WEEK 4 FRONT-END SESSION

Prototype부터 TODO List까지

최 강 프 론 트

01 프로토타입

02 Callback VS Promise

03 async & await

04 DOM

05 이벤트

06 TO-DO List 만들기 # 뼈대 구성

최 강 프 론 트

01 프로토타입

Javascript

- 명령형
- 함수형
- 프로토타입 기반
- 객체지향 프로그래밍

지원

프로토타입 기반 객체지향 프로그래밍 언어

01 프로토타입

class or 생성자 함수

```
class Person {
    constructor(student_number, phone_number, email) {
        // 인수로 인스턴스 초기화
        this.student_number = student_number;
        this.phone_number = phone_number;
        this.email = email;
    }
}
```

class : 프로토타입 기반 객체 생성 패턴의
문법적 설탕 역할
+ 새로운 객체 생성 메커니즘

Javascript : 프로토타입을 기반으로 상속 구현

두 방법의 차이점을 알아보자 !

01 프로토타입

class or 생성자 함수

```
class Person {
    constructor(student_number, phone_number, email) {
        // 인수로 인스턴스 초기화
        this.student_number = student_number;
        this.phone_number = phone_number;
        this.email = email;
    }
}
```

상속

prototype

```
function Ultra2() {}
Ultra2.prototype.ultraProp = true;

function Super2(){}
Super2.prototype = new Ultra2();

function Sub2(){}
Sub2.prototype = new Super2();
Sub2.prototype.ultraProp = 2; // Sub2에서 ultraProp 값 지정

let o2 = new Sub2();
console.log(o2.ultraProp); // 2
```

Javascript : 프로토타입을 기반으로 상속 구현

두 방법의 차이점을 알아보자 !

01 프로토타입

프로토타입 동작

```
function Person2(name, first, second) {
    this.name = name;
    this.first = first;
    this.second = second;
}

Person2.prototype.sum = function() { // 한번만 정의됨 & 실행됨 (원형 정의)
    return this.first + this.second;
}

let park = new Person2('park', 10, 20);
park.sum = function() {
    return 'prototype modified : ' + (this.first + this.second);
}
let son = new Person2('son', 10, 10);
console.log("park.sum() : ", park.sum());
console.log("son.sum() : ", son.sum());
```

1. 본인이 해당 요소를 가지고 있는지
2. 부모(생성자 함수)가 해당 요소를 가지고 있는지
3. ...

prototype을 통해 재사용성 up!

02 Callback VS Promise

비동기 처리 – callback

Javascript : 동기적 (synchronous)
호이스팅된 이후 순차적으로 코드 실행

```
console.log("sync 1");
console.log("sync 2");
setTimeout(function () {
    console.log("async 1");
}, 1000);
console.log("sync 3");
```

```
sync 1
sync 2
sync 3
async 1
```

비동기 처리 방식으로 동작
ex) setTimeout, setInterval, HTTP 요청, 이벤트 핸들러

```
// 동기적 콜백함수
function printMessage(param_func) {
    param_func();
}

// 비동기적 콜백함수
function printDelayMessage(param_func, delay) {
    setTimeout(param_func, delay);
}

printDelayMessage(() => console.log('asynchronous - arrow function'), 3000);
printMessage(() => console.log('synchronous - arrow function'));
```

```
synchronous - arrow function
asynchronous - arrow function
```

02 Callback VS Promise

비동기 처리 – callback

Javascript : 동기적 (synchronous)
호이스팅된 이후 순차적으로 코드 실행

```
console.log("sync 1");
console.log("sync 2");
setTimeout(function () {
    console.log("async 1");
}, 1000);
console.log("sync 3");
```

```
sync 1
sync 2
sync 3
async 1
```

```
// 동기적 콜백함수
function printMessage(param_func) {
    param_func();
}

// 비동기적 콜백함수
function printDelayMessage(param_func, delay) {
    setTimeout(param_func, delay);
}

printDelayMessage(() => console.log('asynchronous - arrow function'), 3000);
printMessage(() => console.log('synchronous - arrow function'));
```

```
synchronous - arrow function
asynchronous - arrow function
```

[콜백 헬]로 가봅시다 !

비동기 처리 방식으로 동작
ex) setTimeout, setInterval, HTTP 요청, 이벤트 핸들러

02 Callback VS Promise

비동기 처리 – promise

Promise : javascript 안에 내장되어있는 객체(object)
비동기적 연산 수행을 위해 사용

```
var Promise: PromiseConstructor
new <any>(executor: (resolve: (value: any) => void, reject: (reason?: any) => void) =>
void) => Promise<any>
```

Creates a new Promise.

@param `executor`
A callback used to initialize the promise. This callback is passed two arguments: a resolve callback used to resolve the promise with a value or the result of another promise, and a reject callback used to reject the promise with a provided reason or error.

