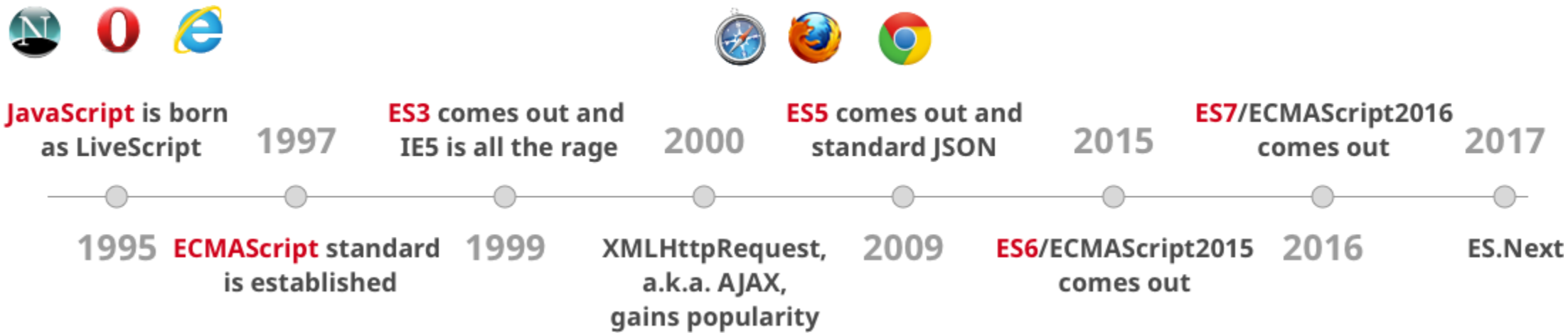
JavaScript Quick Guide

with Node.js

JavaScript
: A Brief History

- 넷스케이프가 클라이언트 측에서 사용할 목적으로 개발한 라이브스크립트에서 시작
- 1995년 JavaScript로 명칭을 변경 후 발표
- Microsoft JScript 등 브라우저 개발사별 스크립트의 난립으로 호환성에 문제 발생
- 1997년 넷스케이프가 JavaScript 스펙을 ECMA에 제출, 첫 버전 ECMA-262 발표
- 현재 대부분 브라우저가 ECMA-262를 기반으로 한 JavaScript 1.6, 1.7을 지원
- 2015년 ECMAScript Edition 6(ES6, ES2015) 발표



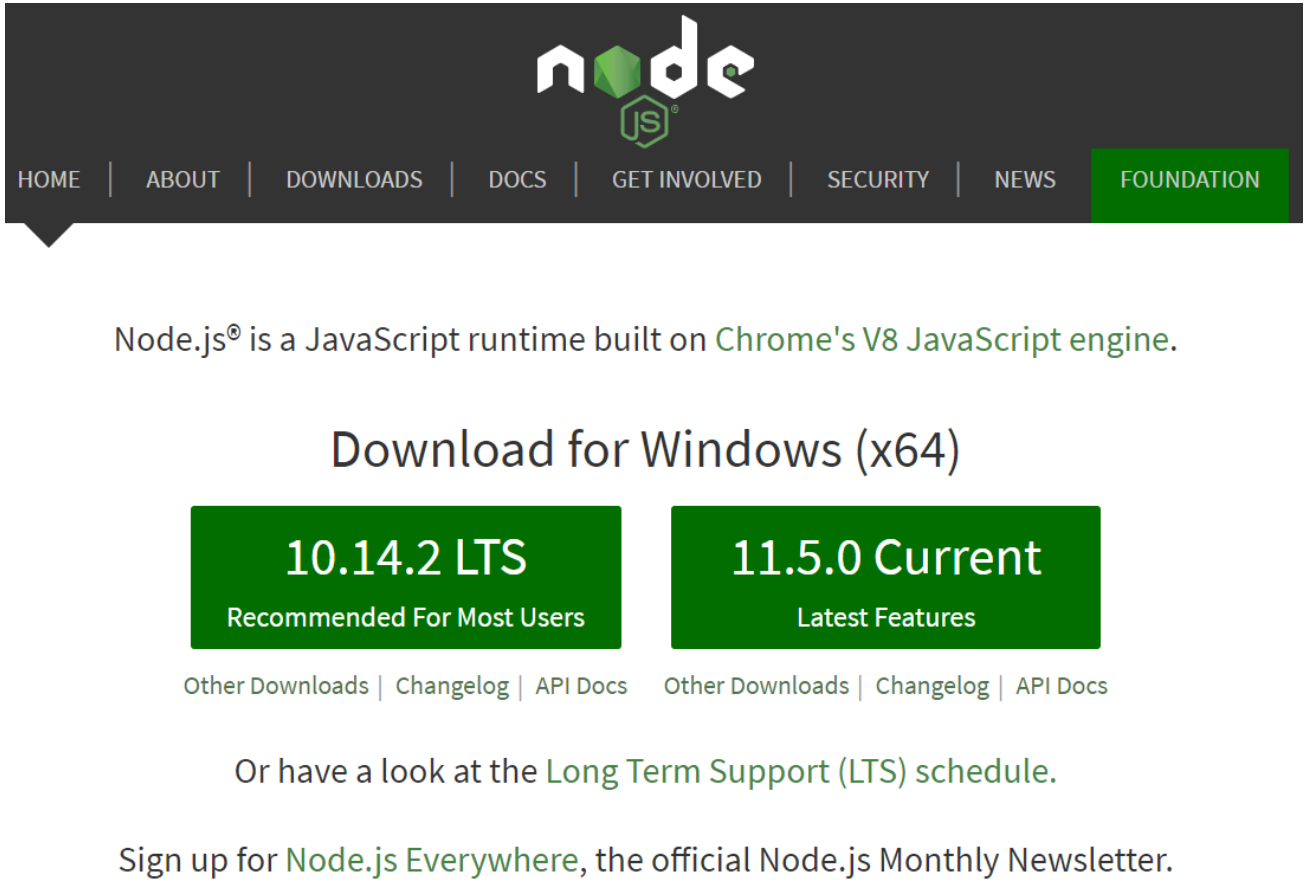
ECMAScript 2015

- JavaScript 출범 이후 최대의 변화
- JavaScript가 가지고 있던 단점들을 극복하고자 하는 노력
  - 블록 스코프 : let/const
  - 컬렉션 : Map, Set, WeakMap, WeakSet 등
  - class
  - 향상된 for loop
  - Promises
  - Template String
  - 모듈화

- 현실 세계의 문제
  - ~~실제 브라우저들이 ES2015를 지원하려면 많은 시간이 필요할 것~~

개발환경의 구축
: node.js 직접 설치

- node.js Homepage : https://nodejs.org/en/
  - 운영체제와 플랫폼에 맞는 버전 다운로드 후 설치
  - 버전의 구분 : LTS vs Current
    - LTS(Long Term Support)
      : 장기 지원을 약속하는 버전
    - Current
      : 가장 최신 버전
  - 버전의 구분 : 홀수 버전 vs 짝수 버전
    - 홀수 버전 : 개발 버전
    - 짝수 버전 : 안정화 버전



개발환경의 구축
: nvm을 이용한 설치

- nvm을 사용하면 필요에 따라 여러 버전의 node를 설치, 실행할 수 있다
  - Mac or Unix : http://nvm.sh
  - nvm windows : https://github.com/coreybutler/nvm-windows

- 설치 후 작업
  - 설치된 node 플랫폼 확인

```
C:\> nvm list
```

  - 설치 가능한 node 플랫폼 확인

```
C:\>nvm list available

|   CURRENT    |     LTS      |  OLD STABLE   | OLD UNSTABLE |
|--------------|--------------|---------------|--------------|
|    11.5.0    |   10.14.2    |   0.12.18     |   0.11.16    |
|    11.4.0    |   10.14.1    |   0.12.17     |   0.11.15    |
|    11.3.0    |   10.14.0    |   0.12.16     |   0.11.14    |
|    11.2.0    |   10.13.0    |   0.12.15     |   0.11.13    |
...
```

