Document Object Model

Cos'è?

Il Document Object Model (DOM) è una rappresentazione gerarchica dei documenti HTML, XHTML o XML di una pagina web.

Il DOM definisce la struttura di un documento e la relativa rappresentazione come un albero di oggetti, dove ogni nodo rappresenta un'entità all'interno del documento, come elementi HTML, attributi, testo e così via.

Il DOM è essenziale per la manipolazione e la gestione dinamica dei contenuti web attraverso scripting, consentendo agli sviluppatori web di accedere, modificare e aggiornare dinamicamente il contenuto, la struttura e lo stile di una pagina web utilizzando linguaggi di scripting come JavaScript.

Concetti chiave:

1. **Nodi**: I nodi sono gli elementi costitutivi del DOM. Ci sono diversi tipi di nodi, inclusi nodi elemento, nodi testo, nodi attributo, nodi commento, ecc.
2. **Albero di nodi**: Il DOM organizza i nodi in una struttura gerarchica ad albero, con un nodo radice che rappresenta il documento nel suo complesso e nodi figlio che rappresentano gli elementi contenuti all'interno del documento.
3. **Accesso e manipolazione**: Gli sviluppatori possono accedere ai nodi del DOM utilizzando metodi e proprietà JavaScript, come `getElementById`, `querySelector`, `innerHTML`, `appendChild`, ecc. Questo consente loro di modificare dinamicamente il contenuto della pagina, aggiungere nuovi elementi, rimuovere elementi esistenti, modificare attributi e molto altro ancora.
4. **Eventi**: Il DOM supporta la gestione degli eventi, consentendo agli sviluppatori di associare azioni a eventi come il clic del mouse, il caricamento della pagina, l'inserimento di testo, ecc. Questo consente di creare interazioni dinamiche e reattive sulla pagina web.

5. **Rendering**: Il DOM è utilizzato dai browser per generare la rappresentazione visiva di una pagina web. Quando un documento HTML viene caricato in un browser, il browser crea un albero DOM corrispondente al documento e lo utilizza per generare l'output visivo sulla schermata.

Esempio di DOM

Supponiamo di avere il seguente codice html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-s
    <title>Esempio DOM</title>
</head>
<body>
    <header>
        <h1>Benvenuti!</h1>
    </header>
    <nav>
        <ul>
            <li><a href="#">Home</a></li>
            <li><a href="#">Chi siamo</a></li>
            <li><a href="#">Contatti</a></li>
        </ul>
    </nav>
    <main>
        <section>
            <h2>Chi siamo</h2>
            <p>Benvenuti sul nostro sito web! Siamo un'azienda c
        </section>
        <section>
            <h2>Ultimi articoli</h2>
            <article>
```

```
                <h3>Titolo articolo 1</h3>
                <p>Contenuto dell'articolo 1...</p>
            </article>
            <article>
                <h3>Titolo articolo 2</h3>
                <p>Contenuto dell'articolo 2...</p>
            </article>
        </section>
    </main>
    <footer>
        <p>&copy; 2024 Esempio DOM</p>
    </footer>
</body>
</html>
```

Questo codice HTML rappresenta una pagina web di base con un'intestazione, una barra di navigazione, un contenuto principale e un piè di pagina.

Ora, possiamo rappresentare la struttura DOM corrispondente a questo codice HTML in forma di albero:

