recupero del post:", error);
13  });

```



Fetch API e richieste HTTP

POST request con fetch:

```
1 const nuovoPost = {
2     title: 'Titolo del post',
3     body: 'Contenuto del post',
4     userId: 1
5 };
6
7 fetch('https://jsonplaceholder.typicode.com/posts', {
8     method: 'POST',
9     headers: {
10         'Content-Type': 'application/json'
11     },
12     body: JSON.stringify(nuovoPost)
13 })
14 .then(response => {
15     if (!response.ok) {
16         throw new Error(`Errore HTTP: ${response.status}`);
17     }
18     return response.json();
19 })
20 .then(data => {
21     console.log("Post creato:", data);
22 })
23 .catch(error => {
24     console.error("Errore nella creazione del post:", error);
25 })
```

Fetch API e richieste HTTP

Fetch con `async/await`:

```
1 async function recuperaPost(id) {
2   try {
3     const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`);
4
5     if (!response.ok) {
6       throw new Error(`Errore HTTP: ${response.status}`);
7     }
8
9     const post = await response.json();
10    return post;
11  } catch (error) {
12    console.error("Errore nel recupero del post:", error);
13    throw error;
14  }
15 }
```

```
1 async function recuperaCommentiPost(postId) {
2   try {
3     const response = await fetch(`https://jsonplaceholder.typicode.com/posts/${postId}/comments`);
4
5     if (!response.ok) {
6       throw new Error(`Errore HTTP: ${response.status}`);
7     }
8
9     return await response.json();
10  } catch (error) {
11    console.error("Errore nel recupero dei commenti:", error);
12    throw error;
13  }
14 }
```

```
1 // Utilizzo
2 async function mostraPostECommenti(id) {
3   try {
4     const post = await recuperaPost(id);
5     console.log("Post:", post);
6
7     const commenti = await recuperaCommentiPost(id);
8     console.log(`Commenti (${commenti.length}):`, commenti);
9   } catch (error) {
10    console.error("Impossibile mostrare post e commenti:", error);
11  }
12 }
13
14 mostraPostECommenti(1);
```



Fetch API e richieste HTTP

Riepilogo

- **Callbacks:** La forma più basilare di gestione dell'asincronicità, ma può portare al "callback hell"
- **Promises:** Oggetti che rappresentano un risultato futuro, con metodi `then`, `catch` e `finally`
- **Async/Await:** Sintassi moderna basata su Promises che rende il codice asincrono più leggibile
- **Fetch API:** Interfaccia moderna per effettuare richieste HTTP che restituisce Promises



Esercizi pratici

Livello base

1. **Timer con Promises:** Crea una funzione `aspetta` che accetta un numero di millisecondi e restituisce una Promise che si risolve dopo quel tempo.
2. **Conversione da callback a Promise:** Prendi una funzione che utilizza callback e convertila in una funzione che restituisce una Promise.



Esercizi pratici

Livello intermedio

1. **Gestore di Richieste API:** Crea una classe che gestisce le richieste a un'API, implementando metodi get, post, put e delete, con gestione degli errori e possibilità di impostare headers predefiniti.
2. **Caricamento in Parallello con Limite:** Implementa una funzione che accetta un array di URL e un numero massimo di richieste parallele, quindi scarica tutti gli URL rispettando il limite di concorrenza.



Esercizi pratici

Livello difficile



— www.Duccio.ME —

Agenda

Manipolazione del DOM

Il DOM è la rappresentazione ad albero di tutti gli elementi di una pagina web, e JavaScript ci permette di interagire con esso per creare pagine dinamiche e interattive.

- Selettori e navigazione del DOM
- Creazione e modifica di elementi
- Eventi e gestione degli eventi

Selettori e navigazione del DOM

Il primo passo per manipolare il DOM è selezionare gli elementi con cui vogliamo interagire. JavaScript offre diversi metodi per farlo.