개발환경의 구축
: nvm을 이용한 설치

- 설치 후 작업
  - 원하는 node 플랫폼 설치

```
C:\> nvm install 10.14.2
...
C:\> nvm list
```

  - 사용하고자 하는 node 플랫폼 선택

```
C:\>nvm use 10.14.2
Now using node v10.14.2 (64-bit)
```

  - 설치 확인

```
C:\>nvm list

  * 10.14.2 (Currently using 64-bit executable)
```

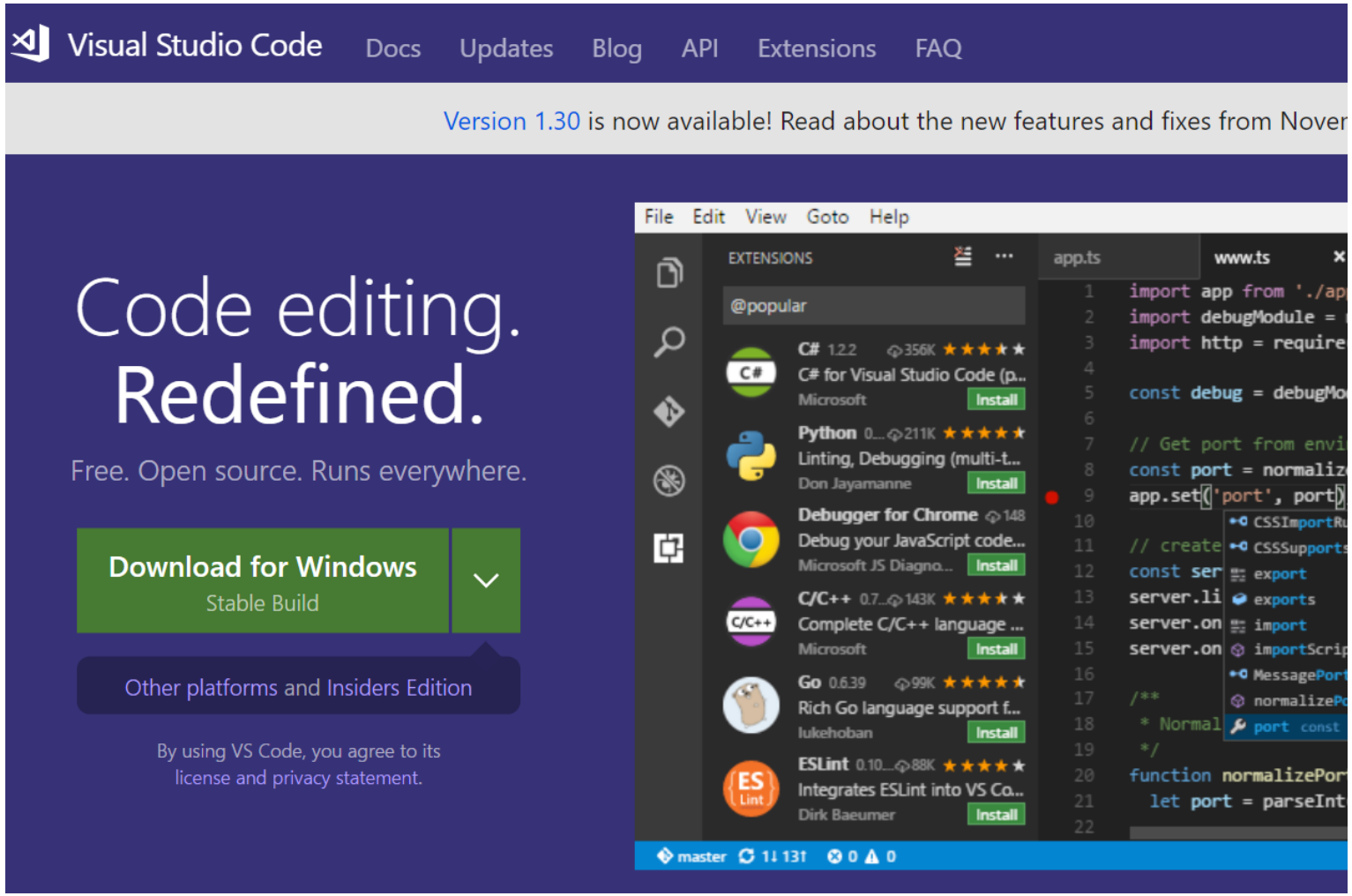
  - 현재 사용중인 node 버전 확인

```
C:\>node --version
v10.14.2
```

개발환경의 구축
: IDE 설치

- Visual Studio Code : https://code.visualstudio.com/

- 그 외 추천 IDE
  - Jetbrains WebStorm
  - Atom
  - Bracket 등



Node REPL

- REPL : Read Evaluate Print Loop
  - 명령 줄에서 직접 커맨드를 입력하고 코드를 테스트할 수 있는 커맨드 라인 기반 개발 환경

  - 커맨드라인에 node를 입력

```
C:\>node
>
```

  - 간단한 계산식을 입력해 봅니다

```
> 1 + 2 + 3 + 4 + 5
15
```

  - Node REPL 빠져 나오기

```
> process.exit()
```

첫 번째 JavaScript (node) 프로그램

- 다음의 소스코드를 입력하고 app.js (파일)로 저장

```
console.log("Hello, Node.js");
console.log(process.version, process.platform);
```

- 커맨드라인에서 다음 명령어를 입력 후 엔터

```
PS C:\myrepos\node-spt> node app.js
Hello, Node.js
v10.14.2 win32
```

- JavaScript 엔진은 Console에 출력하기 위한 console 객체를 전역 객체로 포함하고 있다

```
console.error("에러 메시지");
console.warn("경고 메시지");
console.debug("디버그 메시지");
console.log("일반 로그 메시지");
console.info("정보 메시지");
```

JavaScript 기본 문법

Syntax and Data Structure

JavaScript 구문 작성

- 세미콜론은 반드시 붙여야 하는 것은 아니다.

```
console.log("Hello")
console.log("Node.js");
```

  - 단, 두 문장 이상을 한 라인에 입력하고자 할 때는 세미콜론을 붙여 주어야 자바스크립트 엔진이 별도 문장임을 인지할 수 있다

- 주석은 한 줄 주석 (//), 여러 줄 주석 (/* ~ */) 모두 가능

```
// 한 줄 주석입니다.
/*
    여러 줄 주석입니다.
*/
```

- 주요 참고 자료
  - https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference
  - https://www.w3schools.com/js/default.asp

JavaScript 구문 작성

- Java와 마찬가지로 CamelCase가 기본

```
// CamelCase is norm
if (fooBar == bazBat) {
    console.log(fooBar, "is", bazBat);
}
```

- 객체의 속성, 메서드에 접근할 때는 . 으로 접근

```
someObject.someMethod();
```

- JavaScript는 객체 기반 언어(Object-Base Language)로 객체는 프로퍼티(Property: 데이터)와 메서드(Method: 실행 명령)으로 이루어져 있다.

