

브라우저 동작과정 뜯어보기

개요

- 브라우저가 웹 페이지를 화면에 표시하기 위한 브라우저의 기본적인 동작과정을 파악한다.
- 브라우저의 역할과 종류, 엔진과 구조에 대해서 학습한다.
- 웹 페이지를 렌더링 하기 위한 주요 경로를 학습하고 개발자 도구로 분석한다.

프론트엔드 개발자에게는 기본 소양이자 필수 지식, 면접 단골 질문!

웹 애플리케이션 구동 과정

1. URL entered : 사용자가 웹 브라우저에서 사이트 주소를 입력한다.
2. DNS Lookup : DNS 를 이용하여 사이트 주소에 해당되는 Server IP 를 접근한다.
3. Socket Connection : Client (브라우저) 와 Server 간 접속을 위한 TCP 소켓 연결.
4. HTTP Request : Client 에서 HTTP Header 와 데이터가 서버로 전송.
5. Content Download : 해당 요청이 Server 에 도달하면 사용자가 원하는 문서를 다시 웹 브라우저에 전송한다.

6. Browser Rendering : 웹 브라우저의 렌더링엔진에서 해당 문서를 다음과 같은 순서로 파싱

- HTML 를 DOM (Document Object Model) 으로 변환
- CSS 를 DOM 에 추가 (CSSOM 생성)
- DOM 으로 렌더트리 생성
- 렌더트리 배치
- 렌더트리 그리기

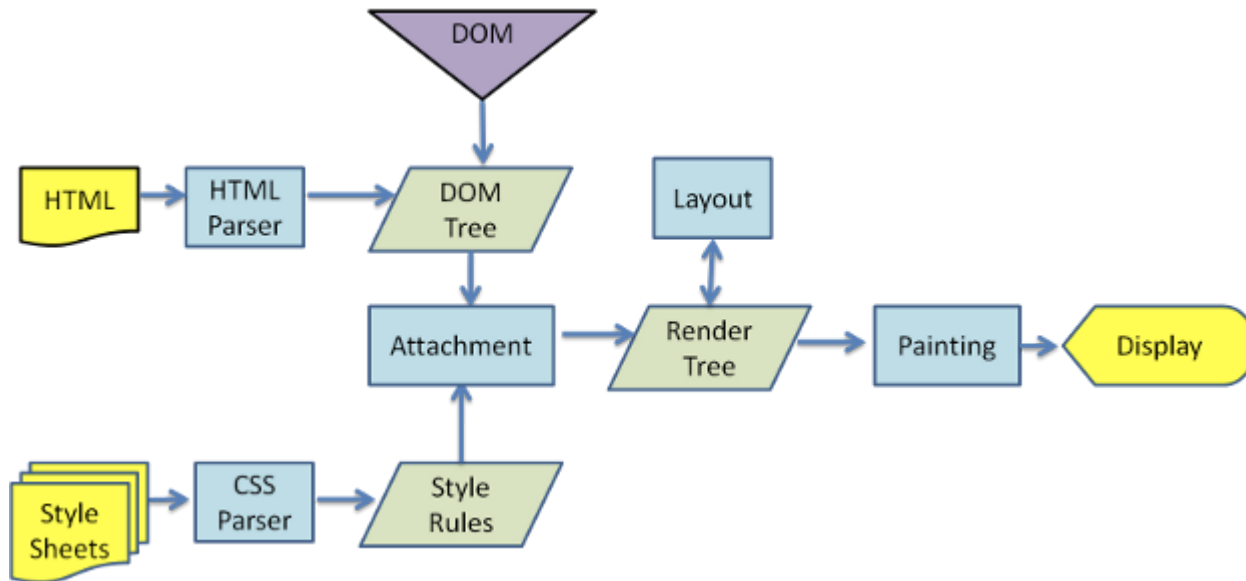
7. Display Content : 렌더트리를 브라우저에 표시 후 사용자에게 웹 페이지로 보여준다.