```
- Document (Documento)
  - html
    - head
      - meta
      - meta
      - title
    - body
      - header
        - h1
      - nav
        - ul
          - li
            - a
          - li
            - a
```

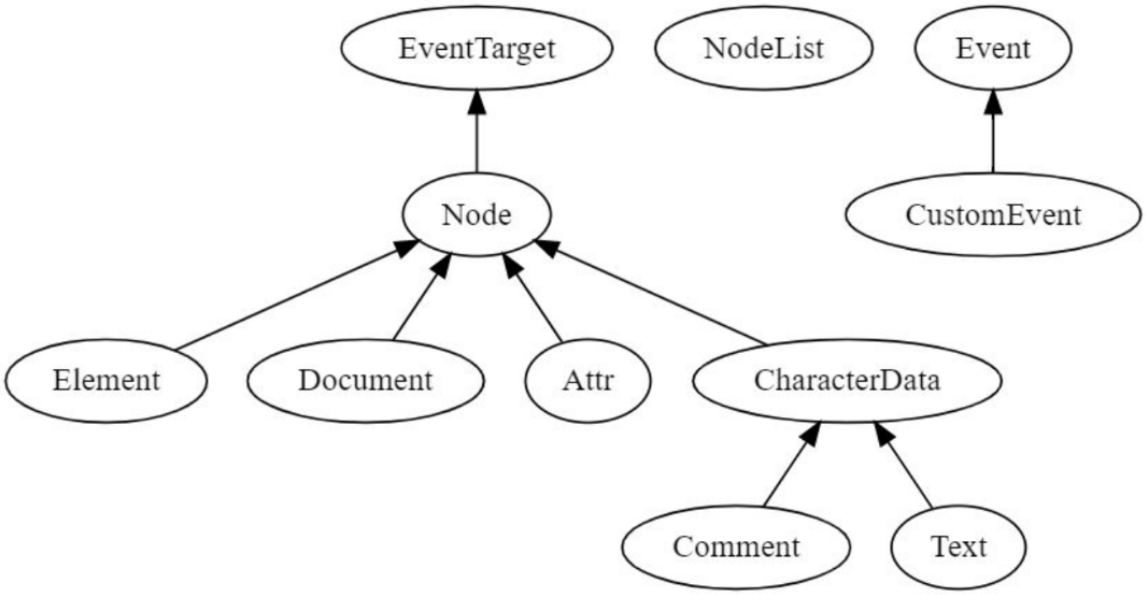
```
          - li
            - a
      - main
        - section
          - h2
          - p
        - section
          - h2
          - article
            - h3
            - p
          - article
            - h3
            - p
      - footer
        - p
```

Come mai è importante?

1. **Manipolazione dinamica del contenuto**: Il DOM consente agli sviluppatori di manipolare il contenuto HTML di una pagina in modo dinamico utilizzando JavaScript. Questo significa che è possibile aggiungere, rimuovere o modificare elementi HTML, attributi e contenuto testuale in risposta a eventi utente o a determinate condizioni.
2. **Interazione utente**: Grazie al DOM, è possibile rendere le pagine web interattive e reattive agli input dell'utente. Ad esempio, è possibile aggiungere eventi come clic del mouse, pressione dei tasti e hover sui vari elementi della pagina e reagire a tali eventi modificando dinamicamente il DOM.
3. **Creazione di interfacce utente complesse**: Il DOM consente di creare interfacce utente complesse e dinamiche, come applicazioni web, giochi e strumenti di produttività. Utilizzando JavaScript per manipolare il DOM, è possibile creare esperienze utente ricche e coinvolgenti direttamente nel browser.

4. **Rendering dei contenuti web**: I browser utilizzano il DOM per rappresentare e renderizzare i contenuti HTML di una pagina web. Il DOM funge da modello dati per il rendering dei contenuti, determinando la struttura, la disposizione e lo stile degli elementi sulla pagina.
5. **Compatibilità e accessibilità**: Utilizzando il DOM in modo appropriato, è possibile garantire che le pagine web siano compatibili con una vasta gamma di dispositivi e browser, nonché accessibili a utenti con disabilità utilizzando tecnologie assistive come lettori di schermo.

Interfacce e classi principali



Event

Sono azioni o situazioni che si verificano durante l'interazioni dell'utente con una pagina web. Questi eventi possono essere generati da varie azioni dell'utente, come click del mouse, pressione dei tast, movimenti del mouse ecc

1. **Tipi di eventi**: Esistono molti tipi di eventi nel DOM, ognuno corrispondente a un'azione specifica. Alcuni esempi comuni includono `click`, `mouseover`, `keydown`, `submit`, `load`, `change`, `scroll` e così via.
2. **Gestione degli eventi**: Per rispondere agli eventi, è possibile utilizzare il metodo `addEventListener()` o le proprietà degli eventi direttamente sugli elementi del DOM. Questo consente di associare una funzione (detta "gestore degli eventi" o "event listener") a un evento specifico. Quando l'evento si verifica, la funzione associata viene eseguita.
3. **Event object**: Quando un evento viene generato, viene creato un oggetto evento che contiene informazioni sull'evento stesso, come il tipo di evento, l'elemento su cui si è verificato l'evento e altre informazioni pertinenti. Questo oggetto evento viene passato come argomento alla funzione gestore dell'evento.
4. **Propagation e bubbling degli eventi**: Gli eventi nel DOM seguono il modello di propagazione degli eventi, che può essere "bubbling" (bolle) o "capturing" (cattura). Il bubbling è il processo in cui un evento generato su un elemento genitore viene propagato verso il basso attraverso i suoi elementi figlio, mentre la cattura è il processo inverso, in cui l'evento viene propagato dall'elemento genitore verso l'alto attraverso i suoi elementi genitori.
5. **Prevenzione degli eventi predefiniti**: È possibile prevenire il comportamento predefinito di un evento utilizzando il metodo `preventDefault()`. Ad esempio, questo può essere utile per impedire che un modulo venga inviato quando viene premuto il pulsante "invia".

EventTarget

`EventTarget` è un'interfaccia nel Document Object Model (DOM) che rappresenta oggetti che possono essere bersagli per eventi e possono ricevere eventi. È l'interfaccia base per tutti gli oggetti che possono essere bersagli per eventi, inclusi gli elementi del DOM, i documenti e gli oggetti di finestra.

1. **Gerarchia degli oggetti nel DOM**: `EventTarget` è l'interfaccia base per tutti gli oggetti che possono ricevere e gestire eventi nel DOM. Gli elementi HTML, i

documenti e gli oggetti di finestra sono tutti esempi di oggetti che implementano l'interfaccia `EventTarget`.

2. **Metodi per la gestione degli eventi**: Gli oggetti che implementano `EventTarget` forniscono metodi per la registrazione e la rimozione di gestori di eventi. I metodi principali sono `addEventListener()` e `removeEventListener()`, che consentono di associare e rimuovere gestori di eventi a un oggetto `EventTarget`.
3. **Disponibilità di eventi**: Gli oggetti `EventTarget` possono generare e ricevere una vasta gamma di eventi, tra cui eventi standard come `click`, `mouseover`, `keydown`, `submit` e molti altri. Questi eventi possono essere generati dall'utente (ad esempio, un clic del mouse) o possono essere generati programmaticamente (ad esempio, con `dispatchEvent()`).
4. **Event bubbling e capturing**: Gli oggetti `EventTarget` partecipano alla propagazione degli eventi nel DOM, che può avvenire in due modi: bubbling e capturing. Il bubbling è il processo in cui un evento viene propagato dai figli verso il genitore fino all'elemento radice, mentre il capturing è il processo inverso, in cui l'evento viene propagato dall'elemento radice fino al bersaglio dell'evento.

`addEventListener()`

Metodo utilizzato per associare un gestore di eventi a un oggetto `EventTarget`, come un elemento HTML, un documento o un oggetto di finestra. Questo metodo consente di specificare il tipo di evento da gestire (come "click", "mouseover", "keydown", ecc.) e la funzione da eseguire quando l'evento si verifica.

```
target.addEventListener(type, listener [, options]);
```

- `target`: L'oggetto `EventTarget` al quale associare il gestore di eventi.
- `type`: Una stringa che specifica il tipo di evento da gestire, come "click", "mouseover", "keydown", ecc.
- `listener`: La funzione da eseguire quando l'evento si verifica.
- `options` (opzionale): Un oggetto che specifica opzioni aggiuntive per la gestione degli eventi, come la modalità di propagazione degli eventi (bubbling

o capturing).

esempio di utilizzo del metodo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale
<title>addEventListener Example</title>
</head>
<body>

<button id="myButton">Clicca qui</button>

<script>
// Otteniamo il riferimento al pulsante
const myButton = document.getElementById('myButton');

// Definiamo la funzione da eseguire quando viene cliccato il pu
function handleClick(event) {
  console.log('Hai cliccato il pulsante!');
}

// Associamo il gestore di eventi alla pressione del pulsante
myButton.addEventListener('click', handleClick);
</script>

</body>
</html>
```

In questo esempio, abbiamo un semplice pulsante HTML con un ID "myButton". Utilizziamo JavaScript per ottenere il riferimento a questo pulsante e definiamo una funzione di gestione `handleClick()` che verrà eseguita quando il pulsante viene

cliccato. Successivamente, utilizziamo `addEventListener()` per associare questa funzione al pulsante, in modo che venga eseguita ogni volta che viene cliccato.

