**Linguaggio di alto livello e leggibile**: JavaScript è un linguaggio di programmazione ad alto livello progettato per essere facilmente comprensibile dagli sviluppatori.

**Interpretato**: JavaScript è un linguaggio interpretato, il che significa che il codice viene eseguito direttamente senza un passaggio di compilazione.

**Tradotto a run-time**: Il codice JavaScript viene tradotto a run-time dagli engine JavaScript come V8, SpiderMonkey, Nitro, Chakra, che eseguono il codice nel browser o nell'ambiente di esecuzione.

**Errori a run-time**: Poiché non c'è un passaggio di compilazione, è facile commettere errori a run-time durante l'esecuzione del codice JavaScript.

**Supporto per tipi, operatori, oggetti e metodi**: JavaScript supporta vari tipi di dati, operatori per manipolarli, oggetti per organizzare i dati e metodi per eseguire operazioni su di essi.

**Inizialmente progettato per l'interattività delle pagine web:** JavaScript è stato originariamente creato per aggiungere interattività alle pagine web attraverso il DOM (Document Object Model), che consente di manipolare gli elementi HTML e CSS.

JavaScript moderno ha anche alcune caratteristiche aggiuntive:

- **Transpilers**: Il codice sorgente scritto in ES6+ può essere convertito in codice ECMAScript 5 compatibile con tutti i browser utilizzando un transpiler come Babel. Ciò consente di utilizzare le nuove funzionalità del linguaggio mantenendo la compatibilità con i browser più vecchi.
- **Utilizzo in browser e backend:** JavaScript, pur essendo nato come linguaggio per eseguire all'interno del browser, è stato esteso a diversi ambienti, come il backend, grazie a progetti come **Node.js**. Node.js consente di utilizzare JavaScript anche lato server.
- **Strumenti di sviluppo**: JavaScript dispone di un'ampia gamma di strumenti di sviluppo per semplificare il processo di sviluppo, tra cui package manager (come npm e yarn) per la gestione delle dipendenze, server HTTP locale per servire i file del progetto, transpiler per convertire il codice in una versione compatibile con i browser e strumenti come linter e test framework per il controllo della qualità del codice.
- **Gestione delle dipendenze con Node.js e npm**: Utilizzando Node.js e il suo package manager npm, è possibile specificare e gestire le dipendenze del progetto. Il file package.json viene utilizzato per elencare le dipendenze e le relative versioni e i pacchetti vengono scaricati e installati dalla community open source.

**JavaScript è un linguaggio di programmazione dinamico** in cui le variabili possono contenere qualsiasi tipo di dato e possono essere modificate nel tempo.

Le variabili dichiarate senza essere inizializzate sono di default "**undefined**".

In JavaScript, le variabili possono essere dichiarate utilizzando le seguenti parole chiave: "**var**" (con scope di funzione), "**let**" (con scope di blocco) e "**const**" (costante con scope di blocco).

Le variabili dichiarate con "var" hanno uno scope di funzione e possono essere visibili all'interno della funzione in cui sono definite o in blocchi interni. Se dichiarate al di fuori di qualsiasi funzione, diventano globali e vengono aggiunte all'oggetto "window".

Le variabili dichiarate con "let" hanno uno scope di blocco e sono visibili solo all'interno del blocco in cui sono definite, compresi blocchi o funzioni interne. Se definite al di fuori di qualsiasi funzione, diventano globali ma non vengono aggiunte all'oggetto "window".

Le variabili dichiarate con "const" hanno uno scope di blocco simile a "let" e il loro valore non può essere modificato. Tuttavia, se una variabile "const" contiene un oggetto, è possibile modificare i campi dell'oggetto.

JavaScript supporta il concetto di "**truthy**" e "**falsy**" values, in cui qualsiasi oggetto può essere valutato come "true" o "false" in un contesto booleano.

L'operatore ternario è un modo conciso per scrivere condizioni if-else in linea.

I template literals consentono di incorporare variabili ed espressioni all'interno di una stringa.

