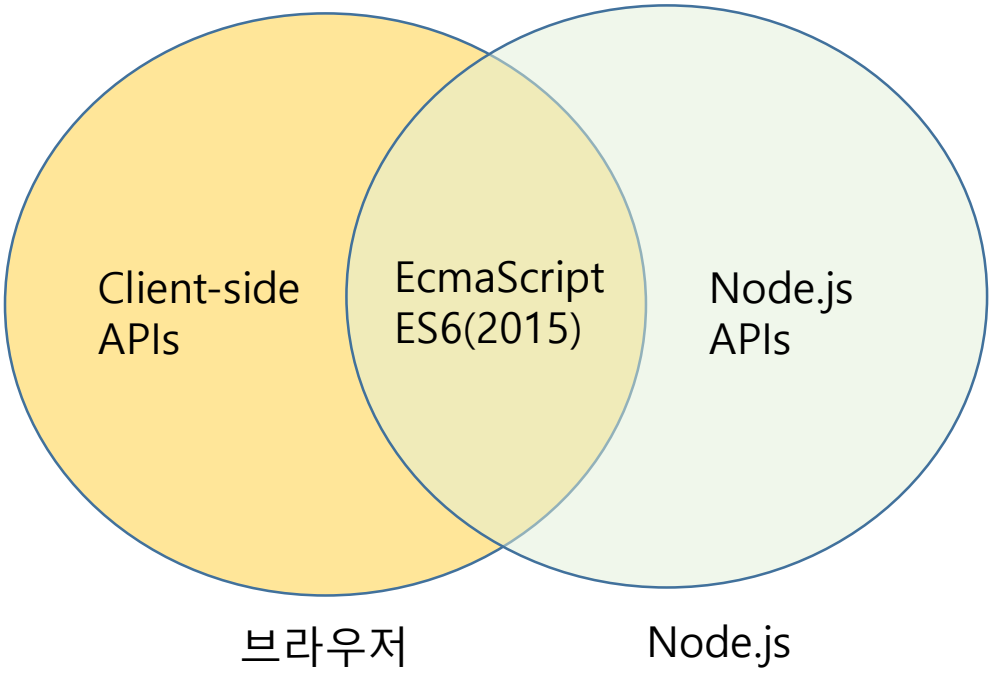
Javascript 실행환경

➢ 브라우저와 Node.js환경



➢ Client-side APIs

- ✓ DOM, BOM, Canvas, XMLHttpRequset, fetch, requestAnimation Frame, SVG, Web Storage, Web Component , Web Worker 등

데이터 타입

➢ **자바스크립트(ES6)는 7개의 데이터 타입을 제공한다.**

구문	데이터 타입	설 명
원시 타입	**숫자(number)**	정수,실수 구분없이 하나의 숫자 타입만 존재
	문자열(string)	문자열, “”, ‘’, ``(표현식)
	불리언(boolean)	논리적 참(true, 0이 아인값), 거짓(false, 0, undefined, null, ‘’)
	undefined	var 키워드로 선언된 변수에 암묵적으로 할당되는 값, undefined를 유일한 값을 가짐
	null	값이 없다는 것을 의도적으로 명할 때 사용하는 값 null을 유일한 값으로 가짐
	심벌	변경 불가능한 원시 타입, 객체의 중복되지 않는 프로퍼티 키를 만들기 위해 사용
객체 타입		객체, 함수, 배열 등.

Symbol

- **다른값과 중복되지 않는 유일무이한 값**
- **용도**
  - ✓ 이름의 충돌 위험이 없는 유일한 프로퍼티 키를 만들기위해사용
    - ex) 표준객체 확장
- **심벌값 생성**
  - ✓ Symbol함수 호출하여 생성하며 값은 외부로 누출되지 않아 확인 불가능.
  - ✓ 심벌값도 문자열,숫자,불리언 같이 객체처럼 접근하면 암묵적으로 래퍼객체를 생성함.
  - ✓ Symbo.for()
    - 심벌 전역 레지스트리에 키값을 키정하여 유일무이한 값을 애플리케이션 전역에 사용 하고자할 때 유용
    - 키값을 주어 심볼값 생성
      - const value = Symbol.for('key')
    - 생성된 심볼값으로 키를 추출하는 방법
      - const key = Symbol.keyFor(value)

심벌과 상수

➢ Java의 enum 흉내 내기

1) Symbol함수를 통해 유일무이한 값생성하여 프로퍼티값에 할다.

2) Object.freeze()의 매개값으로 객체를 전달하여 프로퍼티 변경을 방지.

```
자바언어의 e // JavaScript enum
// Direction 객체는 불변 객체이며 프로퍼티는 유일무이한 값이다.
const Direction = Object.freeze({
  UP: Symbol('up'),
  DOWN: Symbol('down'),
  LEFT: Symbol('left'),
  RIGHT: Symbol('right')
});

const myDirection = Direction.UP;

if (myDirection === Direction.UP) {
  console.log('You are going UP.');
} num타입 흉내내기
```

ES6함수의 구분

➢ 일반함수

- ✓ 표현방법에 따라
  - 함수선언문, 함수표현식
- ✓ 용도에따라
  - 생성자함수 : 인스턴스 생성이 목적, 객체생성 수단.
  - 비생성자함수 : 단순 연산이 목적

ES6함수구분	constructor (인스턴스생성유무)	prototype	super	arguments	this
일반함수 (함수선언문,함수표현식)	O	O	X	O	전역객체
메서드 (ES6의 축약함수)	X	X	O	O	O
화살표함수	X	X	X	X	X

전역객체

➢ 전역객체란?

- ✓ 코드가 실행되기 이전단계에 자바스크립트 엔진에 의해 생성되는 객체
- ✓ 일반 객체가 생성되기전에 전역객체 프로퍼티 중
  생성자함수와 이 생성자함수의 프로토타입객체는 이미 객체화되어 존재한다.

➢ 전역객체 식별자

- ✓ 브라우저 : window
- ✓ Nodes.js : global
- ✓ ES11)환경과 무관하게 globalThis로 통일됨

➢ 전역객체 구성요소

- ✓ 표준 빌트인객체
  - Math, Reflect, JSON 몇 개를 제외하고는 모두 생성자 함수다.
- ✓ 환경에따른 호스트객체(web환경, node.js환경에따라 나뉨)
- ✓ 전역변수(가장 외부에 var로 선언한 변수)
- ✓ 전역함수(가장 외부에 정의한 함수 선언문)

객체

➢ 객체

- ✓ 원시값을 제외한 나머지값(함수,배열,정규표현식)등
- ✓ 다양한 타입(원시값또는 다른객체)의 값을 하나의 단위로 구성한 복합적인 자료구조
- ✓ 상태를 나타내는 프로퍼티와 동작을 나타내는 메소드로 구성된 집합
  - 메소드 : 함수를 값으로 갖는 프로퍼티

객체의 프로퍼티

➢ 프로퍼티의 집합

✓ 프로퍼티는 키와 값으로 구성된다.

- 프로퍼티 : 객체의 상태를 나타내는 값(data)
- 프로퍼티 키 : 빈 문자열('')을 포함하는 모든 문자열 or 심벌 값
  - 프로퍼티 키는 일반적으로 문자열이므로 따옴표로 묶어야 한다.
    그러나 식별자 네이밍 규칙을 준수하는 이름은 따옴표 생략 가능.
- 프로퍼티 값 : 자바스크립트에서 사용할 수 있는 모든 값

➢ 프로퍼티 접근

✓ 마침표 표기법

✓ 대괄호 표기법

- 자바스크립트에서 사용 가능한 유효한 이름이 아니면 반드시 대괄호 표기법 사용. 따옴표 포함.

➢ 프로퍼티 축약표현

✓ 프로퍼티 값으로 변수를 사용하는 경우 프로커티 키와 변수이름이 동일한 경우 프로퍼티 키 생략가능

```
let x=1, y=1;
const obj = { x , y };
console.log(obj); // { x:1, y:1};
```

객체 생성방식

- **다양한 객체 생성방식의 차이가 있지만 추상연산에 의해 생성되는 공통점이 존재한다.**
- **추상연산은 빈 객체를 생성한 후 빈 객체에 프로퍼티와 프로토타입을 추가한 객체를 반환 한다. 프로토타입은 [[prototype]]내부 속성에 할당한다.**
  - ✓ [[prototype]]는 프로그래머가 직접 접근불가 하며 대신에 __proto__속성을 통해 간접 접근한다.
- **객체 생성 방법**
  - ✓ 객체 리터럴
    - 추상연산에 Object.prototype를 전달함.
  - ✓ Object 생성자 함수
    - 인수없이 생성자함수 호출하면 빈객체가 생기고, 추상연산에 Object.prototype를 전달함.
  - ✓ 생성자 함수
    - 추상연산에 생성자함수의 propertype객체를 전달함.
  - ✓ Object.create 메서드
    - Object.create(prototype, [,propertiesObject])
  - ✓ 클래스(Es6)

객체 생성방식

- **Object.create(prototype, [,propertiesObject])**
  - ✓ new 연산자 없이 생성할 수 있다,
  - ✓ 프로타입 객체를 직접 파라미터로 지정할 수 있다,
  - ✓ 객체 리터럴에 의해 생성된 객체도 직접 상속받아 생성 할 수 있다.

배열

- **여러 개의 값을 순차적으로 나열한 자료구조**
  - ✓ 같은 타입의 요소를 연속적으로 위치시키는 것이 최선
  - ✓ const arr = [‘apple’ , ‘banana’ , ’orange’];
- **요소**
  - ✓ 배열이 가지고 있는 값
  - ✓ 요소로 자바스크립트의 모든 값(기본타입,객체,함수,배열 등)이 될 수 있다.
- **인덱스**
  - ✓ 요소는 자신의 위치를 나타내는 0 이상의 정수인 인덱스를 갖는다.

배열

➢ 배열생성

- ✓ 배열리터럴
  - const arr = [1,2,3]
- ✓ Array 생성자함수
  - 전달된 인자가 2개 이상인 요소를 갖는 배열 생성
  - 전달된 인자가 1개 인경우 숫자가 아닌경우는 배열 생성됨.
  - const arr = new Array(1,2,3)
- ✓ Array.of 정적메소드
  - 전달된 인수가 1개이고 숫자이더라도 인수를 요소로 갖는 배열 생성
  - const arr = Array.of(1,2,3);
- ✓ Array.from 정적 메서드
  - 유사배열, 이터러블 객체를 인수로 전달받아 배열로 변환
  - Array.from('hello');

배열

➢ 배열요소 참조

- ✓ 인덱스로 참조
  - const arr = [1, 2];
  - console.log(arr[0]);

➢ 배열요소 추가

- ✓ 존재하지 않는 인덱스를 사용해 값을 할당하면 추가
  - arr[2] = 3; // [1, 2 , 3]

➢ 배열요소 삭제

- ✓ delete 연산자는 희소 배열이 되므로 비권장
- ✓ Array.prototype.splice 메소드 사용
  - arr.splice(2,1); // 2번 인덱스부터 1개 삭제

➢ 배열 매서드

- ✓ console.dir(Array.prototype)로 확인

배열의 고차함수

➢ Array.prototype.foreach

- ✓ 콜백함수 내에서 배열을 순회하면서 요소하나에 대한 수행처리를 하며 반환값은 undefined
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체

- ✓ 용도
  - for문 대체

배열의 고차함수

➢ Array.prototype.map

- ✓ 콜백함수의 반환값들로 구성된 새로운 배열을 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 1:1 값 변환

배열의 고차함수

➢ Array.prototype.filter

- ✓ 콜백함수의 반환값이 true인 요소로만 구성된 새로운 배열을 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체

- ✓ 용도
  - 자신을 호출한 배열에서 특정요소를 추출하거나 제거하기위해 사용

배열의 고차함수

➢ Array.prototype.some

- ✓ 콜백함수의 반환값이 단 한번이라도 참이면 true, 모두 거짓이면 false를 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 배열 요소중 콜백함수 정의 조건을 만족하는 요소가 1개 이상 존재하는지 확인

배열의 고차함수

➢ Array.prototype.every

- ✓ 콜백함수의 반환값이 모두 참이면 true, 단 하나라도 거짓이면 false를 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 배열 요소가 콜백함수 정의 조건을 모두 만족하는지 확인

배열의 고차함수

➢ Array.prototype.find

- ✓ 콜백함수의 반환값이 true인 첫번째 요소를 반환, 존재하지 않으면 undefine를 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 배열 요소중 콜백함수 정의 조건을 만족하는 첫번째 요소 검색
  - 배열 요소중 콜백함수 정의 조건을 만족하는 요소가 있는지 검색

배열의 고차함수

➢ Array.prototype.findIndex

- ✓ 콜백함수의 반환값이 true인 첫번째 요소의 인덱스를 반환, 존재하지 않으면 -1를 반환
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 배열 요소중 콜백함수 정의 조건을 만족하는 첫번째 인덱스 검색
  - 배열 요소중 콜백함수 정의 조건을 만족하는 요소가 있는지 검색

배열의 고차함수

➢ Array.prototype.flatMap

- ✓ map배열을 통해 성성된 새로운 배열 평탄화
  - 인수1 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - map 메서드와 flat메서드를 순차적으로 실행한 결과와 동일
  - 평탄화 1단계만 가능
  - 평탄화 깊이를 지정하려면 map,flat을 수차적으로 적용해야함.

