Browser e sue componenti

Web Browser

- Ad oggi i browser possono essere considerati delle suite di software che offrono più funzionalità
- Sono in grado di interpretare e visualizzare:
  - Pagine Web HTML
  - Script Javascript
  - AJAX
  - Applicazioni
  - Altri contenuti che possono essere ospitati sui server Web
- Molti browser offrono plugin che estendono le loro capacità e caratteristiche, consentendo di navigare ed interagire con contenuti multimediali ed eseguire task specifici (es: video conferenze, aggiungere filtri anti-phishing, progettare pagine Web, ecc.)

Web Browser

- Il modo in cui il browser interpreta e visualizza i file HTML è specificato negli standard HTML e CSS
- Questi standard sono rilasciati e mantenuti dal W3C
- Per molti anni i browser sono stati conformi a solo una parte delle specifiche ed hanno sviluppato le loro proprie estensioni: questo ha scatenato le guerre dei browser, portando a incompatibilità significative, con una conseguente difficoltà da parte degli sviluppatori Web
- Ad oggi, la maggior parte dei browser ha incrementato il livello di conformità rispetto agli standard del W3C

Web Browser

- Come dicevamo nel corso della scorsa lezione, la situazione attuale vede il predominio incontrastato di Chrome e la dismissione annunciata di Internet Explorer, che fu il vincitore delle prima guerra dei browser
- Microsoft ha annunciato che a partire dal 17 agosto 2021 le app della suite Microsoft 365 non supporteranno più IE

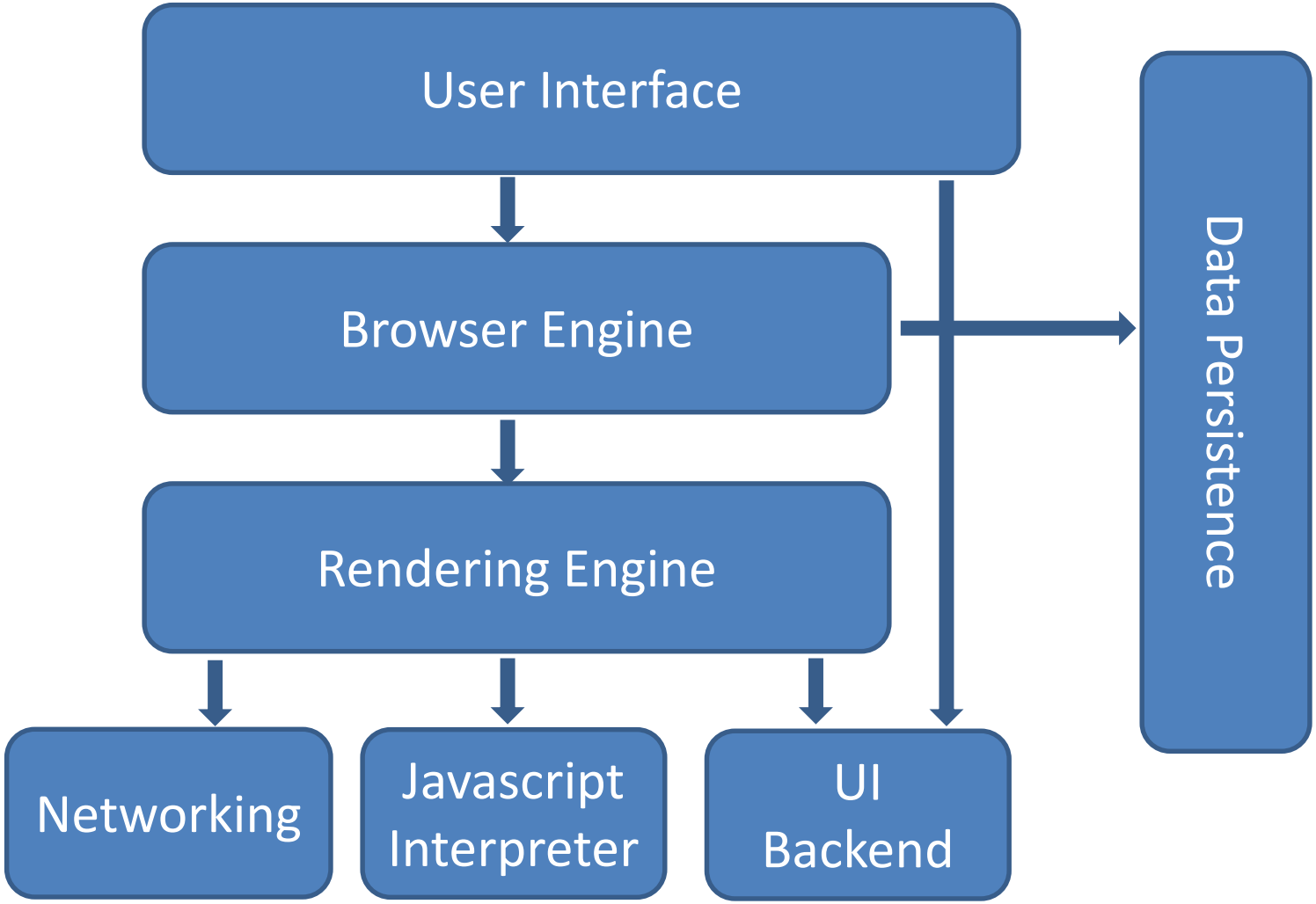
Browser: componenti principali

- Un browser può essere visto come un gruppo di software con codice strutturato che insieme eseguono diversi task per visualizzare una pagina web nella sua finestra
- Questi diversi software sono considerati le componenti del browser:
  1. **Interfaccia utente (UI)**: include tutti gli elementi e le funzionalità a corredo del browser (barra degli indirizzi, bottoni Next e Back, menu per il bookmark, ecc). In pratica, è costituita da tutte le parti della finestra del browser con cui possiamo interagire, tranne la finestra di visualizzazione, in cui viene renderizzata la pagina HTML
  2. **Browser engine**: è la parte del browser che fa da "ponte" tra l'interfaccia utente e il motore di rendering delle pagine HTML

Browser: componenti principali

3. **Rendering engine (o motore di rendering)**: si occupa della visualizzazione della risorsa richiesta (parsing e rendering di HTML, CSS)
4. **Networking (o client HTTP)**: si occupa dell'invio e della ricezione delle richieste HTTP
5. **Interprete (o motore) JavaScript**: si occupa del parsing e dell'interpretazione del codice JavaScript
6. **UI backend**: si occupa del rendering e del disegno di elementi grafici di base (come finestre, bottoni, combo box, ecc); espone elementi di una generica interfaccia, non dipendente da una specifica piattaforma, basandosi sull'interfaccia del sistema operativo
7. **Data Persistence**: si occupa di ospitare e gestire i dati e le informazioni che il browser deve salvare localmente, come ad esempio i cookie (altri meccanismi di storage supportati dal browser: localStorage, WebSQL, IndexedDB, FileSystem, ecc)

Browser: overview dell'architettura



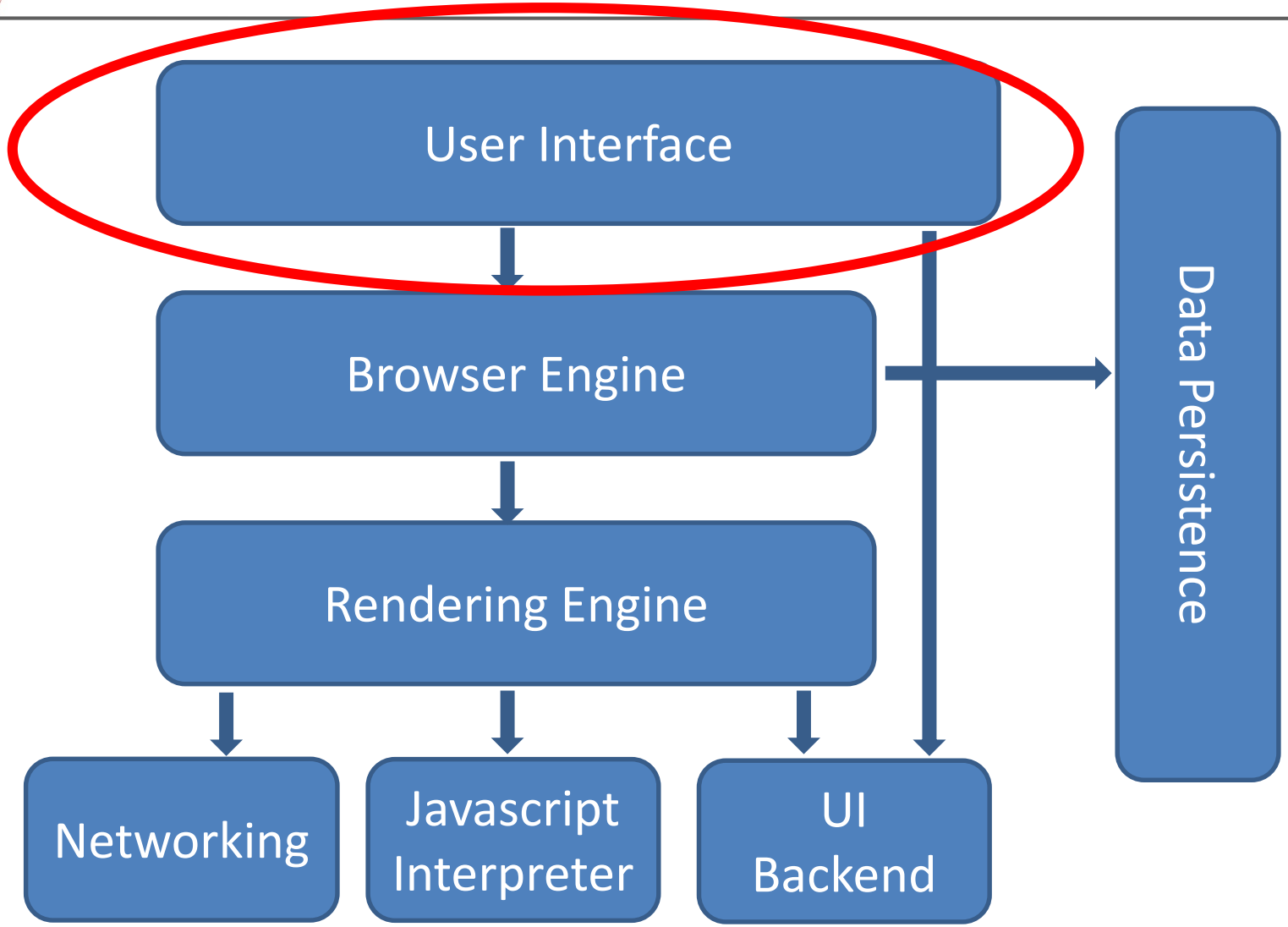
Browser: componenti principali

- Vediamo ora alcuni dettagli per ogni componente
- Approfondiremo maggiormente 2 componenti:
  - Rendering Engine: per alcuni dettagli confronteremo il funzionamento, le caratteristiche e le differenze tra 2 motori di rendering: Gecko (Firefox) e WebKit (Safari)
  - Javascript Interpreter

Q&A

- Ci sono domande?

