Javascript 2

this

Java에서의 this는 인스턴스 자신을 가리키는 참조변수이다. this가 객체 자신에 대한 참조 값을 가지고 있다는 뜻이다. 주로 매개변수와 객체 자신이 가지고 있는 멤버 변수명이 같을 경우 이를 구분하기 위해서 사용된다. 하지만 자바스크립트의 경우 Java와 같이 this에 바인딩 되는 객체는 한가지가 아니라 해당 함수 호출 방식에 따라 this에 바인딩 되는 객체가 달라진다.

함수 호출 방식과 this 바인딩

자바스크립트의 경우 함수 호출방식에 의해 `this`에 바인딩할 어떤 객체가 동적으로 결정된다.

함수의 호출하는 방식

1. 함수 호출
2. 메소드 호출
3. 생성자 함수 호출
4. apply/call/bind 호출

```
var foo = function () {
  console.dir(this);
};

// 1. 함수 호출
foo(); // window
// window.foo();

// 2. 메소드 호출
var obj = { foo: foo };
obj.foo(); // obj

// 3. 생성자 함수 호출
var instance = new foo(); // instance

// 4. apply/call/bind 호출
var bar = { name: 'bar' };
foo.call(bar);   // bar
foo.apply(bar);  // bar
foo.bind(bar)(); // bar
```

함수 호출

전역객체는 모든 객체의 유일한 최상위 객체를 의미하며 일반적으로 Browser-side에서는 `window`,Sercer-side(Node.js)에서는 `global` 객체를 의미한다.

```
// in browser console
this === window // true

// in Terminal
node
this === global // true
```

전역객체는 전역 스코프를 갖는 전역변수를 프로퍼티로 소유한다. 글로벌 영역에 선언한 함수는 전역객체의 프로퍼티로 접근할 수 있는 전역변수의 메소드이다. 기본적으로 this는 전역객체에 바인딩된다. 전역함수는 물론이고 심지어 내부함수의 경우도 this는 외부함수가 아닌 전역객체에 바인딩된다.

```
function foo() {
  console.log("foo's this: ",  this);  // window
  function bar() {
    console.log("bar's this: ", this); // window
  }
  bar();
}
foo();
```

`this` 전역객체에 바인딩

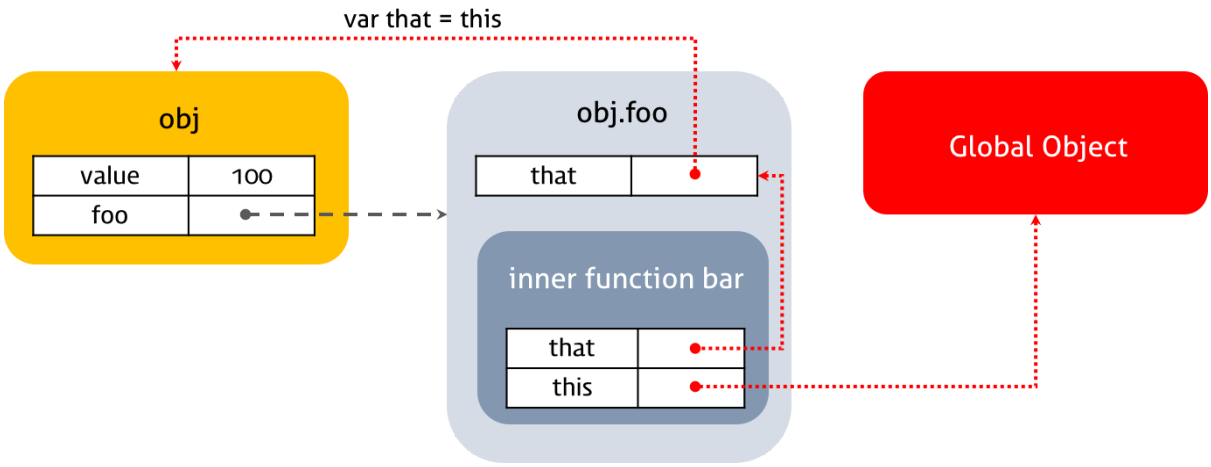
```
var value = 1;

var obj = {
  value: 100,
  foo: function() {
    console.log("foo's this: ",  this);  // obj
    console.log("foo's this.value: ",  this.value); // 100
    function bar() {
      console.log("bar's this: ",  this); // window
      console.log("bar's this.value: ", this.value); // 1
    }
    bar();
  }
};

obj.foo();
```

```
var value = 1;

var obj = {
  value: 100,
  foo: function() {
    setTimeout(function() {
      console.log("callback's this: ",  this);  // window
      console.log("callback's this.value: ",  this.value); // 1
    }, 100);
  }
};

obj.foo();
```

내부함수는 일반 함수, 메소드, 콜백함수 어디에서 선언되었든 관계없이 this는 전역객체를 바인딩한다.

내부함수의 `this`가 전역객체를 참조하는 것을 회피하는 방법은 아래와 같다.

```
var value = 1;

var obj = {
  value: 100,
  foo: function() {
    var that = this;  // Workaround : this === obj

    console.log("foo's this: ",  this);  // obj
    console.log("foo's this.value: ",  this.value); // 100
    function bar() {
      console.log("bar's this: ",  this); // window
      console.log("bar's this.value: ", this.value); // 1

      console.log("bar's that: ",  that); // obj
      console.log("bar's that.value: ", that.value); // 100
    }
    bar();
  }
};

obj.foo();
```



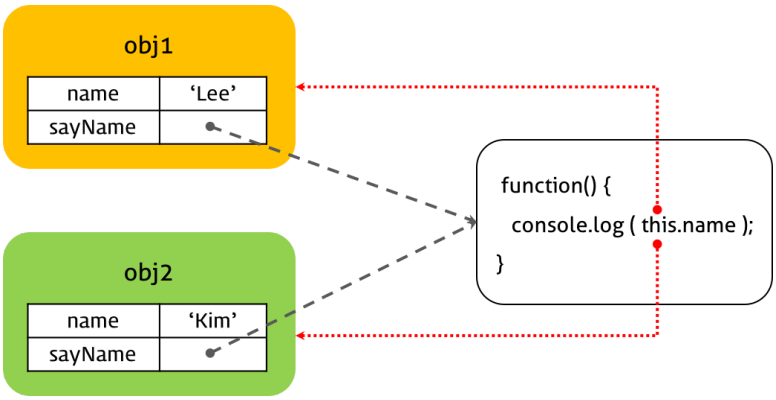
this를 명시적으로 바인딩 할수 있는 apply, call, bind 메소드를 제공

```
var value = 1;

var obj = {
  value: 100,
  foo: function() {
    console.log("foo's this: ",  this);  // obj
    console.log("foo's this.value: ",  this.value); // 100
    function bar(a, b) {
      console.log("bar's this: ",  this); // obj
      console.log("bar's this.value: ", this.value); // 100
      console.log("bar's arguments: ", arguments);
    }
    bar.apply(obj, [1, 2]);
    bar.call(obj, 1, 2);
    bar.bind(obj)(1, 2);
  }
};

obj.foo();
```

메소드 호출

함수가 객체의 프로퍼티 값이면 메소드로서 호출된다. 이때 메소드 내부의 this는 해당 메소드를 소유한 객체 즉 해당 메소드를 호출한 객체에 바인딩된다.

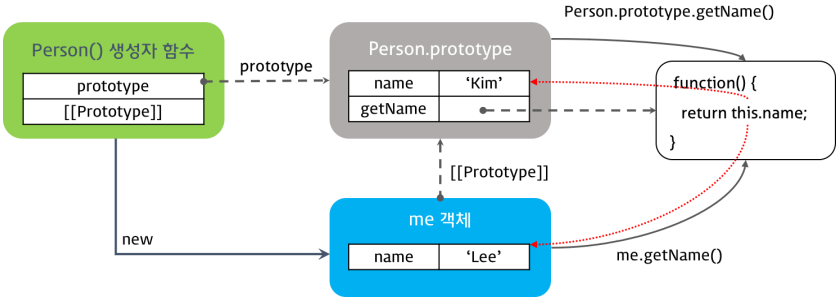
```
var obj1 = {
  name: 'Lee',
  sayName: function() {
    console.log(this.name);
  }
}

var obj2 = {
  name: 'Kim'
}

obj2.sayName = obj1.sayName;

obj1.sayName(); //Lee
obj2.sayName(); //Kim
```



프로토타입 객체 메소드 내부에서 사용된 this도 일반 메소드 방식과 마찬가지로 해당 메소드를 호출한 객체에 바인딩된다.

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

Person.prototype.getName = function() {
  return this.name;
}

var me = new Person('Lee');
console.log(me.getName()); //Lee
```

```
Person.prototype.name = 'Kim';
console.log(Person.prototype.getName()); //Kim
```



생성자 함수 호출

기존 함수에 new 연산자를 붙여서 호출하면 해당 함수는 생성자 함수로 동작한다. 생성자 함수가 아닌 일반 함수에 new 연산자를 붙여 호출하면 생성자 함수처럼 동작할 수 있다. 따라서 일반적으로 생성자 함수명은 첫문자를 대문자로 기술하여 혼란을 방지하려는 노력을 한다.

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

var me = new Person('Lee');
console.log(me); // Person&amp;nbsp;{name: "Lee"}

// new 연산자와 함께 생성자 함수를 호출하지 않으면 생성자 함수로 동작하지 않는다.
var you = Person('Kim');
console.log(you); // undefined
```

생성자 함수 동작 방식

new 연산자와 함께 생성자 함수를 호출하면 다음고 같은 수순으로 동작한다.

1. 빈 객체 생성 및 this 바인딩

생성자 함수의 코드가 실행되기 전 빈 객체가 생성된다. 이 빈 객체가 생성자 함수가 새로 생성하는 객체이다. 이후 **생성자 함수 내에서 사용되는 this는 이 빈 객체를 가리킨다.** 그리고 생성된 빈 객체는 생성자 함수의 prototype 프로퍼티가 가리키는 객체를 자신의 프로토타입 객체로 설정한다.

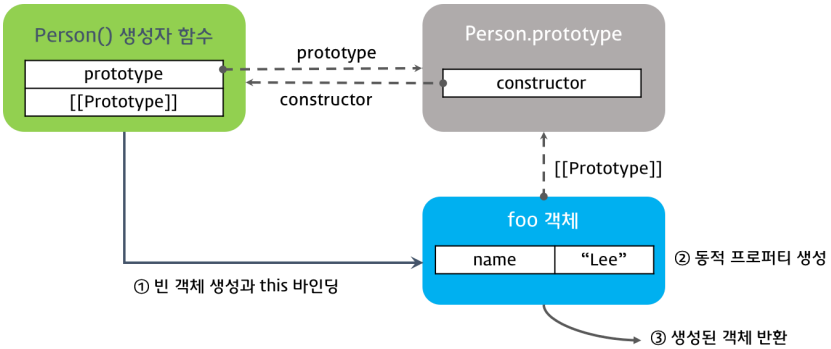
2. this를 통한 프로퍼티 생성

생성된 빈 객체에 this를 사용하여 동적으로 프로퍼티나 메소드를 생성할 수 있다. this는 새로 생성된 객체를 가리키므로 this를 통해 생성한 프로퍼티와 메소드는 새로 생성된 객체에 추가된다.

3. 생성된 객체 반환

• 반환문이 없는 경우, this에 바인딩된 새로 생성한 객체가 반환된다. 명시적으로 this를 반환하여도 결과는 같다.
• 반환문이 this가 아닌 다른 객체를 명시적으로 반환하는 경우, this가 아닌 해당 객체가 반환된다. 이때 this를 반환하지 않은 함수는 생성자 함수로서의 역할을 수행하지 못한다. 따라서 생성자 함수는 반환문을 명시적으로 사용하지 않는다.

```
function Person(name) {
  // 생성자 함수 코드 실행 전 -------- 1
  this.name = name;  // --------- 2
  // 생성된 함수 반환 -------------- 3
}

var me = new Person('Lee');
console.log(me.name); //Lee
```



객체 리터럴 방식과 생성자 함수 방식의 차이

```
// 객체 리터럴 방식
var foo = {
  name: 'foo',
  gender: 'male'
}

console.dir(foo); /* [object Object] {
                       gender: "male",
                       name: "foo"
                     } */

// 생성자 함수 방식
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
}

var me  = new Person('Lee', 'male');
console.dir(me); /* [object Object] {
                      gender: "male",
                      name: "Lee"
                    } */

var you = new Person('Kim', 'female');
console.dir(you); /* [object Object] {
                      gender: "female",
                      name: "Kim"
                    } */
```

객체 리터럴 방식과 생성자 함수 방식의 차이는 프로토타입 객체([[Prototype]])에 있다.

- 객체 리터럴 방식의 경우, 생성된 객체의 프로토타입 객체는 Object.prototype이다.
- 생성자 함수 방식의 경우, 생성된 객체의 프로토타입 객체는 Person.prototype이다.

생성자 함수에 new 연산자를 붙이지 않고 호출할 경우

일반함수와 생성자 함수에 특별한 형식적 차이는 없으며 함수에 new연산자를 붙여서 호출하면 해당 함수는 생성자 함수로 동작한다.

그러나 객체 생성 목적으로 작성한 생성자 함수를 new 없이 호출하거나 일반함수에 new를 붙여 호출하면 오류가 발생할 수 있다. 일반함수와 생성자 함수의 호출시 this 바인딩 방식이 다르기 때문이다.

일반 함수를 호출하면 this는 전역객체에 바인딩되지만 new 연산자와 함께 생성자 함수를 호출하면 this는 생성자 함수가 암묵적으로 생성한 빈 객체에 바인딩된다.

```
function Person(name) {
  // new없이 호출하는 경우, 전역객체에 name 프로퍼티를 추가
  this.name = name;
};

// 일반 함수로서 호출되었기 때문에 객체를 암묵적으로 생성하여 반환하지 않는다.
// 일반 함수의 this는 전역객체를 가리킨다.
var me = Person('Lee');

console.log(me); // undefined
console.log(window.name); // Lee
```

위험성을 회피하기 위해 사용되는 패턴(Scope-Safe Constructor)은 다음과 같다. 이 패턴은 대부분의 라이브러리에서 광범위하게 사용된다.

참고로 대부분의 빌트인 생성자(Object, Regex, Array 등)는 new 연산자와 함께 호출되었는지를 확인한 후 적절한 값을 반환한다.

다시 말하지만 new 연산자와 함께 생성자 함수를 호출하는 경우, 생성자 함수 내부의 this는 생성자 함수에 의해 생성된 인스턴스를 가리킨다. 따라서 아래 A 함수가 new 연산자와 함께 생성자 함수로써 호출되면 A 함수 내부의 this는 A 생성자 함수에 의해 생성된 인스턴스를 가리킨다.

```
// Scope-Safe Constructor Pattern
function A(arg) {
  // 생성자 함수가 new 연산자와 함께 호출되면 함수의 선두에서 빈객체를 생성하고 this에 바인딩한다.

  /*
  this가 호출된 함수(arguments.callee, 본 예제의 경우 A)의 인스턴스가 아니면 new 연산자를 사용하지 않은 것이므로 이 경우 new와 함께 생성자 함수를 호출하여 인스턴
  arguments.callee는 호출된 함수의 이름을 나타낸다. 이 예제의 경우 A로 표기하여도 문제없이 동작하지만 특정함수의 이름과 의존성을 없애기 위해서 arguments.callee를
  */
  if (!(this instanceof arguments.callee)) {
    return new arguments.callee(arg);
  }

  // 프로퍼티 생성과 값의 할당
  this.value = arg ? arg : 0;
}

var a = new A(100);
var b = A(10);

console.log(a.value); //100
console.log(b.value); //10
```

callee는 arguments 객체의 프로퍼티로서 함수 바디 내에서 현재 실행 중인 함수를 참조할 때 사용한다. 다시 말해, 함수 바디 내에서 현재 실행 중인 함수의 이름을 반환한다.

apply/call/bind 호출

this에 바인딩될 객체는 함수 호출 패턴에 의해 결정된다. 이는 자바스크립트 엔진이 수행하는 것이다. 이러한 자바스크립트 엔진의 암묵적 this 바인딩 이외에 this를 특정 객체에 명시적으로 바인딩하는 방법도 제공된다. 이것을 가능하게 하는 것이 Function.prototype.apply, Function.prototype.call 메소드이다. 이 메소드들은 모든 함수 객체의 프로토타입 객체인 Function.prototype 객체의 메소드이다.

```
func.apply(thisArg, [argsArray])

// thisArg: 함수 내부의 this에 바인딩할 객체
// argsArray: 함수에 전달할 argument의 배열
```

apply() 메소드를 호출하는 주체는 함수이며 apply() 메소드는 this를 특정 객체에 바인딩할 뿐 본직적인 기능은 함수 호출이다.

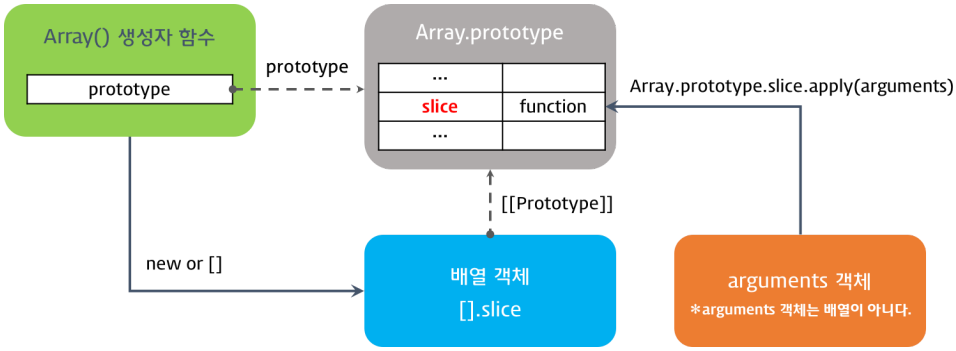
```
var Person = function (name) {
  this.name = name;
};

var foo = {};

// apply 메소드는 생성자함수 Person을 호출한다. 이때 this에 객체 foo를 바인딩한다.
Person.apply(foo, ['name']);

console.log(foo); // { name: 'name' }
```

빈 객체 foo를 apply() 메소드의 첫번째 매개변수에, argument의 배열을 두번째 매개변수에 전달하면서 Person 함수를 호출하였다. 이때 Person 함수의 this는 foo 객체가 된다. Person 함수는 this의 name 프로퍼티에 매개변수 name에 할당된 인수를 할당하는데 this에 바인딩된 foo 객체에는 name 프로퍼티가 없으므로 name 프로퍼티가 동적 추가되고 값이 할당된다.

apply() 메소드의 대표적인 용도는 arguments 객체와 같은 유사 배열 객체에 배열 메소드를 사용하는 경우이다. arguments 객체는 배열이 아니기 때문에 slice() 같은 배열의 메소드를 사용할 수 없으나 apply() 메소드를 이용하면 가능하다.

```
function convertArgsToArray() {
  console.log(arguments); /* [object Arguments] {
                             0: 1,
                             1: 2,
                             2: 3
                           } */
```

```
  // arguments 객체를 배열로 변환
  // slice: 배열의 특정 부분에 대한 복사본을 생성한다.
  var arr = Array.prototype.slice.apply(arguments); // arguments.slice
  // var arr = [].slice.apply(arguments);

  console.log(arr); //[1, 2, 3]
  return arr;
}

convertArgsToArray(1, 2, 3);
```



call() 메소드의 경우, apply()와 기능은 같지만 apply()의 두번째 인자에서 배열 형태로 넘긴 것을 각각 하나의 인자로 넘긴다.

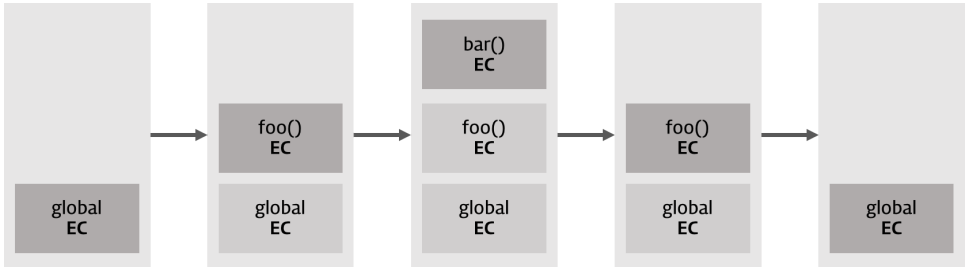
apply() 와 call()메소드는 콜백 함수의 this를 위해서 사용되기도 한다. `Person.apply(foo, [1, 2, 3]);` , `Person.call(foo, 1, 2, 3);`

실행 컨텍스트

실행 가능한 코드를 형상화하고 구분하는 추상적인 개념이라고 정의한다. 실행 가능한 코드가 실행되기 위해 필요한 환경

- 전역 코드 : 전역 영역에 존재하는 코드
- Eval 코드 : eval 함수로 실행되는 코드
- 함수 코드 : 함수 내에 존재하는 코드

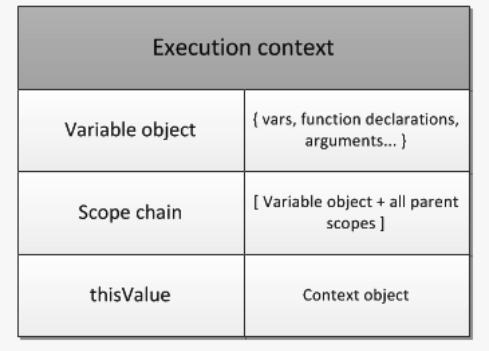
```
var x = 'xxx';

function foo () {
  var y = 'yyy';

  function bar () {
    var z = 'zzz';
    console.log(x + y + z);
  }
  bar();
}
foo();
```



논리적 스택 수조를 가지는 실행 컨텍스트 스택

1. 컨트롤이 실행 가능한 코드로 이동하면 논리적 스택 구조를 가지는 새로운 실행 컨텍스트 스택이 생성된다. 스택은 LIFO(Last In First Out, 후입 선출)의 구조를 가지는 나열 구조이다.
2. 전역 코드(Global code)로 컨트롤이 진입하면 전역 실행 컨텍스트가 생성되고 실행 컨텍스트 스택에 쌓인다. 전역 실행 컨텍스트는 애플리케이션이 종료될 때(웹 페이지에서 나가거나 브라우저를 닫을 때)까지 유지된다.
3. 함수를 호출하면 해당 함수의 실행 컨텍스트가 생성되며 직전에 실행된 코드 블록의 실행 컨텍스트 위에 쌓인다.
4. 함수 실행이 끝나면 해당 함수의 실행 컨텍스트를 파기하고 직전의 실행 컨텍스트에 컨트롤을 반환한다.

실행 컨텍스트의 3가지 객체



실행 컨텍스트의 구조

Variable Object(VO/ 변수객체)

실행 컨텍스트가 생성되면 자바스크립트 엔진은 실행에 필요한 여러 정보들을 담을 객체를 생성한다. 이를 Variable Object라고 한다.

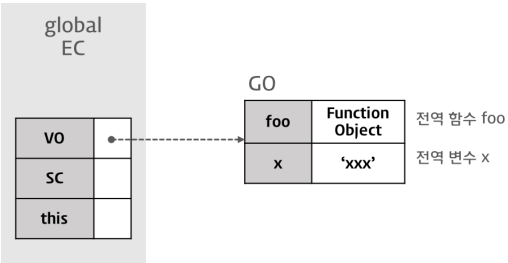
Variable Object는 아래의 정보를 담는 객체이다.

- 변수
- 매개변수(parameter)와 인수 정보(arguments)
- 함수 선언(함수 표현식은 제외)

Variable Object가 가리키는 객체

전역 컨텍스트의 경우

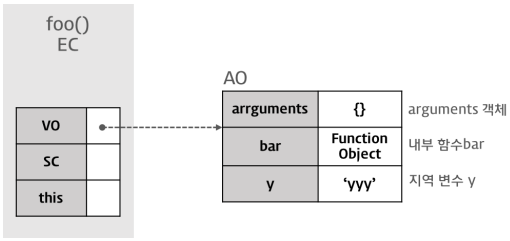
Variable Object는 유일하며 최상위에 위치하고 모든 전역 변수, 전역 함수 등을 포함하는 **전역 객체(Global Object / GO)**를 가리킨다. 전역 객체는 전역에 선언된 전역 변수와 전역 함수를 프로퍼티로 소유한다.



전역 컨텍스트의 경우, Variable Object가 가리키는 전역 객체

함수 컨텍스트의 경우

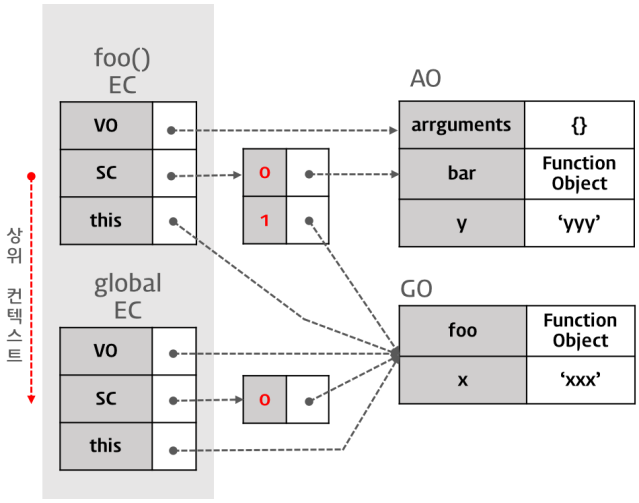
Variable Object는 **Activation Object(AO / 활성 객체)**를 가리키며 매개변수와 인수들의 정보를 배열의 형태로 담고 있는 객체인 arguments object가 추가된다.



함수 컨텍스트의 경우 Variable Object가 가리키는 Activation Object

Scope Chain(SC)

일종의 리스트로서 전역 객체와 중첩된 함수의 스코프의 레퍼런스를 차례로 저장하고 있다. 해당 전역 또는 함수가 참조할 수 있는 변수, 함수 선언 등의 정보를 담고 있는 전역객체(GO) 또는 활성 객체(AO)의 리스트를 가리킨다.



스코프체인

스코프 체인은 식별자 중에서 객체(전역 객체 제외)의 프로퍼티가 아닌 식별자, 즉 변수를 검색하는 메커니즘이다.

식별자 중에서 변수가 아닌 객체의 프로퍼티(물론 메소드도 포함된다)를 검색하는 메커니즘은**프로토타입 체인(Prototype Chain)**이다

this value

this프로퍼티에는 this 값이 할당된다. this에 할당되는 값은 함수 호출 패턴에 의해 결정된다

실행 컨텍스트의 생성과정

```
var x = 'xxx';

function foo () {
  var y = 'yyy';

  function bar () {
    var z = 'zzz';
    console.log(x + y + z);
  }
  bar();
}

foo();
```

전역 코드에의 진입

1. 스코프 체인의 생성과 초기화
2. Variable Instantiation(변수 객체화) 실행

a. 함수 foo의 선언 처리
b. 변수 x의 선언처리

3. this value 결정

전역 코드의 실행

1. 변수 값의 할당
2. 함수 foo의 실행
   a. 스코프 체인의 생성과 초기화
   b. Variable Instantiation 실행
   c. this value 결정

foo 함수 코드의 실행

1. 변수 값의 할당
2. 함수 bar의 실행
   a. 스코프 체인의 생성과 초기화
   b. Variable Instantiation 실행
   c. this value 결정

클로저

함수가 선언됐을때의 렉시컬 환경

```
function outerFunc() {
  var x = 10;
  var innerFunc = function () { console.log(x); };
  innerFunc();
}

outerFunc(); // 10
```

스코프는 함수를 호출할 때가 아니라 함수를 어디에 선언하였는지에 따라 결정된다. 이를 **렉시컬 스코핑(Lexical scoping)** 라 한다. 위 예제의 함수 innerFunc는 함수 outerFunc의 내부에서 선언되었기 때문에 함수 innerFunc의 상위 스코프는 함수 outerFunc이다. 함수 innerFunc가 전역에 선언되었다면 함수 innerFunc의 상위 스코프는 전역 스코프가 된다.

내부함수 innerFunc가 자신을 포함하고 있는 외부함수 outerFunc의 변수 x에 접근할 수 있는 것, 다시 말해 상위 스코프에 접근할 수 있는 것은 렉시컬 스코프의 레퍼런스를 차례대로 저장하고 있는 실행 컨텍스트의 **스코프 체인**을 자바스크립트 엔진이 검색하였기에 가능한 것이다. 좀더 자세히 설명하면 아래와 같다.

1. innerFunc 함수 스코프(함수 자신의 스코프를 가리키는 활성 객체) 내에서 변수 x를 검색한다. 검색이 실패하였다.
2. innerFunc 함수를 포함하는 외부 함수 outerFunc의 스코프(함수 outerFunc의 스코프를 가리키는 함수 outerFunc의 활성 객체)에서 변수 x를 검색한다. 검색이 성공하였다.

```
function outerFunc() {
  var x = 10;
  var innerFunc = function () { console.log(x); };
  return innerFunc;
}

/**
 *  함수 outerFunc를 호출하면 내부 함수 innerFunc가 반환된다.
 *  그리고 함수 outerFunc의 실행 컨텍스트는 소멸한다.
 */
var inner = outerFunc();
inner(); // 10
```

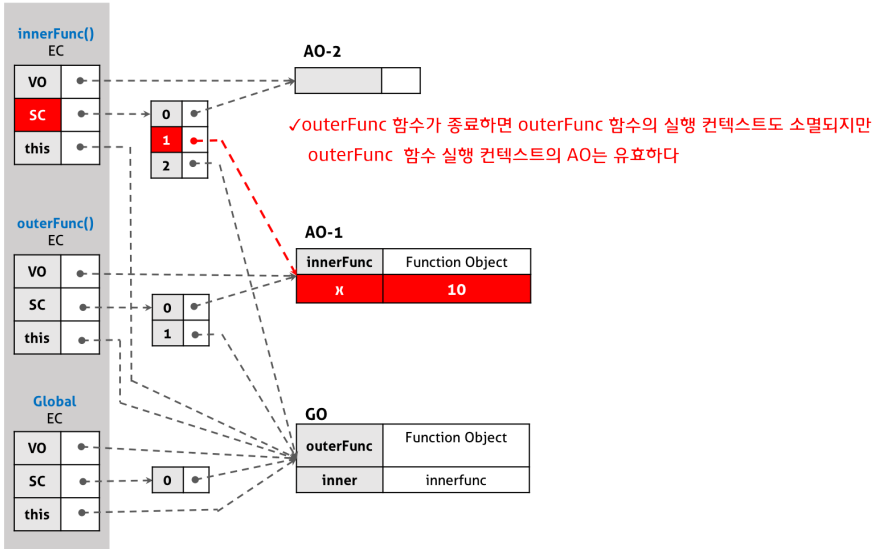
함수 outerFunc는 내부함수 innerFunc를 반환하고 생을 마감했다. 즉, 함수 outerFunc는 실행된 이후 콜스택(실행 컨텍스트 스택)에서 제거되었으므로 함수 outerFunc의 변수 x 또한 더이상 유효하지 않게 되어 변수 x에 접근할 수 있는 방법은 달리 없어 보인다. 그러나 위

코드의 실행 결과는 변수 x의 값인 10이다. 이미 life-cycle이 종료되어 실행 컨텍스트 스택에서 제거된 함수 outerFunc의 지역변수 x가 다시 부활이라도 한 듯이 동작하고 있다. 뭔가 특별한 일이 일어나고 있는 것 같다.

이처럼 자신을 포함하고 있는 외부함수보다 내부함수가 더 오래 유지되는 경우, **외부 함수 밖에서 내부함수가 호출되더라도 외부함수의 지역 변수에 접근할 수 있는데 이러한 함수를 클로저(Closure)**라고 부른다.

클로저는 **반환된 내부함수가 자신이 선언됐을 때의 환경인 스코프를 기억하여 자신이 선언됐을 때의 환경(스코프) 밖에서 호출되어도 그 환경(스코프)에 접근할 수 있는 함수 를 말한다. 자신이 생성될 때의 환경을 기억하는 함수다**

클로저에 의해 참조되는 외부함수의 변수 즉 outerFunc 함수의 변수 x를 자유변수라고 부른다.



클로저의 활용

상태유지

현재 상태를 기억하고 변경된 최신상태를 유지하는 것이다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="toggle">toggle</button>
  <div class="box" style="width: 100px; height: 100px; background: red;"></div>

  <script>
    var box = document.querySelector('.box');
    var toggleBtn = document.querySelector('.toggle');

    var toggle = (function () {
      var isShow = false;

      // ① 클로저를 반환
      return function () {
        box.style.display = isShow ? 'block' : 'none';
        // ③ 상태 변경
        isShow = !isShow;
      };
    })();

    // ② 이벤트 프로퍼티에 클로저를 할당
    toggleBtn.onclick = toggle;
  </script>
</body>
</html>
```

① 즉시실행함수는 함수를 반환하고 즉시 소멸한다. 즉시실행함수가 반환한 함수는 자신이 생성됐을 때의 렉시컬 환경(Lexical environment)에 속한 변수 isShow를 기억하는 클로저다. 클로저가 기억하는 변수 isShow는 box 요소의 표시 상태를 나타낸다.

② 클로저를 이벤트 핸들러로서 이벤트 프로퍼티에 할당했다. 이벤트 프로퍼티에서 이벤트 핸들러인 클로저를 제거하지 않는 한 클로저가 기억하는 렉시컬 환경의 변수 isShow는 소멸하지 않는다. 다시 말해 현재 상태를 기억한다.

③ 버튼을 클릭하면 이벤트 프로퍼티에 할당한 이벤트 핸들러인 클로저가 호출된다. 이때 .box 요소의 표시 상태를 나타내는 변수 isShow의 값이 변경된다. 변수 isShow는 클로저에 의해 참조되고 있기 때문에 유효하며 자신의 변경된 최신 상태를 계속해서 유지한다.

전역 변수의 사용 억제

버튼이 클릭될 때마다 클락한 횟수가 누적되어 화면에 표시되는 카운터

```
<!DOCTYPE html>
<html>
<body>
  <p>전역 변수를 사용한 Counting</p>
  <button id="inclease">+</button>
  <p id="count">0</p>
  <script>
    var incleaseBtn = document.getElementById('inclease');
    var count = document.getElementById('count');

    // 카운트 상태를 유지하기 위한 전역 변수
    var counter = 0;

    function increase() {
      return ++counter;
    }

    incleaseBtn.onclick = function () {
      count.innerHTML = increase();
    };
  </script>
</body>
</html>
```

increase 함수는 호출되기 직전에 전역변수 counter의 값이 반드시 `0` 이여야 제대로 동작한다. 하지만 변수 counter는 전역 변수이기 때문에 언제든지 누구나 접근할 수 있고 변경할 수 있다. 이는 의도치 않게 값이 변경될 수 있다는 것을 의미한다. 만약 누군가에 의해 의도치 않게 전역 변수 counter의 값이 변경됐다면 이는 오류로 이어진다. 변수 counter는 카운터를 관리하는 increase 함수가 관리하는 것이 바람직하다.