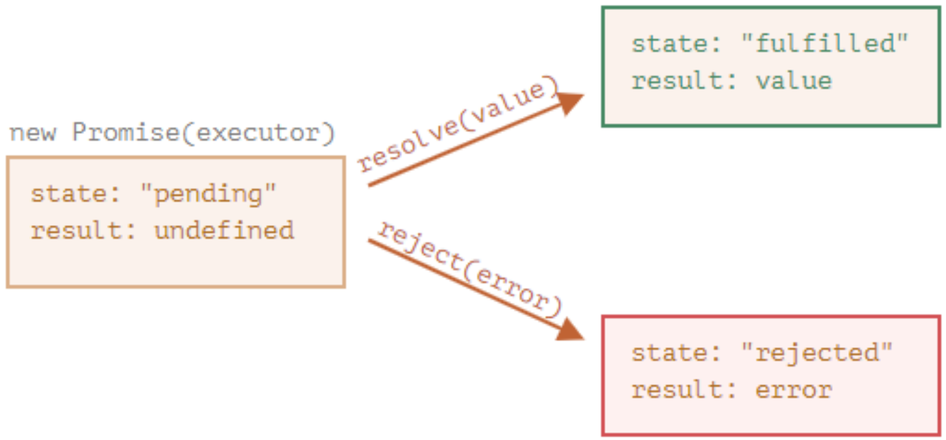
1. state : 비동기 처리가 진행되는 상태 정보

2. Producer(정보 제공자) vs Consumer(정보 소비자)

02 Callback VS Promise

비동기 처리 – promise

프로미스의 상태 정보	의미	상태 변경 조건
pending	비동기 처리가 아직 수행되지 않은 상태	프로미스가 생성된 직후 기본 상태
fulfilled	비동기 처리가 수행된 상태(성공)	resolve 함수 호출
rejected	비동기 처리가 수행된 상태(실패)	reject 함수 호출




```
▼ Promise {<pending>}
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

```
▼ Promise {<fulfilled>: 'front'}
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "front"
```

02 Callback VS Promise

Promise – Producer & Consumer

```
// 1. Producer
    // 새로운 promise가 만들어질 경우, executor 함수가 자동으로 실행됨
const promise = new Promise((resolve, reject) => { // promise : class
    // 시간이 소요되는 작업 ex) 네트워크 통신, 파일 입출력
    console.log("doing something...");
    setTimeout(() => {
        resolve('resolve function called :)');
        reject(new Error('reject function called :('));
    }, 2000);
});
```

```
// 2. Consumer
    // then, catch, finally
promise
    .then((value) => {
        // promise가 정상적으로 수행되어 resolve 콜백 함수를 통해 전달한 값
        console.log(value);
    })
    .catch((value) => {
        // promise가 정상적으로 수행되지 않아 reject 콜백 함수를 통해 전달한 값
        console.log(value);
    })
    .finally(
        // promise 수행에 상관 없이 항상 실행되는 동작
        console.log("finally is executed!")
    );
```

```
doing something...
finally is executed!
resolve function called :)
```

```
doing something...
finally is executed!
Error: reject function called :(
    at promise.js:14:16
```

02 Callback VS Promise

Promise Chaining

```
// 3. Promise chaining
const fetchNumber = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(1)
    }, 1000);
});

fetchNumber
    .then((num) => num * 2)
    .then((num) => num * 3)
    .then((num) => {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve(num - 1)
            }, 1000);
        });
    })
    .then((num) => console.log(num))
```

then을 통해 값 전달 O
새로운 Promise 생성 O

02 Callback VS Promise

Promise Error Handling

```
// 4. Promise Error Handling
const getBread = () =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve('🥨'), 1000);
    });

const getCoffee = (bread) =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${bread} + ☕`), 1000);
    });

const getHappy = (coffee) =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${coffee} = 😈`), 1000);
    });

getBread()
    .then(bread => getCoffee(bread))
    .then(coffee => getHappy(coffee))
    .then(happy => console.log(happy))
```

🥨 + ☕ = 😈

then을 통해 받은 값을
그대로 전달하는 경우
매개변수를 선언하지 않아도 OK !

```
getBread()
    .then(getCoffee)
    .then(getHappy)
    .then(console.log)
```

02 Callback VS Promise

Promise Error Handling

```
// 4. Promise Error Handling
const getBread = () =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve('🥨'), 1000);
    });

const getCoffee = (bread) =>
    new Promise((resolve, reject) => {
        setTimeout(() => reject(new Error(`error occurs : ${bread} + ☕`)), 1000);
    });

const getHappy = (coffee) =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${coffee} = 😈`), 1000);
    });

getBread()
    .then(getCoffee)
    .catch((error) => {
        return '🍆';
    })
    .then(getHappy)
    .then(console.log)
```

02 Callback VS Promise

Promise Error Handling

```
// 4. Promise Error Handling
const getBread = () =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve('🥨'), 1000);
    });

const getCoffee = (bread) =>
    new Promise((resolve, reject) => {
        setTimeout(() => reject(new Error(`error occurs : ${bread} + 🍮`)), 1000);
    });

const getHappy = (coffee) =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${coffee} = 😈`), 1000);
    });

getBread()
    .then(getCoffee)
    .catch((error) => {
        return '🥑';
    })
    .then(getHappy)
    .then(console.log)
```

🥑 = 😈

02 Callback VS Promise

Promise Error Handling

```
// 4. Promise Error Handling
const getBread = () =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve('🥯'), 1000);
    });

const getCoffee = (bread) =>
    new Promise((resolve, reject) => {
        setTimeout(
    });

const getHappy = (coffee) =>
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(`${coffee} = 😇`), 1000);
    });

getBread()
    .then(getCoffee)
    .catch((error) => {
        return '🍗';
    })
    .then(getHappy)
    .then(console.log)
```

[콜백 헬] 처리하러 가봅시다 !

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

[주의] 무작정 async와 await가 좋은 것 X

async : promise를 만들어주는 키워드