배열의 고차함수

- **Array.prototype.flatMap**
  - ✓ map배열을 통해 성성된 새로운 배열 평탄화
    - 인수1 ☞ **배열 요소**
    - 인수3 ☞ 배열 인덱스
    - 인수4 ☞ 배열 자체
  - ✓ map 메서드와 flat메서드를 순차적으로 실행한 결과와 동일

배열의 고차함수

➢ Array.prototype.sort

✓ 원본 배열을 직접 변경하여 정렬된 배열을 반환

- 인수1 ☞ **배열**
  - 기본 정렬순서는 유니코드 코드 포인트의 순서를 따름.

or

- 인수1 ☞ **비교함수**
  - 음수면 오름차순, 0이면 정렬하지않음, 양수면 내림차순

✓ 용도

- 배열의 요소를 정렬

배열의 고차함수

➢ Array.prototype.reduce

- ✓ 콜백함수의 반환값을 다음 순회 시에 콜백함수의 첫번째 인수로 전달하면서 콜백 함수를 호출하며 **하나의 누적된 결과값(숫자,배열,객체)을 만들어 반환한다**.
  - 인수1 ☞ **초기값 또는 콜백 함수의 이전 반환값**
  - 인수2 ☞ **배열 요소**
  - 인수3 ☞ 배열 인덱스
  - 인수4 ☞ 배열 자체
- ✓ 용도
  - 총합, 평균, 최대값, 최소값, 요소의 중복횟수, 충첩 배열 평탄화, 중복 요소 제거

배열의 고차함수

➢ Array.prototype.reduce

reduce() 작동 방식

다음의 예제를 생각해 봅시다.

```
[0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
  return accumulator + currentValue;
});
```

콜백은 4번 호출됩니다. 각 호출의 인수와 반환값은 다음과 같습니다.

callback	accumulator	currentValue	currentIndex	array	반환 값
1번째 호출	0	1	1	[0, 1, 2, 3, 4]	1
2번째 호출	1	2	2	[0, 1, 2, 3, 4]	3
3번째 호출	3	3	3	[0, 1, 2, 3, 4]	6
4번째 호출	6	4	4	[0, 1, 2, 3, 4]	10

용어정리

➢ 유사배열 객체

- ✓ 숫자형식의 문자열을 프로퍼티 키로 가지므로 배열처럼 인덱스로 프로퍼티 값에 접근할 수 있다.
- ✓ length 프로퍼티를 갖는다
  - ex) const arrayLike = { 0:1, 1:2, 2:3, length:3 };
- ✓ 이터러블 객체 아니므로 for~of문에서 사용 가능

➢ 희소배열

- ✓ 배열의 요소가 연속적을 위치하지 않고 일부가 비어있는 배열
- ✓ length와 배열 요소의 개수가 일치하지 않는 배열
  - ex) const arr = [1 , , 3];
- ✓ 희소배열의 lengt는 항상 희소배열의 실제 요소수 보다 크다.
- ✓ 이터러블 객체가 아니므로 for~of문에서 사용 불가능

함수형 프로그래밍 이해를 높이기 위한 사전지식

- **코드의 메인 로직에서 제어 흐름을 명확하게 분리하여 간결하고 확장성 좋은 선언적 프로그램을 작성하기위한 전제조건**
  - ✓ 자료구조를 순차적으로 탐색/변환에 실용적인 연산
    - map, filter, reduce
      - 대부분의 루프는 위의 연산에 해당됨.
      - 수동적 루프를 없애기 위한 목적
      - 항상 새로운 컬렉션을 반환하므로 불변성 보장
    - map : 모든 개별 요소를 변환하여 **동일한 크기의 컬렉션** 반환
    - filter : **특정 조건에 만족하는 요소만**를 추출한 컬렉션 반환
    - reduce : 모든 개별 요소를 누적 평가하여 의미 있는 **하나의 값**으로 반환
  - ✓ 재귀적 사고방식
    - 주어진 문제를 자기 반복적인 문제(단계마다 하는일이 같음)들로 잘게 분해한 다음, 이들을 다시 조합해 원래 문제의 정답을 찾는 기법
    - 루프를 대체할 때 많이 쓰는 기법
    - XML,HTML,그래프 등을 파싱할 때 활용

명령형 프로그램 vs 선언적 프로그램

➢ 명령형 프로그램의 특징

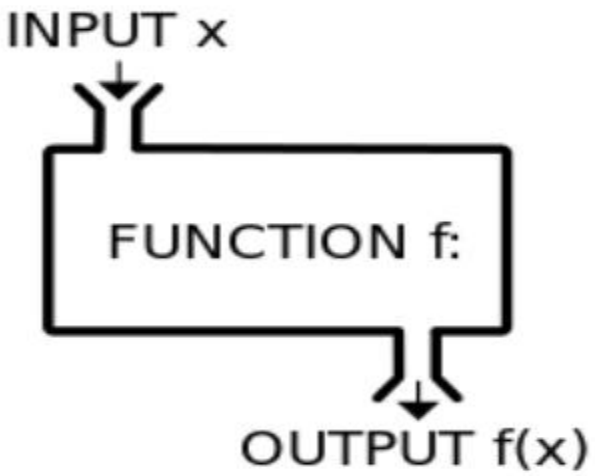
- ✓ 작업에 필요한 전 단계를 노출하여 흐름이나 경로를 아주 자세히 서술
  - 루프, 분기,구문 마다 값이 바뀌는 변수들로 빼곡히 참

➢ 선언적 프로그램의 특징

- ✓ 추상화 수준이 높다.
  - 블랙박스 연산들이 최소한의 제어구조를 통해 연결.(연결 수단은 고차함수를 사용)
- ✓ 데이터와 제어 흐름 자체를 고수준의 컴포넌트(함수) 사이의 단순한 연결로 취급.
- ✓ 제어의 흐름과 로직을 분리할 수 있다.
- ✓ 코드와 데이터를 더욱 효과적으로 헤아릴 수 있다.
- ✓ 자료구조보다 연산에 더 집중한다.
  - 기반 자료구조에 영향을 끼치지 않는 방향으로 연산을 바라 볼 수 있다.

함수란

- 입력(input)을 받아 하나의 출력(output)을 보내는 코드 집합

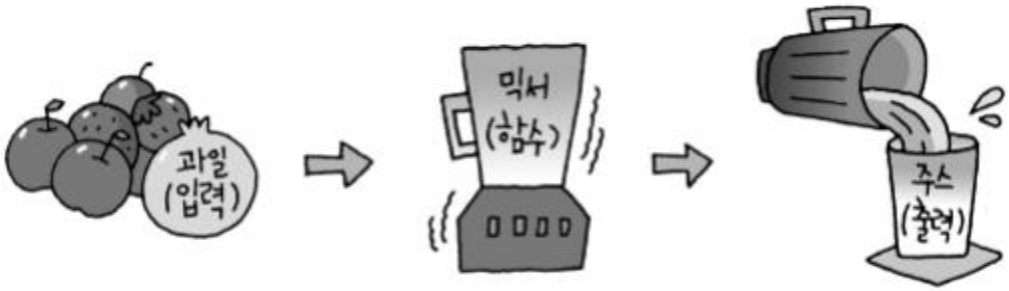



```
> function sum(num1, num2) {
      return num1 + num2;
  }
<· undefined
> const result = sum(4, 7);
<· undefined
> console.log(result);
  11
```

함수의 정의

함수의 호출

자바스크립트 코드



(믹서는 과일을 입력받아 주스를 출력하는 함수와 같다.)

함수형 프로그래밍

➢ 일급객체

- ✓ 값으로 다룰 수 있다.
- ✓ 변수에 저장 할 수 있다.
- ✓ 함수의 매개값로 사용될 수 있다.
- ✓ 함수의 반환값으로 사용될 수 있다.

➢ 함수를 값으로 취급하는 프로그래밍 기법

- ✓ 함수를 변수에 저장할 수 있다
- ✓ 함수의 매개값으로 함수가 올 수 있다.
- ✓ 함수의 반환값으로 함수가 올 수 있다.

➢ 고차함수

- ✓ 함수를 값으로 다루는 함수
  - ▪ 함수를 매개값으로 받아서 실행하는 함수
  - ▪ 함수를 반환값으로 리턴하는 함수(클로저를 만들어 리턴하는 함수)
  - ▪ 외부 상태의 변경이나 가변 데이터를 피하고 불변성을 지향
    - 동일한 값을 입력하면 항상 동일한 값 반환

함수형 프로그래밍

- 정의
  - ✓ 순수함수와 보조함수의 조합을 통해 외부 상태를 변경하는 부수효과를 최소화해서 **불변성을 지향**해 안정성을 높이려는 프로그래밍 패러다임.
- 목표
  - ✓ 순수함수와 보조함수의 조합을 통해 로직 내에 존재하는 **조건문과 반복문을 제거**하여 복잡성을 해결하고 **변수의 사용을 억제**하여 상태변경을 피해 오류를 최소화하고자 하는 프로그래밍 패러다임.
    - ▪ 고차함수 내부에는 조건문,반복문이 사용되어 외부로부터 은닉하여 로직흐름을 이해하기 쉽게하고 복잡성을 해결한다.
  - ✓ 조건문이나 반복문은 로직의 흐름을 이해하기 어렵게 해서 가독성을 해치고 변수의 값은 누군가에 의해 언제든지 변경될 수 있어 오류 발생의 근본적 원인이 될 수 있기때문.
- 활용
  - ✓ 컬렉션 요소를 소트,순회,매핑,필터,리듀싱,분류,집계하는 용도
- 멀티 패러다임 언어란?
  - ✓ 객체지향 프로그래밍, 함수형 프로그래밍 모두를 지원하는 언어
  - ✓ 모던 언어라면 대부분 지원함.

용어정리

➢ 일반함수

- ✓ 함수선언문으로 정의한 함수
- ✓ 함수명을 소문자로 시작
- ✓ this는 전역객체를 참조.

➢ 생성자 함수

- ✓ 객체를 생성하기 위해 정의한 함수선언문
- ✓ 함수명을 대문자로 시작(관례)
- ✓ this는 생성할 인스턴스를 참조

➢ 콜백함수

- ✓ 함수의 매개변수를 통해 다른 함수의 내부로 전달되는 함수, 외부함수를 돕는 보조함수
- ✓ 콜백함수는 고차함수 내에서 호출된다.이때 고차함수는 필요에 따라 인수를 전달할 수 있다.
  - ▪ 예외) 콜백함수가 비동기함수인경우는 태스크에서 이벤트루프를 통해 호출된다.