```
<!DOCTYPE html>
<html>
<body>
  <p>지역 변수를 사용한 Counting</p>
  <button id="inclease">+</button>
  <p id="count">0</p>
  <script>
    var incleaseBtn = document.getElementById('inclease');
    var count = document.getElementById('count');

    function increase() {
      // 카운트 상태를 유지하기 위한 지역 변수
      var counter = 0;
      return ++counter;
    }

    incleaseBtn.onclick = function () {
      count.innerHTML = increase();
    };
  </script>
</body>
</html>
```

전역변수를 지역변수로 변경하여 의도치 않은 상태 변경은 방지했다. 하지만 increase 함수가 호출될 때마다 지역변수 counter를 0으로 초기화하기 때문에 언제나 1이 표시된다. 다시 말해 **변경된 이전 상태를 기억하지 못한다.**

```
<!DOCTYPE html>
<html>
  <body>
  <p>클로저를 사용한 Counting</p>
  <button id="inclease">+</button>
  <p id="count">0</p>
  <script>
    var incleaseBtn = document.getElementById('inclease');
    var count = document.getElementById('count');

    var increase = (function () {
      // 카운트 상태를 유지하기 위한 자유 변수
      var counter = 0;
      // 클로저를 반환
```

```
      return function () {
        return ++counter;
      };
    }());

    incleaseBtn.onclick = function () {
      count.innerHTML = increase();
    };
  </script>
</body>
</html>
```

스크립트가 실행되면 즉시 실행함수가 호출되고 변수 increase에는 함수 `function (){ return ++counter; }` 가 할당된다. 이 함수는 자신이 생성됐을 때의 렉시컬 환경을 기억하는 클로저다. 즉시실행함수는 호출된 이후 소멸되지만 즉시실행함수가 반환한 함수는 변수 increase에 할당되어 inclease 버튼을 클릭하면 클릭 이벤트 핸들러 내부에서 호출된다. 이때 클로저인 이 함수는 자신이 선언됐을 때의 렉시컬 환경인 즉시실행함수의 스코프에 속한 지역변수 counter를 기억한다. 따라서 즉시실행함수의 변수 counter에 접근할 수있고 변수 counter는 자신을 참조하는 함수가 소멸될 때까지 유지된다.

변수의 값은 누군가에 의해 언제든지 변경될 수 있어 오류 발생의 근본적 원인이 될 수 있다. 상태 변경이나 가변(mutable) 데이터를 피하고 **불변성(Immutability)을 지향**하는 함수형 프로그래밍에서 **부수 효과(Side effect)를 최대한 억제**하여 오류를 피하고 프로그램의 안정성을 높이기 위해 클로저는 적극적으로 사용된다.

```
// 함수를 인자로 전달받고 함수를 반환하는 고차 함수
// 이 함수가 반환하는 함수는 클로저로서 카운트 상태를 유지하기 위한 자유 변수 counter을 기억한다.
function makeCounter(predicate) {
  // 카운트 상태를 유지하기 위한 자유 변수
  var counter = 0;
  // 클로저를 반환
  return function () {
    counter = predicate(counter);
    return counter;
  };
}

// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// 함수로 함수를 생성한다.
// makeCounter 함수는 보조 함수를 인자로 전달받아 함수를 반환한다
const increaser = makeCounter(increase);
console.log(increaser()); // 1
console.log(increaser()); // 2

// increaser 함수와는 별개의 독립된 렉시컬 환경을 갖기 때문에 카운터 상태가 연동하지 않는다.
const decreaser = makeCounter(decrease);
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

함수 makeCounter는 보조 함수를 인자로 전달받고 함수를 반환하는 고차 함수이다. 함수 makeCounter가 반환하는 함수는 자신이 생성됐을 때의 렉시컬 환경인 함수 makeCounter의 스코프에 속한 변수 counter을 기억하는 클로저다. 함수 makeCounter는 인자로 전달받은 보조 함수를 합성하여 자신이 반환하는 함수의 동작을 변경할 수 있다.

정보의 은닉

```
function Counter() {
  // 카운트를 유지하기 위한 자유 변수
  var counter = 0;

  // 클로저
  this.increase = function () {
    return ++counter;
  };

  // 클로저
  this.decrease = function () {
    return --counter;
  };
```

```
}

const counter = new Counter();

console.log(counter.increase()); // 1
console.log(counter.decrease()); // 0
```

생성자 함수 Counter는 increase, decrease 메소드를 갖는 인스턴스를 생성한다. 이 메소드들은 모두 자신이 생성됐을 때의 렉시컬 환경인 생성자 함수 Counter의 스코프에 속한 변수 counter를 기억하는 클로저이며 렉시컬 환경을 공유한다. 생성자 함수가 함수가 생성한 객체의 메소드는 객체의 프로퍼티에만 접근할 수 있는 것이 아니며 자신이 기억하는 렉시컬 환경의 변수에도 접근할 수 있다.

이때 생성자 함수 Counter의 변수 counter는 this에 바인딩된 프로퍼티가 아니라 변수다. counter가 this에 바인딩된 프로퍼티라면 생성자 함수 Counter가 생성한 인스턴스를 통해 외부에서 접근이 가능한 `public` 프로퍼티가 되지만 생성자 함수 Counter 내에서 선언된 변수 counter는 생성자 함수 Counter 외부에서 접근할 수 없다. 하지만 생성자 함수 Counter가 생성한 인스턴스의 메소드인 increase, decrease는 클로저이기 때문에 자신이 생성됐을 때의 렉시컬 환경인 생성자 함수 Counter의 변수 counter에 접근할 수 있다. 이러한 클로저의 특징을 사용해 클래스 기반 언어의 `private` 키워드를 흉내낼 수 있다.

자주 발생하는 실수

```
var arr = [];

for (var i = 0; i < 5; i++) {
  arr[i] = function () {
    return i;
  };
}

for (var j = 0; j < arr.length; j++) {
  console.log(arr[j]());
}
```

```
var arr = [];

for (var i = 0; i < 5; i++){
  arr[i] = (function (id) { // ②
    return function () {
      return id; // ③
    };
  }(i)); // ①
}

for (var j = 0; j < arr.length; j++) {
  console.log(arr[j]());
}
```

1 - 배열 arr에 5개의 함수가 할당되고 각각의 함수는 순차적으로 0, 1, 2, 3, 4를 반환할 것으로 기대하겠지만 결과는 그렇지않다. for 문에서 사용한 변수 i는 전역 변수이기 때문이다.

2 - ① 배열 arr에는 즉시실행함수에 의해 함수가 반환된다.

② 이때 즉시실행함수는 i를 인자로 전달받고 매개변수 id에 할당한 후 내부 함수를 반환하고 life-cycle이 종료된다. 매개변수 id는 자유변수가 된다.

③ 배열 arr에 할당된 함수는 id를 반환한다. 이때 id는 상위 스코프의 자유변수이므로 그 값이 유지된다.

위 예제는 자바스크립트의 함수 레벨 스코프 특성으로 인해 for 루프의 초기문에서 사용된 변수의 스코프가 전역이 되기 때문에 발생하는 현상이다.

상속

자바스크립트는 기본적으로 프로토타입을 통해 상속을 구현한다. 이것은 프로토타입을 통해 객체가 다른 객체로 직접 상속된다는 의미이다.

의사 클래스 패턴 상속

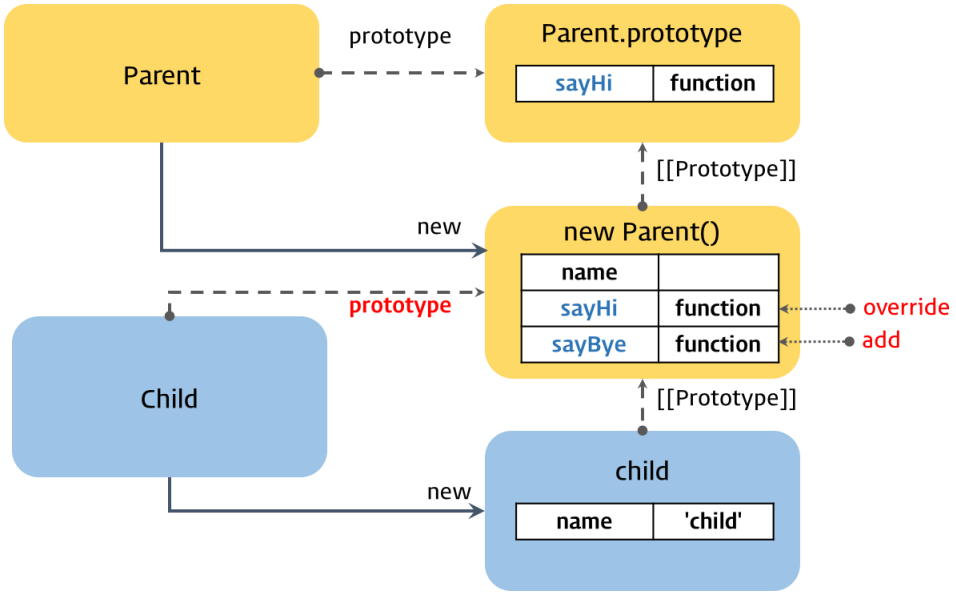
자식 생성자 함수의 prototype 프로퍼티를 부모 생성자 함수의 인스턴스로 교체하여 상속을 구현하는 방법 부모와 자식 모두 생성자 함수를 정의하여야 함

```
// 부모 생성자 함수
var Parent = (function () {
  // Constructor
  function Parent(name) {
    this.name = name;
  }

  // method
  Parent.prototype.sayHi = function () {
    console.log('Hi! ' + this.name);
  };

  // return constructor
  return Parent;
}());
```

```
// 자식 생성자 함수
var Child = (function () {
  // Constructor
  function Child(name) {
    this.name = name;
  }

  // 자식 생성자 함수의 프로토타입 객체를 부모 생성자 함수의 인스턴스로 교체.
  Child.prototype = new Parent(); // ②

  // 메소드 오버라이드
  Child.prototype.sayHi = function () {
    console.log('안녕하세요! ' + this.name);
  };

  // sayBye 메소드는 Parent 생성자함수의 인스턴스에 위치된다
  Child.prototype.sayBye = function () {
    console.log('안녕히가세요! ' + this.name);
  };

  // return constructor
  return Child;
}());

var child = new Child('child'); // ①
console.log(child);  // Parent { name: 'child' }

console.log(Child.prototype); // Parent { name: undefined, sayHi: [Function], sayBye: [Function] }

child.sayHi();  // 안녕하세요! child
child.sayBye(); // 안녕히가세요! child

console.log(child instanceof Parent); // true
console.log(child instanceof Child);  // true
```

Child 생성자 함수가 생성한 인스턴스 child(①)의 프로토타입 객체는 Parent 생성자 함수가 생성한 인스턴스(②)이다. 그리고 Parent 생성자 함수가 생성한 인스턴스의 프로토타입 객체는 Parent.prototype이다.

이로써 child는 프로토타입 체인에 의해 Parent 생성자 함수가 생성한 인스턴스와 Parent.prototype의 모든 프로퍼티에 접근할 수 있게 되었다. 이름은 의사 클래스 패턴 상속이지만 내부에서는 프로토타입을 사용하는 것은 변함이 없다.



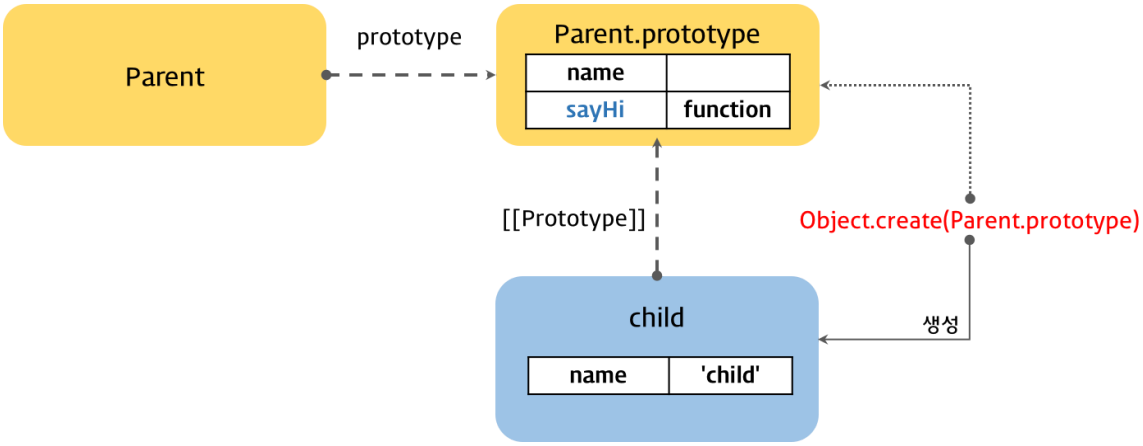
1. new 연산자를 통해 인스턴스를 생성한다.
2. 생성자 링크의 파괴
3. 객체리터럴

프로토타입 패턴 상속

Object.create 함수를 사용하여 객체에서 다른 객체로 직접 상속을 구현하는 방식 프로토타입 패턴 상속은 개념적으로 의사클래스 패턴 상속보다 더 간단, 또한 의사클래스 패턴의 단점인 new 연산자가 필요없으며, 생성자 링크도 파괴되지 않으며, 객체리터럴에도 사용할 수

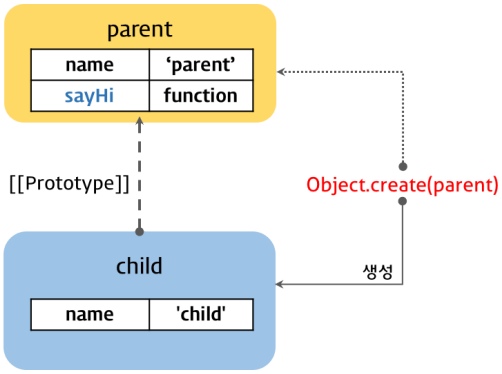
있다.

```
// 부모 생성자 함수
var Parent = (function () {
  // Constructor
  function Parent(name) {
    this.name = name;
  }

  // method
  Parent.prototype.sayHi = function () {
    console.log('Hi! ' + this.name);
  };

  // return constructor
  return Parent;
}());

// create 함수의 인수는 프로토타입이다.
var child = Object.create(Parent.prototype);
child.name = 'child';

child.sayHi();  // Hi! child

console.log(child instanceof Parent); // true
```



객체리터럴 패턴으로 생성한 객체에도 프로토타입 패턴 상속을 사용할 수 있다.

```
var parent = {
  name: 'parent',
  sayHi: function() {
    console.log('Hi! ' + this.name);
  }
};

// create 함수의 인자는 객체이다.
var child = Object.create(parent);
child.name = 'child';

// var child = Object.create(parent, {name: {value: 'child'}});

parent.sayHi(); // Hi! parent
child.sayHi();  // Hi! child

console.log(parent.isPrototypeOf(child)); // true
```

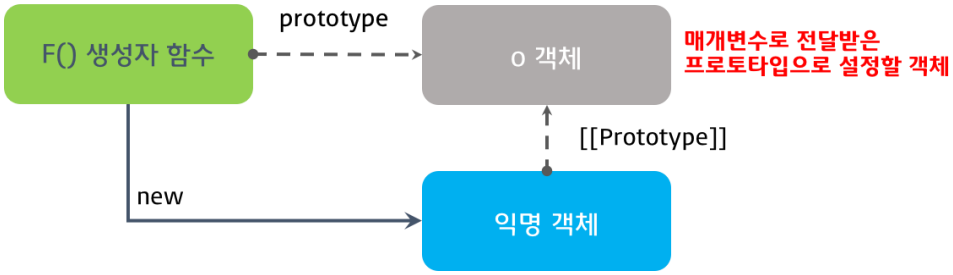


Object.create 함수는 매개변수에 프로토타입으로 설정할 객체 또는 인스턴스를전달하고 이를 상속하는 새로운 객체를 생성한다.

```
// Object.create 함수의 폴리필
if (!Object.create) {
  Object.create = function (o) {
    function F() {}  // 1
    F.prototype = o; // 2
    return new F();  // 3
  };
}
```

위 폴리필은 프로토타입 패턴 상속의 핵심을 담고 있다.

1. 비어있는 생성자 함수 F를 생성한다.
2. 생성자 함수 F의 prototype 프로퍼티에 매개변수로 전달받은 객체를 할당한다.
3. 생성자 함수 F를 생성자로 하여 새로운 객채를 생성하고 반환한다.



캡슐화와 모듈 패턴

캡슐화는 관련있는 멤버 변수와 메소드를 클래스와 같은 하나의 틀 안에 담고 외부에 공개될 필요가 없는 정보는 숨기는 것을 말하며 다른 말로 정보 은닉이라고 한다. 자바스크립트는 자바와같이 `public` 또는 `private` 등의 키워드를 제공하지 않는다.

```
var Person = function(arg) {
  var name = arg ? arg : ''; // ①

  this.getName = function() {
    return name;
  };

  this.setName = function(arg) {
    name = arg;
  };
}

var me = new Person('Lee');

var name = me.getName();

console.log(name);

me.setName('Kim');
name = me.getName();
```

```
console.log(name);
```

①의 name 변수는 private 변수가 된다. 자바스크립트는 function-level scope를 제공하므로 함수 내의 변수는 외부에서 참조할 수 없다. 만약에 var 대신 this를 사용하면 public 멤버가 된다. 단 new 키워드로 객체를 생성하지 않으면 this는 생성된 객체에 바인딩되지 않고 전역객체에 연결된다.

그리고 public 메소드 getName, setName은 클로저로서 private 변수(자유 변수)에 접근할 수 있다. 이것이 기본적인 정보 은닉 방법이다.

```
var person = function(arg) {
  var name = arg ? arg : '';

  return {
    getName: function() {
      return name;
    },
    setName: function(arg) {
      name = arg;
    }
  }
}

var me = person('Lee'); /* or var me = new person('Lee'); */

var name = me.getName();

console.log(name);

me.setName('Kim');
name = me.getName();

console.log(name);
```

person 함수는 객체를 반환한다. 이 객체 내의 메소드 getName, setName은 클로저로서 private 변수 name에 접근할 수 있다. 이러한 방식을 모듈 패턴이라 하며 캡슐화와 정보 은닉을 제공한다.

주의할점

• private 멤버가 객체나 배열일 경우, 반환된 해당 멤버의 변경이 가능하다.

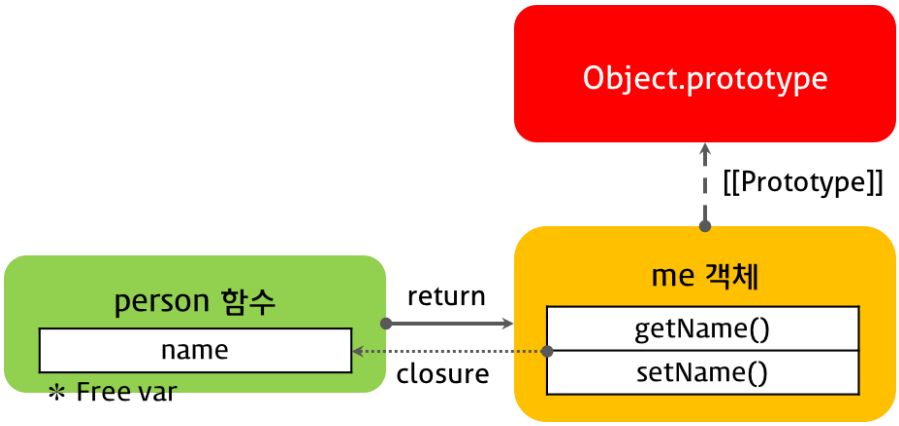
```
var person = function (personInfo) {
  var o = personInfo;

  return {
    getPersonInfo: function() {
      return o;
    }
  };
};

var me = person({ name: 'Lee', gender: 'male' });

var myInfo = me.getPersonInfo();
console.log('myInfo: ', myInfo);
// myInfo:  { name: 'Lee', gender: 'male' }

myInfo.name = 'Kim';

myInfo = me.getPersonInfo();
console.log('myInfo: ', myInfo);
// myInfo:  { name: 'Kim', gender: 'male' }
```

객체를 반환하는 경우 반환값은 얕은 복사로 private 멤버의 참조값을 반환하게 된다. 따라서 외부에서도 private 멤버의 값을 변경할 수 있다. 이를 회피하기 위해서는 객체를 그대로 반환하지 않고 반환해야 할 객체의 정보를 새로운 객체에 담아 반환해야 한다. 반드시 객체 전체가 그대로 반환되어야 하는경우에는 깊은 복사로 복사본을 만들어 반환한다.

• person 함수가 반환한 객체는 person 함수 객체의 프로토타입에 접근할 수 없다. 이는 상속을 구현할 수 없음을 의미한다.

```
var person = function(arg) {
  var name = arg ? arg : '';

  return {
    getName: function() {
      return name;
```

```
    },
    setName: function(arg) {
      name = arg;
    }
  }
}

var me = person('Lee');

console.log(person.prototype === me.__proto__); // false
console.log(me.__proto__ === Object.prototype); // true: 객체 리터럴 방식으로 생성된 객체와 동일하다
```

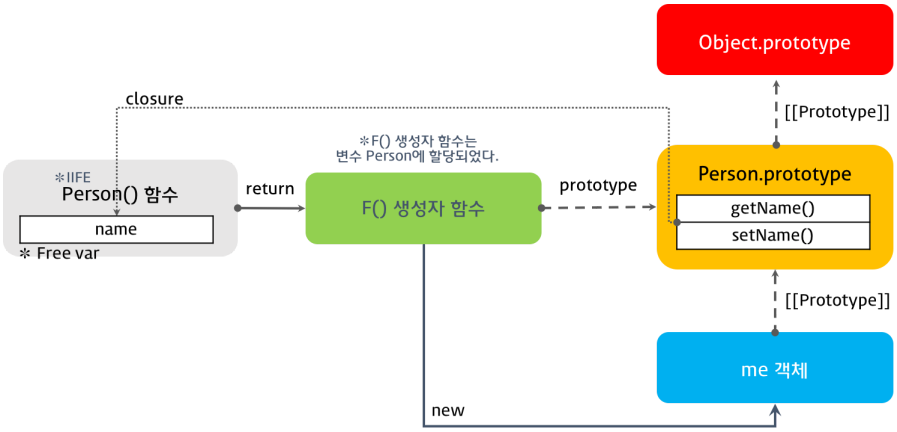


반환된 객체가 함수 person의 프로토타입에 접근할 수 없다는 것은 person을 부모 객체로 상속할 수 없다는 것을 의미한다.

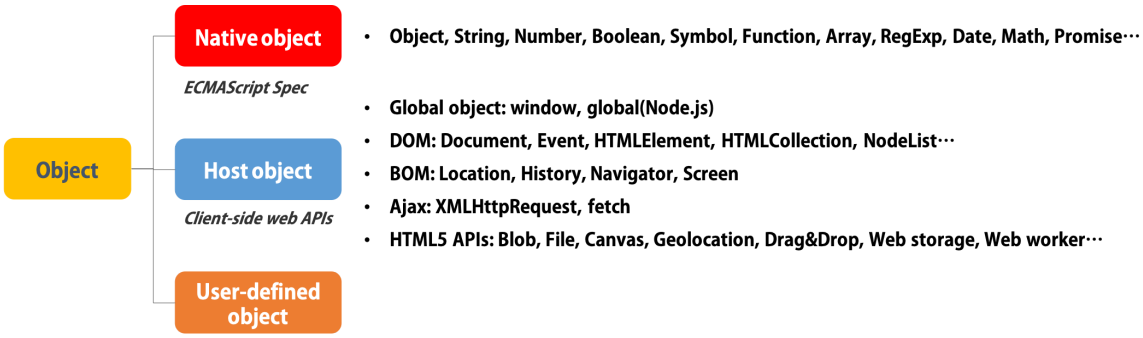
함수 person을 부모 객체로 상속할 수 없다는 것은 함수 person이 반환하는 객체에 모든 메소드를 포함시켜야한다는 것을 의미한다.

이 문제를 해결하기 위해서는 객체를 반환하는 것이 아닌 함수를 반환해야 한다.

```
var Person = function() {
  var name;

  var F = function(arg) { name = arg ? arg : ''; };

  F.prototype = {
    getName: function() {
      return name;
    },
    setName: function(arg) {
      name = arg;
    }
  };

  return F;
}();

var me = new Person('Lee');

console.log(Person.prototype === me.__proto__);

console.log(me.getName());
me.setName('Kim')
console.log(me.getName());
```



빌트인 객체



자바스크립트 객체의 분류

네이티브 객체

Object, String, Number, Function, Array, RegExp, Date, Math와 같은 객체 생성에 관계가 있는 함수 객체와 메소드로 구성된다.

네이티브 객체를 **Global Objects**라고 부르기도 하는데 이것은 전역 객체(Global Object)와 다른 의미로 사용되므로 혼동에 주의하여야 한다.

전역 객체(Global Object)는 모든 객체의 최상위 객체를 의미하며 일반적으로 Browser-side에서는 `window`, Server-side(Node.js)에서는 `global` 객체를 의미한다.

Object

Object() 생성자 함수는 객체를 생성한다. 만약 생성자 인수값이 null이거나 undefined이면 빈 객체를 반환한다. 그 외의경우 생성자 함수의 인수값에 따라 강제 형변환된 객체가 반환된다. 이때 반환된 객체의 [[Prototype]] 프로퍼티에 바인딩된 객체는 Object.prototype이 아니다.

```
// 변수 o에 빈 객체를 저장한다
var o = new Object();
console.log(typeof o + ': ', o);

o = new Object(undefined);
console.log(typeof o + ': ', o);

o = new Object(null);
console.log(typeof o + ': ', o);

// String 객체를 반환한다
// var obj = new String('String');과 동치이다
var obj = new Object('String');
console.log(typeof obj + ': ', obj); //String
console.dir(obj); //String

var strObj = new String('String');
console.log(typeof strObj + ': ', strObj); //String

// Number 객체를 반환한다
```

```
// var obj = new Number(123);과 동치이다
var obj = new Object(123);
console.log(typeof obj + ': ', obj); //123

var numObj = new Number(123);
console.log(typeof numObj + ': ', numObj); //123

// Boolean 객체를 반환한다.
// var obj = new Boolean(true);과 동치이다
var obj = new Object(true);
console.log(typeof obj + ': ', obj); //true

var boolObj = new Boolean(123);
console.log(typeof boolObj + ': ', boolObj); //true

// 객체리터럴을 사용하는 것이 바람직하다.
var o = {};
```

Function

다른 모든 객체들처럼 Function 객체는 new 연산자를 사용해 생성할 수 있다.

```
var adder = new Function('a', 'b', 'return a + b');

adder(2, 6);  // 8
```

Boolean

원시 타입 boolean을 위한 레퍼(wrapper) 객체이다.

```
var foo = new Boolean(true);    // true
var foo = new Boolean('false'); // true

var foo = new Boolean(false); // false
var foo = new Boolean();      // false
var foo = new Boolean('');    // false
var foo = new Boolean(0);     // false
var foo = new Boolean(null);  // false

var x = new Boolean(false);
if (x) { // x는 객체로서 존재한다. 따라서 참으로 간주된다.
  // . . . 이 코드는 실행된다.
}
```

Number

- Number

Math

- Math

Date

- Date

String

- Date

RegExp

- RegExp

Array

- Array

Error

error 객체의 인스턴스는 런타임 에러가 발생하였을 때 throw된다.

```
try {
  // foo();
  throw new Error('Whoops!');
} catch (e) {
  console.log(e.name + ': ' + e.message);
}
```

Error 이외에 Error에 관련한 객체는 아래와 같다.

- EvalError
- InternalError
- RangeError
- ReferenceError
- SyntaxError
- TypeError
- URIError

symbol

유일하고 변경 불가능한 원시 타입으로 Symbol 끄체는 원시타입의 Symbol 값을 생성

원시 타입과 래퍼객체

각 네이티브 객체는 각자의 프로퍼티와 메소드를 가진다. 정적 프로퍼티, 메소드는 해당 인스턴스를 생성하지 않아도 사용할 수 있고 prototype에 속해있는 메소드는 해당 prototype을 상속받은 인스턴스가 있어야만 사용할 수 있다.

```
var str = 'Hello world!';
var res = str.toUpperCase();
console.log(res); // 'HELLO WORLD!'

var num = 1.5;
console.log(num.toFixed()); // 2
```

이는 원시 타입 값에 대해 표준 빌트인 객체의 메소드를 호출할 때 원시타입 값은 연관된 객체(wrapper 객체)로 일시변환 되기 떄문에 가능한 것이다. 그리고 메소드 호출이 종료되면 객체로 변환된 원시 타입 값은 다시 원시 타입값으로 복귀한다.

Wrapper 객체는 String, Number, Boolean이 있다.

호스트 객체

블라우저 환경에서 제공하는 window, CmlHttpRequest,HTMLElement 등의 DOM 노드 객체와 같이 호스트 환겨에 정의된 객체를 말한다. 예를 들어 브라우저에서 동작하는 환경과 브라우저 외부에서 동작하는 환경의 자바스크립트는 다른 호스트 객체를 사용할 수 있다.

전역객체

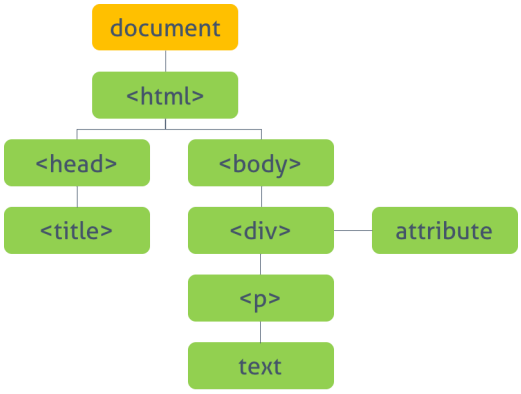
모든 객체의 유일한 최상위 객체를 의미하며 일반적으로 Browser-side에서는 window, Server-side에서는 global 객체를 의미한다.

BOM(Browser Object Model)

브라우저 객체 모델은 브라우저 탭 또는 브라우저 창의 모델을 생성한다. 최상위 객체는 `window`객체로 현재 브라우저 창 또는 탭을 표현하는 객체이다. 또한 이 객체의 자식 객체 들은 브라우저의 다른 기능들을 표현한다. 이 객체들은 Standard Built-in Objects가 구성된 후에 구성된다.

DOM (Document Object Model)

문서 객체 모델은 현재 웹페이지의 모델을 생성한다. 최상위 객체는 `document` 객체로 전체 문서를 표현한다. 또한 이 객체의 자식 객체들은 문서의 다른 요소들을 표현한다. 이 객체들은 Standard Built-in Objects가 구성된 후에 구성된다.



전역 객체

```
// in browser console
this === window // true

// in Terminal
node
this === global // true
```

- 전역 객체는 실행 컨텍스트에 컨트롤이 들어가기 이전에 생성이 되며 constructor가 없기 때문에 new 연산자를 이용하여 새롭게 생성할 수 없다. 즉, 개발자가 전역 객체를 생성하는 것은 불가능하다.
- 전역 객체는 전역 스코프(Global Scope)를 갖게 된다.
- 전역 객체의 자식 객체를 사용할 때 전역 객체의 기술은 생략할 수 있다. 예를 들어 document 객체는 전역 객체 window의 자식 객체로서 window.document...와 같이 기술할 수 있으나 일반적으로 전역 객체의 기술은 생략한다.

```
document.getElementById('foo').style.display = 'none';
// window.document.getElementById('foo').style.display = 'none';
```

• 그러나 사용자가 정의한 변수와 전역 객체의 자식 객체 이름이 충돌하는 경우, 명확히 전역 객체를 기술하여 혼동을 방지할 수 있다.

```
function moveTo(url) {
  var location = {'href':'move to '};
  alert(location.href + url);
  // location.href = url;
```

```
  window.location.href = url;
}
moveTo('http://www.google.com');
```

• 전역 객체는 전역 변수(Global variable)를 프로퍼티로 가지게 된다. 다시 말해 전역 변수는 전역 객체의 프로퍼티이다.

```
var ga = 'Global variable';
console.log(ga);
console.log(window.ga);
```

• 글로벌 영역에 선언한 함수도 전역 객체의 프로퍼티로 접근할 수 있다. 다시 말해 전역 함수는 전역 객체의 메소드이다.

```
function foo() {
  console.log('invoked!');
}
window.foo();
```

• Standard Built-in Objects(표준 빌트인 객체)도 역시 전역 객체의 자식 객체이다. 전역 객체의 자식 객체를 사용할 때 전역 객체의 기술은 생략할 수 있으므로 표준 빌트인 객체도 전역 객체의 기술을 생략할 수 있다.

전역 프로퍼티

전역 객체의 프로퍼티를 의미한다. 애플리케이션 전역에서 사용하는 값들을 나타내기 위해 사용한다. 전역 프로퍼티는 간단한 값이ㅣ 대부분이며 다른 프로퍼티나 메소드를 가지고 있지 않다.