```
function fetchUser() {
    // 시간이 소요되는 작업 ex) 네트워크 통신, 파일 입출력
    return new Promise((resolve, reject) => {
        resolve('front');
    })
}

const user = fetchUser();
user.then(console.log);
console.log(user);
```

async 키워드 →

```
// 1. async
async function fetchUser() {
    return 'front';
}

const user = fetchUser();
user.then(console.log);
console.log(user);
```

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

[주의] 무작정 async와 await가 좋은 것 X

await : async 함수 내부에서 사용할 수 있는 키워드

```
function getHappy() {
    return delay(3000)
    .then(() => '😻🤣😊😆')
}
```

await 키워드

```
async function getHappy() {
    await delay(3000);
    return '😻🤣😊😆';
}
```

await : 프로미스가 settled(비동기 처리가 수행된 상태)가 될 때까지 대기하다가
settled 상태가 되면 프로미스가 resolve한 처리 결과를 반환
await : 반드시 프로미스 앞에서 사용

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

```
async function getHappy() {
    await delay(3000);
    return '😻🤣😊😆';
}

async function getMeat() {
    await delay(3000);
    return '🥩🍗🍖';
}

function happyCollector() {
    return getHappy()
    .then (happy => {
        return getMeat()
        .then(meat => `${happy} + ${meat}`);
    });
}

happyCollector().then(console.log);
```

async & await

```
async function happyCollector() {
    const happy = await getHappy();
    const meat = await getMeat();
    return `${happy} + ${meat}`;
}

happyCollector().then(console.log);
```

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

```
async function getHappy() {
    await delay(3000);
    return '😻🤣😊😆';
}

async function getMeat() {
    await delay(3000);
    return '🥩🍗🍖';
}

function happyCollector() {
    return getHappy()
    .then (happy => {
        return getMeat()
        .then(meat => `${happy} + ${meat}`);
    });
}

happyCollector().then(console.log);
```

async & await

병렬적으로 비동기 처리를 진행할 때

```
async function happyCollector() {
    const happyPromise = getHappy();
    const meatPromise = getMeat();
    const happy = await happyPromise;
    const meat = await meatPromise;
    return `${happy} + ${meat}`;
}

happyCollector().then(console.log);
```

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

```
async function getHappy() {
    await delay(3000);
    return '😻🤣😊😆';
}

async function getMeat() {
    await delay(3000);
    return '🥩🍗🍖';
}

function happyCollector() {
    return getHappy()
    .then (happy => {
        return getMeat()
        .then(meat => `${happy} + ${meat}`);
    });
}

happyCollector().then(console.log);
```

async & await

병렬적으로 비동기 처리를 진행할 때

```
function happyMeatCollector() {
    return Promise.all([getHappy(), getMeat()])
    .then(item => item.join(' + '));
}

happyMeatCollector().then(console.log);
```

03 async & await

Promise 도 복잡하다 ?

async & await를 통해 동기적으로 보이게 만들자 !

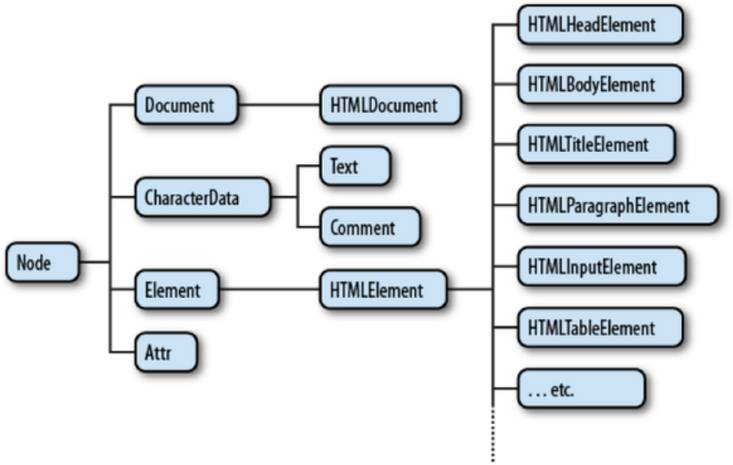
가장 먼저 값을 반환하는 프로미스만 전달

```
async function getHappy() {
    await delay(1000);
    return '😈🤣😊😆';
}

async function getMeat() {
    await delay(3000);
    return '🥩🍗🍖';
}
```

```
function onlyHappy() {
    return Promise.race([getHappy(), getMeat()]);
}

onlyHappy().then(console.log);
```

04 DOM

DOM – document object model



자바스크립트 : DOM을 통해 HTML 제어
문서에 프로그래밍적으로 접근하고 변형하기 위한 프로그래밍 인터페이스

Javascript !== Web browser

DOM 제어의 일반적인 순서

1. 제어할 대상 가져오기 – getElementById( ), getElementsByClassName( ), getElementsByTagName( ) …
2. 대상이 가지고 있는 메소드 실행 or 이벤트 핸들러 설치

웹 페이지를 자바스크립트로 제어하기 위한 객체 모델

04 DOM

DOM – 제어 대상 찾기

1. id를 이용한 요소 노드 취득 : getElementById( )

2. 태그 이름을 이용한 요소 노드 취득 : getElementsByTagName( )

3. class를 이용한 요소 노드 취득 : getElementsByClassName( )

4. css 선택자를 이용한 요소 노드 취득 : querySelector( ) , querySelectorAll( )

04 DOM

DOM – HTMLElement / HTMLCollection