`removeEventListener()`

Metodo utilizzato per rimuovere un gestore di eventi precedentemente associato a un oggetto `EventTarget` tramite il metodo `addEventListener()`. Questo metodo è particolarmente utile quando si desidera interrompere la gestione di un evento dopo averla associata.

```
target.removeEventListener(type, listener [, options]);
```

- `target`: L'oggetto `EventTarget` dal quale rimuovere il gestore di eventi.
- `type`: Una stringa che specifica il tipo di evento per il quale rimuovere il gestore.
- `listener`: Il gestore di eventi da rimuovere.
- `options` (opzionale): Un oggetto che specifica opzioni aggiuntive per la rimozione degli eventi.

È importante notare che il gestore di eventi specificato come `listener` deve essere lo stesso oggetto funzione che è stato originariamente passato a `addEventListener()`. Se il gestore di eventi è stato associato utilizzando una funzione anonima, è necessario utilizzare la stessa funzione anonima per rimuovere il gestore.

`dispatchEvent()`

Metodo utilizzato per innescare manualmente un evento su un oggetto `EventTarget`. Questo metodo consente di creare e inviare un evento personalizzato tramite codice JavaScript.

```
target.dispatchEvent(event);
```

- `target`: L'oggetto `EventTarget` su cui innescare l'evento.
- `event`: L'oggetto evento da inviare. Questo deve essere un oggetto evento valido, creato tramite il costruttore `Event()` o uno dei suoi costruttori specializzati, come `MouseEvent()`, `KeyboardEvent()`, ecc.

Quando `dispatchEvent()` viene chiamato, l'evento specificato viene propagato e gestito secondo le regole di propagazione degli eventi del DOM. Questo significa che se ci sono gestori di eventi registrati per l'evento specificato sull'oggetto `EventTarget`, verranno eseguiti in risposta all'evento inviato.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale
<title>dispatchEvent Example</title>
</head>
<body>

<button id="myButton">Clicca qui</button>

<script>
// Otteniamo il riferimento al pulsante
const myButton = document.getElementById('myButton');

// Definiamo la funzione di gestione dell'evento
function handleClick(event) {
  console.log('Hai cliccato il pulsante!');
}

// Associamo il gestore di eventi alla pressione del pulsante
myButton.addEventListener('click', handleClick);
```

```
// Creiamo un nuovo evento di click
const clickEvent = new Event('click');

// Inviamo manualmente l'evento di click
myButton.dispatchEvent(clickEvent);
</script>

</body>
</html>
```

In questo esempio, creiamo un nuovo oggetto evento di click utilizzando il costruttore `Event('click')`. Successivamente, utilizziamo `dispatchEvent()` per inviare manualmente questo evento al pulsante. Poiché abbiamo associato un gestore di eventi di click al pulsante, la funzione `handleClick()` verrà eseguita immediatamente in risposta all'evento di click inviato

Custom events

Sono eventi definiti dall'utente che possono essere creati, innescati e gestiti in modo simile agli eventi standard del Document Object Model (DOM). Consentono agli sviluppatori di creare e utilizzare eventi specifici per le esigenze della propria applicazione o componente.

Ecco come è possibile creare e utilizzare eventi personalizzati:

1. **Creazione di un evento personalizzato**: Gli eventi personalizzati vengono creati utilizzando il costruttore `CustomEvent()`, che accetta due parametri: il tipo di evento (una stringa che identifica il tipo di evento) e un oggetto opzionale che può contenere dati aggiuntivi associati all'evento.

   ```
   const customEvent = new CustomEvent('customEventName', { d
   etail: { message: 'Custom event data' } });
   ```

2. **Innescare un evento personalizzato**: Una volta creato, è possibile inviare manualmente un evento personalizzato su un oggetto `EventTarget` utilizzando il

metodo `dispatchEvent()`, come mostrato nell'esempio precedente.

3. **Gestione di un evento personalizzato**: Gli eventi personalizzati possono essere gestiti utilizzando `addEventListener()` in modo simile agli eventi standard del DOM. È possibile associare un gestore di eventi personalizzati a qualsiasi oggetto `EventTarget`, come elementi DOM, documenti o oggetti di finestra.