Infinity

양/음의 무한대를 나타내는 숫자값 Intinity를 갖는다.

```
console.log(window.Infinity); // Infinity

console.log(3/0);  // Infinity
console.log(-3/0); // -Infinity
console.log(Number.MAX_VALUE * 2); // 1.7976931348623157e+308 * 2
console.log(typeof Infinity); // number
```

NaN

숫자가 아님을 나타내는 숫자값 NaN을 갖는다. Number.NaN 프로퍼티와 같다.

```
console.log(window.NaN); // NaN

console.log(Number('xyz')); // NaN
console.log(1 * 'string');  // NaN
console.log(typeof NaN);    // number
```

undefined

원시 타입 undefined를 값으로 갖는다.

```
console.log(window.undefined); // undefined

var foo;
console.log(foo); // undefined
console.log(typeof undefined); // undefined
```

전역함수

애플리케이션 전역에서 호출할 수 있는 함수로서 전역 객체의 메소드이다.

eval()

매개변수에 전달된 문자열 구문 또는 표현식을 평가 또는 실행한다, 사용자로 부터 입력받은 콘텐츠를 eval()로 실행하는 것은 보안에 매우 취약하다. eval()의 사용은 가급적으로 금지되어야 한다,

```
eval(string)
// string: code 또는 표현식을 나타내는 문자열. 표현식은 존재하는 객체들의 프로퍼티들과 변수들을 포함할 수 있다.
var foo = eval('2 + 2');
console.log(foo); // 4

var x = 5;
var y = 4;
console.log(eval('x * y')); // 20
```

isFinite()

매개변수에 전달된 값이 정상적인 유한수인지 검사하여 그 결과를 Boolean으로 반환한다. 매개변수에 전달된 값이 숫자가 아닌 경우, 숫자로 변환한 후 검사를 수행한다.

```
isFinite(testValue) // testValue: 검사 대상 값
console.log(isFinite(Infinity));  // false
console.log(isFinite(NaN));       // false
console.log(isFinite('Hello'));   // false
console.log(isFinite('2005/12/12'));   // false

console.log(isFinite(0));         // true
console.log(isFinite(2e64));      // true
console.log(isFinite('10'));      // true: '10' → 10
console.log(isFinite(null));      // true: null → 0
```

isFinite(null)은 true를 반환하는데 이것은 null을 숫자로 변환하여 검사를 수행하였기 때문이다.

```
// null이 숫자로 암묵적 강제 형변환이 일어난 경우
Number(null)  // 0
// null이 불리언로 암묵적 강제 형변환이 일어난 경우
Boolean(null) // false
```

isNaN()

매개변수에 전달된 값이 NaN인지 검사하여 그 결과를 Boolean으로 반환한다. 매개변수에 전달된 값이 숫자가 아닌 경우, 숫자로 변환한 후 검사를 수행한다.

```
isNaN(testValue) // testValue: 검사 대상 값

isNaN(NaN)       // true
isNaN(undefined) // true: undefined → NaN
isNaN({})        // true: {} → NaN
isNaN('blabla')  // true: 'blabla' → NaN

isNaN(true)      // false: true → 1
isNaN(null)      // false: null → 0
isNaN(37)        // false

// strings
isNaN('37')      // false: '37' → 37
isNaN('37.37')   // false: '37.37' → 37.37
isNaN('')        // false: '' → 0
isNaN(' ')       // false: ' ' → 0

// dates
isNaN(new Date())            // false: new Date() → Number
isNaN(new Date().toString())  // true:  String → NaN
```

parseFloat()

매개변수에 전달된 문자열을 부동소수점 숫자로 변환하여 반환한다.

```
parseFloat(string)
// string: 변환 대상 문자열

parseFloat('3.14');     // 3.14
parseFloat('10.00');    // 10
parseFloat('34 45 66'); // 34
parseFloat(' 60 ');     // 60
parseFloat('40 years'); // 40
parseFloat('He was 40') // NaN
```

문자열의 첫 숫자만 반환되며 전후 공백은 무시된다. 그리고 첫문자를 숫자로 변환할 수 없다면 NaN을 반환한다.

parseInt()

매개변수에 전달된 문자열을 정수형 숫자로 해석하여 반환한다. 반환값은 언제나 10진수이다.

```
parseInt(string, radix);
// string: 파싱 대상 문자열
// radix: 진법을 나타내는 기수(2 ~ 36, 기본값 10)

parseInt(10);     // 10
parseInt(10.123); // 10

parseInt('10');     // 10
parseInt('10.123'); // 10

parseInt('10', 2);  // 2진수 10 → 10진수 2
parseInt('10', 8);  // 8진수 10 → 10진수 8
parseInt('10', 16); // 16진수 10 → 10진수 16
```

- 진법을 나타내는 기수를 지정하면 첫번째 매개변수에 전달된 문자열을 해당 기수의 숫자로 해석하여 반환한다. 이때 **반환값은 언제나 10진수이다.**
- 기수를 지정하여 10진수 숫자를 해당 기수의 문자열로 변환하여 반환하고 싶을 때는 Number.prototype.toString 메소드를 사용한다.
- 두번째 매개변수에 진법을 나타내는 기수를 지정하지 않더라도 첫번째 매개변수에 전달된 문자열이 “0x” 또는 “0X”로 시작한다면 16진수로 해석하여 반환한다.
- parseInt는 첫번째 매개변수에 전달된 문자열의 첫번째 문자가 해당 지수의 숫자로 변환될 수 없다면 NaN을 반환한다.

```
parseInt('A0'));   // NaN
parseInt('20', 2); // NaN
```

- 하지만 첫번째 매개변수에 전달된 문자열의 두번째 문자부터 해당 진수를 나타내는 숫자가 아닌 문자(예를 들어 2진수의 경우, 2)와 마주치면 이 문자와 계속되는 문자들은 전부 무시되며 해석된 정수값만을 반환한다.

```
parseInt('1A0'));    // 1
parseInt('102', 2)); // 2
parseInt('58', 8);   // 5
parseInt('FG', 16);  // 15
```

encodeURI() / decodeURI()

encodeURI()은 매개변수로 전달된 URI(Uniform Resource Idenrifier)를 인코딩한다. (인코딩 : URI의 문자들을 이스케이프 처리하는 것을 의미한다.)

이스케이프 처리

네트워크를 통해 정보를 공유할 때 어떤 시스템에서도 읽을수 있는 ASCII Character-set로 변환하는 것이다.

이스케이프 처리 이유

URL은 ASCII Character-set으로만 구성되어야 하며 한글을 포함한 대부분의 외국어나 아스키에 정의되지 않은 특수문자의 경우 URL에 포함될 수 없다. 따라서 URL 내에서 의미를 갖고 있는 문자(%, ?, #)나 URL에 올 수 없는 문자(한글, 공백 등) 또는 시스템에 의해 해석될 수 있는 문자(<, >)를 이스케이프 처리하여 야기될 수 있는 문제를 예방하기 위함이다.

```
encodeURI(URI)
// URI: 완전한 URI
decodeURI(encodedURI)
// encodedURI: 인코딩된 완전한 URI

var uri = 'http://example.com?name=이웅모&job=programmer&teacher';
var enc = encodeURI(uri);
var dec = decodeURI(enc);
console.log(enc);
// http://example.com?name=%EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher
console.log(dec);
// http://example.com?name=이웅모&job=programmer&teacher
```

encodeURIComponent() / decodeURIComponent()

encodeURIComponent()은 매개변수로 전달된 URI(Uniform Resource Identifier) component(구성 요소)를 인코딩한다. 여기서 인코딩이란 URI의 문자들을 이스케이프 처리하는 것을 의미한다.

decodeURIComponent()은 매개변수로 전달된 URI component(구성 요소)를 디코딩한다.

encodeURIComponent()는 인수를 쿼리스트링의 일부라고 간주한다. 따라서 =, ?, &를 인코딩한다. 반면 encodeURI()는 인수를 URI 전체라고 간주하며 파라미터 구분자인 =, ?, &를 인코딩하지 않는다.

```
var uriComp = '이웅모&job=programmer&teacher';

// encodeURI / decodeURI
var enc = encodeURI(uriComp);
var dec = decodeURI(enc);
console.log(enc);
// %EC%9D%B4%EC%9B%85%EB%AA%A8&job=programmer&teacher
console.log(dec);
// 이웅모&job=programmer&teacher

// encodeURIComponent / decodeURIComponent
enc = encodeURIComponent(uriComp);
dec = decodeURIComponent(enc);
console.log(enc);
// %EC%9D%B4%EC%9B%85%EB%AA%A8%26job%3Dprogrammer%26teacher
console.log(dec);
// 이웅모&job=programmer&teacher
```

Number 래퍼 객체

Number 객체는 원시 타입 numbeer를 다룰 때 유용한 프로퍼티와 메소드를 제공하는 레퍼(wrapper)객체이다. 변수 또는 객체의 프로퍼티가 숫자를 값으로 가지고 있다면 Number 객체의 별도 생성없이 Number 객체의 프로퍼티와 메소드를 사용할 수 있다. 원시 타입이 wrapper 객체의 메소드를 사용할 수 있는 이유는 원시 타입으로 프로퍼티나 메소드를 호출할 때 원시 타입과 연관된 wrapper 객체로 일시적으로 변환되어 프로토타입 객체를 공유하게 되기 때문이다.

Number Constructor

Number 객체는 Number() 생성자 함수를 통해 생성할 수 있다.

만일 인자가 숫자로 변환될 수 없다면 NaN을 반환한다.

```
new Number(value);

var x = new Number(123);
var y = new Number('123');
var z = new Number('str');

console.log(x); // 123
console.log(y); // 123
console.log(z); // NaN
```

Number() 생성자 함수를 new 연산자를 붙이지 않아 생성자로 사용하지 않으면 Number 객체를 반환하지 않고 원시 타입 숫자를 반환한다. 이때 형 변환이 발생할 수 있다.

```
var x = Number('123');

console.log(typeof x, x); // number 123

var x = 123;
var y = new Number(123);

console.log(x == y);  // true
console.log(x === y); // false

console.log(typeof x); // number
console.log(typeof y); // object
```

Number Property

정적 프로퍼티로 Number 객체를 생성할 필요없이 `Number.propertyName`의 형태로 사용한다.

Number.EPSILON

JavaScript에서 표현할 수 있는 가장 작은 수이다. 이는 임의의 수와 그 수 보다 큰 수 중 가장 작은 수와의 차이와 같다.

Number.EPSILON은 약 2.220446049250313080847263336181 6E-16 또는 2-52(2의 -52제곱)이다.

```
console.log(0.1 + 0.2);        // 0.30000000000000004
console.log(0.1 + 0.2 == 0.3); // false!!!

function isEqual(a, b){
  // Math.abs는 절댓값을 반환한다.
  // 즉 a와 b의 차이가 JavaScript에서 표현할 수 있는 가장 작은 수인 Number.EPSILON보다 작으면 같은 수로 인정할 수 있다.
  return Math.abs(a - b) < Number.EPSILON;
}

console.log(isEqual(0.1 + 0.2, 0.3));
```

Number.MAX_VALUE

자바스크립트에서 사용 가능한 가장 큰 숫자(1.7976931348623157e+308)를 반환한다. MAX_VALUE보다 큰 숫자는 Infinity 이다.

```
Number.MAX_VALUE; // 1.7976931348623157e+308

var num = 10;
num.MAX_VALUE;    // undefined

console.log(Infinity > Number.MAX_VALUE); // true
```

Number.MIN_VALUE

자바스크립트에서 사용 가능한 가장 작은 숫자(5e-324)를 반환한다.MIN_VALUE는 0에 가장 가까운 양수 값이다. MIN_VALUE보다 작은 숫자는 0으로 변환된다.

```
Number.MIN_VALUE; // 5e-324

var num = 10;
num.MIN_VALUE;    // undefined

console.log(Number.MIN_VALUE > 0); // true
```

Number.POSITIVE_INFINITY

양의 무한대 `Intinity`를 반환한다.

```
Number.POSITIVE_INFINITY // Infinity

var num = 10;
num.POSITIVE_INFINITY;   // undefined
```

Number.NEGATIVE_INFINITY

음의 무한대 `-Infinity`를 반환한다.

```
Number.NEGATIVE_INFINITY // -Infinity

var num = 10;
num.NEGATIVE_INFINITY;   // undefined
```

Number.NaN

숫자가 아님을 나타내는 숫자값이다. Number.NaN 프로퍼티는 window.NaN프로퍼티와 같다.

```
console.log(Number('xyz')); // NaN
console.log(1 * 'string');  // NaN
console.log(typeof NaN);    // number
```

Number Method

Number.isFinite(testValue:number):boolean

매개변수에 전달된 값이 정상적인 유한수인지를 검사하여 그 결과를 Boolean으로 반환한다.

```
/**
 * @param {any} testValue - 검사 대상 값. 암묵적 형변환되지 않는다.
 * @return {boolean}
 */
Number.isFinite(testValue)

Number.isFinite(Infinity)  // false
Number.isFinite(NaN)       // false
Number.isFinite('Hello')   // false
Number.isFinite('2005/12/12')   // false

Number.isFinite(0)         // true
Number.isFinite(2e64)      // true
Number.isFinite(null)      // false. isFinite(null) => true
```

Number.isFinite()는 전역 함수 isFinite()와 차이가 있다. 전역 함수 isFinite()는 인수를 숫자로 변환하여 검사를 수행하지만 Number.isFinite()는 인수를 변환하지 않는다. 따라서 숫자가 아닌 인수가 주어졌을 때 반환값은 언제나 false가 된다.

Number.isInteger(testValue:number):boolean

매개변수에 전달된 값이 정수인지 검사하여 그 결과를 Boolean으로 반환한다. 검사전에 인수를 숫자로 변환하지 않는다.

```
/**
 * @param {any} testValue - 검사 대상 값. 암묵적 형변환되지 않는다.
 * @return {boolean}
 */
Number.isInteger(testValue)

Number.isInteger(123)   //true
Number.isInteger(-123)  //true
Number.isInteger(5-2)   //true
Number.isInteger(0)     //true
Number.isInteger(0.5)   //false
Number.isInteger('123') //false
Number.isInteger(false) //false
Number.isInteger(Infinity)  //false
Number.isInteger(-Infinity) //false
Number.isInteger(0 / 0) //false
```

Number.isNaN(testValue: number):boolean

매개변수에 전달된 값이 NaN인지를 검사하여 그 결과를 Boolean으로 반환한다.

```
/**
 * @param {any} testValue - 검사 대상 값. 암묵적 형변환되지 않는다.
 * @return {boolean}
 */
Number.isNaN(testValue)

Number.isNaN(NaN)       // true
Number.isNaN(undefined) // false. undefined → NaN. isNaN(undefined) → true.
Number.isNaN({})        // false. {} → NaN.        isNaN({}) → true.
Number.isNaN('blabla')  // false. 'blabla' → NaN.  isNaN('blabla') → true.

Number.isNaN(true)      // false
Number.isNaN(null)      // false
Number.isNaN(37)        // false
Number.isNaN('37');     // false
Number.isNaN('37.37');  // false
Number.isNaN('');       // false
Number.isNaN(' ');      // false
Number.isNaN(new Date())            // false
Number.isNaN(new Date().toString())  // false. String → NaN. isNaN(String) → true.
```

Number.isNaN()는 전역 함수 isNaN()와 차이가 있다. 전역 함수 isNaN()는 인수를 숫자로 변환하여 검사를 수행하지만 Number.isNaN()는 인수를 변환하지 않는다. 따라서 숫자가 아닌 인수가 주어졌을 때 반환값은 언제나 false가 된다.

Number.isSafeInteger(testValue: number): boolean

매개변수에 전달된 값이 안전한(safe) 정수값인지 검사하여 그 결과를 Boolean으로 반환한다. 안전한 정수값은 -(253(2의 53제곱)- 1)와 253(2의 53재곱)- 1 사이의 정수값이다. 검사전에 인수를 숫자로 변환하지 않는다.

```
/**
 * @param {any} testValue - 검사 대상 값. 암묵적 형변환되지 않는다.
 * @return {boolean}
 */
Number.isSafeInteger(testValue)

Number.isSafeInteger(123)   //true
Number.isSafeInteger(-123)  //true
Number.isSafeInteger(5-2)   //true
Number.isSafeInteger(0)     //true
Number.isSafeInteger(1000000000000000)  // true
Number.isSafeInteger(10000000000000001) // false
Number.isSafeInteger(0.5)   //false
Number.isSafeInteger('123') //false
Number.isSafeInteger(false) //false
Number.isSafeInteger(Infinity)  //false
Number.isSafeInteger(-Infinity) //false
Number.isSafeInteger(0 / 0) //false
```

Number.prototype.toExponential(fractionDigits?: number): string

대상을 지수 표기법으로 변환하여 문자열로 반환한다. 지수 표기법이란 매우 큰 숫자를 표기할 때 주로 사용하며 e(Exponent) 앞에 있는 숫자에 10의 n승이 곱하는 형식으로 수를 나타내는 방식이다.

```
/**
 * @param {number} [fractionDigits] - 0~20 사이의 정수값으로 소숫점 이하의 자릿수를 나타낸다. 옵션으로 생략 가능하다.
 * @return {string}
 */
numObj.toExponential([fractionDigits])

var numObj = 77.1234;

console.log(numObj.toExponential());  // logs 7.71234e+1
console.log(numObj.toExponential(4)); // logs 7.7123e+1
console.log(numObj.toExponential(2)); // logs 7.71e+1
console.log(77.1234.toExponential()); // logs 7.71234e+1
console.log(77.toExponential());      // SyntaxError: Invalid or unexpected token 소수점인지 그냥 점인지 구분이 안가기때문에
console.log(77 .toExponential());     // logs 7.7e+1
```

Number.prototype.toFixed(fractionDigits?: number): stri

매개변수로 지정된 소숫점자리를 반올림하여 문자열로 반환한다.

```
/**
 * @param {number} [fractionDigits] - 0~20 사이의 정수값으로 소숫점 이하 자릿수를 나타낸다. 기본값은 0이며 옵션으로 생략 가능하다.
 * @return {string}
 */
numObj.toFixed([fractionDigits])

var numObj = 12345.6789;

// 소숫점 이하 반올림
console.log(numObj.toFixed());   // '12346'
// 소숫점 이하 1자리수 유효, 나머지 반올림
console.log(numObj.toFixed(1));  // '12345.7'
// 소숫점 이하 2자리수 유효, 나머지 반올림
console.log(numObj.toFixed(2));  // '12345.68'
// 소숫점 이하 3자리수 유효, 나머지 반올림
console.log(numObj.toFixed(3));  // '12345.679'
// 소숫점 이하 6자리수 유효, 나머지 반올림
console.log(numObj.toFixed(6));  // '12345.678900'
```

Number.prototype.toPrecision(precision?: number): string

매개변수로 지정된 전체 자릿수까지 유효하도록 나머지 자릿수를 반올림하여 문자열로 반환한다. 지정된 전체 자릿수로 표현할 수 없는 경우 지수 표기법으로 결과를 반환한다.

```
/**
 * @param {number} [precision] - 1~21 사이의 정수값으로 전체 자릿수를 나타낸다. 옵션으로 생략 가능하다.
 * @return {string}
 */
```

```
numObj.toPrecision([precision])

var numObj = 15345.6789;

// 전체자리수 유효
console.log(numObj.toPrecision());   // '12345.6789'
// 전체 1자리수 유효, 나머지 반올림
console.log(numObj.toPrecision(1));  // '2e+4'
// 전체 2자리수 유효, 나머지 반올림
console.log(numObj.toPrecision(2));  // '1.5e+4'
// 전체 3자리수 유효, 나머지 반올림
console.log(numObj.toPrecision(3));  // '1.53e+4'
// 전체 6자리수 유효, 나머지 반올림
console.log(numObj.toPrecision(6));  // '12345.7'
```

Number.prototype.toString(radix?: number): string

숫자를 문자열로 변환하여 반환한다.

```
/**
 * @param {number} [radix] - 2~36 사이의 정수값으로 진법을 나타낸다. 옵션으로 생략 가능하다.
 * @return {string}
 */
numObj.toString([radix])

var count = 10;
console.log(count.toString());   // '10'
console.log((17).toString());    // '17'
console.log(17 .toString());     // '17'
console.log((17.2).toString());  // '17.2'

var x = 16;
console.log(x.toString(2));       // '10000'
console.log(x.toString(8));       // '20'
console.log(x.toString(16));      // '10'

console.log((254).toString(16));  // 'fe'
console.log((-10).toString(2));   // '-1010'
console.log((-0xff).toString(2)); // '-11111111'
```

Number.prototype.valueOf(): number

Number 객체의 원시 타입 값(primitive value)을 반환한다.

```
var numObj = new Number(10);
console.log(typeof numObj); // object

var num = numObj.valueOf();
console.log(num);           // 10
console.log(typeof num);    // number
```

Math 객체

Math Property

Math.PI

PI 값(π ≈ 3.141592653589793)을 반환한다.

```
Math.PI; // 3.141592653589793
```

Math Method

Math.abs(x: number): number

인수의 절댓값(absolute value)을 반환한다. 절댓값은 반드시 0 또는 양수이어야 한다.

```
Math.abs(-1);       // 1
Math.abs('-1');     // 1
```

```
Math.abs('');       // 0
Math.abs([]);       // 0
Math.abs(null);     // 0
Math.abs(undefined);// NaN
Math.abs({});       // NaN
Math.abs('string'); // NaN
Math.abs();         // NaN
```

Math.round(x:number):number

인수의 소수점 이하를 반올림한 정수를 반환한다.

```
Math.round(1.4);  // 1
Math.round(1.6);  // 2
Math.round(-1.4); // -1
Math.round(-1.6); // -2
Math.round(1);    // 1
Math.round();     // NaN
```

Math.ceil(x: number): number

인수의 소수점 이하를 올림한 정수를 반환한다.

```
Math.ceil(1.4);  // 2
Math.ceil(1.6);  // 2
Math.ceil(-1.4); // -1
Math.ceil(-1.6); // -1
Math.ceil(1);    // 1
Math.ceil();     // NaN
```

Math.floor(x: number): number

인수의 소수점 이하를 내림한 정수를 반환한다. Math.ceil의 반대 개념이다.

- 양수인 경우, 소수점 이하를 떼어 버린 다음 정수를 반환한다.
- 음수인 경우, 소수점 이하를 떼어 버린 다음 -1을 한 정수를 반환한다.

```
Math.floor(1.9);  // 1
Math.floor(9.1);  // 9
Math.floor(-1.9); // -2
Math.floor(-9.1); // -10
Math.floor(1);    // 1
Math.floor();     // NaN
```

Math.sqrt(x: number): number

인수의 제곱근을 반환한다.

```
Math.sqrt(9);  // 3
Math.sqrt(-9); // NaN
Math.sqrt(2);  // 1.414213562373095
Math.sqrt(1);  // 1
Math.sqrt(0);  // 0
Math.sqrt();   // NaN
```

Math.random(): number

```
Math.random(); // 0 ~ 1 미만의 부동 소수점 (0.8208720231391746)

// 1 ~ 10의 랜덤 정수 취득
// 1) Math.random로 0 ~ 1 미만의 부동 소수점을 구한 다음, 10을 곱해 0 ~ 10 미만의 부동 소수점을 구한다.
// 2) 0 ~ 10 미만의 부동 소수점에 1을 더해 1 ~ 10까지의 부동 소수점을 구한다.
// 3) Math.floor으로 1 ~ 10까지의 부동 소수점의 소수점 이하를 떼어 버린 다음 정수를 반환한다.
const random = Math.floor((Math.random() * 10) + 1);
console.log(random); // 1 ~ 10까지의 정수
```

Math.pow(x: number, y: number): number

첫번째 인수를 밑(base), 두번째 인수를 지수(exponent)로하여 거듭제곱을 반환한다.

```
Math.pow(2, 8);  // 256
Math.pow(2, -1); // 0.5
Math.pow(2);     // NaN

// ES7(ECMAScript 2016) Exponentiation operator(거듭 제곱 연산자)
2 ** 8; // 256
```

Math.max(…values: number[]): number

인수 중에서 가장 큰 수를 반환한다.

```
Math.max(1, 2, 3); // 3

// 배열 요소 중에서 최대값 취득
const arr = [1, 2, 3];
const max = Math.max.apply(null, arr); // 3

// ES6 Spread operator
Math.max(...arr); // 3
```

Math.min(…values: number[]): number

인수 중에서 가장 작은 수를 반환한다.

```
Math.min(1, 2, 3); // 1

// 배열 요소 중에서 최소값 취득
const arr = [1, 2, 3];
const min = Math.min.apply(null, arr); // 1

// ES6 Spread operator
Math.min(...arr); // 1
```

Date 객체

날짜와 시간을 위한 메소드를 제공하는 빌트인 객체이면서 생성자 함수이다.

Date Constructor

Date 객체는 생성자 함수이다. Date 생성자 함수는 날짜와 시간을 가지는 인스턴스를 생성한다. 생성된 인스턴스는 기본적으로 현재 날짜와 시간을 나타내는 값을 가진다. 현재 날짜와 시간이 아닌 다른 날짜와 시간을 다루고 싶은 경우, Date 생성자 함수에 명시적으로 해당 날짜와 시간 정보를 인수로 지정한다.

new Date()

인수를 전달하지 않으면 현재 날짜와 시간을 가지는 인스턴스를 반환한다.

```
const date = new Date();
console.log(date); // Thu May 16 2019 17:16:13 GMT+0900 (한국 표준시)
```

new Date(milliseconds)

인수로 숫자 타입의 밀리초를 전달하면 1970년 1월1일 00:00(UTC)을 기점으로 인수로 전달된 밀리초만큼 경과한 날짜와 시간을 가지는 인스턴스를 반환한다.

```
// KST(Korea Standard Time)는 GMT(그리니치 평균시: Greenwich Mean Time)에 9시간을 더한 시간이다.
let date = new Date(0);
console.log(date); // Thu Jan 01 1970 09:00:00 GMT+0900 (한국 표준시)

// 86400000ms는 1day를 의미한다.
// 1s = 1,000ms
// 1m = 60s * 1,000ms = 60,000ms
// 1h = 60m * 60,000ms = 3,600,000ms
// 1d = 24h * 3,600,000ms = 86,400,000ms
date = new Date(86400000);
console.log(date); // FFri Jan 02 1970 09:00:00 GMT+0900 (한국 표준시)
```

new Date(dateString)

인수로 날짜와 시간을 나타내는 문자열을 전달하면 지정된 날짜와 시간을 가지는 인스턴스를 반환한다. 이때 인수로 전달한 문자열을 Date.parse메소드에 의해 해석 가능한 형식이어야 한다.

```
let date = new Date('May 16, 2019 17:22:10');
console.log(date); // Thu May 16 2019 17:22:10 GMT+0900 (한국 표준시)

date = new Date('2019/05/16/17:22:10');
console.log(date); // Thu May 16 2019 17:22:10 GMT+0900 (한국 표준시)
```

new Date(year, month[,day,hour,minute,second,milisecond])

인수로 년,월,일,시,분,초,밀리초를 의미하는 숫자를 전달하면 지정된 날짜와 시간을 가지는 인스턴스를 반환한다. 이때 년,월은 반드시 지정하여야 한다. 지정하지 않은 옵션 정보는 0 또는 1으로 초기화된다.

인수	내용
year	1990년 이후의 년
month	월을 나타내는 0~ 11까지의 정수 (주의: 0 부터 시작, 0=1월)
day	일을 나타내는 1~31까지의 정수
hour	시를 나타내는 0~23까지의 정수
minute	분을 나타내는 0~59까지의 정수
second	초를 나타내는 0~59까지의 정수
milisecond	밀리초를 나타내는 0~999까지의 정수

```
// 월을 나타내는 4는 5월을 의미한다.
// 2019/5/1/00:00:00:00
let date = new Date(2019, 4);
console.log(date); // Wed May 01 2019 00:00:00 GMT+0900 (한국 표준시)

// 월을 나타내는 4는 5월을 의미한다.
// 2019/5/16/17:24:30:00
date = new Date(2019, 4, 16, 17, 24, 30, 0);
console.log(date); // Thu May 16 2019 17:24:30 GMT+0900 (한국 표준시)

// 가독성이 훨씬 좋다.
date = new Date('2019/5/16/17:24:30:10');
console.log(date); // Thu May 16 2019 17:24:30 GMT+0900 (한국 표준시)
```

Date 생성자 함수를 new 연산자없이 호출

Date 생성자 함수를 new 연산자없이 호출하면 인스턴스를 반환하지 않고 결과값을 문자열로 반환한다.

```
let date = Date();
console.log(typeof date, date); // string Thu May 16 2019 17:33:03 GMT+0900 (한국 표준시)
```

Date 메소드

Date.now

1970년 1월 1일 00:00:00(UTC)을 기점으로 현재 시간까지 경과한 밀리초를 숫자로 반환한다.

```
const now = Date.now();
console.log(now);
```

Date.parse

1970년 1월 1일 00:00:00(UTC)을 기점으로 인수로 전달된 지정 시간(new Date(dateString)의 인수와 동일한 형식)까지의 밀리초를 숫자로 반환한다.

```
let d = Date.parse('Jan 2, 1970 00:00:00 UTC'); // UTC
console.log(d); // 86400000

d = Date.parse('Jan 2, 1970 09:00:00'); // KST
console.log(d); // 86400000

d = Date.parse('1970/01/02/09:00:00'); // KST
console.log(d); // 86400000
```

Date.UTC

1970년 1월1일 00:00:00(UTC)을 기점으로 인수로 전달된 지정 시간까지의 밀리초를 숫자로 반환한다.

Date.UTC 메소드는 `new Date(year, month[, day, hour, minute, second, millisecond])`와 같은 형식의 인수를 사용해야 한다. Date.UTC 메소드의 인수는 local time(KST)가 아닌 UTC로 인식된다.

```
let d = Date.UTC(1970, 0, 2);
console.log(d); // 86400000

d = Date.UTC('1970/1/2');
console.log(d); // NaN
```

month는 월을 의미하는 0~11까지의 정수이다. 0부터 시작하므로 주의가 필요하다

Date.prototyoe.getFullYear

년도를 나타내는 4자리 숫자를 반환한다.

```
const today = new Date();
const year = today.getFullYear();

console.log(today); // Thu May 16 2019 17:39:30 GMT+0900 (한국 표준시)
console.log(year);  // 2019
```

Date.prototype.setFullYear

년도를 나타내는 4자리 숫자를 설정한다. 년도 이외 월, 일도 설정할 수 있다.

```
dateObj.setFullYear(year[, month[, day]])

const today = new Date();

// 년도 지정
today.setFullYear(2000);

let year = today.getFullYear();
console.log(today); // Tue May 16 2000 17:42:40 GMT+0900 (한국 표준시)
console.log(year);  // 2000

// 년도 지정
today.setFullYear(1900, 0, 1);

year = today.getFullYear();
console.log(today); // Mon Jan 01 1900 17:42:40 GMT+0827 (한국 표준시)
console.log(year);  // 1900
```

Date.prototype.getMonth

월을 나타내는 0 ~ 11의 정수를 반환한다. 1월은 0, 12월은 11이다.

```
const today = new Date();
const month = today.getMonth();

console.log(today); // Thu May 16 2019 17:44:03 GMT+0900 (한국 표준시)
console.log(month); // 4
```

Date.prototype.setMonth

월을 나타내는 0 ~ 11의 정수를 설정한다. 1월은 0, 12월은 11이다. 월 이외 일도 설정할 수 있다.

```
dateObj.setMonth(month[, day])

const today = new Date();

// 월을 지정
today.setMonth(0); // 1월

let month = today.getMonth();
console.log(today); // Wed Jan 16 2019 17:45:20 GMT+0900 (한국 표준시)
console.log(month); // 0

// 월/일을 지정
today.setMonth(11, 1); // 12월 1일

month = today.getMonth();
console.log(today); // Sun Dec 01 2019 17:45:20 GMT+0900 (한국 표준시)
console.log(month); // 11
```

Date.prototype.getDate

날짜(1 ~ 31)를 나타내는 정수를 반환한다.

```
const today = new Date();
const date = today.getDate();

console.log(today); // Thu May 16 2019 17:46:42 GMT+0900 (한국 표준시)
console.log(date);  // 16
```

Date.prototype.setDate

날짜(1 ~ 31)를 나타내는 정수를 설정한다.

```
const today = new Date();

// 날짜 지정
today.setDate(1);

const date = today.getDate();
console.log(today); // Wed May 01 2019 17:47:01 GMT+0900 (한국 표준시)
console.log(date);  // 1
```

Date.prototype.getDay

요일(0 ~ 6)를 나타내는 정수를 반환한다. 반환값은 아래와 같다.