```
1 // Seleziona un elemento per ID
2 const titolo = document.getElementById('titolo-principale');
3 console.log(titolo); // <h1 id="titolo-principale">...</h1>
4
5 // Seleziona elementi per classe (restituisce una HTMLCollection)
6 const paragrafi = document.getElementsByClassName('paragrafo');
7 console.log(paragrafi.length); // Numero di elementi con classe "paragrafo"
8
9 // Seleziona elementi per nome del tag (restituisce una HTMLCollection)
10 const bottoni = document.getElementsByTagName('button');
11 console.log(bottoni[0]); // Primo bottone nella pagina
12
13 // querySelector - seleziona il primo elemento che corrisponde al selettore CSS
14 const primoLink = document.querySelector('a.link-esterno');
15 console.log(primoLink.href); // URL del primo link con classe "link-esterno"
16
17 // querySelectorAll - seleziona tutti gli elementi che corrispondono (restituisce una NodeList)
18 const tuttiILink = document.querySelectorAll('nav a');
19 tuttiILink.forEach(link => {
20     console.log(link.textContent);
21 });


```

Selettori e navigazione del DOM

Verificare proprietà e contenuto:

```
1 // Verifica se un elemento ha una certa classe
2 const hasClass = elemento.classList.contains('attivo');
3
4 // Ottenere il contenuto testuale
5 const testo = elemento.textContent; // Tutto il testo, inclusi gli elementi annidati
6 const soloTesto = elemento.innerText; // Solo il testo visibile
7
8 // Ottenere l'HTML interno
9 const html = elemento.innerHTML; // HTML all'interno dell'elemento
10
11 // Ottenere l'HTML completo dell'elemento
12 const outerHtml = elemento.outerHTML; // HTML dell'elemento e del suo contenuto
13
14 // Ottenere e impostare attributi
15 const src = immagine.getAttribute('src');
16 immagine.setAttribute('alt', 'Descrizione immagine');
17 immagine.removeAttribute('data-test');
18
19 // Verifica se un attributo esiste
20 const hasAttr = elemento.hasAttribute('href');
```



Creazione e modifica di elementi

JavaScript permette di creare nuovi elementi, modificare quelli esistenti e inserirli nel DOM, permettendo di aggiornare dinamicamente la pagina.

```
1 // Crea un nuovo elemento
2 const nuovoParagrafo = document.createElement('p');
3
4 // Aggiungi contenuto
5 nuovoParagrafo.textContent = 'Questo è un nuovo paragrafo.';
6
7 // Aggiungi attributi
8 nuovoParagrafo.setAttribute('class', 'paragrafo-dinamico');
9 // oppure
10 nuovoParagrafo.className = 'paragrafo-dinamico';
11
12 // Aggiungi stili
13 nuovoParagrafo.style.color = 'blue';
14 nuovoParagrafo.style.fontSize = '16px';
15 nuovoParagrafo.style.marginTop = '10px';
16
17 // Crea un elemento complesso
18 const nuovoLink = document.createElement('a');
19 nuovoLink.href = 'https://example.com';
20 nuovoLink.textContent = 'Visita Example';
21 nuovoLink.className = 'link-esterno';
22 nuovoLink.target = '_blank';
```



Creazione e modifica di elementi

Inserimento nel DOM:

```
1 // Seleziona l'elemento genitore
2 const contenitore = document.querySelector('.contenitore');
3
4 // Aggiungi alla fine del contenitore
5 contenitore.appendChild(nuovoParagrafo);
6
7 // Inserisci prima di un altro elemento
8 const riferimento = document.querySelector('.riferimento');
9 contenitore.insertBefore(nuovoLink, riferimento);
10
11 // Metodi più moderni
12 contenitore.append(nuovoParagrafo); // Aggiunge alla fine, accetta più argomenti e nodi di testo
13 contenitore.prepend(nuovoLink); // Aggiunge all'inizio
14 riferimento.before(nuovoParagrafo); // Inserisce prima dell'elemento di riferimento
15 riferimento.after(nuovoLink); // Inserisce dopo l'elemento di riferimento
```



Creazione e modifica di elementi

Modifica di elementi esistenti:

```
1 // Modifica il contenuto
2 const titolo = document.getElementById('titolo');
3 titolo.textContent = 'Nuovo titolo';
4
5 // Modifica l'HTML interno
6 const contenitore = document.querySelector('.contenitore');
7 contenitore.innerHTML = '<p>Nuovo contenuto</p><strong>Testo in grassetto</strong>';
8
9 // Attenzione: l'uso di innerHTML può creare rischi di sicurezza con input utente
10
11 // Modifica gli attributi
12 const immagine = document.querySelector('img');
13 immagine.src = 'nuova-immagine.jpg';
14 immagine.alt = 'Descrizione aggiornata';
15
16 // Gestione delle classi con classList
17 const elemento = document.querySelector('.elemento');
18 elemento.classList.add('evidenziato'); // Aggiungi classe
19 elemento.classList.remove('nascosto'); // Rimuovi classe
20 elemento.classList.toggle('attivo'); // Alterna classe (aggiunge se assente, rimuove se presente)
21 elemento.classList.replace('vecchia-classe', 'nuova-classe'); // Sostituisci classe
```



Creazione e modifica di elementi

Rimozione di elementi:

```
1 // Rimuovi un elemento
2 const elementoDaRimuovere = document.querySelector('.da-rimuovere');
3 elementoDaRimuovere.remove(); // Metodo moderno
4
5 // Rimuovi un elemento figlio
6 const genitore = document.querySelector('.genitore');
7 const figlio = document.querySelector('.figlio');
8 genitore.removeChild(figlio); // Metodo tradizionale
9
10 // Svuota un contenitore
11 contenitore.innerHTML = ''; // Modo rapido ma non ottimale per prestazioni
12
13 // Modo più performante per svuotare un contenitore
14 while (contenitore.firstChild) {
15     contenitore.removeChild(contenitore.firstChild);
16 }
```



Eventi e gestione degli eventi

Gli eventi permettono di rilevare e rispondere alle azioni dell'utente e ad altri cambiamenti nel browser.

Aggiunta di event listener:

```
1 // Seleziona un elemento
2 const bottone = document.querySelector('#mio-bottone');
3
4 // Aggiungi un event listener
5 bottone.addEventListener('click', function(evento) {
6     console.log('Il bottone è stato cliccato!');
7     console.log('Elemento target:', evento.target);
8 });
9
10 // Utilizzo di arrow function
11 bottone.addEventListener('click', (evento) => {
12     console.log('Click con arrow function');
13 });
14
15 // Rimozione di un event listener (deve usare la stessa funzione)
16 function gestoreClick(evento) {
17     console.log('Gestore click eseguito');
18 }
19
20 bottone.addEventListener('click', gestoreClick);
21 // Più avanti nel codice
22 bottone.removeEventListener('click', gestoreClick);
```



Eventi e gestione degli eventi

Tipi di eventi comuni:

```
1 // Eventi del mouse
2 elemento.addEventListener('click', handler); // Click del mouse
3 elemento.addEventListener('dblclick', handler); // Doppio click
4 elemento.addEventListener('mousedown', handler); // Pressione del pulsante del mouse
5 elemento.addEventListener('mouseup', handler); // Rilascio del pulsante del mouse
6 elemento.addEventListener('mousemove', handler); // Movimento del mouse
7 elemento.addEventListener('mouseover', handler); // Mouse che entra in un elemento
8 elemento.addEventListener('mouseout', handler); // Mouse che esce da un elemento
9
10 // Eventi della tastiera
11 documento.addEventListener('keydown', handler); // Tasto premuto
12 documento.addEventListener('keyup', handler); // Tasto rilasciato
13 documento.addEventListener('keypress', handler); // Tasto premuto (solo caratteri)
14
15 // Eventi dei form
16 form.addEventListener('submit', handler); // Invio del form
17 input.addEventListener('focus', handler); // Campo che ottiene il focus
18 input.addEventListener('blur', handler); // Campo che perde il focus
19 input.addEventListener('change', handler); // Valore del campo cambiato
20 input.addEventListener('input', handler); // Input in tempo reale
21
22 // Eventi del documento
23 window.addEventListener('load', handler); // Pagina e risorse caricate
24 document.addEventListener('DOMContentLoaded', handler); // DOM costruito (prima delle immagini)
25 window.addEventListener('resize', handler); // Finestra ridimensionata
26 window.addEventListener('scroll', handler); // Scroll della pagina
```



Eventi e gestione degli eventi

Oggetto evento e sue proprietà:

```
1 elemento.addEventListener('click', function(evento) {  
2     // Informazioni generali  
3     console.log(evento.type); // Tipo di evento (es. "click")  
4     console.log(evento.target); // Elemento su cui si è verificato l'evento  
5     console.log(evento.currentTarget); // Elemento a cui è collegato il listener (this)  
6     console.log(evento.timeStamp); // Timestamp dell'evento  
7  
8     // Per eventi del mouse  
9     console.log(evento.clientX, evento.clientY); // Coordinate rispetto alla finestra  
10    console.log(evento.pageX, evento.pageY); // Coordinate rispetto al documento  
11    console.log(evento.button); // Pulsante del mouse (0=sinistro, 1=centrale, 2=destro)  
12  
13    // Per eventi della tastiera  
14    console.log(evento.key); // Carattere premuto  
15    console.log(evento.keyCode); // Codice tasto (deprecato)  
16    console.log(evento.code); // Codice fisico del tasto  
17    console.log(evento.altKey, evento.ctrlKey, evento.shiftKey); // Modificatori  
18  
19    // Prevenire il comportamento predefinito  
20    evento.preventDefault();  
21  
22    // Fermare la propagazione dell'evento  
23    evento.stopPropagation();  
24});
```



Eventi e gestione degli eventi

Propagazione degli eventi (Event Bubbling):

