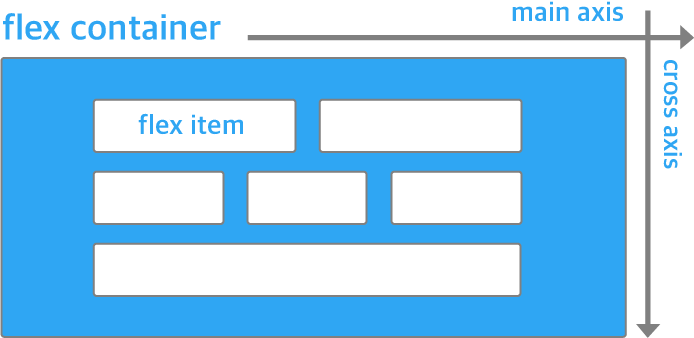
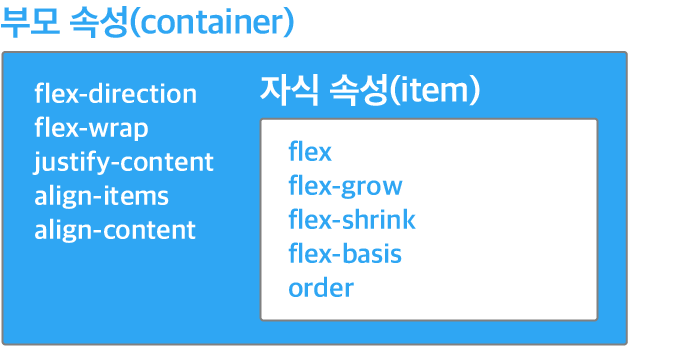
**1. FLEX레이아웃과 GRID 레이아웃을 설명하고 예제를 만드시오.**

**Flex Layout**

flexbox는 뷰포트나 요소의 크기가 불명확하거나 동적으로 변할 때에도 효율적으로 요소를 배치, 정렬, 분산할 수 있는 방법을 제공하는 CSS3의 새로운 레이아웃 방식이다. flexbox의 장점을 한 마디로 표현하면 '복잡한 계산 없이 요소의 크기와 순서를 유연하게 배치할 수 있다'라고 할 수 있다. 정렬, 방향, 순서, 크기 등을 유연하게 조절할 수 있기 때문에 별도의 분기 처리를 줄일 수 있고, CSS만으로 다양한 레이아웃을 구현할 수 있다. flexbox를 만드는 방법은 간단하다. 정렬하려는 요소의 부모 요소에 다음과 같이 display: flex 속성을 선언하면 된다. flexbox는 복수의 자식 요소인 flex item과 그 상위 부모 요소인 flex container로 구성된다.

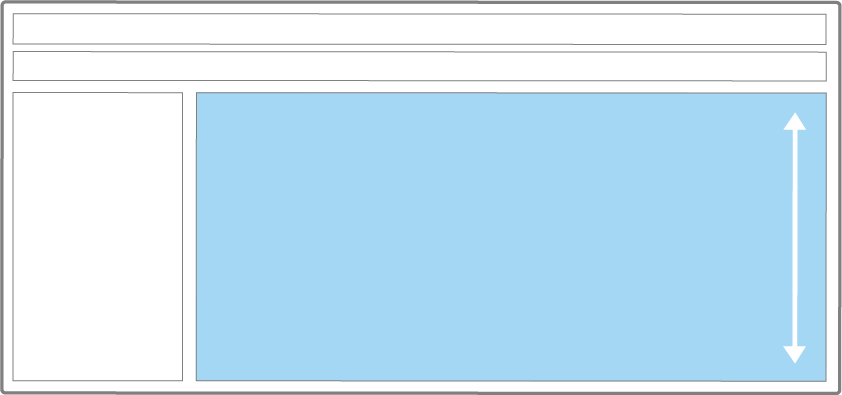


display: flex 속성이 적용된 요소는 flex container가 되고, flex container의 자식 요소는 flex item이 된다. flex container와 flex item은 부모 요소와 자식 요소를 한 세트로 사용하는 ul과 li를 생각하면 쉽게 이해할 수 있다.