요일	반환값
일요일	0
월요일	1
화요일	2
수요일	3
목요일	4
금요일	5
토요일	6

```
const today = new Date();
const day = today.getDay();

console.log(today); // Thu May 16 2019 17:47:31 GMT+0900 (한국 표준시)
console.log(day);   // 4
```

Date.prototype.getHours

시간(0 ~ 23)를 나타내는 정수를 반환한다.

```
const today = new Date();
const hours = today.getHours();
```

```
console.log(today); // Thu May 16 2019 17:48:03 GMT+0900 (한국 표준시)
console.log(hours); // 17
```

Date.prototype.setHours

시간(0 ~ 23)를 나타내는 정수를 설정한다. 시간 이외 분, 초, 밀리초도 설정할 수 있다.

```
dateObj.setHours(hour[, minute[, second[, ms]]])

const today = new Date();

// 시간 지정
today.setHours(7);

let hours = today.getHours();
console.log(today); // Thu May 16 2019 07:49:06 GMT+0900 (한국 표준시)
console.log(hours); // 7

// 시간/분/초/밀리초 지정
today.setHours(0, 0, 0, 0); // 00:00:00:00

hours = today.getHours();
console.log(today); // Thu May 16 2019 00:00:00 GMT+0900 (한국 표준시)
console.log(hours); // 0
```

Date.prototype.getMinutes

분(0 ~ 59)를 나타내는 정수를 반환한다.

```
const today = new Date();
const minutes = today.getMinutes();

console.log(today);   // Thu May 16 2019 17:50:29 GMT+0900 (한국 표준시)
console.log(minutes); // 50
```

Date.prototype.setMinutes

분(0 ~ 59)를 나타내는 정수를 설정한다. 분 이외 초, 밀리초도 설정할 수 있다.

```
dateObj.setMinutes(minute[, second[, ms]])

const today = new Date();

// 분 지정
today.setMinutes(50);

let minutes = today.getMinutes();
console.log(today);   // Thu May 16 2019 17:50:30 GMT+0900 (한국 표준시)
console.log(minutes); // 50

// 분/초/밀리초 지정
today.setMinutes(5, 10, 999); // HH:05:10:999

minutes = today.getMinutes();
console.log(today);   // Thu May 16 2019 17:05:10 GMT+0900 (한국 표준시)
console.log(minutes); // 5
```

Date.prototype.getSeconds

초(0 ~ 59)를 나타내는 정수를 반환한다.

```
const today = new Date();
const seconds = today.getSeconds();

console.log(today);   // Thu May 16 2019 17:53:17 GMT+0900 (한국 표준시)
console.log(seconds); // 17
```

Date.prototype.setSeconds

초(0 ~ 59)를 나타내는 정수를 설정한다. 초 이외 밀리초도 설정할 수 있다.

```
dateObj.setSeconds(second[, ms])

const today = new Date();

// 초 지정
today.setSeconds(30);

let seconds = today.getSeconds();
console.log(today);   // Thu May 16 2019 17:54:30 GMT+0900 (한국 표준시)
console.log(seconds); // 30

// 초/밀리초 지정
today.setSeconds(10, 0); // HH:MM:10:000

seconds = today.getSeconds();
console.log(today);   // Thu May 16 2019 17:54:10 GMT+0900 (한국 표준시)
console.log(seconds); // 10
```

Date.prototype.getMilliseconds

밀리초(0 ~ 999)를 나타내는 정수를 반환한다.

```
const today = new Date();
const ms = today.getMilliseconds();

console.log(today); // Thu May 16 2019 17:55:02 GMT+0900 (한국 표준시)
console.log(ms);    // 905
```

Date.prototype.setMilliseconds

밀리초(0 ~ 999)를 나타내는 정수를 설정한다.

```
const today = new Date();

// 밀리초 지정
today.setMilliseconds(123);

const ms = today.getMilliseconds();
console.log(today); // Thu May 16 2019 17:55:45 GMT+0900 (한국 표준시)
console.log(ms);    // 123
```

Date.prototype.getTime

1970년 1월 1일 00:00:00(UTC)를 기점으로 현재 시간까지 경과된 밀리초를 반환한다.

```
const today = new Date();
const time = today.getTime();

console.log(today); // Thu May 16 2019 17:56:08 GMT+0900 (한국 표준시)
console.log(time);  // 1557996968335
```

Date.prototype.setTime

1970년 1월 1일 00:00:00(UTC)를 기점으로 현재 시간까지 경과된 밀리초를 설정한다.

```
dateObj.setTime(time)

const today = new Date();

// 1970년 1월 1일 00:00:00(UTC)를 기점으로 현재 시간까지 경과된 밀리초 지정
today.setTime(86400000); // 86400000 === 1day

const time = today.getTime();
console.log(today); // Fri Jan 02 1970 09:00:00 GMT+0900 (한국 표준시)
console.log(time);  // 86400000
```

Date.prototype.getTimezoneOffset

UTC와 지정 로케일(Locale) 시간과의 차이를 분단위로 반환한다.

```
const today = new Date();
const x = today.getTimezoneOffset() / 60; // -9

console.log(today); // Thu May 16 2019 17:58:13 GMT+0900 (한국 표준시)
console.log(x);     // -9
```

KST(Korea Standard Time)는 UTC에 9시간을 더한 시간이다. 즉, UTC = KST - 9h이다.

Date.prototype.toDateString

사람이 읽을 수 있는 형식의 문자열로 날짜를 반환한다.

```
const d = new Date('2019/5/16/18:30');

console.log(d.toString());     // Thu May 16 2019 18:30:00 GMT+0900 (한국 표준시)
console.log(d.toDateString()); // Thu May 16 2019
```

Date.prototype.toTimeString

사람이 읽을 수 있는 형식의 문자열로 시간을 반환한다.

```
const d = new Date('2019/5/16/18:30');

console.log(d.toString());     // Thu May 16 2019 18:30:00 GMT+0900 (한국 표준시)
console.log(d.toTimeString()); // 18:30:00 GMT+0900 (한국 표준시)
```

Date Example

현재 날짜와 시간을 초단위로 반복 출력하는 예제이다.

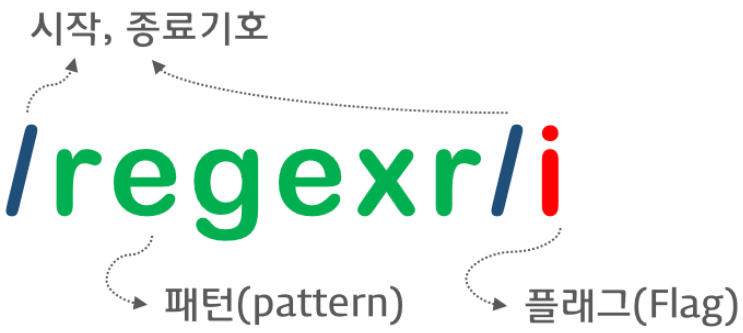
```
(function printNow() {
  const today = new Date();

  const dayNames = ['(일요일)', '(월요일)', '(화요일)', '(수요일)', '(목요일)', '(금요일)', '(토요일)'];
  // getDay: 해당 요일(0 ~ 6)를 나타내는 정수를 반환한다.
  const day = dayNames[today.getDay()];

  const year = today.getFullYear();
  const month = today.getMonth() + 1;
  const date = today.getDate();
  let hour = today.getHours();
  let minute = today.getMinutes();
  let second = today.getSeconds();
  const ampm = hour >= 12 ? 'PM' : 'AM';

  // 12시간제로 변경
  hour %= 12;
  hour = hour || 12; // 0 => 12

  // 10미만인 분과 초를 2자리로 변경
  minute = minute < 10 ? '0' + minute : minute;
  second = second < 10 ? '0' + second : second;

  const now = `${year}년 ${month}월 ${date}일 ${day} ${hour}:${minute}:${second} ${ampm}`;

  console.log(now);
  setTimeout(printNow, 1000);
}());
```

정규표현식

문자열에서 특정 내용을 찾거나 대체 또는 발췌하는데 사용한다. 예를 들어 회원가입 화면에서 사용자로 부터 입력받는 전화번호가 유효한지 체크할 필요가 있다. 이때 정규표현식을 사용하면 간단히 처리할 수 있다.

```
const tel = '0101234567팔';

// 정규 표현식 리터럴
const myRegExp = /^[0-9]+$/;
```

```
console.log(myRegExp.test(tel)); // false
```



정규표현식 리터럴 표기법

정규표현식을 사용하는 자바스크립트 메소드는 RegExp.prototype.exec, RegExp.prototype.test, String.prototype.match, String.prototype.replace, String.prototype.search, String.prototype.split 등이 있다.

```
const targetStr = 'This is a pen.';
const regexr = /is/ig;

// RegExp 객체의 메소드
console.log(regexr.exec(targetStr)); // [ 'is', index: 2, input: 'This is a pen.' ]
console.log(regexr.test(targetStr)); // true

// String 객체의 메소드
console.log(targetStr.match(regexr)); // [ 'is', 'is' ]
console.log(targetStr.replace(regexr, 'IS')); // ThIS IS a pen.
// String.prototype.search는 검색된 문자열의 첫번째 인덱스를 반환한다.
console.log(targetStr.search(regexr)); // 2 ← index
console.log(targetStr.split(regexr));  // [ 'Th', ' ', ' a pen.' ]
```

플래그

Flag	Meaning	Description
i	Ignore Case	대소문자를 구별하지 않고 검색한다.
g	Global	문자열 내의 모든 패턴을 검색한다.
m	Multi Line	문자열의 행이 바뀌더라도 검색을 계속한다.

플래그는 옵션이므로 선택적으로 사용한다. 플래그를 사용하지 않은 경우 문자열 내 검색 매칭 대상이 1개 이상이더라도 첫번째 매칭한 대상만을 검색하고 종료한다.

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

// 문자열 is를 대소문자를 구별하여 한번만 검색한다.
let regexr = /is/;

console.log(targetStr.match(regexr)); // [ 'is', index: 5, input: 'Is this all there is?' ]

// 문자열 is를 대소문자를 구별하지 않고 대상 문자열 끝까지 검색한다.
regexr = /is/ig;

console.log(targetStr.match(regexr)); // [ 'Is', 'is', 'is' ]
console.log(targetStr.match(regexr).length); // 3
```

패턴

패턴에서 검색하고 싶은 문자열을 지정한다. 이때 문자열의 따옴표는 생략한다. 따옴표를 포함하면 따옴표까지도 검색한다. 또한 패턴은 특별한 의미를 가지는 메타문자 또는 기호로 표현할 수 있다.

```
const targetStr = 'AA BB Aa Bb';

// 임의의 문자 3개
const regexr = /.../;
```

`.`은 임의의 문자 한 개를 의미한다. 문자의 내용은 무엇이든지 상관없다. 위 예제의 경우 `.`를 3개 연속하여 패턴을 생성하였으므로 3자리 문자를 추출한다.

```
console.log(targetStr.match(regexr)); // [ 'AA ', index: 0, input: 'AA BB Aa Bb' ]
```

이때 추출을 반복하지 않는다. 반복하기 위해서는 플래그 `g`를 사용한다.

```
const targetStr = 'AA BB Aa Bb';

// 임의의 문자 3개를 반복하여 검색
const regexr = /.../g;

console.log(targetStr.match(regexr)); // [ 'AA ', 'BB ', 'Aa ' ]
```

모든 문자를 선택하려면 `.`와 `g`를 동시에 지정한다.

```
const targetStr = 'AA BB Aa Bb';

// 임의의 한문자를 반복 검색
const regexr = /./g;

console.log(targetStr.match(regexr));
// [ 'A', 'A', ' ', 'B', 'B', ' ', 'A', 'a', ' ', 'B', 'b' ]
```

패턴에 문자 또는 문자열을 지정하면 일치하는 문자 또는 문자열을 추출한다.

```
const targetStr = 'AA BB Aa Bb';

// 'A'를 검색
const regexr = /A/;

console.log(targetStr.match(regexr)); // 'A'
```

이때 대소문자를 구별하며 패턴과 일치한 첫번째 결과만 반환된다. 대소문자를 구별하지 않게 하려면 플래그 `i`를 사용한다.

```
const targetStr = 'AA BB Aa Bb';

// 'A'를 대소문자 구분없이 반복 검색
const regexr = /A/ig;

console.log(targetStr.match(regexr)); // [ 'A', 'A', 'A', 'a' ]
```

앞선 패턴을 최소 한번 반복하려면 앞선 패턴 뒤에 `+`를 붙인다. 아래 예제의 경우, 앞선 패턴은 A이므로 A+는 A만으로 이루어진 문자열('A', 'AA', 'AAA', ...)를 의미한다.

```
const targetStr = 'AA AAA BB Aa Bb';

// 'A'가 한번이상 반복되는 문자열('A', 'AA', 'AAA', ...)을 반복 검색
const regexr = /A+/g;

console.log(targetStr.match(regexr)); // [ 'AA', 'AAA', 'A' ]
```

`|`를 사용하면 or의 의미를 가지게 된다.

```
const targetStr = 'AA BB Aa Bb';

// 'A' 또는 'B'를 반복 검색
const regexr = /A|B/g;

console.log(targetStr.match(regexr)); // [ 'A', 'A', 'B', 'B', 'A', 'B' ]
```

분해되지 않은 단어 레벨로 추출하기 위해서는 `+`를 같이 사용하면 된다.

```
const targetStr = 'AA AAA BB Aa Bb';

// 'A' 또는 'B'가 한번 이상 반복되는 문자열을 반복 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ...
const regexr = /A+|B+/g;

console.log(targetStr.match(regexr)); // [ 'AA', 'AAA', 'BB', 'A', 'B' ]
```

위 예제는 패턴을 or로 한번 이상 반복하는 것인데 간단히 표현하면 아래와 같다. `[]`내의 문자는 or로 동작한다. 그 뒤에 `+`를 사용하여 앞선 패턴을 한번 이상 반복하게 한다.

```
const targetStr = 'AA BB Aa Bb';

// 'A' 또는 'B'가 한번 이상 반복되는 문자열을 반복 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ...
const regexr = /[AB]+/g;

console.log(targetStr.match(regexr)); // [ 'AA', 'BB', 'A', 'B' ]
```

범위를 지정하려면 `[]`내에 `-`를 사용한다. 아래의 경우 대문자 알파벳을 추출한다.

```
const targetStr = 'AA BB ZZ Aa Bb';

// 'A' ~ 'Z'가 한번 이상 반복되는 문자열을 반복 검색
// 'A', 'AA', 'AAA', ... 또는 'B', 'BB', 'BBB', ... ~ 또는 'Z', 'ZZ', 'ZZZ', ...

const regexr = /[A-Z]+/g;

console.log(targetStr.match(regexr)); // [ 'AA', 'BB', 'ZZ', 'A', 'B' ]
```

대소문자를 구별하지 않고 알파벳을 추출하려면 아래와 같이 한다.

```
const targetStr = 'AA BB Aa Bb';

// 'A' ~ 'Z' 또는 'a' ~ 'z'가 한번 이상 반복되는 문자열을 반복 검색
const regexr = /[A-Za-z]+/g;
// 아래와 동일하다.
// const regexr = /[A-Z]+/gi;

console.log(targetStr.match(regexr)); // [ 'AA', 'BB', 'Aa', 'Bb' ]
```

숫자를 추출하는 방법이다.

```
const targetStr = 'AA BB Aa Bb 24,000';

// '0' ~ '9'가 한번 이상 반복되는 문자열을 반복 검색
const regexr = /[0-9]+/g;

console.log(targetStr.match(regexr)); // [ '24', '000' ]
```

컴마 때문에 결과가 분리되므로 패턴에 포함시킨다.

```
const targetStr = 'AA BB Aa Bb 24,000';

// '0' ~ '9' 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
const regexr = /[0-9,]+/g;

console.log(targetStr.match(regexr)); // [ '24,000' ]
```

이것을 간단히 표현하면 아래와 같다. `\d`는 숫자를 의미한다. `\D`는 `\d`와 반대로 동작한다.

```
const targetStr = 'AA BB Aa Bb 24,000';

// '0' ~ '9' 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
let regexr = /[\d,]+/g;

console.log(targetStr.match(regexr)); // [ '24,000' ]
```

```
// '0' ~ '9'가 아닌 문자(숫자가 아닌 문자) 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
regexr = /[\D,]+/g;

console.log(targetStr.match(regexr)); // [ 'AA BB Aa Bb ', ',' ]
```

`\w`는 알파벳과 숫자를 의미한다. `\W`는 `\w`와 반대로 동작한다.

```
const targetStr = 'AA BB Aa Bb 24,000';

// 알파벳과 숫자 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
let regexr = /[\w,]+/g;

console.log(targetStr.match(regexr)); // [ 'AA', 'BB', 'Aa', 'Bb', '24,000' ]

// 알파벳과 숫자가 아닌 문자 또는 ','가 한번 이상 반복되는 문자열을 반복 검색
regexr = /[\W,]+/g;

console.log(targetStr.match(regexr)); // [ ' ', ' ', ' ', ' ', ',' ]
```

자주 사용하는 정규표현식

특정 단어로 시작하는지 검사한다.

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

// 'http'로 시작하는지 검사
// ^ : 문자열의 처음을 의미한다.
const regexr = /^http/;

console.log(regexr.test(url)); // true
```

특정 단어로 끝나는지 검사한다.

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

// 'html'로 끝나는지 검사
// $ : 문자열의 끝을 의미한다.
const regexr = /html$/;

console.log(regexr.test(fileName)); // true
```

숫자인지 검사한다.

```
const targetStr = '12345';

// 모두 숫자인지 검사
// [^]: 부정(not)을 의미한다. 얘를 들어 [^a-z]는 알파벳 소문자로 시작하지 않는 모든 문자를 의미한다.
// [] 바깥의 ^는 문자열의 처음을 의미한다.
const regexr = /^\d+$/;

console.log(regexr.test(targetStr)); // true
```

하나 이상의 공백으로 시작하는지 검사한다.

```
const targetStr = ' Hi!';

// 1개 이상의 공백으로 시작하는지 검사
// \s : 여러 가지 공백 문자 (스페이스, 탭 등) => [\t\r\n\v\f]
const regexr = /^[\s]+/;

console.log(regexr.test(targetStr)); // true
```

아이디로 사용 가능한지 검사한다. (영문자, 숫자만 허용, 4~10자리)

```
const id = 'abc123';

// 알파벳 대소문자 또는 숫자로 시작하고 끝나며 4 ~10자리인지 검사
// {4,10}: 4 ~ 10자리
const regexr = /^[A-Za-z0-9]{4,10}$/;
```

```
console.log(regexr.test(id)); // true
```

메일 주소 형식에 맞는지 검사한다.

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

const regexr = /^[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*@[0-9a-zA-Z]([-_\.]?[0-9a-zA-Z])*\.[a-zA-Z]{2,3}$/;

console.log(regexr.test(email)); // true
```

핸드폰 번호 형식에 맞는지 검사한다.

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

const regexr = /^\d{3}-\d{3,4}-\d{4}$/;

console.log(regexr.test(cellphone)); // true
```

특수 문자 포함 여부를 검사한다.

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

// A-Za-z0-9 이외의 문자가 있는지 검사
let regexr = /[^A-Za-z0-9]/gi;

console.log(regexr.test(targetStr)); // true

// 아래 방식도 동작한다. 이 방식의 장점은 특수 문자를 선택적으로 검사할 수 있다.
regexr = /[\{\}\[\]\/?.,;:|\)*~`!^\-_+<>@\#$%&\\\=\(\'\"]/gi;

console.log(regexr.test(targetStr)); // true

// 특수 문자 제거
console.log(targetStr.replace(regexr, '')); // abc123
```

Javascript Regular Expression

RegExp Constructor

자바스크립트는 정규표현식을 위해 RegExp 객체를 지원한다. RegExp 객체를 생성하기 위해서는 리터럴 방식과 RegExp 생성자 함수를 사용할 수 있다. 일반적인 방법은 리터럴 방식이다.

```
new RegExp(pattern[, flags])
```

- pattern 정규표현식의 텍스트
- flags 정규표현식의 플래그 (g, i, m, u, y)

```
// 정규식 리터럴
/ab+c/i;

new RegExp('ab+c', 'i');

new RegExp(/ab+c/, 'i');

new RegExp(/ab+c/i); // ES6
```

정규표현식을 사용하는 메소드는 RegExp.prototype.exec, RegExp.prototype.test, String.prototype.match, String.prototype.replace, String.prototype.search, String.prototype.split 등이 있다.

RegExp Method

RegExp.prototype.exec(target: string): RegExpExecArray | null

문자열을 검색하여 매칭 결과를 반환한다. 반환값은 배열 또는 null이다.

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

const res = regExp.exec(target);
console.log(res); // [ 'is', index: 5, input: 'Is this all there is?' ]
```

exec 메소드는 g 플래그를 지정하여도 첫번째 메칭 결과만을 반환한다.

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

const res = regExp.exec(target);
console.log(res); // [ 'is', index: 5, input: 'Is this all there is?' ]
```

RegExp.prototype.test(target: string): boolean

문자열을 검색하여 매칭 결과를 반환한다. 반환값은 true 또는 false이다.

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

const res = regExp.test(target);
console.log(res); // true
```

String 래퍼 객체

원시 타입인 문자열을 다룰때 유용한 프로퍼티와 메소드를 제공하는 래퍼(wrapper)객체이다. 변수 또는 객체 프로퍼티가 문자열을 값으로 가지고 있다면 String 객체의 별도 생성없이 String 객체의 프로퍼티와 메소드를 사용할 수있다. 원시 타입이 wrapper 객체의 메소드를 사용할 수 있는 이유는 원시 타입으로 프로퍼티나 메소드를 호출할 때 원시 타입과 연관된 wrapper 객체로 일시적으로 변환되어 프로토타입 객체를 공유하게 되기 때문이다.

```
const str = 'Hello world!';
console.log(str.toUpperCase()); // 'HELLO WORLD!'
```

위에서 원시 타입 문자열을 담고 있는 변수 str이 String.prototype.toUpperCase() 메소드를 호출할 수 있는 것은 변수 str의 값이 일시적으로 wrapper객체로 변환되었기 때문이다.

String Constructor

String 객체는 String 생성자 함수를 통해 생성할 수 있다. 이때 전달된 인자는 모두 문자열로 변환된다.

```
new String(value);

let strObj = new String('Lee');
console.log(strObj); // String {0: 'L', 1: 'e', 2: 'e', length: 3, [[PrimitiveValue]]: 'Lee'}

strObj = new String(1);
console.log(strObj); // String {0: '1', length: 1, [[PrimitiveValue]]: '1'}

strObj = new String(undefined);
console.log(strObj); // String {0: 'u', 1: 'n', 2: 'd', 3: 'e', 4: 'f', 5: 'i', 6: 'n', 7: 'e', 8: 'd', length: 9, [[PrimitiveValu
e]]: 'undefined'}
```

new 연산자를 사용하지 않고 String 생성자 함수를 호출하면 String 객체가 아닌 문자열 리터럴을 반환한다. 이때 형 변환이 발생할 수 있다.

```
var x = String('Lee');

console.log(typeof x, x); // string Lee
```

일반적으로 문자열을 사용할 때는 원시 타입 문자열을 사용한다.

```
const str = 'Lee';
const strObj = new String('Lee');

console.log(str == strObj);  // true
console.log(str === strObj); // false

console.log(typeof str);    // string
console.log(typeof strObj); // object
```

String Property

String.length

문자열 내의 문자 갯수를 반환한다. String 객체는 length 프로퍼티를 소유하고 있으므로 유사 배열 객체이다.

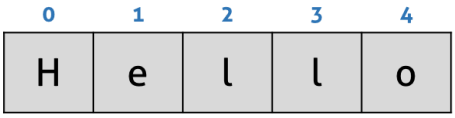
```
const str1 = 'Hello';
console.log(str1.length); // 5

const str2 = '안녕하세요!';
console.log(str2.length); // 6
```

String Method

String 객체의 모든 메소드는 언제나 새로운 문자열을 반환한다. 문자열은 변경 불가능한 원기 값이기 때문이다.

String.prototype.charAt(pos:number): string

인수로 전달한 index를 사용하여 index에 해당하는 위치의 문자를 반환한다. index는 0 ~(문자열 길이-1)사이의 정수이다. 지정한 index가 문자열의 범위(0~(문자열 길이 -1))를 벗어난 경우 빈문자열을 반환한다.




```
const str = 'Hello';

console.log(str.charAt(0)); // H
console.log(str.charAt(1)); // e
console.log(str.charAt(2)); // l
console.log(str.charAt(3)); // l
console.log(str.charAt(4)); // o
// 지정한 index가 범위(0 ~ str.length-1)를 벗어난 경우 빈문자열을 반환한다.
console.log(str.charAt(5)); // ''

// 문자열 순회. 문자열은 length 프로퍼티를 갖는다.
for (let i = 0; i < str.length; i++) {
  console.log(str.charAt(i));
}

// String 객체는 유사 배열 객체이므로 배열과 유사하게 접근할 수 있다.
for (let i = 0; i < str.length; i++) {
  console.log(str[i]); // str['0']
}
```

String.prototype.concat(..strings: string[]); string

인수로 전달한 1개 이상의 문자열과 연결하여 새로운 문자열을 반환한다. concat 메소드를 사용하는 것보다는 `+`, `+=` 할당 연산자를 사용하는 것이 성능상 유리하다.

```
/**
 * @param {...string} str - 연결할 문자열
 * @return {string}
 */
str.concat(str1[,str2,...,strN])

console.log('Hello '.concat('Lee')); // Hello Lee
```

String.prototype.indexOf(searchString:string,fromIndex=0): number

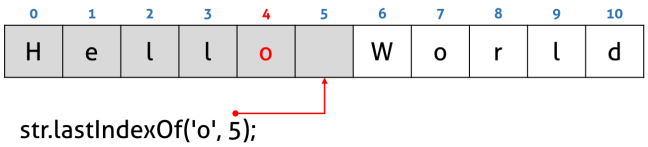
인수로 전달한 문자 또는 문자열에서 검색하여 처음 발견된 곳의 index를 반환한다. 발경하지 못한 경우 -1을 반환한다.

```
/**
 * @param {string} searchString - 검색할 문자 또는 문자열
 * @param {string} [fromIndex=0] - 검색 시작 index (생략할 경우, 0)
 * @return {number}
 */
str.indexOf(searchString[, fromIndex])

const str = 'Hello World';

console.log(str.indexOf('l'));  // 2
console.log(str.indexOf('or')); // 7
console.log(str.indexOf('or' , 8)); // -1

if (str.indexOf('Hello') !== -1) {
  // 문자열 str에 'hello'가 포함되어 있는 경우에 처리할 내용
}

// ES6: String.prototype.includes
if (str.includes('Hello')) {
  // 문자열 str에 'hello'가 포함되어 있는 경우에 처리할 내용
}
```

String.prototype.lastIndexOf(searchString: string,fromIndex=this.length-1): number

인수로 전달한 문자 또는 문자열을 대상 문자열에서 검색하여 마지막으로 발견된 곳의 index를 반환한다. 발견하지 못한 경우 -1을 반환한다. 2번째 인수(fromIndex)가 전달되면 검색 시작 위치를 fromIndex으로 이동하여 역방향으로 검색을 시작한다. 이때 검색 범위는 0 ~ fromIndex이며 반환값은 indexOf 메소드와 동일하게 발견된 곳의 index이다.

```
/**
 * @param {string} searchString - 검색할 문자 또는 문자열
 * @param {number} [fromIndex=this.length-1] - 검색 시작 index (생략할 경우, 문자열 길이 - 1)
 * @return {number}
 */
str.lastIndexOf(searchString[, fromIndex])

const str = 'Hello World';

console.log(str.lastIndexOf('World')); // 6
console.log(str.lastIndexOf('l'));     // 9
console.log(str.lastIndexOf('o', 5));  // 4
console.log(str.lastIndexOf('o', 8));  // 7
console.log(str.lastIndexOf('l', 10)); // 9

console.log(str.lastIndexOf('H', 0));  // 0
console.log(str.lastIndexOf('W', 5));  // -1
console.log(str.lastIndexOf('x', 8));  // -1
```



String.prototype.replace(searchValue: string | RegExp, replaceValue: string | replacer: (substring: string, …args: any[]) => string): string

첫번째 인수로 전달한 문자열 또는 정규표현식을 대상 문자열에서 검색하여 두번째 인수로 전달한 문자열로 대체한다. 원본 문자열은 변경되지 않고 결과가 반영된 새로운 문자열을 반환한다.

검색된 문자열이 여럿 존재할 경우 첫번째로 검색된 문자열만 대체된다.

```
/**
 * @param {string | RegExp} searchValue - 검색 대상 문자열 또는 정규표현식
 * @param {string | Function} replacer - 치환 문자열 또는 치환 함수
 * @return {string}
 */
str.replace(searchValue, replacer)
```

```
const str = 'Hello world';

// 첫번째로 검색된 문자열만 대체하여 새로운 문자열을 반환한다.
console.log(str.replace('world', 'Lee')); // Hello Lee

// 특수한 교체 패턴을 사용할 수 있다. ($& => 검색된 문자열)
console.log(str.replace('world', '<strong>$&</strong>')); // Hello <strong>world</strong>

/* 정규표현식
g(Global): 문자열 내의 모든 패턴을 검색한다.
i(Ignore case): 대소문자를 구별하지 않고 검색한다.
*/
console.log(str.replace(/hello/gi, 'Lee')); // Lee Lee

// 두번째 인수로 치환 함수를 전달할 수 있다.
// camelCase => snake_case
const camelCase = 'helloWorld';

// /.[A-Z]/g => 1문자와 대문자의 조합을 문자열 전체에서 검색한다.
console.log(camelCase.replace(/.[A-Z]/g, function (match) {
  // match : oW => match[0] : o, match[1] : W
  return match[0] + '_' + match[1].toLowerCase();
})); // hello_world

// /(.)([A-Z])/g => 1문자와 대문자의 조합
// $1 => (.)
// $2 => ([A-Z])
console.log(camelCase.replace(/(.)([A-Z])/g, '$1_$2').toLowerCase()); // hello_world

// snake_case => camelCase
const snakeCase = 'hello_world';

// /_./g => _와 1문자의 조합을 문자열 전체에서 검색한다.
console.log(snakeCase.replace(/_./g, function (match) {
  // match : _w => match[1] : w
  return match[1].toUpperCase();
})); // helloWorld
```

String.prototype.split(separator: string | RegExp, limit?: number): string[]

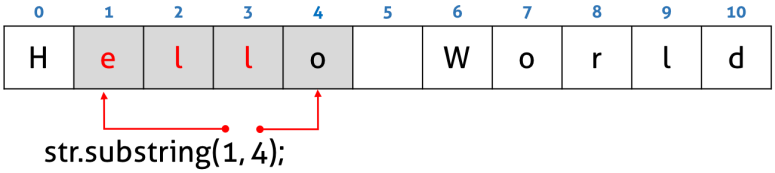
첫번째 인수로 전달한 문자열 또는 정규표현식을 대상 문자열에서 검색하여 문자열을 구분한 후 분리된 각 문자열로 이루어진 배열을 반환한다. 원본 문자열은 변경되지 않는다.

인수가 없는 경우, 대상 문자열 전체를 단일 요소로 하는 배열을 반환한다.

```
/**
 * @param {string | RegExp} [separator] - 구분 대상 문자열 또는 정규표현식
 * @param {number} [limit] - 구분 대상수의 한계를 나타내는 정수
 * @return {string[]}
 */
str.split([separator[, limit]])

const str = 'How are you doing?';

// 공백으로 구분(단어로 구분)하여 배열로 반환한다
console.log(str.split(' ')); // [ 'How', 'are', 'you', 'doing?' ]

// 정규 표현식
console.log(str.split(/\s/)); // [ 'How', 'are', 'you', 'doing?' ]

// 인수가 없는 경우, 대상 문자열 전체를 단일 요소로 하는 배열을 반환한다.
console.log(str.split()); // [ 'How are you doing?' ]

// 각 문자를 모두 분리한다
console.log(str.split('')); // [ 'H','o','w',' ','a','r','e',' ','y','o','u',' ','d','o','i','n','g','?' ]

// 공백으로 구분하여 배열로 반환한다. 단 요소수는 3개까지만 허용한다
console.log(str.split(' ', 3)); // [ 'How', 'are', 'you' ]

// 'o'으로 구분하여 배열로 반환한다.
console.log(str.split('o')); // [ 'H', 'w are y', 'u d', 'ing?' ]
```

String.prototype.substring(start: number, end=this.length): string

첫번째 인수로 전달한 start 인덱스에 해당하는 문자부터 두번째 인자에 전달된 end 인덱스에 해당하는 문자의 **바로 이전 문자까지**를 모두 반환한다. 이때 첫번째 인수 < 두번째 인수의 관계가 성립된다.



substring

- 첫번째 인수 > 두번째 인수 : 두 인수는 교환된다.
- 두번째 인수가 생략된 경우 : 해당 문자열의 끝까지 반환한다.
- 인수 < 0 또는 NaN인 경우 : 0으로 취급된다.
- 인수 > 문자열의 길이(str.length) : 인수는 문자열의 길이(str.length)으로 취급된다.

```
/**
 * @param {number} start - 0 ~ 해당문자열 길이 -1 까지의 정수
 * @param {number} [end=this.length] - 0 ~ 해당문자열 길이까지의 정수
 * @return {string}
 */
str.substring(start[, end])

const str = 'Hello World'; // str.length == 11

console.log(str.substring(1, 4)); // ell

// 첫번째 인수 > 두번째 인수 : 두 인수는 교환된다.
console.log(str.substring(4, 1)); // ell

// 두번째 인수가 생략된 경우 : 해당 문자열의 끝까지 반환한다.
console.log(str.substring(4)); // o World

// 인수 < 0 또는 NaN인 경우 : 0으로 취급된다.
console.log(str.substring(-2)); // Hello World

// 인수 > 문자열의 길이(str.length) : 인수는 문자열의 길이(str.length)으로 취급된다.
console.log(str.substring(1, 12)); // ello World
console.log(str.substring(11)); // ''
console.log(str.substring(20)); // ''
console.log(str.substring(0, str.indexOf(' '))); // 'Hello'
console.log(str.substring(str.indexOf(' ') + 1, str.length)); // 'World'
```

String.prototype.slice(start?: number, end?: number): string

String.prototype.substring과 동일하다. 단, String.prototype.slice는 음수의 인수를 전달할 수 있다.

```
const str = 'hello world';

// 인수 < 0 또는 NaN인 경우 : 0으로 취급된다.
console.log(str.substring(-5)); // 'hello world'
// 뒤에서 5자리를 잘라내어 반환한다.
console.log(str.slice(-5)); // 'world'

// 2번째부터 마지막 문자까지 잘라내어 반환
console.log(str.substring(2)); // llo world
console.log(str.slice(2)); // llo world

// 0번째부터 5번째 이전 문자까지 잘라내어 반환
console.log(str.substring(0, 5)); // hello
console.log(str.slice(0, 5)); // hello
```

String.prototype.toLowerCase(): string

대상 문자열의 모든 문자를 소문자로 변경한다.

```
console.log('Hello World!'.toLowerCase()); // hello world!
```

String.prototype.toUpperCase(): string

대상 문자열의 모든 문자를 대문자로 변경한다.

```
console.log('Hello World!'.toUpperCase()); // HELLO WORLD!
```

String.prototype.trim(): string

대상 문자열 양쪽 끝에 있는 공백 문자를 제거한 문자열을 반환한다.