JavaScript 변수와 상수
: var, let vs const

- let : mutable (변경 가능)
  - 변수에 재 할당이 가능
- const : immutable (변경 불가)
  - 변수에 재 할당이 불가 -> 상수로 사용 가능
  - 재 할당이 불가하므로 선언과 할당을 함께 해주어야 한다
- var : ES6 이전, JavaScript 에서 사용하던 변수 할당을 위한 키워드
  - ES6 이상의 환경에서는 var 사용을 지양하고 let과 const를 활용할 것

```
var testVar = "var";
let testLet = "let";
const testConst = "const";

testVar = "var changed";
testLet = "let changed";
testConst = "const changed"; // -> Error: Cannot Change
```

JavaScript 변수와 데이터타입

- 변수 이름 명명 규칙
  - 알파벳(A-Z, a-z), 숫자(0-9), 밑줄(_), 달러($) 기호를 사용할 수 있음
  - 숫자로 시작해서는 안됨

- 변수의 데이터타입은 선언시 지정하는 것이 아니라
  할당과 함께 결정된다
  - 변수에 담긴 데이터의 타입을 확인하고자 하면 typeof 키워드를 이용한다

```
let v = "This is String";
console.log("v is", typeof v);
v = 2018;
console.log("v is", typeof v);
```

- JavaScript의 기본 데이터타입은
  - Number, String, Boolean

Number in JavaScript

- 숫자를 표현하거나 산술 연산을 하는 데 사용되는 데이터타입
  - 기본적으로는 산술 연산(+, -, *, / 등)이 가능
  - Math 내장 객체를 이용하여 수학 함수를 이용한 결과값을 받을 때 활용

```
let n1 = 5; // Literal
let n2 = Number(5); // Number 객체
console.log(typeof n1, typeof n2);

console.log(Math.round(Math.PI));
console.log(Math.min(7, 3, 5, 2, 9, 13), Math.max(7, 3, 5, 2, 9, 13));
console.log(Math.round(3.578), Math.floor(3.578));
```

  - Math.round() : 반올림, Math.floor() : 버림

Number in JavaScript
: Casting

- 자바스크립트는 변수 선언시 데이터타입을 지정하지 않음
  - 문자열 데이터로 숫자를 표현하는 경우(특히 외부 파라미터를 이용한 전송)가 많음 -> 데이터 형 변환이 필요한 경우 많음
  - parseInt(), parseFloat()를 이용하면 Number 형으로 변환 가능

```
console.log(parseInt("011")); // 11
console.log(parseInt("011", 2)); // 3: 진법 변환
console.log(parseInt("123.456")); // 123: 정수 형태만 반환한다
console.log(parseFloat("123.456")); // 123.456
```

  - 숫자로 변환할 수 없는 형태의 문자열을 캐스팅할 경우, NaN(Not a Number)를 반환

```
console.log(parseFloat("a123.456")); // NaN
```

  - 수치형 데이터가 NaN인지 확인하려면 isNaN() 내장 객체 함수를 이용

```
let v = parseInt("abcde");
console.log(typeof v, isNaN(v));
```

Number in JavaScript
: Infinity

- 산술 연산 결과가 무한대임을 표시하는 자바스크립트의 표현식
  - 양의 무한대(Infinity)와 음의 무한대(-Infinity)로 구분

```
console.log(1 / 0); // Infinity
console.log(-1 / 0); // -Infinity
```

  - 산술 결과 혹은 전달된 인자값이 유한한 형태를 가진 것인지 검증하기 위해서는 `isFinite()` 내장 객체 함수를 활용
    - 인자 값이 숫자이면서 유한하면 true
    - 인자 값이 `NaN, Infinity, -Infinity`면 false

```
console.log(isFinite(1/0)); // false
console.log(isFinite(2018)); // true
console.log(isFinite(NaN)); // false
```

String in JavaScript

- String : 문자열을 표현하는데 사용되는 데이터 타입
  - JavaScript의 문자열은 유니코드 문자들의 연결 구조이기도 하다
    = 문자 하나하나가 연결되어 하나의 표현을 이루는 데이터

```
let s1 = "JavaScript";
let s2 = String("JavaScript");

console.log(typeof s1, typeof s2);
```

  - Property : length

```
console.log("Hello".length);
```

  - 자동 형변환 : JavaScript에서는 String과 다른 타입들을 합칠 때, String으로 자동 변환된다.

String in JavaScript

- String 문자열 추출
  - charAt(index) : 문자열 내 특정 인덱스의 문자를 추출
  - substr(from, length) : 문자열 내의 from 인덱스부터 lengh 길이만큼의 문자열을 추출
  - substring(from, end) : 문자열 내의 from 인덱스부터 end 인덱스까지의 문자열을 추출

```
const s = "Modern JavaScript";

console.log(s.charAt(7)); // J
console.log(s.substr(7, 10)); // JavaScript
console.log(s.substring(7, 17)); // JavaScript
```

String in JavaScript

- String 문자열 검색
  - indexOf(searchString[, position]) : 문자열 내 position 인덱스로부터 searchString을 검색, 해당 인덱스를 반환 (position 파라미터는 옵션)
  - 검색하는 searchString 이 문자열 내에 없을 때는 -1을 반환
  - 뒤로부터 검색시에는 lastIndexOf() 메서드를 사용

```
const s = "Modern JavaScript";

console.log(s.indexOf("Java")); // 7
console.log(s.indexOf("java")); // -1
console.log(s.lastIndexOf("Script")); // 11
```

String in JavaScript

- String 문자열 치환
  - replace(searchValue, replaceValue) : 문자열 내 searchValue를 replaceValue로 치환

```
const s = "Modern JavaScript";

console.log(s.replace("JavaScript", "JS")); // Modern JS
```

- String 화이트 스페이스 제거
  - trim() : 문자열의 시작과 끝의 빈 문자 제거

```
const s = "      Need to be trimmed        ";
console.log(s.trim());
```

String in JavaScript

- TODO: Template String 설명 추가

Boolean과 ==, ===

- Boolean : 논리값 true / false를 다루기 위한 데이터 타입
  - Boolean()을 이용하여 논리 검증을 수행할 수 있다

```
console.log(Boolean("JavaScript")); // true
console.log(Boolean("")); // false
```

- ==와 ===
  - == : 타입과 상관 없이 값을 비교 (Equal Operator)
  - === : 값과 함께 타입도 함께 비교 (Strict Equal Operator)

```
console.log(123 == "123"); // true
console.log(123 === "123"); // false
```

null과 undefined

- JavaScript는 값이 없음을 나타내는 null과 초기화(선언)되지 않은 경우를 나타내는 undefined라는 특별한 데이터 타입이 있음
  - null : 개발자가 의도적으로 빈 값을 할당한 경우
  - undefined : 할당 자체가 이루어지지 않은 것

```
let v1, v2 = null;
console.log(v1 == v2); // true
console.log(v1 === v2); // false

console.log(typeof v1, typeof v2); // undefined object
```

조건문
: if ~ else if ~ else

- 타 언어와 마찬가지로 조건 분기를 위한 if ~ else if ~ else 블록을 사용할 수 있다
  - 문법 구조는 C 계열의 언어 조건문과 유사

```
let num = 3;
if (num > 0) {
    console.log("양수입니다.");
} else if (num < 0) {
    console.log("음수입니다.");
} else {
    console.log("0입니다.");
}
```

조건문
: switch

- 타 언어와 마찬가지로 조건 분기를 위한 switch 문을 사용할 수 있다

```
let grade = "C";
switch (grade) {
    case "A":
    case "B":
    case "C":
    case "D":
        console.log("Pass");
        break;
    case "F":
        console.log("Fail");
        break
    default:
        console.log("?");
}
```

반복문

- 반복을 위한 for, while, do ~ while 문도 C 계열 언어와 거의 유사

```
for (let i = 0; i < 10; i++) {
    if (i % 2 == 0) console.log(i);
}
```

- 반복문 연습
  - [연습 1] 2단 ~ 9단까지 구구단을 출력해 봅시다
  - [연습 2] 아래와 같이 출력하는 프로그램을 만들어 봅시다

*****

****

***

**

*

반복문
: for ... in, for ... of

- for ... in: 객체들의 속성(property)을 반복하여 탐색
  - JavaScript의 모든 객체에서 사용 가능

```
let obj = {
    name: "홍길동",
    age: 28,
    job: "도적"
};

for (let key in obj) {
    console.log(key, "->", obj[key]);
}
```

함수

- 특정 실행 코드의 묶음을 프로그램 상의 다른 영역에서도 재호출 할 수 있도록 그룹화한 기능 객체

- 함수의 선언 : 함수 선언식(Function Declaration)

```
function sum(a, b) { // 선언
    return a + b;
}
console.log(sum(3, 7)); // 사용
```

- 함수도 객체로 간주함 : 함수 표현식(Function Expression)

```
const sum = function(a, b) {
    return a + b;
}
console.log(typeof sum, sum(3, 7)); // function 10
```