```
myElement.addEventListener('customEventName', function(eve
nt) {
  console.log('Custom event triggered:', event.detail.mess
age);
});
```

4. **Accesso ai dati dell'evento**: I dati associati a un evento personalizzato possono essere accessibili attraverso la proprietà `detail` dell'oggetto evento. Questa proprietà contiene qualsiasi dato aggiuntivo specificato al momento della creazione dell'evento.

Gli eventi personalizzati sono utili quando si desidera creare una comunicazione personalizzata tra diversi componenti o moduli all'interno di un'applicazione. Possono essere utilizzati per notificare cambiamenti di stato, aggiornamenti di dati, azioni utente personalizzate e molto altro ancora. Consentono una maggiore flessibilità e modularità nel design delle applicazioni JavaScript.

Node

L'oggetto `Node` è un'interfaccia che rappresenta un singolo nodo all'interno dell'albero DOM. Questo nodo può essere di diversi tipi, tra cui elementi HTML, testo, commenti e così via. L'interfaccia `Node` fornisce una serie di proprietà e metodi che consentono di manipolare e navigare attraverso la struttura ad albero del DOM.

Ecco alcune caratteristiche e funzionalità dell'oggetto `Node`:

1. **Tipi di nodi**: Un nodo può essere di diversi tipi, inclusi:
   - `ELEMENT_NODE` (1): Rappresenta un elemento HTML.
   - `TEXT_NODE` (3): Rappresenta un nodo di testo all'interno di un elemento.

- `COMMENT_NODE` (8): Rappresenta un commento HTML.
- `DOCUMENT_NODE` (9): Rappresenta il nodo radice del documento.
- E molti altri.

2. **Proprietà comuni**: L'oggetto `Node` fornisce diverse proprietà comuni a tutti i tipi di nodi, tra cui:
   - `nodeType`: Il tipo numerico del nodo.
   - `nodeName`: Il nome del nodo.
   - `nodeValue`: Il valore del nodo (ad esempio, il testo di un nodo di testo).
   - `parentNode`: Il nodo genitore del nodo corrente.
   - `childNodes`: Una lista di nodi figlio del nodo corrente.
   - `firstChild`, `lastChild`: Il primo e l'ultimo nodo figlio del nodo corrente.
   - `nextSibling`, `previousSibling`: Il nodo successivo e precedente al nodo corrente.
   - E molti altri.
3. **Metodi di manipolazione del DOM**: L'oggetto `Node` fornisce anche metodi per manipolare la struttura del DOM, inclusi:
   - `appendChild()`, `removeChild()`: Per aggiungere e rimuovere nodi figlio.
   - `cloneNode()`: Per clonare un nodo.
   - `insertBefore()`, `replaceChild()`: Per inserire e sostituire nodi.
   - E altri metodi.
4. **Metodi di navigazione del DOM**: L'oggetto `Node` offre metodi per navigare attraverso l'albero DOM, tra cui:
   - `parentNode`, `firstChild`, `lastChild`, `nextSibling`, `previousSibling`: Per navigare tra i nodi fratelli e genitori.
   - `childNodes`: Per accedere ai nodi figlio.
   - `contains()`: Per verificare se un nodo è discendente di un altro nodo.
   - E altri metodi.

`parentsNode` *vs* `parentsElement`

Sono due proprietà nel DOM che consentono di accedere al nodo genitore di un dato nodo, ma ci sono alcune differenze importanti tra di loro:

1. `parentNode`:
    - La proprietà `parentNode` restituisce il nodo genitore più vicino di un dato nodo, indipendentemente dal tipo di nodo genitore.
    - Può restituire qualsiasi tipo di nodo genitore, inclusi nodi testo, elementi, commenti, documenti, ecc.
    - Se il nodo non ha un nodo genitore (ad esempio, se è il nodo radice del documento), la proprietà `parentNode` restituirà `null`.
2. `parentElement`:
    - La proprietà `parentElement` restituisce il nodo genitore più vicino di tipo elemento di un dato nodo.
    - Restituirà solo nodi genitori di tipo elemento, ignorando i nodi genitori di altri tipi (come nodi testo o commenti).
    - Se il nodo non ha un nodo genitore di tipo elemento, la proprietà `parentElement` restituirà `null`.

Ecco un esempio che mostra le differenze tra le due proprietà:

```
<div id="parentDiv">
  <p id="childParagraph">Questo è un paragrafo.</p>
</div>

<script>
const childNode = document.getElementById('childParagraph');
console.log(childNode.parentNode); // Restituisce <div id="pa
rentDiv">
console.log(childNode.parentElement); // Restituisce <div id
="parentDiv">
</script>
```

In questo esempio, entrambe le proprietà restituiranno il nodo `<div id="parentDiv">`, poiché è il nodo genitore più vicino dell'elemento `<p>`. Tuttavia, se il nodo genitore fosse stato un nodo di testo o un commento, solo `parentNode` lo avrebbe restituito, mentre `parentElement` avrebbe restituito `null`.

NodeTypes

Oggetto	Tipo	Valore
Element	ELEMENT_NODE	1
Attribute	ATTRIBUTE_NODE	2
Text	TEXT_NODE	3
CDATASection	CDATA_SECTION_NODE	4
ProcessingInstruction (XML)	PROCESSING_INSTRUCTION_NODE	7
Comment	COMMENT_NODE	8
Document	DOCUMENT_NODE	9
DocumentType (<!DOCTYPE html>)	DOCUMENT_TYPE_NODE	10
DocumentFragment	DOCUMENT_FRAGMENT_NODE	11

NodeList

Rappresenta una raccolta di nodi. Questi nodi sono ordinati nello stesso ordine in cui appaiono nel documento HTML. Un `NodeList` è simile a un array in quanto fornisce un'interfaccia simile a un array per accedere ai suoi elementi, ma non è tecnicamente un array. Ecco alcuni punti chiave riguardo ai `NodeList`:

1. **Contenuto**: Un `NodeList` contiene nodi, che possono essere di vari tipi, tra cui elementi, nodi di testo, commenti, ecc. Quando si seleziona un insieme di elementi HTML tramite metodi come `document.querySelectorAll()` o `parentNode.childNodes`, viene restituito un `NodeList` contenente quegli elementi.

2. **Accesso agli elementi**: Come accennato, un `NodeList` fornisce un'interfaccia simile a un array per accedere ai suoi elementi. Ciò significa che è possibile utilizzare la notazione delle parentesi quadre `[ ]` e i metodi come `forEach()`, `map()`, `indexOf()`, ecc. per lavorare con i nodi all'interno del `NodeList`.
3. **Staticità**: Un `NodeList` può essere statico o dinamico. Un `NodeList` statico non si aggiorna automaticamente quando il DOM cambia, mentre un `NodeList` dinamico può essere aggiornato in tempo reale per riflettere le modifiche al DOM.
4. **Iterazione**: È possibile iterare su un `NodeList` utilizzando un ciclo `for`, un ciclo `for...of`, o metodi come `forEach()` o `entries()`.

