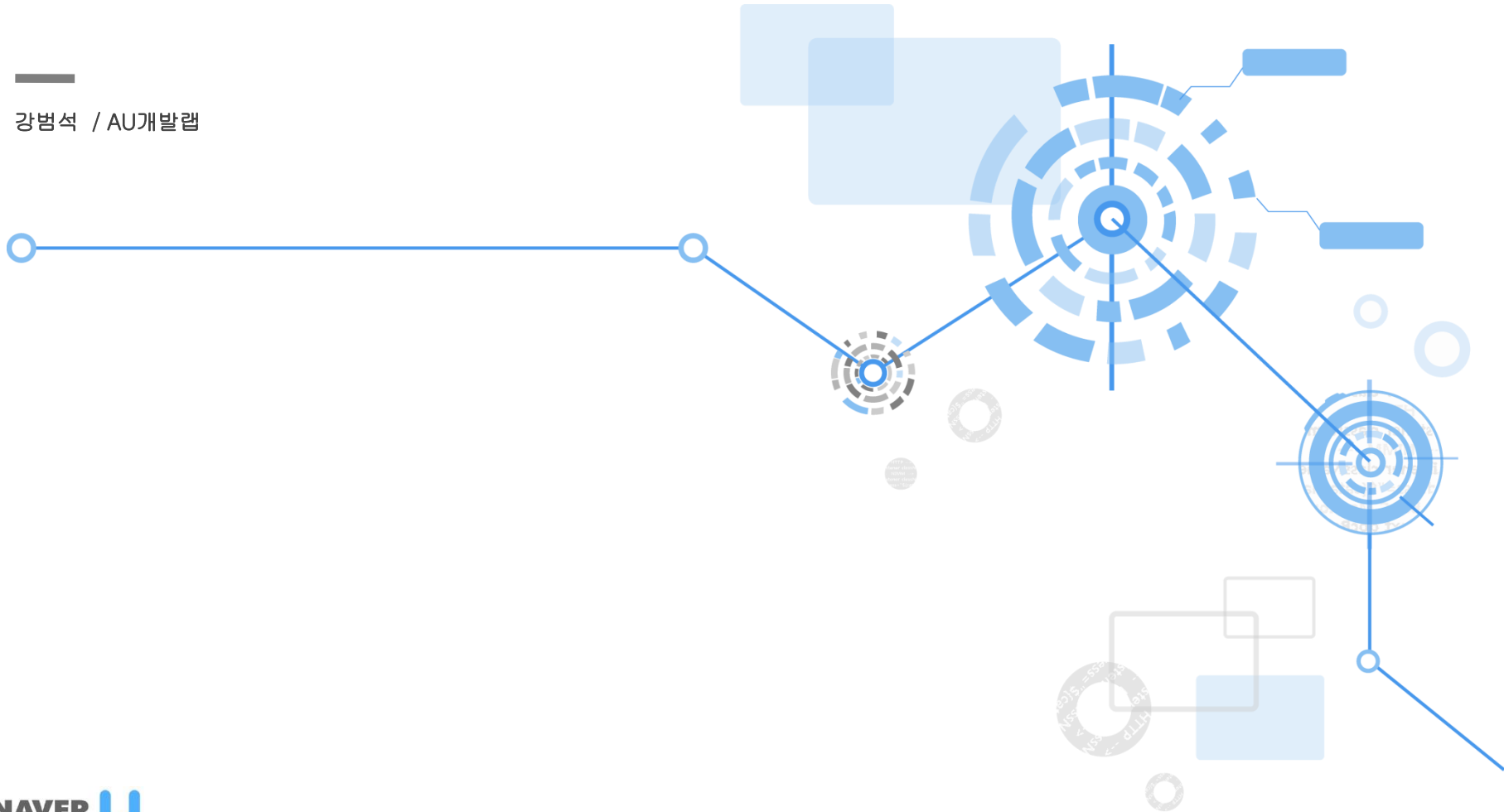


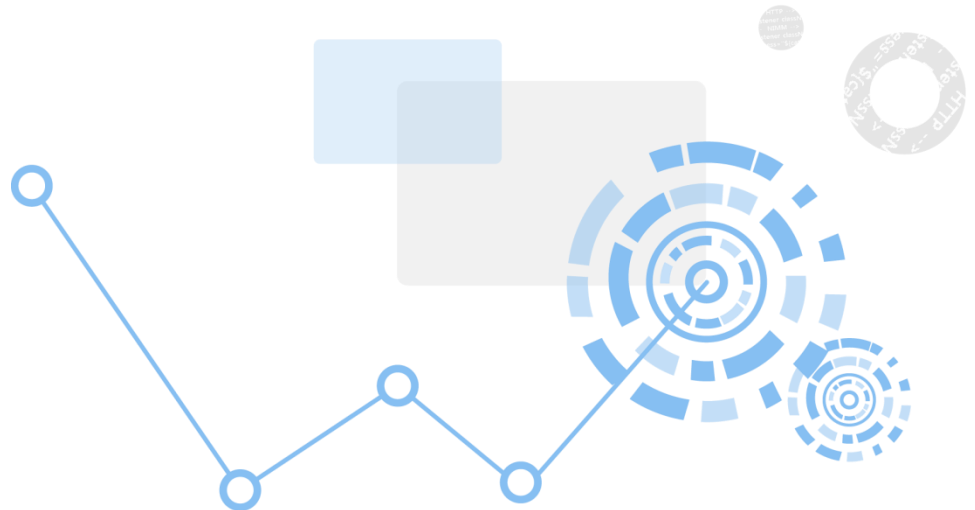
JS 교육

강범석 / AU개발팀

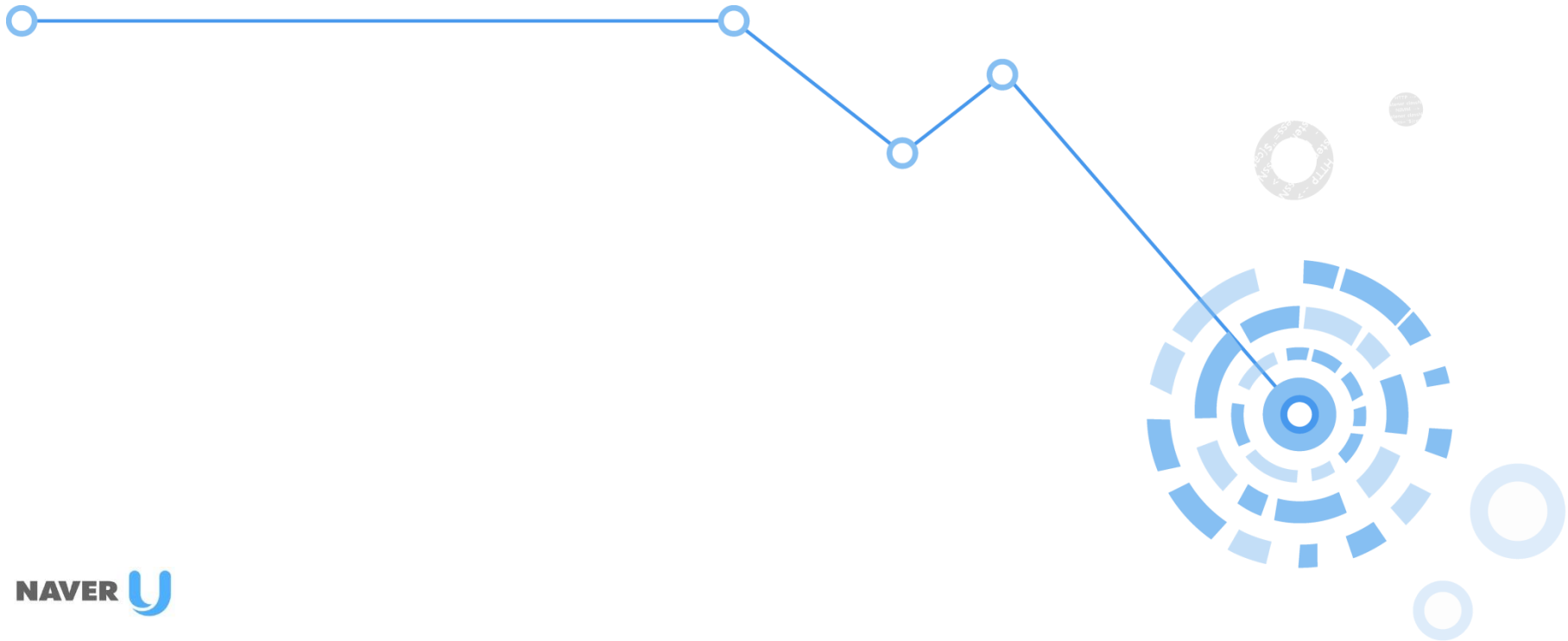


목차

1. 소개
2. 기본타입
3. 연산자
4. 문법
5. 객체
6. 함수
7. 함수 스코프 & 실행 컨텍스트
8. 클로저
9. 이벤트 루프
10. DOM



1. 소개





진입장벽이 낮은 만큼 잘못 사용되어 많은 오해를 받은 언어
하지만, 클로저 같은 고차원 개념까지 다루는 고급언어.