Browser 역할 & 종류

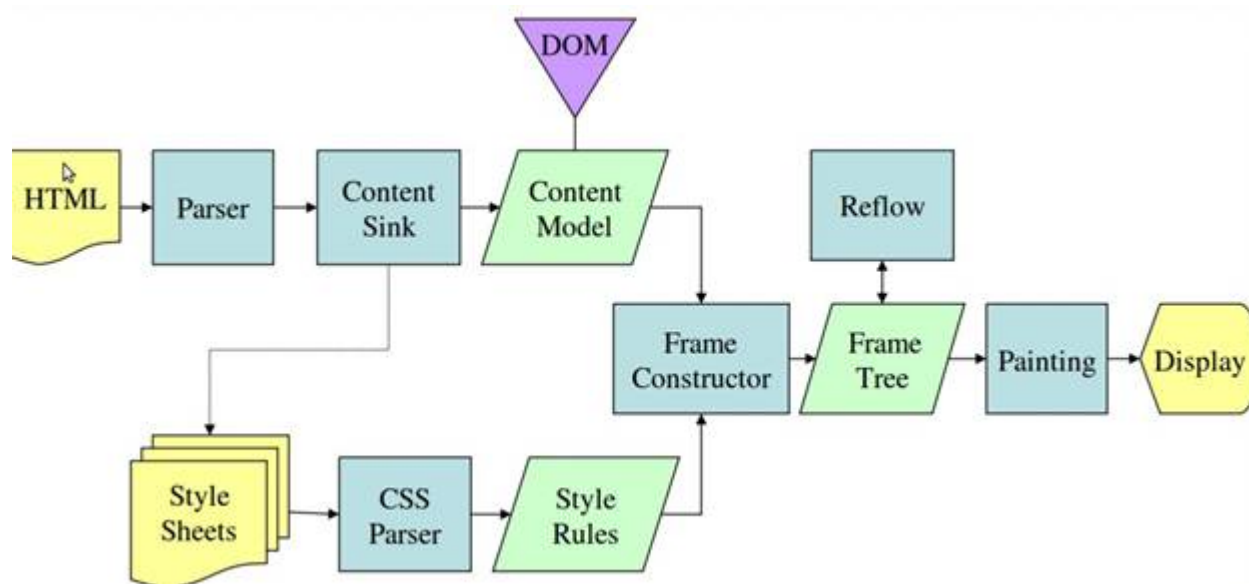
- 사용자가 선택한 자원 (URL) 을 서버에 요청하고 받아 화면에 표시
- 주요 브라우저
 - Google Chrome - Webkit
 - Safari - Webkit
 - Mozilla Firefox (Escape) - Gecko
 - Microsoft Internet Explorer
 - Opera

브라우저 엔진

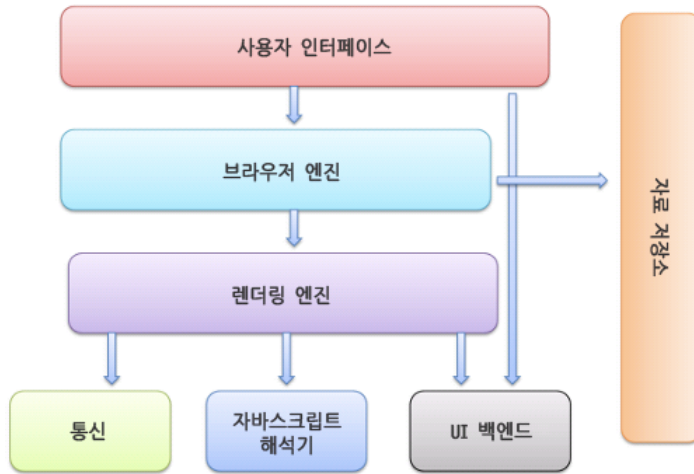
- Webkit : Google, Apple 이 공동 개발한 오픈소스 기반 엔진. 주요 모바일 브라우저가 모두 웹킷 기반



- Gecko : C++ 기반 엔진, Mozilla 에서 유지보수 수행중, 상업용 오픈소스가 아니라 일반 개발자도 참여 가능



Browser 기본 구조



- UI : 주소 창, 즐겨찾기 등 사용자가 조작 가능한 영역
- 브라우저 엔진 : UI 와 렌더링 엔진 동작 제어
- 렌더링 엔진 : 요청된 자원을 화면에 표시
- 네트워킹 : HTTP 요청과 같은 네트워크 호출
- UI 백엔드 : OS 사용자 인터페이스 방법을 활용하여 기본적인 위젯 (콤보 박스 등)을 그림
- 자바스크립트 인터프리터 : 자바스크립트를 해석하고 실행
- 데이터 저장소 : Local Storage, Indexed DB, 쿠키 등 브라우저 메모리를 활용하여 저장하는 영역

렌더링 엔진

- 서버로부터 요청받은 내용을 브라우저에 표시하는 역할
- 동작과정
 - HTML -> DOM 파싱
 - Render Tree 구축
 - Render Tree 배치
 - Render Tree 그리기