```
<ul id="myList">
  <li>Elemento 1</li>
  <li>Elemento 2</li>
  <li>Elemento 3</li>
</ul>

<script>
// Seleziona tutti gli elementi <li> all'interno di <ul id="myLi
const listItems = document.querySelectorAll('#myList li');

// Utilizza forEach per iterare su ogni elemento <li> e stampar
listItems.forEach(function(item, index) {
  console.log(`Elemento ${index + 1}: ${item.textContent}`);
});
</script>
```

In questo esempio, `listItems` è un `NodeList` contenente tutti gli elementi `<li>` all'interno dell'elemento `<ul id="myList">`. Usiamo `forEach()` per iterare su ciascun elemento `<li>` e stampiamo il loro testo.

Document

Oggetto fondamentale nel Document Object Model (DOM) che rappresenta l'intero documento HTML. È il punto di ingresso principale per interagire con il contenuto di una pagina web tramite JavaScript. Ecco alcuni punti chiave riguardo all'oggetto `Document`:

1. **Rappresentazione del documento**: Il `Document` rappresenta l'intero documento HTML caricato nel browser. Include tutti gli elementi HTML, i commenti, i nodi di testo e altri elementi presenti nella pagina web.
2. **Punto di accesso**: Il `Document` è il punto di accesso principale per selezionare e manipolare gli elementi della pagina web. Fornisce metodi per accedere agli elementi utilizzando selezioni CSS come `getElementById()`, `getElementsByClassName()`, `getElementsByTagName()`, `querySelector()`, `querySelectorAll()`, ecc.
3. **Creazione e manipolazione degli elementi**: Il `Document` fornisce metodi per creare nuovi elementi HTML dinamicamente e aggiungerli alla pagina web utilizzando `createElement()`, `createTextNode()`, `appendChild()`, `insertBefore()`, ecc.
4. **Metodi per la gestione del documento**: Il `Document` fornisce metodi per gestire aspetti del documento, come la manipolazione del titolo della pagina (`title`), la gestione degli eventi di caricamento della pagina (`DOMContentLoaded`), la gestione degli eventi globali (`addEventListener()`), ecc.
5. **Accesso ad altri oggetti del DOM**: Attraverso il `Document`, è possibile accedere ad altri oggetti del DOM, come l'oggetto `window`, che rappresenta la finestra del browser, e l'oggetto `document.documentElement`, che rappresenta l'elemento radice (`<html>`) del documento.

Ecco un esempio di come utilizzare l'oggetto `Document` per selezionare un elemento e aggiungere del testo ad esso:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>Document Example</title>
```

```
</head>
<body>

<div id="myDiv"></div>

<script>
// Seleziona l'elemento <div> con id "myDiv"
const myDiv = document.getElementById('myDiv');

// Crea un nuovo nodo di testo
const newText = document.createTextNode('Questo è un nuovo te
sto.');

// Aggiungi il nodo di testo come figlio dell'elemento <div>
myDiv.appendChild(newText);
</script>

</body>
</html>
```

In questo esempio, utilizziamo l'oggetto `Document` per selezionare un elemento `<div>` con l'ID "myDiv" utilizzando `getElementById()`. Successivamente, creiamo un nuovo nodo di testo utilizzando `createTextNode()`, e infine aggiungiamo il nodo di testo come figlio dell'elemento `<div>` utilizzando `appendChild()`.

Proprietà:

Queste proprietà consentono di accedere a vari aspetti del documento, come i suoi elementi, le sue informazioni e i suoi metadati. Di seguito sono elencate alcune delle proprietà più comuni dell'oggetto `document`:

1. `document.documentElement`: Restituisce l'elemento radice (`<html>`) del documento.
2. `document.body`: Restituisce l'elemento `<body>` del documento, che contiene il contenuto principale della pagina.

3. `document.head`: Restituisce l'elemento `<head>` del documento, che contiene metadati, script e altri elementi di intestazione.
4. `document.title`: Restituisce o imposta il titolo del documento, che appare nella barra del titolo del browser.
5. `document.URL`: Restituisce l'URL del documento corrente.
6. `document.domain`: Restituisce il dominio del documento corrente.
7. `document.referrer`: Restituisce l'URL del documento che ha caricato la pagina corrente (la pagina di provenienza).
8. `document.location`: Restituisce un oggetto `Location` che rappresenta l'URL del documento corrente.
9. `document.forms`: Restituisce una collezione di tutti gli elementi `<form>` presenti nel documento.
10. `document.images`: Restituisce una collezione di tutti gli elementi `<img>` presenti nel documento.
11. `document.links`: Restituisce una collezione di tutti gli elementi `<a>` presenti nel documento.
12. `document.scripts`: Restituisce una collezione di tutti gli elementi `<script>` presenti nel documento.
13. `document.styleSheets`: Restituisce una collezione di tutti i fogli di stile (`<link>` e `<style>`) presenti nel documento.
14. `document.all`: Restituisce una collezione di tutti gli elementi HTML presenti nel documento.
15. `document.readyState`: Restituisce lo stato di pronto del documento (come "loading", "interactive" o "complete").

Come selezionare i nodi del documento HTML

1. **getElementById()**: Seleziona un elemento tramite il suo ID univoco.

```
const element = document.getElementById('idDelElemento');
```

2. **getElementsByClassName()**: Seleziona una collezione di elementi che hanno una specifica classe.

```
const elements = document.getElementsByClassName('nomeClasse');
```

3. **getElementsByTagName()**: Seleziona una collezione di elementi basata sul loro nome di tag.

```
const elements = document.getElementsByTagName('tagname');
```

4. **querySelector()**: Seleziona il primo elemento che corrisponde a un selettore CSS specificato.

```
const element = document.querySelector('selettoreCSS');
```

5. **querySelectorAll()**: Seleziona tutti gli elementi che corrispondono a un selettore CSS specificato e restituisce una NodeList.

```
const elements = document.querySelectorAll('selettoreCSS');
```

Ecco alcuni esempi di utilizzo di questi metodi:

```
// Seleziona un elemento tramite ID
const elementById = document.getElementById('idDelElemento');

// Seleziona una collezione di elementi per classe
const elementsByClass = document.getElementsByClassName('nomeClasse');
```

```
// Seleziona una collezione di elementi per tag
const elementsByTag = document.getElementsByTagName('tagnam
e');

// Seleziona il primo elemento che corrisponde al selettore C
SS
const elementByQuery = document.querySelector('selettoreCS
S');

// Seleziona tutti gli elementi che corrispondono al selettor
e CSS
const elementsByQueryAll = document.querySelectorAll('seletto
reCSS');
```

Possono anche esser combinati diversi metodi:

```
// Seleziona tutti gli elementi con classe "example" all'interno
const elementsInContainer = document.querySelectorAll('#containe
```

Creare i nodi

Per creare nuovi nodi nel Document Object Model (DOM), puoi utilizzare vari metodi offerti dall'oggetto `document`. Di seguito sono mostrati alcuni dei metodi più comuni per creare nodi:

1. **createElement()**: Crea un nuovo elemento con il nome di tag specificato.