위상

- 전세계에서 오픈 소스 프로젝트가 가장 많은 언어
- 웹브라우저 뿐 아니라, 서버사이드 스크립트 언어로 까지 활용 (ex. Node.js)
- Native App 개발도 가능한 언어 (제한적)
 - ex. 라인툴즈 (JS로 작성되고, 타이타늄 SDK로 아이폰/안드로이드 앱 생성)

요약

JS 없는 웹개발(Front End)은 상상X
알면 알 수록 활용도가 높은 언어
마음만 먹으면, 무엇이든 할 수 있는 언어
(but.. JS로 모든걸 하는 것은 쉽지 않다. 성능이슈 등...



But...





순수 JS 개발의 어려움

- 플랫폼 분기 (PC/모바일)
- OS 버전 분기 (winXP, win7, win10, iOS, Android...)
- S/W (브라우저) 분기
- H/W (모바일 기기) 분기

JS를 숙지하기 어려운 이유?

- 언어는 문법이 전부가 아니다
 - 내장 메소드
 - 실행환경 제공 API (DOM, BOM 등등...)
- 배우기 쉬운 언어 === 하나만 알아도 열을 예측 가능한 일관성을 가진 언어
 - JS 내장 메소드와 browser **DOM API** 는 **일관성 없기로 악명이 높다.**
- JS의 메소드는 아무리 봐도 잘 기억 나지 않는다.
 - 메소드마다 일관성 없는 파라미터 순서
 - 심지어... 환경에 따라 다르게 동작하는 메소드



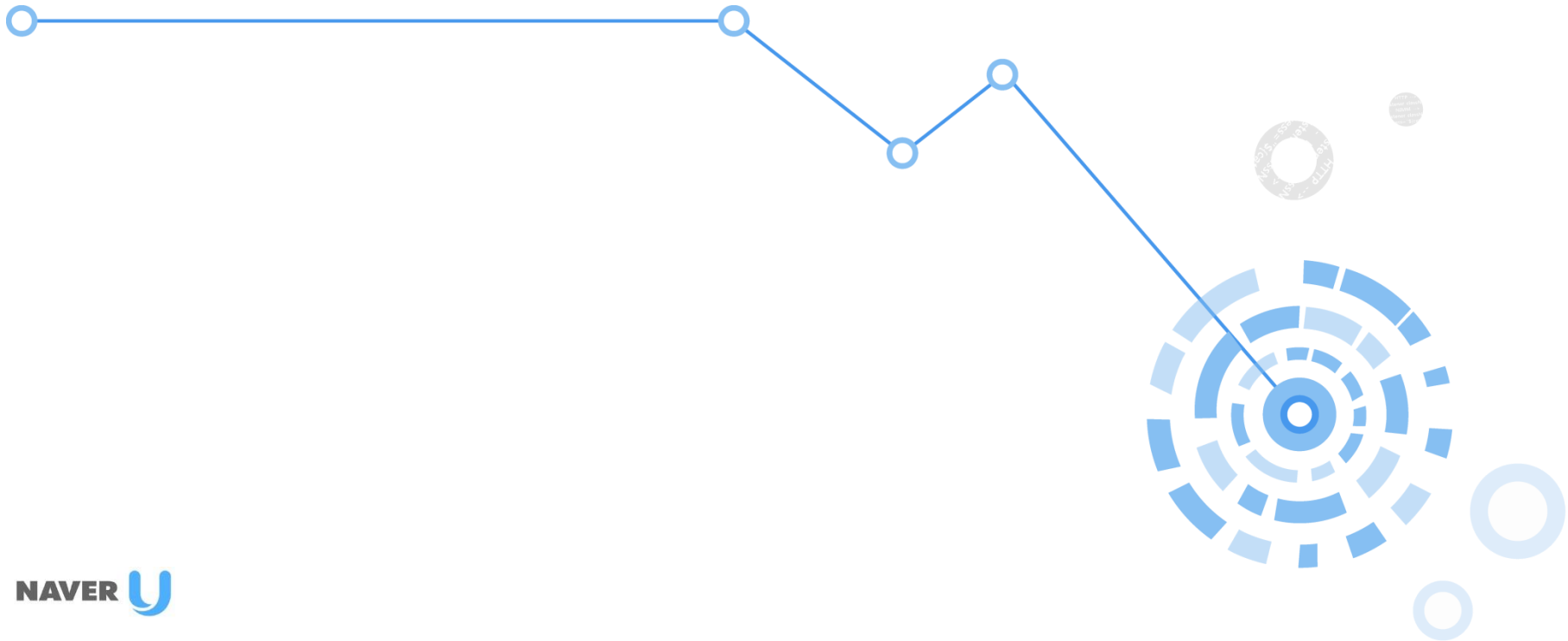
JS 라이브러리

- Jindo, jQuery 등등
- 플랫폼, S/W, H/W 별로 같은 동작을 보장
- 하나를 배워도 열을 배운 것 같은 착각을 불러일으키는 일관성
- 기본 문법만 알아도 문제없는 실무개발 가능

하지만

- 더 강력하고 깊이있게 사용하려면, 그 내부 동작과 원리를 아는 것은 중요하다.

준비...



시작전에...



크롬 브라우저

- Chrome DevTools
 - 메뉴 > 도구 더보기 > 개발자도구

For Test

- <http://codepen.io/pen/>
- <http://jsfiddle.net/>
- <http://jsbin.com/>

코드 검사

- <http://jshint.com/>
- <http://jsbeautifier.org/>

Hello world!



```
> console.log("hello world!");
```

```
hello world!
```

```
< undefined
```



```
var obj = {};  
console.log(obj); // 변수 내용 확인  
console.dir(obj); // 객체 트리구조 확인  
|
```



W3schools

- <http://www.w3schools.com/js/default.asp>

MDN (Built-in objects)