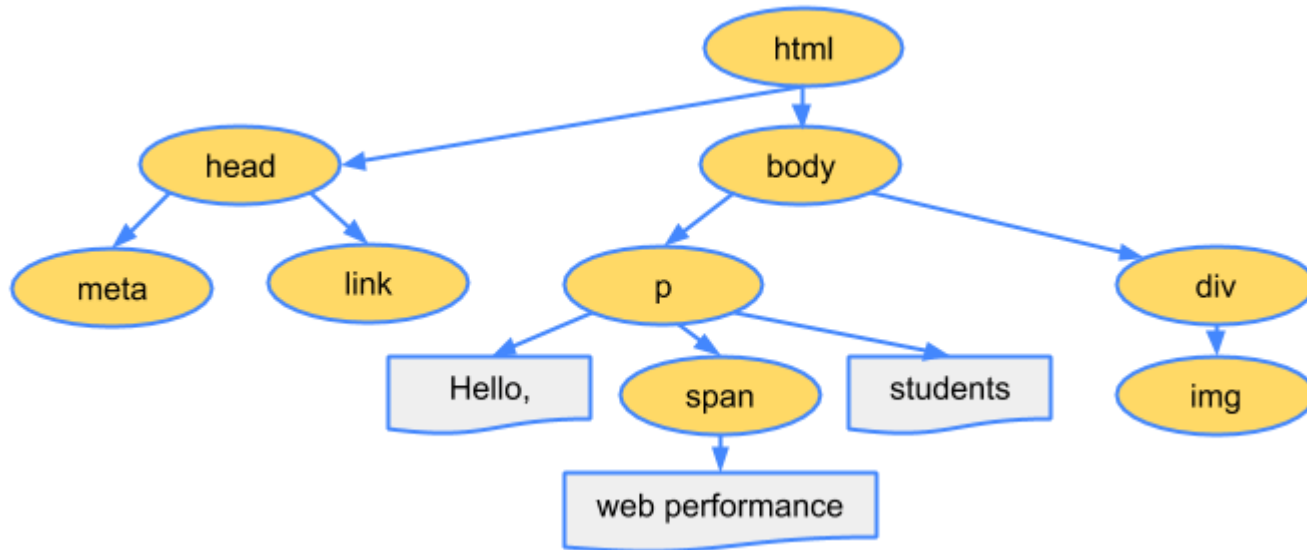
Render Tree : HTML 요소 + CSS 스타일링 정보를 포함한 트리, DOM + CSSOM

Critical Rendering Path - 주요 렌더링 경로 소개

- 브라우저가 HTML, CSS, Javascript 등의 파일을 변환하여 화면에 픽셀 단위로 나타내기 위해 거쳐야 하는 일련의 과정
- 렌더링 최적화의 과정은 항상 *측정을 먼저*하고 *최적화를 진행*해야 한다.

DOM (Document Object Model)

- HTML 의 내용과 속성을 노드 (오브젝트) 로 갖고 각 노드의 관계를 나타내는 트리
- HTML 문서를 구조화 하여 스크립트 또는 프로그래밍 언어에서 접근 가능한 형태로 제공한다.