Browser: overview dell'architettura

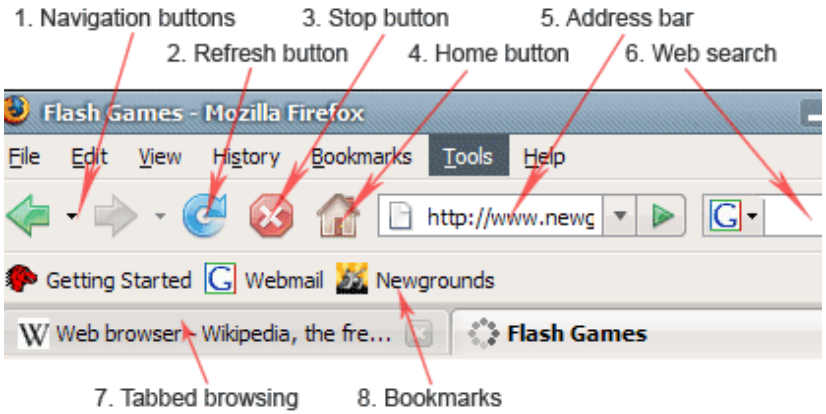


Interfaccia utente

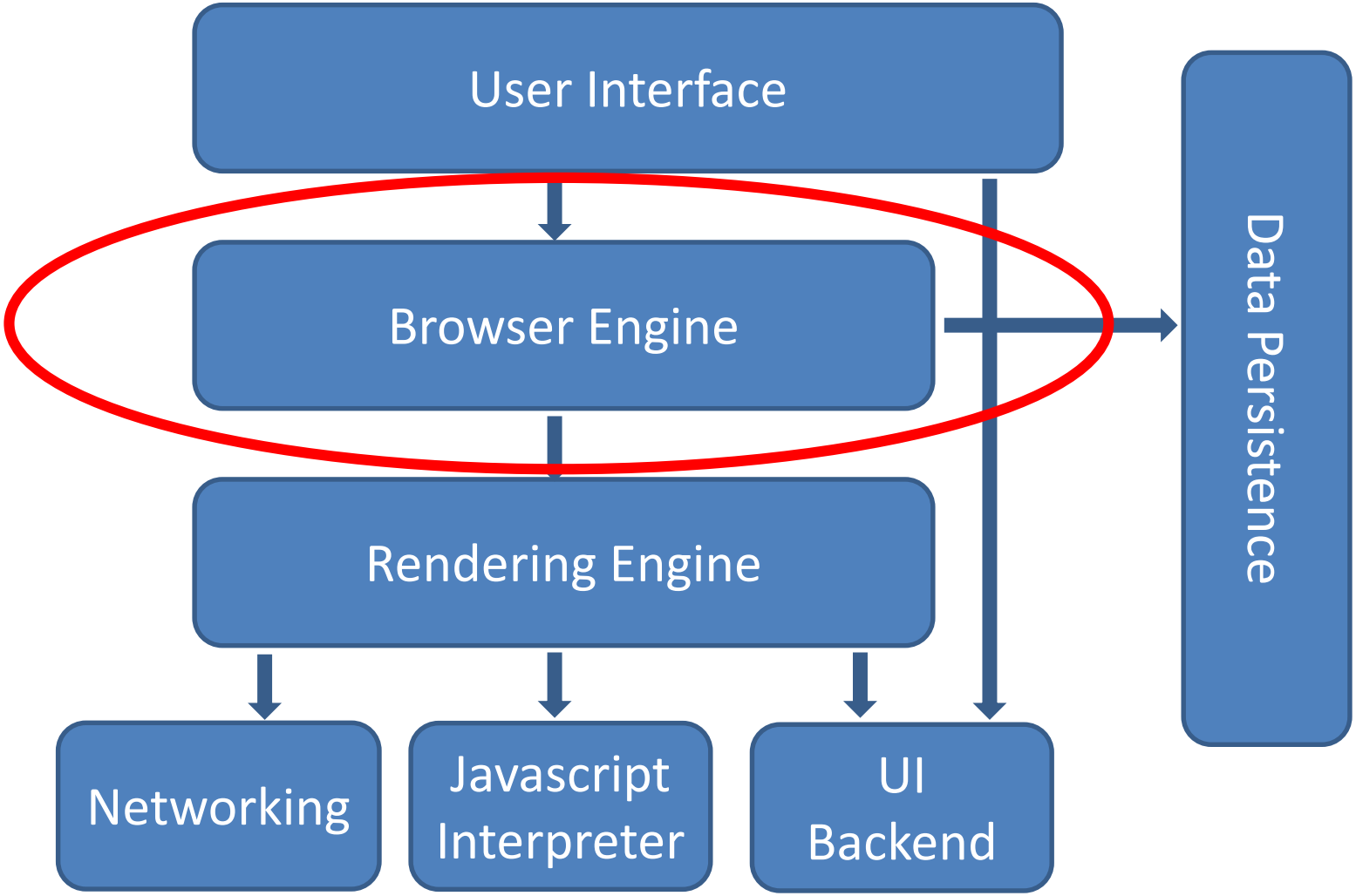
- L'interfaccia utente del browser è composta dagli elementi attraverso i quali l'utente interagisce con il browser
- L'interfaccia utente comunica con gli altri componenti del browser per mostrare il contenuto delle risorse Web e interagisce anche con il sistema operativo
- Non ci sono standard o specifiche che definiscano in modo formale il look&feel e l'organizzazione dell'interfaccia utente del browser
- Deriva da best practice che si sono raccolte nel corso degli anni e che i browser hanno imitato l'uno dall'altro
- Riporta elementi simili (per funzionamento e per aspetto) a quelli delle interfacce di altre applicazioni

Interfaccia utente

- Lo standard HTML5 non definisce gli elementi GUI che i browser devono necessariamente avere, ma elenca alcuni elementi comuni (https://www.w3.org/TR/2014/REC-html5-20141028/browsers.html)
- Tra gli altri, include i seguenti elementi:
  - Barra degli indirizzi (URI)
  - Bottoni di Back e Next
  - Bottone di Home page
  - Bottone di Refresh e Stop
  - Elementi per la gestione del bookmark
  - Tab
  - Ogni altro elemento al di fuori della finestra di rendering (***viewport***)



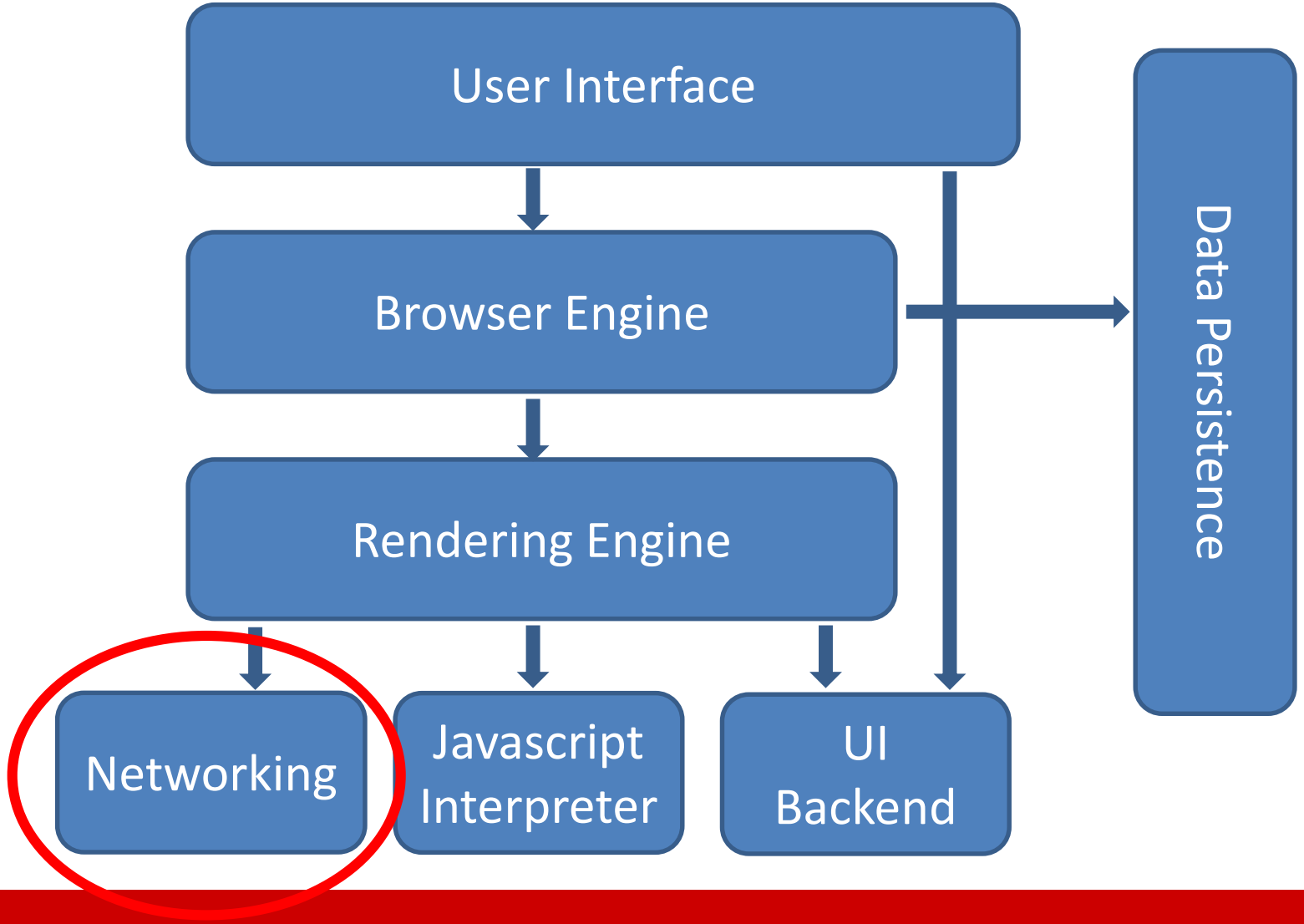
Browser: overview dell'architettura



Browser engine

- Il motore del browser funziona come una sorta di “ponte” tra l’interfaccia utente del browser e il motore di rendering
- Sulla base degli input ricevuti attraverso l’interfaccia utente, si occupa di effettuare le richieste al motore di rendering
- Si occupa del caricamento dell’URL e supporta le primitive relative alle principali azioni di browsing, come ad esempio Back, Next, Reload
- Si occupa di vari aspetti della sessione di navigazione, come ad esempio la visualizzare dei progressi nel caricamento della pagina corrente e gli alert relativi all’esecuzione di script Javascript

Browser: overview dell'architettura



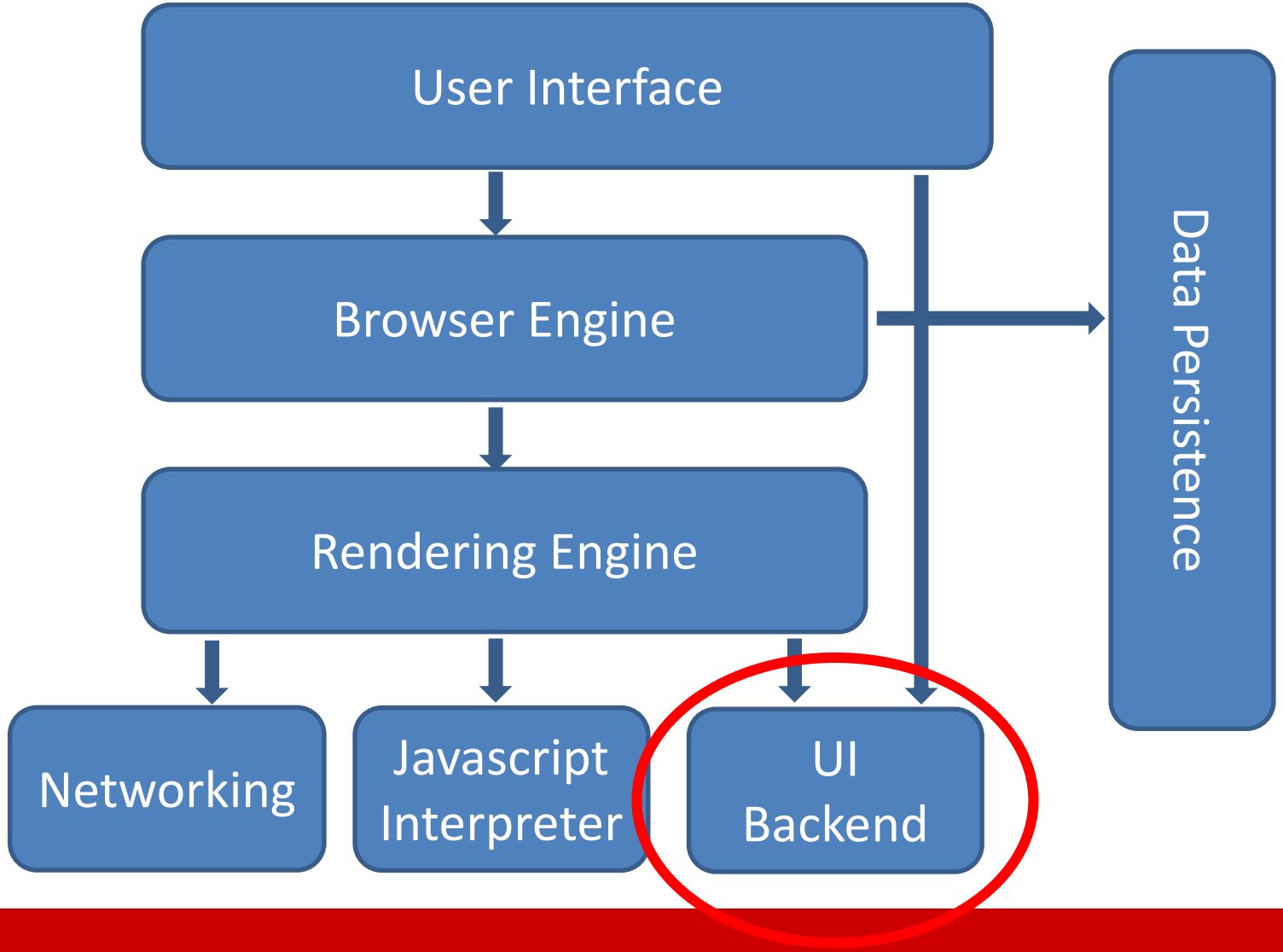
Networking – Client HTTP

- Si occupa di recuperare gli URL richiesti dall'utente (attraverso l'interfaccia) utilizzando i principali protocolli di comunicazione (HTTP, ma anche FTP), in particolare è in grado di:
  - Inviare una Richiesta HTTP di una risorsa (sulla base del suo URL) ad un server Web (richiesta ricevuta attraverso l'interfaccia utente o attraverso la risorsa attualmente renderizzata nel viewport)
  - Ricevere dal parte del server una risorsa richiesta (che permette di alimentare il motore di rendering)
- Si occupa di gestire tutti gli aspetti di comunicazione attraverso la rete Internet (inclusi quelli di Sicurezza)
- Potrebbe rappresentare un bottleneck delle performance del browser durante la navigazione, dato che deve necessariamente restare in attesa delle risorse che arrivano dai server per comporre la pagina Web

Networking – Client HTTP

- Può implementare una cache dei documenti recuperati, per ridurre il traffico sulla rete
- Se il sito web richiesto implementa una cache, allora viene fatta una copia dei dati in AppCache o Service Workers, per l'accesso successivo (risposte rapide, risparmio di tempo per accessi regolari e ripetuti)
- Il browser inizialmente controlla se sono presenti cache di dati e risorse nella memoria locale per ogni URL richiesto, se non trova nulla allora viene creata una richiesta HTTP con un domain name per richiedere la risorsa attraverso la rete Internet