```
<body>
    <ul>
        <li>HTML</li>
        <li>CSS</li>
        <li id="active">JavaScript</li>
    </ul>
    <script>
        var li = document.getElementById('active');
        console.log(li.constructor.name);
        var lis = document.getElementsByTagName('li');
        console.log(lis.constructor.name);
    </script>
</body>
```

- document.getElementById : HTMLLIELement 리턴
- document.getElementsByTagName : HTMLCollection 리턴

실행결과가 하나인 경우 HTMLLIELement 리턴

복수인 경우 HTMLCollection을 리턴

```
HTMLLIElement
HTMLCollection
```

04 DOM

DOM – HTMLElement / HTMLCollection

```
<body>
    <ul>
        <li>HTML</li>
        <li>CSS</li>
        <li id="active">JavaScript</li>
    </ul>
    <script>
        var li = document.getElementById('active');
        console.log(li.constructor.name);
        var lis = document.getElementsByTagName('li');
        console.log(lis.constructor.name);
    </script>
</body>
```

- document.getElementById : HTMLLIELement 리턴
- document.getElementsByTagName : HTMLCollection 리턴

실행결과가 하나인 경우 HTMLLIELement 리턴

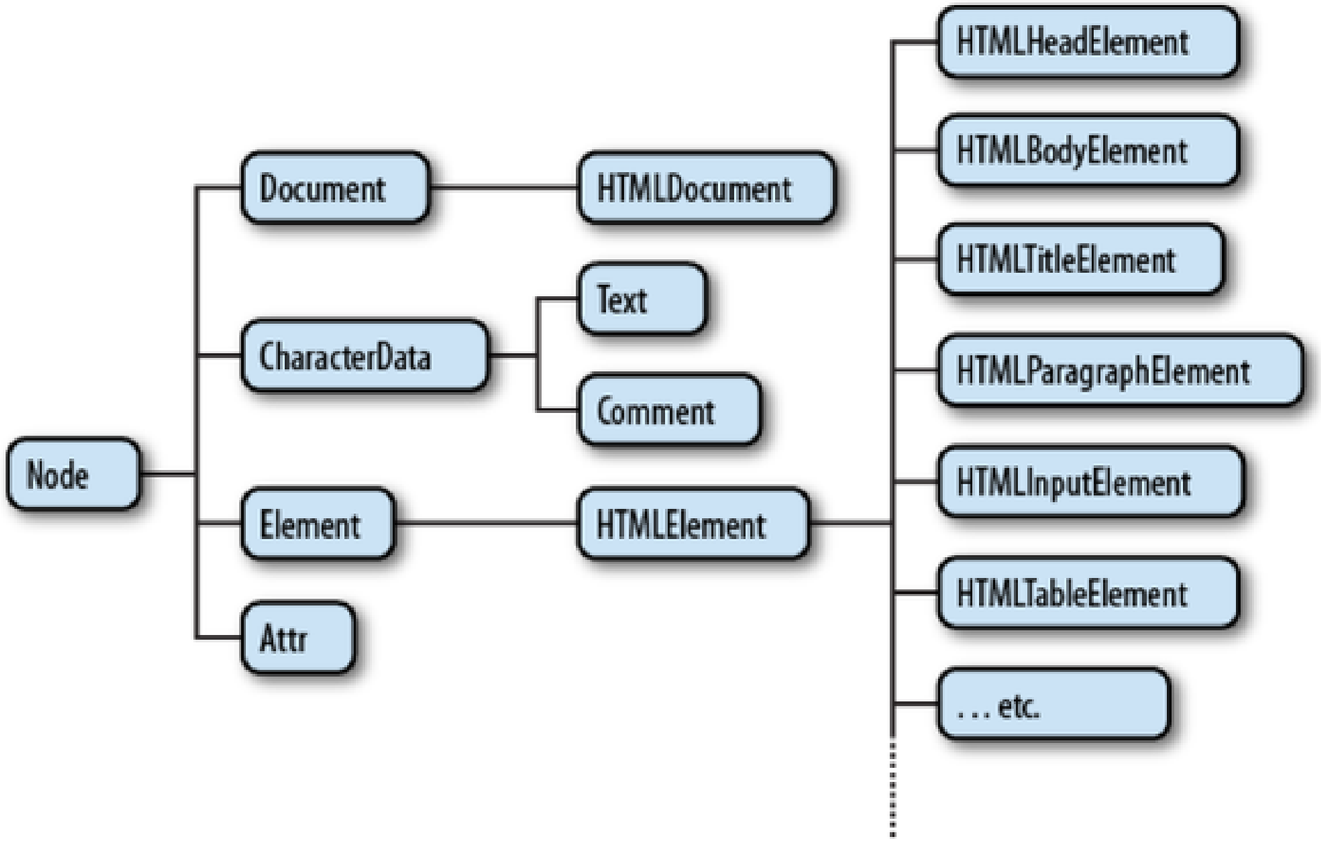
복수인 경우 HTMLCollection을 리턴