**Flex Layout 예제**

1. **스크롤 없는 100% 레이아웃**



.flex\_container {

  display: flex;

  flex-direction: column;

  height: 100%;

}

.flex\_item {

  flex: 1; */\* flex: 1 1 0 \*/*

  overflow: auto;

}

1. **내비게이션 영역**



.flex-container {

  display: flex;

}

.flex-item {

  flex: none;

*/\* flex: 0 0 auto \*/*

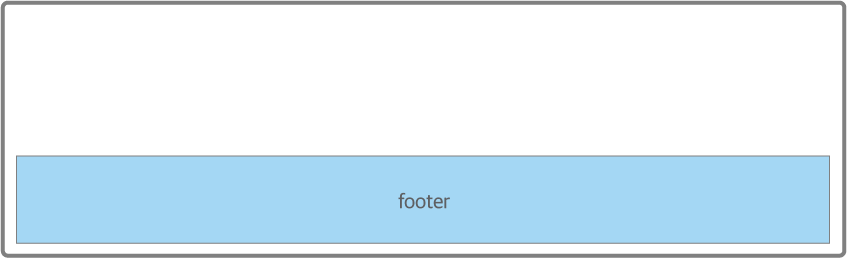
}

.flex-item-gnb {

  margin-left: auto;

}

1. **브라우저 화면 아래에 붙는 푸터**



.flex-container {

  display: flex;

  flex-direction: column;

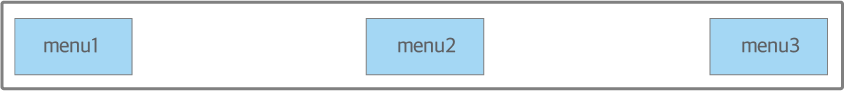
}

.flex\_item {

  margin-top: auto;

}

1. **정렬이 다른 메뉴**



.flex-container {

  display: flex;

  justify-content: space-between;

}

1. **폼 레이블 수직 중앙 정렬**



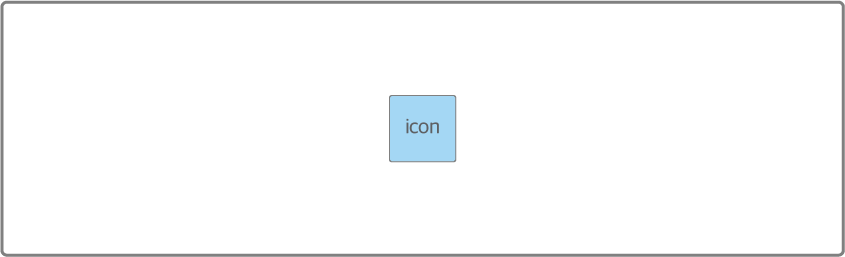
.flex\_container {

  display: flex;

  align-items: center;

}

1. **중앙 정렬 아이콘**



.flex\_container {

  display: flex;

  align-items: center;

  justify-content: center;

}

1. **유동 너비 박스**



.flex\_container {

  display: flex;

}

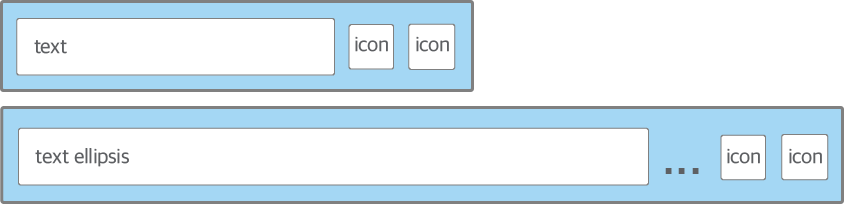
.flex\_item {

*/\* flex : initial \*/*

  max-width: 300px;

}

1. **말줄임과 아이콘**



.flex\_container {

  display: inline-flex;

  max-width: 100%;

}

.text {

*/\*flex: 0 1 auto\*/*

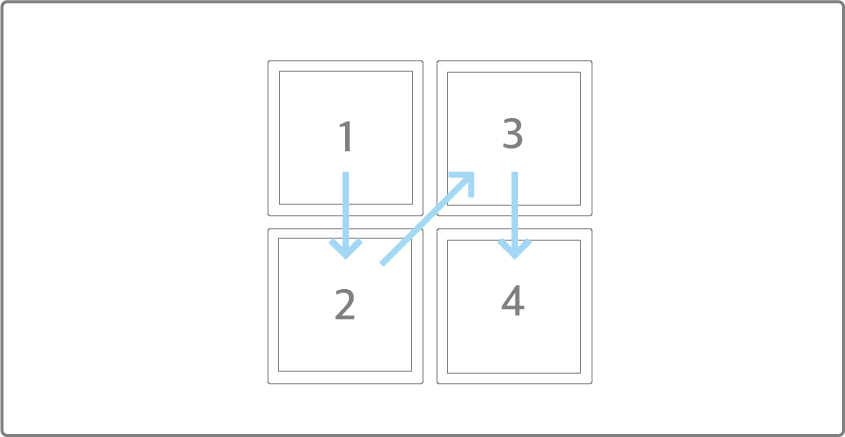
    overflow: hidden;

    text-overflow: ellipsis;

    white-space: nowrap;