   ```
   const nuovoElemento = document.createElement('tagname');
   ```

2. **createTextNode()**: Crea un nuovo nodo di testo con il testo specificato.

   ```
   const nuovoTesto = document.createTextNode('testo del nod
   o');
   ```

3. **createComment()**: Crea un nuovo nodo commento con il testo specificato.

```
const nuovoCommento = document.createComment('testo del commento');
```

Dopo aver creato i nodi, è possibile manipolarli e aggiungerli al documento utilizzando vari metodi di manipolazione del DOM, come `appendChild()`, `insertBefore()`, `replaceChild()`, ecc.

Ecco alcuni esempi di come creare e aggiungere nodi al documento:

```
// Creare un nuovo elemento <div>
const nuovoDiv = document.createElement('div');

// Aggiungere una classe all'elemento <div>
nuovoDiv.classList.add('nuovaClasse');

// Creare un nuovo nodo di testo
const nuovoTesto = document.createTextNode('Questo è un nuovo testo.');

// Creare un nuovo nodo commento
const nuovoCommento = document.createComment('Questo è un nuovo commento.');

// Aggiungere il nodo di testo come figlio dell'elemento <div>
nuovoDiv.appendChild(nuovoTesto);

// Aggiungere il nodo commento come figlio dell'elemento <body> prima dell'elemento con ID "footer"
const body = document.body;
body.insertBefore(nuovoCommento, body.getElementById('footer'));
```

In questo esempio, abbiamo creato un nuovo elemento `<div>` utilizzando `createElement()` e aggiunto una classe ad esso. Successivamente, abbiamo creato un nuovo nodo di testo e un nuovo nodo commento utilizzando rispettivamente `createTextNode()` e `createComment()`. Infine, abbiamo aggiunto il nodo di testo come figlio dell'elemento `<div>` utilizzando `appendChild()` e il nodo commento come figlio dell'elemento `<body>` prima di un altro elemento specifico utilizzando `insertBefore()`.

Element

L'oggetto `Element` rappresenta un elemento HTML all'interno del documento. Gli elementi HTML sono i componenti fondamentali di una pagina web e corrispondono agli elementi definiti nel markup HTML, come `<div>`, `<p>`, `<span>`, `<a>`, ecc.

Ecco alcune caratteristiche e funzionalità dell'oggetto `Element`:

1. **Accesso agli attributi**: L'oggetto `Element` fornisce metodi e proprietà per accedere e manipolare gli attributi degli elementi HTML, come `getAttribute()`, `setAttribute()`, `removeAttribute()`, ecc.
2. **Accesso ai contenuti**: È possibile accedere e manipolare i contenuti degli elementi HTML utilizzando proprietà come `innerHTML`, `textContent`, `innerText`, `outerHTML`, ecc.
3. **Accesso agli stili**: L'oggetto `Element` fornisce metodi e proprietà per accedere e manipolare gli stili CSS degli elementi HTML, come `style`, `getComputedStyle()`, ecc.
4. **Manipolazione della struttura**: È possibile manipolare la struttura degli elementi HTML aggiungendo, rimuovendo o modificando gli elementi figlio utilizzando metodi come `appendChild()`, `removeChild()`, `insertBefore()`, `replaceChild()`, ecc.
5. **Accesso ai genitori e ai fratelli**: È possibile accedere ai genitori e ai fratelli degli elementi HTML utilizzando proprietà come `parentNode`, `parentElement`, `nextSibling`, `previousSibling`, ecc.

6. **Gestione degli eventi**: Gli elementi HTML possono essere resi interattivi associando loro gestori di eventi tramite metodi come `addEventListener()`, `removeEventListener()`, ecc.
7. **Manipolazione delle classi**: È possibile aggiungere, rimuovere o verificare classi CSS sugli elementi HTML utilizzando metodi come `classList.add()`, `classList.remove()`, `classList.contains()`, ecc.

Ecco un esempio di come utilizzare l'oggetto `Element` per manipolare un elemento HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>Element Example</title>
<style>
  .highlight {
    background-color: yellow;
  }
</style>
</head>
<body>

<div id="myDiv">Questo è un div.</div>

<script>
// Otteniamo il riferimento all'elemento <div>
const myDiv = document.getElementById('myDiv');

// Aggiungiamo una classe all'elemento
myDiv.classList.add('highlight');

// Modifichiamo il testo dell'elemento
myDiv.textContent = 'Questo è un div modificato.';
```

```
// Aggiungiamo un gestore di eventi all'elemento
myDiv.addEventListener('click', function() {
  alert('Hai cliccato sul div!');
});
</script>