```
HTMLLIElement
HTMLCollection
```

앨리먼트 종류에 따라 리턴되는 객체가 다름

04 DOM

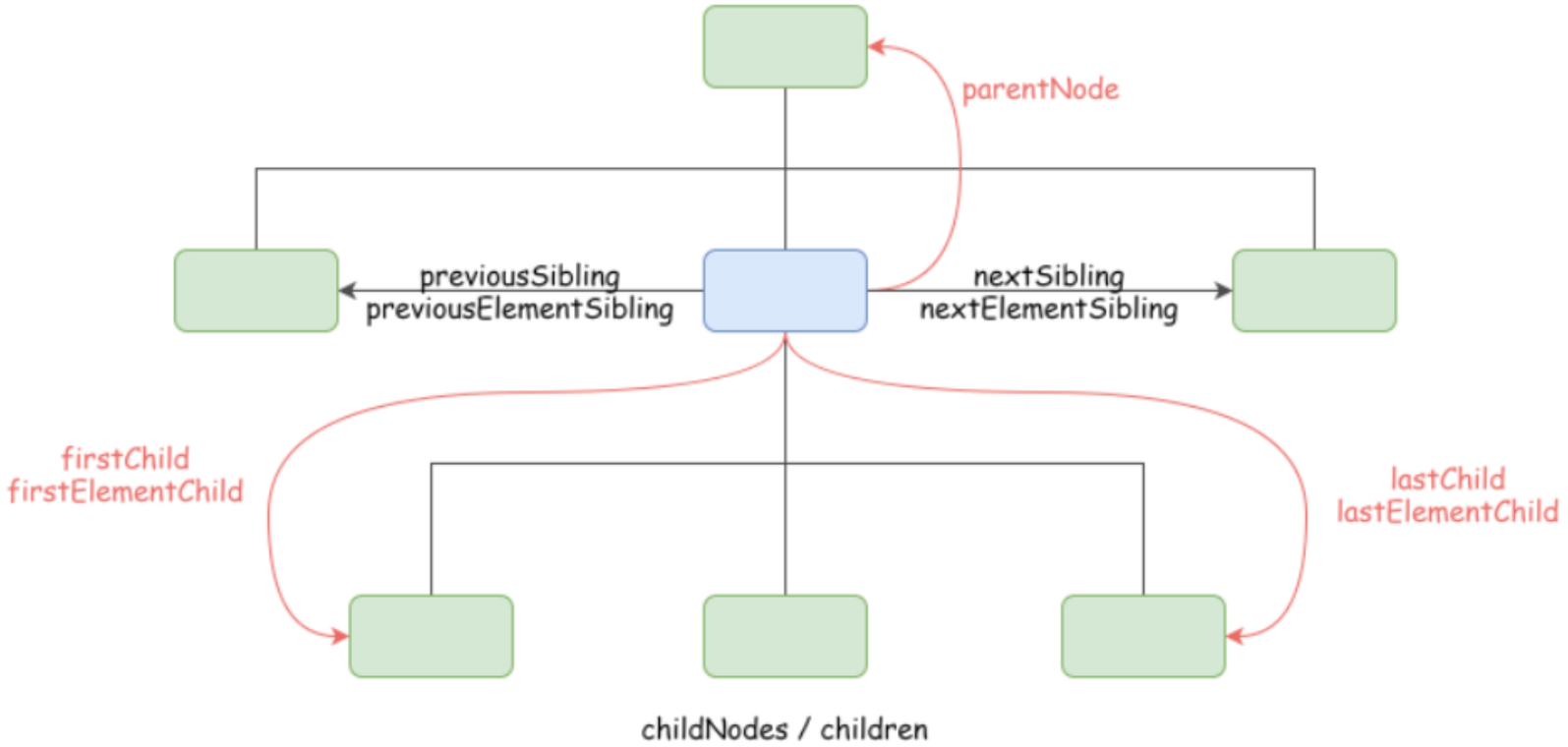
DOM – DOM Tree



모든 노드 객체 : Objece, EventTarget, Node 인터페이스를 상속받음

04 DOM

DOM – 노드 탐색



Node.prototype & Element.prototype에서 위의 프로퍼티를 제공

04 DOM

DOM – 노드 조작

노드 생성 :

Document.createElement, Document.createTextNode, Document.createDocumentFragment

노드 삽입 : Node.appendChild, Node.insertBefore

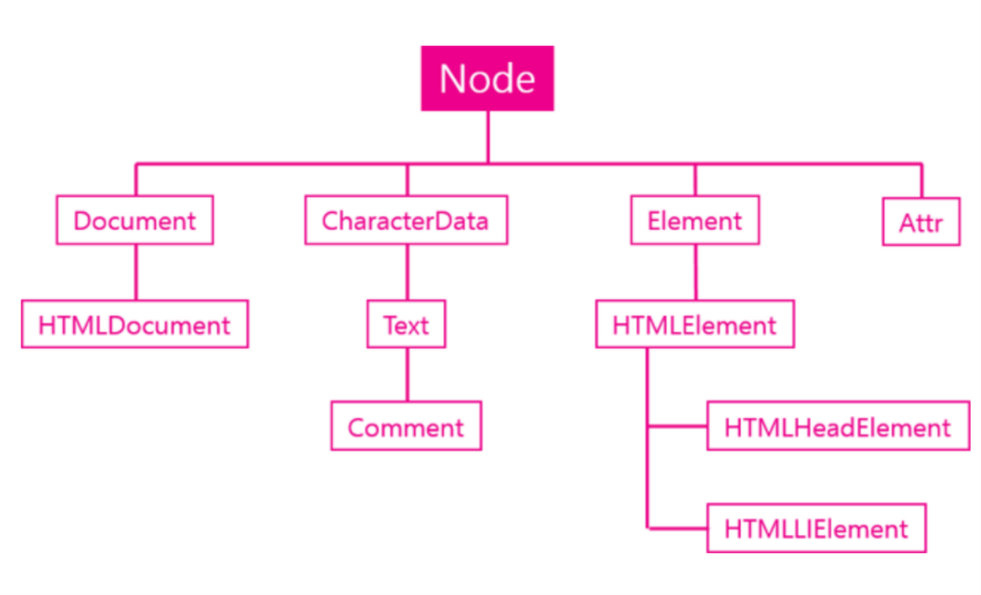
노드 복사 : Node.cloneNode

노드 교체 : Node.replaceChild

노드 삭제 : Node.removeChild

04 DOM

DOM – Node 객체의 기능



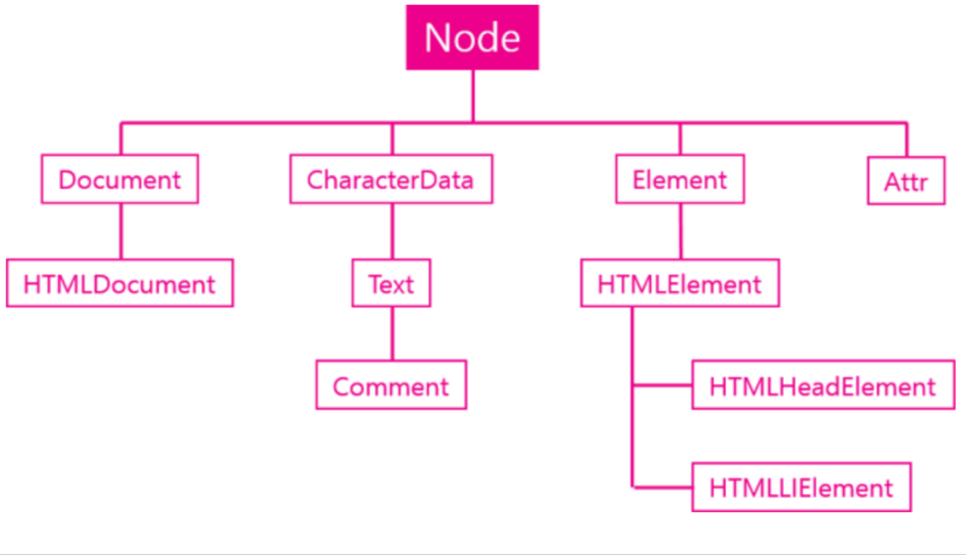