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

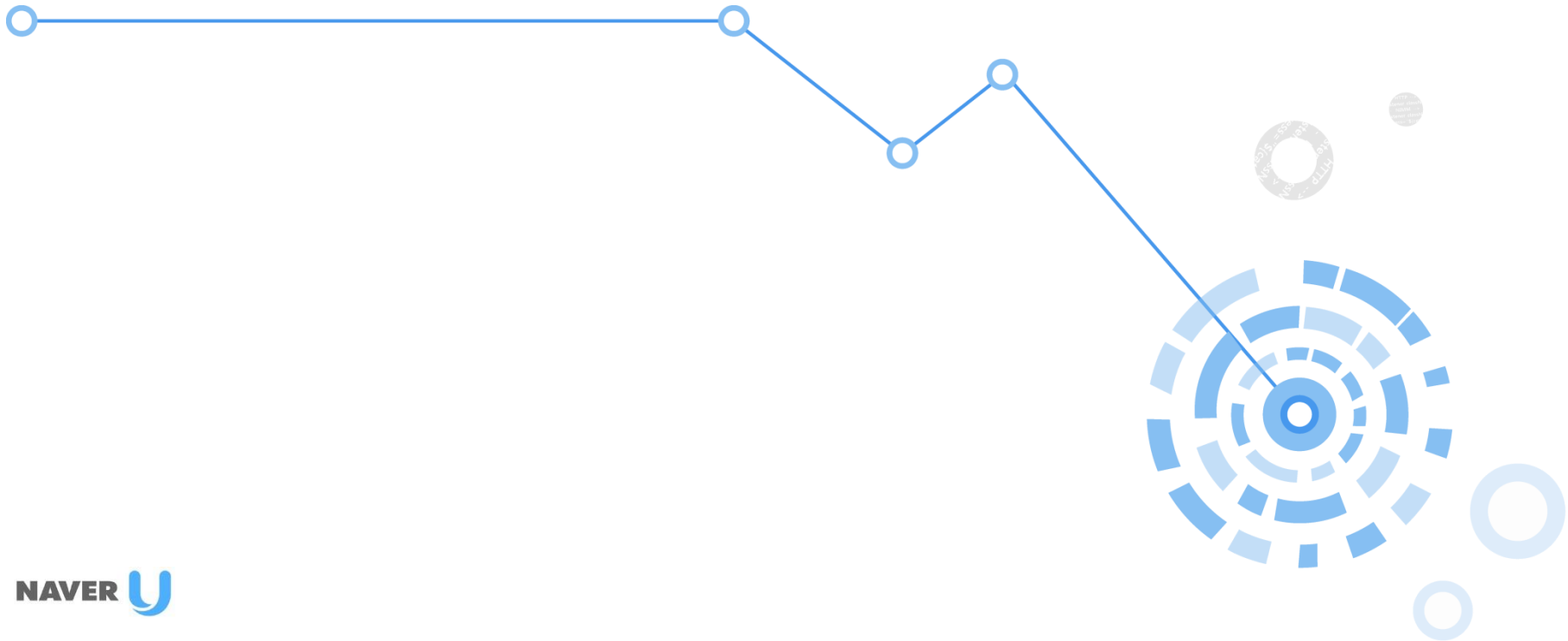
MSDN

- [https://msdn.microsoft.com/ko-kr/library/ce57k8d5\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/ce57k8d5(v=vs.94).aspx)

QnA

- <http://stackoverflow.com/>

2. 기본 타입





기본타입 외에는 모두 object

- number
- string
- boolean
- undefined 선언되지 않았거나 값이 할당되지 않은 상태
- null 아무값도 나타내지 않는 특수한 값



타입을 확인하는 타입 연산자

- Syntax: `typeof <value>`
- Return: 타입 식별자 문자열

```
> typeof 1
```

```
< "number"
```

```
> typeof "1"
```

```
< "string"
```

```
> typeof true
```

```
< "boolean"
```

```
> typeof a
```

```
< "undefined"
```

```
> typeof null
```

```
< "object"
```

```
>
```



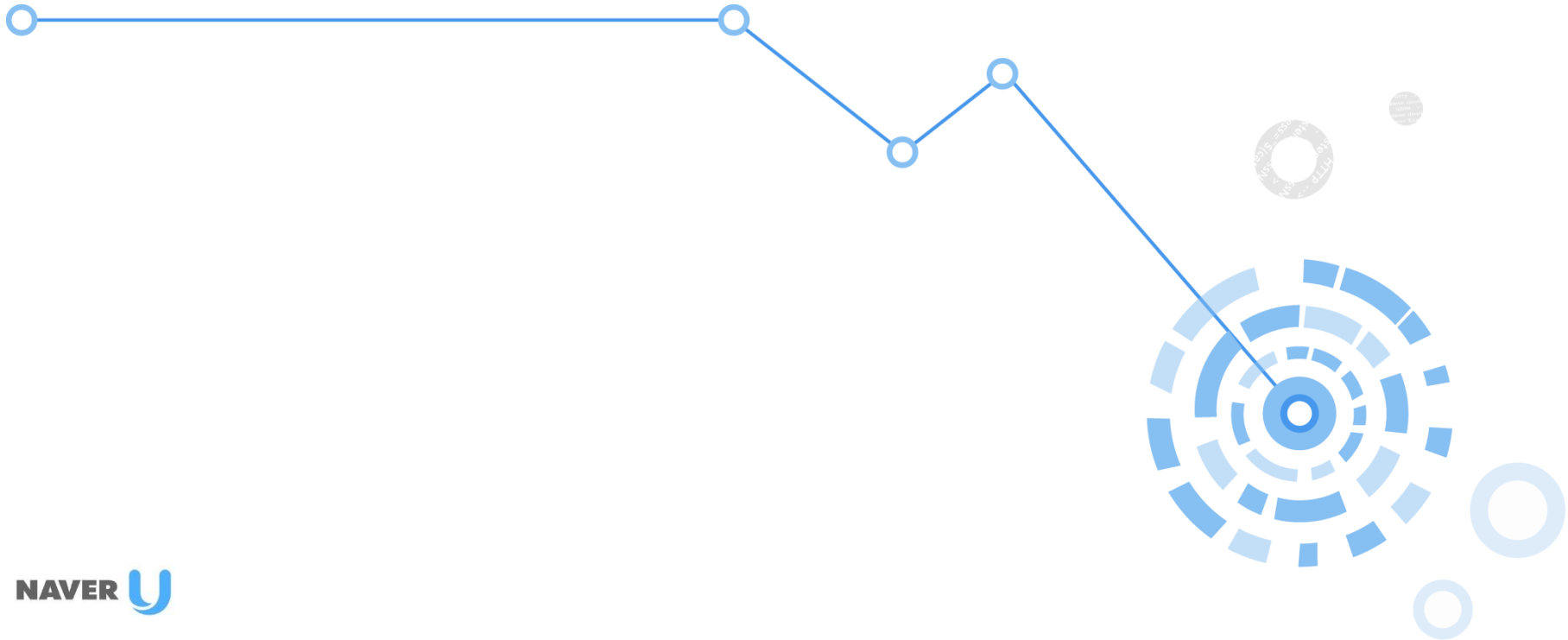
null 과 object 를 구별하지 못하는 JS

```
1  (function () {  
2  
3  
4      var foo = null;  
5      var bar = {};  
6  
7      typeof foo === typeof bar; // ???  
8      foo === bar;                // ???  
9      foo === null;               // ???  
10     bar === null;               // ???  
11  
12  
13 }));|
```

=== : type & value check

== : only value check

3. 연산자





typeof 연산자

```
1  (function () {  
2  
3      typeof {};  
4      typeof [];  
5      typeof null;  
6      typeof true;  
7      typeof 10;  
8      typeof 'hello';  
9      typeof undefined;  
10  
11  })();
```

다른 연산자들...

- 산술연산자: + - * / %
- 비교연산자: ==, != < > <= >=
- 논리연산자: && || !
- 비트연산자: & | ^ >> >>> <<
- 3항 연산자: ? :



+

숫자 끼리의 연산은 합

문자열 끼리의 연산은 대상 문자열을 모두 포함하는 새로운 문자열 반환

```
> 'a' + 1 + 2; // 문자열과 숫자이면??
```

*

Number 끼리의 연산처럼 동작, 곱 반환

```
> "1" * 2; // 숫자문자, number  
"2" * "3"; // 숫자문자, 숫자문자  
"A" * 2; // 알파벳, number
```



NaN

숫자 연산 결과가 숫자가 아닐때 반환
타입식별자로 NaN 과 숫자를 구별할 수 없다.

```
> typeof NaN;  
◀ "number"
```

isNaN()

숫자 여부를 확인할 수 있는 JS 메소드

```
> isNaN("A" * 2);  
◀ true
```



&&

첫번째가 거짓이면, 첫번째 결과를 반환
첫번째가 참이면, 두번째 결과를 반환

```
> false && true;
```

```
< false
```

```
> true && false;
```

```
< false
```

```
> true && true;
```

```
< true
```

```
>
```

quiz

```
> false && 1+2; // ???  
true && 1+2; // ???  
0 && 1+2; // ???
```



&& 특징

첫번째가 참이면 두번째 조건식은 수행하지 않는다.

```
> // obj 가 참일 때만 alert() 실행  
var obj = {};  
obj && alert('run');
```

거짓인 value 들..

false	? true : false;
null	? true : false;
undefined	? true : false;
NaN	? true : false;
"	? true : false;
0	? true : false;



||

첫번째가 참이면, 첫번째 결과를 반환
첫번째가 거짓이면, 두번째 결과를 반환

quiz

```
> false || 1+2; // ???  
true  || 1+2;  // ???  
0     || 1+2;   // ???
```

활용

// 옵션 미지정시 디폴트 처리

```
var defaultVal = window.option || "defaultValue";
```

== === != !==

의도한 것이 아니라면 항상 ===, !== 를 사용한다. (type & value check)

```
> 0 == "0"; // ??  
0 == ""; // ??
```

quiz

```
1  /**  
2   * 변수 obj 가 객체인지 여부를 확인하는 조건문 만들기  
3   * - hint. null 은 거짓이면서 object 타입  
4   */  
5  (function () {  
6  
7      var obj = null;  
8  
9      if ( ??? ) {  
10         console.log('객체 입니다.');11     } else {  
12         console.log('null 이거나 객체가 아닙니다.');13     }  
14  
15 })();
```



Ref.