```
const str = '   foo  ';

console.log(str.trim()); // 'foo'

// String.prototype.replace
console.log(str.replace(/\s/g, ''));   // 'foo'
console.log(str.replace(/^\s+/g, '')); // 'foo  '
console.log(str.replace(/\s+$/g, '')); // '   foo'

// String.prototype.{trimStart,trimEnd} : Proposal stage 3
console.log(str.trimStart()); // 'foo  '
console.log(str.trimEnd());   // '   foo'
```

String.prototype.repeat(count: number): string

인수로 전달한 숫자만큼 반복해 연결한 새로운 문자열을 반환한다. count가 0이면 빈 문자열을 반환하고 음수이면 RangeError를 발생시킨다.

```
console.log('abc'.repeat(0));   // ''
console.log('abc'.repeat(1));   // 'abc'
console.log('abc'.repeat(2));   // 'abcabc'
console.log('abc'.repeat(2.5)); // 'abcabc' (2.5 → 2)
console.log('abc'.repeat(-1));  // RangeError: Invalid count value
```

String.prototype.includes(searchString: string, position?: number): boolean

인수로 전달한 문자열이 포함되어 있는지를 검사하고 결과를 불리언 값으로 반환한다. 두번째 인수는 옵션으로 검색할 위치를 나타내는 정수이다.

```
const str = 'hello world';

console.log(str.includes('hello')); // true
console.log(str.includes(' '));     // true
console.log(str.includes('wo'));    // true
console.log(str.includes('wow'));   // false
console.log(str.includes(''));      // true
console.log(str.includes());        // false

// String.prototype.indexOf 메소드로 대체할 수 있다.
console.log(str.indexOf('hello')); // 0
```

배열

1개의 변수에 여러 개의 값을 순차적으로 저장할 때 사용한다. 자바스크립트의 배열은 객체이며 유용한 내장 메소드를 포함하고 있다. 배열은 Array 생성자로 생성된 Array 타입의 객체이며 프로토타입 객체는 Array.prototype이다.

배열의 생성

배열 리터럴

0개 이상의 값을 쉼표로 구분하여 대괄호([])로 묶는다. 첫번째 값은 인덱스 '0'으로 읽을 수 있다. 존재하지 않는 요소에 접근하면 `undefined`를 반환한다.

```
const emptyArr = [];

console.log(emptyArr[1]); // undefined
console.log(emptyArr.length); // 0

const arr = [
```

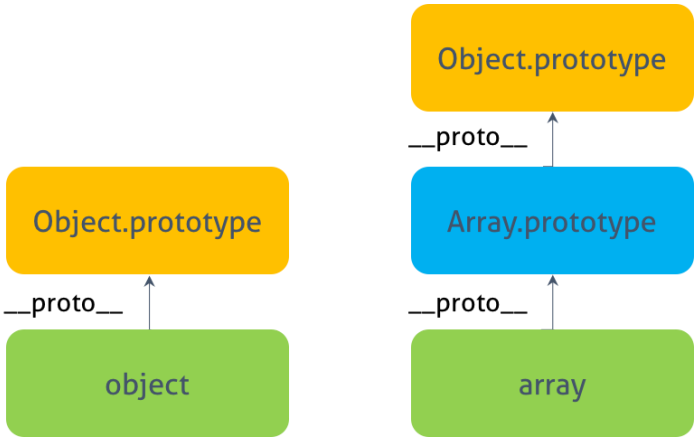
```
  'zero', 'one', 'two', 'three', 'four',
  'five', 'six', 'seven', 'eight', 'nine'
];

console.log(arr[1]);      // 'one'
console.log(arr.length);  // 10
console.log(typeof arr);  // object
```

위의 배열을 객체 리터럴로 유사하게 표현하면 다음과 같다.

```
const obj = {
  '0': 'zero',  '1': 'one',   '2': 'two',
  '3': 'three', '4': 'four',  '5': 'five',
  '6': 'six',   '7': 'seven', '8': 'eight',
  '9': 'nine'
};
```

배열 리터럴은 객체 리터럴과 달리 프로퍼티명이 없고 각 요소의 값만이 존재한다. 객체는 프로퍼티 값에 접근하기 위해 대괄호 표기법 또는 마침표 표기법을 사용하며 프로퍼티명을 키로 사용한다. 배열은 요소에 접근하기 위해 대괄호 표기법만을 사용하며 대괄호 내에 접근하고자 하는 요소의 인덱스를 넣어준다. 인덱스는 0부터 시작한다.

두 객체의 근본적 차이는 배열 리터럴 `arr`의 프로토타입 객체는 `Array.prototype`이지만 객체 리터럴 `obj`의 프로토타입 객체는 `Object.prototype`이라는 것이다.



객체리터럴과 배열리터럴의 프로토타입

```
const emptyArr = [];
const emptyObj = {};

console.dir(emptyArr.__proto__);
console.dir(emptyObj.__proto__);
```

대부분의 프로그래밍 언어에서 배열의 요소들은 모두 같은 데이터 타입이어야 하지만, 자바스크립트 배열은 어떤 데이터 타입의 조합이라도 포함할 수 있다.

```
const misc = [
  'string',
  10,
  true,
  null,
  undefined,
  NaN,
  Infinity,
  ['nested array'],
  { object: true },
  function () {}
];

console.log(misc.length); // 10
```

Array() 생성자 함수

배열은 일반적으로 배열 리터럴 방식으로 생성하지만 배열 리터럴 방식도 결국 내장 함수 Array() 생성자 함수로 배열을 생성하는 것을 단순화 시킨 것이다. Array() 생성자 함수는 Array.prototype.constructor프로퍼티로 접근할 수 있다. Array() 생성자 함수는 매개변수의 갯수에 따라 다르게 동작한다. 매개변수가 1개이고 숫자인 경우 매개변수로 전달된 숫자를 length 값으로 가지는 빈 배열을 생성한다.

```
const arr = new Array(2);
console.log(arr); // (2) [empty × 2]
```

그 외의 경우 매개변수로 전달된 값들을 요소로 가지는 배열을 생성한다.

```
const arr = new Array(1, 2, 3);
console.log(arr); // [1, 2, 3]
```

배열 요소의 추가와 삭제

배열요소의 추가

객체가 동적으로 프로퍼티를 추가할 수 있는 것처럼 배열도 동적으로 요소를 추가할 수있다. 이때 순서에 맞게 값을 할당할 필요는 없고 인덱스를 사용하여 필요한 위치에 값을 할당한다. 배열의 길이는 마지막 인덱스를 기준으로 산정된다.

```
const arr = [];
console.log(arr[0]); // undefined

arr[1] = 1;
arr[3] = 3;

console.log(arr); // (4) [empty, 1, empty, 3]
console.log(arr.lenth); // 4
```

값이 할당되지 않은 인덱스 위치의 요소는 생성되지 않는다는 것에 주의하자. 단, 존재하지 않는 요소를 참조하면 undefined가 반환된다.

```
// 값이 할당되지 않은 인덱스 위치의 요소는 생성되지 않는다.
console.log(Object.keys(arr)); // [ '1', '3' ]

// arr[0]이 undefined를 반환한 이유는 존재하지 않는 프로퍼티에 접근했을 때 undefined를 반환하는 것과 같은 이치다.
console.log(arr[0]); // undefined
```

배열요소의 삭제

배열은 객체이기 때문에 배열의 요소를 삭제하기 위해 `delete` 연산자를 사용할 수 있다. 이때 length에는 변함이 없다. 해당 요소를 완전히 삭제하여 length에도 반영되게 하기 위해서는 Array.prototype.splice
메소드를 사용한다.

```
const numbersArr = ['zero', 'one', 'two', 'three'];

// 요소의 값만 삭제된다
delete numbersArr[2]; // (4) ["zero", "one", empty, "three"]
console.log(numbersArr);
```

```
// 요소 값만이 아니라 요소를 완전히 삭제한다
// splice(시작 인덱스, 삭제할 요소수)
numbersArr.splice(2, 1); // (3) ["zero", "one", "three"]
console.log(numbersArr);
```

배열의 순회

배열은 객체이기 떄문에 프로퍼티를 가질수 있다. 그래서 배열의 순회에는 forEach 메소드, for 문, for ..of 문을 사용하는 것이 좋다.

```
const arr = [0, 1, 2, 3];
arr.foo = 10;

for (const key in arr) {
  console.log('key: ' + key, 'value: ' + arr[key]);
}
// key: 0 value: 0
// key: 1 value: 1
// key: 2 value: 2
// key: 3 value: 3
// key: foo value: 10 => 불필요한 프로퍼티까지 출력

arr.forEach((item, index) => console.log(index, item));

for (let i = 0; i < arr.length; i++) {
  console.log(i, arr[i]);
}

for (const item of arr) {
  console.log(item);
}
```

Array Property

Array.length

length 프로퍼티는 요소의 개수(배열의 길이)를 나타낸다.

```
const arr = [1, 2, 3, 4, 5];
console.log(arr.length); // 5

arr[4294967294] = 100;
console.log(arr.length); // 4294967295
console.log(arr); // (4294967295) [1, 2, 3, 4, 5, empty × 4294967289, 100]

arr[4294967295] = 1000;
console.log(arr.length); // 4294967295
console.log(arr); // (4294967295) [1, 2, 3, 4, 5, empty × 4294967289, 100, 4294967295: 1000]
```

주의할 것은 배열 요소의 개수와 length 프로퍼티의 값이 반드시 일치하지는 않는다는 것이다.

배열 요소의 개수와 length 프로퍼티의 값이 일치하지 않는 배열을 **희소 배열**(sparse array)이라 한다. 희소 배열은 배열의 요소가 연속적이지 않은 배열을 의미한다. 희소 배열이 아닌 일반 배열은 배열의 요소 개수와 length 프로퍼티의 값이 언제나 일치하지만 희소 배열은 배열의 요소 개수보다 length 프로퍼티의 값이 언제나 크다. 희소 배열은 일반 배열보다 느리며 메모리를 낭비한다.

현재 length 프로퍼티 값보다 더 큰 인덱스로 요소를 추가하면 새로운 요소를 추가할 수 있도록 자동으로 length 프로퍼티의 값이 늘어난다. length 프로퍼티의 값은 가장 큰 인덱스에 1을 더한 것과 같다.

```
const arr = [];
console.log(arr.length); // 0

arr[1000] = true;

console.log(arr);        // (1001) [empty × 1000, true]
console.log(arr.length); // 1001
console.log(arr[0]);     // undefined
```

length 프로퍼티의 값은 명시적으로 변경할 수 있다. 만약 length 프로퍼티의 값을 현재보다 작게 변경하면 변경된 length 프로퍼티의 값보다 크거나 같은 인덱스에 해당하는 요소는 모두 삭제된다.

```
const arr = [ 1, 2, 3, 4, 5 ];
```

```
// 배열 길이의 명시적 변경
arr.length = 3;
console.log(arr); // (3) [1, 2, 3]
```

Array Method

- ✏️ 메소드는 `this`(원본 배열)를 변경한다.
- 🔒 메소드는 `this`(원본 배열)를 변경하지 않는다.

Array,isArray(arg:any): boolean

정적 메소드 Array.isArray는 주어진 인수가 배열이면 true, 배열이 아니면 false를 반환한다.

```
// true
Array.isArray([]);
Array.isArray([1, 2]);
Array.isArray(new Array());

// false
Array.isArray();
Array.isArray({});
Array.isArray(null);
Array.isArray(undefined);
Array.isArray(1);
Array.isArray('Array');
Array.isArray(true);
Array.isArray(false);
```

Array.from

유사 배열 객체(array-like object) 또는 이터러블 객체(iterable object)를 변환하여 새로운 배열을 생성한다.

```
// 문자열은 이터러블이다.
const arr1 = Array.from('Hello');
console.log(arr1); // [ 'H', 'e', 'l', 'l', 'o' ]

// 유사 배열 객체를 새로운 배열을 변환하여 반환한다.
const arr2 = Array.from({ length: 2, 0: 'a', 1: 'b' });
console.log(arr2); // [ 'a', 'b' ]

// Array.from의 두번째 매개변수에게 배열의 모든 요소에 대해 호출할 함수를 전달할 수 있다.
// 이 함수는 첫번째 매개변수에게 전달된 인수로 생성된 배열의 모든 요소를 인수로 전달받아 호출된다.
const arr3 = Array.from({ length: 5 }, function (v, i) { return i; });
console.log(arr3); // [ 0, 1, 2, 3, 4 ]
```

Array.of

전달된 인수를 요소로 갖는 배열을 생성한다. Array.of는 Array 생성자 함수와 다르게 전달된 인수가 1개이고 숫자이더라도 인수를 요소로 갖는 배열을 생성한다.

```
// 전달된 인수가 1개이고 숫자이더라도 인수를 요소로 갖는 배열을 생성한다.
const arr1 = Array.of(1);
console.log(arr1); // // [1]

const arr2 = Array.of(1, 2, 3);
console.log(arr2); // [1, 2, 3]

const arr3 = Array.of('string');
console.log(arr3); // 'string'
```

Array.prototype.indexOf(searchElement: T, fromIndex?: number): number

원본 배열에서 인수로 전달된 요소를 검색하여 인덱스를 반환한다.

- 중복되는 요소가 있는 경우, 첫번째 인덱스를 반환한다.
- 해당하는 요소가 없는 경우, -1을 반환한다.

```
const arr = [1, 2, 2, 3];

// 배열 arr에서 요소 2를 검색하여 첫번째 인덱스를 반환
```

```
arr.indexOf(2);    // -> 1
// 배열 arr에서 요소 4가 없으므로 -1을 반환
arr.indexOf(4);    // -1
// 두번째 인수는 검색을 시작할 인덱스이다. 두번째 인수를 생략하면 처음부터 검색한다.
arr.indexOf(2, 2); // 2
```

indexOf 메소드는 배열에 요소가 존재하는지 여부를 확인할 때 유용하다.

```
const foods = ['apple', 'banana', 'orange'];

// foods 배열에 'orange' 요소가 존재하는지 확인
if (foods.indexOf('orange') === -1) {
  // foods 배열에 'orange' 요소가 존재하지 않으면 'orange' 요소를 추가
  foods.push('orange');
}

console.log(foods); // ["apple", "banana", "orange"]
```

ES7에서 새롭게 도입된 Array.prototype.includes 메소드를 사용하면 보다 가독성이 좋다.

```
const foods = ['apple', 'banana'];

// ES7: Array.prototype.includes
// foods 배열에 'orange' 요소가 존재하는지 확인
if (!foods.includes('orange')) {
  // foods 배열에 'orange' 요소가 존재하지 않으면 'orange' 요소를 추가
  foods.push('orange');
}

console.log(foods); // ["apple", "banana", "orange"]
```

Array.prototype.concat(…items: Array<T[] | T>): T[]

인수로 전달된 값들(배열 또는 값)을 원본 배열의 마지막 요소로 추가한 새로운 배열을 반환한다. 인수로 전달한 값이 배열인 경우, 배열을 해체하여 새로운 배열의 요소로 추가한다. 원본 배열은 변경되지 않는다.

```
const arr1 = [1, 2];
const arr2 = [3, 4];

// 배열 arr2를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환
// 인수로 전달한 값이 배열인 경우, 배열을 해체하여 새로운 배열의 요소로 추가한다.
let result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4]

// 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환
result = arr1.concat(3);
console.log(result); // ["1, 2, 3]

//  배열 arr2와 숫자를 원본 배열 arr1의 마지막 요소로 추가한 새로운 배열을 반환
result = arr1.concat(arr2, 5);
console.log(result); // [1, 2, 3, 4, 5]

// 원본 배열은 변경되지 않는다.
console.log(arr1); // [1, 2]
```

Array.prototype.join(separator?: string): string

원본 배열의 모든 요소를 문자열로 변환한 후, 인수로 전달받은 값, 즉 구분자(separator)로 연결한 문자열을 반환한다. 구분자(separator)는 생략 가능하며 기본 구분자는 `,`이다.

```
const arr = [1, 2, 3, 4];

// 기본 구분자는 ','이다.
// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 기본 구분자 ','로 연결한 문자열을 반환
let result = arr.join();
console.log(result); // '1,2,3,4';

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 빈문자열로 연결한 문자열을 반환
result = arr.join('');
console.log(result); // '1234'

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 구분자 ':'로 연결한 문자열을 반환
result = arr.join(':');
console.log(result); // '1:2:3:4'
```

Array.prototype.push(…items: T[]): number

인수로 전달받은 모든 값을 원본 배열의 마지막에 요소로 추가하고 변경된 length 값을 반환한다. push 메소드는 원본 배열을 직접 변경한다.

```
const arr = [1, 2];

// 인수로 전달받은 모든 값을 원본 배열의 마지막에 요소로 추가하고 변경된 length 값을 반환한다.
let result = arr.push(3, 4);
console.log(result); // 4

// push 메소드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 2, 3, 4]
```

push 메소드와 concat 메소드는 유사하게 동작하지만 미묘한 차이가 있다.

- push 메소드는 원본 배열을 직접 변경하지만 concat 메소드는 원본 배열을 변경하지 않고 새로운 배열을 반환한다.

```
const arr1 = [1, 2];
// push 메소드는 원본 배열을 직접 변경한다.
arr1.push(3, 4);
console.log(arr1); // [1, 2, 3, 4]

const arr2 = [1, 2];
// concat 메소드는 원본 배열을 변경하지 않고 새로운 배열을 반환한다.
const result = arr2.concat(3, 4);
console.log(result); // [1, 2, 3, 4]
```

- 인수로 전달받은 값이 배열인 경우, push 메소드는 배열을 그대로 원본 배열의 마지막 요소로 추가하지만 concat 메소드는 배열을 해체하여 새로운 배열의 마지막 요소로 추가한다.

```
const arr1 = [1, 2];
// 인수로 전달받은 배열을 그대로 원본 배열의 마지막 요소로 추가한다
arr1.push([3, 4]);
console.log(arr1); // [1, 2, [3, 4]]

const arr2 = [1, 2];
// 인수로 전달받은 배열을 해체하여 새로운 배열의 마지막 요소로 추가한다
const result = arr2.concat([3, 4]);
console.log(result); // [1, 2, 3, 4]
```

push 메소드는 성능면에서 좋지 않다. push 메소드는 배열의 마지막에 요소를 추가하므로 length 프로퍼티를 사용하여 직접 요소를 추가할 수도 있다. 이 방법이 push 메소드보다 빠르다.

```
const arr = [1, 2];

// arr.push(3)와 동일한 처리를 한다. 이 방법이 push 메소드보다 빠르다.
arr[arr.length] = 3;

console.log(arr); // [1, 2, 3]
```

push 메소드는 원본 배열을 직접 변경하는 부수 효과가 있다. 따라서 push 메소드보다는 spread 문법을 사용하는 편이 좋다.

```
const arr = [1, 2];

// ES6 spread 문법
const newArr = [...arr, 3];
// arr.push(3);

console.log(newArr); // [1, 2, 3]
```

Array.prototype.pop():T | undefined

원본 배열에서 마지막 요소를 제거하고 제거한 요소를 반환한다. 원본 배열이 빈 배열이면 undefined를 반환한다. pop 메소드는 원본 배열을 직접 변경한다.

```
const arr = [1, 2];

// 원본 배열에서 마지막 요소를 제거하고 제거한 요소를 반환한다.
let result = arr.pop();
console.log(result); // 2

// pop 메소드는 원본 배열을 직접 변경한다.
console.log(arr); // [1]
```

pop 메소드와 push 메소드를 사용하면 스택을 쉽게 구현할 수 있다.

스택(stack)은 데이터를 마지막에 밀어 넣고, 마지막에 밀어 넣은 데이터를 먼저 꺼내는 후입 선출(LIFO - Last In First Out) 방식의 자료 구조이다. 스택은 언제나 가장 마지막에 밀어 넣은 최신 데이터를 취득한다. 스택에 데이터를 밀어 넣는 것을 푸시(push)라 하고 스택에서 데이터를 꺼내는 것을 팝(pop)이라고 한다.

```
// 스택 자료 구조를 구현하기 위한 배열
const stack = [];

// 스택의 가장 마지막에 데이터를 밀어 넣는다.
stack.push(1);
console.log(stack); // [1]

// 스택의 가장 마지막에 데이터를 밀어 넣는다.
stack.push(2);
console.log(stack); // [1, 2]

// 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신 데이터를 꺼낸다.
let value = stack.pop();
console.log(value, stack); // 2 [1]

// 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신 데이터를 꺼낸다.
value = stack.pop();
console.log(value, stack); // 1 []
```

Array.prototype.reverse(): this

배열 요소의 순서를 반대로 변경한다. 이때 원본 배열이 변경된다. 반환값은 변경된 배열이다.

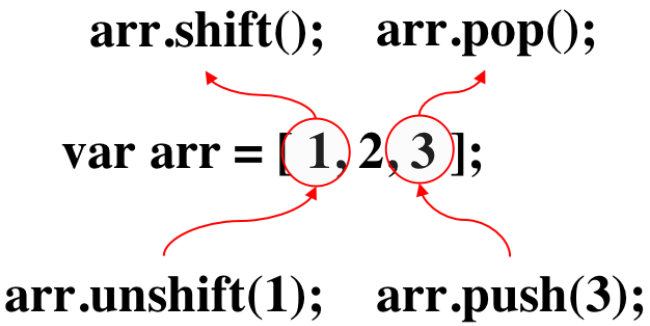
```
const a = ['a', 'b', 'c'];
const b = a.reverse();

// 원본 배열이 변경된다
console.log(a); // [ 'c', 'b', 'a' ]
console.log(b); // [ 'c', 'b', 'a' ]
```

`shift`는 `push`와 함께 배열을 큐(FIFO: First In First Out)처럼 동작하게 한다.

```
const arr = [];

arr.push(1); // [1]
arr.push(2); // [1, 2]
arr.push(3); // [1, 2, 3]

arr.shift(); // [2, 3]
arr.shift(); // [3]
arr.shift(); // []
```

Array.prototype.pop()은 마지막 요소를 제거하고 제거한 요소를 반환한다.

```
const a = ['a', 'b', 'c'];
const c = a.pop();

// 원본 배열이 변경된다.
console.log(a); // a --> ['a', 'b']
console.log(c); // c --> 'c'
```



Array.prototype.slice(start=0, end=this.length): T[]

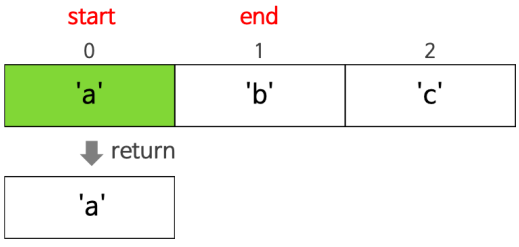
인자로 지정된 배열의 부분을 복사하여 반환한다. 원본 배열은 변경되지 않는다.

첫번째 매개변수 start에 해당하는 인덱스를 갖는 요소부터 매개변수 end에 해당하는 인덱스를 가진 요소 전까지 복사된다.

- 매개변수

start복사를 시작할 인텍스. 음수인 경우 배열의 끝에서의 인덱스를 나타낸다. 예를 들어 slice(-2)는 배열의 마지막 2개의 요소를 반환한다.**end**옵션이며 기본값은 length 값이다.

```
const items = ['a', 'b', 'c'];

// items[0]부터 items[1] 이전(items[1] 미포함)까지 반환
let res = items.slice(0, 1);
console.log(res);  // [ 'a' ]

// items[1]부터 items[2] 이전(items[2] 미포함)까지 반환
res = items.slice(1, 2);
console.log(res);  // [ 'b' ]

// items[1]부터 이후의 모든 요소 반환
res = items.slice(1);
console.log(res);  // [ 'b', 'c' ]

// 인자가 음수인 경우 배열의 끝에서 요소를 반환
res = items.slice(-1);
console.log(res);  // [ 'c' ]

res = items.slice(-2);
console.log(res);  // [ 'b', 'c' ]

// 모든 요소를 반환 (= 복사본(shallow copy) 생성)
res = items.slice();
console.log(res);  // [ 'a', 'b', 'c' ]

// 원본은 변경되지 않는다.
console.log(items); // [ 'a', 'b', 'c' ]
```



Array.prototype.slice 메소드

slice 메소드에 인자를 전달하지 않으면 원본 배열의 복사본을 생성하여 반환한다.

```
const arr = [1, 2, 3];

// 원본 배열 arr의 새로운 복사본을 생성한다.
const copy = arr.slice();
console.log(copy, copy === arr); // [ 1, 2, 3 ] false
```

이때 원본 배열의 각 요소를 얕은 복사(shallow copy)하여 새로운 복사본을 생성한다.

```
const todos = [
  { id: 1, content: 'HTML', completed: false },
  { id: 2, content: 'CSS', completed: true },
  { id: 3, content: 'Javascript', completed: false }
];

// shallow copy
const _todos = todos.slice();
// const _todos = [...todos];
console.log(_todos === todos); // false

// 배열의 요소는 같다. 즉, 얕은 복사되었다.
console.log(_todos[0] === todos[0]); // true
```

Spread 문법과 Object.assign는 원본을 shallow copy한다. Deep copy를 위해서는 lodash의 deepClone을 사용하는 것을 추천한다.

이를 이용하여 arguments, HTMLCollection, NodeList와 같은 유사 배열 객체(Array-like Object)를 배열로 변환할 수 있다.

```
function sum() {
  // 유사 배열 객체 => Array
  const arr = Array.prototype.slice.call(arguments);
  console.log(arr); // [1, 2, 3]

  return arr.reduce(function (pre, cur) {
    return pre + cur;
  });
}

console.log(sum(1, 2, 3));
```

유사 배열 객체를 배열로 변환하는 방법은 아래와 같다.

```
// 유사 배열 객체 => Array
function sum() {
  ...
  // Spread 문법
  const arr = [...arguments];
  // Array.from 메소드는 유사 배열 객체를 복사하여 배열을 생성한다.
  const arr = Array.from(arguments);
  ...
}
```

Array.prototype.splice(start: number, deleteCount=this.length-start, …items: T[]): T[]

기존의 배열의 요소를 제거하고 그 위치에 새로운 요소를 추가한다. 배열 중간에 새로운 요소를 추가할 때도 사용된다.

- 매개변수

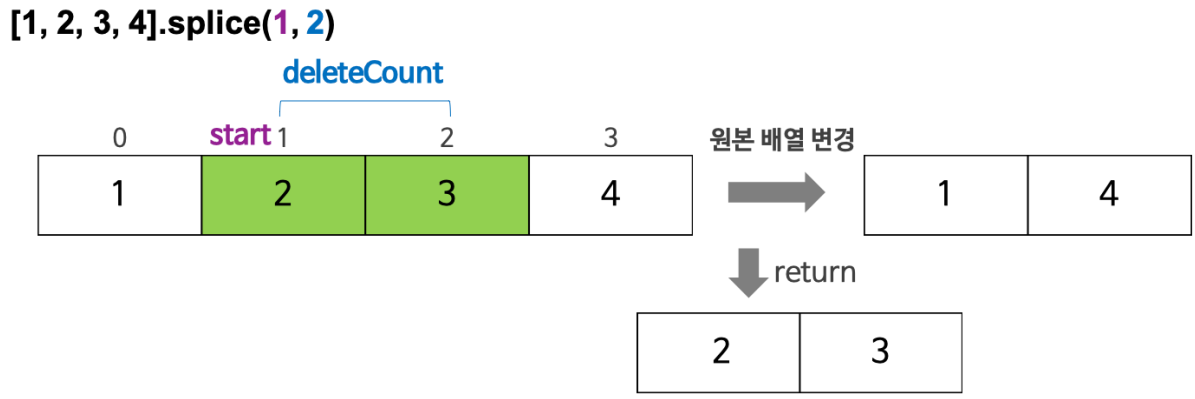
start배열에서의 시작 위치이다. start 만을 지정하면 배열의 start부터 모든 요소를 제거한다.**deleteCount**시작 위치(start)부터 제거할 요소의 수이다. deleteCount가 0인 경우, 아무런 요소도 제거되지 않는다. (옵션)**items**삭제한 위치에 추가될 요소들이다. 만약 아무런 요소도 지정하지 않을 경우, 삭제만 한다. (옵션)

- 반환값 삭제한 요소들을 가진 배열이다.

이 메소드의 가장 일반적인 사용은 배열에서 요소를 삭제할 때다.

```
const items1 = [1, 2, 3, 4];

// items[1]부터 2개의 요소를 제거하고 제거된 요소를 배열로 반환
const res1 = items1.splice(1, 2);

// 원본 배열이 변경된다.
console.log(items1); // [ 1, 4 ]
// 제거한 요소가 배열로 반환된다.
console.log(res1);   // [ 2, 3 ]

const items2 = [1, 2, 3, 4];

// items[1]부터 모든 요소를 제거하고 제거된 요소를 배열로 반환
const res2 = items2.splice(1);

// 원본 배열이 변경된다.
console.log(items2); // [ 1 ]
```

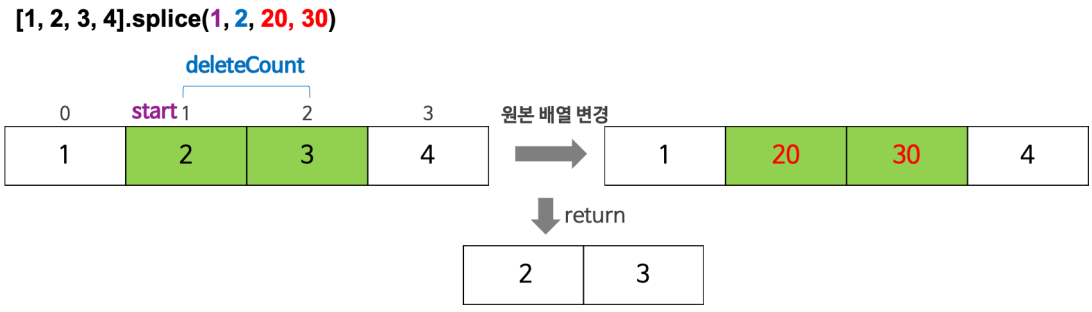
```
// 제거한 요소가 배열로 반환된다.
console.log(res2);   // [ 2, 3, 4 ]
```



Array.prototype.splice 메소드

배열에서 요소를 제거하고 제거한 위치에 다른 요소를 추가한다.

```
const items = [1, 2, 3, 4];

// items[1]부터 2개의 요소를 제거하고 그자리에 새로운 요소를 추가한다. 제거된 요소가 반환된다.
const res = items.splice(1, 2, 20, 30);

// 원본 배열이 변경된다.
console.log(items); // [ 1, 20, 30, 4 ]
// 제거한 요소가 배열로 반환된다.
console.log(res);   // [ 2, 3 ]
```



Array.prototype.splice 메소드

배열 중간에 새로운 요소를 추가할 때도 사용된다.

```
const items = [1, 2, 3, 4];

// items[1]부터 0개의 요소를 제거하고 그자리(items[1])에 새로운 요소를 추가한다. 제거된 요소가 반환된다.
const res = items.splice(1, 0, 100);

// 원본 배열이 변경된다.
console.log(items); // [ 1, 100, 2, 3, 4 ]
// 제거한 요소가 배열로 반환된다.
console.log(res);   // [ ]
```

배열 중간에 배열의 요소들을 해체하여 추가할 때도 사용된다.

```
const items = [1, 4];

// items[1]부터 0개의 요소를 제거하고 그자리(items[1])에 새로운 배열를 추가한다. 제거된 요소가 반환된다.
// items.splice(1, 0, [2, 3]); // [ 1, [ 2, 3 ], 4 ]
Array.prototype.splice.apply(items, [1, 0].concat([2, 3]));
// ES6
// items.splice(1, 0, ...[2, 3]);

console.log(items); // [ 1, 2, 3, 4 ]
```

slice는 배열의 일부분을 복사해서 반환하며 원본을 훼손하지 않는다. splice는 배열에서 요소를 제거하고 제거한 위치에 다른 요소를 추가하며 원본을 훼손한다.

자바스크립트의 배열

```
console.log(Object.getOwnPropertyDescriptors([1, 2, 3]));
/*
{
  '0': { value: 1, writable: true, enumerable: true, configurable: true },
  '1': { value: 2, writable: true, enumerable: true, configurable: true },
  '2': { value: 3, writable: true, enumerable: true, configurable: true },
  length: { value: 3, writable: true, enumerable: false, configurable: false }
}
*/
```

자바스크립트 배열은 인덱스를 프로퍼티 키로 갖으며 length 프로퍼티를 갖는 특수한 객체이다. 자바스크립트 배열의 요소는 사실 프로퍼티 값이다. 자바스크립트에서 사용할 수 있는 모든 값은 객체의 프로퍼티 값이 될 수 있으므로 어떤 타입의 값이라도 배열의 요소가 될 수 있다.

```
const arr = [
  'string',
  10,
  true,
  null,
  undefined,
  NaN,
  Infinity,
  [ ],
  { },
  function () {}
];
```

자바스크립트 배열은 인덱스로 배열 요소에 접근하는 경우에는 일반적인 배열보다 느리지만 특정 요소를 탐색하거나 요소를 삽입 또는 삭제하는 경우에는 일반적인 배열보다 빠르다. 자바스크립트 배열은 인덱스로 접근하는 경우의 성능 대신 특정 요소를 탐색하거나 배열 요소를 삽입 또는 삭제하는 경우의 성능을 선택한 것이다.

이처럼 인덱스로 배열 요소에 접근할 때 일반적인 배열보다 느릴 수 밖에 없는 구조적인 단점을 보완하기 위해 대부분의 모던 자바스크립트 엔진은 배열을 일반 객체와 구별하여 보다 배열처럼 동작하도록 최적화하여 구현하였다.