}

1. **위아래로 흐르는 목록**



.flex\_container {

  display: flex;

  flex-direction: column;

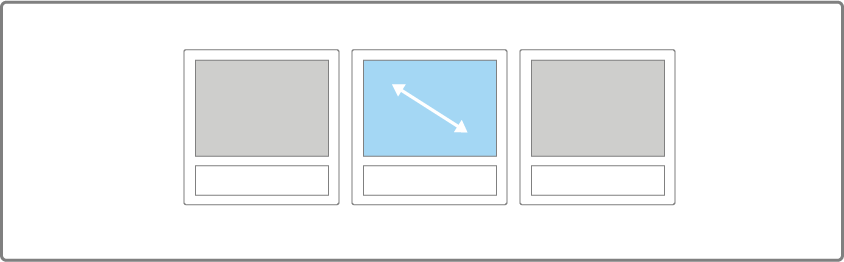
  flex-wrap: wrap;

  justify-content: space-around;

  align-content: space-around;

}

1. **가로세로 비율을 유지하는 반응형 박스**



.flex\_container {

  display: flex;

  flex-wrap: wrap;

}

.flex\_item\_list {

  flex-basis: 33.3%;

  display: flex;

  flex-direction: column;

}

.flex\_item\_image {

  flex: auto;

}

출처 :<https://d2.naver.com/helloworld/8540176>

**Grid Layout**

CSS 그리드 레이아웃(이하 ‘그리드’)은 그리드 기반 사용자 인터페이스 설계 방식을 완전히 바꾸기 위하여 탄생한 2차원 그리드 기반 레이아웃 시스템이다. 웹 페이지를 레이아웃할 때 항상 CSS를 사용하지만 생각만큼 쉽지 않다. 먼저 테이블을 만들고 float, position, inline-block을 사용한다. 하지만 이 모든 것은 근본적으로 세분화되어 있고 여러 중요한 기능(예를 들어 수직 중앙 정렬)이 누락되어 있다. 플렉스박스가 도움이 되지만 복잡한 2차원 레이아웃이 아닌 1차원 레이아웃이기 때문에 한계가 있다(플렉스박스와 그리드는 서로 호환이 잘 된다). 그리드는 지금껏 우리가 웹 페이지를 만들면서 힘들어했던 레이아웃의 문제점을 해결할 수 있는 최초의 CSS 모듈이다.

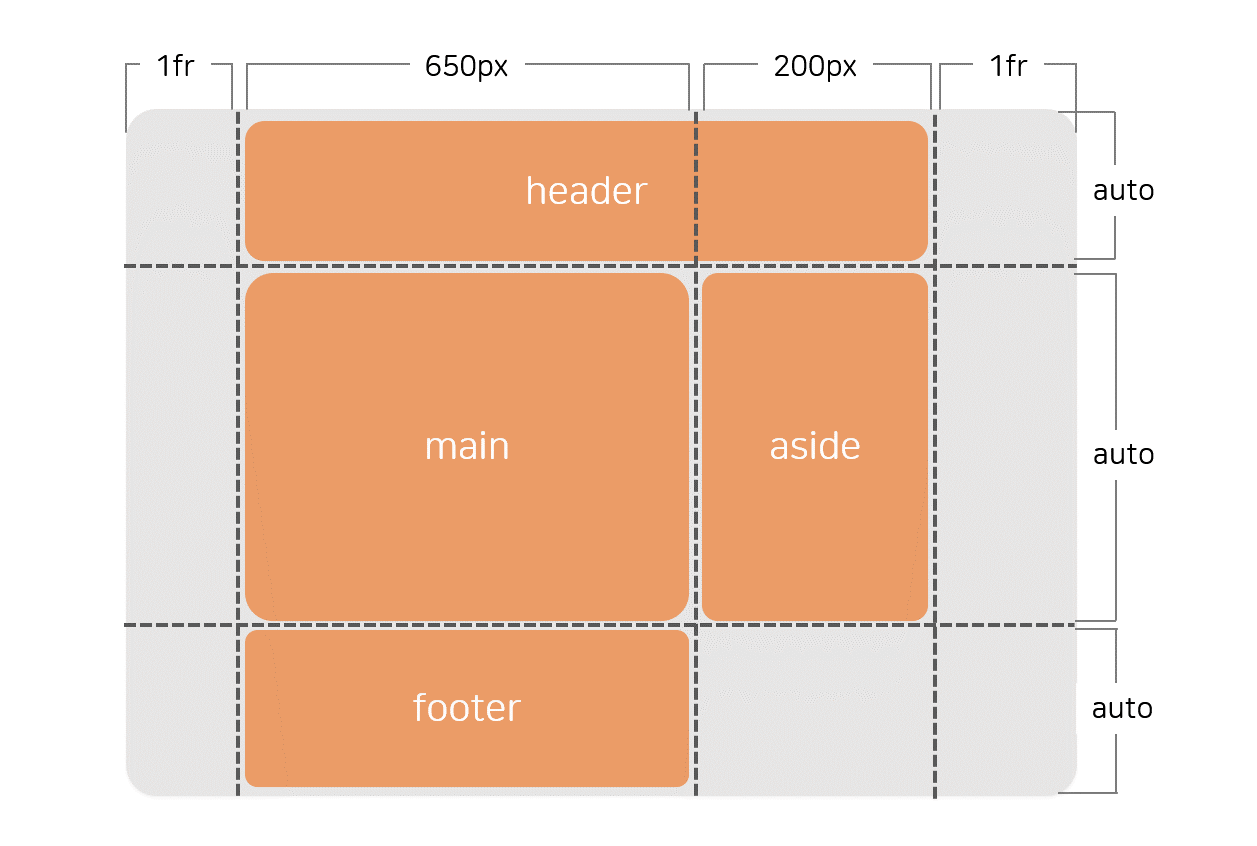
우선 그리드의 컨테이너 요소인 display: grid, 열과 행 크기를 정하는 grid-template-columns 와 grid-template-rows, 자식 요소인 grid-column과 grid-row 개념을 잘 알아야 한다.