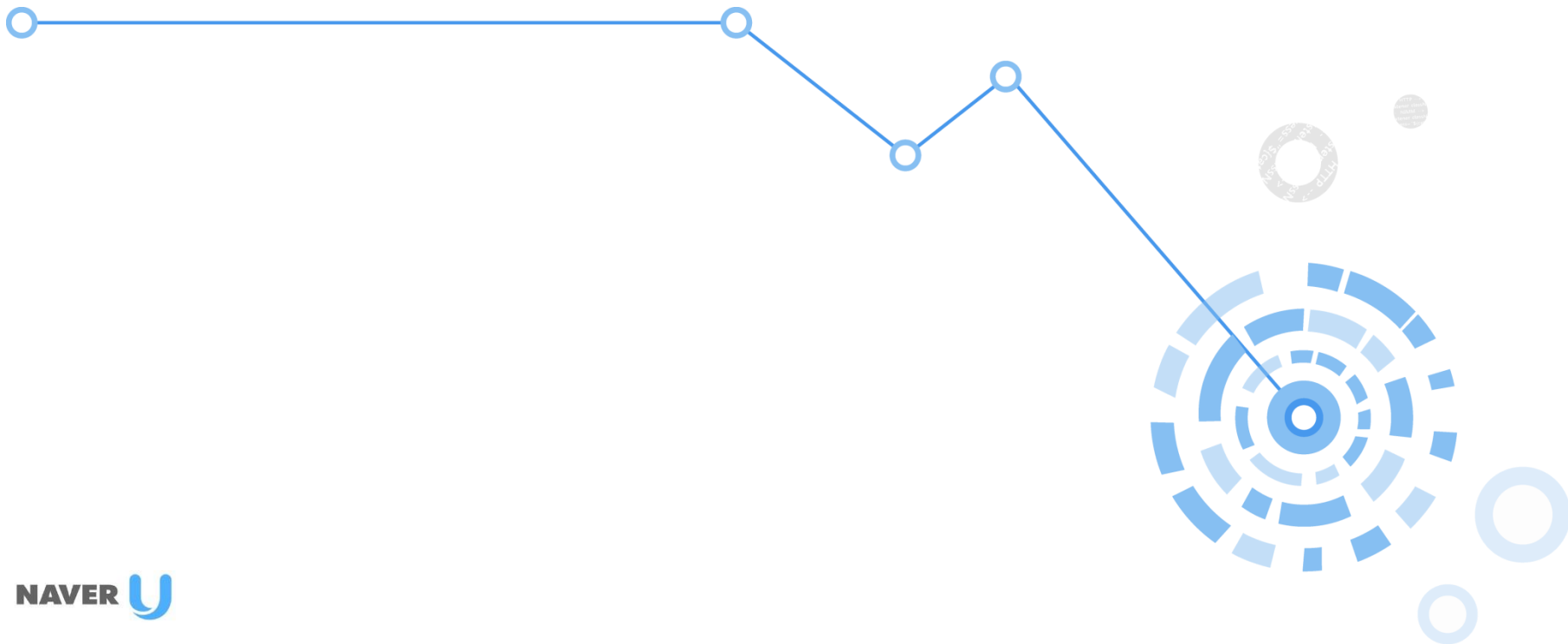
연산자

[https://msdn.microsoft.com/ko-kr/library/ce57k8d5\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/ce57k8d5(v=vs.94).aspx)

연산자 우선순위

[https://msdn.microsoft.com/ko-kr/library/z3ks45k7\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/z3ks45k7(v=vs.94).aspx)

4. 문법





표현식

- var
- if
- switch
- while
- do
- for
- break
- continue
- return



var: 변수명 지정

```
var name;  
var a, b, c;  
var a = 1;
```

변수명등으로 사용할 수 없는 단어들..

문법을 위해 예약된 키워드 & 기타 허용되지 않은 특수문자

```
abstract  
boolean break type  
case catch char class const continue  
debugger default delete do double  
else enum export extends  
false final finally float for function goto  
if implements import in instanceof int interface  
long native new null  
package private protected public return  
short static super switch synchronized  
this throw throws transient true try typeof  
var volatile void while with
```



if

```
> if (true) {  
    console.log("참");  
} else {  
    console.log("거짓");  
}
```

참



switch

```
> switch ('a') {  
    case 'a': console.log('a'); break;  
    case 'b': console.log('b'); break;  
    default:  
        console.log('default');  
}
```

a

```
> switch ('a') {  
    case 'a': console.log('a');  
    case 'b': console.log('b'); break;  
    default:  
        console.log('default');  
}
```



for

```
> for (var i=0; i<5; i++) {  
    console.log(i);  
}
```

0

1

2

3

4



for in: object 프로퍼티 검사

```
> var obj = {  
    "name": "foo",  
    "age": 27  
};  
  
for (var prop in obj) {  
    console.log(prop + " => " + obj[prop]);  
}
```

```
name => foo
```

```
age => 27
```



in 연산자

- syntax: <value> **in**
- desc: 개체에 속성이 존재하는지 여부를 테스트

```
> var obj = {"method" : function () {}};
```

```
"method" in obj;  
obj.hasOwnProperty('method');
```




hasOwnProperty

➤ `// 표준객체 기반의 인스턴스의 내부구조`
`console.dir(obj);`

▼ Object i

▶ `method: function ()`

▼ `__proto__: Object`

▶ `__defineGetter__: function __de`

▶ `__defineSetter__: function __de`

▶ `__lookupGetter__: function __lo`

▶ `__lookupSetter__: function __lo`

▶ `constructor: function Object()`

▶ `hasOwnProperty: function hasOw`



```
> "hasOwnProperty" in obj; // true
```

```
obj.hasOwnProperty( 'hasOwnProperty' ); // false why?
```

in 연산자와 hasOwnProperty 메소드 차이점

in 연산자: 상속받은(프로토타입 체인에 있는) 속성까지 검사

hasOwnProperty: 자기 자신의 속성인지만 검사



```
1  /**
2   * 프로토타입 맛보기
3   */
4   var obj = {"name" : "prototype test"};
5
6   // 임의로 프로토타입 속성 추가
7   // 표준객체 기반의 객체는 Object.prototype 의 속성을 상속 받는다
8   Object.prototype.job = "developer";
9
10  // 모든 속성을 검사하기
11  for (var prop in obj) {
12      console.log(prop + " => " + obj[prop]);
13  }
14
15
16  // 프로토타입으로 부터 상속받은 속성은 제외하고 자기 자신의 속성만 검사하기
17  for (var prop in obj) {
18      if (obj.hasOwnProperty(prop)) {
19          console.log(prop + " => " + obj[prop]);
20      }
21  }
```