```
const arr = [];

console.time('Array Performance Test');

for (let i = 0; i < 10000000; i++) {
  arr[i] = i;
}
console.timeEnd('Array Performance Test');
// 약 340ms

const obj = {};

console.time('Object Performance Test');

for (let i = 0; i < 10000000; i++) {
  obj[i] = i;
}

console.timeEnd('Object Performance Test');
// 약 600ms
```

배열 고차 함수

고차함수는 함수를 인자로 전달받거나 함수를 결과로 반환하는 함수를 말한다. 다시말해, 고차 함수는 인자로 받은 함수를 필요한 시점에 호출하거나 클로저를 생성하여 반환한다. 자바스크립트의 함수는 일급객체 이므로 값처럼 인자로 전달할 수 있으며 반환할 수도 있다.

```
// 함수를 인자로 전달받고 함수를 반환하는 고차 함수
function makeCounter(predicate) {
```

```
  // 자유 변수. num의 상태는 유지되어야 한다.
  let num = 0;
  // 클로저. num의 상태를 유지한다.
  return function () {
    // predicate는 자유 변수 num의 상태를 변화시킨다.
    num = predicate(num);
    return num;
  };
}

// 보조 함수
function increase(n) {
  return ++n;
}

// 보조 함수
function decrease(n) {
  return --n;
}

// makeCounter는 함수를 인수로 전달받는다. 그리고 클로저를 반환한다.
const increaser = makeCounter(increase);
console.log(increaser()); // 1
console.log(increaser()); // 2

// makeCounter는 함수를 인수로 전달받는다. 그리고 클로저를 반환한다.
const decreaser = makeCounter(decrease);
console.log(decreaser()); // -1
console.log(decreaser()); // -2
```

고차 함수는 외부 상태 변경이나 가변 데이터를 피하고 불변성을 지향하는 함수형 프로그래밍에 기반을 두고 있다.

함수형 프로그래밍은 순수 함수(Pure function)와 보조 함수의 조합을 통해 로직 내에 존재하는 조건문과 반복문을 제거하여 복잡성을 해결하고 변수의 사용을 억제하여 상태 변경을 피하려는 프로그래밍 패러다임이다. 조건문이나 반복문은 로직의 흐름을 이해하기 어렵게 하여 가독성을 해치고, 변수의 값은 누군가에 의해 언제든지 변경될 수 있어 오류 발생의 근본적 원인이 될 수 있기 때문이다.

함수형 프로그래밍은 결국 순수 함수를 통해 **부수 효과(Side effect)를 최대한 억제**하여 오류를 피하고 프로그램의 안정성을 높이려는 노력의 한 방법이라고 할 수 있다.

Array.prototype.sort(compareFn?: (a: T, b: T) => number): this

배열의 요소를 적절하게 정렬, 원본 배열을 직접 변경하며 정렬된 배열을 반환한다.

```
const fruits = ['Banana', 'Orange', 'Apple'];

// ascending(오름차순)
fruits.sort();
console.log(fruits); // [ 'Apple', 'Banana', 'Orange' ]

// descending(내림차순)
fruits.reverse();
console.log(fruits); // [ 'Orange', 'Banana', 'Apple' ]
```

숫자 정렬

```
const points = [40, 100, 1, 5, 2, 25, 10];

// 숫자 배열 오름차순 정렬
// 비교 함수의 반환값이 0보다 작은 경우, a를 우선하여 정렬한다.
points.sort(function (a, b) { return a - b; });
// ES6 화살표 함수
// points.sort((a, b) => a - b);
console.log(points); // [ 1, 2, 5, 10, 25, 40, 100 ]

// 숫자 배열에서 최소값 취득
console.log(points[0]); // 1

// 숫자 배열 내림차순 정렬
// 비교 함수의 반환값이 0보다 큰 경우, b를 우선하여 정렬한다.
points.sort(function (a, b) { return b - a; });
// ES6 화살표 함수
// points.sort((a, b) => b - a);
console.log(points); // [ 100, 40, 25, 10, 5, 2, 1 ]

// 숫자 배열에서 최대값 취득
console.log(points[0]); // 100
```

객체를 요소로 갖는 배열을 정렬

```
const todos = [
  { id: 4, content: 'JavaScript' },
  { id: 1, content: 'HTML' },
  { id: 2, content: 'CSS' }
];

// 비교 함수
function compare(key) {
  return function (a, b) {
    // 프로퍼티 값이 문자열인 경우, - 산술 연산으로 비교하면 NaN이 나오므로 비교 연산을 사용한다.
    return a[key] > b[key] ? 1 : (a[key] < b[key] ? -1 : 0);
  };
}

// id를 기준으로 정렬
todos.sort(compare('id'));
console.log(todos);

// content를 기준으로 정렬
todos.sort(compare('content'));
console.log(todos);
```

Array.prototype.forEach(callback: (value: T, index: number, array: T[]) => void, thisArg?: any): void

- forEach 메소드는 for 문 대신 사용할 수 있다.
- 배열을 순회하며 배열의 각 요소에 대하여 인자로 주어진 콜백함수를 실행한다. **반환값은 undefined이다.**
- 콜백 함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, forEach 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.
- forEach 메소드는 원본 배열(this)을 변경하지 않는다. 하지만 콜백 함수는 원본 배열(this)을 변경할 수는 있다.
- **forEach 메소드는 for 문과는 달리 break 문을 사용할 수 없다.** 다시 말해, 배열의 모든 요소를 순회하며 중간에 순회를 중단할 수 없다.
- forEach 메소드는 for 문에 비해 성능이 좋지는 않다. 하지만 for 문보다 가독성이 좋으므로 적극 사용을 권장한다.
- IE 9 이상에서 정상 동작한다.

```
const numbers = [1, 2, 3];
let pows = [];

// for 문으로 순회
for (let i = 0; i < numbers.length; i++) {
  pows.push(numbers[i] ** 2);
}

console.log(pows); // [ 1, 4, 9 ]

pows = [];

// forEach 메소드로 순회
numbers.forEach(function (item) {
  pows.push(item ** 2);
});

// ES6 화살표 함수
// numbers.forEach(item => pows.push(item ** 2));

console.log(pows); // [ 1, 4, 9 ]
```

```
const numbers = [1, 3, 5, 7, 9];
let total = 0;

// forEach 메소드는 인수로 전달한 보조 함수를 호출하면서
// 3개(배열 요소의 값, 요소 인덱스, this)의 인수를 전달한다.
// 배열의 모든 요소를 순회하며 합산한다.
numbers.forEach(function (item, index, self) {
  console.log(`numbers[${index}] = ${item}`);
  total += item;
});

// Array#reduce를 사용해도 위와 동일한 결과를 얻을 수 있다
// total = numbers.reduce(function (pre, cur) {
```

```
//   return pre + cur;
// });

console.log(total); // 25
console.log(numbers); // [ 1, 3, 5, 7, 9 ]
```

```
const numbers = [1, 2, 3, 4];

// forEach 메소드는 원본 배열(this)을 변경하지 않는다. 하지만 콜백 함수는 원본 배열(this)을 변경할 수는 있다.
// 원본 배열을 직접 변경하려면 콜백 함수의 3번째 인자(this)를 사용한다.
numbers.forEach(function (item, index, self) {
  self[index] = Math.pow(item, 2);
});

console.log(numbers); // [ 1, 4, 9, 16 ]
```

```
// forEach 메소드는 for 문과는 달리 break 문을 사용할 수 없다.
[1, 2, 3].forEach(function (item, index, self) {
  console.log(`self[${index}] = ${item}`);
  if (item > 1) break; // SyntaxError: Illegal break statement
});
```

forEach 메소드에 두번째 인자로 this를 전달할 수 있다.

```
function Square() {
  this.array = [];
}

Square.prototype.multiply = function (arr) {
  arr.forEach(function (item) {
    // this를 인수로 전달하지 않으면 this === window
    this.array.push(item * item);
  }, this);
};

const square = new Square();
square.multiply([1, 2, 3]);
console.log(square.array); // [ 1, 4, 9 ]
```

ES6의 Arrow function를 사용하면 this를 생략하여도 동일한 동작을 한다.

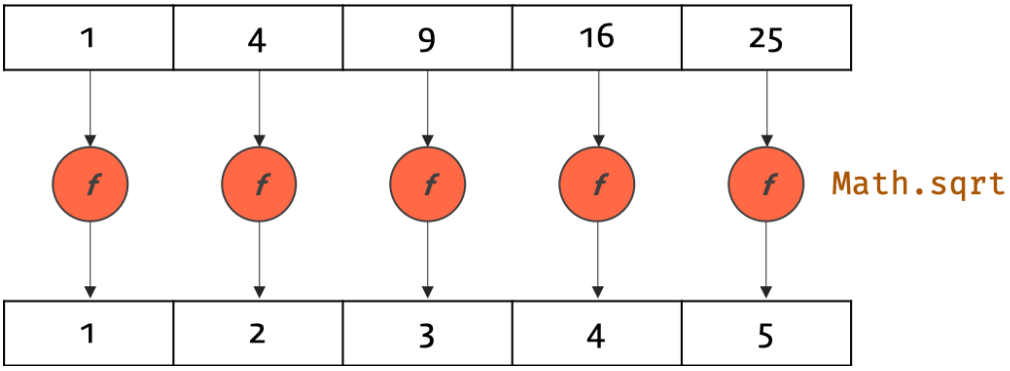
```
Square.prototype.multiply = function (arr) {
  arr.forEach(item => this.array.push(item * item));
};
```

forEach 메소드의 이해를 돕기 위해 forEach의 동작을 흉내낸 myForEach 메소드를 작성해 보자.

```
Array.prototype.myForEach = function (f) {
  // 첫번재 매개변수에 함수가 전달되었는지 확인
  // console.log((function(){}).toString()); // function(){}
  // console.log(Object.prototype.toString.call(function(){})); // [object Function]
  // poiemaweb.com/js-type-check 참고
  if (!f || {}.toString.call(f) !== '[object Function]') {
    throw new TypeError(f + ' is not a function.');
  }

  for (let i = 0; i < this.length; i++) {
    // 배열 요소의 값, 요소 인덱스, forEach 메소드를 호출한 배열, 즉 this를 매개변수에 전달하고 콜백 함수 호출
    f(this[i], i, this);
  }
};

let total = 0;

[0, 1, 2, 3].myForEach(function (item, index, array) {
  console.log(`[${index}]: ${item} of [${array}]`);
  total += item;
});

console.log('Total: ', total);
```

Array.prototype.map<U>(callbackfn: (value: T, index: number, array: T[]) => U, thisArg?: any): U[]

- 배열을 순회하며 각 요소에 대하여 인자로 주어진 **콜백 함수의 반환값(결과값)으로 새로운 배열을 생성하여 반환한다.** 이때 원본 배열은 변경되지 않는다.
- forEach 메소드는 배열을 순회하며 요소 값을 참조하여 무언가를 하기 위한 함수이며 map 메소드는 배열을 순회하며 요소 값을 다른 값으로 맵핑하기 위한 함수이다.



Array.prototype.map

- 콜백 함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, map 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.
- IE 9 이상에서 정상 동작한다.

```
const numbers = [1, 4, 9];

// 배열을 순회하며 각 요소에 대하여 인자로 주어진 콜백함수를 실행
const roots = numbers.map(function (item) {
  // 반환값이 새로운 배열의 요소가 된다. 반환값이 없으면 새로운 배열은 비어 있다.
  return Math.sqrt(item);
});

// 위 코드의 축약표현은 아래와 같다.
// const roots = numbers.map(Math.sqrt);

// map 메소드는 새로운 배열을 반환한다
console.log(roots);   // [ 1, 2, 3 ]
// map 메소드는 원본 배열은 변경하지 않는다
console.log(numbers); // [ 1, 4, 9 ]
```

map 메소드에 두번째 인자로 this를 전달할 수 있다.

```
function Prefixer(prefix) {
  this.prefix = prefix;
}

Prefixer.prototype.prefixArray = function (arr) {
  // 콜백함수의 인자로 배열 요소의 값, 요소 인덱스, map 메소드를 호출한 배열, 즉 this를 전달할 수 있다.
  return arr.map(function (x) {
    // x는 배열 요소의 값이다.
    return this.prefix + x; // 2번째 인자 this를 전달하지 않으면 this === window
  }, this);
};

const pre = new Prefixer('-webkit-');
const preArr = pre.prefixArray(['linear-gradient', 'border-radius']);
console.log(preArr);
// [ '-webkit-linear-gradient', '-webkit-border-radius' ]
```

ES6의 Arrow function를 사용하면 this를 생략하여도 동일한 동작을 한다.

map 메소드의 이해를 돕기 위해 map의 동작을 흉내낸 myMap 메소드를 작성해 보자.

```
Array.prototype.myMap = function (iteratee) {
  // 첫번재 매개변수에 함수가 전달되었는지 확인
  if (!iteratee || {}.toString.call(iteratee) !== '[object Function]') {
    throw new TypeError(iteratee + ' is not a function.');
```

```
  }

  const result = [];
  for (let i = 0, len = this.length; i < len; i++) {
    /**
     * 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 매개변수를 통해 iteratee에 전달하고
     * iteratee를 호출하여 그 결과를 반환용 배열에 푸시하여 반환한다.
     */
    result.push(iteratee(this[i], i, this));
  }
  return result;
};

const result = [1, 4, 9].myMap(function (item, index, self) {
  console.log(`[${index}]: ${item} of [${self}]`);
  return Math.sqrt(item);
});

console.log(result); // [ 1, 2, 3 ]
```

Array.prototype.filter(callback: (value: T, index: number, array: Array) => any, thisArg?: any): T[]

- filter 메소드를 사용하면 if 문을 대체할 수 있다.
- 배열을 순회하며 각 요소에 대하여 인자로 주어진 **콜백함수의 실행 결과가 true인 배열 요소의 값만을 추출한 새로운 배열을 반환한다.**
- 배열에서 특정 케이스만 필터링 조건으로 추출하여 새로운 배열을 만들고 싶을 때 사용한다. 이때 원본 배열은 변경되지 않는다.
- 콜백 함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, filter 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.
- IE 9 이상에서 정상 동작한다.

```
const result = [1, 2, 3, 4, 5].filter(function (item, index, self) {
  console.log(`[${index}] = ${item}`);
  return item % 2; // 홀수만을 필터링한다 (1은 true로 평가된다)
});

console.log(result); // [ 1, 3, 5 ]
```

filter도 map, forEach와 같이 두번째 인자로 this를 전달할 수 있다.

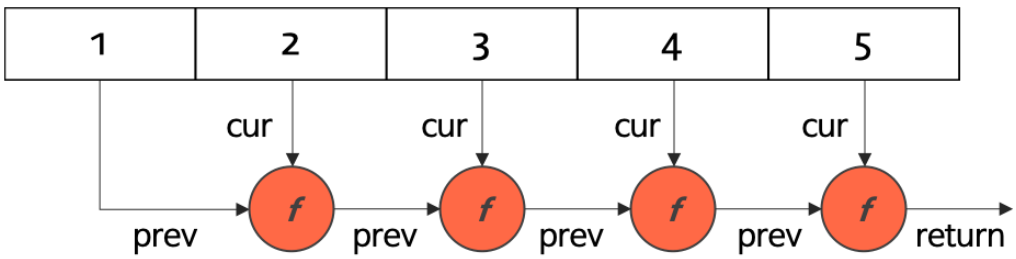
filter 메소드의 이해를 돕기 위해 filter의 동작을 흉내낸 myFilter 메소드를 작성해 보자.

```
Array.prototype.myFilter = function (predicate) {
  // 첫번재 매개변수에 함수가 전달되었는지 확인
  if (!predicate || {}.toString.call(predicate) !== '[object Function]') {
    throw new TypeError(predicate + ' is not a function.');
  }

  const result = [];
  for (let i = 0, len = this.length; i < len; i++) {
    /**
     * 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 매개변수를 통해 predicate에 전달하고
     * predicate를 호출하여 그 결과가 참인 요소만을 반환용 배열에 푸시하여 반환한다.
     */
    if (predicate(this[i], i, this)) result.push(this[i]);
  }
  return result;
};

const result = [1, 2, 3, 4, 5].myFilter(function (item, index, self) {
  console.log(`[${index}]: ${item} of [${self}]`);
  return item % 2; // 홀수만을 필터링한다 (1은 true로 평가된다)
});

console.log(result); // [ 1, 3, 5 ]
```

Array.prototype.reduce<U>(callback: (state: U, element: T, index: number, array: T[]) => U, firstState?: U): U

배열을 순회하며 각 요소에 대하여 이전의 콜백함수 실행 반환값을 전달하여 콜백함수를 실행하고 그 결과를 반환한다. IE 9 이상에서 정상 동작한다.

```
const arr = [1, 2, 3, 4, 5];

/*
previousValue: 이전 콜백의 반환값
currentValue : 배열 요소의 값
currentIndex : 인덱스
array        : 메소드를 호출한 배열, 즉 this
*/
// 합산
const sum = arr.reduce(function (previousValue, currentValue, currentIndex, self) {
  console.log(previousValue + '+' + currentValue + '=' + (previousValue + currentValue));
  return previousValue + currentValue; // 결과는 다음 콜백의 첫번째 인자로 전달된다
});

console.log(sum); // 15: 1~5까지의 합
/*
1: 1+2=3
2: 3+3=6
3: 6+4=10
4: 10+5=15
15
*/

// 최대값 취득
const max = arr.reduce(function (pre, cur) {
  return pre > cur ? pre : cur;
});

console.log(max); // 5: 최대값
```



Array.prototype.reduce

Array.prototype.reduce의 두번째 인수로 초기값을 전달할 수 있다. 이 값은 콜백 함수에 최초로 전달된다.

```
const sum = [1, 2, 3, 4, 5].reduce(function (pre, cur) {
  return pre + cur;
}, 5);

console.log(sum); // 20
// 5 + 1 => 6 + 2 => 8 + 3 => 11 + 4 => 15 + 5
```

객체의 프로퍼티 값을 합산하는 경우를 생각해 보자.

```
const products = [
  { id: 1, price: 100 },
  { id: 2, price: 200 },
  { id: 3, price: 300 }
];

// 프로퍼티 값을 합산
const priceSum = products.reduce(function (pre, cur) {
  console.log(pre.price, cur.price);
  // 숫자값이 두번째 콜백 함수 호출의 인수로 전달된다. 이때 pre.price는 undefined이다.
  return pre.price + cur.price;
});

console.log(priceSum); // NaN
```

이처럼 객체의 프로퍼티 값을 합산하는 경우에는 반드시 초기값을 전달해야 한다.

```
const products = [
  { id: 1, price: 100 },
```

```
  { id: 2, price: 200 },
  { id: 3, price: 300 }
];

// 프로퍼티 값을 합산
const priceSum = products.reduce(function (pre, cur) {
  console.log(pre, cur.price);
  return pre + cur.price;
}, 0);

console.log(priceSum); // 600
```

reduce로 빈 배열을 호출하면 에러가 발생한다.

```
const sum = [].reduce(function (pre, cur) {
  console.log(pre, cur);
  return pre + cur;
});
// TypeError: Reduce of empty array with no initial value
```

초기값을 전달하면 에러를 회피할 수 있다. 따라서 reduce를 호출할 때는 **언제나 초기값을 전달하는 것이 보다 안전하다.**

```
const sum = [].reduce(function (pre, cur) {
  console.log(pre, cur);
  return pre + cur;
}, 0);

console.log(sum); // 0
```

Array.prototype.some(callback: (value: T, index: number, array: Array) => boolean, thisArg?: any): boolean

배열 내 일부 요소가 콜백 함수의 테스트를 통과하는지 확인하여 그 결과를 boolean으로 반환한다. IE 9 이상에서 정상 동작한다.

콜백함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.

```
// 배열 내 요소 중 10보다 큰 값이 1개 이상 존재하는지 확인
let res = [2, 5, 8, 1, 4].some(function (item) {
  return item > 10;
});
console.log(res); // false

res = [12, 5, 8, 1, 4].some(function (item) {
  return item > 10;
});
console.log(res); // true

// 배열 내 요소 중 특정 값이 1개 이상 존재하는지 확인
res = ['apple', 'banana', 'mango'].some(function (item) {
  return item === 'banana';
});
console.log(res); // true
```

some()도 map(), forEach()와 같이 두번째 인자로 this를 전달할 수 있다.

Array.prototype.every(callback: (value: T, index: number, array: Array) => boolean, thisArg?: any): boolean 🔒 ES5

배열 내 모든 요소가 콜백함수의 테스트를 통과하는지 확인하여 그 결과를 boolean으로 반환한다. IE 9 이상에서 정상 동작한다.

콜백함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.

```
// 배열 내 모든 요소가 10보다 큰 값인지 확인
let res = [21, 15, 89, 1, 44].every(function (item) {
  return item > 10;
});
console.log(res); // false

res = [21, 15, 89, 100, 44].every(function (item) {
  return item > 10;
});
console.log(res); // true
```

every()도 map(), forEach()와 같이 두번째 인자로 this를 전달할 수 있다.

Array.prototype.find(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): T | undefined

ES6에서 새롭게 도입된 메소드로 Internet Explorer에서는 지원하지 않는다.

배열을 순회하며 각 요소에 대하여 인자로 주어진 **콜백함수를 실행하여 그 결과가 참인 첫번째 요소를 반환한다.** 콜백함수의 실행 결과가 참인 요소가 존재하지 않는다면 `undefined`를 반환한다.

콜백함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.

참고로 filter는 **콜백함수의 실행 결과가 true인 배열 요소의 값만을 추출한 새로운 배열을 반환한다.** 따라서 filter의 반환값은 언제나 배열이다. 하지만 find는 **콜백함수를 실행하여 그 결과가 참인 첫번째 요소를 반환**하므로 find의 결과값은 해당 요소값이다.

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// 콜백함수를 실행하여 그 결과가 참인 첫번째 요소를 반환한다.
let result = users.find(function (item) {
  return item.id === 2;
});

// ES6
// const result = users.find(item => item.id === 2;);

// Array#find는 배열이 아니라 요소를 반환한다.
console.log(result); // { id: 2, name: 'Kim' }

// Array#filter는 콜백함수의 실행 결과가 true인 배열 요소의 값만을 추출한 새로운 배열을 반환한다.
result = users.filter(function (item) {
  return item.id === 2;
});

console.log(result); // [ { id: 2, name: 'Kim' },{ id: 2, name: 'Choi' } ]
```

find 메소드의 이해를 돕기 위해 find의 동작을 흉내낸 myFind 메소드를 작성해 보자.

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

Array.prototype.myFind = function (predicate) {
  // 첫번재 매개변수에 함수가 전달되었는지 확인
  if (!predicate || {}.toString.call(predicate) !== '[object Function]') {
    throw new TypeError(predicate + ' is not a function.');
  }

  /**
   * 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 매개변수를 통해 predicate에 전달하고
   * predicate를 호출하여 그 결과가 참인 요소를 반환하고 처리를 종료한다.
   */
  for (let i = 0, len = this.length; i < len; i++) {
    if (predicate(this[i], i, this)) return this[i];
  }
};

const result = users.myFind(function (item, index, array) {
  console.log(`[${index}]: ${JSON.stringify(item)} of [${JSON.stringify(array)}]`);
  return item.id === 2; // 요소의 id 프로퍼티의 값이 2인 요소를 검색
});

console.log(result); // { id: 2, name: 'Kim' }
```

Array.prototype.findIndex(predicate: (value: T, index: number, obj: T[]) => boolean, thisArg?: any): number

ES6에서 새롭게 도입된 메소드로 Internet Explorer에서는 지원하지 않는다.

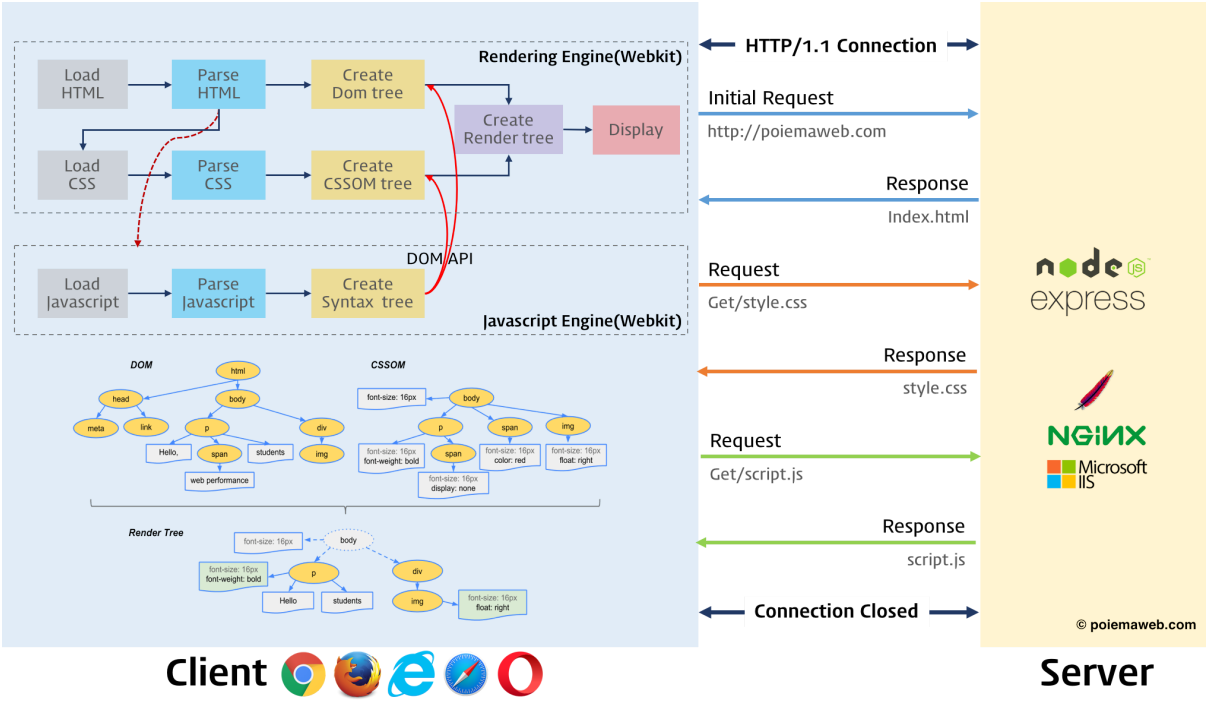
배열을 순회하며 각 요소에 대하여 인자로 주어진 **콜백함수를 실행하여 그 결과가 참인 첫번째 요소의 인덱스를 반환한다.** 콜백함수의 실행 결과가 참인 요소가 존재하지 않는다면 -1을 반환한다.

콜백함수의 매개변수를 통해 배열 요소의 값, 요소 인덱스, 메소드를 호출한 배열, 즉 this를 전달 받을 수 있다.

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// 콜백함수를 실행하여 그 결과가 참인 첫번째 요소의 인덱스를 반환한다.
function predicate(key, value) {
  return function (item) {
    return item[key] === value;
  };
}

// id가 2인 요소의 인덱스
let index = users.findIndex(predicate('id', 2));
console.log(index); // 1

// name이 'Park'인 요소의 인덱스
index = users.findIndex(predicate('name', 'Park'));
console.log(index); // 3
```

DOM (Document Object Model

브라우저의 렌더링 엔진은 웹 문서를 로드한 후, 파싱하여 웹 문서를 브라우저가 이해할 수 있는 구조로 구성하여 메모리에 적재하는데 이를 DOM이라한다. 즉 모든 요소와 요소의 어트리뷰트. 텍스르를 각각의 객체로 만들고 이들 객체를 부자 관계를 표현할 수 있는 트리구조로 구성한것이 DOM이다.



브라우저는 웹 문서(HTML, XML, SVG)를 로드한 후, 파싱하여 DOM(문서 객체 모델)을 생성한다.

DOM은 프로그래밍 언어가 자신에 접근하고 수정할 수 있는 방법을 제공하는데 일반적으로 프로퍼티와 메소드를 갖는 자바스크립트 객체로 제공된다. 이를 DOM API라고 부른다.

DOM 기능

HTML 문서에 대한 모델 구성

브라우저는 HTML 문서를 로드한 후 해당 문서에 대한 모델을 메모리에 생성한다. 이때 모델은 객체의 트리로 구성되는데 이것을 **DOM tree**라 한다.

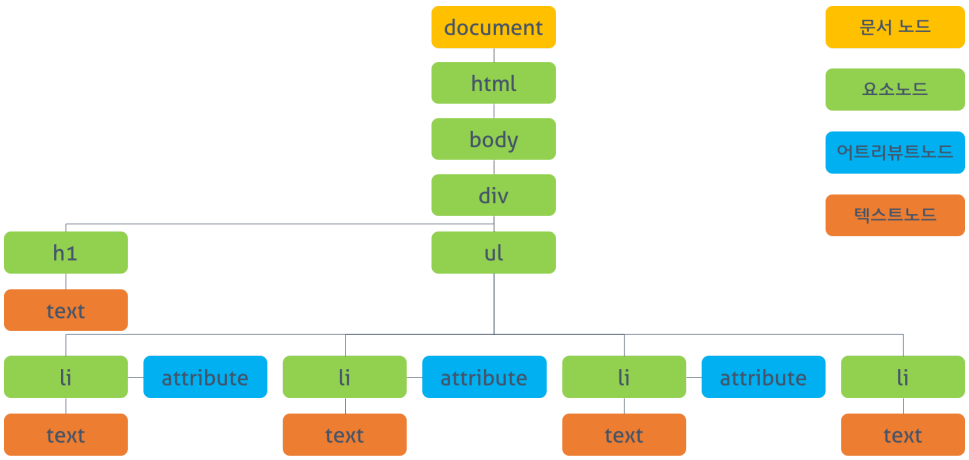
HTML 문서 내의 각 요소에 접근 / 수정

DOM은 모델 내의 각 객체에 접근하고 수정할 수 있는 프로퍼티와 메소드를 제공한다. DOM이 수정되면 브라우저를 통해 사용자가 보게 될 내용 또한 변경된다.

DOM tree

브라우저가 HTML문서를 로드한 후 파싱하여 생성하는 모데를 의미한다. 객체의 트리로 구조화되어 있기 때문에 DOM tree라 부른다.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .red  { color: #ff0000; }
      .blue { color: #0000ff; }
    </style>
  </head>
  <body>
    <div>
      <h1>Cities</h1>
      <ul>
        <li id="one" class="red">Seoul</li>
        <li id="two" class="red">London</li>
        <li id="three" class="red">Newyork</li>
        <li id="four">Tokyo</li>
      </ul>
    </div>
  </body>
</html>
```



DOM tree 노드

문서 노드(Document Node)트리의 최상위에 존재하며 각각 요소, 어트리뷰트, 텍스트 노드에 접근하려면 문서 노드를 통해야 한다. 즉, DOM tree에 접근하기 위한 시작점(entry point)이다.

요소 노드(Element Node)요소 노드는 HTML 요소를 표현한다. HTML 요소는 중첩에 의해 부자 관계를 가지며 이 부자 관계를 통해 정보를 구조화한다. 따라서 요소 노드는 문서의 구조를 서술한다고 말 할 수 있다. 어트리뷰트, 텍스트 노드에 접근하려면 먼저 요소 노드를 찾아 접근해야 한다. 모든 요소 노드는 요소별 특성을 표현하기 위해 HTMLElement 객체를 상속한 객체로 구성된다. (그림: DOM tree의 객체 구성 참고)

어트리뷰트 노드(Attribute Node)어트리뷰트 노드는 HTML 요소의 어트리뷰트를 표현한다. 어트리뷰트 노드는 해당 어트리뷰트가 지정된 요소의 자식이 아니라 해당 요소의 일부로 표현된다. 따라서 해당 요소 노드를 찾아 접근하면 어트리뷰트를 참조, 수정할 수 있다.

텍스트 노드(Text Node)텍스트 노드는 HTML 요소의 텍스트를 표현한다. 텍스트 노드는 요소 노드의 자식이며 자신의 자식 노드를 가질 수 없다. 즉, 텍스트 노드는 DOM tree의 최종단이다.

DOM을 통해 웹페이지를 조작(manipulate)하기 위해서는 다음과 같은 수순이 필요하다.

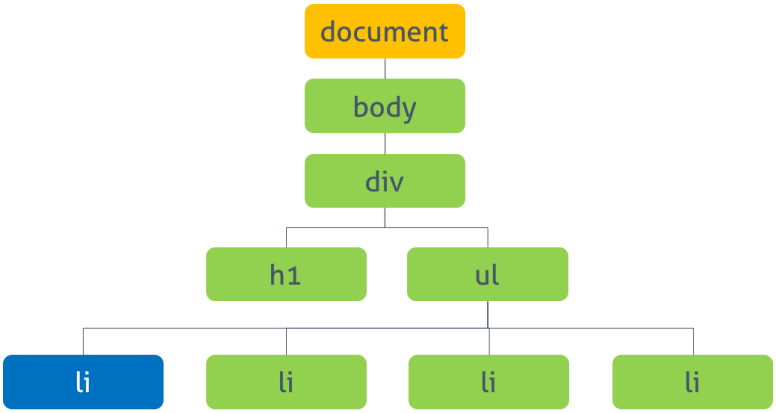
- 조작하고자하는 요소를 선택 또는 탐색한다.

- 선택된 요소의 콘텐츠 또는 어트리뷰트를 조작한다.

자바스크립트는 이것에 필요한 수단(API)을 제공한다.

DOM Query / Traversing (요소에의 접근)

하나의 요소 노드 선택(DOM Query)



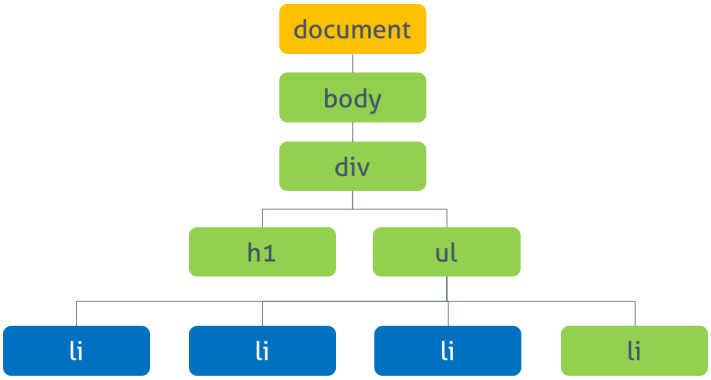
document.getElementById(id)
• id 어트리뷰트 값으로 요소 노드를 한 개 선택한다. 복수개가 선택된 경우, 첫번째 요소만 반환한다.
• Return: HTMLElement를 상속받은 객체
• 모든 브라우저에서 동작

```
// id로 하나의 요소를 선택한다.
const elem = document.getElementById('one');
// 클래스 어트리뷰트의 값을 변경한다.
elem.className = 'blue';

// 그림: DOM tree의 객체 구성 참고
console.log(elem); // <li id="one" class="blue">Seoul</li>
console.log(elem.__proto__);           // HTMLLIElement
console.log(elem.__proto__.__proto__); // HTMLElement
console.log(elem.__proto__.__proto__.__proto__);           // Element
console.log(elem.__proto__.__proto__.__proto__.__proto__); // Node
```

document.querySelector(cssSelector)
• CSS 셀렉터를 사용하여 요소 노드를 한 개 선택한다. 복수개가 선택된 경우, 첫번째 요소만 반환한다.
• Return: HTMLElement를 상속받은 객체
• IE8 이상의 브라우저에서 동작

```
// CSS 셀렉터를 이용해 요소를 선택한다
const elem = document.querySelector('li.red');
// 클래스 어트리뷰트의 값을 변경한다.
elem.className = 'blue';
```

여러 개의 요소 노드 선택



document.getElementsByClassName(class)

• class 어트리뷰트 값으로 요소 노드를 모두 선택한다. 공백으로 구분하여 여러 개의 class를 지정할 수 있다.
• Return: HTMLCollection (live)
• IE9 이상의 브라우저에서 동작

```
// HTMLCollection을 반환한다. HTMLCollection은 live하다.
const elems = document.getElementsByClassName('red');

for (let i = 0; i < elems.length; i++) {
  // 클래스 어트리뷰트의 값을 변경한다.
  elems[i].className = 'blue';
}
```

위 예제를 실행해 보면 예상대로 동작하지 않는다. (두번째 요소만 클래스 변경이 되지 않는다.)

위 예제가 예상대로 동작하지 않은 이유를 알아보자.

elems.length는 3이므로 3번의 loop가 실행된다.

1. i가 0일때, elems의 첫 요소(li#one.red)의 class 어트리뷰트의 값이 className 프로퍼티에 의해 red에서 blue로 변경된다. 이때 elems는 실시간으로 Node의 상태 변경을 반영하는 HTMLCollection 객체이다. elems의 첫 요소는 li#one.red에서 li#one.blue로 변경되었으므로 getElementsByClassName 메소드의 인자로 지정한 선택 조건('red')과 더이상 부합하지 않게 되어 반환값에서 실시간으로 제거된다.
2. i가 1일때, elems에서 첫째 요소는 제거되었으므로 elems[1]은 3번째 요소(li#three.red)가 된다. li#three.red의 class 어트리뷰트 값이 blue로 변경되고 마찬가지로 HTMLCollection에서 제외된다.
3. i가 2일때, HTMLCollection의 1,3번째 요소가 실시간으로 제거되었으므로 2번째 요소(li#two.red)만 남았다. 이때 elems.length는 1이므로 for 문의 조건식 i < elems.length가 false로 평가되어 반복을 종료한다. 따라서 elems에 남아 있는 2번째 li 요소(li#two.red)의 class 값은 변경되지 않는다.

회피방법

- 반복문을 역방향으로 돌린다.