출처 : <https://webactually.com/2019/06/%EA%B7%B8%EB%A6%AC%EB%93%9C-%EC%99%84%EB%B2%BD-%EA%B0%80%EC%9D%B4%EB%93%9C/>

그 외 기본적인 그리드 개념관련 : <https://ibrahimovic.tistory.com/23>

**Flex Layout 예제**

1. **웹페이지 레이아웃**



body {

  display: grid;

  grid-template-columns: 1fr 650px 200px 1fr;

  grid-template-rows: repeat(3, auto);

  grid-template-areas:

    ". header header ."

    ". main aside ."

    ". footer . .";

}

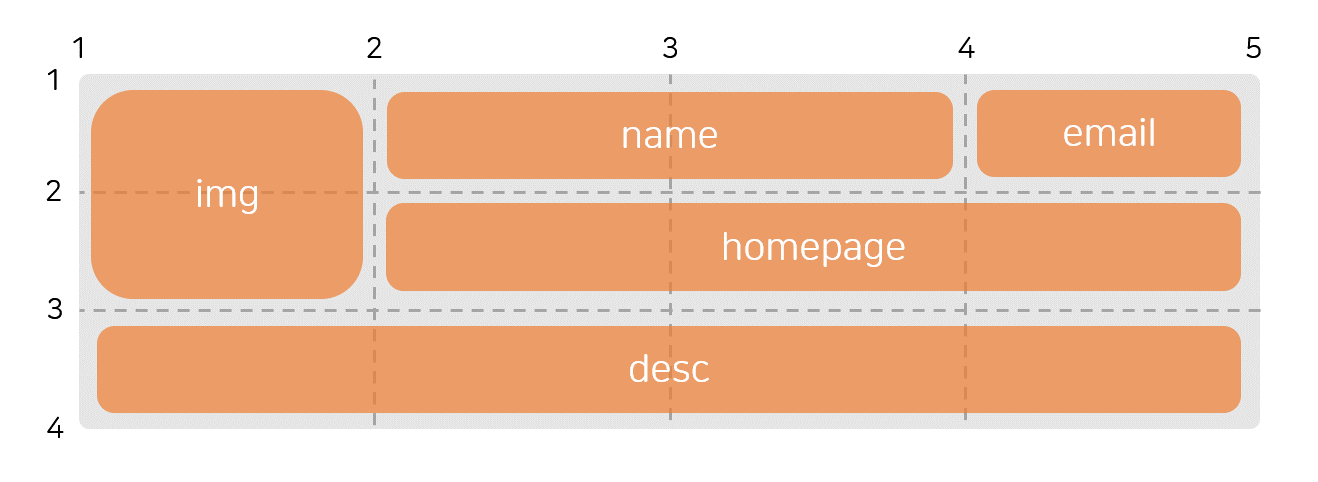
header {grid-area: header;}

main   {grid-area: main;}

aside  {grid-area: aside;}

footer {grid-area: footer;}

1. **사용자 카드**



.profile {

  display: grid;

  grid-template-columns: repeat(4, 1fr);

  grid-template-rows: repeat(3, auto);