관계

각 엘리먼트 : 부모, 자식, 형제/자매 관계로 연결

- Node.childNodes
- Node.firstChild
- Node.lastChild
- Node.nextSibling
- Node.previousSibling
- Node.contains()
- Node.hasChildNodes()

04 DOM

DOM – Node 객체의 기능



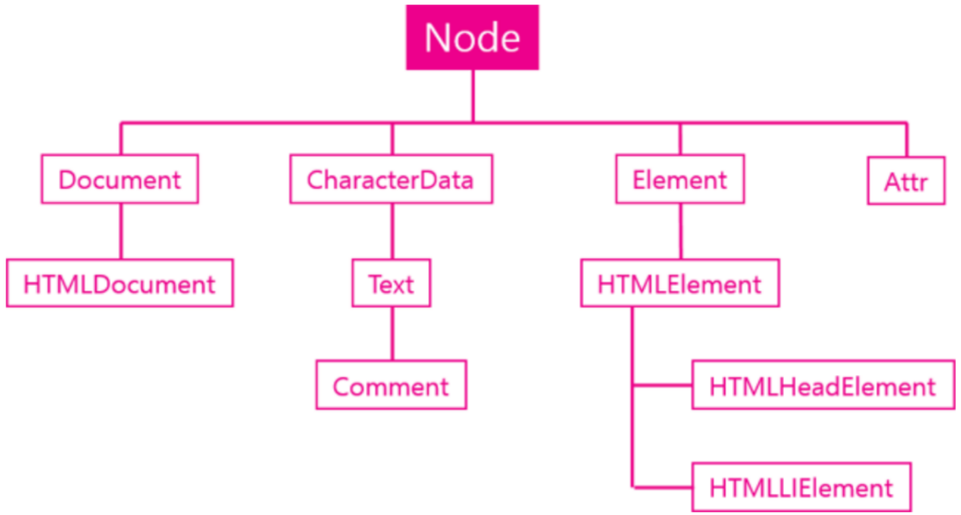
노드의 종류

Node 객체 : 모든 구성요소를 대표
각 구성요소가 어떤 카테고리에 속하는지 알려주는 식별자 제공

- Node.nodeType
- Node.nodeName

04 DOM

DOM – Node 객체의 기능



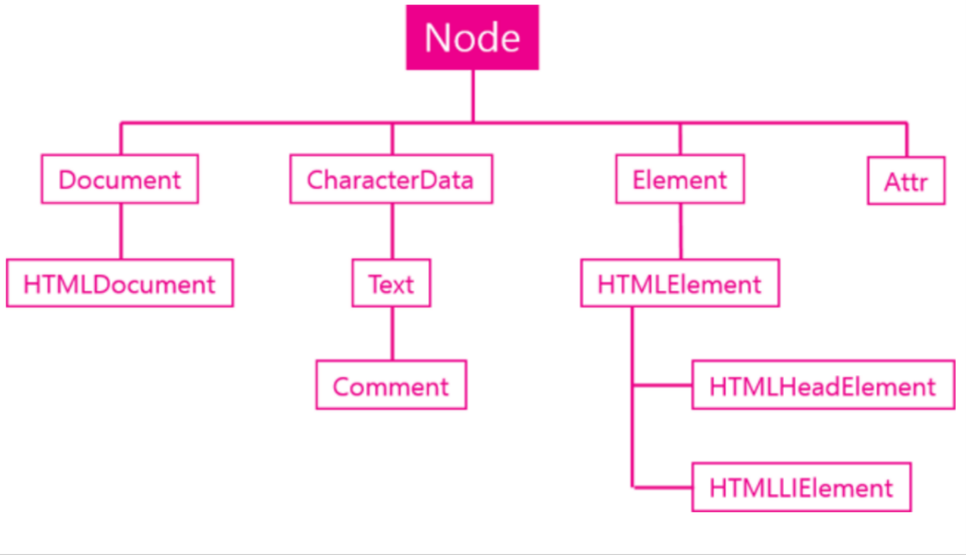
값

Node 객체의 값을 제공

- Node.nodeValue
- Node.textContent

04 DOM

DOM – Node 객체의 기능



자식 관리

Node 객체의 자식 추가 / 삭제 / 제거

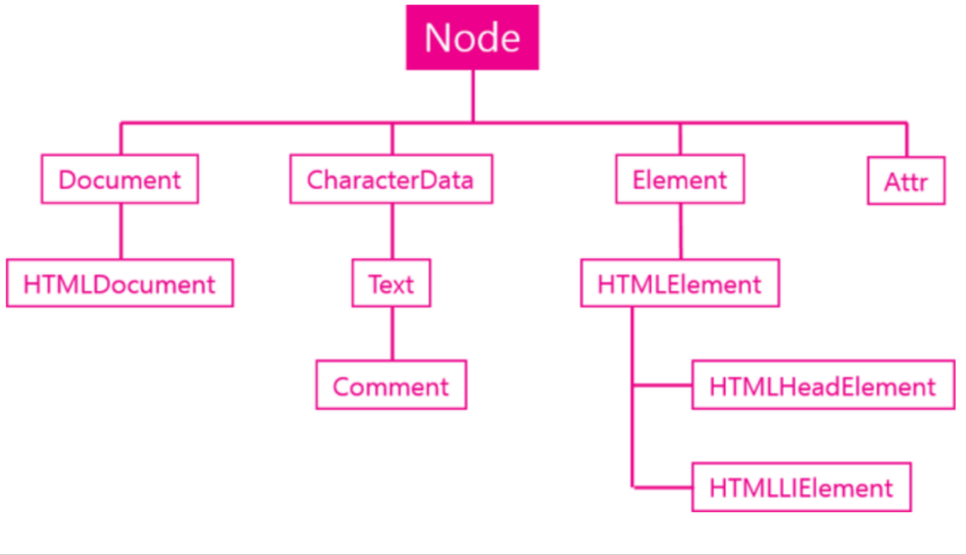