➢ 고차함수

- ✓ 함수를 인수로 전달받거나(콜백함수) 함수를 반환하는 함수(클로저)
- ✓ 고차함수는 매개변수로 대입되는 콜백함수의 인터페이스를 정의 (추상화) 해 놓고 있다.

용어정리

➢ 중첩함수

- ✓ 함수 내부에 정의된 함수

➢ 클로저

- ✓ 중첩함수로 아래의 상황에 해당될 경우 클로저라 불린다.
  - **중첩함수가 상위스코프의 식별자를 참조하고 있고**
    중첩함수가 외부함수보다 더 오래 유지되는 경우
- ✓ 용도 : 상태를 안전하게 변경하고 유지하기위해 사용
- ✓ 렉스컬 스코프(=정적스코프)
  - 함수를 정의한 곳에서 상위스코프를 결정하는 방식으로 식별자 탐색시 메커니즘이 적용됨

용어정리

- **순수함수**
  - ✓ 함수 외부의 상태에 의존하지 않으며 함수 외부상태를 변경하지 않는, 즉 부수효과가 없는 함수
  - ✓ 오직 매개변수를 통해 함수 내부로 전달된 인수에게만 의존해 반환값을 만든다.
  - ✓ 동일한 입력값을 주입하면 항상 동일한 결과가 나오는 함수
- **비순수함수**
  - ✓ 외부상태에 의존하거나 외부 상태를 변경하는 함수

프로토타입

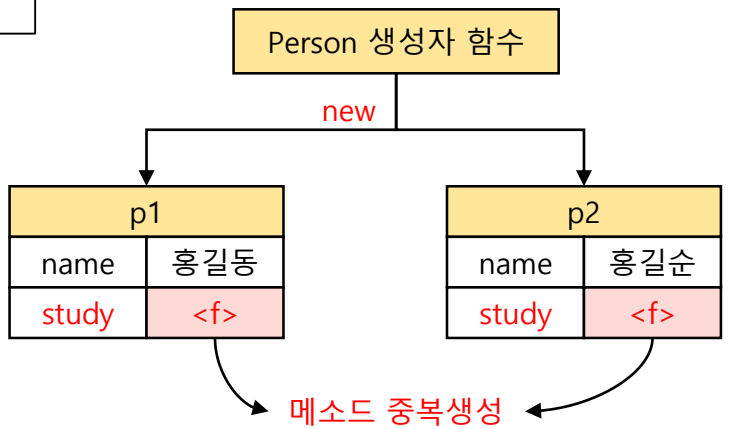
- **자바스크립트는 프로토타입을 기반으로 상속을 구현여 불필요한 중복을 제거한다.**
- **constructor함수 (함수선언문, 함수표현식)는 propertype 프로퍼티를 소유한다.**
  - ✓ prototyp속성에 인스턴스 메소드를 추가하여 상속 메커니즘이 적용된다.
- **non-constructor 인 화살표함수와 ES6의 메서드 축약표현으로 정의한 메서드는 propertype 프로퍼티를 소유하지 않으며 프로토타입 객체도 생성하지 않는다.**
- **모든객체에는 [[prototype]]내장 속성이 있고 Object.prototype.__proto__ 접근자 메소드를 통해서 부모객체 참조를 확인 할 수있다.**

프로토타입

➢ 생성자함수 내의 메서드는 객체 생성시 중복 생성된다.

- ✓ 인스턴스 중복 메서드 생성은 메모리 낭비 초래
- ✓ 인스턴스를 생성할때 마다 메서드를 중복 생성하므로 퍼퍼먼스 악영향

```
function Person(name){
  this.name = name;
  this.study = function(){
    console.log(`${this.name}님이 javascript 공부합니다`);
  }
}

const p1 = new Person('홍길동');
const p2 = new Person('홍길순');

p1.study(); //홍길동님이 javascript 공부합니다
p2.study(); //홍길순님이 javascript 공부합니다
```



프로토타입

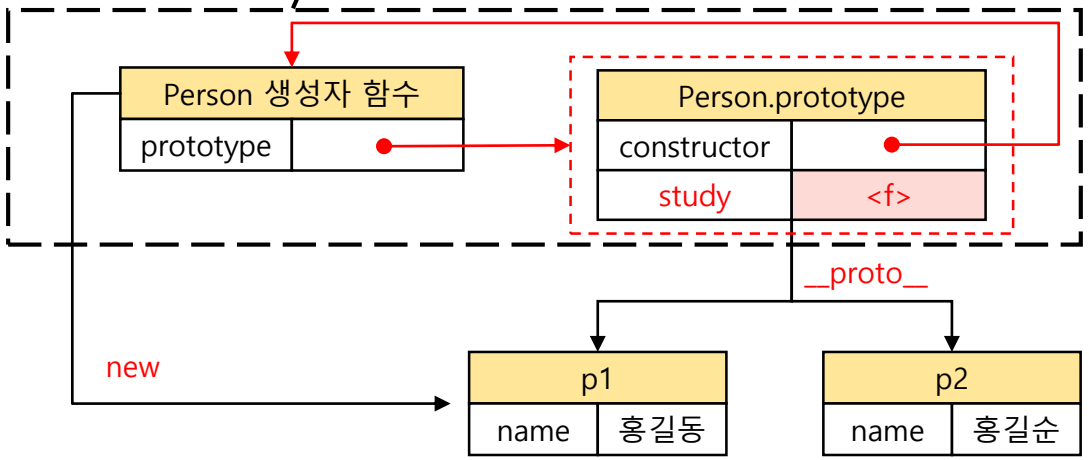
➢ 자바스크립트는 프로토타입을 통해 상속을 구현한다.

✓ 생성자함수는 개별적으로 소유하는 프로퍼티만 소유하고
동일한 프로퍼티는 프로토타입 객체에 정의 한다.

```
function Person(name){
  this.name = name;
}

Person.prototype.study = function() {
  console.log(`${this.name}님이 javascript 공부합니다`)
};

const p1 = new Person('홍길동');
const p2 = new Person('홍길순');

p1.study();//홍길동님이 javascript 공부합니다
p2.study();//홍길순님이 javascript 공부합니다
```

생성자함수는 사용자정의함수와 자바스크립트가 기본으로 제공하는 빌트인 함수가 있다.

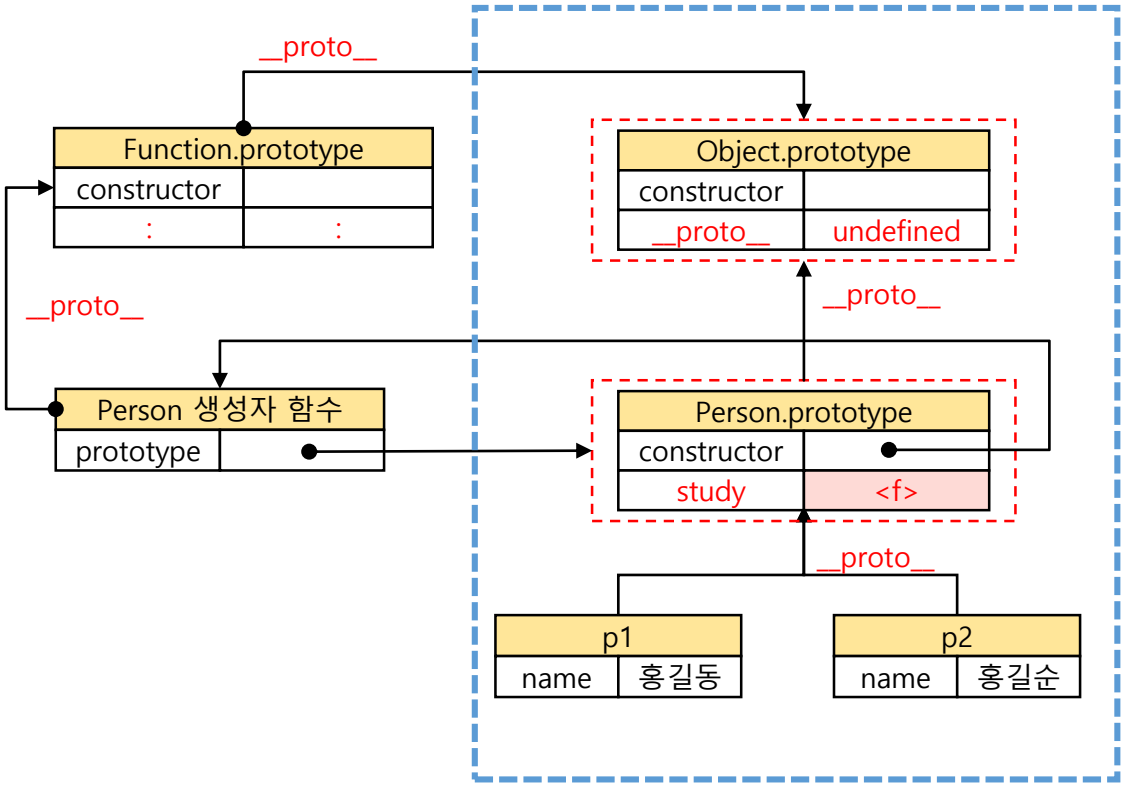
생성자함수가 생성될 때 프로토타입 객체도 생성되어 언제나 쌍으로 존재함.



프로토타입

➢ prototype, constructor, __proto__([[prototype]]) 속성관계

- ✓ 모든 생성자함수 객체는 prototype속성을 갖는다.
- ✓ 모든 프로토타입 객체는 constroctor속성을 갖는다.
- ✓ 모든 객체는 Object.prototype을 상속받아 __proto__속성을 갖는다.
- ✓ 항상 생성자 함수와 프로토타입은 한쌍으로 존재함.



프로토타입 체인

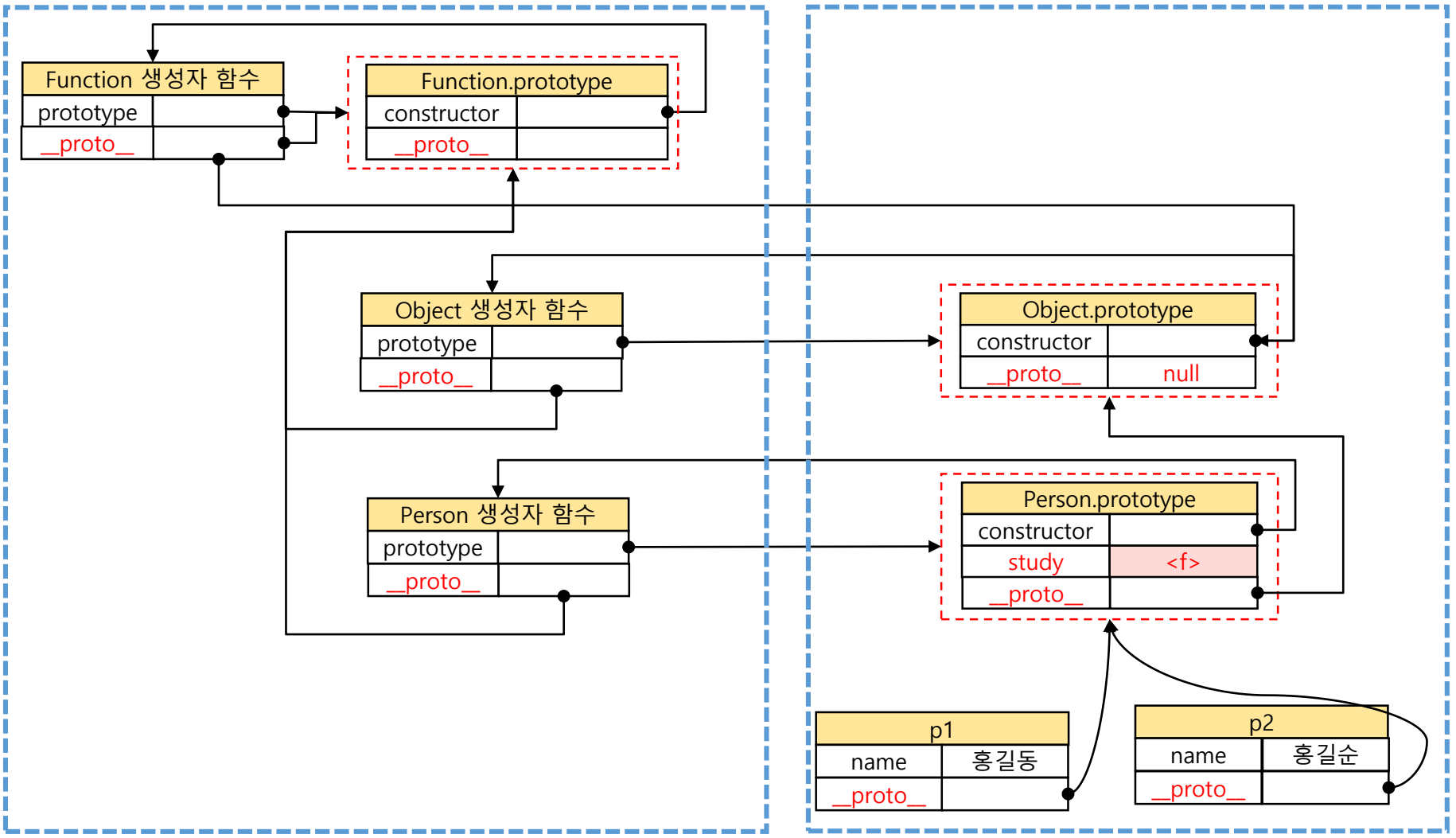
➢ 프로토타입 체인

- ✓ 종점엔 Object.prototype가 있다
- ✓ 상속과 프로퍼티 검색을 위한 메커니즘
- ✓ 프로퍼티가 아닌 식별자는 스코프체인에서 검색
- ✓ 스코프체인은 식별자 검색 메커니즘이고 프로토타입 체인은 프로퍼티 검색 메커니즘이다