Gli oggetti in JavaScript possono essere creati utilizzando shorthand properties, computed property names e destructuring per estrarre proprietà da un oggetto o assegnare valori a nuove variabili.

È possibile creare copie superficiali (shallow copy) o copie profonde (deep copy) di un oggetto utilizzando diverse tecniche come Object.assign(), lo spread operator o JSON.stringify() seguito da JSON.parse().

Le funzioni in JavaScript possono essere definite utilizzando function declarations, function expressions o arrow functions. Le function declarations possono essere chiamate prima della loro definizione grazie al meccanismo di **hoisting**.

**JavaScript: Eventi**

- Le applicazioni web spesso reagiscono alle azioni dell'utente o ad altri eventi.
- Gli eventi possono essere scatenati da interazioni dell'utente, scambi di dati con server o modifiche della struttura e del comportamento dell'interfaccia.
- Quando un evento ha come destinazione l'oggetto Window o un altro oggetto autonomo, il browser risponde all'evento invocando gli handler appropriati su quell'oggetto.
- Quando il target dell'evento è un Document o un Element, la situazione è più complessa.
- È possibile intercettare un evento in tre modi: nel markup, aggiungendo la funzione direttamente all'elemento interessato o utilizzando addEventListener.
- La modalità di addEventListener è consigliata in quanto permette di aggiungere e rimuovere facilmente un numero arbitrario di event handlers e di specificare la fase di propagazione dell'evento.

**Interazione - Eventi e loro propagazione**

- Gli eventi seguono tre fasi: capturing, at target e bubbling.
- Nel **capturing**, l'evento si propaga dal contenitore verso i discendenti.
- Nella fase **at target**, l'evento è esattamente sull'oggetto su cui è stato richiesto.

- Nel **bubbling**, l'evento si propaga dal nodo su cui è stato agganciato fino alla radice del documento.
- Ogni handler può accedere alle proprietà dell'oggetto event, come event.target, event.currentTarget e event.eventPhase.
- Gli handlers possono fermare la propagazione dell'evento utilizzando event.stopPropagation(), anche se ciò non è raccomandato.

```
<div id="outer">
  <div id="inner">
    <button id="bon">Click me!</button>
  </div>
</div>
document.getElementById("outer").addEventListener("click", function() {
  console.log("Outer div clicked!");
}, true); // Capturing phase listener
document.getElementById("inner").addEventListener("click", function() {
  console.log("Inner div clicked!");
}, true); // Capturing phase listener
document.getElementById("btn").addEventListener("click", function() {
  console.log("Button clicked!");
}); // Bubbling phase listener (default)
// Output when clicking the button:
// Outer div clicked!
// Inner div clicked!
// Button clicked!
```

**Event delegation**

- Event delegation è una tecnica che permette di gestire gli eventi in modo efficiente associando un unico handler a un elemento padre comune invece di assegnare un handler a ogni elemento figlio.
- Ciò semplifica l'inizializzazione, risparmia memoria, richiede meno codice e facilita le modifiche al DOM.
- Tuttavia, la delegation richiede che gli eventi siano in fase di bubbling o capturing e può aggiungere un leggero carico alla CPU.

JavaScript: **Asincrono**

- JavaScript in un browser web è guidato dagli eventi, mentre sui server basati su JS, come Node.js, si attendono le richieste dei client o le risposte dai database.
- La programmazione asincrona è comune in JS per essere efficienti in un'architettura mono thread.
- La programmazione asincrona può essere eseguita utilizzando callbacks, ma il callback hell può rendere il codice difficile da gestire.
- Le Promises sono un modo per chiamare il codice di lunga esecuzione e attendere il ritorno di un risultato senza cadere nel callback hell.
- Una Promise rappresenta il risultato di un calcolo asincrono e può essere in uno dei tre stati: **pending**, **fulfilled** o **rejected**.
- Le Promises consentono di gestire gli errori in modo standardizzato e di esprimere le callback annidate in una catena lineare di Promises.
- La funzione fetch utilizza Promises per gestire le richieste asincrone.