Browser: overview dell'architettura



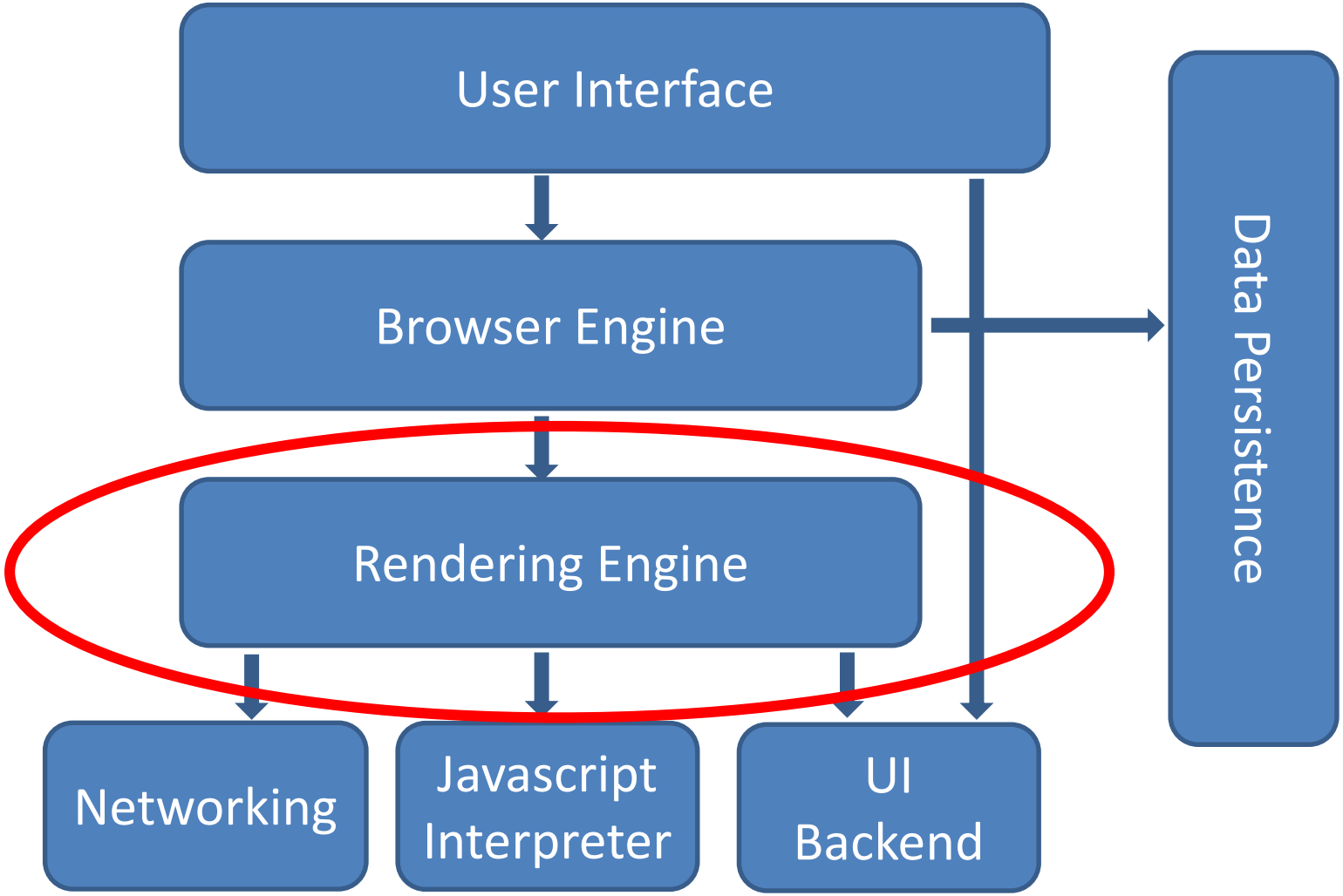
User Interface Backend

- Si occupa di “disegnare”, di effettuare il rendering degli elementi grafici di base, come ad esempio i widget, i bottoni, le combo box, le finestre
- Espone elementi di una interfaccia generica, basandosi sull’interfaccia del sistema operativo e suoi widget grafici

Q&A

- Ci sono domande?

Browser: overview dell'architettura



Rendering Engine

- Si occupa del rendering della pagina web richiesta
- Se si tratta di una pagina HTML, il motore di rendering si occupa di:
  - effettuare il parsing del codice HTML
  - identificare e richiedere le risorse associate (CSS, immagini, video, ecc)
  - “disegnare” una rappresentazione grafica della pagina all’interno della finestra del contenuto (sulla base delle coordinate ricevute)
- E’ in grado di effettuare il parsing e la visualizzazione di documenti XML e di immagini di diversi formati (es: GIF, JPEG, PNG)
- Si occupa della gestione dei link, dei form e dei dettagli ad essi correlati
- Ogni tab del browser lancia una propria istanza del motore di rendering in un processo separato

Principali Motori di Rendering

- Trident (Internet Explorer)
- EdgeHTML (Microsoft Edge)
- Gecko (Firefox, SeaMonkey, Netscape, …)
- KHTML (Konqueror)
- WebKit (iOS, Safari, Google Chrome fino alla versione 27, …)
- Blink (Google Chrome, Opera, …)

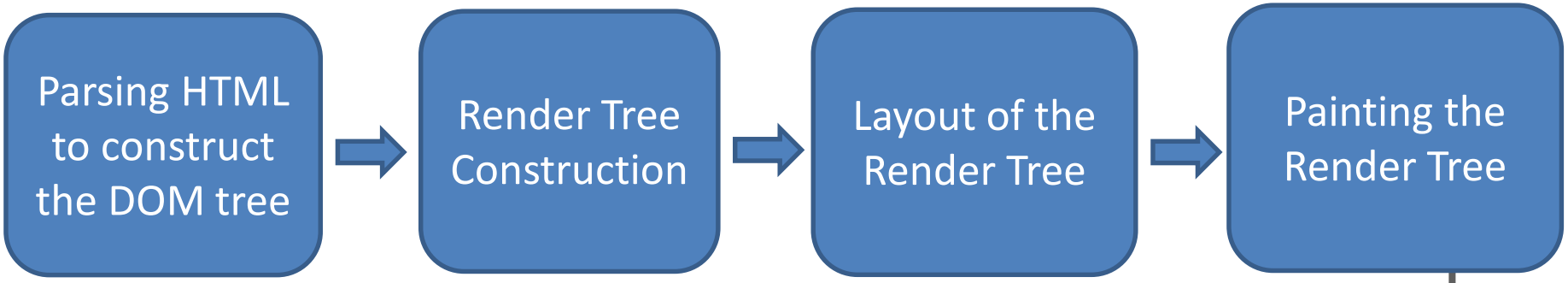
Motore di rendering

Il motore di rendering opera in diverse fasi:

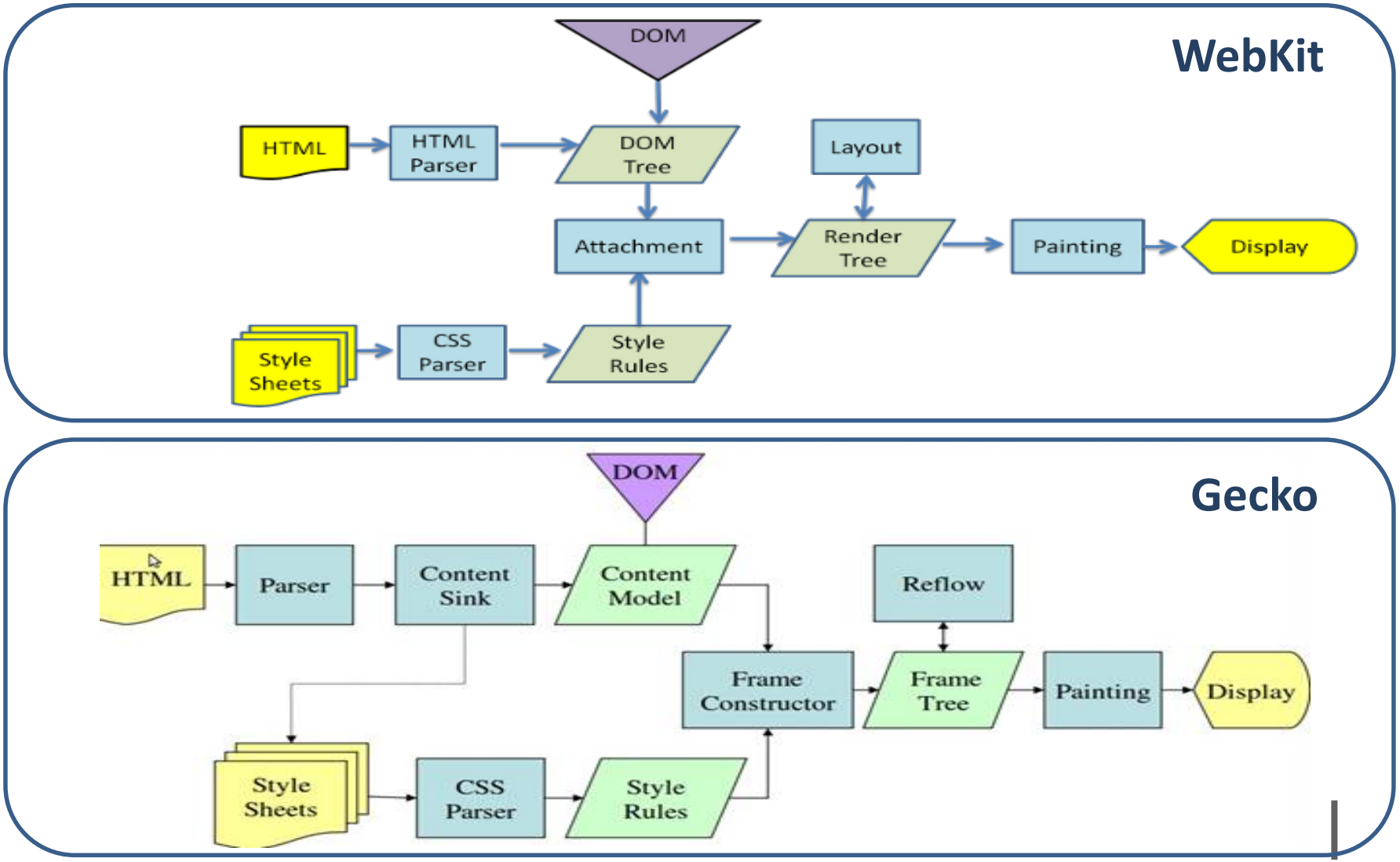
- Ottiene il contenuto dal componente “Networking”, ovvero dal Client HTTP
- Effettua il parsing del documento HTML e converte gli elementi in nodi DOM in un albero denominato ***content tree***. Effettua il parsing delle informazioni relative allo stile, considerando CSS esterni, interni e inline
- Queste informazioni, insieme alle indicazioni di layout e di resa grafica presenti nel codice HTML, servono a creare un ulteriore albero, il ***render tree***, che contiene i rettangoli (o scatole) con le informazioni sulla resa grafica (colori e dimensioni). I rettangoli sono nell’ordine in cui saranno mostrati nella finestra del browser
- Viene poi effettuato il processing del ***layout***: ad ogni nodo del render tree vengono assegnate le coordinate in cui dovrà apparire sullo screen
- Viene eseguito il ***painting***: ogni nodo del render tree viene disegnato sulla base del componente UI backend corrispondente