프로토타입 체인

생성자함수는 Function.prototype 을 상속받음

프로토타입 체인은 객체의 __proto__ 를 탐색



☞ 객체의 prototyp의 접근은 __proto__로 접근하지말고 Object.getPrototypeOf, Object.setPrototypeOf 메소드 사용할것!

프로퍼티 존재확인 / 열거

➢ 프로퍼티 존재 확인

- ✓ in 연산자
  - 프로토타입 체인에 속한 프로퍼티이면 true를 반환한다.
- ✓ Reflect.has
  - 프로토타입 체인에 속한 프로퍼티이면 true를 반환한다.
  - in 연산자와 동일
- ✓ Object.prototype.hasOwnProperty
  - 객체 고유의 프로퍼티인 경우 true를 반환한다.

➢ 프로퍼티 열거

- ✓ for(변수선언 in 객체){ } : 객체의 프로퍼티 키를 변수에 할당
  - 프로토타입 체인에 속한 프로퍼티까지 열거한다.
    - 프로퍼티 어트리뷰트 [[Enumerable]]가 false인 경우는 제외 한다.
    - 심벌 프로퍼티는 제외 한다.
    - 프로퍼티 순서는 보장하지 않는다.(숫자인 경우는 정렬실시)
- ✓ Object.keys/ Object.values / Object.entries
  - 객체 자신의 고유 프로퍼티만 열거하여 배열로 반환 한다.

자바스크립트 엔진 실행의 2단계

- **소스코드 평가단계**
  - ✓ 변수 선언을 포함한 모든 선언문이 실행된다
  - ✓ **호이스팅** 발생
    - 선언문을 코드의 선두로 끌어 올리는 것처럼 동작되는 특징
    - 변수,함수등.
- **소스코드 실행단계**
  - ✓ 모든 선언문 제외한 소스코드를 한 줄 씩 순차적 실행

실행컨텍스트

- **소스코드 유형 4가지는 실행 컨텍스트를 생성한다.**
  - ✓ 전역코드
  - ✓ 함수코드
  - ✓ eval 코드
  - ✓ 모듈코드
- **실행컨텍스트**
  - ✓ 소스코드를 실행하기 위해 필요한 환경을 제공하고 코드의 실행 결과를 실제로 관리하는 영역
- **식별자(변수,함수,클래스 등의 이름)를 등록하고 관리하는 스코프와 코드실행 순서 관리를 구현한 내부 메커니즘으로, 모든 코드는 실행 컨텍스트를 통해 실행되고 관리된다.**

실행컨텍스트

➢ 실행컨텍스트 구조

- ✓ 렉시컬환경(LexicalEnvironment)
  - ▪ 환경레코드(EnvironmentRecord) : 식별자와 식별자에 바인딩된 값,
    - 객체환경레코드(BindingObject) : 전역객체(window)를 참조, var로 선언한 전역변수
    - 선언적환경레코드(Declarative Environment Record):let, const로 선언한 전역변수
    - [[GlobalThisValue]]내부 슬롯에 this가 바인딩 되며 전역객체(window)를 참조
  - ▪ 외부렉시컬환경에 대한참조(OuterLexicalEnvironmentReference):상위스코프에 대한 참조를 기록
- ✓ VariableEnvironment

➢ 식별자 결정

- ✓ 실행중인 실행 컨텍스트에서 식별자를 검색하기 시작하여 없으면 외부렉시컬환경을 참조한다.
  즉 상위 스코프 체인을 따라 외부코드로 이동한다.

this

➢ this 란?

- ✓ **자신이 속한 객체 또는 자신이 생성할 인스턴스를 가리키는 자기 참조변수**
  - 객체는 상태를 나타내는 프로퍼티와 동작을 나타내는 메서드를 하나의 논리적인 단위로 묶은 복합적인 자료구조다.
  - 동작을 나타내는 메서드는 자신이 속한 객체의 상태, 즉 프로퍼티를 참조하고 변경할 수 있어야 함

➢ this 바인딩이란?

- ✓ **this와 this가 가리키는 객체를 연결하는 과정**
  - 바인딩 : 식별자와 값을 연결하는 과정
- ✓ **this 바인딩은 함수 호출 방식에 의해 동적을 결정된다.**
- ✓ this는 자바스크립트 엔진에 의해 암묵적으로 생성되며, 코드 어디서든 참조할 수 있다.
- ✓ 자바스크립트의 this는 함수가 호출되는 방식에 따라 동적으로 객체(인스턴스)가 결정된다.
- ✓ 함수를 호출하면 arguments 객체와 this가 암묵적으로 생성되어 함수 내부에 전달된다.

this

➢ this의 사용위치

- ✓ 코드 어디서든 참조 가능하다
- ✓ 하지만, this는 객체의 프로퍼티나 메소드를 참조하기 위한 자기참조 변수이므로 **객체의 메서드 내부 또는 생성자함수 내부에서만 의미가 있다.**
  - strict mode가 적용된 일반함수 내부의 this는 undefined가 바인딩 된다.
  - class내부는 strict mode가 암묵적으로 적용된다.따라서 class 내부에 사용된 일반함수내의 this는 undefined가 바인딩된다.

this

➢ this의 사용위치에 따른 의미

- ✓ **전역**
  - this는 전역객체(window, globalThis)를 뜻함.
- ✓ **일반함수**(중첩함수,콜백함수포함)
  - this는 **전역객체**
- ✓ **생성자함수**
  - this는 생성자 함수가 **미래에 생성할 인스턴스**
- ✓ **메서드(es6의축약함수)내**
  - this는 메서드를 **호출한 객체**에 바인딩
  - **메서드와 메서드 내부의 일반함수의 this바인딩 일치시키는 방법**
    - ① 메소드의 this를 복사 ② bind함수 ③ 화살표함수
- ✓ **화살표함수**
  - **this바인딩을 갖지 않는다.**
  - 화살표 함수 내부의 this는 상위 스코프의 this를 참조한다.

함수 호출방식

- **일반 함수 호출**
- **메서드 호출**
- **생성자함수 호출**
- **Function.prototype.apply/call/bind 메서드에 의한 간접호출**
  - ✓ apply,call,bind메서드에 첫번째 인수로 전달한 객체

함수 호출방식과 this 바인딩

```
// this 바인딩은 함수 호출 방식에 따라 동적으로 결정된다.
const foo = function () {
  console.dir(this);
};
// 동일한 함수도 다양한 방식으로 호출할 수 있다.
// 1. 일반 함수 호출
// foo 함수를 일반적인 방식으로 호출
// foo 함수 내부의 this는 전역 객체 window를 가리킨다.
foo(); // window

// 2. 메서드 호출
// foo 함수를 프로퍼티 값으로 할당하여 호출
// foo 함수 내부의 this는 메서드를 호출한 객체 obj를 가리킨다.
const obj = { foo };
obj.foo(); // obj
```

```
// 3. 생성자 함수 호출
// foo 함수를 new 연산자와 함께 생성자 함수로 호출
// foo 함수 내부의 this는 생성자 함수가 생성한 인스턴스를 가리킨다.
new foo(); // foo {}

// 4. Function.prototype.apply/call/bind 메서드에 의한 간접 호출
// foo 함수 내부의 this는 인수에 의해 결정된다.
const bar = { name: 'bar' };

foo.call(bar);   // bar
foo.apply(bar);  // bar
foo.bind(bar)(); // bar
```

이터레이션/이터러블/이터레이터 프로토콜

➢ 이터레이션 프로토콜

- ✓ ES6에서 순회가능한 테이터 컬렉션(자료구조)을 만들기 위해 정의된 규칙
- ✓ 데이터 공급자가 하나의 순회 방식을 갖도록 규정
  - 이 규칙나오기전에는 개별 객체의 메소드를 사용해서 요소를 순회하는 방식 이었고 이를 통일하여 요소 순회 방식을 일원화 하였음.
- ✓ ex) Array,Set,Map,String,TypedArray,arguments,DOM컬렉션 등
- ✓ 유사배열 객체이면서 이터러블인 객체가 있다.(arguments,NodeList,HTMLCollection 등)
  - 유사배열이란? 배열처럼 인덱스로 접근할 수 있고 length프로퍼티가 있다. (**배열 메소드사용불가**)

➢ 이터러블 프로토콜

- ✓ 이터레이션 인터페이스(Symbol.iterator)를 구현한 객체
  - 이터러블 객체의 Symbol.iterator를 호출하면 이터레이터를 반환
- ✓ for ~ of, 스프레드문법, 디스트럭처링 할당의 대상으로 사용할 수 있다.

➢ 이터레이터 프로토콜

- ✓ 이터러블의 요소를 탐색하기 위한 포인터
- ✓ { value, done } 이터레이터 리절트 객체를 반환하는 next메소드를 가진 객체

이터러블 이면서 이터레이터인 객체?

- **Symbol.iterator(), next()를 소유한 객체**
  - ✓ Symbol.iterator()를 호출하지 않고도 next()를 직접 호출할 수 있는 객체
  - ✓ 내부구조

```
{
  [symbol.iterator]() { return this; }
  next() {
    return{ value:any, done:Boolean }
  }
}
```

제너레이터

- 이터러블 이면서 이터레이터인 객체를 생성하는 함수

지연 평가(Lazy evaluation)

➢ 평가 결과가 필요할 때까지 평가를 늦추는 기법

- ✓ 데이터가 필요한 시점 이전까지는 미리 데이터를 생성하지 않다가
  데이터가 필요한 시점이 되면 비로소 데이터를 생성하는 기법
- ✓ 무한 이터러블을 통해 지연평가를 가능케한다.
- ✓ for~of,스프레드, 배열 디스트럭처링이 적용되기 전까지 데이터를 생성하지 않음.
  - for~of문에서는 이터러블을 순회할 때 이터레이터의 next()를 호출할때 데이터가 생성됨.
- ✓ cf) 배열,문자열 등은 모든 데이터를 메모리에 확보한 다음 데이터를 공급한다.