JavaScript: async/await

L'uso di async/await è un approccio più leggibile e intuitivo per la gestione del codice asincrono in JavaScript. Ecco alcuni punti chiave:

- Dichiarare una funzione con la parola chiave `**async**` indica che il valore restituito dalla funzione sarà sempre una Promise, anche se nel corpo della funzione non è presente codice relativo a una Promise esplicita.
- L'uso di `await` all'interno di una funzione `async` sospende l'esecuzione della funzione finché la Promise non viene risolta. Durante questo periodo di attesa, il controllo passa ad altre attività.
- È possibile annidare espressioni `await` con funzioni `async` a piacimento, senza alcun limite specifico. Ciò consente di gestire facilmente catene complesse di operazioni asincrone.
- Tuttavia, se ci si trova all'interno di una funzione non `async`, non è possibile utilizzare l'operatore `await`, poiché non si è in grado di gestire direttamente la Promise ritornata. In questo caso, è necessario lavorare con la Promise ritornata utilizzando metodi come `.then()` o `.catch()` per gestire i risultati o gli errori.

L'utilizzo di async/await semplifica la scrittura e la lettura del codice asincrono, eliminando la necessità di gestire esplicitamente le callback e consentendo di scrivere il flusso del programma in modo più lineare e simile al codice sincrono.

**Array Methods: map, filter, reduce:**
Gli array methods come map, filter e reduce offrono modi potenti per manipolare gli array. Il metodo "map" crea un nuovo array applicando una funzione a ogni elemento dell'array originale. Il metodo "filter" genera un nuovo array contenente solo gli elementi che soddisfano una condizione specificata. Il metodo "reduce" combina gli elementi di un array in un unico valore utilizzando una funzione di riduzione.

```
// Array methods: map, filter, reduce
var numbers = [1, 2, 3, 4, 5];
// map: raddoppia ogni numero
var doubled = numbers.map((num) => {
  return num * 2;
});
console.log(doubled); // Output: [2, 4, 6, 8, 10]
// filter: filtra solo i numeri pari
var evenNumbers = numbers.filter((num) => {
  return num % 2 === 0;
});
console.log(evenNumbers); // Output: [2, 4]
// reduce: somma tutti i numeri
var sum = numbers.reduce((accumulator, current) => {
  return accumulator + current;
});
console.log(sum); // Output: 15
```

**Eventi**: Le applicazioni web spesso reagiscono alle azioni dell'utente o ad altri avvenimenti. Possono scambiare dati con i server e modificare la struttura, i contenuti e il comportamento dell'interfaccia in base a tali azioni. Gli eventi in JavaScript possono essere gestiti su oggetti Window o su Document ed Element, ma la gestione sugli ultimi due è più complessa.

```
// Gestione di un evento su un elemento
document.querySelector('div').addEventListener('click', function() {
  alert('Hello!');
});
// Gestione di un evento nel markup (bad practice)
<div onclick="alert('Hello!')"></div>
```

JavaScript: **Moduli**

Nel mondo dello sviluppo software, è fondamentale organizzare il codice in modo efficiente per favorire il riuso, la manutenibilità e la testabilità. Anche in JavaScript, esistono approcci per ottenere una buona organizzazione del codice.