Main Flow

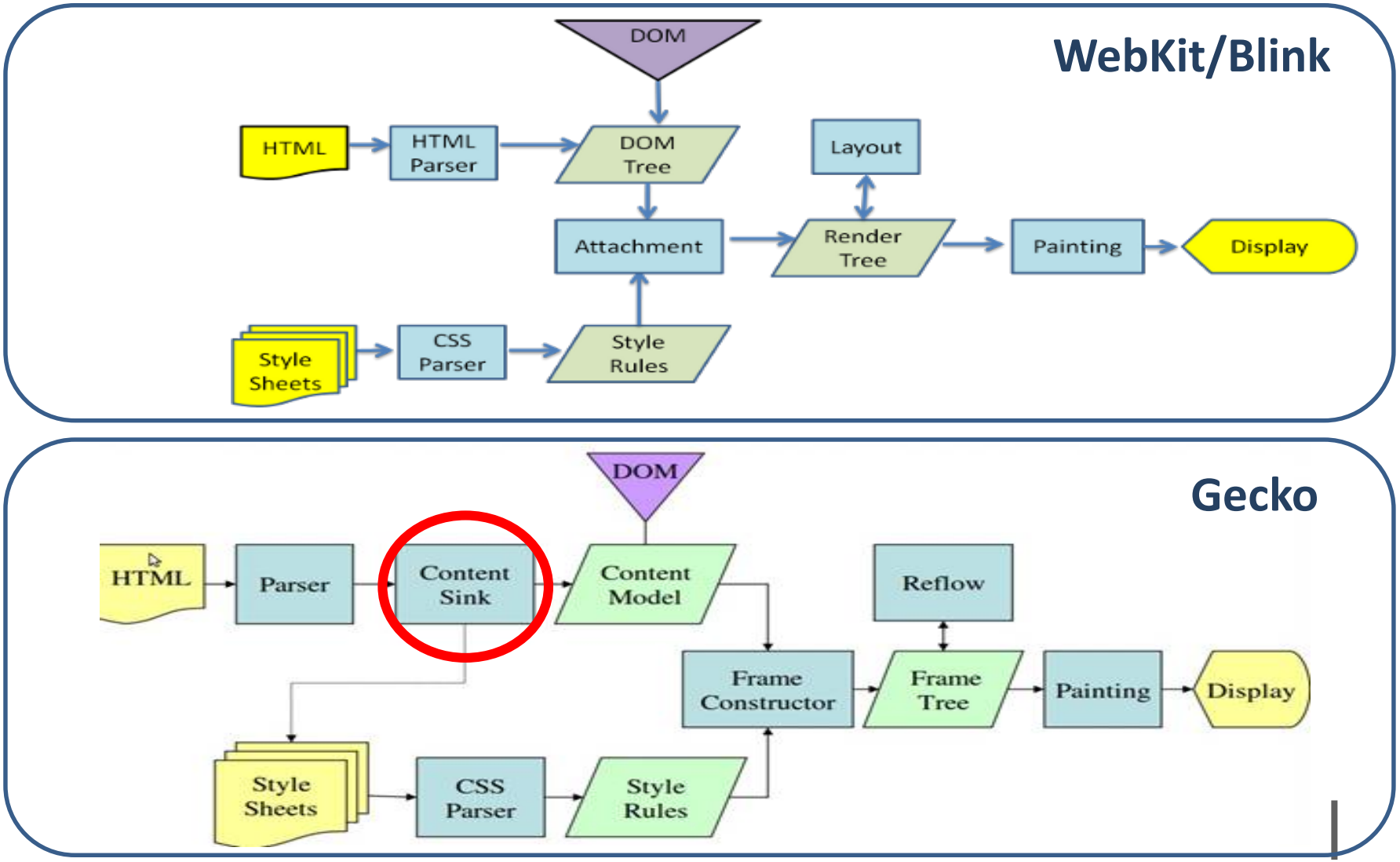
- Si tratta di un processo graduale
- Per migliorare la user experience, il motore di rendering cerca di mostrare il contenuto nella finestra del browser il prima possibile
- Non attenderà il completamento del parsing di tutta la pagina prima di costruire il render tree
- Parte del contenuto sarà “parsata” e mostrata non appena terminato il processo, mentre saranno «parsate» e mostrate le altre parti, mentre queste continuano ad arrivare dal client HTTP



Main Flow di WebKit e Gecko

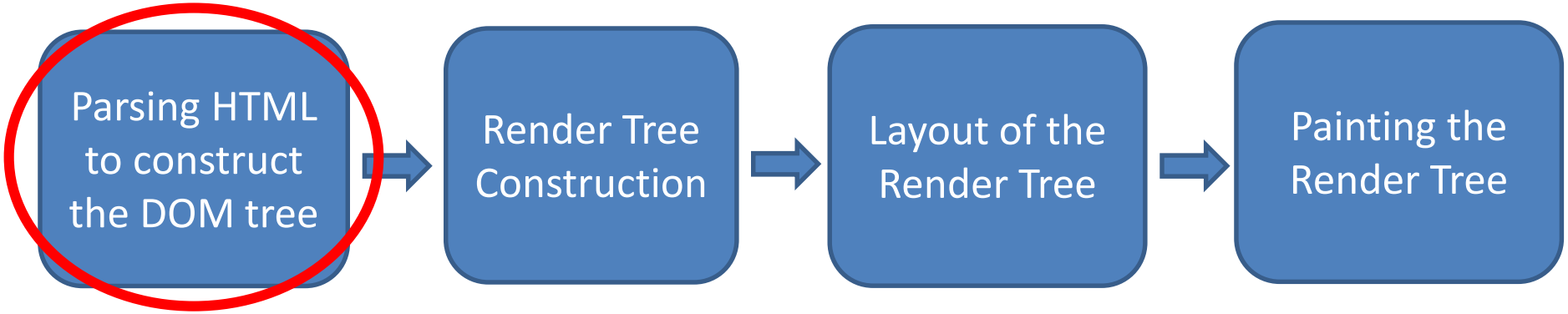


Main Flow di WebKit e Gecko



Parsing dell'HTML

- Vediamo ora le sotto-fasi della prima fase di cui si occupa il motore di rendering, ovvero il parsing del codice HTML (e delle regole di stile) per ottenere la costruzione del DOM tree



Parser HTML

- Il parser HTML prende in input il markup HTML e genera un albero
- Il vocabolario e la sintassi HTML sono definite dallo standard W3C
- HTML non può essere definito da una grammatica context-free come altri linguaggi (es. CSS e JavaScript): HTML può essere definito in modo formale tramite DTD o XMLSchema, ma non si tratta di una grammatica context-free
- HTML consente una sintassi “soft”, ammettendo non conformità (es. certi tag possono essere omessi, in particolare quelli di chiusura)
- Questo comporta facilità di utilizzo da parte degli sviluppatori, ma difficoltà nel definire una grammatica formale e nel realizzare un parser convenzionale, dato che non si tratta di una grammatica context-free

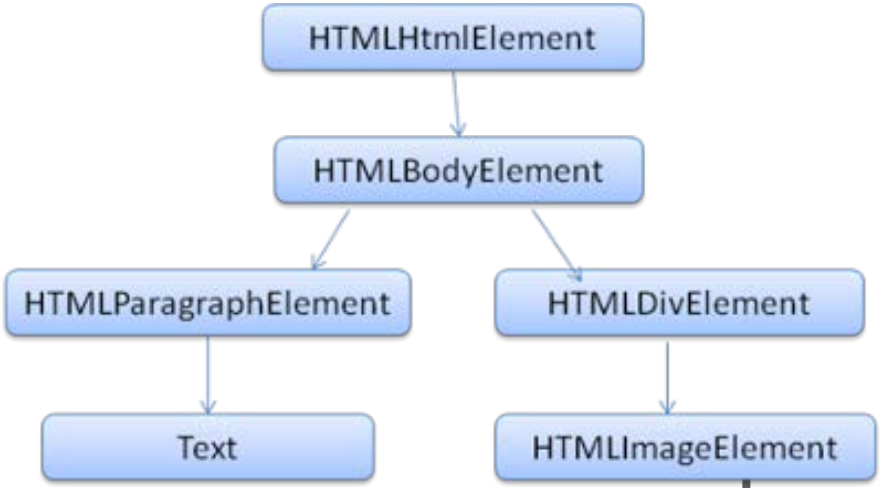
Q&A

- Ci sono domande?

DOM Tree

- Il parsing del codice HTML produce un albero formato da elementi e attributi DOM, o DOM Tree
- Si tratta di una rappresentazione ad albero del documento HTML e dell'interfaccia degli elementi HTML verso il resto del mondo (come JavaScript)
- La radice dell'albero è l'oggetto **`Document`**

```
<html>
  <body>
    <p>Hello World</p>
    <div>
      <img src=''example.png''/>
    </div>
  </body>
</html>
```

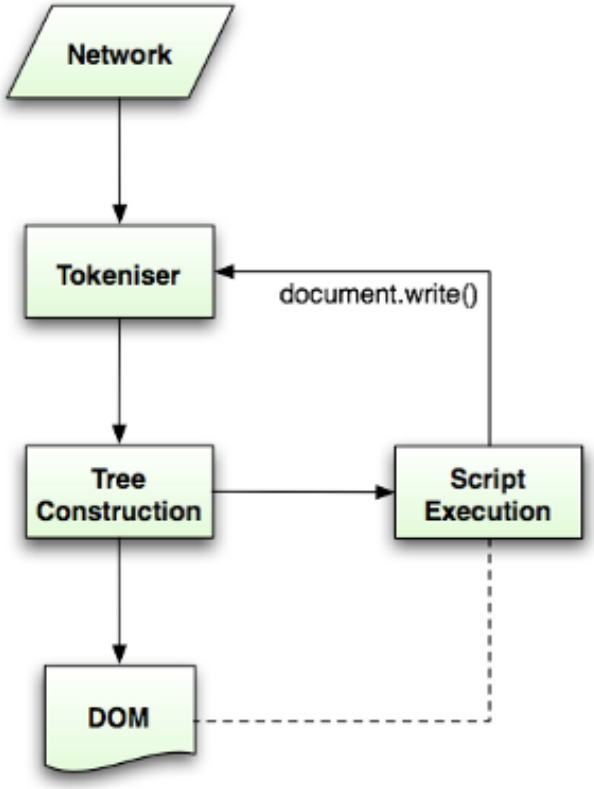


L'algoritmo di parsing

- HTML non può essere "parsato" utilizzando algoritmi di parsing "tradizionali" (ovvero quelli usati per grammatiche context-free) che possono essere top-down oppure bottom-up
- Le principali ragioni sono:
  - I browser per tradizione hanno una elevata tolleranza agli errori nel caso di pagine HTML non valide
  - Il codice sorgente HTML è dinamico e potrebbe cambiare durante il parsing stesso
- I browser quindi hanno sempre adottato dei parser customizzati
- L'algoritmo di parsing (descritto in dettaglio nella specifica HTML5) è caratterizzato da 2 fasi:
  - ***Tokenization***
  - ***Costruzione dell'albero***

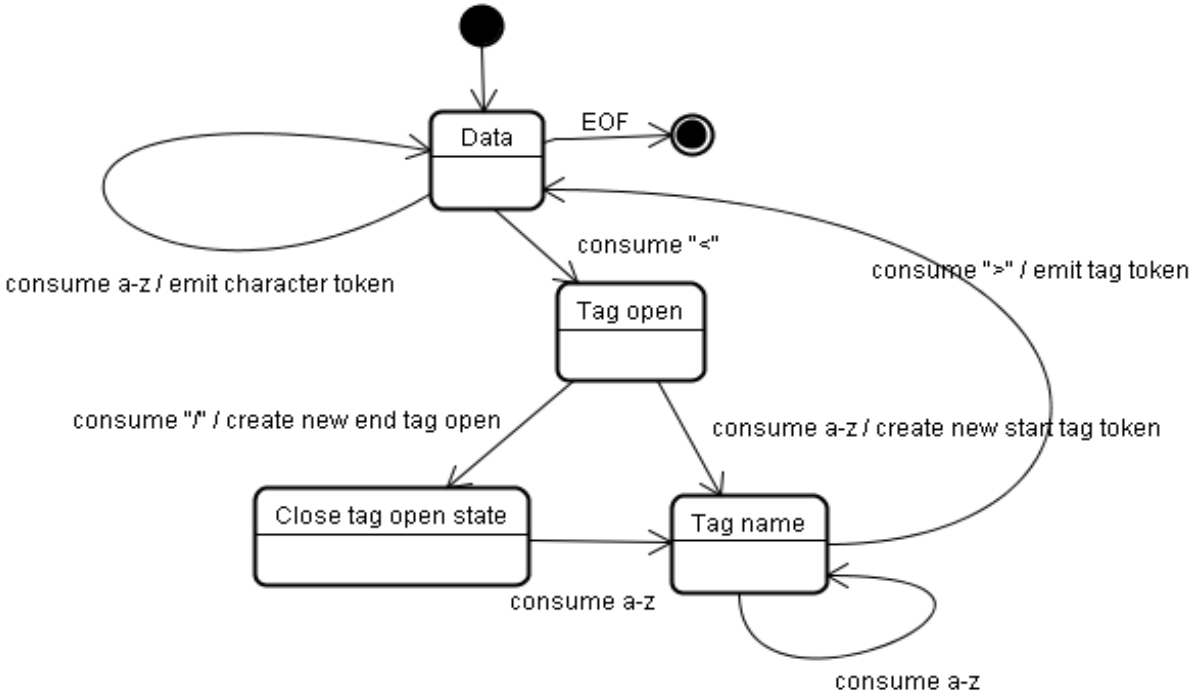
Tokenization

- Si tratta dell'*analisi lessicale*, che si occupa di trasformare codice HTML di input in *token*
- Ad esempio, tra i token HTML ci sono i nomi dei tag di apertura e i valori degli attributi
- Il tokenizer riconosce il token, lo passa al costruttore dell'albero, consuma il carattere successivo per riconoscere il prossimo token e così via



Tokenization: l'algoritmo

- L'output dell'algoritmo di tokenization è un token HTML
- L'algoritmo è espresso come un automa a stati finiti in cui ogni stato consuma uno o più caratteri da una stringa di input e aggiorna lo stato successivo sulla base di questi caratteri



Tokenization: esempio

```
<html>
  <body>
    <p>
     Hello World!
    </p>
  </body>
</html>
```

- Lo stato iniziale è “Data state”
- Quando si incontra il carattere “<”, lo stato viene cambiato in “Tag open state”
- Vengono consumati i caratteri alfabetici a-z, creando un “Start tag token” e modificando lo stato in “Tag name state”, finché il carattere “>” viene consumato
- Viene fatto l’append di ogni carattere al nuovo token (nell’esempio è **`html`**)
- Quando “>” viene raggiunto, il token viene emesso e lo stato torna a “Data state”
- Il tag **<body>** e il tag **<p>** sono trattati con gli stessi passi, fino all’emissione dei token **body** e **p**