➢ 장점

- ✓ 빠른 실행 속도
  - 불필요한 데이터를 미리 생성할 필요가 없다.
- ✓ 불필요한 메모리 소비 방지
- ✓ 무한 표현이 가능

Set

- **중복되지 않는 유일한 값들의 집합. 수학적 집합을 구현하기위한 자료구조**

구분	배열	Set 객체
동일한 값 중복 유무	O	X
요소의 순서의미	O	X
인덱스 접근	O	X
요소 개수	arr.length	set.size

Set

➢ 생성

- ✓ const set = new Set([1,2,3]);

➢ 요소갯수 확인

- ✓ const { size } = new Set([1,2,3]);

➢ 요소추가

- ✓ set.add(4).add(5).add(5); //중복은 무시됨
- ✓ set.add(6).add('a').add(true).add(undefined).add(null)
  .add({}).add([]).add(()=>{}) // 배열, 객체(함수포함) 모든값 요소추가가능

➢ 요소존재여부 확인

- ✓ set.has(2); //true

➢ 요소삭제

- ✓ set.delete(2); //개별삭제
- ✓ set.clear(); //일괄삭제

Set

➢ 요소순회

- ✓ forEach((p1,p2,p3)=>{ })
  - p1,p2 : 현재 순회중인 요소값
  - p3: 현재 순회중인 Set객체 자체
- ✓ Set는 이터러블객체이브로 for~of 사용

➢ 집합연산

- ✓ 교집합 : Set.Prototype.intersection()
- ✓ 합집합 : Set.prototype.union()
- ✓ 차집합 : Set.prototype.difference

Map

- **키와 값의 쌍으로 이뤄진 객체를 저장하는 자료구조**

구분	객체	Map 객체
키로 사용할수 있는 값	문자열(빈문자포함),심벌	객체를 포함한 모든값
이터러블	X	O
요소 개수 확인	Object.keys(obj).length	map.size

Map

➢ 생성

- ✓ const map = new Map([['key1','value1'], ['key2','value2'], ['key3','value3']);

➢ 요소갯수 확인

- ✓ const { size } = new Map([['key1','value1'], ['key2','value2'], ['key3','value3']);

➢ 요소추가

- ✓ map.set([['key4','value4'], ['key4','value5']); //동일키는 값에대해 덮어쓰기됨.
- ✓ map.set(lee, 'developer') // 배열, 객체(함수포함) 모든값을 키로 사용 가능

➢ 요소취득

- ✓ map.get('key'); // 'value1'

➢ 요소존재여부 확인

- ✓ map.has('key1'); //true

➢ 요소삭제

- ✓ map.delete('key1'); //개별삭제
- ✓ map.clear(); //일괄삭제

Map

➢ 요소순회

- ✓ forEach((p1,p2,p3)=>{ })
  - p1 : 현재 순회중인 요소값
  - p2 : 현재 순회중인 요소키
  - p3: 현재 순회중인 Map객체 자체
- ✓ Map는 이터러블객체이브로 for~of 사용

➢ 이터러블이면서 이터레이터 객체를 반환하는 메소드

Map 메서드	설명
Map.prototype.keys	Map객체에서 요소키를 값으로 갖는 객체 반환
Map.prototype.values	Map객체에서 요소값를 값으로 갖는 객체 반환
Map.prototype.entries	Map객체에서 요소키와 요소값을 값으로 갖는 객체 반환

spread syntax(스프레드 문법,전개 연산자)

- 이터러블 객체 요소를 펼쳐서(전개,분산,spread) **개별적인 값들의 목록**으로 만든다.
  - ✓ Rest 파라미터(목록을 배열로)와 반대개념
- 대상
  - ✓ 이터러블 객체 : for~of문으로 순회할 수 있는 객체
    ex) Array, String, Map, Set, DOM컬렉션(NodeList, HTMLCollection),arguments
  - ✓ 일반 객체를 대상으로도 사용가능
- 문법
  - ✓ ... + 이터러블
  - ✓ 스프레드 문법의 결과는 변수에 할당할 수 없다.
- 용도
  - ✓ 배열 복사, 추가, 병합
  - ✓ 개체 복사, 추가, 병합
    - object.assign()대체 문법으로 유용

spread syntax(스프레드 문법,전개 연산자)

- 사용 문맥
  - ✓ 함수 호출문의 인수 목록
  - ✓ 배열 리터럴의 요소 목록
    - 배열 복사, 병합, 추가
    - 이터러블을 배열로 변환 [...iter], Array.from[iter]
  - ✓ 객체 리터럴의 프로퍼티 목록
    - 객체 복사, 병합, 추가, 변경

이터러블 객체를 배열로 변환하는 방법 2가지

- **변환하는 이유**
  - ✓ 배열 객체가 가지고 있는 고차함수를 사용하기 위함.
- **스프레드문법**
  - ✓ 이터러블 객체 => 배열로변환(O)
  - ✓ 유사배열 객체 => 배열로변환(X)
- **Array.from()**
  - ✓ 이터러블 객체 => 배열로변환(O)
  - ✓ 유사배열 객체 => 배열로변환(O)

Destructuring assignment(디스트럭처링 할당,구조분해)

- 구조화된 배열과 같은 이터러블 또는 객체를 destructuring(비구조화,구조파괴)하여 **1개 이상의 변수에 개별적으로 할당 할때** 사용되는 문법
- 대상
  - ✓ 배열, 객체
- 용도
  - ✓ 객체 프로퍼티키중 필요한 프로퍼티 값만 추출하여 개별 변수에 저장하고 싶을때 사용
  - ✓ 배열 디스트럭처링과 객체 디스트럭처링을 혼용 하여도 사용가능
    - ▪ 배열의 요소가 객체인 경우 배열 디스트럭처링과 객체 디스트럭처링 할당 혼용

Destructuring assignment(디스트럭처링 할당,구조분해)

- 배열 디스트럭처링
  - ✓ **할당 기준은 인덱스 순서**
  - ✓ const [one, two, three] = [1,2,3];
    console.log(one, two, three); // 1 2 3
  - ✓ const [e, f=10, g=3] = [1, 2];
    console.log(e,f,g); // 1 2 3
  - ✓ const[x, ...y] = [1,2,3]; // ...y : Rest요소 cf) Rest파라미터: 메소드의 매개변수 사용시(목록->배열)
    console(x, y); // 1 [2,3]
- 객체 디스트럭처링
  - ✓ **할당 기준은 프로퍼티 키** 즉, 선언될 변수명과 프로퍼티키가 일치하면 된다.
    - const {lastname, firstname} = { firstname:"sanghak", lastname:"yi" }
      console.log(lastname,firstname) // yi sanghak
  - ✓ 변수 이름을 바꾸고자할때 (:뒤 변경할 이름지정)
    - const {lastname:ln, firstname:fn} = { firstname:"sanghak", lastname:"yi" }
      console.log(fn, ln); // yi sanghak

옵셔널체이닝 / null병합 연산자

➢ 옵셔널체이닝

- ✓ 좌항의 피연산자가 null 또는 undefined인 경우 undefined를 반환하고
  그렇지 않으면 우항의 프로퍼티 참조를 이어간다.
- ✓ 문법(?.)
  - 좌항?.우항
  - ex) const value = elem?.value;
- ✓ 문맥
  - 변수가 참조하는 객체가 있는 경우만 프로퍼티에 접근하고자 할 때 유용.

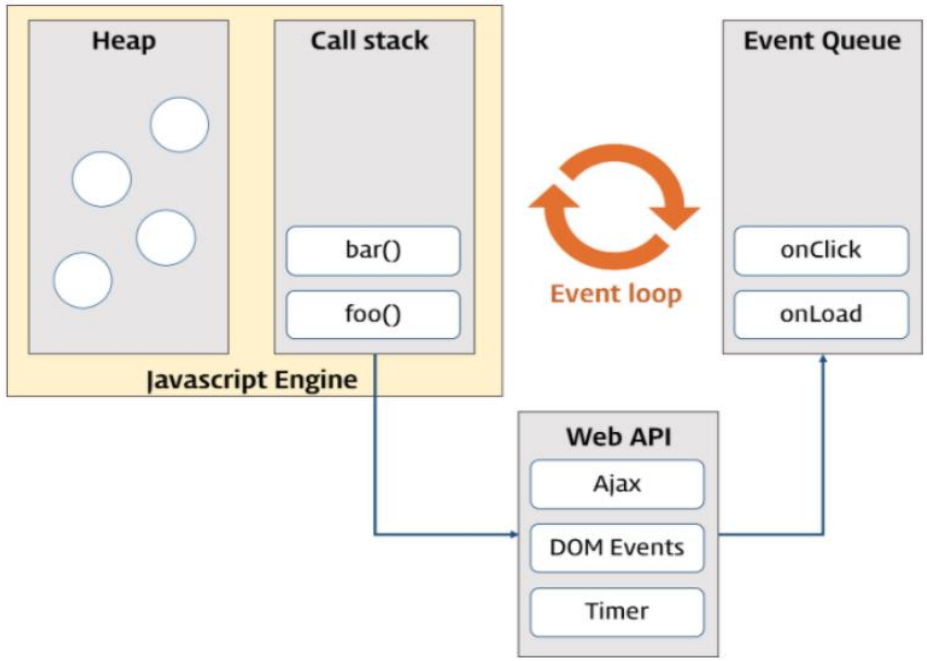
➢ null병합 연산자

- ✓ 좌항의 피연산자가 null 또는 undefine인경우 우항을 반환하고
  그렇지 않으면 좌항의 피연산자를 반환한다.
- ✓ 문법(??)
  - ex) var foo = null ?? 'default string';
- ✓ 문맥
  - 변수에 기본값을 설정할 때 유용하다.

자바스크립트 엔진

- **자바스크립트는 싱글 스레드 기반 언어다. 호출 스택이 하나.**
  - ✓ 한번에 한 작업만 순차 처리할 수 있다.
  - ✓ 작업이 길어지는 경우는 비동기 콜백 사용
- **엔진의 주요 2가지 구성요소**
  - ✓ Memory Heap : 메모리 할당이 일어나는 곳
  - ✓ Call Stack : 코드 실행에 따라 호출 스택이 쌓이는 곳
- **Web APIs**
  - ✓ 브라우저에서 제공하는 API
- **EventQueue**
  - ✓ 마이크로태스크 큐
    - exPromise, Async
  - ✓ 매크로태스크 큐

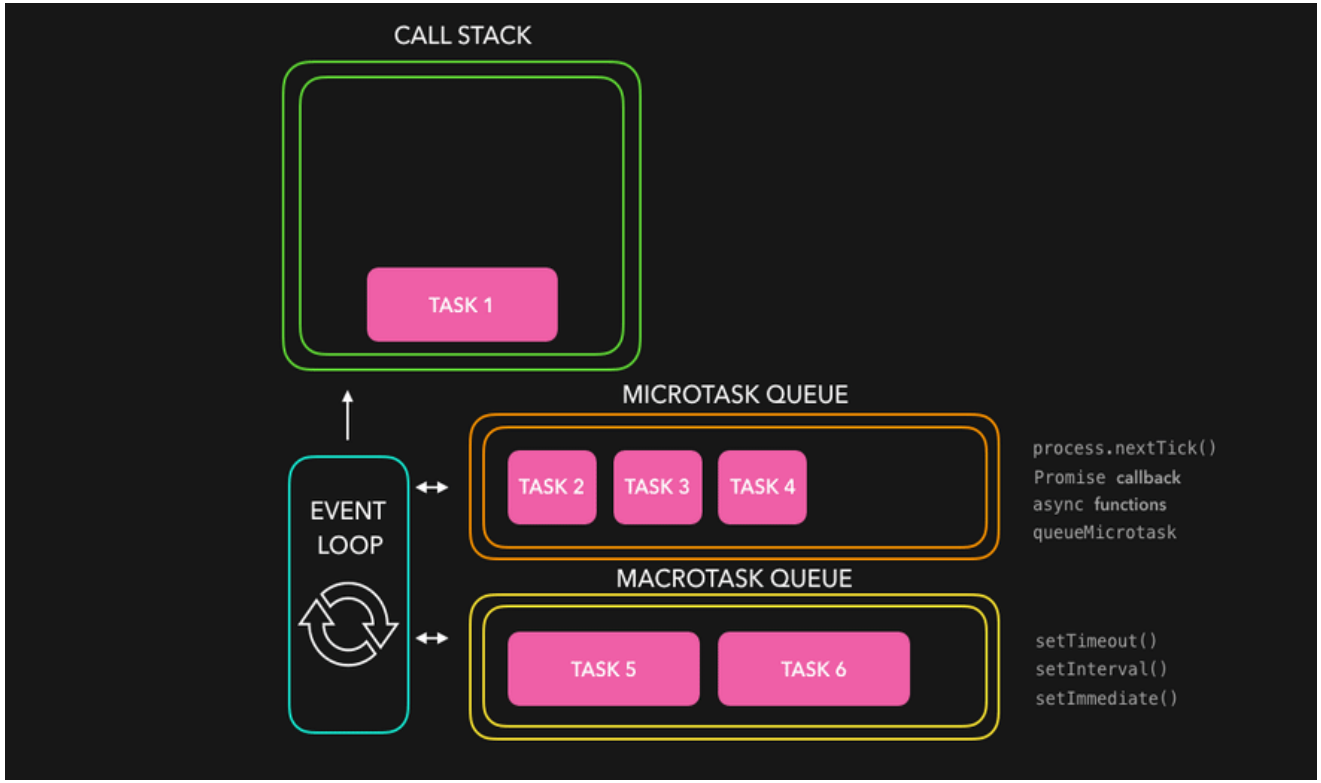
    ex) Ajax, Dom evnet, Timer



- **Event Loop**
  - ✓ 자바스크립트의 **동시성을 지원**하는 브라우저의 내장 기능 중 하나.