- **Organizzazione del codice**: Per organizzare il codice in modo efficace, è consigliabile utilizzare funzioni come base. Tuttavia, se si desidera condividere o riutilizzare funzioni e variabili in altre parti del codice, è necessario un meccanismo per farlo.
- **Approcci storici:** Prima dell'introduzione dei moduli nativi in ES6, esistevano diversi approcci per simulare i moduli in JavaScript. Alcuni di questi includono l'uso di import globali tramite l'oggetto globale "window" e l'uso del Module Pattern, un pattern di software engineering.
- **Global Imports:** Questo approccio coinvolge l'aggancio di oggetti e funzioni all'oggetto globale. Tuttavia, ciò non garantisce che un altro script non possa modificare o sovrascrivere gli elementi definiti dal modulo. Inoltre, è necessario importare gli script nell'ordine corretto. Pertanto, questo approccio è da evitare.
- **Module Pattern**: Un approccio comune per ottenere la modularità in JavaScript è sfruttare il Module Pattern. Questo pattern sfrutta le Immediately Invoked Function Expression (IIFE) per creare un contesto privato e isolato per le funzioni e le variabili del modulo. Esistono molte varianti del Module Pattern e può essere utile per mantenere la privacy e impedire la contaminazione dello spazio dei nomi globale.
- **Moduli ES nativi**: A partire da ES6, JavaScript ha introdotto i moduli nativi. Questi moduli eliminano la necessità di simulare il comportamento dei moduli utilizzando approcci come il Module Pattern. I moduli ES hanno alcune caratteristiche chiave, come la definizione di un modulo all'interno di ogni file con un contesto privato, la possibilità di importare ed esportare esplicitamente elementi da altri moduli e l'utilizzo della modalità strict.
- **Esportazione di moduli**: Un modulo può esportare elementi utilizzando la parola chiave `export`. Gli elementi esportati possono essere distinti utilizzando il loro nome (named exports). È possibile definire anche più export in un unico modulo.
- **Importazione di moduli:** Un modulo può importare elementi da altri moduli utilizzando la parola chiave `import`. È possibile specificare i nomi degli elementi da importare e il percorso relativo al modulo. L'uso di `*` e la keyword `as` consentono di importare l'intero modulo e fare riferimento ai named exports utilizzando la notazione delle proprietà.
- **Export Default**: Un modulo può definire un export default, che semplifica la sintassi di importazione. È possibile avere sia export di default che named exports contemporaneamente nel modulo.
- **Importazione di Default:** L'importazione di un elemento esportato con default può essere fatta utilizzando la sintassi `import someName
- from 'myModule'` o `import { default as someName } from 'myModule'`, dove `someName` è un nome arbitrario.
- **Rinominare gli import:** È possibile specificare un nome alternativo per gli elementi importati utilizzando la keyword `as`.

I moduli JavaScript offrono un'organizzazione più chiara e una gestione migliore del codice. Con l'introduzione dei moduli nativi in ES6, è possibile sfruttare le loro caratteristiche per creare applicazioni più modulari e manutenibili. L'utilizzo di un module bundler come webpack o parcel può semplificare la combinazione dei moduli in un singolo file per il deployment.

**MV* sta per Model-View-*.** È un termine usato per riferirsi ai framework che implementano una delle numerose varianti dell'architettura Model-View. Alcuni esempi comuni sono MVC (Model-View-Controller), MVVM (Model-View-ViewModel) e MVP (Model-View-Presenter). La maggior parte dei framework front-end segue il pattern MVVM.

La programmazione imperativa è un paradigma di programmazione che si concentra sulla descrizione di come eseguire un compito, specificando le istruzioni passo-passo per raggiungere il risultato desiderato. La programmazione dichiarativa, invece, si concentra maggiormente su ciò che deve essere fatto, senza specificare i passaggi esatti per ottenerlo.

Framework come React, Ember, Angular e Vue seguono un approccio dichiarativo. Consentono agli sviluppatori di descrivere lo stato desiderato dell'applicazione e il framework si occupa di aggiornare l'interfaccia utente di conseguenza. Questo approccio offre diversi vantaggi, tra cui una manutenzione più semplice, la testabilità e la scalabilità rispetto alla programmazione imperativa.

Gli esempi forniti dimostrano la differenza tra programmazione imperativa e dichiarativa. L'esempio imperativo manipola direttamente il DOM utilizzando codice JavaScript, mentre l'esempio dichiarativo di React definisce l'aspetto dell'interfaccia utente in base allo stato dell'applicazione, senza manipolare direttamente il DOM.