HTML 의 DOM 변환 과정

URL 에서 사이트 주소를 했을 때 서버에서 아래와 같은 문서를 브라우저에 넘겨준다고 하자.

```
<html>
  <head>
    <meta charset="utf-8">
    <title>...</title>
  </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

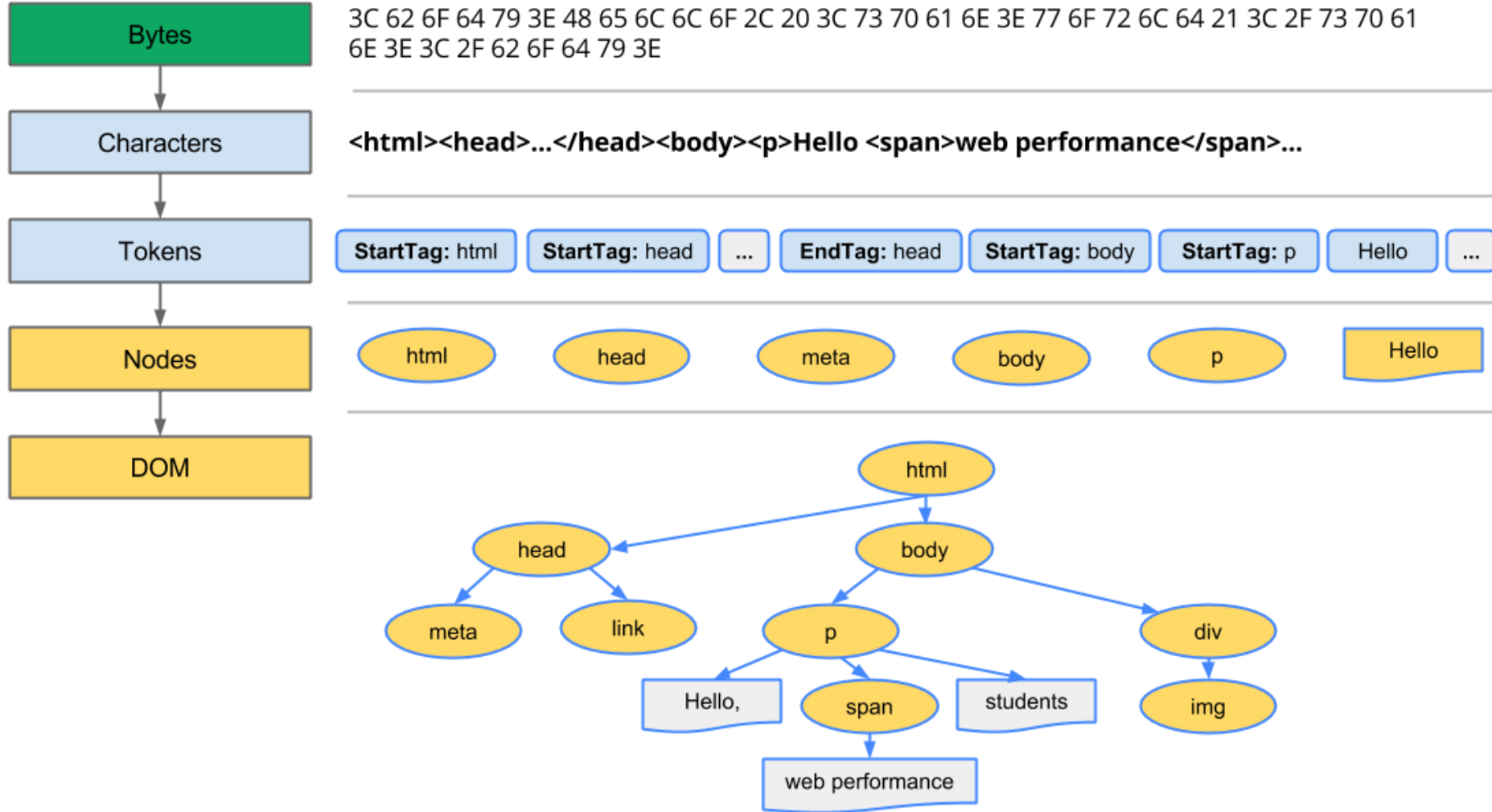
위와 같은 HTML 문서를 다음과 같이 변환한다.

1. **태그 -> 토큰**, HTML 태그를 토큰라이저를 이용하여 토큰으로 변환
2. **토큰 -> 노드**, 토큰을 Tree 구조의 노드로 변환
3. 모두 변환된 노드를 이용하여 DOM 을 구성

아래 html 코드 파싱하는 과정을 직접 그려보기

- 준비물 : 종이, 펜

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div></div>
  </body>
</html>
```



결론 : 바이트 → 문자 → 토큰 → 노드 → DOM

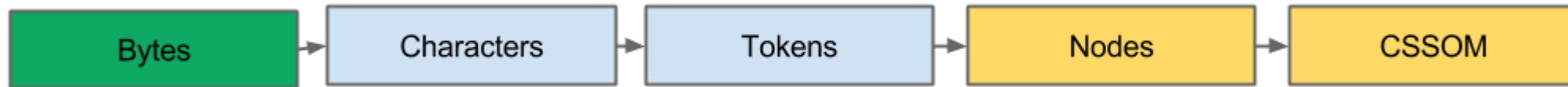
Google 의 Incremental HTML Delivery

- 구글 메인 화면에서 검색 쿼리를 날리면, 검색 결과 페이지의 헤더만 일단 받아와서 DOM 을 생성하고, 화면에 뿌려준다.
- 그리고 나서, 검색 결과에 따라 나머지 HTML 의 DOM 을 생성하고, 화면에 렌더한다.
- 이와 같이 사용자의 반응에 따라 HTML 을 순차적으로 화면에 그리는 것이 성능에 도움이 된다.

위 동작 Dev Tool 로 확인

CSSOM (CSS Object Model)

- DOM 생성과 마찬가지로 body, p 와 같은 토큰들을 노드로 변환하여 CSS Obejct Model 로 변환한다.