}

.img     {grid-area: 1 / 1 / 2 / 3;}

.name    {grid-area: 1 / 2 / 2 / 4;}

.email   {grid-area: 1 / 4 / 2 / 5;}

.hompage {grid-area: 2 / 2 / 3 / 5;}

.desc    {grid-area: 3 / 1 / 4 / 5;}

**2. ES6문법을 10가지이상 설명하고 각 예제를 만드시오.**

**- 01. 템플릿문자열**

기존 자바스크립트에서는 문자열과 문자열 또는 문자열과 변수를 연결하려면 병합연산자(+)를 사용해야 했습니다. 병합 연산자를 사용했을때의 코드의 복잡성을 해결하기 위해 ES6에서는 템플릿 문자열을 도입했습니다. 템플릿 문자열을 작은따옴표 대신 백틱( ` ) 으로 문자열을 표현합니다. 또한 템플릿 문자열에 특수기호 $를 사용하여 변수 또는 식을 포함할 수도 있습니다.

ex)

**(병합연산자방식)** ‘장바구니에’ + cart.name + ‘가 있습니다. 총 금액은’ + getTotal(cart) + ‘입니다.’;  
**(템플릿문자열방식)** `장바구니에 ${cart.name}가 있습니다. 총 금액은 ${getTotal(cart)} 입니다.`

**- 02. 전개연산자**

전개연산자는 함수를 호출하는 인자로 배열을 사용하고 싶을 때나 배열을 정의하는 리터럴 내에서 사용할 수 있습니다. 사용 방법은 배열이나 객체, 변수명 앞에 마침표 세개 (…) 를 입력합니다. 다만, 배열, 객체, 함수 인자 표현식 ( [ ].{ },( ) ) 안에서만 사용해야 합니다.

S6이전 문법에서는 배열의 일부 요소만 잘라내거나 연결하려면 배열 인덱스와 함께 배열 내장함수들을 사용해야 했고, 객체또한 객체의 키나 값을 추출할때 객체 내장 함수를 사용했습니다. 이제는 전개연산자를 이용하여 코드를 간결하게 작성 할 수 있습니다.

ex)

**1) 전개연산자의 함수호출용의 용도**

const add = (a, b, c) => {

    return a+b+c;

}

var arr = [2, 4, 5];

add(...arr); // 11이 출력

**2) 배열 리터럴에서 전개연산자를 활용하는 방법**

var arr1 = [3, 4, 5];

var arr2 = [1, 2, ...arr1, 6, 7];

console.log(arr2);

// [1, 2, 3, 4, 5, 6, 7]

참고 : <https://pro-self-studier.tistory.com/13?category=659080>

**\*추가개념 : Rest Parameter ( 나머지 매개변수)**

함수를 선언 할 때, 해당 함수가 호출될 때 값으로 들어올 인자를 정하는 것을 매개변수라고 합니다. 이 매개변수를 지정할 때 매개변수의 이름 앞에 ...을 붙이면 이는 나머지 매개변수를 의미합니다. 그리고 이는 함수내에서 배열로 인식됩니다. 나머지 매개변수는 arguments와 다르게 유사 배열이 아닌 자바스크립트 표준 배열로 대체되고, 마지막 파라미터만 Rest 파라미터가 될 수 있습니다. 그리고 나머지 매개변수는 반드시 하나여야 합니다. 또한, arguments 객체는 호출될 때 들어온 모든 인자를 의미하지만 나머지 매개변수는 사용자가 원하는 인자만 배열로 정의할 수 있습니다.

function add2(a, b, ...rest) {

    console.log(arguments);

    console.log(rest);

}

add2(2,3,4,5,6);

첫 번째 콘솔은 유사배열로 호출될때 들어온 2, 3, 4, 5, 6 모두를 찍지만,

두 번째 콘솔은 나머지 매개변수로 정의한 4, 5, 6 만을 찍습니다.

나머지 매개변수의 효용은 인자로 들어온 값에 대해서 배열의 매서드를 이용해 어떤 작업을 하려고 할 때나 정해지지 않은 부정수인 인자들을 배열로 처리하고자 할 때 사용됩니다. arguments 객체와 분명히 다른, 좀 더 활용성이 높은 표현이다 라고 생각하시면 좋을 것 같습니다.