name => prototype test

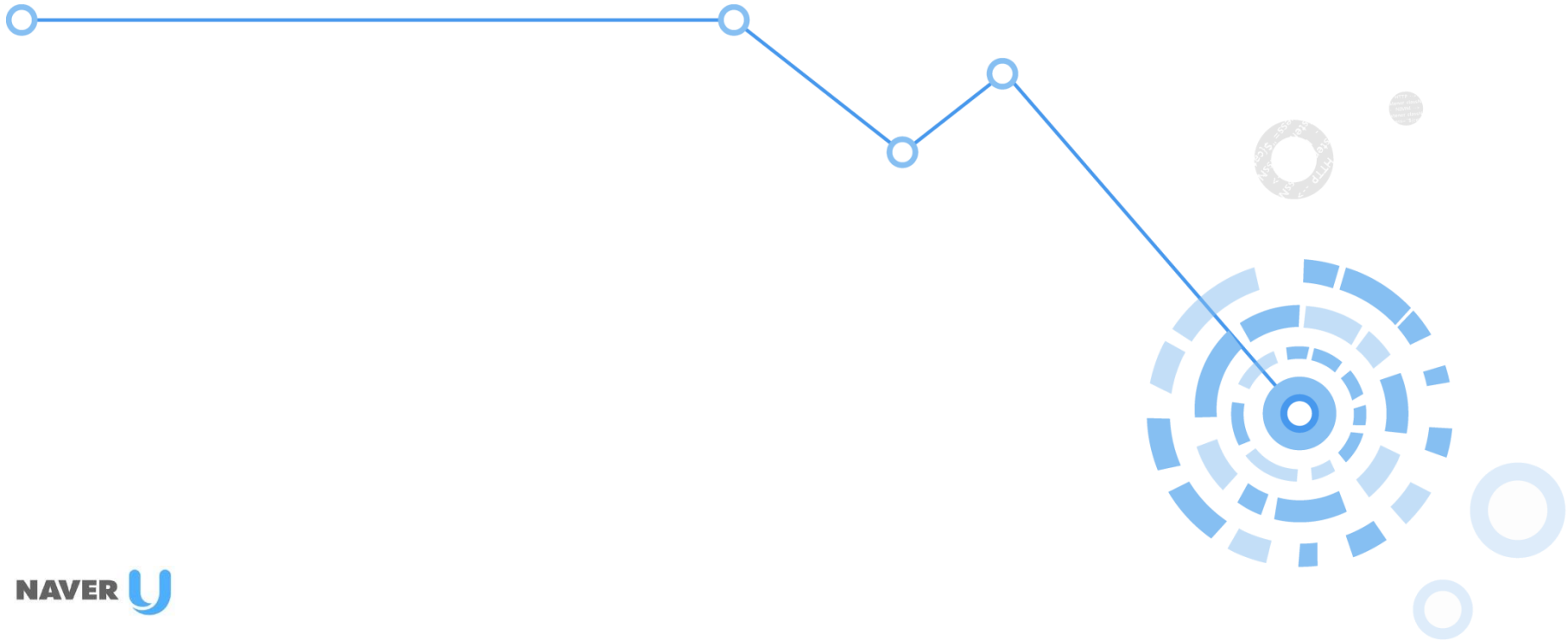
job => developer

name => prototype test

Ref.

http://www.w3schools.com/js/js_statements.asp

5. 객체





JSON

JavaScript Object Notation

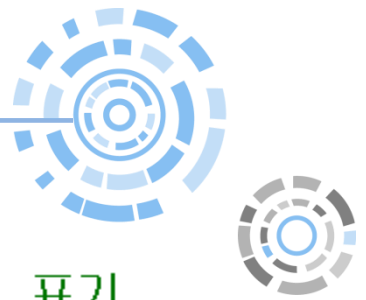
자바스크립트 객체 표기법

→ JS Object: 문자열 표기 가능

* JS객체는 문자열 표기가 가능해서 다른 언어 혹은 플랫폼(서버) 사이에 경량의 데이터를 주고 받을 때 사용되는 데이터 포맷으로도 활용된다.

> `var obj = new Object();` // 자바스크립트 표준 객체

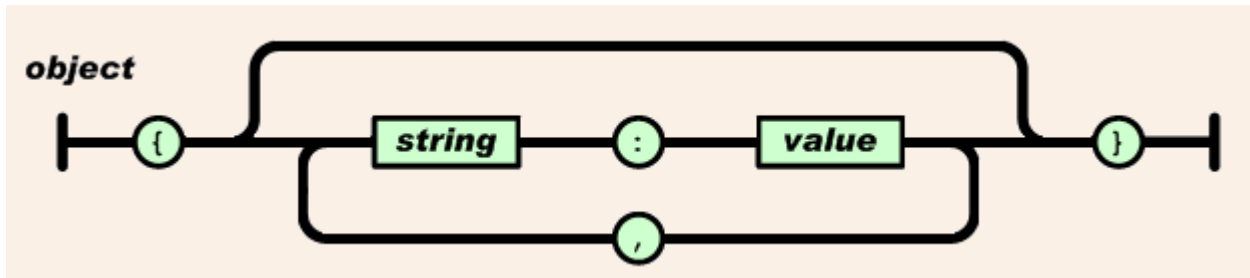
`var obj = {};` // 리터럴 표현방식 (JSON)



> // 일반 객체는 key, value 의 쌍으로 이루어진 중괄호로 표기

```
var obj = {  
  "name": "홍길동",  
  "age": 27  
};
```

<http://www.json.org/json-ko.html>





```
> var obj = {  
    "name": "my obj"  
};
```

```
obj.name;    // dot 접근 표기법
```

```
obj["name"]; // 대괄호 접근 표기법
```

```
// 대괄호 표기법은 인덱스에 변수를 사용할 수 있다.  
var str = "name";  
obj[str];
```




```
> var obj = {};
```

```
> // 프로퍼티 추가  
obj.name = "foo";  
console.log(obj);  
Object {name: "foo"}
```

```
> // 프로퍼티 삭제  
delete obj.name;  
console.log(obj);  
Object {}
```



```
> /**
 * 모든 객체는 자바스크립트 표준객체인
 * Object의 prototype 객체를 상속 받는다.
 * - 숨겨진 __proto__ 속성에 연결된다.
 */
```

```
console.dir({});
```

▼ Object ⓘ

▼ __proto__: Object

- ▶ __defineGetter__: function __defineGetter__()
- ▶ __defineSetter__: function __defineSetter__()
- ▶ __lookupGetter__: function __lookupGetter__()
- ▶ __lookupSetter__: function __lookupSetter__()
- ▶ constructor: function Object()
- ▶ hasOwnProperty: function hasOwnProperty()
- ▶ isPrototypeOf: function isPrototypeOf()
- ▶ propertyIsEnumerable: function propertyIsEnumerable()
- ▶ toLocaleString: function toLocaleString()
- ▶ toString: function toString()
- ▶ valueOf: function valueOf()
- ▶ get __proto__: function get __proto__()
- ▶ set __proto__: function set __proto__()



> console.dir(Object); // 표준객체는 타입이 **Object** 인 함수

▶ *function* Object()

> console.dir(Object.prototype); // 일반객체가 연결되는 **Object**의 **prototype**

▼ Object 

- ▶ *__defineGetter__*: *function* __defineGetter__()
- ▶ *__defineSetter__*: *function* __defineSetter__()
- ▶ *__lookupGetter__*: *function* __lookupGetter__()
- ▶ *__lookupSetter__*: *function* __lookupSetter__()
- ▶ *constructor*: *function* Object()
- ▶ *hasOwnProperty*: *function* hasOwnProperty()
- ▶ *isPrototypeOf*: *function* isPrototypeOf()
- ▶ *propertyIsEnumerable*: *function* propertyIsEnumerable()
- ▶ *toLocaleString*: *function* toLocaleString()
- ▶ *toString*: *function* toString()
- ▶ *valueOf*: *function* valueOf()
- ▶ *get* *__proto__*: *function* get __proto__()
- ▶ *set* *__proto__*: *function* set __proto__()



```
> var obj1 = {},  
    obj2 = {},  
    obj3 = {};
```

```
// 인스턴스의 속성을 동적으로 추가하면 해당 인스턴스에만 영향을 미치지만,  
obj1.name = 'foo'  
obj1.name; // foo  
obj2.name; // undefined  
obj3.name; // undefined
```

```
// 표준객체의 프로토타입 속성은 모든 인스턴스에 영향을 미친다.  
Object.prototype.name = "bar";  
obj1.name; // foo 자신에게 없는 속성만 prototype 에서 찾는다.  
obj2.name; // bar  
obj3.name; // bar
```