```
1 // HTML: <div id="esterno"><div id="interno">Cliccami</div></div>
2
3 document.getElementById('esterno').addEventListener('click', function() {
4     console.log('Evento catturato dall\'elemento esterno (bubbling)');
5 });
6
7 document.getElementById('interno').addEventListener('click', function(evento) {
8     console.log('Evento originato dall\'elemento interno');
9     // evento.stopPropagation(); // Uncomment to stop the bubbling
10 });
11
12 // Fase di cattura (opposta al bubbling)
13 document.getElementById('esterno').addEventListener('click', function() {
14     console.log('Elemento esterno (fase di cattura)');
15 }, true); // Il terzo parametro true attiva la fase di cattura
```



Agenda

Ripasso di JavaScript Fondamentale

- Sintassi moderna e best practices
- Variabili, scope e closure
- Funzioni e paradigmi funzionali
- Oggetti in JavaScript
-

Framework e Applicazioni Pratiche

- Hooks nei framework moderni
- Routing e navigazione
- Esercitazione pratica

Sintassi Moderna e Best Practices

Dichiarazione di Variabili



```
1 // ES6+ best practices
2 const nome = 'Mario';           // Preferire const quando possibile
3 let età = 30;                  // let per variabili che cambiano
4 // Evitare var (problemi di scoping)
```

Template Literals



```
1 const messaggio = `Ciao ${nome}, hai ${età} anni`;
```

Funzioni e Paradigmi Funzionali

Arrow Functions

```
● ● ●  
1 // Funzione tradizionale  
2 function somma(a, b) {  
3   return a + b;  
4 }  
5  
6 // Arrow function equivalente  
7 const somma = (a, b) => a + b;
```

Funzioni di ordine superiore

```
● ● ●  
1 const numeri = [1, 2, 3, 4, 5];  
2  
3 // map, filter, reduce  
4 const doppi = numeri.map(n => n * 2);  
5 const pari = numeri.filter(n => n % 2 === 0);  
6 const somma = numeri.reduce((acc, n) => acc + n, 0);
```

Oggetti in JavaScript

Creazione e manipolazione di oggetti

```
● ○ ● ●  
1 // Oggetto letterale  
2 const persona = {  
3   nome: 'Laura',  
4   età: 28,  
5   saluta() {  
6     return `Ciao, sono ${this.nome}`;  
7   }  
8 };  
9  
10 // Accesso alle proprietà  
11 console.log(persona.nome);      // Dot notation  
12 console.log(persona['nome']);    // Bracket notation
```

Funzioni di ordine superiore

```
● ○ ● ●  
1 // Spread operator  
2 const personaEstesa = { ...persona, città: 'Milano' };  
3  
4 // Destructuring  
5 const { nome, età } = persona;
```

Prototype e Ereditarietà

```
1 function Persona(nome) {  
2   this.nome = nome;  
3 }  
4  
5 Persona.prototype.saluta = function() {  
6   return `Ciao, sono ${this.nome}`;  
7 };  
8  
9 const mario = new Persona('Mario');  
10 console.log(mario.saluta()); // "Ciao, sono Mario"
```

Classi ES6

```
● ● ●  
1 class Persona {  
2   constructor(nome) {  
3     this.nome = nome;  
4   }  
5  
6   saluta() {  
7     return `Ciao, sono ${this.nome}`;  
8   }  
9 }  
10  
11 class Impiegato extends Persona {  
12   constructor(nome, ruolo) {  
13     super(nome);  
14     this.ruolo = ruolo;  
15   }  
16  
17   presentati() {  
18     return `${this.saluta()}, lavoro come ${this.ruolo}`;  
19   }  
20 }
```

Aggiungere come passare da js a react

async await

Template 1

- Repository su GitHub
- Slide su Google drive e Github

Template 2

- Repository su GitHub
- Slide su Google drive e Github

- Altro Testo

Template 3

- Repository su GitHub
- Slide su Google drive e Github



— WWW.DUCKTALE.ME —