마이크로태스크 큐

- **프라미스의 후속처리 메소드는 마이크로 태스크 큐에 등록된다.**
  - ✓ 콜 스택이 비어있을때 이벤트 루프에의해 호출되는 태스크 큐의 우선순위
    - 마이크로 태스크 큐 > 매크로 태스크 큐



자바스크립트의 비동기 처리 패턴

- callback
- promise
- promise + generator
- async & await

타이머

➢ 호출 스케줄링

- ✓ 함수 호출을 예약하려면 타이머 함수 사용
- ✓ setTimeout, setInterval : Ecmascript사양에 정의된 빌트인함수가 아니라 실행환경(브라우저,Node.js)에서 모두 전역객체로 제공되는 호스트객체다

➢ 타이머 함수는 생성한 타이머가 만료되면 콜백 함수가 호출된다.

- ✓ setTimeout / clearTimeout : 단 한번동작
  - setTimeout(func|code[, delay, param1, param2, …]);
- ✓ setInterval / clearInterval : 반복동작
  - setInterval(func|code[, delay, param1, param2, …]);

타이머

➢ 디바운스,스크틀

- ✓ 과도한 이벤트 핸들러의 호출을 방지하는 프로그래밍 기법
  - ex)scroll,resize,input,mouseove,mouseover 같은 이벤트는 짧은 시간 간격으로 연속해서 과도하게 호출되어 성는에 문제를 일으킬 수 있다.

➢ 디바운스

- ✓ 짧은 시간 간격으로 발생하는 **이벤트를 그룹화 해서 마지막에 한번만 이벤트 핸들러가 호출되도록** 하는 프로그래밍 기법.
  - 사용사례
    - resize 이벤트처리
    - input요소에 입력된 값으로 ajax요청하는 필드 자동완성 UI구현
    - 버는 중복 클릭 방지

➢ 스로틀

- ✓ 짧은 시간 간격으로 이벤트가 연속해서 발생하더라도 **일정 시간 간격으로 이벤트 핸들러가 최대 한번만 호출되도록** 하는 프로그래밍 기법
  - 사용사례
    - scroll 이벤트 처리나 무한 스크롤 UI구현에 유용

타이머

- **예시**

```
const debounce = (callback, delay) => {
      let timerId;
      return event => {
        if (timerId) clearTimeout(timerId);
        timerId = setTimeout(callback, delay, event);
      };
};

const throttle = (callback, delay) => {
    let timerId;
    return event => {
      if (timerId) return;
      timerId = setTimeout(() => {
        callback(event);
        timerId = null;
      }, delay, event);
    };
};
```

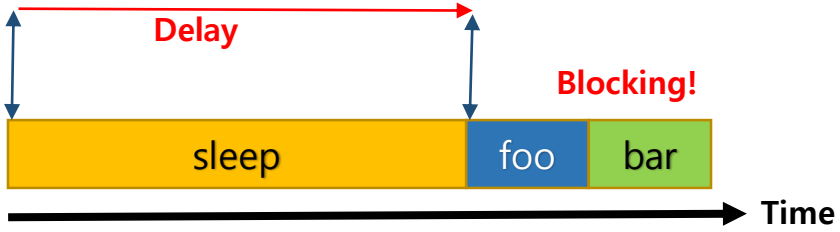
비동기 프로그래밍

- 함수가 종료하지 않더라도 호출자에게 제어권을 반환하는 경우 이 함수를 비동기 함수라하고 이런 처리방식을 비동기 처리라고 함. 이경우 호출자 이후 로직이 블로킹(작업중단)없이 다음 흐름을 이어갈 수 있음.
- 비동기함수는 비동기 처리 결과를 외부에 반환할수 없고 상위 스코프의 변수에 할당할 수도 없다. 따라서 비동 기함수의 처리결과에 대한 후속 처리는 비동기 함수 내부에서 수행해야 한다. 이때 콜백함수를 전달하는것이 일반적이다.
- 타이머함수인 setTimeout, setInterval, HTTP요청, 이벤트 핸들러는 비동기 처리 방식으로 동작한다.

동기/비동기 비교

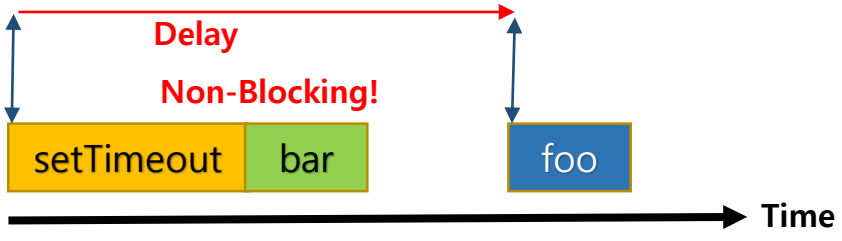
➢ 동기프로그래밍 예제

```
function sleep(func, delay) {
  const delayUntil = Date.now() + delay;
  while (Date.now() < delayUntil);
  func();
}
function foo() {  console.log('foo'); }
function bar() {  console.log('bar'); }
sleep(foo, 3 * 1000);
// bar 함수는 sleep 함수의 실행이 종료된 이후에 호출되므로 3초 이상 블로킹된다.
bar();
// (3초 경과 후) foo 호출 -> bar 호출
```



➢ 비동기 프로그래밍 예제

```
function foo() {  console.log('foo'); }
function bar() {  console.log('bar'); }
setTimeout(foo, 3 * 1000);
// 타이머 함수 setTimeout은 일정 시간이 경과한 이후에
// 콜백 함수 foo를 호출한다.
// 타이머 함수 setTimeout은 bar 함수를 블로킹하지 않는다.
bar();
// bar 호출 -> (3초 경과 후) foo 호출
```



동기/비동기 비교

➢ 동기/비동기 장단점

	장점	단점
동기	순서보장	블로킹
비동기	블로킹(X)	순서보장(X)

➢ 비동기방식 예제

- ✓ 타이머함수
  - setTimeout / setInterval
- ✓ HTTP요청
  - Ajax(HttpXMLRequest, fetch, promise, async)
- ✓ DOM 이벤트 핸들러
  - on+이벤트타입 프로퍼티, addEventListener

HTTP요청 전송 방법

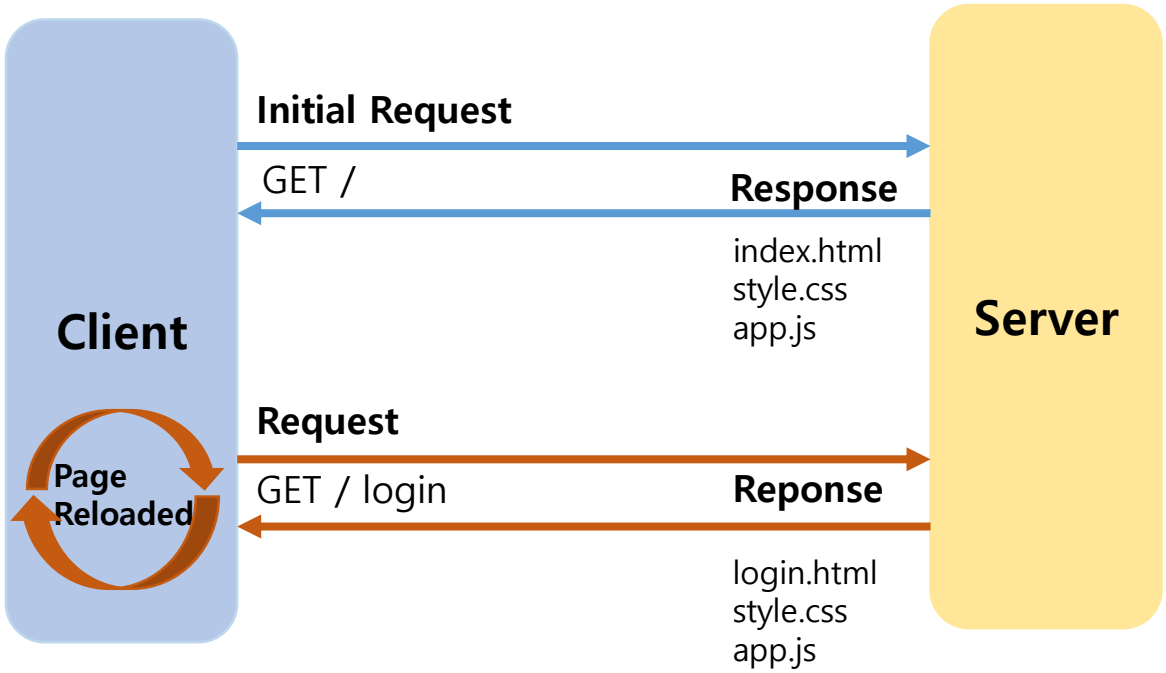
- **브라우저의 주소창**
  - ✓ HTTP요청 방식 : GET
- **HTML의 form태그의 href속성 metho속성**
  - ✓ HTTP요청 방식 : GET, POST
- **HTML의 a태그의 href속성**
  - ✓ HTTP요청 방식 : GET
- **JS의 XMLHttpRequest객체 사용**
  - ✓ HTTP요청 방식 : GET, POST, PUT, PATCH, DELETE 등
- **Spring은 이를 위해서 HiddenHttpMethodFilter를 제공**
  - ✓ Spring에서는 form태그에 hidden필드를 두어 이를 해결함.
  - ✓ Thymeleaf를 사용할경우 form태그에 th:method="put" 속성을 주면 hidden필드가 자동생성됨.
  - ✓ 서버설정 : application.properties
    - spring.mvc.hiddenmethod.filter.enabled=true

AJAX(Asynchornous JavaScript and XML)

- 자바스크립트를 사용하여 브라우저가 서버에게 **비동기 방식으로 데이터를 요청**하고 서버가 응답한 **데이터를 수신**하여 **웹페이지를 동적으로 갱신**하는 프로그래밍 방식
- Web API XMLHttpRequest객체를 기반으로 동작

AJAX(Asynchornous JavaScript and XML)