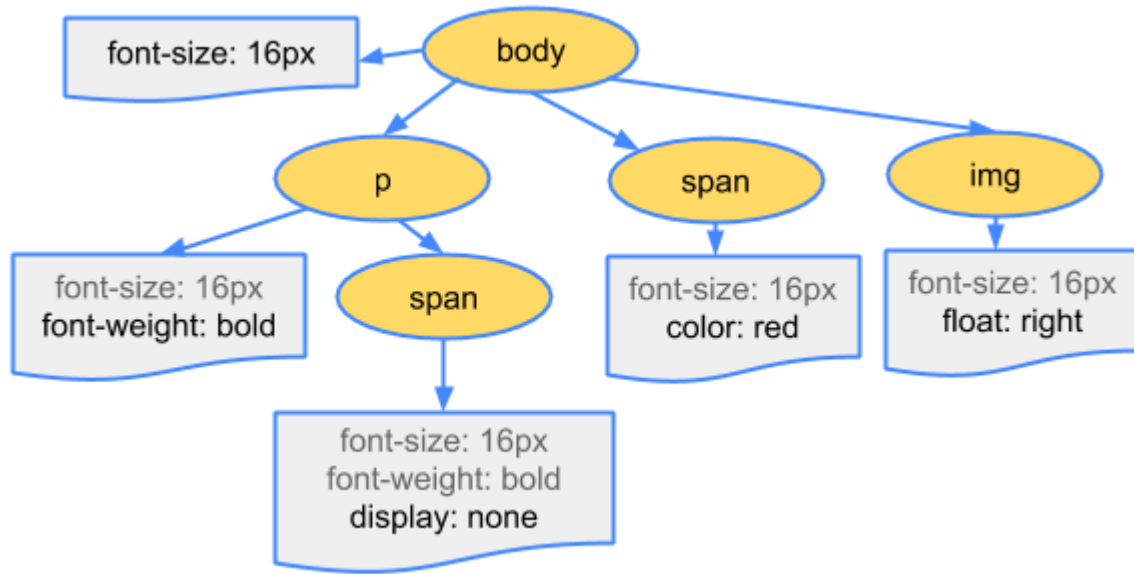
- Cascading Style Sheets 는 Body 와 같이 페이지 구조상 상위에 있는 HTML 요소의 스타일이 하위 요소에 상속된다는 의미
- CSS 는 페이지 렌더링을 방해한다. 브라우저가 모든 CSS 를 파싱하고 처리할 때까지 페이지가 화면에 그려지지 않는다.

- 개발자 콘솔의 타임라인에서 Recalculate Style 시 CSSOM 를 생성함

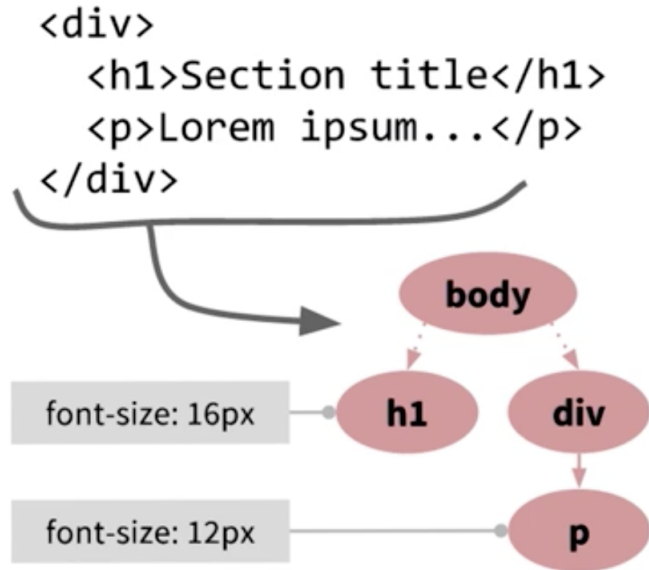
1408.9ms	0.1ms	0.1ms	Receive Response
1417.1ms	0.0ms	0.0ms	Receive Data
1417.5ms	0.1ms	0.1ms	Finish Loading
1418.8ms	74.8ms	75.1ms	Parse HTML
1495.9ms	0.0ms	0.0ms	Update Layer Tree
1500.8ms	0.9ms	0.9ms	Recalculate Style
1501.7ms	0.1ms	0.2ms	Layout
1501.9ms	0.1ms	0.1ms	Update Layer Tree
1502.0ms	0.0ms	0.0ms	Paint

```
body { font-size: 16px }  
p { font-weight: bold }  
span { color: red }  
p span { display: none }  
img { float: right }
```

위 css model 은 아래와 같은 형태로 생성된다.