```
const elems = document.getElementsByClassName('red');

for (let i = elems.length - 1; i >= 0; i--) {
  elems[i].className = 'blue';
}
```

- while 반복문을 사용한다. 이때 elems에 요소가 남아 있지 않을 때까지 무한반복하기 위해 index는 0으로 고정시킨다.

```
const elems = document.getElementsByClassName('red');

let i = 0;
while (elems.length > i) { // elems에 요소가 남아 있지 않을 때까지 무한반복
  elems[i].className = 'blue';
  // i++;
}
```

- HTMLCollection을 배열로 변경한다. 이 방법을 권장한다.

```
const elems = document.getElementsByClassName('red');

// 유사 배열 객체인 HTMLCollection을 배열로 변환한다.
// 배열로 변환된 HTMLCollection은 더 이상 live하지 않다.
console.log([...elems]); // [li#one.red, li#two.red, li#three.red]

[...elems].forEach(elem => elem.className = 'blue');
```

- querySelectorAll 메소드를 사용하여 HTMLCollection(live)이 아닌 NodeList(non-live)를 반환하게 한다.

```
// querySelectorAll는 Nodelist(non-live)를 반환한다. IE8+
const elems = document.querySelectorAll('.red');

[...elems].forEach(elem => elem.className = 'blue');
```

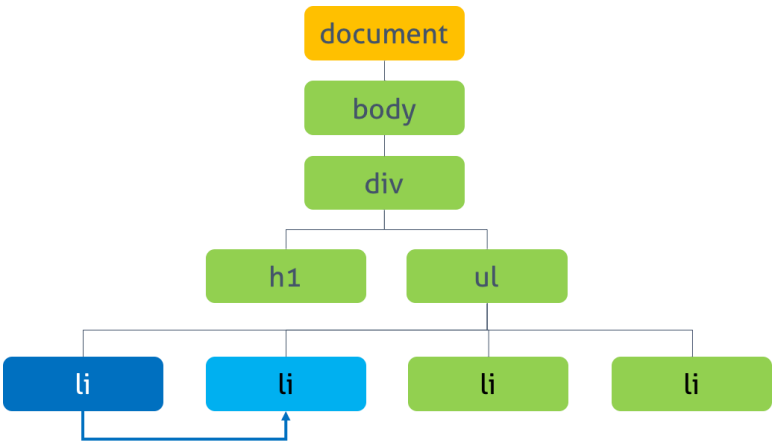
document.getElementsByTagName(tagName)
• 태그명으로 요소 노드를 모두 선택한다.
• Return: HTMLCollection (live)
• 모든 브라우저에서 동작

```
// HTMLCollection을 반환한다.
const elems = document.getElementsByTagName('li');

[...elems].forEach(elem => elem.className = 'blue');
```

document.querySelectorAll(selector)
• 지정된 CSS 선택자를 사용하여 요소 노드를 모두 선택한다.
• Return: NodeList (non-live)
• IE8 이상의 브라우저에서 동작

```
// Nodelist를 반환한다.
const elems = document.querySelectorAll('li.red');

[...elems].forEach(elem => elem.className = 'blue');
```

DOM Traversing (탐색)



DOM Traversing: nextSibling

parentNode
• 부모 노드를 탐색한다.
• Return: HTMLElement를 상속받은 객체
• 모든 브라우저에서 동작

```
const elem = document.querySelector('#two');

elem.parentNode.className = 'blue';
```

firstChild, lastChild
• 자식 노드를 탐색한다.
• Return: HTMLElement를 상속받은 객체
• IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

// first Child
elem.firstChild.className = 'blue';
// last Child
elem.lastChild.className = 'blue';
```

위 예제를 실행해 보면 예상대로 동작하지 않는다. 그 이유는 IE를 제외한 대부분의 브라우저들은 요소 사이의 공백 또는 줄바꿈 문자를 텍스트 노드로 취급하기 때문이다. 이것을 회피하기 위해서는 아래와 같이 HTML의 공백을 제거하거나 jQuery: .prev()와 jQuery: .next()를 사용한다.

```
<ul><li
  id='one' class='red'>Seoul</li><li
  id='two' class='red'>London</li><li
  id='three' class='red'>Newyork</li><li
  id='four'>Tokyo</li></ul>
```

또는 firstElementChild, lastElementChild를 사용할 수도 있다. 이 두가지 프로퍼티는 모든 IE9 이상에서 정상 동작한다.

```
const elem = document.querySelector('ul');

// first Child
elem.firstElementChild.className = 'blue';
// last Child
elem.lastElementChild.className = 'blue';
```

hasChildNodes()
• 자식 노드가 있는지 확인하고 Boolean 값을 반환한다.
• Return: Boolean 값
• 모든 브라우저에서 동작**childNodes**
• 자식 노드의 컬렉션을 반환한다. **텍스트 요소를 포함한 모든 자식 요소를 반환한다.**
• Return: NodeList (non-live)
• 모든 브라우저에서 동작**children**
• 자식 노드의 컬렉션을 반환한다. **자식 요소 중에서 Element type 요소만을 반환한다.**
• Return: HTMLCollection (live)
• IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

if (elem.hasChildNodes()) {
  console.log(elem.childNodes);
  // 텍스트 요소를 포함한 모든 자식 요소를 반환한다.
  // NodeList(9) [text, li#one.red, text, li#two.red, text, li#three.red, text, li#four, text]

  console.log(elem.children);
  // 자식 요소 중에서 Element type 요소만을 반환한다.
  // HTMLCollection(4) [li#one.red, li#two.red, li#three.red, li#four, one: li#one.red, two: li#two.red, three: li#three.red, four:
li#four]
  [...elem.children].forEach(el => console.log(el.nodeType)); // 1 (=> Element node)
}
```

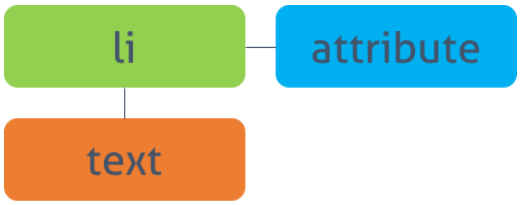
previousSibling, nextSibling
• 형제 노드를 탐색한다. **text node를 포함한 모든 형제 노드를 탐색한다.**
• Return: HTMLElement를 상속받은 객체
• 모든 브라우저에서 동작**previousElementSibling, nextElementSibling**

• 형제 노드를 탐색한다. **형제 노드 중에서 Element type 요소만을 탐색한다.**
• Return: HTMLElement를 상속받은 객체
• IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

elem.firstElementChild.nextElementSibling.className = 'blue';
elem.firstElementChild.nextElementSibling.previousElementSibling.className = 'blue';
```

DOM Manipulation (조작)

텍스트 노드에의 접근/수정



요소의 텍스트는 텍스트 노드에 저장되어 있다. 텍스트 노드에 접근하려면 아래와 같은 수순이 필요하다.

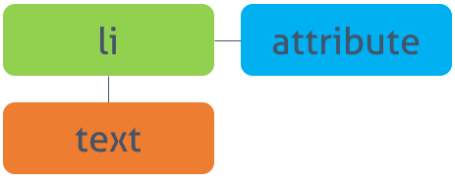
1. 해당 텍스트 노드의 부모 노드를 선택한다. 텍스트 노드는 요소 노드의 자식이다.
2. firstChild 프로퍼티를 사용하여 텍스트 노드를 탐색한다.
3. 텍스트 노드의 유일한 프로퍼티(`nodeValue`)를 이용하여 텍스트를 취득한다.
4. `nodeValue`를 이용하여 텍스트를 수정한다.

nodeValue
• 노드의 값을 반환한다.
• Return: 텍스트 노드의 경우는 문자열, 요소 노드의 경우 null 반환
• IE6 이상의 브라우저에서 동작한다.

nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.

```
// 해당 텍스트 노드의 부모 요소 노드를 선택한다.
const one = document.getElementById('one');
console.dir(one); // HTMLLIElement: li#one.red

// nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.
console.log(one.nodeName); // LI
console.log(one.nodeType); // 1: Element node

// firstChild 프로퍼티를 사용하여 텍스트 노드를 탐색한다.
const textNode = one.firstChild;

// nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.
console.log(textNode.nodeName); // #text
console.log(textNode.nodeType); // 3: Text node

// nodeValue 프로퍼티를 사용하여 노드의 값을 취득한다.
console.log(textNode.nodeValue); // Seoul

// nodeValue 프로퍼티를 이용하여 텍스트를 수정한다.
textNode.nodeValue = 'Pusan';
```

어트리뷰트 노드에의 접근/수정



어트리뷰트 노드을 조작할 때 다음 프로퍼티 또는 메소드를 사용할 수 있다.

className
• class 어트리뷰트의 값을 취득 또는 변경한다. className 프로퍼티에 값을 할당하는 경우, class 어트리뷰트가 존재하지 않으면 class 어트리뷰트를 생성하고 지정된 값을 설정한다. class 어트리뷰트의 값이 여러 개일 경우, 공백으로 구분된 문자열이 반환되므로 String 메소드 `split(' ')`를 사용하여 배열로 변경하여 사용한다.
• 모든 브라우저에서 동작한다.**classList**
• add, remove, item, toggle, contains, replace 메소드를 제공한다.
• IE10 이상의 브라우저에서 동작한다.

```
const elems = document.querySelectorAll('li');

// className
[...elems].forEach(elem => {
  // class 어트리뷰트 값을 취득하여 확인
  if (elem.className === 'red') {
    // class 어트리뷰트 값을 변경한다.
    elem.className = 'blue';
  }
});

// classList
[...elems].forEach(elem => {
  // class 어트리뷰트 값 확인
  if (elem.classList.contains('blue')) {
    // class 어트리뷰트 값 변경한다.
    elem.classList.replace('blue', 'red');
  }
});
```

id
• id 어트리뷰트의 값을 취득 또는 변경한다. id 프로퍼티에 값을 할당하는 경우, id 어트리뷰트가 존재하지 않으면 id 어트리뷰트를 생성하고 지정된 값을 설정한다.
• 모든 브라우저에서 동작한다.

```
// h1 태그 요소 중 첫번째 요소를 취득
const heading = document.querySelector('h1');

console.dir(heading); // HTMLHeadingElement: h1
console.log(heading.firstChild.nodeValue); // Cities

// id 어트리뷰트의 값을 변경.
// id 어트리뷰트가 존재하지 않으면 id 어트리뷰트를 생성하고 지정된 값을 설정
heading.id = 'heading';
console.log(heading.id); // heading
```

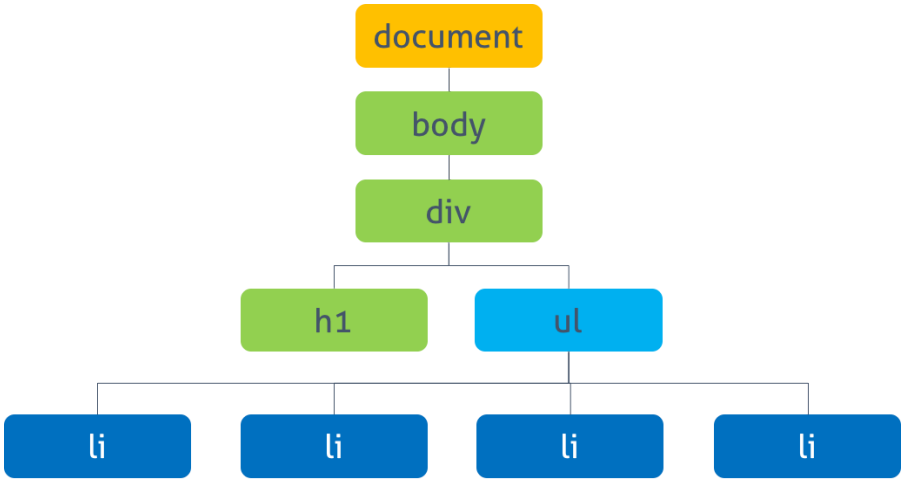
hasAttribute(attribute)
• 지정한 어트리뷰트를 가지고 있는지 검사한다.
• Return : Boolean
• IE8 이상의 브라우저에서 동작한다.**getAttribute(attribute)**
• 어트리뷰트의 값을 취득한다.
• Return : 문자열
• 모든 브라우저에서 동작한다.**setAttribute(attribute, value)**
• 어트리뷰트와 어트리뷰트 값을 설정한다.
• Return : undefined
• 모든 브라우저에서 동작한다.**removeAttribute(attribute)**
• 지정한 어트리뷰트를 제거한다.
• Return : undefined
• 모든 브라우저에서 동작한다.

```
<!DOCTYPE html>
<html>
  <body>
  <input type="text">
  <script>
  const input = document.querySelector('input[type=text]');
  console.log(input);

  // value 어트리뷰트가 존재하지 않으면
```

```
  if (!input.hasAttribute('value')) {
    // value 어트리뷰트를 추가하고 값으로 'hello!'를 설정
    input.setAttribute('value', 'hello!');
  }

  // value 어트리뷰트 값을 취득
  console.log(input.getAttribute('value')); // hello!

  // value 어트리뷰트를 제거
  input.removeAttribute('value');

  // value 어트리뷰트의 존재를 확인
  console.log(input.hasAttribute('value')); // false
  </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
  <input class="password" type="password" value="123">
  <button class="show">show</button>
  <script>
    const $password = document.querySelector('.password');
    const $show = document.querySelector('.show');

    function makeClickHandler() {
      let isShow = false;
      return function () {
        $password.setAttribute('type', isShow ? 'password' : 'text');
        isShow = !isShow;
        $show.innerHTML = isShow ? 'hide' : 'show';
      };
    }

    $show.onclick = makeClickHandler();
  </script>
</body>
</html>
```

HTML 콘텐츠 조작(Manipulation)



HTML 콘텐츠를 조작(Manipulation)하기 위해 아래의 프로퍼티 또는 메소드를 사용할 수 있다. 마크업이 포함된 콘텐츠를 추가하는 행위는 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하므로 주의가 필요하다.

textContent

• 요소의 텍스트 콘텐츠를 취득 또는 변경한다. 이때 마크업은 무시된다. textContent를 통해 요소에 새로운 텍스트를 할당하면 텍스트를 변경할 수 있다. 이때 순수한 텍스트만 지정해야 하며 마크업을 포함시키면 문자열로 인식되어 그대로 출력된다.
• IE9 이상의 브라우저에서 동작한다.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
```

```
      .red  { color: #ff0000; }
      .blue { color: #0000ff; }
    </style>
  </head>
  <body>
    <div>
      <h1>Cities</h1>
      <ul>
        <li id="one" class="red">Seoul</li>
        <li id="two" class="red">London</li>
        <li id="three" class="red">Newyork</li>
        <li id="four">Tokyo</li>
      </ul>
    </div>
    <script>
    const ul = document.querySelector('ul');

    // 요소의 텍스트 취득
    console.log(ul.textContent);
    /*
            Seoul
            London
            Newyork
            Tokyo
    */

    const one = document.getElementById('one');

    // 요소의 텍스트 취득
    console.log(one.textContent); // Seoul

    // 요소의 텍스트 변경
    one.textContent += ', Korea';

    console.log(one.textContent); // Seoul, Korea

    // 요소의 마크업이 포함된 콘텐츠 변경.
    one.textContent = '<h1>Heading</h1>';

    // 마크업이 문자열로 표시된다.
    console.log(one.textContent); // <h1>Heading</h1>
    </script>
  </body>
</html>
```

innerText

• innerText 프로퍼티를 사용하여도 요소의 텍스트 콘텐츠에만 접근할 수 있다. 하지만 아래의 이유로 사용하지 않는 것이 좋다.
  ◦ 비표준이다.
  ◦ CSS에 순종적이다. 예를 들어 CSS에 의해 비표시(visibility: hidden;)로 지정되어 있다면 텍스트가 반환되지 않는다.
  ◦ CSS를 고려해야 하므로 textContent 프로퍼티보다 느리다**innerHTML**
• 해당 요소의 모든 자식 요소를 포함하는 모든 콘텐츠를 하나의 문자열로 취득할 수 있다. 이 문자열은 마크업을 포함한다.

```
const ul = document.querySelector('ul');

// innerHTML 프로퍼티는 모든 자식 요소를 포함하는 모든 콘텐츠를 하나의 문자열로 취득할 수 있다. 이 문자열은 마크업을 포함한다.
console.log(ul.innerHTML);
// IE를 제외한 대부분의 브라우저들은 요소 사이의 공백 또는 줄바꿈 문자를 텍스트 노드로 취급한다
/*
        <li id="one" class="red">Seoul</li>
        <li id="two" class="red">London</li>
        <li id="three" class="red">Newyork</li>
        <li id="four">Tokyo</li>
*/
```

innerHTML 프로퍼티를 사용하여 마크업이 포함된 새로운 콘텐츠를 지정하면 새로운 요소를 DOM에 추가할 수 있다.

```
const one = document.getElementById('one');

// 마크업이 포함된 콘텐츠 취득
console.log(one.innerHTML); // Seoul

// 마크업이 포함된 콘텐츠 변경
one.innerHTML += '<em class="blue">, Korea</em>';

// 마크업이 포함된 콘텐츠 취득
console.log(one.innerHTML); // Seoul <em class="blue">, Korea</em>
```

위와 같이 마크업이 포함된 콘텐츠를 추가하는 것은 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하다.

```
// 크로스 스크립팅 공격 사례

// 스크립트 태그를 추가하여 자바스크립트가 실행되도록 한다.
// HTML5에서 innerHTML로 삽입된 <script> 코드는 실행되지 않는다.
// 크롬, 파이어폭스 등의 브라우저나 최신 브라우저 환경에서는 작동하지 않을 수도 있다.
elem.innerHTML = '<script>alert("XSS!")</script>';

// 에러 이벤트를 발생시켜 스크립트가 실행되도록 한다.
// 크롬에서도 실행된다!
elem.innerHTML = '<img src="#" onerror="alert(\'XSS\')">';
```

DOM 조작 방식

innerHTML 프로퍼티를 사용하지 않고 새로운 콘텐츠를 추가할 수 있는 방법은 DOM을 직접 조작하는 것이다. 한 개의 요소를 추가하는 경우 사용한다. 이 방법은 다음의 수순에 따라 진행한다.

1. 요소 노드 생성 createElement() 메소드를 사용하여 새로운 요소 노드를 생성한다. createElement() 메소드의 인자로 태그 이름을 전달한다.
2. 텍스트 노드 생성 createTextNode() 메소드를 사용하여 새로운 텍스트 노드를 생성한다. 경우에 따라 생략될 수 있지만 생략하는 경우, 콘텐츠가 비어 있는 요소가 된다.
3. 생성된 요소를 DOM에 추가 appendChild() 메소드를 사용하여 생성된 노드를 DOM tree에 추가한다. 또는 removeChild() 메소드를 사용하여 DOM tree에서 노드를 삭제할 수도 있다.

createElement(tagName)
• 태그이름을 인자로 전달하여 요소를 생성한다.
• Return: HTMLElement를 상속받은 객체
• 모든 브라우저에서 동작한다.**createTextNode(text)**
• 텍스트를 인자로 전달하여 텍스트 노드를 생성한다.
• Return: Text 객체
• 모든 브라우저에서 동작한다.**appendChild(Node)**
• 인자로 전달한 노드를 마지막 자식 요소로 DOM 트리에 추가한다.
• Return: 추가한 노드
• 모든 브라우저에서 동작한다.**removeChild(Node)**
• 인자로 전달한 노드를 DOM 트리에 제거한다.
• Return: 추가한 노드
• 모든 브라우저에서 동작한다.

```
// 태그이름을 인자로 전달하여 요소를 생성
const newElem = document.createElement('li');
// const newElem = document.createElement('<li>test</li>');
// Uncaught DOMException: Failed to execute 'createElement' on 'Document': The tag name provided ('<li>test</li>') is not a valid n
ame.

// 텍스트 노드를 생성
const newText = document.createTextNode('Beijing');

// 텍스트 노드를 newElem 자식으로 DOM 트리에 추가
newElem.appendChild(newText);

const container = document.querySelector('ul');

// newElem을 container의 자식으로 DOM 트리에 추가. 마지막 요소로 추가된다.
container.appendChild(newElem);

const removeElem = document.getElementById('one');

// container의 자식인 removeElem 요소를 DOM 트리에 제거한다.
container.removeChild(removeElem);
```

insertAdjacentHTML()

insertAdjacentHTML(position, string)

• 인자로 전달한 텍스트를 HTML로 파싱하고 그 결과로 생성된 노드를 DOM 트리의 지정된 위치에 삽입한다. 첫번째 인자는 삽입 위치, 두번째 인자는 삽입할 요소를 표현한 문자열이다. 첫번째 인자로 올 수 있는 값은 아래와 같다.

    - ‘beforebegin’
    - ‘afterbegin’
    - ‘beforeend’
    - ‘afterend’
- 모든 브라우저에서 동작한다.

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

```
const one = document.getElementById('one');

// 마크업이 포함된 요소 추가
one.insertAdjacentHTML('beforeend', '<em class="blue">, Korea</em>');
```

innerHTML vs DOM 조작방식 vs insertAdjacentHTML()

innerHTML

장점	단점
DOM 조작 방식에 비해 빠르고 간편하다.	XSS공격에 취약점이 있기 때문에 사용자로부터 입력받은 콘텐츠를 추가할때 주의하여야 한다.
간편하게 문자열로 정의한 여러 요소를 DOM에 추가 할 수 있다.	해당 요소의 내용을 덮어 쓴다. 즉 HTML을 다시 피싱한다. 이것은 비효율적이다.
콘텐츠를 취득 할 수 있다.	

DOM 조작 방식

장점	단점
특정 노드 한개를 DOM에 추가할 때 적합하다.	innerHTML보다 느리고 더 많은 코드가 필요하다.

insertAdjacentHTML()

장점	단점
간편하게 문자열로 정의된 여러 요소를 DOM에 추가 할수있다.	XSS공격에 취약점이 있기 때문에 사용자로 부터 입력받은 콘텐츠를 추가할때 주의하여야 한다.
삽입되는 위치를 선정할 수 있다.	

결론

innerHTML과 insertAdjacentHTML()은 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하다. 따라서 untrusted data의 경우, 주의하여야 한다. 텍스트를 추가 또는 변경시에는 textContent, 새로운 요소의 추가 또는 삭제시에는 DOM 조작 방식을 사용하도록 한다.

style

style 프로퍼티를 사용하면 inline 스타일 선언을 생성한다. 특정 요소에 inline 스타일을 지정하는 경우 사용한다.

```
const four = document.getElementById('four');

// inline 스타일 선언을 생성
four.style.color = 'blue';

// font-size와 같이 '-'으로 구분되는 프로퍼티는 카멜케이스로 변환하여 사용한다.
four.style.fontSize = '2em';
```

style 프로퍼티의 값을 취득하려면 window.getComputedStyle을 사용한다. window.getComputedStyle 메소드는 인자로 주어진 요소의 모든 CSS 프로퍼티 값을 반환한다.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>style 프로퍼티 값 취득</title>
  <style>
    .box {
      width: 100px;
      height: 50px;
      background-color: red;
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <div class="box"></div>
  <script>
    const box = document.querySelector('.box');

    const width = getStyle(box, 'width');
    const height = getStyle(box, 'height');
    const backgroundColor = getStyle(box, 'background-color');
    const border = getStyle(box, 'border');

    console.log('width: ' + width);
    console.log('height: ' + height);
    console.log('backgroundColor: ' + backgroundColor);
    console.log('border: ' + border);

    /**
     * 요소에 적용된 CSS 프로퍼티를 반환한다.
     * @param {HTTPElement} elem - 대상 요소 노드.
     * @param {string} prop - 대상 CSS 프로퍼티.
     * @returns {string} CSS 프로퍼티의 값.
     */
    function getStyle(elem, prop) {
      return window.getComputedStyle(elem, null).getPropertyValue(prop);
    }
  </script>
</body>
</html>
```

동기식 처리 모델 vs 비동기식 처리 모델

동기식 처리 모델은 직렬적으로 태스크를 수행한다. 즉 테스크는 순차적으로 실행되며 어떤 작업이 수행 중이면 다음 작업은 대기하게 된다. 서버에 데이터를 요청하고 데이터가 응답될 때까지 이후 태스크들은 블로킹(blocking, 작업 중단)된다.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  console.log('func2');
  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

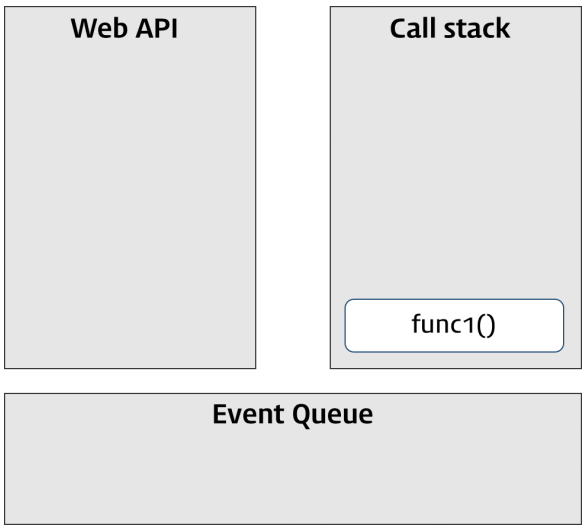
비동기식 처리 모델은 병렬적으로 태스크를 수행한다. 즉 태스크가 종료되지 않은 상태라 하더라도 대기하지 않고 다음 태스크를 실행한다. 예를들어 서버에서 데이터를 가져와서 화면에 표시하는 태스크를 수행할 때, 서버에 데이터를 요청한 이후 서버로부터 데이터가 응답될 때까지 대기하지 않고(Non-Blocking) 즉시 다음 태스크를 수행한다. 이후 서버로부터 데이터가 응답되면 이벤트가 발생하고 이벤트 핸들러가 데이터를 가지고 수행할 태스크를 계속해 수행한다.

자바스크립트의 대부분의 DOM 이벤트 핸들러와 Timer 함수(setTimeout, setInterval), Ajax 요청은 비동기식 처리 모델로 동작한다.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  setTimeout(function() {
    console.log('func2');
  }, 0);

  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

setTimeout 메소드가 비동기 함수

setTimeout의 콜백함수는 즉시 실행되지 않고 지정 대기 시간만큼 기다리다가 “tick” 이벤트가 발생하면 태스크 큐로 이동한 후 Call Stack이 비어졌을 때 Call Stack으로 이동되어 실행된다.



이벤트

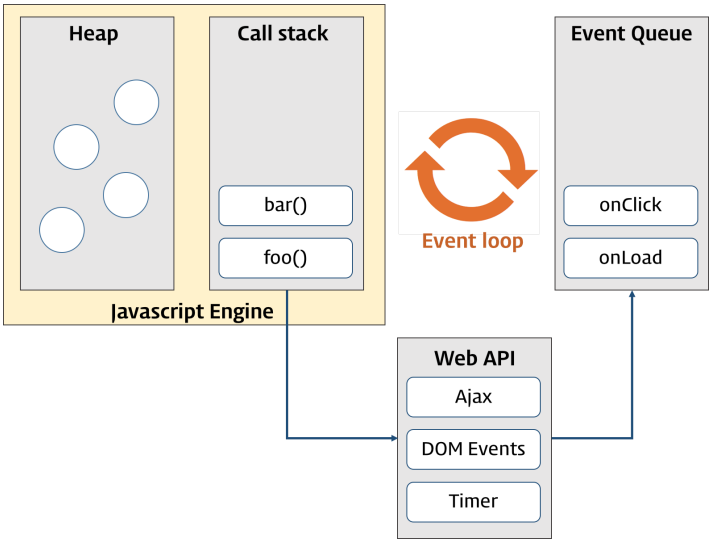
Introduction

이벤트는 어떤 사건을 의미한다. 브라우저에서의 이벤트란 예를 들어 사용자가 버튼을 클릭했을 때, 웹페이지가 로드되었을 때와 같은 것인데 이것은 DOM 요소와 관련이 있다. 이벤트가 발생하면 누군가 이를 감지할 수 있어야 하며 그에 대응하는 처리를 호출해 주어야 한다. 이벤트가 발생하면 그에 맞는 반응을 하여야 한다. 이를 위해 이벤트는 일반적으로 함수에 연결되며 그 함수는 이벤트가 발생하기 전에는 실행되지 않다가 이벤트가 발생되면 실행된다. 이 함수를 **이벤트 핸들러**라 하며 이벤트에 대응하는 처리를 기술한다.

이벤트 루프와 동시성(Concurrency)

브라우저는 단일 쓰레드에서 이벤트 드리븐 방식으로 동작한다.

단일 쓰레드는 쓰레드가 하나뿐이라는 의미이며 이 말은 곧 하나의 작업만을 처리할 수 있다는 것을 의미한다.하지만 실제로 동작하는 웹 애플리케이션은 많은 task가 동시에 처리되는 것처럼 느껴진다. 이처럼 자바스크립트의 동시성을 지원하는 것이 바로 이벤트루프 이다.



구글의 V8을 비롯한 대부분의 자바스크립트 엔진은 크게 2개의 영역으로 나뉜다.

Call Stack(호출 스택)작업이 요청되면(함수가 호출되면) 요청된 작업은 순차적으로 Call Stack에 쌓이게 되고 순차적으로 실행된다. 자바스크립트는 단 하나의 Call Stack을 사용하기 때문에 해당 task가 종료하기 전까지는 다른 어떤 task도 수행될 수 없다.**Heap**동적으로 생성된 객체 인스턴스가 할당되는 영역이다.

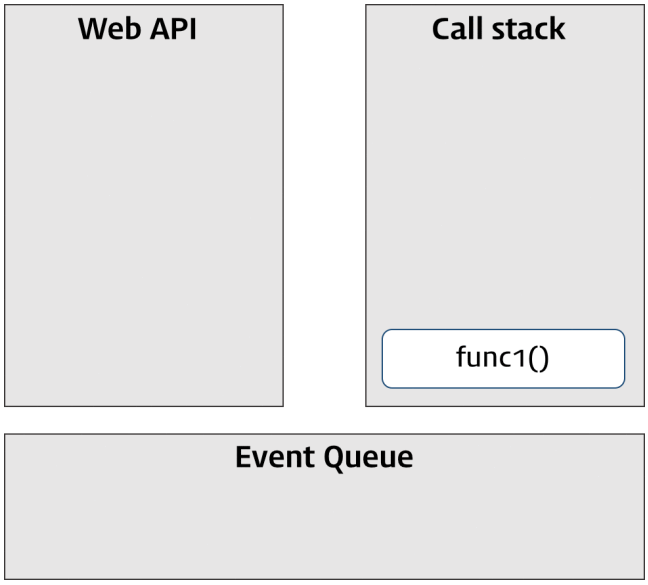
이와 같이 자바스크립트 엔진은 단순히 작업이 요청되면 Call Stack을 사용하여 요청된 작업을 순차적으로 실행할 뿐이다. 앞에서 언급한 동시성(Concurrency)을 지원하기 위해 필요한 비동기 요청(이벤트를 포함) 처리는 자바스크립트 엔진을 구동하는 환경 즉 브라우저(또는 Node.js)가 담당한다.