- Node.appendChild()
- insertBefore(newElement, referenceElement)

- Node.removeChild()

- replaceChild(newChild, oldChild)

04 DOM

DOM – Node 객체의 기능



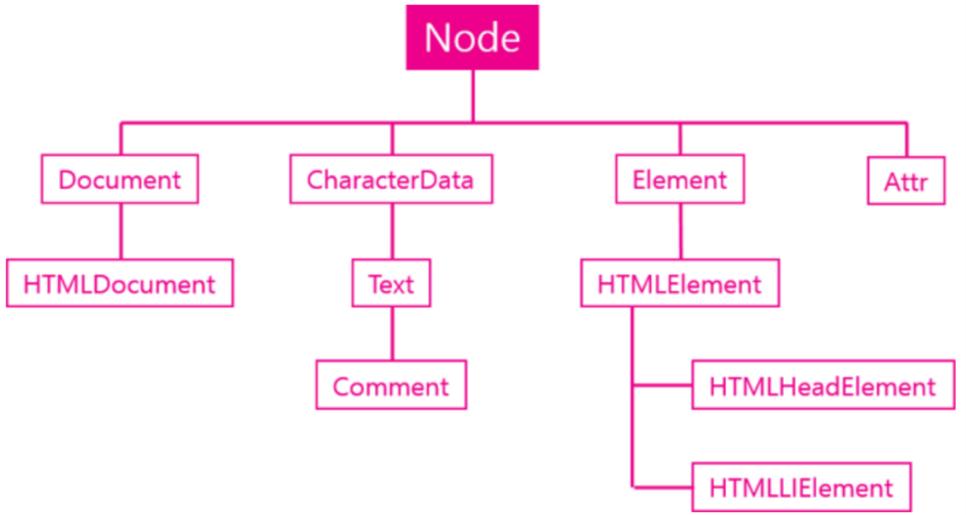
노드 추가

Node 추가 관련 API

- appendChild(child)
  노드의 마지막 자식으로 주어진 엘리먼트 추가
- insertBefore(newElement, referenceElement)
  appendChild와 동작방법 같음
  두번째 인자로 엘리먼트 전달 시 이것 앞에 엘리먼트 추가

04 DOM

DOM – Node 객체의 기능



노드 생성

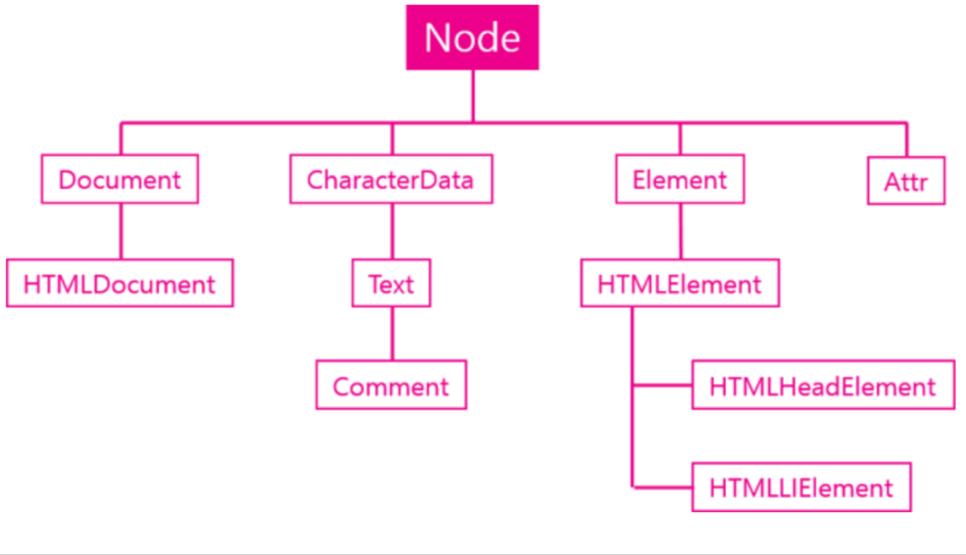
Node 추가 – 엘리먼트 생성 필요 (document 객체 영역)
노드 생성 API

- document.createElement(tagname) : 엘리먼트 노드 추가
- document.createTextNode(data) : 텍스트 노드 추가