함수

- 함수로 전달된 모든 매개변수는 함수 내부에서 arguments 객체로 참조할 수 있다
  - 가변 인수처럼 활용할 수 있음

```
function getNumberTotal() {
    let result = 0;
    for (let i = 0; i < arguments.length; i++) {
        if (typeof arguments[i] == "number") {
            result += arguments[i];
        }
    }
    return result;
}

console.log(getNumberTotal(1, 2, 3, 4));
console.log(getNumberTotal(1, 2, "3", 4));
```

함수

- JavaScript의 함수는 전혀 별개의 문법적 기능이 아니라 Number, String 등과 동등한 객체이다.
  - 함수에 Number, String 등을 전달할 수 있는 것처럼 매개변수로 함수 자체도 전달할 수 있다

```
function sandBox(val1, val2, func) {
    if (typeof func == "function")
        func(val1, val2);
}

sandBox(3, 4, function(v1, v2) {
    console.log(v1 + v2);
});
```

  - 다른 코드의 인수로 넘겨주는 실행 가능한 코드를 콜백(Callback)이라 함

JavaScript의 객체

- 객체 : 정보를 관리하기 위해 의미를 부여하고 분류하는 논리적 단위
  - 객체는 속성(attributes)을 가지고 있으며 속성은 다음과 같이 분류된다
    - Property : 객체가 관리하는 정보
    - Method : 객체가 수행할 수 있는 기능 (객체가 가지고 있는 함수)
  - 객체의 생성

```
const person = new Object();
person.name = "홍길동";
person.job = "도적";
person.showInfo = function() {
    console.log("Name: " + this.name);
    console.log("Job: " + this.job);
}
person.showInfo();
```

JavaScript의 객체
: JSON을 이용한 객체의 생성

- JSON(JavaScript Object Notation)
  - JavaScript에서 객체를 표기하기 위한 표기법
  - 어떤 객체이던 표기할 수 있고 바로 생성 가능

KEY:VALUE 의 쌍으로 어트리뷰트 구성
콤마(,) 사용에 유의

```
const person = {
    name: "홍길동",
    job: "도적",
    showInfo: function() {
        console.log("Name:" + this.name);
        console.log("Job:" + this.job);
    }
}

person.showInfo();
```

JavaScript의 객체
: prototype 기반 상속

- JavaScript는 객체 지향이 아니라 객체 기반 언어
  - prototype 기반의 상속을 이용, 객체지향의 특성을 구현할 수 있다(속성 공유)

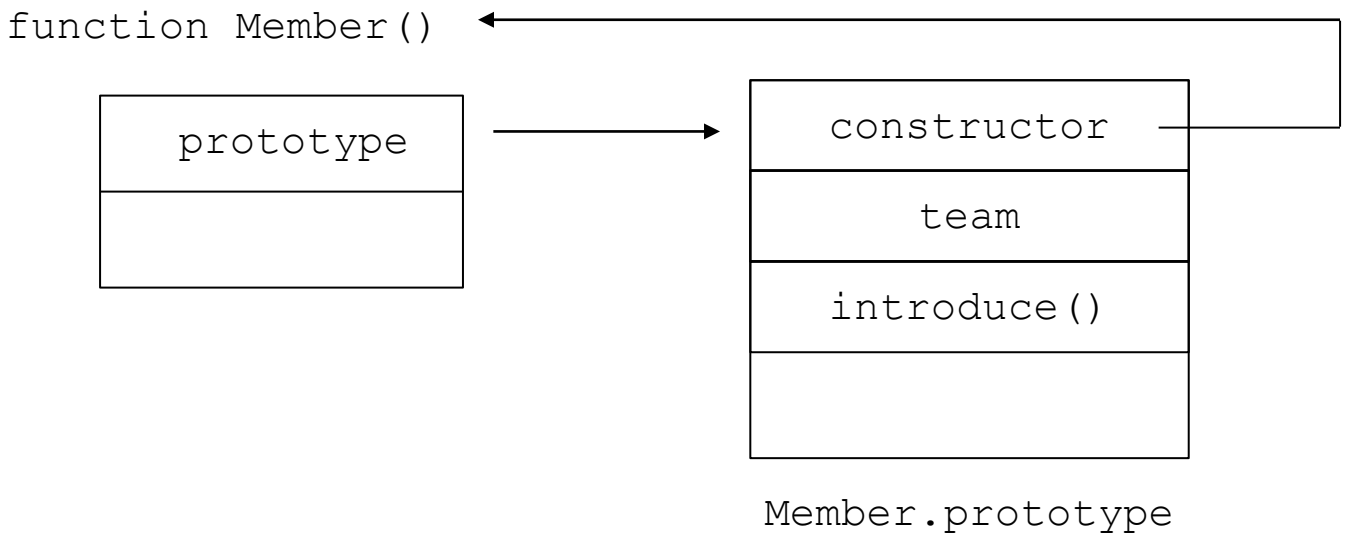
```
const Member = function(name, position) {
    this.name = name;
    this.position = position;
};
Member.prototype.team = "상북";
Member.prototype.introduce = function() {
    console.log("Name:" + this.name);
    console.log("Position:" + this.position);
}

const m1 = new Member("강백호", "PF");
const m2 = new Member("서태웅", "SF");
m1.introduce();
m2.introduce();
```

JavaScript의 객체
: prototype 기반 상속

- 객체의 constructor로 어떤 객체가 사용되고 있는지 확인해 봅시다

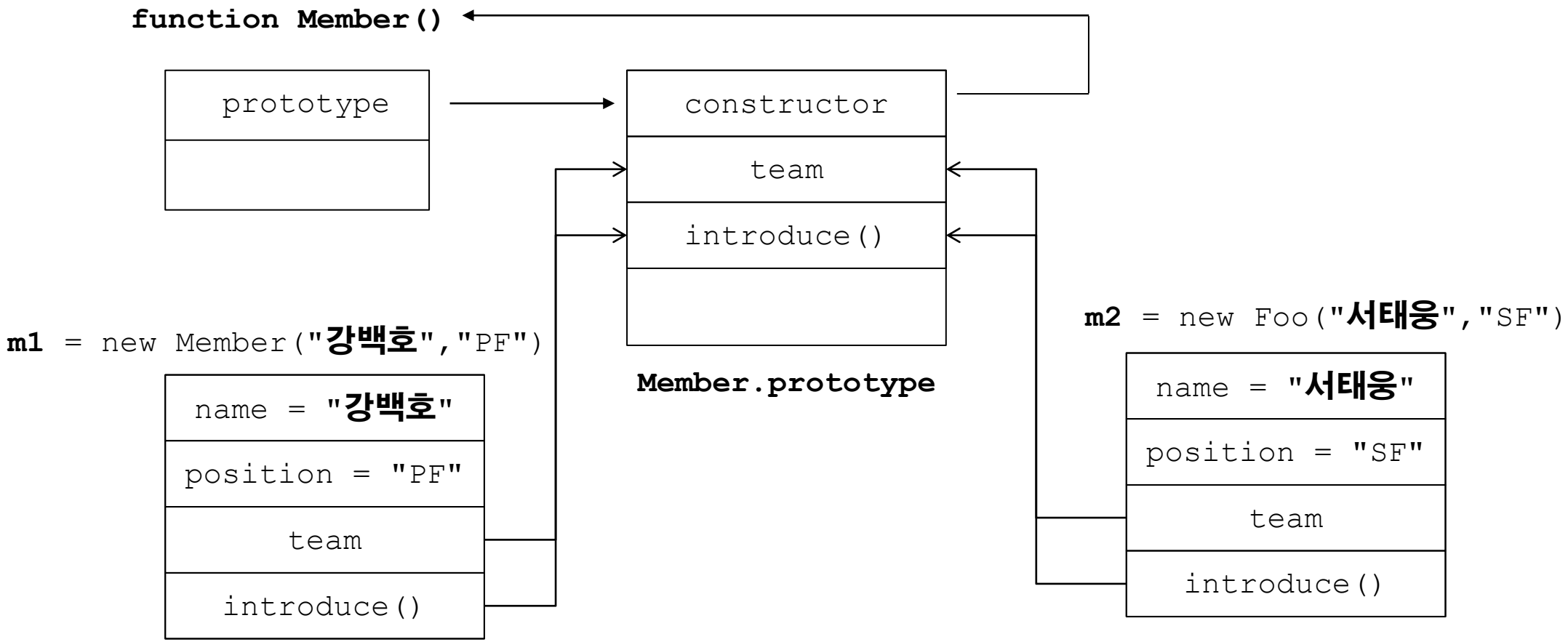
```
console.log(m1.constructor);
console.log(Member.prototype);
```

- prototype 기반 상속



JavaScript의 객체
: prototype 기반 상속

- prototype 기반 상속



Binding
: call, apply, bind

- JavaScript의 this
  - 자바스크립트 코드의 실행 컨텍스트(Execution Context) 자체를 의미
  - 실행 상황에 따라 this가 가리키는 객체는 바뀔 수 있다.