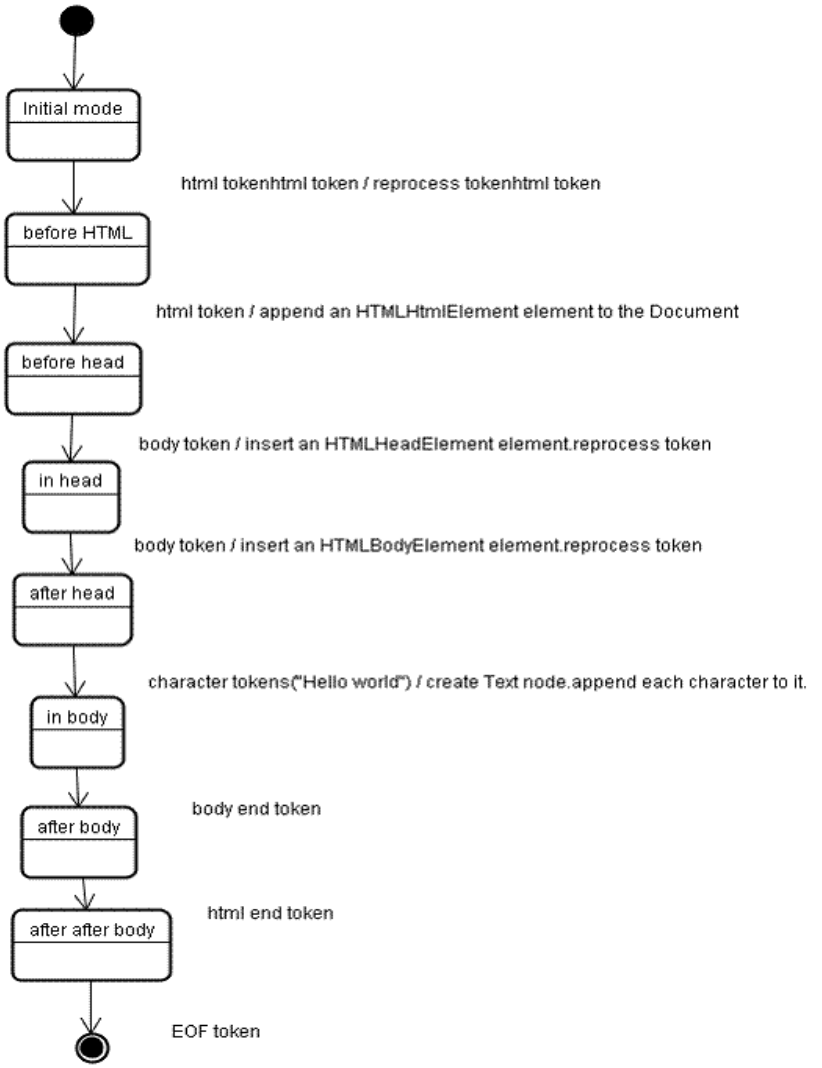
Tokenization: esempio

```
<html>
  <body>
    <p>
     Hello World!
    </p>
  </body>
</html>
```

- …
- Lo stato torna a “Data state”
- Si procede a consumare il carattere “H” della foglia “Hello World”, creando ed emettendo un token del carattere (un token per ogni carattere di “Hello World”), fino al carattere “<” di **</p>**
- Siamo tornati allo stato “Tag open state”, consumando il carattere “/” che comporta la creazione di un token corrispondente ad un tag di chiusura, passando allo stato “Tag name state”, fino a raggiungere il carattere “>”: il token viene emesso e siamo di nuovo allo stato “Data state” (si ripete per **</body>** e per **</html>**)

Tree construction: l'algoritmo

- Ogni nodo emesso dal tokenizer sarà processato dal Tree constructor
- Per ogni token viene definito quale elemento DOM è rilevante e sarà creato per quel token e l'elemento viene aggiunto al DOM tree
- Lo stack degli elementi aperti viene usato per annidare correttamente gli elementi, considerando anche i tag non chiusi
- Anche questo algoritmo è espresso come un automa a stati finiti, i cui stati sono nominati "insertion mode"



Tree construction: esempio

```
<html>
  <body>
    <p>
     Hello World!
    </p>
  </body>
</html>
```

- L'input della fase di costruzione dell'albero è la sequenza di token (dalla tokenization), con modalità iniziale "*initial mode*"
- Viene ricevuto il token **`html`**, che implica il passaggio al "*before html mode*" e reprocessing del token, provocando la creazione dell'elemento **HTMLHtmlElement**, del quale viene fatto l'append all'oggetto radice **`Document`**, modificando lo stato a "*before head*".
- Il token **body** viene ricevuto e un elemento **HTMLHeadElement** viene creato (anche senza il token **head**) e aggiunto all'albero, con passaggio alla modalità "*in head*" e poi in "*after head*".
- …

Tree construction: esempio

```
<html>
  <body>
    <p>
     Hello World!
    </p>
  </body>
</html>
```

- Il token **body** viene riprocessato, un elemento **HTMLBodyElement** viene creato e inserito nell'albero e si passa alla modalità "*in body*"
- Il token **p** viene processato, un elemento **HTMLParagraphElement** viene creato e inserito nell'albero
- I token per la stringa "*Hello World*" sono ricevuti, creando la creazione e l'inserimento di un nodo **`Text`** (con append a questo nodo dei caratteri successivi)
- Una volta ricevuto il token di chiusura di **body** si passa alla modalità "*after body*" e successivamente il token di chiusura di **`html`** provoca la modalità "*after after body*". La ricezione del token di **`end of file`** termina il parsing

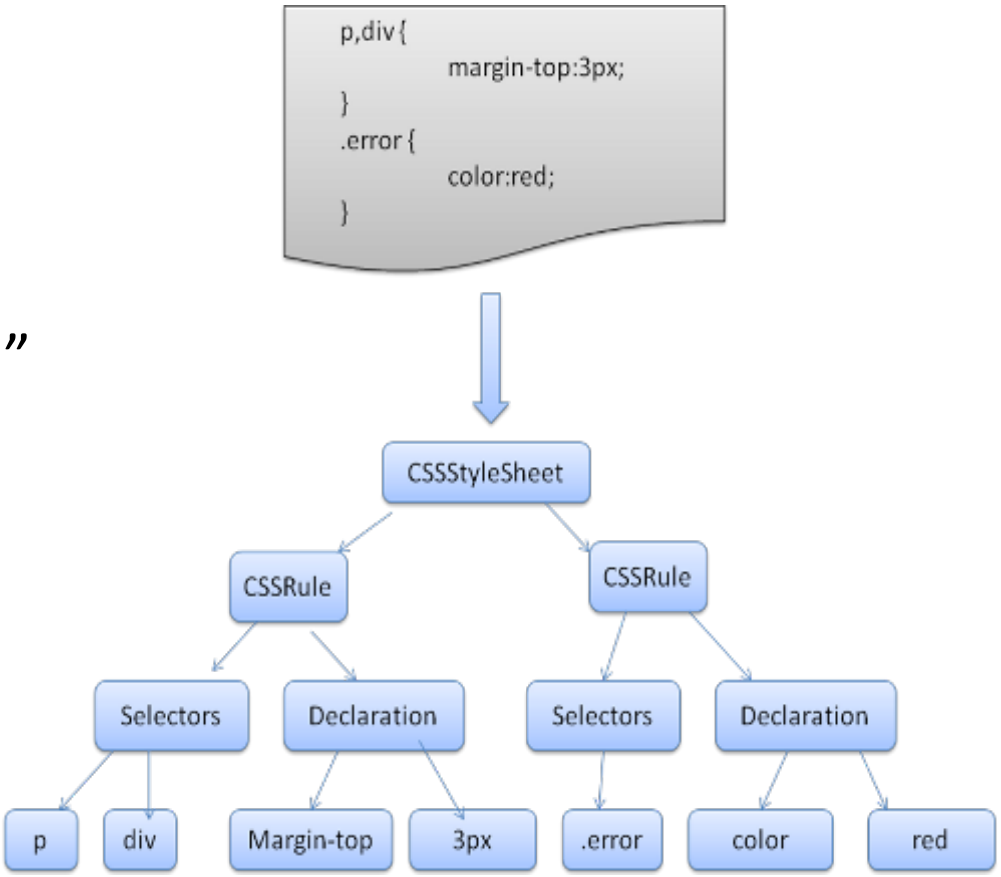
Tolleranza agli errori

```
<html>
  <mytag>
  </mytag>
  <div>
    <p>
  </div>
    Pessimo codice!
  </p>
</html>
```

- I browser operano in modo tale da non restituire mai errori di tipo “Invalid Syntax” sulle pagine HTML, ma si prendono l’onere di correggere errori di invalidità del codice di markup così da poter caricare comunque la pagina
- Nel codice di esempio troviamo tantissimi errori!
- Il browser è in grado di mostrarlo correttamente, grazie al lavoro del parser
- La gestione degli errori è abbastanza consistente tra tutti i browser, anche se non è inclusa nella specifica dell’HTML
- Ci sono errori legati alla non validità di codice HTML che sono abbastanza frequenti e comuni, che vengono gestiti dai browser in modo conforme gli uni con gli altri (sulla base di anni di esperienze simili)

Parsing dei Fogli di stile CSS

- CSS ha una grammatica context-free (la specifica CSS definisce una grammatica sintattica e lessicale), quindi è possibile effettuare il parsing utilizzando parser “tradizionali”
- Dal parsing del CSS risulta un oggetto **StyleSheet**
- Ogni oggetto contiene regole CSS (che contengono il selettore e le dichiarazioni, sulla base della grammatica CSS)

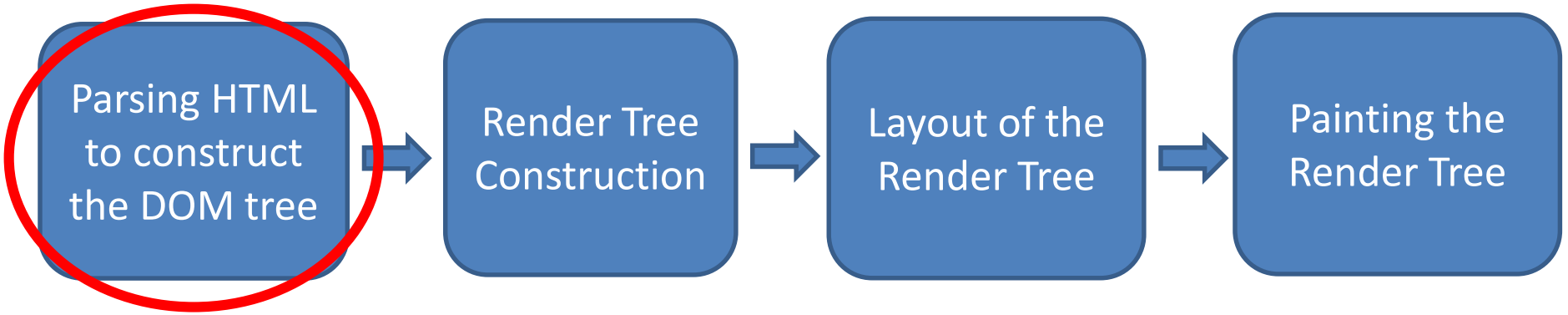


Q&A

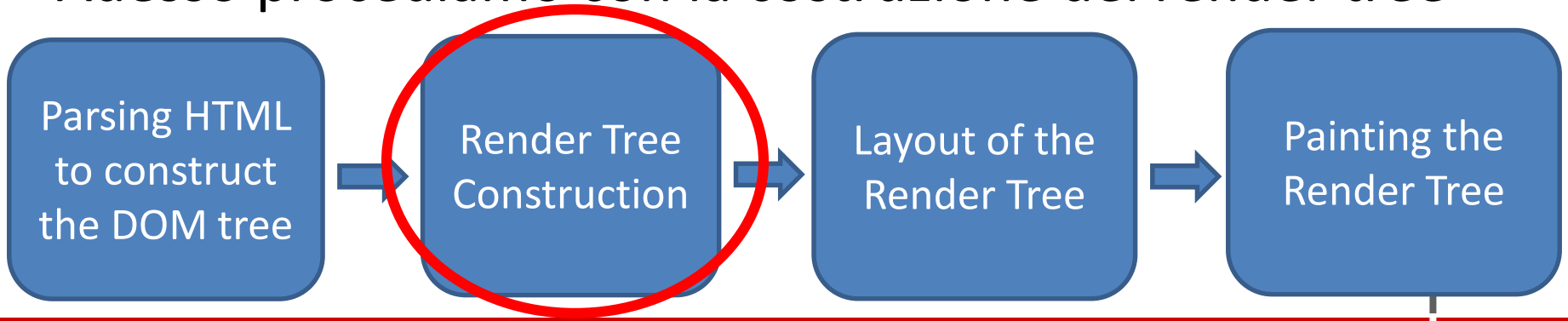
- Ci sono domande?