**MVVM (Model-View-ViewModel)** è una variante del pattern MVC adattata alle moderne piattaforme di sviluppo dell'interfaccia utente, dove la vista è spesso responsabilità del progettista. In MVVM, il **Modello** rappresenta i dati e la business logic, la **View** rappresenta gli elementi dell'interfaccia utente e il **ViewModel** funge da intermediario tra la **Vista** e il **Modello**. Il **ViewModel** fornisce dati e comandi alla View e gestisce le interazioni e gli aggiornamenti della View.

**Il data binding** è una caratteristica comunemente utilizzata nei framework MVVM. Consente la sincronizzazione automatica dei dati tra la Vista e il ViewModel. Il data binding unidirezionale propaga le modifiche dal Modello/ViewModel alla Vista, mentre il data binding bidirezionale consente aggiornamenti immediati in entrambe le direzioni.

Le proprietà e lo stato sono concetti utilizzati nei framework MVVM per gestire i dati nei componenti. Le proprietà (**props**) sono dati esterni che un componente deve rendere, mentre lo **stato** rappresenta i dati interni che possono essere modificati e che influenzano il rendering. I framework forniscono meccanismi per definire e aggiornare le proprietà e lo stato all'interno dei componenti.

**Props**=immutabili **State**=mutabili.

Componente padre non può modificare **state** dei figli, comunicano tramite props.

Il DOM virtuale è una tecnica utilizzata da molti framework di front-end per migliorare le prestazioni, evitando la manipolazione diretta del DOM reale. Al contrario, viene creata una rappresentazione virtuale del DOM, che viene aggiornata in base alle modifiche allo stato dell'applicazione. Il framework

```
function taggato(tagName) {
  return function (strings, ...values) {
    const html = strings.map((string, index) => {
      const value = values[index] ? values[index] : '';
      return string + value;
    }).join('');

    return `<${tagName}>${html}</${tagName}>`;
  };
}

const saluta = taggato('p');
const titoloTesto = taggato('h1');

const nome = 'Marco';
const risultatoSaluto = saluta`Ciao, ${nome}!`;

const testo = 'Benvenuti nel mio sito!';
const risultatoTitolo = titoloTesto`${testo}`;

console.log(risultatoSaluto);
console.log(risultatoTitolo);
```

calcola quindi le differenze tra il DOM virtuale attuale e quello precedente e aggiorna il DOM reale di conseguenza, applicando solo le modifiche necessarie.

I template literals, noti anche come stringhe template, sono una caratteristica di JavaScript che consente un'interpolazione delle stringhe più espressiva e flessibile. Sono racchiusi da backtick ( ) invece che da virgolette singole o doppie e possono contenere segnaposto indicati dal segno del dollaro e dalle parentesi graffe (${espressione}). I letterali di template supportano stringhe multilinea, interpolazione delle espressioni e template etichettati.

I template taggati sono una caratteristica avanzata dei template letterali che consente di analizzarli con una funzione personalizzata. Ciò può essere utile per manipolare e trasformare i template letterali prima che vengano interpolati. I template taggati vengono invocati anteponendo il nome di una funzione al template letterale, seguito da parentesi.

**La porta 5001 TCP è diversa dalla porta 5001 UDP**

http è un protocollo stateless

**un sistema deve essere scalabile**

**i socket consentono la connessione tra due host, un socket è formato da ip:porta**

Il protocollo HTTP (Hypertext Transfer Protocol) e i suoi componenti chiave come gli URL (Uniform Resource Locator), le porte, le risorse e i protocolli correlati. Si sottolinea che il package java.net in Java fornisce le classi necessarie per la programmazione di rete.

la classe URL in Java permette di creare connessioni a indirizzi URL. Si elencano i vari costruttori disponibili e si sottolinea l'importanza di gestire le eccezioni che possono verificarsi durante l'utilizzo delle classi URL.

La classe URLConnection offre un controllo più avanzato sulla connessione e permette di leggere informazioni come il tipo e la lunghezza del documento collegato all'URL. Viene menzionato che questa classe può essere sottoclasse di **HttpURLConnection**, se l'URL rappresenta una connessione HTTP.