  - 함수가 호출되었을 때 그 함수가 속해 있던 객체의 참조

```
const testFunc = function(location) {
    console.log(this === global);
    console.log(`나는 ${location}에 사는 ${this.name} 입니다.`);
}
```

    - 이 함수를 실행했을 때, this.name은 무엇입니까?
    - 실행 상황과 무관하게 참조하는 this를 선택할 수는 없을까?

Binding
: call, apply, bind

- apply, call
  - this와 함수의 인수를 사용하여 함수를 실행하는 메서드

```
const obj = { name: "둘리" };
```

  - 다음과 같이 호출해 보고, 출력 값의 변화와 참조하는 this를 확인해 봅시다.

```
testFunc.call(obj, "서울");
testFunc.apply(obj, ["서울"]);
```

  - call과 apply의 동작은 본질적으로 같음
    - 차이점: 함수에 인수를 넘길 때, apply는 배열로, call의 인수는 쉼표로 구분한 값의 목록을 전달함

- bind

```
const boundTestFunc = testFunc.bind(obj);
boundTestFunc("서울");
```

JavaScript의 객체
: 내장 객체 Array

- Array : 하나의 변수에 여러 개의 값을 저장하기 위한 JavaScript의 내장 객체
  - JavaScript의 Array는 인덱스의 범위를 엄격하게 체크하지 않으며 C 계열의 배열에 비해 다양한 방식으로 작동
  - 배열의 생성 : 배열 객체를 이용한 생성

```
const v1 = new Array(10);
const v2 = new Array();
const v3 = new Array(1, "ABC", true);

console.log(v1.length, v2.length, v3.length); // 10 0 3
```

  - 배열의 생성 : 리터럴

```
const v4 = [];
const colors = ["red", "green", "blue", "yellow"];
```

JavaScript의 객체
: 내장 객체 Array

- Array와 객체의 관계
  - 객체의 속성에 접근할 때, 다음과 같이 배열처럼 접근할 수 있음

```
const person = {
    name: "홍길동",
    job: "도적"
};

console.log(person['name'] + ":" + person['job']);
```

  - JavaScript의 Array는 배열의 인덱스를 엄격하게 체크하지 않는다

```
const arr = [];
console.log(arr.length);
arr[10] = 5;
console.log(arr.length);
```

JavaScript의 객체
: 내장 객체 Array

- Array 주요 메서드

메서드	설명
concat(array1, ...)	여러 배열을 하나로 연결
join(str)	배열 내의 객체들을 str 구분자를 가지는 하나의 문자열로 변환
pop()	배열의 맨 마지막 객체를 추출(추출된 객체는 삭제)
push(item1, ...)	배열의 맨 마지막에 객체들을 추가
reverse()	배열의 순서를 뒤집음
shift()	배열의 맨 처음 객체를 추출(추출된 객체는 삭제)
slice()	배열의 일부분만을 추출하여 새 배열을 만듦
sort(function)	배열을 정렬
splice(start[, count])	원하는 index 위치의 요소를 삭제하거나 추가

JavaScript의 객체
: 내장 객체 Array

- Array 주요 메서드 : concat

```
const veges = ['배추', '무', '쪽파'];
const sources = ['소금', '고춧가루', '새우젓'];
const ingredients = veges.concat(sources);

console.log(ingredients);
```

- Array 주요 메서드 : join

```
const veges = ['배추', '무', '쪽파'];
const sources = ['소금', '고춧가루', '새우젓'];
const ingredients = veges.concat(sources);

console.log("김장 재료: " + ingredients.join(", "));
```

JavaScript의 객체
: 내장 객체 Array

- Array 주요 메서드 : push 와 pop

```
let fruits = ["Banana", "Orange", "Apple"];
fruits.push("Kiwi");
console.log(fruits);
console.log(fruits.pop());
console.log(fruits.pop());
console.log(fruits);
```

Stack 자료형처럼 사용 가능

- Array 주요 메서드 : shift

```
let fruits = ["Banana", "Orange", "Apple"];
fruits.push("Kiwi");
console.log(fruits);
console.log(fruits.shift());
console.log(fruits.shift());
console.log(fruits);
```

Queue 자료형처럼 사용 가능

JavaScript의 객체
: 내장 객체 Array

- Array 주요 메서드 : splice
  - 원하는 위치의 요소를 삭제하거나 추가하는 메서드
  - 인수가 한 개일 경우 : 해당 인덱스부터 끝까지 요소를 반환 후 제거

```
fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.splice(2));
console.log(fruits);
```

  - 인수가 두 개일 경우 : 첫 번째 인수 인덱스부터 두 번째 인수 개수만큼 반환 후 제거

```
fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.splice(2, 1)); // 시작 인덱스, 개수
console.log(fruits);
```

  - 인수가 세 개 이상 : 첫 번째 인수 인덱스부터 두 번째 인수 개수만큼 반환 후 제거,
    제거된 위치에 세 번째 이후 인수를 새로운 요소로 추가

JavaScript의 객체
: 내장 객체 Array

- Array 주요 메서드 : reverse, slice, sort

```
let fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();
console.log(fruits);

let slices = fruits.slice(1, 2);
console.log(slices);

fruits.sort();
console.log(fruits);
```

JavaScript의 객체
: 내장 객체 Array

- String의 split 메서드는 특정 구분자를 기준으로 문자열을 분리, Array로 반환한다

```
const str = "JavaScript is something strange than other languages";
let chunks = str.split(" ");

for (let i = 0; i < chunks.length; i++) {
    console.log("Chunk", i, ":" + chunks[i]);
}
```

JavaScript의 객체
: 내장 객체 Date

- Date 객체는 날짜와 시간을 다루는 객체

```
let now = new Date(); // 현재 시간

console.log(now);
```

- Date 객체를 생성하는 다양한 생성자
  - Date(year, month, day) // month는 0부터 시작하니 주의하자
  - Date(yyyy, mm, dd, hh, mi, ss)
  - Date(milliseconds)

  - 각 생성자로 Date 객체를 생성하고 console.log 메서드로 결과를 확인해 봅시다

JavaScript의 객체
: 내장 객체 Date

- Date 내장 메서드

메서드	설명
getYear() / setYear()	년도(기준 1900년)
getMonth() / setMonth()	월 (0=1월 ~ 11=12월)
getDate() / setDate()	일 (1 ~ 31)
getDay() / setDay()	요일(0=일요일 ~ 6=토요일)
getHours() / setHours()	시간(0~23)
getMinutes() / setMinutes()	분(0~59)
getSeconds() / setSeconds()	초(0~59)

JavaScript의 객체
: 내장 객체 Date

- Date 내장 메서드 사용 예

```
let d = new Date(2012, 8, 24); // 2012년 9월 24일
console.log(d);
console.log("년도:", d.getYear() + 1900);
console.log("년도:", d.getFullYear());
console.log("월:", d.getMonth() + 1);
console.log("일:", d.getDate());
console.log("요일:", d.getDay());

d.setYear(2018);
console.log(d);
```

EcmaScript 6

보충수업

전개 연산자

- 나열형 자료를 추출하거나 연결할 때 사용
  - 사용법: 배열이나 객체, 변수명 앞에 ...를 입력
  - 제약: 배열, 객체, 함수 인자 표현식([], {}, ()) 안에서만 활용 가능