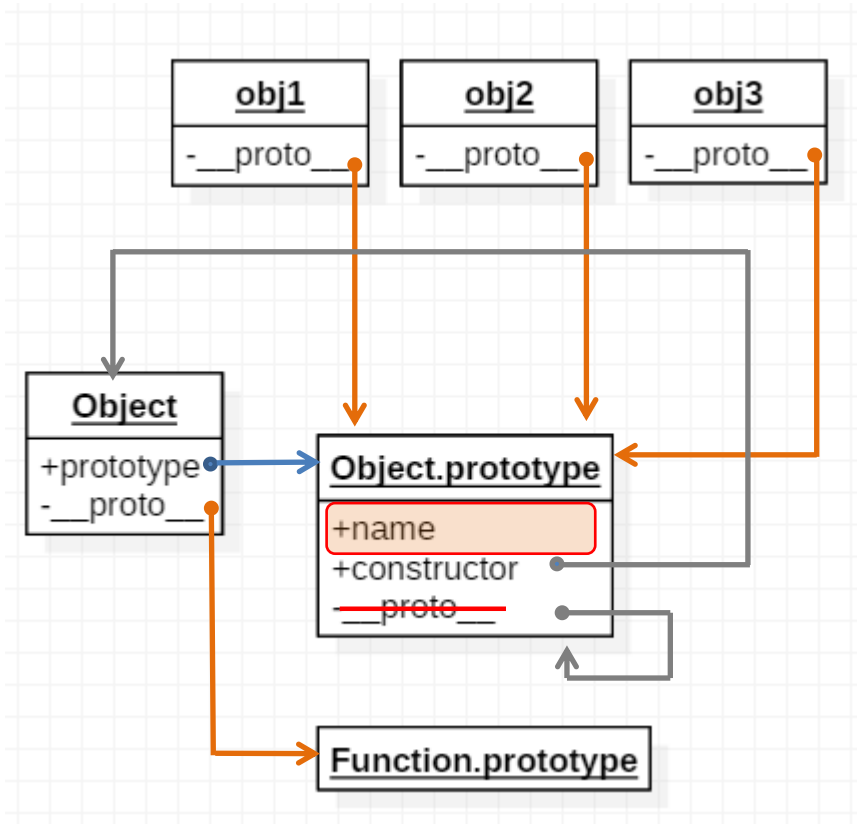
객체 속성 접근 순서

1. 자신의 속성에서 찾는다.
2. 자신에게 **없는 속성**은 **prototype 객체**에서 찾는다.
3. 최상위 연결이 Object.prototype (표준객체) 일때까지 반복..

ps.

- prototype 속성과 연결된 객체가 변경되면, 생성된 모든 인스턴스가 영향 받는다. (by step 2)
- 프로토타입의 특성을 이용하여, 다른 언어의 상속 유사한 기능을 구현할 수 있다.

prototype chain (기초)



```
> var obj1 = {},  
    obj2 = {},  
    obj3 = {};
```

```
> Object.prototype.name = "bar";
```

```
> console.dir(Object);  
  ► function Object()
```

객체의 종류 (class)



```
> /**
 * 인스턴스(객체복사본)의 타입 -> 프로토타입에 연결된 객체
 */
```

```
// 일반 객체: Object.prototype 에 연결..
console.dir({});
```

```
▼ Object ⓘ
  ► __proto__: Object
```

```
> // 함수 객체: Function.prototype 에 연결..
console.dir(function() {});
```

```
▼ function anonymous()
  arguments: null
  caller: null
  length: 0
  name: ""
  ► prototype: Object
  ► __proto__: function ()
  ► <function scope>
```

Object → 함수 ? Type ?



```
> // 타입이 Object 인 객체 (class Object)  
console.dir(Object);
```

```
▶ function Object()
```

```
> // 타입이 Function 인 객체 (class Function)  
console.dir(Function);
```

```
▶ function Function()
```

```
> // 기본타입도 객체타입(class) 함수로 제공되어 객체처럼 쓸 수 있다.  
console.dir(String); // class String
```

```
▶ function String()
```

```
> console.dir(Number); // class Number
```

```
▶ function Number()
```




```
> /**
 * 인스턴스 오브젝트 생성
 * - 타 언어의 class 의 인스턴스 생성과 유사
 */
var obj = new Object();    // Object 타입(class)인 인스턴스 obj
var arr = new Array();     // Array 타입(class)인 인스턴스 arr
var func = new Function(); // Function 타입(class)인 인스턴스 func

var obj = {};              // new Object()   표준객체 복제(인스턴스)
var arr = [];              // new Array()    배열객체 복제(인스턴스)
var func = function () {}; // new Function() 함수객체 복제(인스턴스)

> /**
 * typeof 연산자
 * - 기본타입 object 여부 식별 가능 but 정확한 타입까지는 알 수 없다.
 */
typeof obj;    // object
typeof arr;    // object (Array Object 모두 object 를 반환한다.)
typeof func;   // function
```

인스턴스 타입 확인 : 새로 만든 타입의 인스턴스 구별



```
> /**
 * instanceof 연산자
 * 사용법: <변수> instanceof <타입>
 */
```

```
var MyFunc = function () {}; // MyFunc 사용자 정의 타입 선언
var oMyFunc = new MyFunc(); // 사용자 정의 타입의 인스턴스(복제본) 생성
```

```
oMyFunc instanceof MyFunc;
```

```
< true
```

```
> // constructor 속성값을 이용하여 타입을 알 수 있다.
oMyFunc.constructor === MyFunc;
```

```
< true
```



```
> // oMyFunc 의 프로토타입은 MyFunc  
console.dir(oMyFunc);
```

▼ MyFunc i

▶ `__proto__`: MyFunc

```
> // MyFunc 프로토타입의 객체는 자신과 연결  
console.dir(MyFunc);
```

▼ *function* MyFunc()

arguments: null

caller: null

length: 0

name: ""

▼ `prototype`: MyFunc

▶ `constructor`: *function* ()

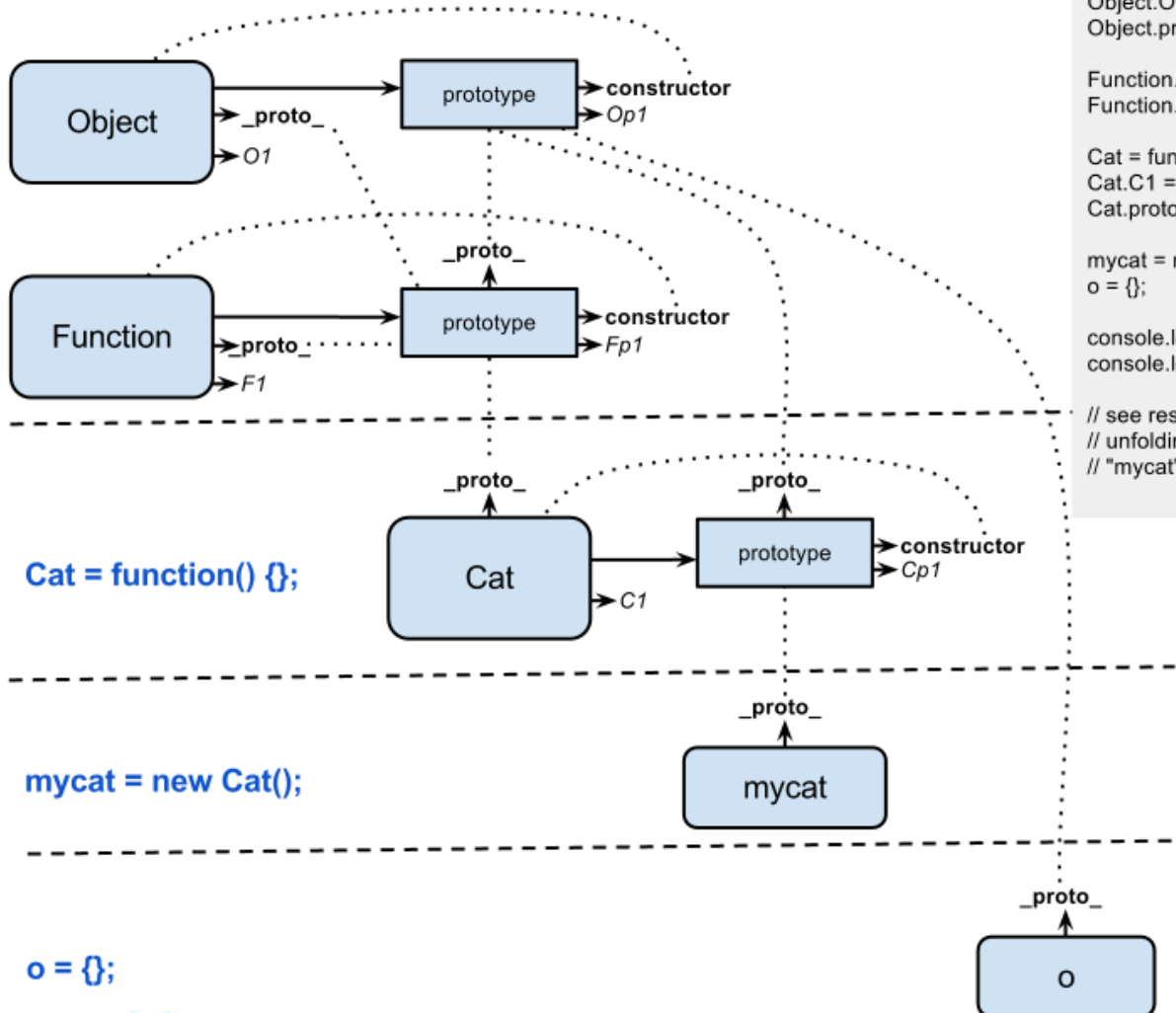
▶ `__proto__`: Object

▶ `__proto__`: *function* ()