참고 : <https://pro-self-studier.tistory.com/15>

**- 03. 가변 변수와 불변 변수**

기존 자바스크립트 문법은 변수선언에 var 키워드를 사용했지만, ES6에서는 값을 수정할 수 있는 가변 변수를 위한 let 키워드와, 값을 수정할 수 없는 불변 변수를 위한 const키워드를 사용합니다.

**가변 변수 사용법** : **let**키워드로 선언합니다. let으로 선언한 변수는 읽거나 수정할 수 있습니다.

**불변 변수 사용법** : **const**키워드로 선언합니다. const로 선언한 변수는 읽기만 가능합니다.   
불변 변수는 값을 다시 할당할 수 없는 것이지, 값을 변경할 수는 있습니다. 하지만 불변 변수로 정의된 배열이나 객체를 수정하는 것은 ‘무결성 제약조건에 위반되었다’ 라고 합니다. 무결성을 유지하면서 불변 변수의 값을 수정해야 하는 경우가 있을때에는, 수정할 불변 변수를 새로 만들어 새 값을 할당하는 방법으로 수정해야 합니다. 불변 변수를 사용하면 무결성 제약 규칙에 의해 변수가 변하는 시점을 쉽게 파악할 수 있고, 수행 전과 후의 변수값을 비교할 수 있어 가변 변수보다 더 유용합니다.

**- 04. 클래스**

ECMAScript 5 이전 버전의 JavaScript에는 클래스가 없었습니다. 클래스에 가장 가까운 방법은 생성자를 생성한 다음 생성자의 프로토 타입에 메서드를 할당하는 것으로, 일반적으로 사용자 정의 타입 생성이라고 하는 접근 방식입니다.

클래스는 조금 더 연관있는 데이터(속성값(field) 과 행동 (method))를 하나로 묶어놓는 컨테이너 같은 개념. 클래스 안에서도 내부적으로 보여지는 변수와 외부로 보이는 변수를 나누어서 캡슐화를 해줍니다. 클래스를 통해 상속과 다양화가 일어날 수 있습니다.

Class는 template, 즉 틀과 같은 역할을 해서, 이러이러한 데이터가 들어갈 수 있다고 정의해서 선언해주고, 실제로 데이터를 넣어 만들어 낸 것은 object 라고 합니다.

**- 05. 화살표 함수**

화살표함수는 ES6에 추가된 표현식을 사용하는 함수로, 화살표기호 ( => ) 로 함수를 선언합니다.  
함수를 선언할때, function 키워드를 생략하고 인자블록 ( ) 과 본문블록 { } 사이에 =>를 표기합니다.

ex) 기존 : var add = function(first,second){ } 화살표함수 : var add = (first,second) => { }

화살표 함수를 통해, 계단형 함수 구조가 만들어지지 않게 할 수 있고, 콜백함수의 this 범위로 생기는 오류를 피하기 위해 bind() 함수를 사용하여 this 객체를 전달하는 과정을 포함하고 있어서, 해당 과정을 생략해서 사용할 수 있습니다.

**- 06. 객체 확장 표현식과 구조 분해 할당**

ES6에서는 이전 자바스크립트 문법에서 객체를 선언할 때 불편했던 점을 개선하고, 객체나 배열의 특정값을 손쉽게 추출할 수 있는 표현식을 추가했습니다.

ex) 객체 선언

var obj = { x : x , y : y }

* var obj = { x, y }

ex) 객체나 배열의 구조 분해

var key1 = obj.key1;

var key2 = obj.key2;

var newKey1 = obj.key1;

* var {

key 1 : newKey1,

key 2

}

**- 07. 라이브러리 의존성 관리**

의존성( Dependency )란?

어떤 코드나 파일에서 다른 코드나 파일을 필요로 하는 것입니다. 주로 객체지향언어에서는 두 클래스 간의 관계라고 말하기도 합니다.

기존 자바스크립트는 라이브러리나 모듈의 의존성을 script 엘리먼트를 이용하여 관리했습니다. 이때 정의되지 않은 함수를 참조하는 의존성 문제가 발생 할 수 있었는데, ES6는 이러한 문제를 import구문을 사용하여 script 엘리먼트 없이 연결된 파일 및 의존파일을 모두 먼저 내려받고 코드를 구동하도록 변경하여 해결하였습니다.