- 배열의 연결

```
const arr1 = ['홍길동', '장길산'];
const arr2 = ['임꺽정', '전우치'];

const combined = [...arr1, ...arr2];
console.log("ES6 combine:", combined);
```

전개 연산자

```
arr1: [ '홍길동', '장길산' ]
```

- 배열의 전개 1

```
var [ first, second, third = "empty", ...others ] = arr1;
console.log(first, second, third, others);
// "홍길동" "장길산" empty []
```

```
combined: [ '홍길동', '장길산', '임꺽정', '전우치' ]
```

- 배열의 전개 2

```
var [ first, second, third = "empty", ...others ] = combined;
console.log(first, second, third, others);
// 홍길동 장길산 임꺽정 [ '전우치' ]
```

전개 연산자

- 객체 전개 연산
  - 객체의 키나 값을 추출할 때 활용 가능

```
var obj1 = { one: 1, two: 2, other: 0 };
var obj2 = { three: 3, four: 4, other: -1};

var combined = {
    ...obj1,    //  obj1의 모든 속성을 전개
    ...obj2,    //  obj2의 모든 속성을 전개
}
console.log(combined);

// { one: 1, two: 2, other: -1, three: 3, four: 4 }
```

전개 연산자

- 객체 전개 연산
  - 객체 전개

```
var { other, ...others } = combined;
console.log(other, others);
// 0 { three: 3, four: 4, one: 1, two: 2 }
```

Class

- 기존 자바스크립트는 클래스 표현 식 없이 prototype으로 클래스를 다룸
  - ES6에서는 클래스 정의 문법으로 클래스를 정의, 사용 가능
  - constructor: 생성자
  - static : 정적 멤버

```
class Shape {
    static create(x, y) {
       return new Shape(x, y);
    }
    constructor(x, y) {
        this.name = "Shape";
        this.move(x, y);
    }
    move(x, y) {
        this.x = x;
        this.y = y;
    }
    area() {
        return 0;
    }
}

var s = new Shape(0, 0);
console.log(s, s.area());
```

Class
: 상속

- 기존 자바스크립트는 클래스 표현 식 없이 prototype으로 클래스를 다룸
  - ES6에서는 클래스 정의 문법으로 클래스를 정의, 사용 가능
  - constructor: 생성자
  - static : 정적 멤버
  - this : 객체 자신을 가리킴
  - super : 부모객체를 가리킴

```
class Circle extends Shape {
    constructor(x, y, radius) {
        super(x, y); // 부모 생성자
        this.radius = radius;
    }
    area() {
        if (this.radius === 0)
            return super.area();
        return this.radius ** 2 * Math.PI;
    }
}

var c = new Circle(0, 0, 3);
console.log(c, c.area());
```

Array

- 배열은 JavaScript에서 가장 활용도가 높은 범용 객체
  - ES6 이전부터 많은 메서드들이 추가되고 확장됨

- forEach : 배열의 개별 요소들을 추출하여 콜백 함수로 넘겨준다

```
var source = ["Banana", "Orange", "Apple", "Mango"];

source.forEach(item => {
    console.log(item);
});
```

Array

- every, some
  - 특정 조건을 만족하는지 배열 내부 원소를 순회하면서 검사
  - 조건에 만족하면 true, 아니면 false
  - 배열 내부 원서의 값에 대해 검토가 필요한 경우 사용

- every : 배열 내부의 모든 원소가 조건을 만족하면 true
- some: 배열 내부의 요소 중 일부가 조건을 만족하면 true

```
let data = [
        { name: "홍길동", age: 28 },
        { name: "장길산", age: 35 },
        { name: "전우치", age: 25 },
    ];
```

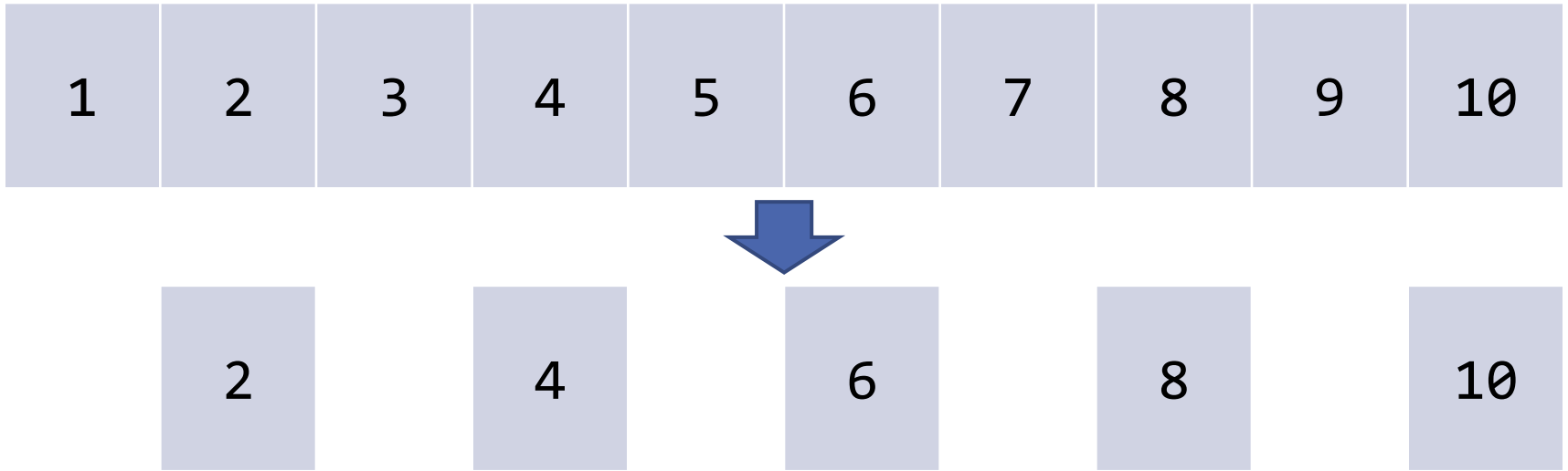
```
let result = data.some(x => {
    return x.age > 25;
});
```

```
let result = data.any(x => {
    return x.age > 25;
});
```

Array

- filter : 배열의 요소들을 특정 조건을 기준으로 필터링하여 새로운 배열로 반환
  - 배열에서 필요한 것만 남김

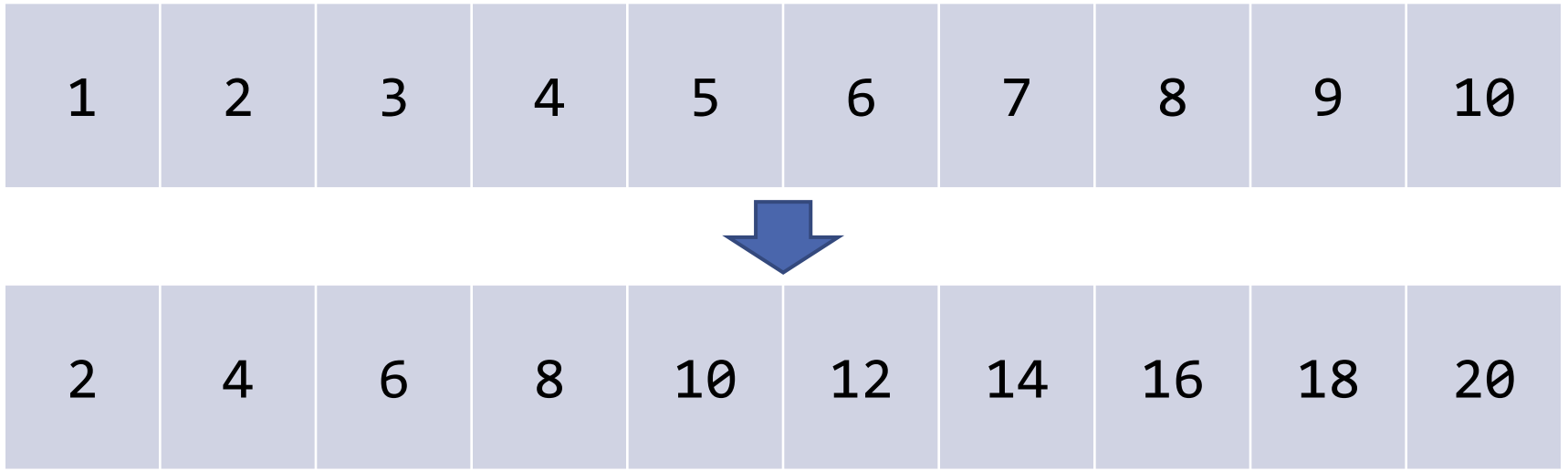
```
var source = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
var result = source.filter( item => item % 2 == 0 );
```