브라우저 관점에서 어떤 CSS 룰이 효과적일까?



Q) 위와 같은 이미지에서 `h1` 과 `div p` 중 어느 스타일 속성이 브라우저 관점에서 효과적일까?

A) `h1` 와 같이 일차원적 선택자. `h1` 과 같은 일반 지정자는 바로 접근이 가능하고, `div p` 같은 경우에는 `p` 를 찾은 후 다시 DOM 을 거슬러 올라가, 오직 `div` 를 부모 요소로 갖고 있을 때 속성을 적용하기 때문에 브라우저는 더 많은 일을 하게 된다.

CSS 선택자 접근은 오른쪽에서 왼쪽 순으로 읽는다고 생각!

Render Tree

- 브라우저가 DOM + CSSOM 을 가지고 화면의 픽셀로 변환하려면 Render Tree 가 필요하다.
- Render Tree 는 DOM 과 CSSOM 을 조합하여 오직 화면에 표시할 요소들만 포함한다.
- DOM Tree 의 노드에 그대로 스타일을 입혀 Render Tree 로 전환되는 것은 아님
 - `<head>` 나 `display:none` 등의 비가시적인 태그들은 트리에 포함되지 않음

```
// 렌더 트리 C++ 클래스 명세
class RenderObject {
    virtual void layout();
    virtual void paint(PaintInfo);
    virtual void rect repaintRect();
    Node* node; //the DOM node
    RenderStyle* style; // the computed style
    RenderLayer* containingLayer; //the containing z-index layer
}
```

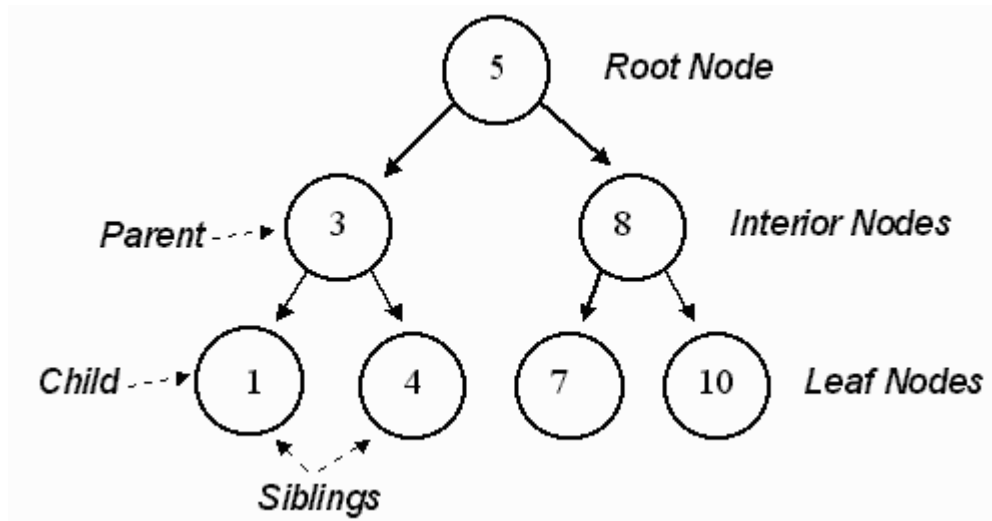
Render Tree 구성

렌더 트리는 아래 4가지 트리를 조합하여 구성

- RenderObjects : DOM 에 상응
- RenderStyles : DOM 요소에 적용되는 Style 과 상응
- RenderLayers : DOM 요소들이 화면에서 실제로 차지하는 위치나 크기 지정
- Line boxes : 텍스트를 구성하는 textbox 와 같은 요소의 비율 조정