**- 08. 배열 함수**

**keys()와 entries()**

keys()는 배열의 각 인덱스에 대한 key들을 가지는 Array Iterator객체를 반환합니다. entries는 배열 형태로 전부다 반환합니다..

const 큰돌몬 = {

"이름":"큰돌",

"나이":"25"

};

console.log(Object.keys(큰돌몬));

console.log(Object.entries(큰돌몬));

//["이름", "나이"]

//["이름", "큰돌"]["나이", "25"]

**find와 findIndex**

find는 배열의 첫번째요소부터 원하는 "요소"를 찾아서 그 요소를 찾으면 바로.. break를 걸며 그 요소를 반환하며, 못찾을 경우, undefined를 반환합니다. findIndex 는 그 인덱스를 반환합니다. 못찾을 경우, -1을 반환합니다.

const numbers = [1, 3, 4, 5, 6];

const findEven = numbers.find(element => {

return element % 2 == 0;

})

console.log(findEven) //4

const findEvenIndex = numbers.findIndex(element => {

return element % 2 == 0;

})

console.log(findEvenIndex) //2

**forEach, map, filter, reduce**

forEach 는 배열을 이용하여 "깔끔하게" 돌릴 때. (가장 성능이 좋습니다)  
map 은 배열을 이용하여 "새로운" 배열을 만들 때.   
filter 은 배열을 이용하여 "조건"에 맞는 원소로 배열을 만들 때   
reduce는 배열을 통해 "하나의" 값을 추출해 낼때 사용됩니다.

//forEach는 배열을 다시 만들지는 못하지만 요소 안에서 로직을 짤 때 가장 빠릅니다..!

const func1 = (element, index)=>{

console.log(`${index}번째 요소는 ${element} 입니다.`);

}

//map은 새로운 배열..!

const func2 = (element, index)=>{

return element \* 2;

}

//reduce는 배열을 통해 하나의 값을..!

const func3 = (prev, curr, index)=>{

return prev + curr;

}

const sf = [1, 2, 3, 4].forEach(func1);

const sm = [1, 2, 3, 4].map(func2);

const sr = [1, 2, 3, 4].reduce(func3);

console.log(sf, sm, sr)

//sf는 리턴값이 없으므로 undefined를 반환합니다.

//sm은 배의 원소들을 반환하고 sr은 다 합친 10을 반환합니다.

//filter 함수를 돌려서 true값이 나오면 배열에 push를 합니다. 이 때 원소에 어떠한 행위를 할 수는 없습니다.

const s = [1, 2, 3, 4].filter(e=> e % 2);

console.log(s); // [1, 3]

const s = [1, 2, 3, 4].filter(e=> {

if(e % 2){

return true;

}else{

return false;

}

});

console.log(s); // [1, 3]

const s = [1, 2, 3, 4].filter(e=> {

if(e % 2){

return e \* 2;

}else{

return false;

}

});

console.log(s); // 이 때 [1, 3] 요소의 변환은 되지 않습니다.

**every와 some**

every는 배열의 모든 요소가 특정조건을 만족해야 true, some은 하나라도 만족한다면 true, 이것들이 아니라면 false를 반환합니다.

const love = ['나','는','너','를','사랑','해'];

const findLoveSome = love.some(element=>{

return element == '사랑';

})

console.log(findLoveSome) //true

const findLoveEvery = love.every(element=>{

return element == '사랑';

})

console.log(findLoveEvery) //false

**- 09. 비동기 함수**

**비동기 처리란?**

작업시간이 많이 필요한 작업A를 처리하느라 다른 작업들이 대기 하고 있는 상태라면 일단 다른 작업들을 먼저 진행하고 작업A와 작업A와 연관된 작업을 이후에 처리하는 방식을 말합니다.   
비동기는 준비가 완료되는 것 부터 실행하기때문에 동시에 실행될 수 있습니다. 비동기의 가치는 처리중인 지연된 정보를 기다리는동안 다른 코드를 또 실행할 수 있다는 점입니다. API응답을 기다리는동안 사용자는 다른 요청을 할 수 있습니다.

**프로미스(Promise)**

promise로 구현된 비동기 처리 함수는 콜백을 예측 가능한 패턴으로 사용하도록 도와주며, 콜백함수 안에서 생성된 프로미스 객체를 활용해 콜백함수가 성공,실패,오류 각각의 경우에 따라 후속 처리를 할 수 있습니다. 순차적이지 않는 비동기함수의 실행순서를 제어할 수 있게 도와줍니다. 콜백패턴에 비해 코드 가독성이 좋고 반환된 결과물을 사용하기 편합니다.

프로미스를 만드는 방법은 다음과 같습니다.

구현하려는 비동기 함수안에 Promise 객체를 만드는(new) 로직을 넣습니다.

프로미스 생성자 함수에 비동기 처리를 담당할 함수를 인자로 넣고,(=new Promise 생성자 함수 안에 비동기 처리함수를 인자로 넣고) 다시 그 처리 함수는 resolve와 reject 라는 인자를 전달 받습니다. 이 생성자 함수를 new를 통해 인스턴스화 된 객체가 바로 프로미스 입니다.