</body>
</html>
```

In questo esempio, otteniamo il riferimento all'elemento `<div>` utilizzando `getElementById()` e poi manipoliamo l'elemento aggiungendo una classe, modificando il testo e aggiungendo un gestore di eventi.

Alcune proprietà:

1. `classList`: Restituisce una lista di classi dell'elemento e fornisce metodi per manipolarle (`add`, `remove`, `toggle`, `contains`).
2. `className`: Restituisce o imposta l'attributo `class` dell'elemento come una stringa.
3. `id`: Restituisce o imposta l'attributo `id` dell'elemento come una stringa.
4. `tagName`: Restituisce il nome del tag dell'elemento come una stringa.
5. `textContent`: Restituisce o imposta il testo contenuto nell'elemento, includendo i testi dei suoi nodi figlio.
6. `innerHTML`: Restituisce o imposta il markup HTML contenuto all'interno dell'elemento.
7. `style`: Restituisce un oggetto che rappresenta gli stili CSS dell'elemento, consentendo di accedere e modificare gli stili inline.
8. `attributes`: Restituisce una raccolta di tutti gli attributi dell'elemento.
9. `parentNode`: Restituisce il nodo genitore dell'elemento.
10. `children`: Restituisce una collezione di tutti i nodi figlio dell'elemento che sono elementi HTML.

11. `firstChild`, `lastChild`: Restituiscono il primo e l'ultimo nodo figlio dell'elemento, inclusi nodi di testo e commenti.
12. `previousSibling`, `nextSibling`: Restituiscono il nodo fratello precedente e successivo dell'elemento, inclusi nodi di testo e commenti.
13. `offsetWidth`, `offsetHeight`: Restituiscono le dimensioni dell'elemento, compresa la dimensione dei bordi e dello spazio per il riempimento.
14. `offsetTop`, `offsetLeft`: Restituiscono la posizione assoluta dell'angolo superiore sinistro dell'elemento rispetto al suo genitore offset.
15. `clientWidth`, `clientHeight`: Restituiscono le dimensioni dell'elemento, escludendo i bordi e la barra di scorrimento, ma includendo lo spazio per il riempimento.

Gestire gli attributi di un elemento HTML in Javascript

Per gestire gli attributi di un elemento HTML in JavaScript, puoi utilizzare una serie di metodi e proprietà offerti dall'oggetto `Element`. Ecco alcuni dei principali metodi e proprietà per lavorare con gli attributi degli elementi:

1. **getAttribute()**: Restituisce il valore di un attributo specificato dell'elemento.

```
const valore = elemento.getAttribute('attributo');
```

2. **setAttribute()**: Imposta il valore di un attributo specificato dell'elemento.

```
elemento.setAttribute('attributo', 'valore');
```

3. **hasAttribute()**: Verifica se l'elemento ha un determinato attributo.

```
const esiste = elemento.hasAttribute('attributo');
```

4. **removeAttribute()**: Rimuove un attributo specificato dall'elemento.

```
elemento.removeAttribute('attributo');
```

5. **attributes**: Restituisce una collezione degli attributi dell'elemento come oggetti `Attr`.

```
const attributi = elemento.attributes;
```

Ecco un esempio che mostra come utilizzare questi metodi per gestire gli attributi di un elemento:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>Gestione attributi</title>
</head>
<body>

<button id="myButton" tipo="pulsante">Clicca qui</button>

<script>
// Otteniamo il riferimento all'elemento <button>
const myButton = document.getElementById('myButton');

// Otteniamo il valore dell'attributo "tipo"
const tipo = myButton.getAttribute('tipo');
console.log('Tipo:', tipo);

// Impostiamo un nuovo valore per l'attributo "tipo"
myButton.setAttribute('tipo', 'reset');

// Verifichiamo se l'elemento ha l'attributo "tipo"
const haTipo = myButton.hasAttribute('tipo');
console.log('Ha attributo "tipo":', haTipo);
```

```
// Rimuoviamo l'attributo "tipo"
myButton.removeAttribute('tipo');

// Verifichiamo nuovamente se l'elemento ha l'attributo "tip
o"
const haAncoraTipo = myButton.hasAttribute('tipo');
console.log('Ha ancora attributo "tipo":', haAncoraTipo);
</script>

</body>
</html>
```

In questo esempio, selezioniamo un pulsante con un attributo `tipo` utilizzando `getElementById()`. Successivamente, utilizziamo `getAttribute()` per ottenere il valore dell'attributo `tipo`, `setAttribute()` per cambiarlo, `hasAttribute()` per verificare se l'elemento ha un determinato attributo, e infine `removeAttribute()` per rimuoverlo.

DOM e CSS

Ecco come interagiscono tra di loro:

1. **Selezione degli elementi**: Utilizzando il DOM, è possibile selezionare gli elementi HTML all'interno del documento utilizzando JavaScript. Questi elementi possono poi essere manipolati e modificati dinamicamente. I selettori CSS vengono utilizzati per selezionare gli elementi in base al loro tipo, classe, ID o altri attributi. Ad esempio, `document.getElementById('myElement')` utilizza il DOM per selezionare un elemento per ID, mentre `.myClass` in CSS seleziona elementi con una classe specifica.
2. **Modifica dello stile**: Una volta selezionati gli elementi utilizzando il DOM, è possibile modificare i loro stili utilizzando JavaScript. Ciò può essere fatto manipolando le proprietà `style` dell'oggetto Element. Tuttavia, è generalmente considerato una pratica migliore utilizzare i fogli di stile CSS per definire gli stili e cambiare dinamicamente le classi degli elementi HTML in modo da applicare gli stili desiderati. Ad esempio, invece di modificare direttamente

`element.style.color = 'red'` in JavaScript, è preferibile aggiungere o rimuovere una classe CSS che definisce il colore desiderato.

3. **Eventi CSS**: CSS offre la possibilità di definire transizioni, animazioni e pseudoclassi che possono essere attivate in risposta a determinati eventi, come il passaggio del mouse sopra un elemento (`:hover`). Questi eventi sono indipendenti dal DOM e possono essere applicati a qualsiasi elemento che corrisponde al selettore CSS specificato.
4. **Visualizzazione dinamica**: Il DOM e i fogli di stile CSS lavorano insieme per creare una visualizzazione dinamica della pagina web. Quando gli elementi vengono modificati o aggiornati nel DOM, il browser ri-renderizza la pagina applicando i fogli di stile CSS appropriati. Questo processo consente di creare esperienze utente interattive e responsive.

kebab-case e lowerCamelCase

Sono due convenzioni di denominazione utilizzate nella programmazione per definire il nome delle variabili, delle funzioni e degli identificatori in generale. Ecco una spiegazione di entrambe le convenzioni:

1. **kebab-case**:
   - Nomi in kebab-case sono composti da parole in minuscolo separate da trattini (o "kebab"), ad esempio: `my-variable`, `some-example-name`.
   - È comunemente utilizzato in CSS per i nomi delle proprietà, come `font-size`, `background-color`.
   - È anche utilizzato negli URL di alcuni framework o librerie JavaScript, ad esempio Angular, Vue.js, per definire nomi di componenti o percorsi delle rotte.
2. **lowerCamelCase**:
   - Nomi in lowerCamelCase iniziano con una lettera minuscola e le parole successive sono iniziate con una lettera maiuscola, senza spazi o altri separatori, ad esempio: `myVariable`, `someExampleName`.