Tree

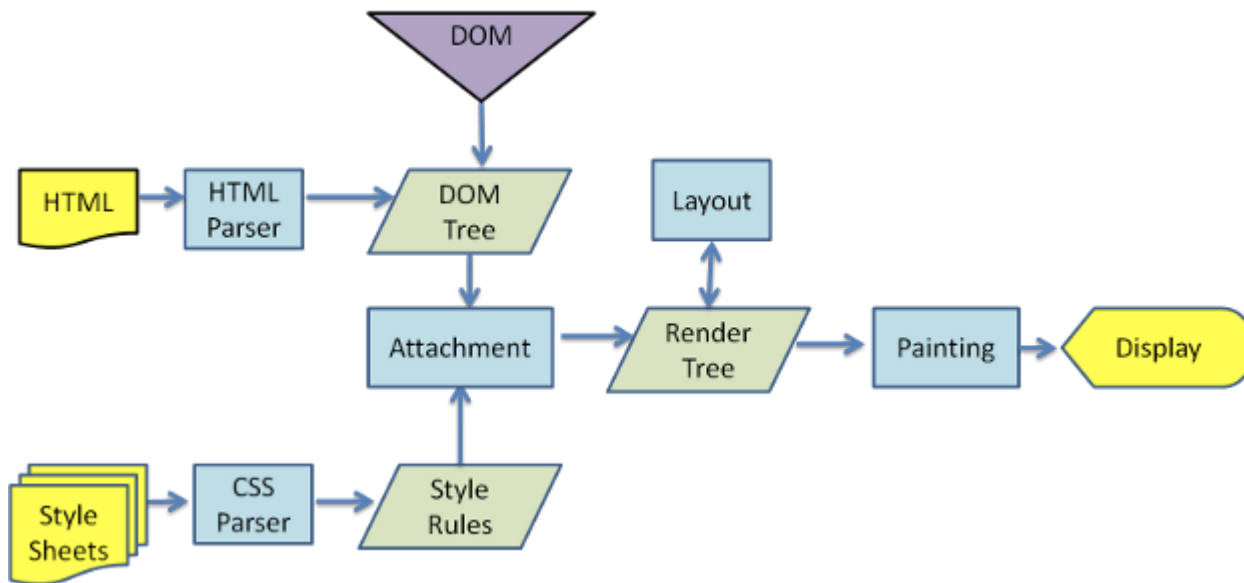
- 부모 - 자식 관계로 정립된 트리 형태의 자료구조
- Linux / Windows 의 파일시스템의 기반이며 주요 DB 구조의 기반이기도 하다.
- AI 의 의사결정 트리 등 다양한 분야에서 사용 중