Basic Flow

- Tutto quanto abbiamo visto finora faceva parte della macro-fase “Parsing HTML to construct the DOM tree”



- Adesso procediamo con la costruzione del render tree

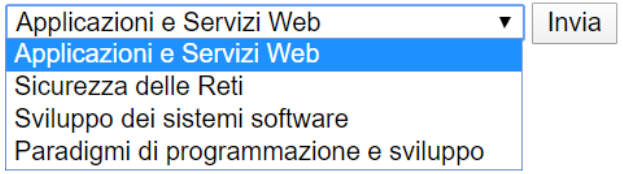


Costruzione del Render Tree

- Mentre viene costruito il DOM Tree, il browser costruisce un altro albero, il Render Tree
- Il Render Tree è composto dagli elementi visuali nell'ordine in cui saranno visualizzati nella finestra del browser
- L'obiettivo di questo albero è quello di agevolare il ***painting*** dei contenuti della pagina Web nell'ordine corretto
- Gecko chiama *frames* gli elementi del Render Tree, mentre WebKit li chiama *renderer* oppure *render object*
- Ogni renderer ha le informazioni relative al suo layout e a come deve essere disegnato, sia per sé stesso che per i suoi figli
- Ogni renderer rappresenta un'area rettangolare che solitamente corrisponde ad un box CSS. Include le informazioni geometriche (**`width`**, **`height`**, **`position`**) e quelle relative al *display* dell'elemento specifico

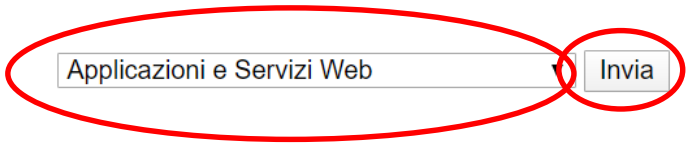
Relazione tra Render Tree e DOM Tree

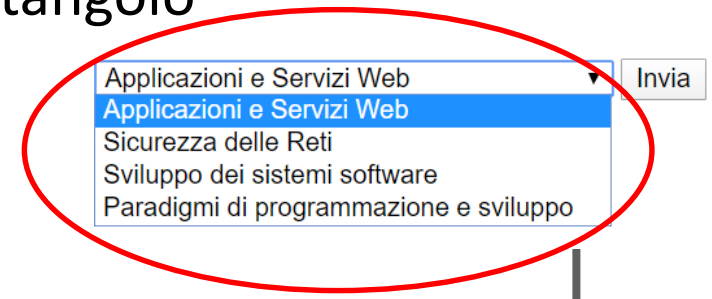
- I renderer corrispondono agli elementi del DOM, anche se la relazioni non è necessariamente uno-a-uno
- Nel DOM ci sono anche elementi non visuali che non sono visualizzati e che quindi non fanno parte del Render Tree (es: **`<head>`** e i metatag, gli elementi che hanno la regola di stile "**`display: none;`**", mentre quelli con la regola "**`visibility: hidden;`**" compaiono nel Render Tree)
- Alcuni elementi del DOM invece corrispondono a più elementi visuali. Alcuni elementi con una struttura complessa non possono essere descritti con un solo rettangolo
- Es: **`<select>`** richiede 3 renderer



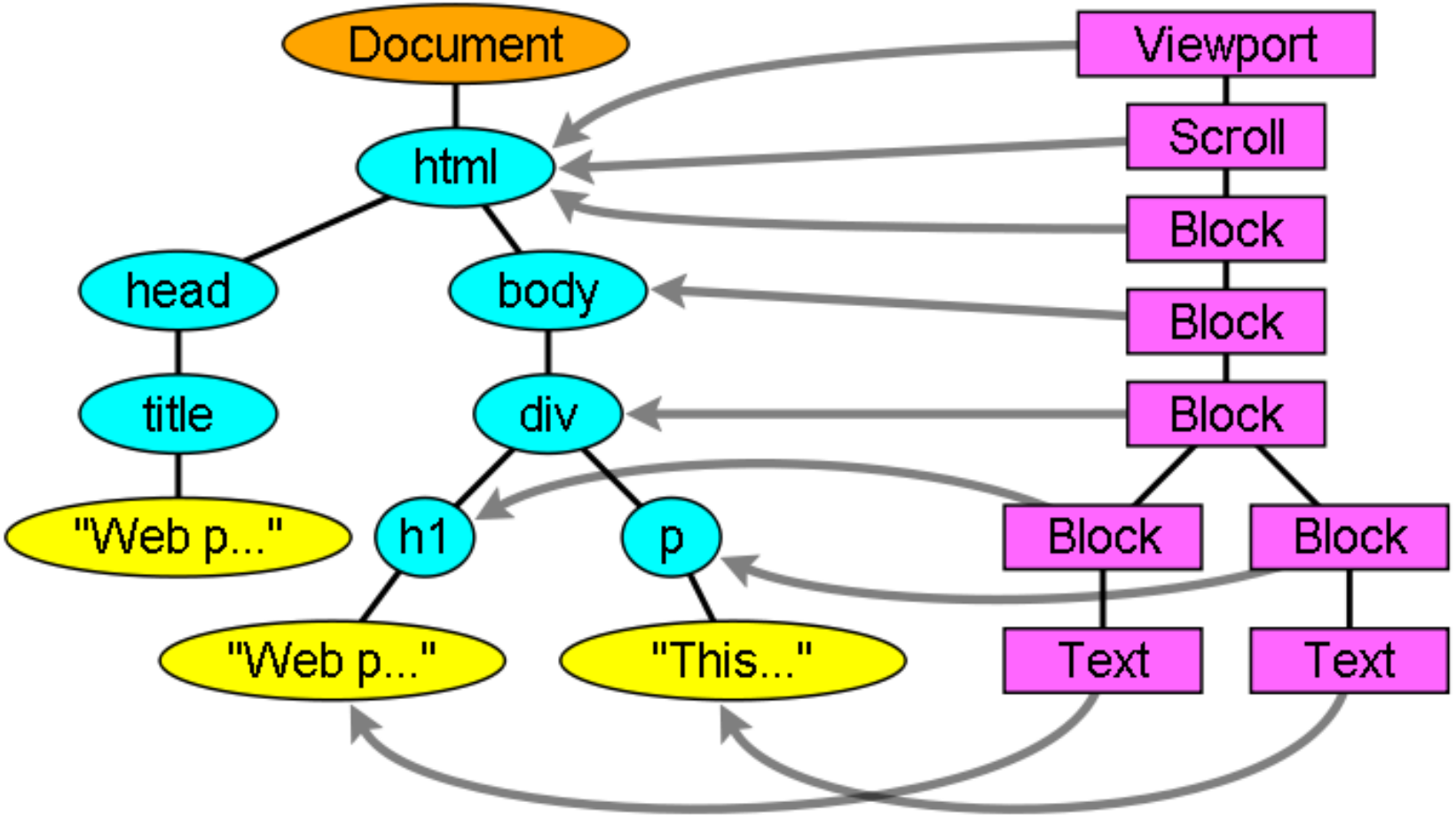
Relazione tra Render Tree e DOM Tree

- I renderer corrispondono agli elementi del DOM, anche se la relazioni non è necessariamente uno-a-uno
- Nel DOM ci sono anche elementi non visuali che non sono visualizzati e che quindi non fanno parte del Render Tree (es: **`<head>`** e i metatag, gli elementi che hanno la regola di stile "**`display: none;`**", mentre quelli con la regola "**`visibility: hidden;`**" compaiono nel Render Tree)
- Alcuni elementi del DOM invece corrispondono a più elementi visuali. Alcuni elementi con una struttura complessa non possono essere descritti con un solo rettangolo
- Es: **`<select>`** richiede 3 renderer

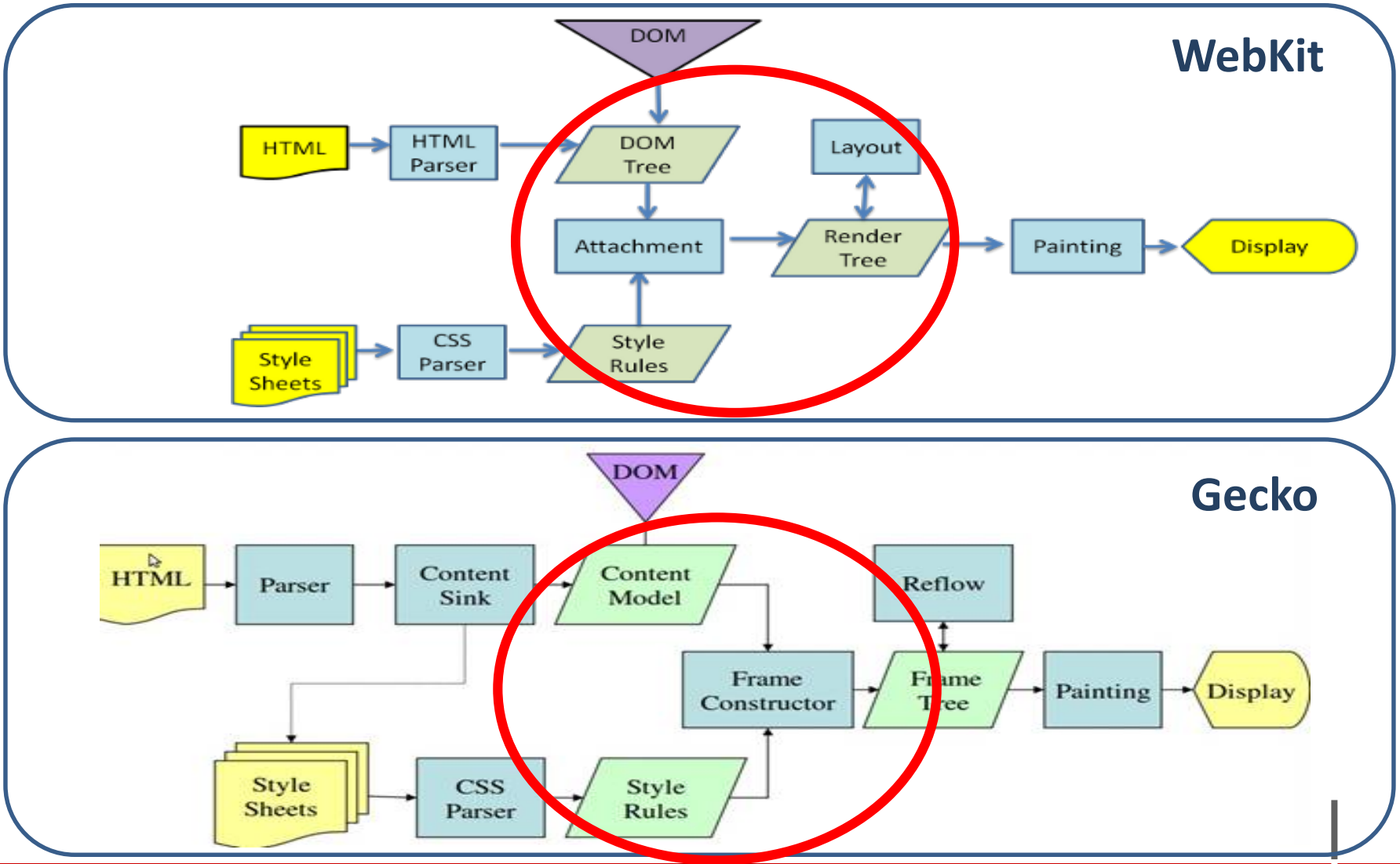




Relazione tra DOM Tree e Render Tree



Main Flow di WebKit e Gecko



Calcolo dello stile di un elemento

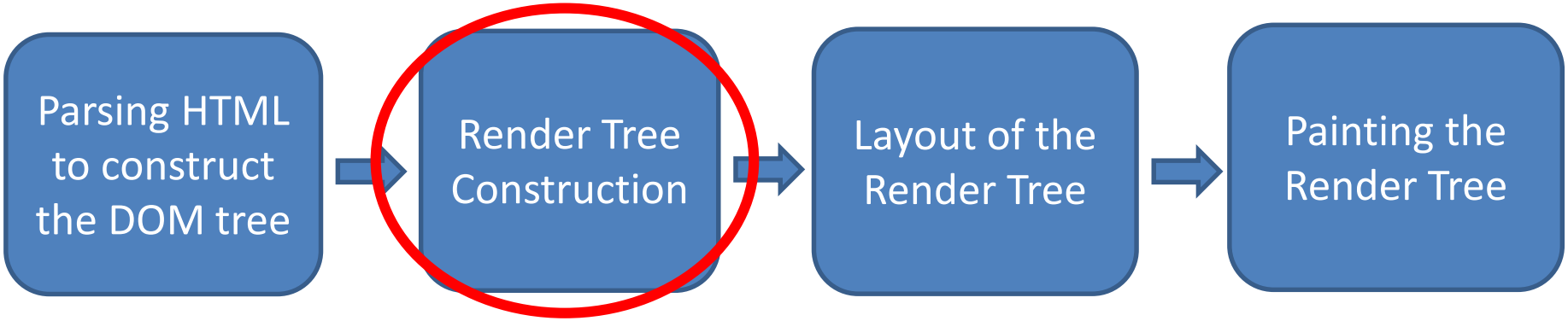
- Costruire il Render Tree significa calcolare le proprietà e le caratteristiche che dovranno essere mostrate da ogni elemento, considerando le sue regole di stile
- Si devono quindi considerare i fogli di stile esterni, interni, inline e le proprietà definite con attribute in HTML (ad esempio l'attributo **`bgcolor`**, anche se deprecato), incluso il foglio di stile del browser ed eventuali proprietà di stile specificate dall'utente
- Il calcolo dello stile per ogni elemento potrebbe essere complesso, a causa di: molteplici regole di stile, fogli di stile e regole multiple (distribuite su più file e su più livelli), selettori con struttura complessa, meccanismo della cascata, diversa priorità per **`id`**, classi e selettori, pseudo-classi e pseudo-selettori