- È comunemente utilizzato in JavaScript per i nomi delle variabili e delle funzioni, ad esempio: `firstName`, `calculateArea`.

Ognuna di queste convenzioni ha i suoi vantaggi e le sue situazioni in cui è più appropriata:

- **kebab-case** è utile quando si lavora con lingue che non supportano la distinzione tra maiuscole e minuscole nei nomi degli identificatori, come HTML, CSS, URL. È inoltre più leggibile per i file di configurazione o le definizioni di stile.
- **lowerCamelCase** è più comune in JavaScript e altre lingue di programmazione che seguono la convenzione camelCase. È più leggibile e conveniente quando si lavora con nomi di variabili, funzioni e identificatori all'interno del codice sorgente.

Attr

Rappresentano gli attributi degli elementi HTML all'interno del documento. Ogni elemento può avere zero o più attributi, ognuno dei quali è rappresentato da un oggetto `Attr`. Ecco alcune informazioni importanti sugli oggetti `Attr`:

1. **Nome e valore dell'attributo**: Un oggetto `Attr` contiene il nome e il valore di un attributo di un elemento HTML.
2. **Accesso tramite elementi**: Gli oggetti `Attr` non possono essere creati direttamente, ma possono essere ottenuti attraverso gli elementi HTML utilizzando i metodi come `getAttributeNode()`, `getAttribute()` o attraverso la proprietà `attributes` dell'elemento.
3. **Manipolazione degli attributi**: Gli oggetti `Attr` forniscono metodi e proprietà per accedere e manipolare gli attributi degli elementi HTML. Ad esempio, è possibile ottenere il nome e il valore dell'attributo, modificarlo o rimuoverlo.
4. **Iterazione sugli attributi**: Utilizzando la proprietà `attributes` dell'elemento, è possibile ottenere una raccolta di tutti gli oggetti `Attr` corrispondenti agli attributi dell'elemento. È quindi possibile iterare su questa collezione per esaminare e manipolare gli attributi.

Ecco un esempio di come ottenere e manipolare gli oggetti `Attr` attraverso un elemento HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>Attr Example</title>
</head>
<body>

<div id="myDiv" data-custom="value">Contenuto del div</div>

<script>
// Otteniamo il riferimento all'elemento <div>
const myDiv = document.getElementById('myDiv');

// Otteniamo un oggetto Attr per l'attributo 'data-custom' de
ll'elemento <div>
const customAttr = myDiv.getAttributeNode('data-custom');

// Otteniamo il nome e il valore dell'attributo
const nomeAttributo = customAttr.name;
const valoreAttributo = customAttr.value;

console.log('Nome attributo:', nomeAttributo); // 'data-custo
m'
console.log('Valore attributo:', valoreAttributo); // 'value'

// Modifichiamo il valore dell'attributo
customAttr.value = 'nuovoValore';

// Otteniamo nuovamente il valore dell'attributo dopo la modi
fica
```

```
const nuovoValoreAttributo = myDiv.getAttribute('data-custo
m');
console.log('Nuovo valore attributo:', nuovoValoreAttributo);
// 'nuovoValore'
</script>

</body>
</html>
```

In questo esempio, otteniamo un oggetto `Attr` per l'attributo `data-custom` dell'elemento `<div>` utilizzando `getAttributeNode()`. Successivamente, otteniamo il nome e il valore dell'attributo utilizzando le proprietà `name` e `value` dell'oggetto `Attr`. Infine, modificiamo il valore dell'attributo e otteniamo nuovamente il suo valore per confermare la modifica.

CharacterData

Rappresentano i nodi di testo e i nodi commento all'interno di un documento HTML. Questi oggetti rappresentano i dati di testo effettivi o il contenuto di un commento all'interno della struttura del documento. L'interfaccia `CharacterData` fornisce metodi e proprietà per accedere e manipolare questi dati di testo o contenuti dei commenti.

Ecco alcuni punti importanti relativi agli oggetti `CharacterData`:

1. **Nodi di testo e nodi commento**: Gli oggetti `CharacterData` possono rappresentare sia nodi di testo che nodi commento nel DOM. I nodi di testo contengono testo effettivo all'interno di un elemento HTML, mentre i nodi commento contengono commenti.
2. **Accesso ai dati**: Gli oggetti `CharacterData` forniscono la proprietà `data` che restituisce il contenuto del nodo di testo o del nodo commento come una stringa.
3. **Manipolazione dei dati**: È possibile modificare il contenuto del nodo di testo o del nodo commento assegnando una nuova stringa alla proprietà `data`.

4. **Lunghezza dei dati**: La proprietà `length` restituisce la lunghezza del contenuto del nodo di testo o del nodo commento.
5. **Eventi**: Gli oggetti `CharacterData` possono generare eventi quando il loro contenuto viene modificato. Gli eventi come `DOMNodeInserted`, `DOMNodeRemoved`, `DOMCharacterDataModified`, ecc., vengono generati quando il contenuto del nodo viene modificato.

Ecco un esempio di come utilizzare un oggetto `CharacterData` per manipolare il contenuto di un nodo di testo:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
<title>CharacterData Example</title>
</head>
<body>

<div id="myDiv">Testo originale</div>

<script>
// Otteniamo il riferimento al nodo di testo all'interno del
div
const myDiv = document.getElementById('myDiv');
const nodoTesto = myDiv.firstChild;

// Otteniamo il contenuto del nodo di testo
console.log('Contenuto originale:', nodoTesto.data);

// Modifichiamo il contenuto del nodo di testo
nodoTesto.data = 'Nuovo testo modificato';

// Otteniamo la nuova lunghezza del contenuto
console.log('Nuova lunghezza:', nodoTesto.length);
```

```
</script>

</body>
</html>
```