비동기 처리함수가 어떤 결과값을 갖는냐에 따라, 생성된 프로미스 객체는 아래와 같은 3가지 상태값을 갖습니다.

1. **pending(비동기 처리 로직이 아직 완료되지 않는 상태)**

new Promise로 Promise가 생성된 직후부터 resolve나 reject가 호출되기 전까지의 순간을 pending 이라고 합니다.

1. **fullfilled(처리가 완료되어 프로미스가 결과값을 반환한 상태)**

만약 비동기 처리함수 값이 성공이라면 resolve 메소드가 호출되고,resolve 메소드의 인자로 설정한, 비동기 함수의 결과값이 전달된다. 이 결과값은 then을 통해 후속 처리된다.

1. **Rejected(처리가 실패하거나 오류가 발생한 상태)**

[성공했을 경우]

비동기 처리함수가 성공이라면 ->resolve 메소드 실행 -> 이때 resolve 메소드 인자에 이미 설정한 처리함수의 결과값이 (resolve 메소드를 통해) 이동 -> then메서드를 통해 성공의 후속처리 실행



출처 : <https://moonsupport.tistory.com/m/159?category=797572>

**- 10. 모듈**

일반적으로 모듈은 파일 단위로 분리되어 있으며 애플리케이션은 필요에 따라 명시적으로 모듈을 로드하여 재사용한다. 즉, 모듈은 애플리케이션에 분리되어 개별적으로 존재하다가 애플리케이션의 로드에 의해 비로소 애플리케이션의 일원이 된다. 모듈은 기능별로 분리되어 작성되므로 코드의 단위를 명확히 분리하여 애플리케이션을 구성할 수 있으며 재사용성이 좋아서 개발 효율성과 유지보수성을 높일 수 있다. ES6부터는 모듈 시스템을 사용할 수 있게 만들었지만 브라우저에서 지원이 되지 않고있기때문에 webpack같은 모듈 번들러를 사용해야만 모듈시스템(import, export)을 사용할 수 있다.

**export, import**

export const pi = Math.PI;

 export function square(*x*) {

   return *x* \* *x*;

 }

 export class Person {

   constructor(*name*) {

     this.name = *name*;

   }

 }

export 키워드로 변수, 함수, 클래스를 외부의 스크립트로 모듈화 시킬 수 있다.

또한 export default pi; 이렇게 default키워드를 붙일 수도 있는데 default키워드를 사용하면 이 모듈을 사용하는 스크립트에서 따로 모듈 변수명을 입력하지 않으면 자동으로 default키워드가 있는 모듈을 가져온다.

*// main.js*

import { pi, square, Person } from './lib';

console.log(pi);         *// 3.141592653589793*

console.log(square(10)); *// 100*

console.log(new Person('Lee')); *// Person { name: 'Lee' }*

반대로 가져오는 방법은 위와 같이 "import" 키워드를 사용하고 from 으로 해당 모듈의 js파일을 가리키면 된다.

**출처 : Do it! 리액트 프로그래밍 정석 교재**

3. props 와 state는 언제 쓰는가.

**Props**

props 는 컴포넌트에서 사용 할 데이터 중 변동되지 않는 데이터를 다룰 때 사용됩니다. parent 컴포넌트에서 child 컴포넌트로 데이터를 전할 때, props 가 사용됩니다.

**State**

props처럼 부모 컴포넌트가 내려주는 것 말고, 컴포넌트 자신이 어떤 값을 관리하고싶다면 그때는 state를 사용합니다. 컴포넌트에서 유동적인 데이터를 다룰 때, state 를 사용합니다. React.js 어플리케이션을 만들 땐, state를 사용하는 컴포넌트의 갯수를 최소화 하는 것 을 노력해야합니다. 예를들어, 10 개의 컴포넌트에서 유동적인 데이터를 사용 하게 될 땐, 각 데이터에 state를 사용 할 게 아니라, props 를 사용하고 10 개의 컴포넌트를 포함시키는 container 컴포넌트를 사용하는것이 효율적입니다.

props 와 state, 생긴건 비슷하지만 용도는 다릅니다. 헷갈리지 않도록 다음 특성을 기억하세요.

| **특성** | **props** | **state** |
| --- | --- | --- |
| **parent 컴포넌트에 의해 값이 변경 될 수 있는가?** | **예** | **아니오** |
| **컴포넌트 내부에서 변경 될 수 있는가?** | **아니오** | **예** |

출처 : <https://velopert.com/921>