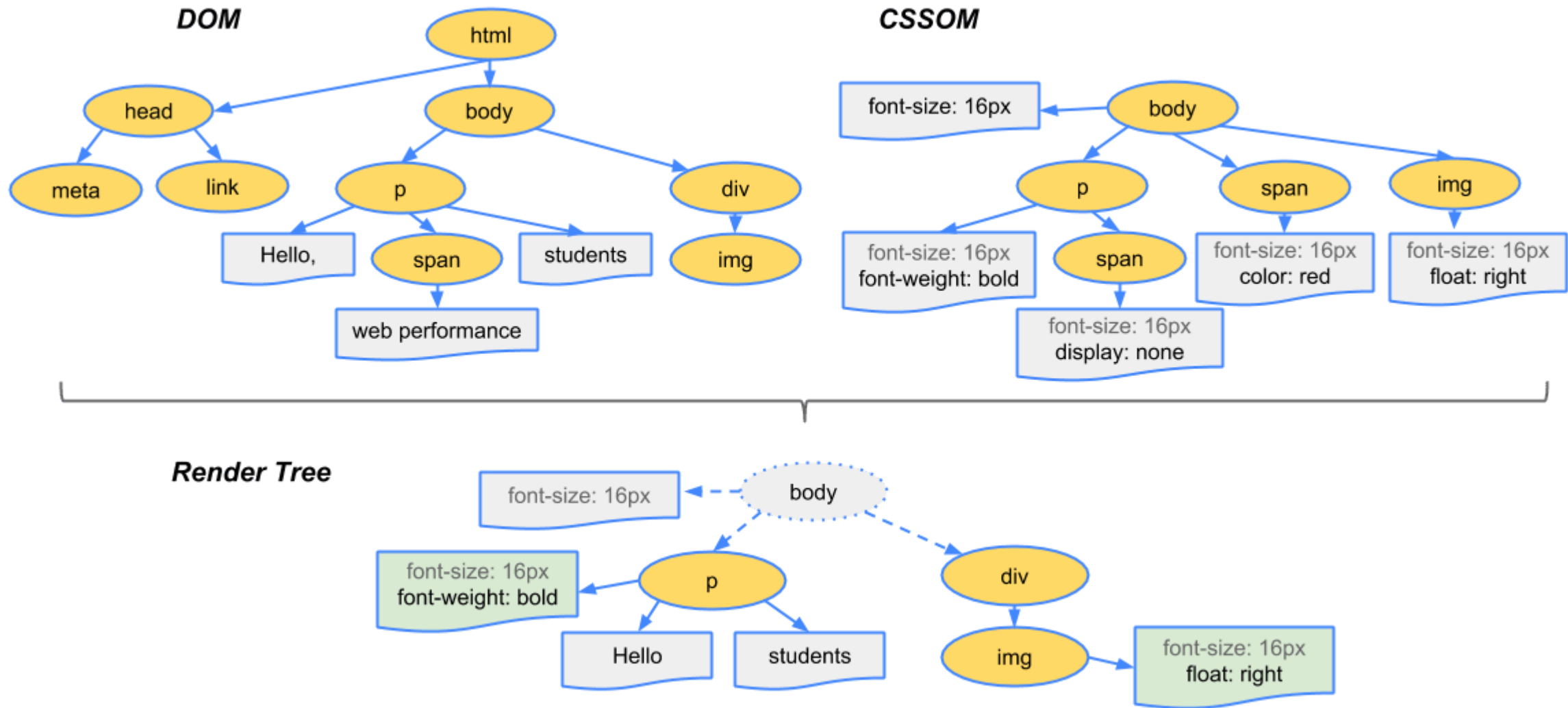
Render Tree 생성 과정

Render Tree = DOM + CSSOM

- DOM 에 CSS 스타일링을 추가하는 작업을 Attachment 라고 한다.



- DOM 트리에 노드가 추가될 때마다 new attach 메서드를 발생시킨다.
- HTML 파일의 html, body 태그를 처리하면 Render Tree 의 root 노드가 생성된다.
- Root 노드란 트리 하부의 모든 구성정보 (DOM & CSSOM) 를 포함할 트리의 시작지점.



Layout

- Render Tree 를 브라우저에 표시하기 위해서는 각 픽셀을 어디에 나타낼지 정해야 한다. 이를 위해서 레이아웃 작업이 필요하다.
- 레이아웃 시 주의할 점은 레이아웃 작업의 재 배치 비용이 비싸기 때문에, 가능한 한번에 업데이트 하고 자주 Recalculate Style 을 하지 않도록 한다. **Reflow 를 유발하는 Style 목록**

Filter <input type="text"/> All <input type="checkbox"/> Loading <input type="checkbox"/> Scripting <input checked="" type="checkbox"/> Rendering <input checked="" type="checkbox"/> Painting				
Start Time ▲	Self Time	Total Time	Activity	
17.8ms	0.0ms	0.0ms	■	Send Request
301.3ms	0.1ms	0.1ms	■	Receive Response
306.6ms	0.0ms	0.0ms	■	Receive Data
306.9ms	0.0ms	0.0ms	■	Finish Loading
307.4ms	79.3ms	80.5ms	▶ ■	Parse HTML
390.3ms	0.1ms	0.1ms	■	Update Layer Tree
390.4ms	0.0ms	0.0ms	■	Paint
390.7ms	0.2ms	0.2ms	■	Composite Layers
391.0ms	0.0ms	0.0ms	■	Update Layer Tree
432.6ms	0.0ms	0.0ms	■	Recalculate Style
432.6ms	0.0ms	0.0ms	■	Update Layer Tree
432.6ms	0.0ms	0.0ms	■	Composite Layers

[detector.js:1](#)

Mozilla Browser Engine Layout Process 시청

Paint

- 렌더트리를 배치하는 작업까지 마치면 이제 실제로 브라우저에 그리는 작업을 한다.
- Performance 패널의 `Paint` 에 해당하고, 렌더링 트리를 화면의 픽셀로 전환하는 작업
- 브라우저 입장에서는 같은 요소라 할지라도 더 적은 스타일 속성을 갖고 있으면 더 빠르게 그릴 수 있다.
- *따라서, 불필요한 속성을 제거하고 필수 속성들만 추가하는 것이 성능에 도움이 된다.*

실습 - 데모 페이지 로딩으로 CRP 동작과정 확인

렌더링 순서 : DOM 생성 -> CSSOM 생성 -> 렌더트리 생성 -> 렌더트리 배치 -> 렌더트리 그림

1. 일반적인 웹 페이지 로딩
2. 점진적 웹 페이지 로딩

퀴즈

1. 브라우저에 웹 페이지가 로딩되는 과정을 최대한 상세히 적어서 jangkeehyo@gmail.com 로 제출
2. HTML, CSS, JS 를 이용해 간단한 웹 페이지를 생성해보고 Performance 패널을 이용하여 동작 과정을 요약해보세요.

참고

- [How browser works](#)
- [Critical Rendering Path](#)
- [브라우저는 어떻게 동작하는가 - Naver D2](#)
- [How does browser render a website](#)

끝