Array

- map : 배열의 요소들을 콜백함수에 전달하여 새로운 배열을 만듦
  - 배열의 각 요소를 변형하는 역할을 수행

```
var source = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
var multiply = source.map(item => item * 2);
```



Array

- reduce : map이 배열의 각 요소를 변형한다면, reduce는 배열 자체를 변형함
  - 일반적으로 배열을 값 하나로 줄인다
  - Basic Syntax

```
// Basic Syntax
arr.reduce(callback[, initialValue]);
```

  - callback
    - previousValue : 마지막 콜백에서 반환된 값(누계) or initialValue
    - currentValue : 현재 처리되고 있는 배열의 요소
    - currentIndex : 현재 처리되고 있는 요소의 인덱스
    - array : reduce 호출에 사용되는 원 배열

Array

- reduce : 일반적인 사용 패턴

```
var arr = [12, 4, 19, 33, 86];
var sum = arr.reduce( (acc, value, idx, arr) => {
    console.log(`누산 값은 ${acc}`);
    console.log(`${arr}의 ${idx}번째 요소는 ${value}`);
    return acc + value;
}, 0);

console.log(sum);
```

  - map과 filter과 같은 함수형 메서드를 모두 reduce로 모두 구현할 수 있음

Array

- reduce : map을 대체하는 사용 패턴

```
var arr = [12, 4, 19, 33, 86];
var sum = arr.reduce( (acc, value, idx, arr) => {
    console.log(`누산 값은 ${acc}`);
    console.log(`${arr}의 ${idx}번째 요소는 ${value}`);
    return acc + value;
}, 0);

console.log(sum);
```

  - 반복되는 모든 것에는 reduce를 쓸 수 있다는 점을 기억하자!

비동기(Asynchronous)

- JavaScript 엔진은 Single Thread -> 두 가지 작업을 동시에 하지 못함
- 동기 vs 비동기
  - 동기적 처리(Synchronous) : 작업을 요청함과 동시에 작업의 결과를 그 자리에서 받음
  - 비동기적 처리(Asynchronous) : 작업을 요청하지만 결과는 그 자리에서 꼭 받지 않아도 되는 처리 방식
- "비동기 처리"
  - 비동기로 일어나는 일을 동기적으로 처리하는 방법

비동기(Asynchronous)

- Callback 사용하기

```
console.log("begin logicA");

setTimeout(() => {
    console.log("callbackA called");
    console.log("begin logicB")
    setTimeout(() => {
        console.log("callbackB called");
    }, 2000);
    console.log("end logicB");
}, 2000);
console.log("end logicA");
```

  - 여러 비동기 로직이 의존 관계에 있을 때,
    콜백 지옥(Callback Hell)에 빠질 우려가 있다

Callback Hell!

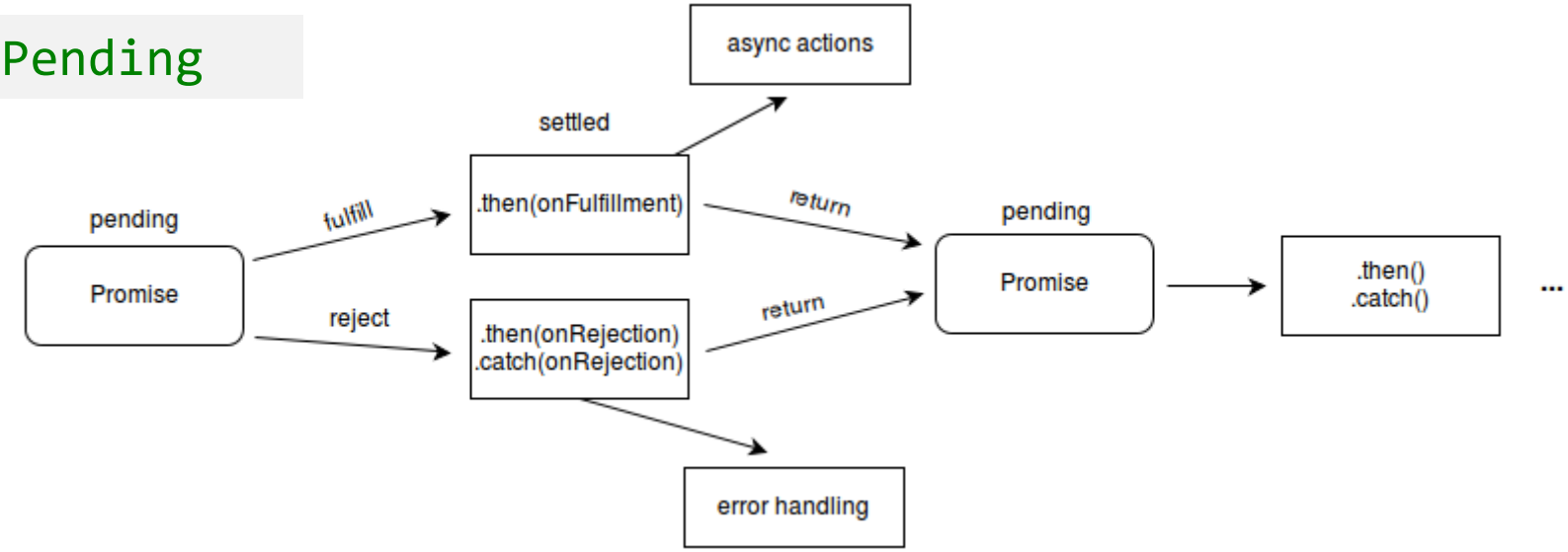
```
foo(() => {
    bar(() => {
        baz(() => {
            qux(() => {
                quux(() => {
                    quuz(() => {
                        corge(() => {
                            grault(() => {
                                run();
                            }).bind(this);
                        }).bind(this);
                    }).bind(this);
                }).bind(this);
            }).bind(this);
        }).bind(this);
    }).bind(this);
}).bind(this);
```

비동기(Asynchronous)
: Promise

- Promise의 세 가지 상태
  - Pending(대기) : 비동기 처리 로직이 아직 완료되지 않을 상태
  - Fulfilled(이행) : 비동기 처리가 완료되어 프로미스가 결과 값을 반환해준 상태
  - Rejected(실패) : 비동기 처리가 실패하거나 오류가 발생한 상태

- Pending(대기)