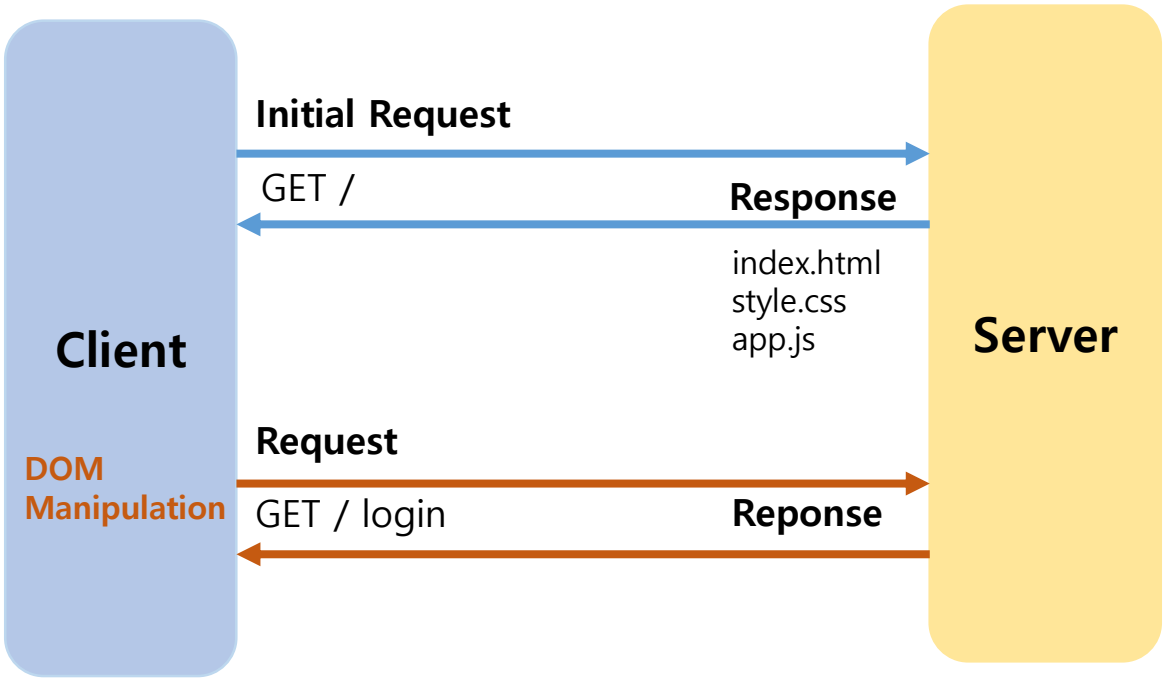
➢ 전통적인 웹페이지 생명주기



- ✓ 이전 웹페이지와 차이가 없는 불필요한 데이터 통신이 발생
- ✓ 처음부터 다시 렌더링으로 인해 화면 전환시 순간적인 깜박임 현상이 발생
- ✓ 클라이언트와 서버와의 통신이 동기방식으로 동작하기 때문에 블로킹 발생

AJAX(Asynchornous JavaScript and XML)

➢ AJAX



- ✓ 이전 웹페이지와 차이가 없는 불필요한 데이터 통신이 발생하지 않는다
- ✓ 변경할 필요가 없는 부분은 다시 렌더링하지 않으므로 순간적인 깜박임 현상이 발생하지 않는다
- ✓ 클라이언트와 서버와의 통신이 비동기방식으로 동작하기 때문에 블로킹 발생하지 않는다.

JSON

- **클라이언트와 서버간의 HTTP통신을 위한 텍스트 데이터 포맷**
- **JSON표기방식**

```
{
   "이름": "홍길동",
   "나이": 25,
   "성별": "여",
   "주소": "서울특별시 양천구 목동",
   "특기": ["농구", "도술"],
   "가족관계": {"#": 2, "아버지": "홍판서", "어머니": "춘섬"},
   "회사": "경기 수원시 팔달구 우만동"
}
```

  - ✓ JSON의 키는 반드시 큰따옴표
  - ✓ JSON의 값은 객체 리터럴과 같은 표기법, 문자열은 반드시 큰따옴표
  - ✓ 숫자와 불린값은 큰따옴표 생략가능

- **JSON의 정적메소드**
  - ✓ JSON.stringify : JS객체를 JSON포멧의 문자열로 변환
  - ✓ JSON.parse : JSON포맷의 문자열을 JS객체로 변환

REST API(Representational State Transfer)

- **REST**
  - ✓ HTTP를 기반으로 클라이언트가 서버의 리소스에 접근하는 방식을 규정한 아키텍처
- **RESTful**
  - ✓ REST의 기본 원칙을 성실히 지킨 서비스 디자인
- **REST API**
  - ✓ REST를 기반으로 구현한 서비스API
  - ✓ 자원(resource),행위(verb),표현(representations)의 3가지 요소로 구성

구성요소	내용	표현방법
자원(resource)	자원	URI
행위(verb)	자원에대한 행위	HTTP요청메서드
표현representations	자원에 대한 행위의 구체적 내용	페이로드(근본데이터)

➢ REST API 설계원칙

- ✓ URI : 리소스, 명사를 사용
- ✓ HTTP요청 메서드 : 리소스에 대한 행위(GET,POST,PUT,PATCH,DELETE등)

HTTP요청 메소드	종류	목적	페이로드
GET	Index/retrieve	모든/특정 리소스 취득	X
POST	Create	리소스 생성	O
PUT	Replace	리소스의 전체 교체	O
PATCH	Modify	리소스의 일부 수정	O
DELETE	delete	모든/특정 리소스 삭제	X

REST API

➢ 회원예제

- ✓ URI : 리소스, 명사를 사용
- ✓ HTTP요청 메서드 : 리소스에 대한 행위(GET,POST,PUT,PATCH,DELETE등)

요청	Method	URI	페이로드	양식
회원가입양식	GET	/members	X	joinForm.html
회원가입처리	POST	/members	O	
회원수정양식	GET	/members/{id}/edit /members/edit?id=xxx	X	editForm.html
회원수정처리	PATCH	/members/{id}/edit /members/edit?id=xxx	O	
회원조회	GET	/members/{id} /members?id=xxx	X	view.html, detail.html
회원탈퇴양식	GET	/members/{id}/out /members/out?id=xxx	X	outForm.html
회원탈퇴	DELETE	/members/{id} /members?id=xxx	X	
회원목록	GET	/members/{page}/{size} /members?page=xx&size=xx	X	list.html

REST API

➢ 공지사항

- ✓ URI : 리소스, 명사를 사용
- ✓ HTTP요청 메서드 : 리소스에 대한 행위(GET,POST,PUT,PATCH,DELETE등)

요청	Method	URI	페이로드	양식
등록양식	GET	/notices	X	registForm.html
등록처리	POST	/notices	O	
수정양식	GET	/notices/{no}/edit	X	editForm.html
수정처리	PATCH	/notices/{no}/edit	O	
조회	GET	/notices/{no}	X	detail.html
회원목록	GET	/notices/list	X	list.html

Rest API 연습

- **JSONPlaceholder에서 가상REST API사용**
  - ✓ https://jsonplaceholder.typicode.com/

Rest API 연습

➢ JSON Server사용

- ✓ $ mkdir json-server-exam && json-server-exam
- ✓ $ npm init –y
- ✓ $npm install json-server –save-dev

➢ db.json파일 생성

```
{
  "todos": [
    {
      "id": 1,
      "content": "HTML",
      "completed": true
    },
    {
      "id": 2,
      "content": "CSS",
      "completed": false
    },
    {
      "id": 3,
      "content": "Javascript",
      "completed": true
    }
  ]
}
```

➢ JSON Server 실행

- ✓ json-server –watch db.json //기본포트 3000
- ✓ json-server –watch db.json –port 5000

➢ package.json파일

```
{
  "name": "json-server-exam",
  "version": "1.0.0",
  "scripts": {
    "start": "json-server --watch db.json"
  },
  "devDependencies": {
    "json-server": "^0.16.1"
  }
}
```

➢ JSON Server 실행 => npm start;

➢ 서버종료 => ctrl + C

➢ 소스파일 위치

- ✓ json-server-exam/public 폴더

Promise

- **ECMAScript 사양에 정의된 표준 빌트인 객체다.**

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

- **프로미스는 처리상태와 처리결과를 관리하는 객체다**

- **비동기 함수**
  - ✓ 비동기로 동작하는 코드를 포함하는 함수

Promise의 후속처리 메소드

- **Promise.prototype.then**
  - ✓ 인수는 콜백함수로 프로미스의 비동기처리 결과를 전달 받는다.
  - ✓ 인수로 전달된 콜백함수의 반환은 언제나 프로미스다.
    - 값을 반환하면 암묵적으로 resolve 또는 reject하여 프로미스를 생성해 반환한다.
- **Promise.prototype.catch**
  - ✓ 인수는 콜백함수로 프로미스의 비동기처리 결과가 rejected상태인 경우만 호출된다.
  - ✓ 인수로 전달된 콜백함수의 반환은 언제나 프로미스다.
  - ✓ 모든 then메서드를 호출한 이후 호출하면 비동기 처리에서 발생한 에러뿐아니라 then메소드 내부에서 발생한 에러까지 모두 캐치한다.
- **Promise.prototype.finally**
  - ✓ 인수는 콜백함수로 프로미스의 비동기처리 결과상태와 상관없이 무조건 한번 호출된다.
  - ✓ 인수로 전달된 콜백함수의 반환은 언제나 프로미스다.

async/await

- 프로미스의 후속처리 메서드 없이 마치 동기처럼 프로미스가 처리결과를 반환 하도록 구현

정규표현식

➢ 정규표현식이란?

- ✓ 일정한 패턴을 가진 문자열의 집합을 표현하기 위해 사용하는 형식 언어(formal language)
- ✓ 검색 대상 문자열에 패턴 매칭 기능을 제공
  - ▪ 패턴매칭기능이란? 특정 패턴과 일치하는 문자열을 검색,추출,치환할 수 있는 기능
- ✓ 정규표현식을 사용하지 않으면 반복문과 조건식을 통해 한 문자씩 연속 체크로직이 필요

```
// 사용자로부터 입력받은 휴대폰 전화번호
const tel = '010-1234-5678';

// 정규 표현식 리터럴로 휴대폰 전화번호 패턴을 정의한다.
const regExp = /^\d{3}-\d{4}-\d{4}$/;

// tel이 휴대폰 전화번호 패턴에 매칭하는지 테스트(확인)한다.
regExp.test(tel); // -> false
```

정규표현식

➢ 정규표현식생성

✓ 정규표현식 리터럴

- 패턴: 문자열의 일정한 규칙을 표현
- 플래그:정규표현식의 검색 방식 설정

✓ 예제]

```
const target = 'Is this all there is?';

// 패턴: is
// 플래그: i => 대소문자를 구별하지 않고 검색한다.
const regexp = /is/i;

// test 메서드는 target 문자열에 대해 정규표현식 regexp의 패턴을 검색하여 매칭 결과를 불리언 값으로 반환한다.
regexp.test(target); // -> true
```

정규표현식

➢ 정규표현식생성

- ✓ RegExp생성자함수

```
const target = 'Is this all there is?';

const regexp = new RegExp(/is/i); // ES6
// const regexp = new RegExp(/is/, 'i');
// const regexp = new RegExp('is', 'i');

regexp.test(target); // -> true
```

정규표현식

➢ RegExp 메소드

✓ RegExp.prototype.exec : 매칭결과를 배열로 반환하며 매칭결과가 없는경우 null반환

- g플래그가 지정되면 첫번째 매칭결과만 배열로 반환

```
const target = 'Is this all there is?';
const regExp = /is/;

regExp.exec(target); // -> ["is", index: 5, input: "Is this all there is?", groups: undefined]
```

✓ **RegExp.prototype.test** : 매칭결과를 불리언값으로 반환

```
const target = 'Is this all there is?';
const regExp = /is/;

regExp.test(target); // -> true
```

정규표현식

➢ String 메소드

- ✓ String.prototype.search : 매칭결과를 일치하는 문자의 인덱스를 반환 없으면 -1반환
- ✓ String.prototype.replace : 매칭결과를 일치하는 문자를 첫번째 인수로 검색하고 두번째 인수로 대체
- ✓ **String.prototype.match** : 매칭결과를 배열로 반환
  - g플래그가 지정되면 모든 매칭결과를 배열로 반환

```
const target = 'Is this all there is?';
const regExp = /is/g;

target.match(regExp); // -> ["is", "is"]
```

정규표현식

➢ 플래그

✓ 정규표현 검색하는 방식 설정

플래그	의미	설명
i	Ignore case	대소문자를 구별하지 않고 패턴 검색
g	Global	대상 문자열 내에서 패턴과 일치하는 모든 문자열 전역 검색
m	Multi line	문자열의 행이 바뀌더라도 패턴 검색
y		시작 위치 고정 검색

```
const target = 'Is this all there is?';

// target 문자열에서 is 문자열을 대소문자를 구별하여 한 번만 검색한다.
target.match(/is/);
// -> ["is", index: 5, input: "Is this all there is?", groups: undefined]

// target 문자열에서 is 문자열을 대소문자를 구별하지 않고 한 번만 검색한다.
target.match(/is/i);
// -> ["Is", index: 0, input: "Is this all there is?", groups: undefined]

// target 문자열에서 is 문자열을 대소문자를 구별하여 전역 검색한다.
target.match(/is/g);
// -> ["is", "is"]

// target 문자열에서 is 문자열을 대소문자를 구별하지 않고 전역 검색한다.
target.match(/is/ig);
// -> ["Is", "is", "is"]
```

정규표현식

➢ 패턴

- ✓ 문자열의 일정한 규칙을 **메타문자** 또는 **기호**로 표현

➢ 메타문자

- ✓ 정규 표현식에서 특별한 뜻을 갖는 문자 ex) ^ $ \ . * + ? ( ) [ ] { } |
- ✓ 문자클래스 안에 사용하면 일반문자로 취급됨
  - 단] \ -를 원래문자로 사용할때는 \ 문자를 붙여야함.