▶ <function scope>

Javascript objects treasure map

david@utsaina.com - v0.2 - 2012/06/28



Code for testing the map

```
Object.O1="";
Object.prototype.Op1="";

Function.F1="";
Function.prototype.Fp1="";

Cat = function(){};
Cat.C1="";
Cat.prototype.Cp1="";

mycat = new Cat();
o = {};

console.log(mycat);
console.log(o);

// see result i.e. in Chrome console,
// unfolding the links of the
// "mycat" and "o" objects
```



자동차 신모델

프로토타입 제작

대량 생산

생산 자동화 공정(틀)

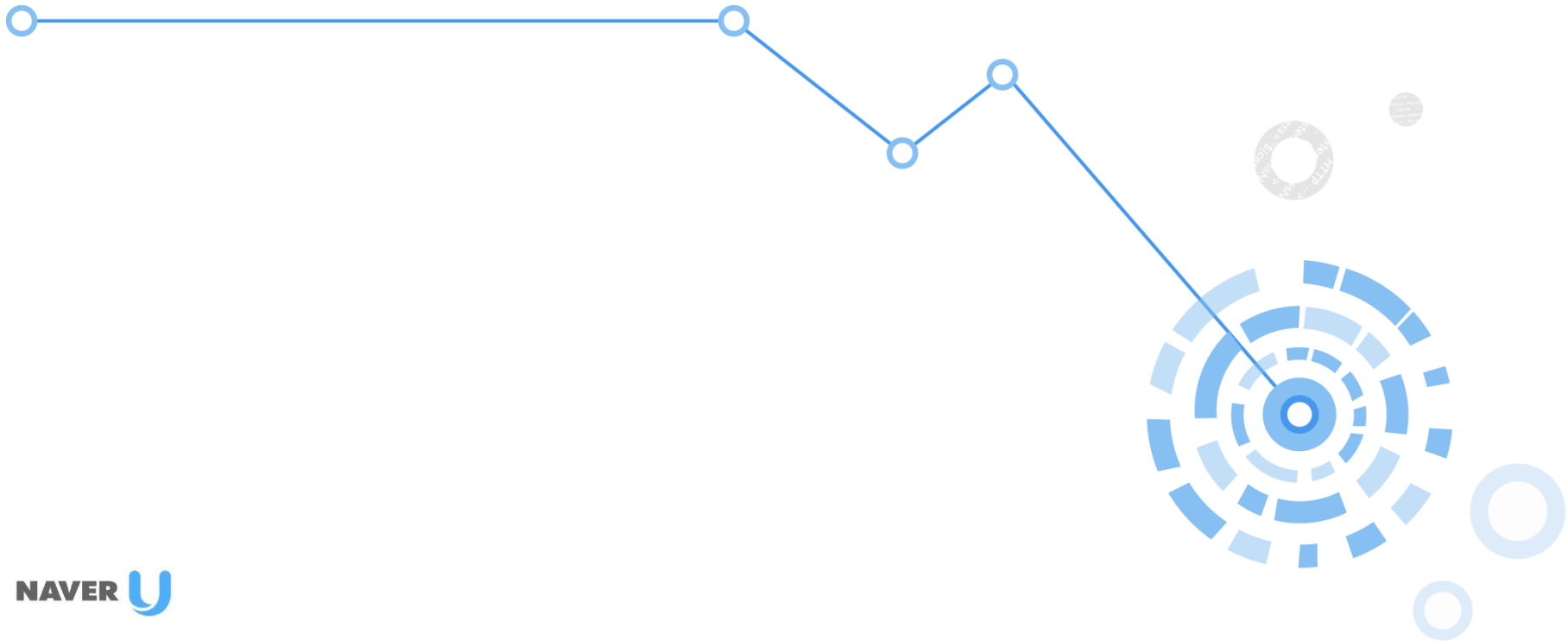
```
> // FType 모델 자동화  
function FType() {}
```

```
var oCar1 = new FType(); // 재규어 FType 1호  
var oCar2 = new FType(); // 재규어 FType 2호
```

제품

프로토타입
기반

6. 함수





$$y = f(x)$$

```
> // 함수 선언
function Func(input) {
    var output = input+'의 출력';
    return output;
}
< undefined
> Func(1);
< "1의 출력"
```



- > // 리터럴 표기법으로 변수에 저장할 수 있다.

```
var Func = function (a, b) {  
    return a+b;  
};
```

// 또는, 다음과 같이 생성할 수도 있다.

```
var Func = new Function("a,b", "return a+b");
```

- > // 파라미터로 메소드도 전달할 수 있다.
// 콜백 함수 파라미터로 전달되어 수행되는 함수

```
function Test(fBefore, fAfter) {  
    fBefore(); // 전처리 콜백  
    console.log("hello world!");  
    fAfter(); // 후처리 콜백  
}
```

```
Test(function() {console.log(1);}, function() {console.log(2)});
```

1

hello world!

2



함수 선언 :

→ 코드 평가 전에 미리 함수를 생성



함수 표현식 :

→ 코드를 실행하는 시점에 함수를 생성

→ 함수명을 지정하지 않으므로 직접 함수를 변수에 할당하여 사용



생성자 함수 :

→ new 키워드로 객체를 생성할 수 있는 함수

→ 자신과 동일한 복제본을 생성할때 사용 (인스턴스 생성)



```
> console.log(typeof fn1); // undefined  
  
var fn1 = function () {};  
  
console.log(typeof fn1); // function
```

```
> console.log(typeof fn2); // function  
  
function fn2() {  
}  
  
console.log(typeof fn2); // function
```



```
> bar(); // bar  
foo(); // undefined
```

```
// 선언만 호이스팅 된다. (생성은 실행시점)
```

```
var bar = function() { console.log('foo');};
```

```
// 선언과 생성이 모두 된다. (실행전 생성)
```

```
function bar() {  
  console.log('bar');  
}
```

```
bar
```

```
✖ ▶ Uncaught ReferenceError: foo is not defined(...)
```

```
> |
```



var bar;

> bar(); // bar

// 실행 시점에 함수 생성

// var bar; 구분은 호이스팅 된다.

var bar = function() { console.log('foo');};

// 코드 실행전에 미리 함수 생성

```
function bar() {  
  console.log('bar');  
}
```

bar(); // foo

bar

foo



```
> function Test () {  
    var func = function a () {  
        console.log('inner function');  
    };  
  
    func();  
}
```

```
> Test();  
inner function
```



익명함수를 실행하는 방법

→ 함수 표현식을 변수에 할당하지 않고 익명 스코프 내에서 즉시 실행

```
> var msg = 'global';  
  
  (function () {  
    var msg = 'inner'; // 로컬변수이므로 전역변수와 충돌없이 사용  
    console.log(msg);  
  })();  
inner
```



JS 는 **싱글스레드**: 동시에 여러함수를 수행할 수 없다.

따라서, 함수가 호출이 되면...

- 현재 함수의 실행을 중단
- 전달하는 매개변수와 함께 호출한 함수로 제어를 넘김
- 함수의 수행이 끝나면, 실행이 중단되었던 함수로 return



```
> var Func = function () {  
    // this와 arguments 라는 매개변수 2개를 더 참조할 수 있다.  
    console.log(arguments);  
    console.log(this);  
};
```

```
> // 매개변수가 없더라도, 넘긴 파라미터를 알 수 있다.  
Func(1, 2, 3);
```

```
▶ [1, 2, 3]
```

```
▶ Window {external: Object, chrome: Object, docu
```



this :

- 함수 호출방식에 따라 가리키는 대상이 달라지는 지시어
- 실행 시점에 동적으로 바인딩 된다.