```
const ul = document.createElement('ul'); // 요소 노드 생성
const li = document.createElement('li'); // 요소 노드 생성
const textNode = document.createTextNode('banana'); // 텍스트 노드 생성

li.appendChild(textNode); // 텍스트 노드를 li 요소 노드의 자식노드로 추가
ul.appendChild(li); // li 요소노드를 ul 요소 노드의 마지막 자식 노드로 추가
```

04 DOM

DOM – Node 객체의 기능



문자열로 Node 제어

보다 간편한 노드 조작법

- innerHTML : 문자열로 자식 노드 만들 수 있는 기능 / 읽어오는 기능
- outerHTML : 선택한 엘리먼트를 포함해 처리
- innerText, outerText : HTML 코드를 제외한 문자열 리턴
  (!== innerHTML & outerHTML)
  값 변경 시 HTML 코드 그대로 추가

05 이벤트

이벤트 : 브라우저가 처리해야 할 특정 사건이 발생하면 이를 감지해 발생시키는 것

이벤트 핸들러 : 이벤트 발생 시 호출될 함수

Event	Handler	이벤트 발생 시기
focus	onFocus	입력양식을 선택해 포커스가 주어졌을 때
blur	onBlur	포커스가 폼의 입력 양식을 벗어났을 때
select	onSelect	입력 양식에서 한 필드를 선택했을 때
change	onChange	입력 양식에서 값이 바뀌었을 때 발생
load	onLoad	해당 페이지가 로딩 되었을 때(처음 읽힐 때)
unload	onUnload	해당 페이지를 빠져 나갈 때
mousemove	onMouseMove	해당 영역에서 마우스 포인터가 움직일 때
mouseover	onMouseover	해당 영역에 마우스 포인터가 올라갈 때
mouseout	onMouseout	해당 영역에서 마우스 포인터가 빠져 나갈 때
mousedown	onMousedown	해당 영역을 마우스로 클릭할 때
mouseup	onMouseup	해당 영역을 마우스로 클릭했다 뗄 때
click	onClick	해당 영역을 마우스로 클릭할 때
keydown	onKeydown	해당 영역에서 키보드를 눌렀을 때
keyup	onKeyup	해당 영역에서 키보드를 눌렀다 뗄 때
keypress	onKeypress	해당 영역에서 키보드를 계속 누르고 있을 때
submit	onSubmit	입력 양식의 내용을 전송 할 때
reset	onReset	입력 양식의 내용을 초기화 할 때