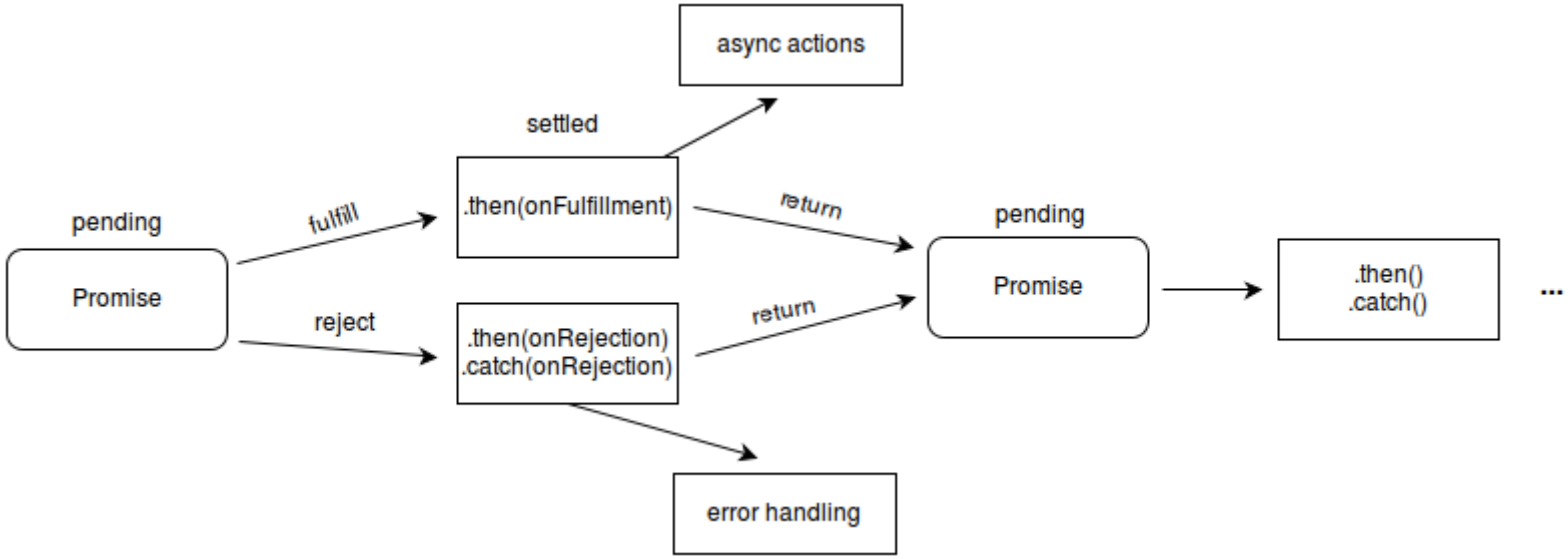
```
new Promise();  //  Pending
```



비동기(Asynchronous)
: Promise

- Promise를 생성할 때, 콜백함수(실행자:Executor)를 선언할 수 있음
  - 콜백 함수의 인자는
    resolve, reject

```
new Promise((resolve, reject) => {
    // 비동기 처리 로직: 실행자(executor)
});
```

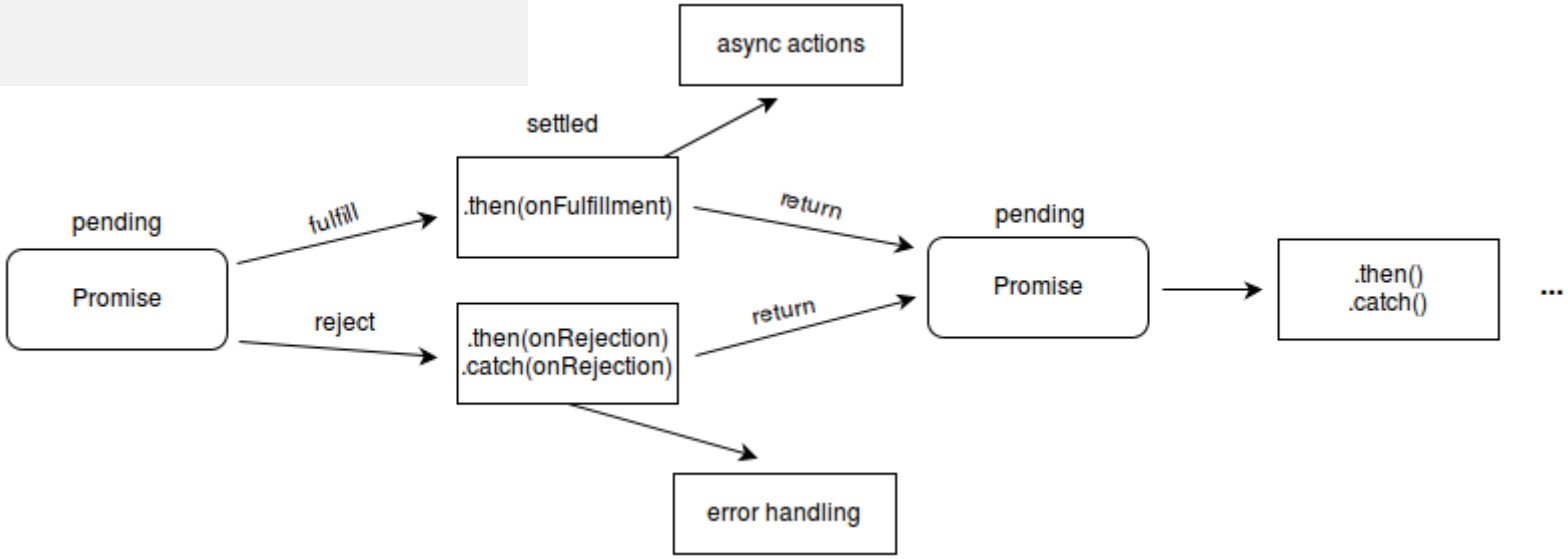


비동기(Asynchronous)
: Promise

- 콜백으로 전달된 두 개의 인자로 이행(Fulfilled), 실패(rejected) 단계로 전달할 수 있음

```
new Promise((resolve, reject) => {
    // 성공했을 때
    resolve({value});
    // 실패했을 때
    reject({reason});
});
```

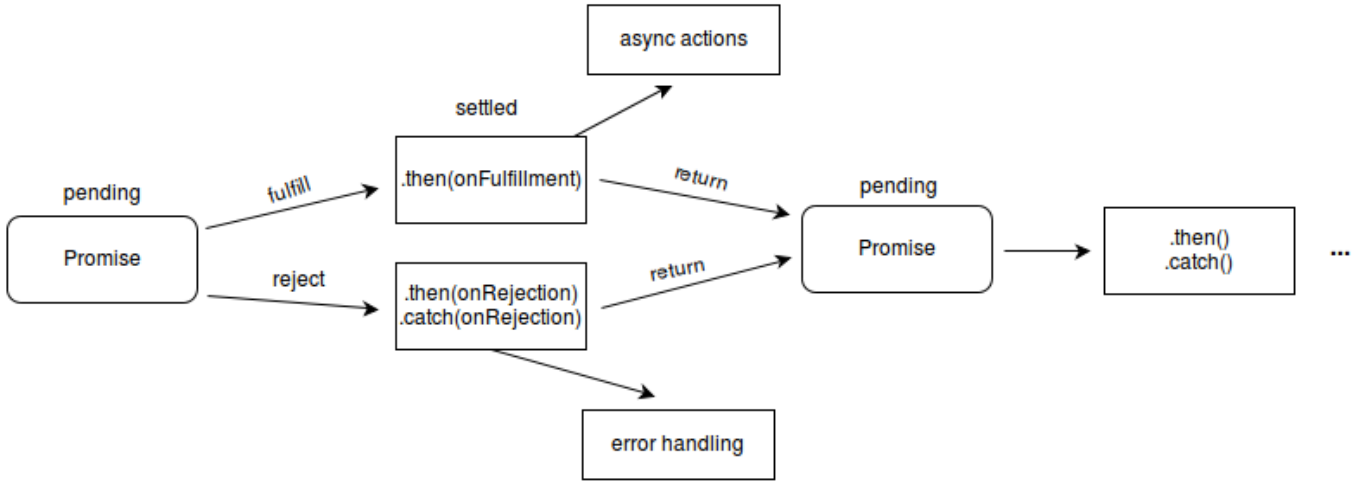
- Executor는 resolve나
- reject 중 하나를 반드시 호출해야 한다



비동기(Asynchronous)
: Promise

- resolve로 전달된 데이터는 then으로,
  reject로 전달된 실패 사유는 catch 블록의 콜백으로 전달됨

```
new Promise((resolve, reject) => {
    // 성공했을 때
    resolve({value});
    // 실패했을 때
    reject({reason});
}).then(value => {
    // 성공했을 때의 처리
}).catch(reason => {
    // 실패했을 때의 처리
});
```



비동기
: async, await

- async/await 키워드는 프라미스를 좀 더 편하게 사용할 수 있게 해 줌

- function 앞에 async를 붙이면 해당 함수는 항상 프라미스를 반환

```
async function f() {
    console.log("async function");
    return 1;
}

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