L'ordine della cascata per lo stile

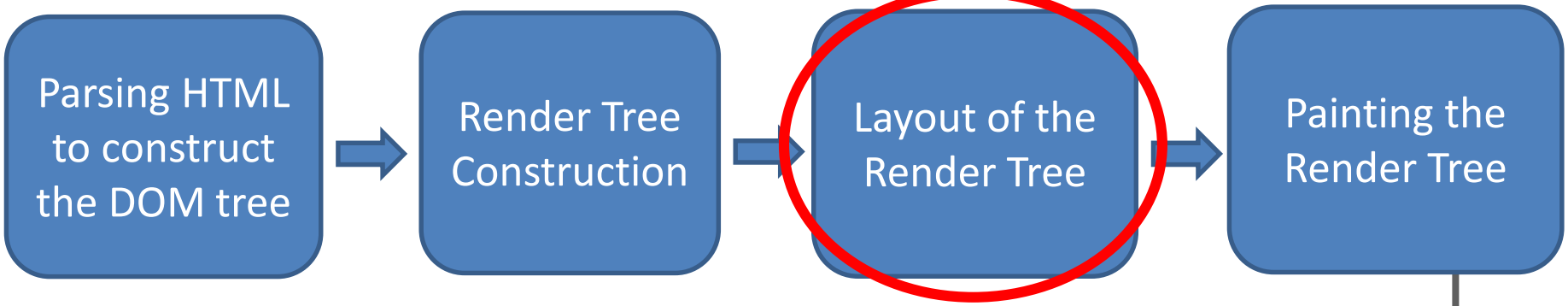
- Lo stile ha proprietà che corrispondono a tutti gli attributi di visualizzazione (colore, larghezza, font-family, ecc)
- Se una proprietà (es: **color**) non è definita esplicitamente da alcuna regola, allora viene ereditata dall'elemento padre
- Alcune proprietà hanno dei valori di default (es: **background-color**)
- Quando ci sono diverse definizioni da più "sorgenti" è necessario considerare il seguente ordine della cascata (dal più basso al più alto):
  - Foglio di stile di default del browser
  - Preferenze dell'utente (che possono essere espresse nel browser)
  - Foglio di stile dell'autore (esterno, interno, inline + attributi HTML)
  - Foglio di stile dell'autore con **`!important`**
  - Preferenze dell'utente con **`!important`**

Basic Flow

- Tutto quanto abbiamo visto finora faceva parte della macro-fase “Render Tree Construction”



- Adesso procediamo con il “Layout del Render Tree”



Q&A

- Ci sono domande?

Layout

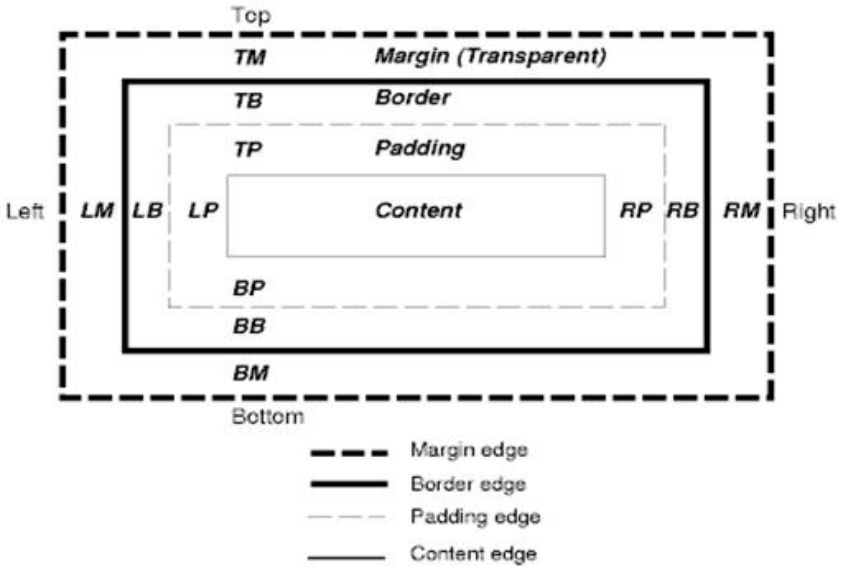
- Una volta che un renderer viene creato e aggiunto all'albero, ancora non ha associate le informazioni relative al **position** e alle **dimensioni**
- Il calcolo di questi valori è chiamato ***Layout*** (o ***Reflow***)
- Le coordinate solo relative rispetto all'area di visualizzazione del browser, l'origine degli assi è l'angolo in alto a sinistra (**`top`**, **`left`**)
- Questa fase è un processo ricorsivo: comincia con il root renderer (che corrisponde a **`<html>`**) e continua ricorsivamente, calcolando le informazioni geometriche richieste da ogni renderer
- La posizione del root renderer è **`0,0`** e le sue dimensioni sono quelle del viewport

Layout

- Il calcolo del layout solitamente segue questo iter:
  - La larghezza **`width`** del renderer è determinata da quella del suo elemento padre
  - L'elemento padre definisce per ogni figlio:
    - La posizione (impostando le coordinate x e y, rispetto a origine degli assi)
    - Chiama il calcolo per il layout del filio, calcolando la sua altezza **`height`**
  - L'elemento padre utilizza le altezze dei figli (considerando anche **`margin`** e **`padding`**) per calcolare la propria altezza (che viene utilizzata a sua volta dal suo elemento padre, ovvero dall'elemento nonno o grand-parent)

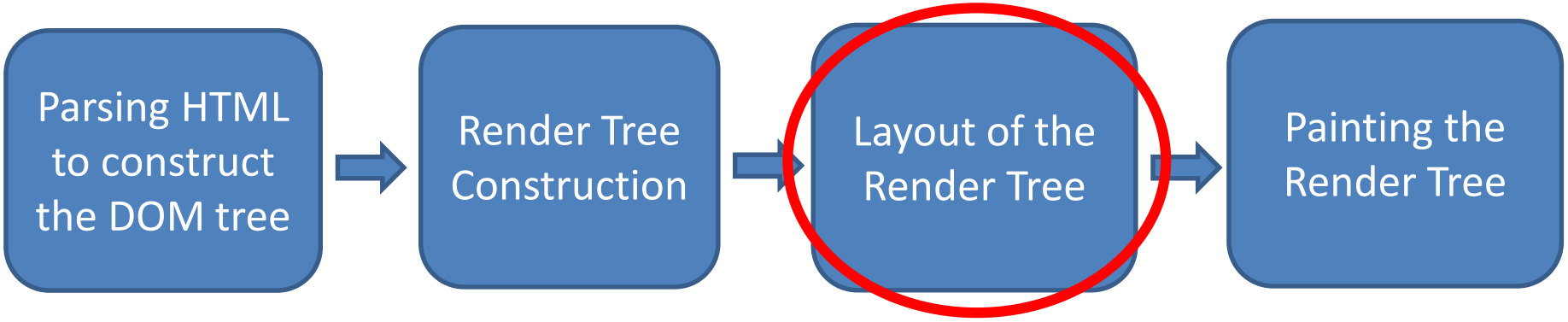
Calcolo della larghezza

- La larghezza **`width`** del renderer è calcolata usando la larghezza dell'elemento di blocco che contiene il renderer, sulla base del valore per la proprietà **`width`** nel foglio di stile, i margini e i bordi
- Esempio: **`<div style="width: 30%"/>`**
- La **`width`** del contenitore è calcolata come la **`width`** della finestra del browser (esclusi bordi e scrollbar) a cui vengono sottratti il **`padding-left`** e il **`padding-right`**
- La larghezza dell'elemento **`<div>`** viene calcolata come valore della percentuale della **`width`** del contenitore
- Successivamente vengono aggiunti gli spessori per i bordi orizzontali e i **`padding`**

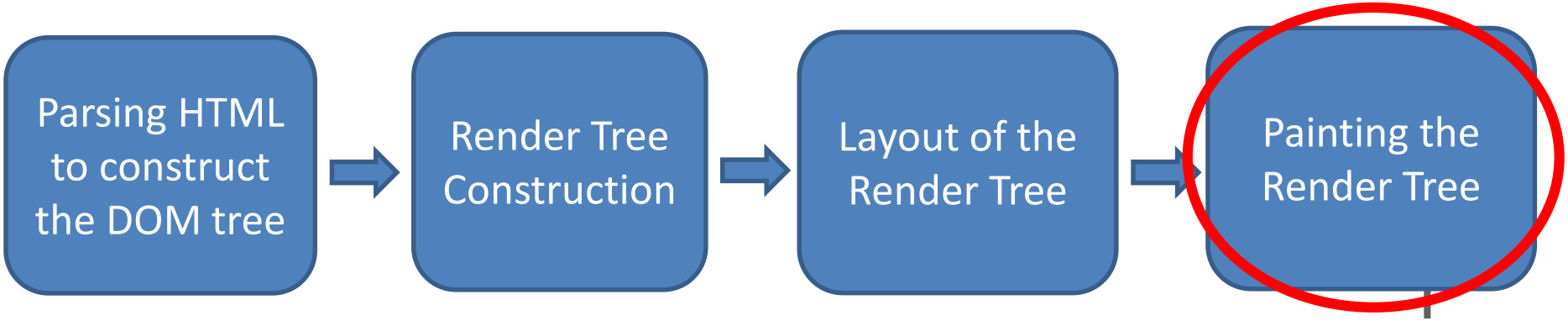


Basic Flow

- Tutto quanto abbiamo visto finora faceva parte della macro-fase “Layout of the render tree”



- Adesso procediamo con il “Painting del Render Tree”



Painting

- Il render tree viene visitato/navigato per poter effettivamente “disegnarne” il contenuto sullo screen
- L’ordine del processo di *painting* è definito dalla specifica CSS, cominciando dal background andando verso il foreground
- L’ordine per ogni renderer di blocco è il seguente:
  1. Background-color
  2. Background-image
  3. Border
  4. Children
  5. Outline

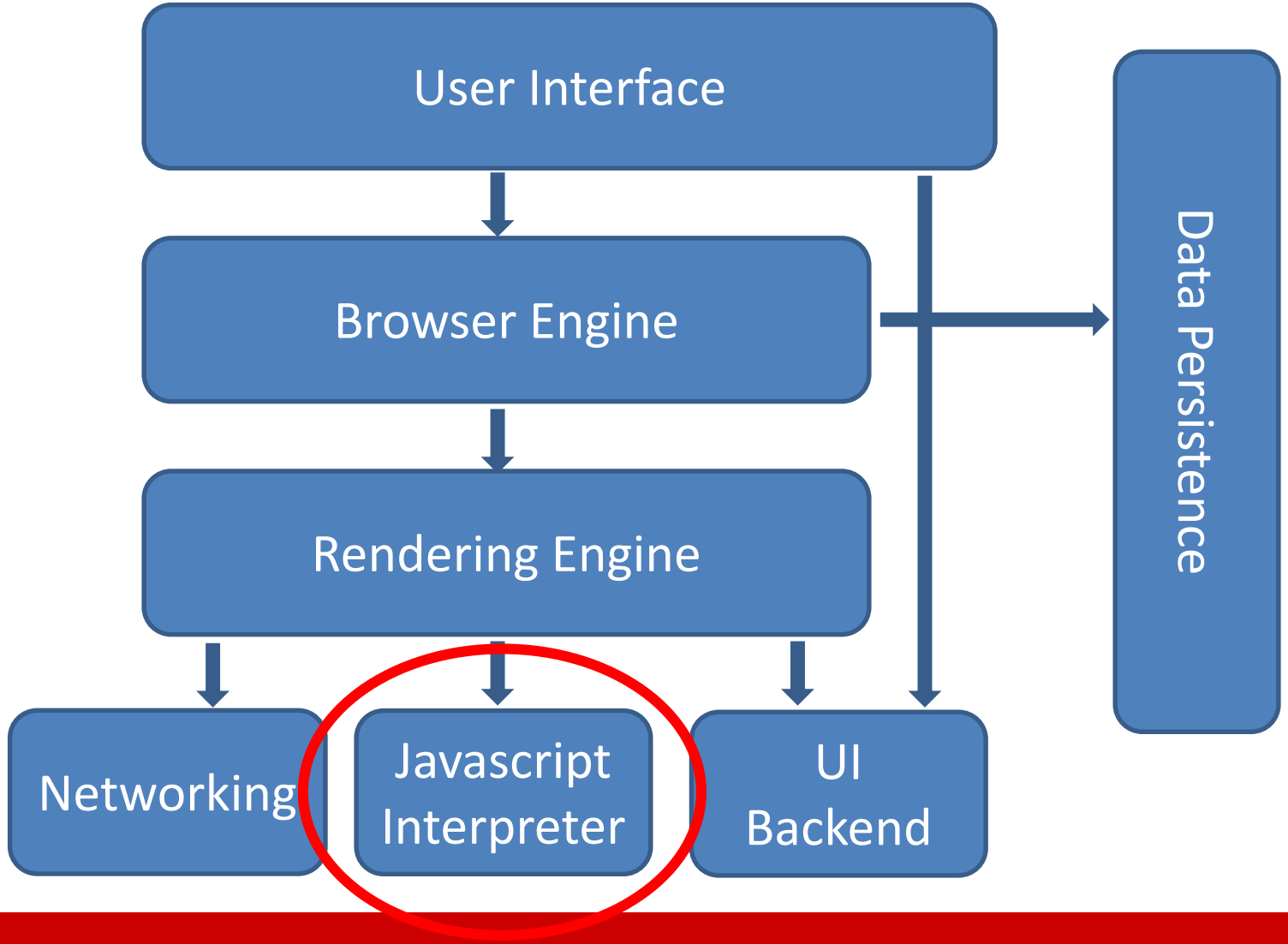
Painting

- Il browser visita il render tree e crea una lista degli elementi da disegnare per ogni rettagolo disegnato
- La lista contiene i renderer rilevanti per il rettangolo, nell'ordine corretto da seguire per disegnarli (**background**, bordi, ecc)

Q&A

- Ci sono domande?