Event Queue(Task Queue)비동기 처리 함수의 콜백 함수, 비동기식 이벤트 핸들러, Timer 함수(setTimeout(), setInterval())의 콜백 함수가 보관되는 영역으로 **이벤트 루프(Event Loop)에 의해 특정 시점(Call Stack이 비어졌을 때)에 순차적으로 Call Stack으로 이동되어 실행**된다.**Event Loop(이벤트 루프)**Call Stack 내에서 현재 실행중인 task가 있는지 그리고 Event Queue에 task가 있는지 반복하여 확인한다. 만약 Call Stack이 비어있다면 Event Queue 내의 task가 Call Stack으로 이동하고 실행된다.

아래의 예제가 어떻게 동작할지 살펴보자.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  setTimeout(function () {
    console.log('func2');
  }, 0);

  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

함수 func1이 호출되면 함수 func1은 Call Stack에 쌓인다. 그리고 함수 func1은 함수 func2을 호출하므로 함수 func2가 Call Stack에 쌓이고 setTimeout가 호출된다. **setTimeout의 콜백함수는 즉시 실행되지 않고 지정 대기 시간만큼 기다리다가 “tick” 이벤트가 발생하면 태스크 큐로 이동한 후 Call Stack이 비어졌을 때 Call Stack으로 이동되어 실행된다.**



이벤트 루프(Event Loop)에 의한 setTimeout 콜백함수의 실행

DOM 이벤트 핸들러도 이와 같이 동작한다.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  // <button class="foo">foo</button>
  const elem = document.querySelector('.foo');

  elem.addEventListener('click', function () {
    this.style.backgroundColor = 'indigo';
    console.log('func2');
  });

  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

함수 func1이 호출되면 함수 func1은 Call Stack에 쌓인다. 그리고 함수 func1은 함수 func2을 호출하므로 함수 func2가 Call Stack에 쌓이고 addEventListener가 호출된다. **addEventListener의 콜백함수는 foo 버튼이 클릭되어 click 이벤트가 발생하면 태스크 큐로 이동한 후 Call Stack이 비어졌을 때 Call Stack으로 이동되어 실행된다.**

이벤트의 종류

UI Event

Event	Description
load	웹페이지의 로드가 완료되었을 때
unload	웹페이지가 언로드 될때(주로 새로운 페이지를 요청한 경우)
error	브라우저가 자바스크립트 오류를 만났거나 요청한 자원이 존재하지 않는 경우
resize	브라우저 창의크기를 조절했을 때
scroll	사용자가 페이지를 위아래로 스크롤 할때
select	텍스트를 선택했을 때

Keyboard Event

Event	Description
keydown	키를 누르고 있을 때
keyup	누르고 있던 키를 뗄 때
keypress	키를 누르고 뗏을 때

Mouse Event

Event	Description
click	마우스 버튼을 클릭했을 때
dbclick	마우스 버튼을 더블 클릭했을 때
mousedown	마우스 버튼을 누르고 있을 때
mouseup	누르고 있던 마우스 버튼을 땔 때
mousemove	마우스를 움직일 때(터치스크린에서 동작하지 않는다)
mouseover	마우스 요소 위로 움직였을 때 (터치스크린에서 동작하지 않는다)
mouseout	마우스 요소 밖으로 움직였을 때 (터치스크린에서 동작하지 않는다)

Focus Event

Event	Description
focus/focusin	요소가 포커스를 얻었을 때
blur/foucusout	요소가 포커스를 잃었을 때

Form Event

Event	Description
input	input 또는 textarea 요소의 값이 변경되었을 때
	contenteditable 어트리뷰트를 가진 요소의 값이 변경되었을 때
change	select box, checkbox, radio button의 상태가 변경되었을 때
submit	form을 submit할 때(버튼 또는 키)
reset	reset 버튼을 클릭할 때(최근에는 사용 안함)

Clipboard Event

Event	Description
cut	콘텐츠를 잘라내기할 때
copy	콘텐츠를 복사할 때
paste	콘텐츠를 붙여넣기할 때

이벤트 핸들러 등록

인라인 이벤트 핸들러 방식

HTML요소의 이벤트 핸들러 어트리뷰트에 이벤트 핸들러를 등록하는 방법이다.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="myHandler()">Click me</button>
  <script>
    function myHandler() {
      alert('Button clicked!');
    }
  </script>
</body>
</html>
```

이 방식은 더 이상 사용되지 않으며 사용해서도 않된다. 오래된 코드에서 간혹 이 방식을 사용한 것이 있기 때문에 알아둘 필요는 있다.
HTML과 Javascript는 관심사가 다르므로 분리하는 것이 좋다.

이벤트 핸들러 프로퍼티 방식에는 DOM 요소의 이벤트 핸들러 프로퍼티에 함수 호출이 아닌 함수를 전달한다.

이때 이벤트 어트리뷰트의 값으로 전달한 함수 호출이 즉시 호출되는 것은 아니다. 사실은 이벤트 어트리뷰트 키를 이름으로 갖는 함수를 암묵적으로 정의하고 그 함수의 몸체에 이벤트 어트리뷰트의 값으로 전달한 함수 호출을 문으로 갖는다. 위 예제의 경우, button 요소의 onclick 프로퍼티에 함수 `function onclick(event) { foo(); }`가 할당된다.

즉, 이벤트 어트리뷰트의 값은 암묵적으로 정의되는 이벤트 핸들러의 문이다.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="myHandler1(); myHandler2();">Click me</button>
  <script>
    function myHandler1() {
      alert('myHandler1');
    }
    function myHandler2() {
      alert('myHandler2');
    }
  </script>
</body>
</html>
```

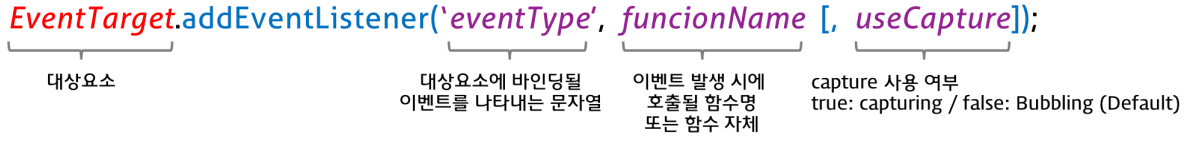
이벤트 핸들러 프로퍼티 방식

인라인 이벤트 핸들러 방식처럼 HTML과 Javascript가 뒤섞이는 문제는 해결할 수 있는 방식이다. 하지만 이벤트 핸들러 프로퍼티에 하나의 ㅁ이벤트 핸들러만을 바인딩 할수 있다는 단점이 있다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="btn">Click me</button>
  <script>
    const btn = document.querySelector('.btn');

    // 이벤트 핸들러 프로퍼티 방식은 이벤트에 하나의 이벤트 핸들러만을 바인딩할 수 있다
    // 첫번째 바인딩된 이벤트 핸들러 => 실행되지 않는다.
    btn.onclick = function () {
      alert('① Button clicked 1');
    };

    // 두번째 바인딩된 이벤트 핸들러
    btn.onclick = function () {
      alert('① Button clicked 2');
    };

    // addEventListener 메소드 방식
    // 첫번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('② Button clicked 1');
    });

    // 두번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('② Button clicked 2');
    });
  </script>
</body>
</html>
```

addEventListener 메소드 방식

`addEventListener` 메소드를 이용하여 대상 DOM 요소에 이벤트를 바인딩하고 해당 이벤트가 발생했을 때 실행될 콜백 함수(이벤트 핸들러)를 지정한다.



addEventListener 메소드

addEventListener 함수 방식은 이전 방식에 비해 아래와 같이 보다 나은 장점을 갖는다.

- 하나의 이벤트에 대해 하나 이상의 이벤트 핸들러를 추가할 수 있다.
- 캡처링과 버블링을 지원한다.
- HTML 요소뿐만아니라 모든 DOM 요소(HTML, XML, SVG)에 대해 동작한다. 브라우저는 웹 문서(HTML, XML, SVG)를 로드한 후, 파싱하여 DOM을 생성한다.

addEventListener 메소드는 IE 9 이상에서 동작한다. IE 8 이하에서는 attachEvent 메소드를 사용한다.

```
if (elem.addEventListener) {    // IE 9 ~
  elem.addEventListener('click', func);
} else if (elem.attachEvent) {  // ~ IE 8
  elem.attachEvent('onclick', func);
}
```

addEventListener 메소드의 사용 예제를 살펴보자.

```
<!DOCTYPE html>
<html>
<body>
  <script>
    addEventListener('click', function () {
      alert('Clicked!');
    });
  </script>
</body>
</html>
```

위와 같이 대상 DOM 요소(target)를 지정하지 않으면 전역객체 window, 즉 DOM 문서를 포함한 브라우저의 윈도우에서 발생하는 click 이벤트에 이벤트 핸들러를 바인딩한다. 따라서 브라우저 윈도우 어디를 클릭하여도 이벤트 핸들러가 동작한다.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>

  <script>
    const input = document.querySelector('input[type=text]');

    input.addEventListener('blur', function () {
      alert('blur event occurred!');
    });
  </script>
</body>
</html>
```

위 예제는 input 요소에서 발생하는 blur 이벤트에 이벤트 핸들러를 바인딩하였다. 사용자 이름이 최소 2자 이상이야한다는 규칙을 세우고 이에 부합하는지 확인해보자.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>
  <em class="message"></em>

  <script>
    const input = document.querySelector('input[type=text]');
    const msg = document.querySelector('.message');

    input.addEventListener('blur', function () {
      if (input.value.length < 2) {
        msg.innerHTML = '이름은 2자 이상 입력해 주세요';
      } else {
        msg.innerHTML = '';
      }
    });
  </script>
</body>
</html>
```

2자 이상이라는 규칙이 바뀌면 이 규칙을 확인하는 모든 코드를 수정해야 한다. 따라서 이러한 방식의 코딩은 바람직하지 않다. 이유는 규모가 큰 프로그램의 경우 수정과 테스트에 소요되는 자원의 낭비도 문제이지만 수정에는 거의 대부분 실수가 동반되기 때문이다.

2자 이상이라는 규칙을 상수화하고 함수의 인수로 전달도록 수정하자. 이렇게 하면 규칙이 변경되어도 함수는 수정하지 않아도 된다.

그런데 addEventListener 메소드의 두번째 매개변수는 이벤트가 발생했을 때 호출될 이벤트 핸들러이다. 이때 두번째 매개변수에는 함수 호출이 아니라 함수 자체를 지정하여야 한다.

```
function foo() {
  alert('clicked!');
}
// elem.addEventListener('click', foo()); // 이벤트 발생 시까지 대기하지 않고 바로 실행된다
elem.addEventListener('click', foo);      // 이벤트 발생 시까지 대기한다
```

따라서 이벤트 핸들러 프로퍼티 방식과 같이 이벤트 핸들러 함수에 인수를 전달할 수 없는 문제가 발생한다. 이를 우회하는 방법은 아래와 같다.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>
  <em class="message"></em>

  <script>
    const MIN_USER_NAME_LENGTH = 2; // 이름 최소 길이

    const input = document.querySelector('input[type=text]');
    const msg = document.querySelector('.message');

    function checkUserNameLength(n) {
      if (input.value.length < n) {
        msg.innerHTML = '이름은 ' + n + '자 이상이어야 합니다';
      } else {
        msg.innerHTML = '';
      }
    }

    input.addEventListener('blur', function () {
      // 이벤트 핸들러 내부에서 함수를 호출하면서 인수를 전달한다.
      checkUserNameLength(MIN_USER_NAME_LENGTH);
    });

    // 이벤트 핸들러 프로퍼티 방식도 동일한 방식으로 인수를 전달할 수 있다.
    // input.onblur = function () {
    //   // 이벤트 핸들러 내부에서 함수를 호출하면서 인수를 전달한다.
    //   checkUserNameLength(MIN_USER_NAME_LENGTH);
    // };
  </script>
</body>
</html>
```

이벤트 핸들러 함수 내부의 this

인라인 이벤트 핸들러 방식

인라인 이벤트 핸들러 방식의 경우, 이벤트 핸들러는 일반 함수로서 호출되므로 이벤트 핸들러 내부의 **this는 전역 객체 window**를 가리킨다.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="foo()">Button</button>
  <script>
    function foo () {
      console.log(this); // window
    }
  </script>
</body>
</html>
```

이벤트 핸들러 프로퍼티 방식

이벤트 핸들러 프로퍼티 방식에서 이벤트 핸들러는 메소드이므로 이벤트 핸들러 내부의 **this는 이벤트에 바인딩된 요소**를 가리킨다. 이것은 이벤트 객체의 currentTarget 프로퍼티와 같다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="btn">Button</button>
  <script>
    const btn = document.querySelector('.btn');

    btn.onclick = function (e) {
      console.log(this); // <button id="btn">Button</button>
      console.log(e.currentTarget); // <button id="btn">Button</button>
      console.log(this === e.currentTarget); // true
    };
  </script>
</body>
</html>
```

addEventListener 메소드 방식

addEventListener 메소드에서 지정한 이벤트 핸들러는 콜백 함수이지만 이벤트 핸들러 내부의 **this는 이벤트 리스너에 바인딩된 요소(currentTarget)**를 가리킨다. 이것은 이벤트 객체의 currentTarget 프로퍼티와 같다.

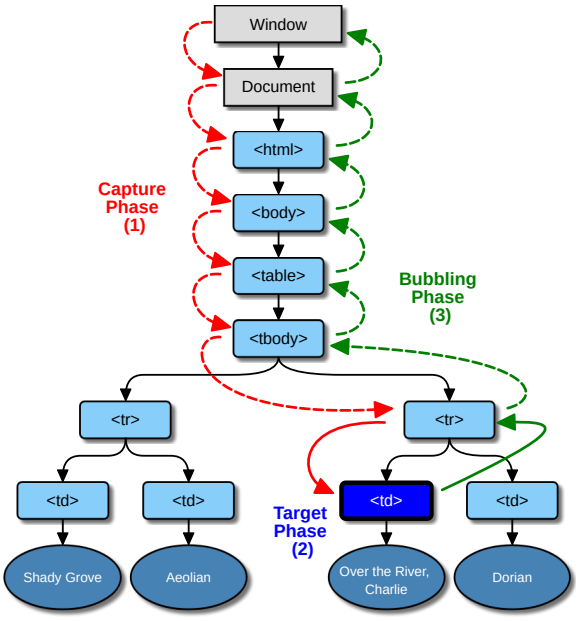
```
<!DOCTYPE html>
<html>
<body>
  <button class="btn">Button</button>
  <script>
    const btn = document.querySelector('.btn');

    btn.addEventListener('click', function (e) {
      console.log(this); // <button id="btn">Button</button>
      console.log(e.currentTarget); // <button id="btn">Button</button>
      console.log(this === e.currentTarget); // true
    });
  </script>
</body>
</html>
```

이벤트의 흐름

계층적 구조에 포함되어 있는 HTML 요소에 이벤트가 발생할 경우 연쇄적 반응이 일어난다. 즉, 이벤트가 전파(Event Propagation)되는데 전파 방향에 따라 버블링(Event Bubbling)과 캡처링(Event Capturing)으로 구분할 수 있다.

자식 요소에서 발생한 이벤트가 부모 요소로 전파되는 것을 버블링이라 하고, 자식 요소에서 발생한 이벤트가 부모 요소부터 시작하여 이벤트를 발생시킨 자식 요소까지 도달하는 것을 캡처링이라 한다. **주의할 것은 버블링과 캡처링은 둘 중에 하나만 발생하는 것이 아니라 캡처링부터 시작하여 버블링으로 종료한다는 것이다.** 즉, 이벤트가 발생했을 때 캡처링과 버블링은 순차적으로 발생한다.

캡처링은 IE8 이하에서 지원되지 않는다.



www.w3.org/TR/DOM-Level-3-Events

addEventListener 메소드의 세번째 매개변수에 true를 설정하면 캡처링으로 전파되는 이벤트를 캐치하고 false 또는 미설정하면 버블링으로 전파되는 이벤트를 캐치한다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html { border:1px solid red; padding:30px; text-align: center; }
    body { border:1px solid green; padding:30px; }
    .top {
      width: 300px; height: 300px;
      background-color: red;
      margin: auto;
    }
    .middle {
      width: 200px; height: 200px;
      background-color: blue;
      position: relative; top: 34px; left: 50px;
    }
    .bottom {
      width: 100px; height: 100px;
      background-color: yellow;
      position: relative; top: 34px; left: 50px;
      line-height: 100px;
    }
  </style>
</head>
<body>
  body
  <div class="top">top
    <div class="middle">middle
      <div class="bottom">bottom</div>
    </div>
  </div>
  <script>
  // true: capturing / false: bubbling
  const useCature = true;

  const handler = function (e) {
    const phases = ['capturing', 'target', 'bubbling'];
    const node = this.nodeName + (this.className ? '.' + this.className : '');
    // eventPhase: 이벤트 흐름 상에서 어느 phase에 있는지를 반환한다.
    // 0 : 이벤트 없음 / 1 : 캡처링 단계 / 2 : 타깃 / 3 : 버블링 단계
    console.log(node, phases[e.eventPhase - 1]);
    alert(node + ' : ' + phases[e.eventPhase - 1]);
  };

  document.querySelector('html').addEventListener('click', handler, useCature);
  document.querySelector('body').addEventListener('click', handler, useCature);

  document.querySelector('div.top').addEventListener('click', handler, useCature);
  document.querySelector('div.middle').addEventListener('click', handler, useCature);
  document.querySelector('div.bottom').addEventListener('click', handler, useCature);
  </script>
</body>
</html>
```

좀 더 자세히 살펴보자. 먼저 버블링의 경우 어떻게 동작하는지 알아본다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
  </style>
<body>
  <p>버블링 이벤트 <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');

    // 버블링
    body.addEventListener('click', function () {
      console.log('Handler for body.');
    });

    // 버블링
    para.addEventListener('click', function () {
      console.log('Handler for paragraph.');
```

```
    });

    // 버블링
    button.addEventListener('click', function () {
      console.log('Handler for button.');
    });
  </script>
</body>
</html>
```

위 코드는 모든 이벤트 핸들러가 이벤트 흐름을 버블링만 캐치한다. 즉, 캡쳐링 이벤트 흐름에 대해서는 동작하지 않는다. 따라서 button에서 이벤트가 발생하면 모든 이벤트 핸들러는 버블링에 대해 동작하여 아래와 같이 로그된다.

```
Handler for button.
Handler for paragraph.
Handler for body.
```

만약 p 요소에서 이벤트가 발생한다면 p 요소와 body 요소의 이벤트 핸들러는 버블링에 대해 동작하여 아래와 같이 로그된다.

```
Handler for paragraph.
Handler for body.
```

캡처링의 경우 어떻게 동작하는지 살펴보자.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
  </style>
<body>
  <p>캡처링 이벤트 <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');

    // 캡처링
    body.addEventListener('click', function () {
      console.log('Handler for body.');
    }, true);

    // 캡처링
    para.addEventListener('click', function () {
      console.log('Handler for paragraph.');
    }, true);

    // 캡처링
    button.addEventListener('click', function () {
      console.log('Handler for button.');
    }, true);
  </script>
</body>
</html>
```

위 코드는 모든 이벤트 핸들러가 이벤트 흐름을 캡처링만 캐치한다. 즉, 버블링 이벤트 흐름에 대해서는 동작하지 않는다. 따라서 button에서 이벤트가 발생하면 모든 이벤트 핸들러는 캡처링에 대해 동작하여 아래와 같이 로그된다.

```
Handler for body.
Handler for paragraph.
Handler for button.
```

만약 p 요소에서 이벤트가 발생한다면 p 요소와 body 요소의 이벤트 핸들러는 캡처링에 대해 동작하여 아래와 같이 로그된다.

```
Handler for body.
Handler for paragraph.
```

다음은 캡처링과 버블링이 혼용되는 경우이다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
  </style>
<body>
  <p>버블링과 캡처링 이벤트 <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');

    // 버블링
    body.addEventListener('click', function () {
      console.log('Handler for body.');
    });

    // 캡처링
    para.addEventListener('click', function () {
      console.log('Handler for paragraph.');
    }, true);

    // 버블링
    button.addEventListener('click', function () {
      console.log('Handler for button.');
    });
  </script>
</body>
</html>
```

위 코드의 경우, body, button 요소는 버블링 이벤트 흐름만을 캐치하고 p 요소는 캡처링 이벤트 흐름만을 캐치한다. 따라서 button에서 이벤트가 발생하면 먼저 캡처링이 발생하므로 p 요소의 이벤트 핸들러가 동작하고, 그후 버블링이 발생하여 button, body 요소의 이벤트 핸들러가 동작한다.

```
Handler for paragraph.
Handler for button.
Handler for body.
```

만약 p 요소에서 이벤트가 발생한다면 캡처링에 대해 p 요소의 이벤트 핸들러가 동작하고 버블링에 대해 body 요소의 이벤트 핸들러가 동작한다.

```
Handler for paragraph.
Handler for body.
```

Event 객체

event 객체는 이벤트를 발생시킨 요소와 발생한 이벤트에 대한 유용한 정보를 제공한다. 이벤트가 발생하면 event 객체는 동적으로 생성되며 이벤트를 처리할 수 있는 이벤트 핸들러에 인자로 전달된다.

```
<!DOCTYPE html>
<html>
<body>
  <p>클릭하세요. 클릭한 곳의 좌표가 표시됩니다.</p>
  <em class="message"></em>
  <script>
  function showCoords(e) { // e: event object
    const msg = document.querySelector('.message');
    msg.innerHTML =
      'clientX value: ' + e.clientX + '<br>' +
      'clientY value: ' + e.clientY;
  }
  addEventListener('click', showCoords);
  </script>
</body>
</html>
```

위와 같이 event 객체는 이벤트 핸들러에 암묵적으로 전달된다. 그러나 이벤트 핸들러를 선언할 때, event 객체를 전달받을 첫번째 매개변수를 명시적으로 선언하여야 한다. 예제에서 e라는 이름으로 매개변수를 지정하였으나 다른 매개변수 이름을 사용하여도 상관없다.

```
<!DOCTYPE html>
<html>
<body>
  <em class="message"></em>
  <script>
  function showCoords(e, msg) {
    msg.innerHTML =
      'clientX value: ' + e.clientX + '<br>' +
      'clientY value: ' + e.clientY;
  }

  const msg = document.querySelector('.message');

  addEventListener('click', function (e) {
    showCoords(e, msg);
  });
  </script>
</body>
</html>
```

Event Property

Event.target

```
<!DOCTYPE html>
<html>
<body>
  <div class="container">
    <button id="btn1">Hide me 1</button>
    <button id="btn2">Hide me 2</button>
  </div>

  <script>
    function hide(e) {
      e.target.style.visibility = 'hidden';
      // 동일하게 동작한다.
      // this.style.visibility = 'hidden';
    }

    document.getElementById('btn1').addEventListener('click', hide);
    document.getElementById('btn2').addEventListener('click', hide);
  </script>
</body>
</html>
```

hide 함수를 특정 노드에 한정하여 사용하지 않고 범용적으로 사용하기 위해 event 객체의 target 프로퍼티를 사용하였다. 위 예제의 경우, hide 함수 내부의 e.target은 언제나 이벤트가 바인딩된 요소를 가리키는 this와 일치한다. 하지만 버튼별로 이벤트를 바인딩하고 있기 때문에 버튼이 많은 경우 위 방법은 바람직하지 않아 보인다.

이벤트 위임을 사용하여 위 예제를 수정

```
<!DOCTYPE html>
<html>
<body>
  <div class="container">
    <button id="btn1">Hide me 1</button>
    <button id="btn2">Hide me 2</button>
  </div>

  <script>
    const container = document.querySelector('.container');

    function hide(e) {
      // e.target은 실제로 이벤트를 발생시킨 DOM 요소를 가리킨다.
      e.target.style.visibility = 'hidden';
      // this는 이벤트에 바인딩된 DOM 요소(.container)를 가리킨다. 따라서 .container 요소를 감춘다.
      // this.style.visibility = 'hidden';
    }

    container.addEventListener('click', hide);
  </script>
</body>
</html>
```

위 예제의 경우, this는 이벤트에 바인딩된 DOM 요소(.container)를 가리킨다. 따라서 container 요소를 감춘다. e.target은 실제로 이벤트를 발생시킨 DOM 요소(button 요소 또는 .container 요소)를 가리킨다. Event.target은 this와 반드시 일치하지는 않는다.

Event.currentTarget

이벤트에 바인딩된 DOM 요소를 가리킨다. 즉, addEventListener 앞에 기술된 객체를 가리킨다.

addEventListener 메소드에서 지정한 이벤트 핸들러 내부의 this는 이벤트에 바인딩된 DOM 요소를 가리키며 이것은 이벤트 객체의 currentTarget 프로퍼티와 같다. 따라서 이벤트 핸들러 함수 내에서 currentTarget과 this는 언제나 일치한다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
    div { height: 100%; }
  </style>
</head>
<body>
  <div>
    <button>배경색 변경</button>
  </div>
  <script>
    function bluify(e) {
      // this: 이벤트에 바인딩된 DOM 요소(div 요소)
      console.log('this: ', this);
      // target: 실제로 이벤트를 발생시킨 요소(button 요소 또는 div 요소)
      console.log('e.target:', e.target);
      // currentTarget: 이벤트에 바인딩된 DOM 요소(div 요소)
      console.log('e.currentTarget: ', e.currentTarget);

      // 언제나 true
      console.log(this === e.currentTarget);
      // currentTarget과 target이 같은 객체일 때 true
      console.log(this === e.target);

      // click 이벤트가 발생하면 이벤트를 발생시킨 요소(target)과는 상관없이 this(이벤트에 바인딩된 div 요소)의 배경색이 변경된다.
      this.style.backgroundColor = '#A5D9F3';
    }

    // div 요소에 이벤트 핸들러가 바인딩되어 있다.
    // 자식 요소인 button이 발생시킨 이벤트가 버블링되어 div 요소에도 전파된다.
    // 따라서 div 요소에 이벤트 핸들러가 바인딩되어 있으면 자식 요소인 button이 발생시킨 이벤트를 div 요소에서도 핸들링할 수 있다.
    document.querySelector('div').addEventListener('click', bluify);
  </script>
</body>
</html>
```

Event.type

발생한 이벤트의 종류를 나타내는 문자열을 반환한다.

```
<!DOCTYPE html>
<html>
<body>
  <p>키를 입력하세요</p>
  <em class="message"></em>
  <script>
  const body = document.querySelector('body');

  function getEventType(e) {
    console.log(e);
    document.querySelector('.message').innerHTML = `${e.type} : ${e.keyCode}`;
  }

  body.addEventListener('keydown', getEventType);
  body.addEventListener('keyup', getEventType);
  </script>
</body>
</html>
```

Event.cancelable

요소의 기본 동작을 취소시킬 수 있는지 여부(true/false)를 나타낸다.

```
<!DOCTYPE html>
<html>
<body>
  <a href="poiemaweb.com">Go to poiemaweb.com</a>
  <script>
  const elem = document.querySelector('a');
```

```
  elem.addEventListener('click', function (e) {
    console.log(e.cancelable);

    // 기본 동작을 중단시킨다.
    e.preventDefault();
  });
  </script>
</body>
</html>
```

Event.eventPhase

이벤트 흐름상에서 어느 단계에 있는지를 반환한다.

반환값	의미
0	이벤트 없음
1	캡쳐링 단계
2	타깃
3	버블링 단계

Event Delegation(이벤트 위임)

```
<ul id="post-list">
  <li id="post-1">Item 1</li>
  <li id="post-2">Item 2</li>
  <li id="post-3">Item 3</li>
  <li id="post-4">Item 4</li>
  <li id="post-5">Item 5</li>
  <li id="post-6">Item 6</li>
</ul>
```

모든 li 요소가 클릭 이벤트에 반응하는 처리를 구현하고 싶은 경우, li 요소에 이벤트 핸들러를 바인딩하면 총 6개의 이벤트 핸들러를 바인딩하여야 한다.

```
function printId() {
  console.log(this.id);
}

document.querySelector('#post-1').addEventListener('click', printId);
document.querySelector('#post-2').addEventListener('click', printId);
document.querySelector('#post-3').addEventListener('click', printId);
document.querySelector('#post-4').addEventListener('click', printId);
document.querySelector('#post-5').addEventListener('click', printId);
document.querySelector('#post-6').addEventListener('click', printId);
```

만일 li 요소가 100개라면 100개의 이벤트 핸들러를 바인딩하여야 한다. 이는 실행 속도 저하의 원인이 될 뿐 아니라 코드 또한 매우 길어지며 작성 또한 불편하다.

그리고 **동적으로 li 요소가 추가되는 경우, 아직 추가되지 않은 요소는 DOM에 존재하지 않으므로 이벤트 핸들러를 바인딩할 수 없다.** 이러한 경우 이벤트 위임을 사용한다.

이벤트 위임(Event Delegation)은 다수의 자식 요소에 각각 이벤트 핸들러를 바인딩하는 대신 하나의 부모 요소에 이벤트 핸들러를 바인딩하는 방법이다. 위의 경우 6개의 자식 요소에 각각 이벤트 핸들러를 바인딩하는 것 대신 부모 요소(ul#post-list)에 이벤트 핸들러를 바인딩하는 것이다.

또한 DOM 트리에 새로운 li 요소를 추가하더라도 이벤트 처리는 부모 요소인 ul 요소에 위임되었기 때문에 새로운 요소에 이벤트를 핸들러를 다시 바인딩할 필요가 없다.

이는 이벤트가 이벤트 흐름에 의해 이벤트를 발생시킨 요소의 부모 요소에도 영향(버블링)을 미치기 때문에 가능한 것이다.

실제로 이벤트를 발생시킨 요소를 알아내기 위해서는 `Event.target`을 사용한다.

```
<!DOCTYPE html>
<html>
<body>
  <ul class="post-list">
    <li id="post-1">Item 1</li>
    <li id="post-2">Item 2</li>
    <li id="post-3">Item 3</li>
    <li id="post-4">Item 4</li>
```

```
    <li id="post-5">Item 5</li>
    <li id="post-6">Item 6</li>
  </ul>
  <div class="msg">
  <script>
    const msg = document.querySelector('.msg');
    const list = document.querySelector('.post-list')

    list.addEventListener('click', function (e) {
      // 이벤트를 발생시킨 요소
      console.log('[target]: ' + e.target);
      // 이벤트를 발생시킨 요소의 nodeName
      console.log('[target.nodeName]: ' + e.target.nodeName);

      // li 요소 이외의 요소에서 발생한 이벤트는 대응하지 않는다.
      if (e.target && e.target.nodeName === 'LI') {
        msg.innerHTML = 'li#' + e.target.id + ' was clicked!';
      }
    });
  </script>
</body>
</html>
```

기본 동작의 변경

이벤트 객체는 요소의 기본 동작과 요소의 부모 요소들이 이벤트에 대응하는 방법을 변경하기 위한 메소드는 가지고 있다.

Event.preventDefault()

폼을 submit하거나 링크를 클릭하면 다른 페이지로 이동하게 된다. 이와 같이 요소가 가지고 있는 기본 동작을 중단시키기 위한 메소드가 preventDefault()이다.

```
<!DOCTYPE html>
<html>
<body>
  <a href="http://www.google.com">go</a>
  <script>
  document.querySelector('a').addEventListener('click', function (e) {
    console.log(e.target, e.target.nodeName);

    // a 요소의 기본 동작을 중단한다.
    e.preventDefault();
  });
  </script>
</body>
</html>
```

Event.stopPropagation()

어느 한 요소를 이용하여 이벤트를 처리한 후 이벤트가 부모 요소로 이벤트가 전파되는 것을 중단시키기 위한 메소드이다. 부모 요소에 동일한 이벤트에 대한 다른 핸들러가 지정되어 있을 경우 사용된다.

아래 코드를 보면, 부모 요소와 자식 요소에 모두 mousedown 이벤트에 대한 핸들러가 지정되어 있다. 하지만 부모 요소와 자식 요소의 이벤트를 각각 별도로 처리하기 위해 button 요소의 이벤트의 전파(버블링)를 중단시키기 위해서는 stopPropagation 메소드를 사용하여 이벤트 전파를 중단할 필요가 있다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%;}
  </style>
</head>
<body>
  <p>버튼을 클릭하면 이벤트 전파를 중단한다. <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');

    // 버블링
    body.addEventListener('click', function () {
      console.log('Handler for body.');
    });

    // 버블링
    para.addEventListener('click', function () {
```

```
      console.log('Handler for paragraph.');
    });

    // 버블링
    button.addEventListener('click', function (event) {
      console.log('Handler for button.');

      // 이벤트 전파를 중단한다.
      event.stopPropagation();
    });
  </script>
</body>
</html>
```

preventDefault & stopPropagation

기본 동작의 중단과 버블링 또는 캡처링의 중단을 동시에 실시하는 방법은 아래와 같다.

```
return false;
```

단 이 방법은 jQuery를 사용할 때와 아래와 같이 사용할 때만 적용된다.

```
<!DOCTYPE html>
<html>
<body>
  <a href="http://www.google.com" onclick='return handleEvent()'>go</a>
  <script>
  function handleEvent() {
    return false;
  }
  </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
  <div>
    <a href="http://www.google.com">go</a>
  </div>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.3/jquery.min.js"></script>
  <script>

  // within jQuery
  $('a').click(function (e) {
    e.preventDefault(); // OK
  });

  $('a').click(function () {
    return false; // OK --> e.preventDefault() & e.stopPropagation().
  });

  // pure js
  document.querySelector('a').addEventListener('click', function(e) {
    // e.preventDefault(); // OK
    return false;       // NG!!!!!
  });
  </script>
</body>
</html>
```