Viene introdotto il concetto di stream di input/output, che permette di inviare e ricevere dati attraverso una connessione. Si presentano le classi del package java.io come InputStream, OutputStream, Reader e Writer, che consentono di gestire l'invio e la ricezione di dati.

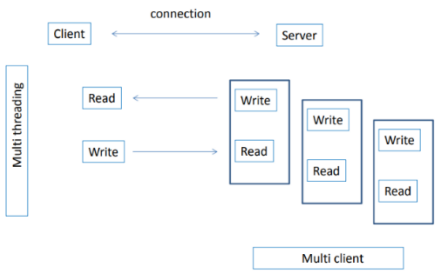
Si spiega come scrivere dati su un URL utilizzando la classe URLConnection e come leggere dati da un URL utilizzando il metodo openStream(). Viene fornito un esempio di codice che legge il contenuto di un URL e lo stampa a schermo.

Il testo approfondisce il funzionamento del protocollo HTTP, che è "stateless" (senza stato) e permette di richiedere e ottenere informazioni in modo rapido. Si descrive il ruolo del server web nel fornire i file richiesti e si menziona che il browser agisce come un client.

Si spiega il processo di richiesta e risposta HTTP, evidenziando che una richiesta viene inviata dal client al server e il server risponde con i dati richiesti. Viene introdotta la classe HttpURLConnection in Java, che offre supporto per le caratteristiche specifiche dell'HTTP.

I socket utilizzano il protocollo TCP (o in forma minore UDP) per creare connessioni su Internet.

Nel linguaggio di programmazione Java, il package java.net fornisce le classi Socket e ServerSocket per l'utilizzo dei socket. Socket viene utilizzata per creare una connessione tra il client e il server, mentre ServerSocket viene utilizzata solo dal server per "ascoltare" le richieste di connessione.




Un'applicazione client-server richiede che uno dei due rimanga in attesa di connessioni. Questo ruolo è svolto dal server, che viene avviato automaticamente all'avvio del sistema operativo.

Un client comunica con un server inviando una richiesta e attendendo una risposta. Per stabilire la connessione, il client crea un oggetto Socket e crea gli stream di input e output associati ad esso.

Per la comunicazione, vengono utilizzati gli stream di input e output del socket. È possibile leggere i dati in arrivo dallo stream di input e scrivere dati da inviare all'altra parte utilizzando lo stream di output.

La classe ServerSocket viene utilizzata dal server per accettare connessioni dai client. Quando un client si collega, viene creato un nuovo oggetto Socket per comunicare con esso.

Per gestire più connessioni contemporaneamente, è necessario utilizzare strutture multithreading. Sono necessari thread separati per gestire le richieste dei client in modo concorrente.

si utilizzano 2 thread uno di lettura e uno di scrittura

Le socket UDP sono utilizzate per la comunicazione di dati tramite il protocollo UDP, che è un protocollo di trasporto senza connessione. A differenza del protocollo TCP (Transmission Control Protocol), che è orientato alla connessione e garantisce la consegna dei dati in ordine e senza errori, UDP non garantisce la consegna dei dati e non gestisce l'ordine di consegna. Tuttavia, UDP è noto per la sua maggiore velocità rispetto a TCP.

Nel contesto di Java, il pacchetto java.net fornisce le classi DatagramPacket e DatagramSocket per supportare la comunicazione UDP. Il DatagramPacket rappresenta un pacchetto di dati che viene inviato o ricevuto tramite la rete, mentre il DatagramSocket viene utilizzato per inviare e ricevere DatagramPacket.

Per inviare un pacchetto tramite DatagramSocket, puoi creare un oggetto DatagramPacket con i dati da inviare e l'indirizzo e la porta di destinazione, quindi utilizzare il metodo send() di DatagramSocket. Per ricevere un pacchetto, puoi creare un oggetto DatagramPacket vuoto e passarlo al metodo receive() di DatagramSocket**. Il programma WhatsApp sul mio telefono invia la fotografia al server di WhatsApp, il server invia la foto al mio contatto appena quest'ultimo è disponibile**.