Browser: overview dell'architettura



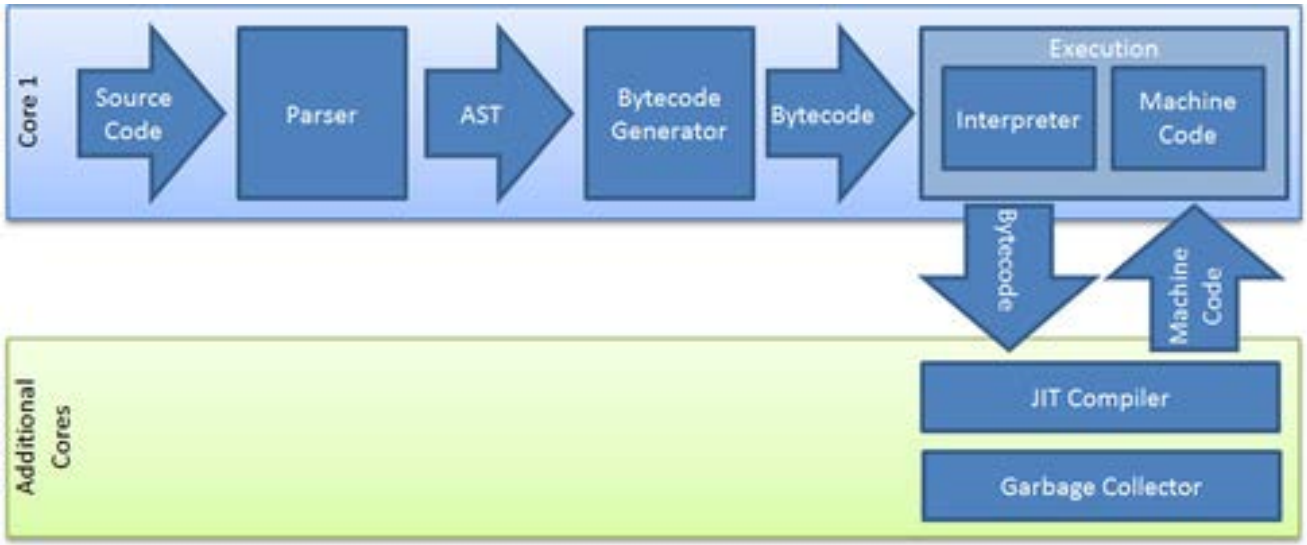
Interprete Javascript

- I browser attuali includono un motore dedicato all'esecuzione del codice JavaScript
- Componente separato dal motore di rendering, con cui condivide il DOM di ogni pagina Web renderizzata
- Alcuni esempi
  - V8 (in Chrome, con Blink, e in Node.js, sviluppato da Google)
  - SpiderMonkey (in Firefox, con Gecko, sviluppato da Mozilla, nato per Netscape Navigator)
  - JavaScriptCore (in Safari, sviluppato da Apple)

V8 Javascript engine

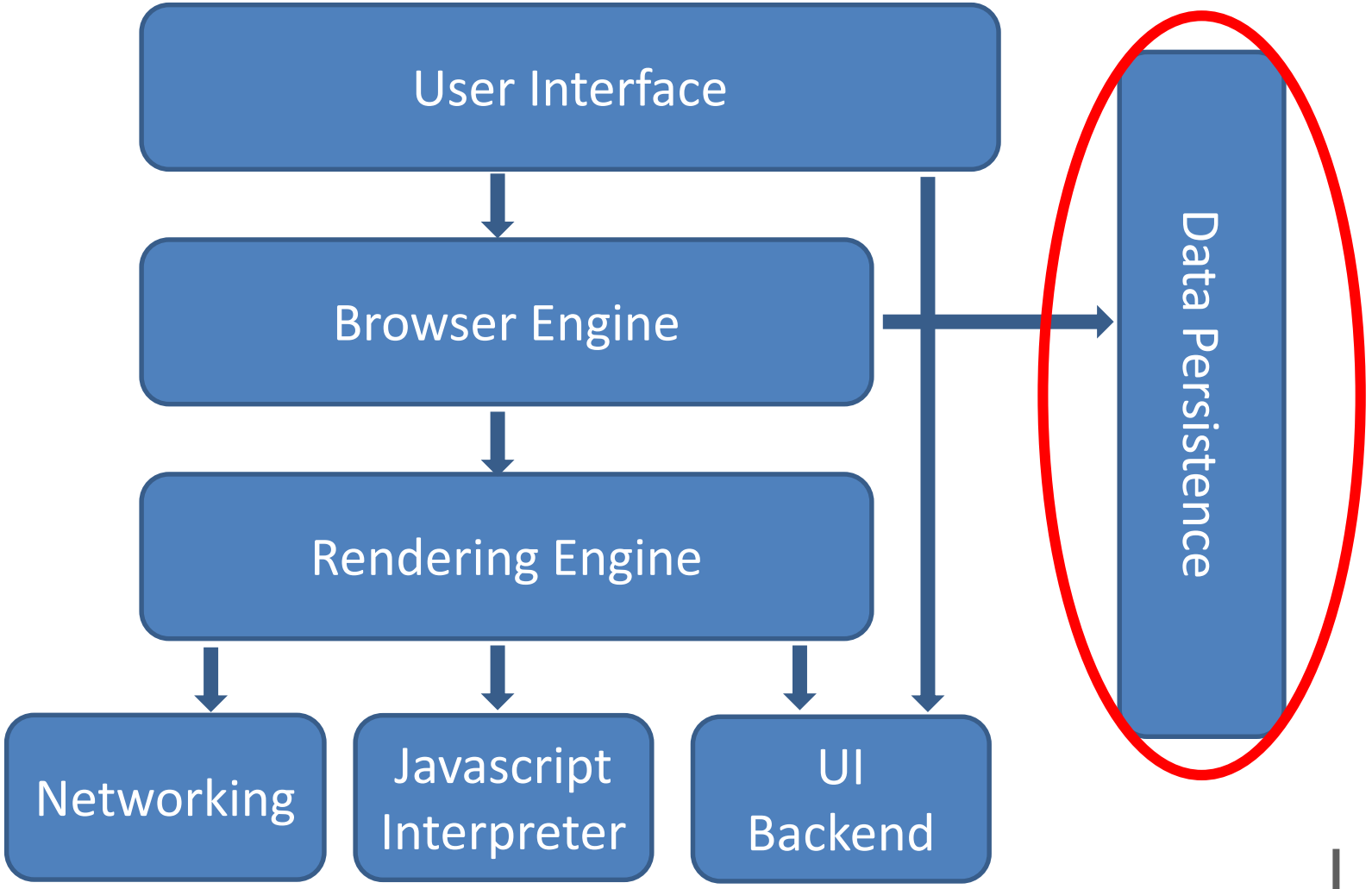
- Scritto in C++, supporta ECMAScript
- Progettato da Google con l'obiettivo di migliorare le performance dell'esecuzione di JavaScript all'interno dei browser
- Per ottenere questi miglioramenti, V8 traduce il codice JavaScript in un codice macchina più efficiente, invece di usare un interprete
- V8 compila JavaScript in codice macchina all'esecuzione, implementando un compilatore JIT (Just-In-Time), come fanno altri Javascript engine (es: SpiderMonkey)

V8 Javascript engine



- Quando V8 riceve il codice di uno script, esegue il parsing per creare un Abstract Syntax Tree (AST)
- L'AST è dato in input al Bytecode Generator per produrre il bytecode
- Il bytecode è interpretato con l'aiuto del Just-In-Time compiler, per produrre il linguaggio macchina nativo che viene eseguito

Browser: overview dell'architettura



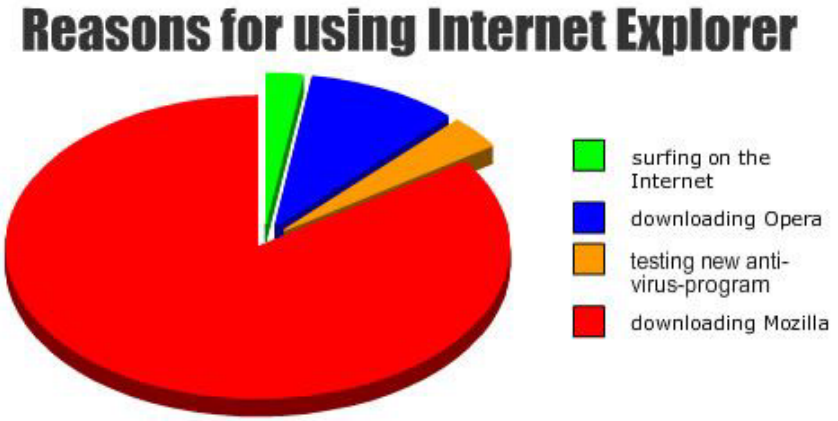
Data Persistence

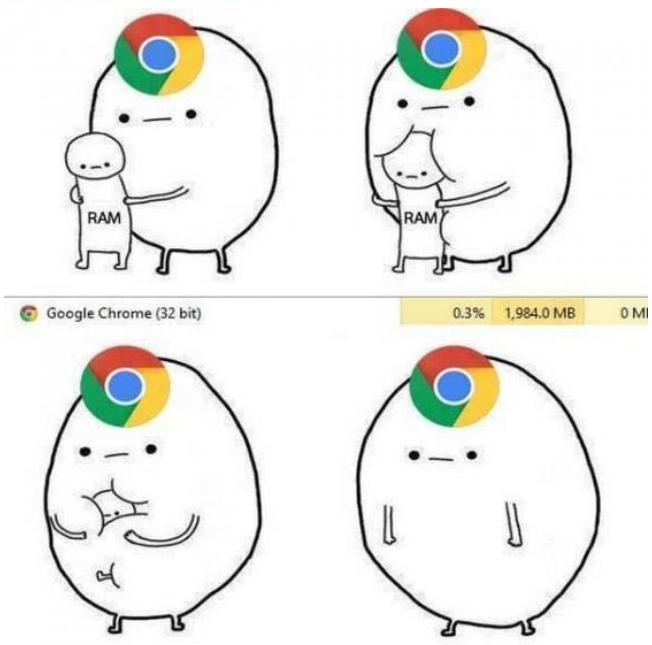
- Si occupa di ospitare e gestire i dati e le informazioni che il browser deve salvare localmente, come ad esempio i cookie (altri meccanismi di storage supportati dal browser tramite specifiche API: localStorage, sessionStorage, WebSQL, IndexedDB, FileSystem, ecc)
- Si tratta di una specie di database creato in locale, sul calcolatore su cui il browser è installato
- Gestisce i dati dell'utente (cache, cookie, bookmark e preferenze dell'utente)

Data Persistence

- Tra le varie API del browser che consentono l'archiviazione dei dati ci sono le seguenti:
  - **Local storage** e **Session Storage**: permettono di archiviare dati, oggetti e funzioni JS nel browser. **Session storage** permette di mantenere dati sul browser fintanto che la sessione è attiva. **Local storage** invece mantiene i dati in locale sul browser in modo persistente. Il limite è 5MB per storage e di 50 MB per sistema
  - **Cookies**: sono scambiati tra browser e server, molto utili in contesti specifici, come ad esempio quelli correlati agli aspetti di privacy e sicurezza, oppure possono essere utilizzati per profilare l'utente. Sono meno performanti e considerati "costosi" rispetto agli altri metodi
  - **AppCache**: introdotta come API di HTML5. Memorizza contenuti statici
  - **Service Workers**: API introdotta da Google, permette di fare caching dei dati di siti web per il loro utilizzo in modalità offline. Le funzionalità sono molto simili a quelli di AppCache

Web Browser: Memes





If Internet Explorer can be brave enough to ask you to set him as your default browser, please don't tell me you can't dare ask a girl out.

Q&A

- Ci sono domande?

Riferimenti

- Behind the scene: https://www.youtube.com/watch?v=z0HN-fG6oT4
- Grosskurth, Alan, and Michael W. Godfrey. "A reference architecture for web browsers." In 21st IEEE International Conference on Software Maintenance (ICSM'05), pp. 661-664. IEEE, 2005.
- Garsiel, Tali, and Paul Irish. "How browsers work: Behind the scenes of modern web browsers." Google Project, August (2011).
- Arun Sena Bojja. “How do Web Browsers work?” (April 21st 2019), https://hackernoon.com/how-do-web-browsers-work-40cefd2cb1e1
- Monica Raghuwanshi. «How does web browsers work?» (May 18, 2017), https://medium.com/@monica1109/how-does-web-browsers-work-c95ad628a509
- HTML - Living Standard (Last Updated 14 March 2020), https://html.spec.whatwg.org/multipage/
- CSS Syntax Module Level 3, W3C Candidate Recommendation, 16 July 2019, https://www.w3.org/TR/css-syntax-3/
- V8 Documentation https://v8.dev/docs
- Alexander Zlatkov. ”How JavaScript works: inside the V8 engine + 5 tips on how to write optimized code” (Aug 21, 2017) https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e