정규표현식

➢ 문자클래스[...]

- ✓ []내의 문자는 선택(or)으로 동작함.
- ✓ 범위를 지정하려면 – 사용

➢ 부정문자클래스[^...]

- ✓ [ ]내의 ^은 부정(not)으로 동작함.
  - **[ ]밖의 ^은 문자열의 시작을 의미, [ ]밖의 $은 문자열의 마지막을 의미**
- ✓ [0-9] 은 \d 와 동일, [^0-9]은 \D 와 동일
- ✓ [a-zA-Z0-9_] 은 \w 와 동일, [^a-zA-Z0-9_] 은 \W 와 동일

➢ 문자클래스의 단축표기

- ✓ 임의의 문자 한 개 .
- ✓ 숫자와 순자외문자 \d, \D
- ✓ 단어문자(알파벳,순자,언더스코어)와 단어문자 외의 문자 \w, \W
  - [a-zA-Z0-9_]의 단축표기
- ✓ 공백문자(공백문자,탭문자,개행문자등)와 공백문자 외의 문자 \s, \S

정규표현식

➢ 반복패턴

- ✓ 바로앞의 요소가 최소m번, 최대 n번 : {m,n}
- ✓ 바로앞의 요소를 최소n번 : {n, }
- ✓ 바로앞의 요소를 n번 : {n }
- ✓ 최대 한번(0번포함) 반복 : ?
- ✓ 최소 한번 이상 반복 : +
- ✓ 최소 0번 이상 반복 : *

➢ 그룹화참조

- ✓ 그룹화(...)
  - ▪ 부분 정규 표현식이 되며 일치한 값은 별도로 저장(캡처링)되어 다시 참조 할 수 있음.
    - Ex) const header = /<(h[1-6])>.*< \/**\1**>/;
- ✓ 캡처링 없는 그룹화(?:...)

정규표현식

➢ 위치를 기준으로 매칭

- ✓ 문자열의 시작위치 ^
- ✓ 문자열의 마지막위치 $
- ✓ 영어단어의 경계 \ b
- ✓ 영어단어경계 외의 위치 ₩B
- ✓ 전방탐색 (?=pattern)
  - ex) x(?=y) 해석) x 다음에 y가 나오는 패턴
- ✓ 전방 부정 탐색 (?!pattern)
  - ex) x(?!y) 해석) x 다음에 y가 나오지 않는 패턴

➢ 선택(or) 패턴

- ✓ |

정규표현식

➢ 자주사용하는 정규표현식

✓ 특정단어로 시작하는지 검사

```
const url = 'https://example.com';

// 'http://' 또는 'https://'로 시작하는지 검사한다.
/^https?:\/\//.test(url); // -> true
```

```
/^(http|https):\/\//.test(url); // -> true
```

✓ 특정 단어로 끝나는지 검사

```
const fileName = 'index.html';

// 'html'로 끝나는지 검사한다.
/html$/.test(fileName); // -> true
```

✓ 숫자로만 이루어진 문자열인지 검사

```
const target = '12345';

// 숫자로만 이루어진 문자열인지 검사한다.
/^\d+$/.test(target); // -> true
```

정규표현식

➢ 자주사용하는 정규표현식

✓ 하나 이상의 공백으로 시작하는지 검사

```
const target = ' Hi!';

// 하나 이상의 공백으로 시작하는지 검사한다.
/^[\s]+/.test(target); // -> true
```

✓ 아이디로 사용 가능한지 검사

```
const id = 'abc123';

// 알파벳 대소문자 또는 숫자로 시작하고 끝나며 4 ~10자리인지 검사한다.
/^[A-Za-z0-9]{4,10}$/.test(id); // -> true
```

✓ 핸드폰 번호 형식에 맞는지 검사

```
const cellphone = '010-1234-5678';

/^\d{3}-\d{3,4}-\d{4}$/.test(cellphone); // -> true
```

정규표현식

➢ 자주사용하는 정규표현식

✓ 메일주소 형식에 맞는지 검사

```
const email = 'ungmo2@gmail.com';

/^[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*@[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*\.[a-zA-Z]{2,3}$/.test(email); // -> true
```

✓ 특수문자 포함 여부 검사

```
const target = 'abc#123';

// A-Za-z0-9 이외의 문자가 있는지 검사한다.
(/[^A-Za-z0-9]/gi).test(target); // -> true
```

```
(/[\{\}\[\]\/?.,;:|\)*~`!^\-_+<>@\#$%&\\\=\(\'\"]/gi).test(target); // -> true
```

```
target.replace(/[^A-Za-z0-9]/gi, ''); // -> abc123
```

용어정리

- **변수**
  - ✓ 하나의 값을 저장하기 위해 확보한 메모리 공간을 식별하기 위해 붙인 이름
  - ✓ 값의 위치를 가리키는 상징적인 이름.
    - 여러 개의 값을 저장 하기 위해 배열이나 객체같은 자료구조를 사용, 즉 관련 있는 여러 개 값을 그룹화함.
- **값(value)**
  - ✓ 표현식(expression)이 평가(evaluate)되어 실행된 결과
- **표현식(expression)**
  - ✓ 값으로 평가(evaluation)될 수 있는 모든 문(statement) , 표현식 여부는 변수에 할당해보면 판단가능
  - ✓ ex) 리터럴, 식별자(변수,함수,객체 등의 이름), 연산식, 함수 호출, 할당문 등
- **평가(evaluate) = 계산**
  - ✓ 표현식을 해석해서 **값을 생성**하거나 **참조**
  - ✓ 평가하려면 기호(리터럴과 연산자)의 의미를 알아야함
- **리터럴(literal)**
  - ✓ 사람이 이해할 수 있는 문자 또는 약속된 기호를 사용해 **값을 생성하는 표기법**
  - ✓ ex) 정수,문자열,불리언,null,객체,함수,정규표현식

용어정리

- **문(statement)**
  - ✓ 프로그램을 구성하는 기본 단위이자 최소 실행 단위
  - ✓ 즉 컴퓨터에 내리는 명령문
  - ✓ 문은 여러 토큰으로 구성된다
  - ✓ ex) 선언문, 할당문, 조건문, 반복문 등
- **토큰**
  - ✓ 문법적인 의미를 가지며 문법적으로 더 이상 나눌 수 없는 코드의 기본 요소
  - ✓ ex)키워드,식별자,연산자,리터럴,세미콜론,마침표
- **세미콜론**
  - ✓ 문의 종료
  - ✓ 블록{ } 뒤에는 세미콜론을 붙이지 않는다. 자체 종결성을 갖기 때문.
- **템플릿 리터럴**
  - ✓ 멀티 라인 문자열, 표현식 삽입, 태그드 템플릿 등을 사용해 편리한 **문자열 생성을 위한 표기법**
  - ✓ 백틱(`)을 사용해 런타임에 일반 문자열로 변환됨.
  - ✓ **표현식은 ${ }로 감싸며 평가결과는 문자열로 강제 형 변환됨.**

용어정리

➢ 식별자

- ✓ 어떤 값을 구별해서 식별할 수 있는 **고유한 이름**

➢ 식별자 네이밍 규칙

- ✓ 특수문자를 제외한 문자,숫자,언더스코어(_),달러기호($)를 포함 할 수 있다.
- ✓ 특수문자를 제외한 문자,언더스코어,달러기호로 시작할수 있다.(**숫자로 시작불가**)
- ✓ 예약어는 식별자 사용불가
- ✓ 관례
  - 상수
    - 대문자표기, 합성어는 스네이크케이스 표기
  - 카멜 케이스(camelCase) : 변수,함수
    - firstName
  - 파스칼 케이스(PascalCase) : 생성자함수, 클래스 이름
    - FirstName;
  - 스네이크 케이스(snake_case) : 상수
    - first_name

용어정리

- **변수 선언**
  - ✓ 변수를 생성 하는 문
  - ✓ 값을 저장하기위한 메모리공간 확보하고 변수이름과 메모리공간의 주소를 연결해서 값을 저장 할 수 있도록 준비
  - ✓ ex) var, let, const
- **초기화**
  - ✓ 변수가 선언된 이후 최초로 값을 할당하는 표현식
- **값의 할당**
  - ✓ 할당 연산자(=)를 사용하여 우변의 값을 좌변의 변수에 할당
- **값의 재할당**
  - ✓ 이미 할당되어 있는 변수에 새로운 값을 또다시 할당
  - ✓ const 키워드 변수는 재할당이 금지

용어정리

➢ 모듈성

- ✓ 프로그램을 더 작고 독립적인 부분(컴포넌트)으로 나눌 수 있는 정도
- ✓ 더 작은 조각으로 쪼갠 후 이들을 다시 재구성하여 해법을 완성하는 방식
- ✓ 개발자의 생산성,코드의 유지보수성 및 가독성 향상

➢ 함수 체인

- ✓ 고차 함수를 써서 하나의 래퍼 객체를 중심으로 단단히 결합된 메서드 체인으로 문제 해결

➢ 파이프라인

- ✓ 함수의 출력이 다음 함수의 입력이 되게끔 느슨하게 배열한,방향성 함수 순차열
  - ▪ 느슨한 결합
  - ▪ 무인수 형태로 문제를 해결하려면 코드를 적정 수준으로 추상해야 함.
  - ▪ 함수의 호환조건
    - 앞선함수의 반환타입과 수신함수의 입력타입 타입이 동일해야 한다
    - 수신함수는 한 개 이상의 매개변수를 선언해야 함.

용어정리

➢ 튜플

- ✓ 두가지 다른 값을 동시에 반환하는 불변성 자료구조
- ✓ 특징
  - 불변성 : 한번 만들어지면 값을 바꿀 수 없음
  - 임의 타입 생성 방지: 전혀 무관한 값을 서로 연관 지을 수 있음.
  - 요소에 다른 타입생성방지 : 동일한 타입의 배열구조.

➢ 커링

- ✓ 다항 함수가 인수를 전부 받을 때까지 실행을 보류(지연) 시켜 단계별로 나뉜 단항 함수의 순차열로 전환하는 기법
- ✓ 효과
  - 함수의 항수를 줄일 수 있다
  - 모듈성,재사용성을 높일 수 있다.

표현식인 문 & 표현식이 아닌 문

- **구별법**
  - ✓ **변수에 할당해 본다**
  - ✓ 표현식인 문은 값으로 평가되므로 변수에 할당 할 수 있다.
  - ✓ 표현식이 아닌 문은 값으로 평가할 수 없으므로 에러가 발생한다.