함수 실행 패턴 (this 바인딩 패턴)

1. 함수 호출 패턴
2. 생성자 호출 패턴
3. 메소드 호출 패턴
4. apply 호출 패턴



this :

- 함수 호출방식에 따라 가리키는 대상이 달라지는 지시어
- 실행 시점에 동적으로 바인딩 된다.

1. 함수 호출 패턴



```
> function MyFunc() {  
    console.log(this); // this === window (전역객체: global)  
}
```

```
// 함수 호출 패턴  
MyFunc();
```

```
▶ Window {external: Object, chrome: Object, document: document,  
MyFunc...}
```

2. 생성자 호출 패턴



```
> // 새로운 타입 정의(선언)
function Func(sName) {
    this.name = sName;
}

// Func 타입 확장 (by prototype)
Func.prototype.test = function () {
    console.log(this.name);
}

/**
 * 생성자 호출 패턴
 * - 함수가 생성자 역할 : Func.prototype.constructor === Func
 * - this 는 새로 생성된 인스턴스에 바인딩 된다.
 */
var oType1 = new Func('type1'); // this === oType1

oType1.test(); // ==> type1
type1
< undefined
```

3. 메소드 호출 패턴



```
> /**
 * 메소드 호출패턴
 * - this는 메소드를 호출하는 객체에 바인딩 된다.
 * - 메소드란? 객체의 속성으로 저장된 함수
 */
var obj = {
  name: 'obj',
  func1: function () {
    console.log(this.name);
  }
};

obj.func1();
```



apply()

- syntax: `apply([thisObj[,argArray]])`
- desc: 함수를 호출하여 지정된 개체를 함수의 `this` 값으로 대체하고 지정된 배열을 함수의 인수로 대체합니다. (다른 객체의 메소드를 마치 내 것처럼 실행)

ref.

apply: [https://msdn.microsoft.com/ko-kr/library/4zc42wh1\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/4zc42wh1(v=vs.94).aspx)

call: [https://msdn.microsoft.com/ko-kr/library/h2ak8h2y\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/h2ak8h2y(v=vs.94).aspx)

4.2. apply 호출 패턴



```
> var obj = {  
  name: 'obj',  
  func1: function () {  
    console.log(this.name);  
  }  
};
```

```
var obj2 = {name: 'obj2'};
```

```
obj.func1();
```

```
obj
```

```
> // obj.func1 이 마치 obj2 객체의 메소드 인 것처럼 실행  
obj.func1.apply(obj2);
```

```
obj2
```



Function 객체에서 제공하는 또 다른 호출 방법


- 문법: `function.bind(thisArg[,arg1[,arg2[,argN]]])`
- 활용: 파라미터로 전달되는 익명함수객체(콜백함수)의 `this`를 특정객체와 연결시킨다.
- 장점: 쉽고 직관적

```
> // 전역객체
window.name = 'global';

// obj 객체
var obj = {
  name: 'object'
};

> var fCallback = function () {
    console.log(this.name);
  };

  setTimeout(fCallback.bind(obj), 300);
```





Function.prototype.bind 는 최신브라우저에서 지원

지원되지 않는 브라우저인 경우에는??

Function 객체의 프로토타입을 확장하여 구현할 수 있다. (Polyfill)

- 라이브러리들은 구버전 브라우저에서도 동작을 보장하기 위한 기법으로 Polyfill로 브라우저의 기본 객체를 확장한다.

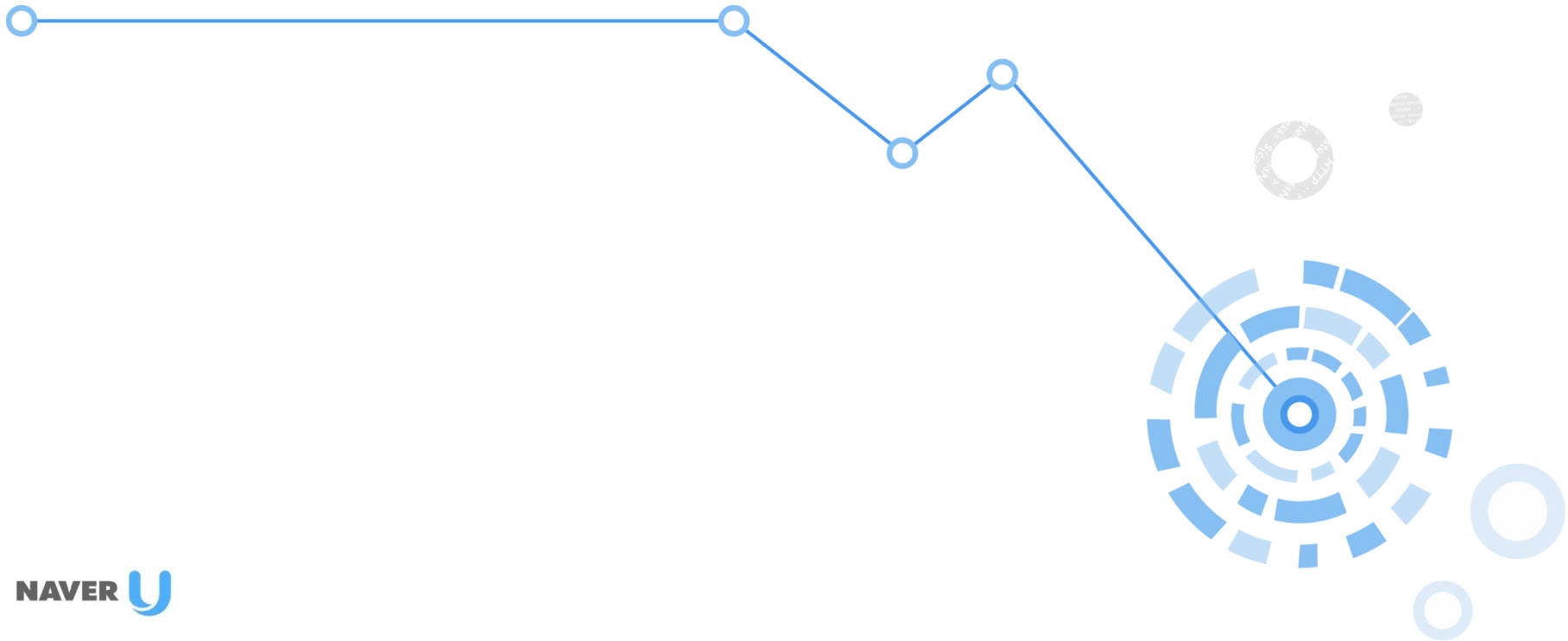
ref.

<http://stackoverflow.com/questions/15455009/js-call-apply-vs-bind>

[https://msdn.microsoft.com/ko-kr/library/ff841995\(v=vs.94\).aspx](https://msdn.microsoft.com/ko-kr/library/ff841995(v=vs.94).aspx)

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

7. 실행문맥 (Execution Context)





Execution Context (EC)

1. 함수가 실행되는 실행환경 객체
2. 자바스크립트가 변수를 저장하는 내부객체



JS는 싱글 스레드 랭귀지

- 한번에 하나의 작업만 수행
- JS엔진은 실행시점의 함수 환경에서 메모리를 참조한다



즉, 함수가 실행되면 그때 참조할 수 있는 메모리 영역이 생성되는데 이 영역을 실행컨텍스트라고 한다.



EC 구성 (함수내부 참조 가능 공간)

1. 로컬변수 저장공간
2. 스코프 체인(상위 컨텍스트 접근 순서)
3. 멤버변수 영역 (this)



변수참조 접근 순서 (함수가 중첩으로 수행되는 경우)

1. 자신의 EC에서 찾는다
2. 없다면, 상위 함수의 EC에서 찾는다.
3. Global EC 까지 반복... (chain)



Execution Context

Variable Object (VO)
(로컬변수, 매개변수 등)

Scope Chain
(상위 컨텍스트 객체 정보)

this
(실행시점에 바인딩 되는 객체)

Scope Chain

Parent EC (VO)

...

Global EC (VO)



스코프 체인

함수 실행 환경에 대한 연결

→ Function 객체의 Scope 공간(EC) 접근 순서



함수 스코프

JS엔진(인터프리터)은 함수를 실행하기 직전에 EC를 구성하고, 이렇게 생성된 EC에 의해 함수 스코프가 결정된다.

즉, 함수가 **실행될 때** 함수내의 **scope** 가 **동적으로 결정**된다.

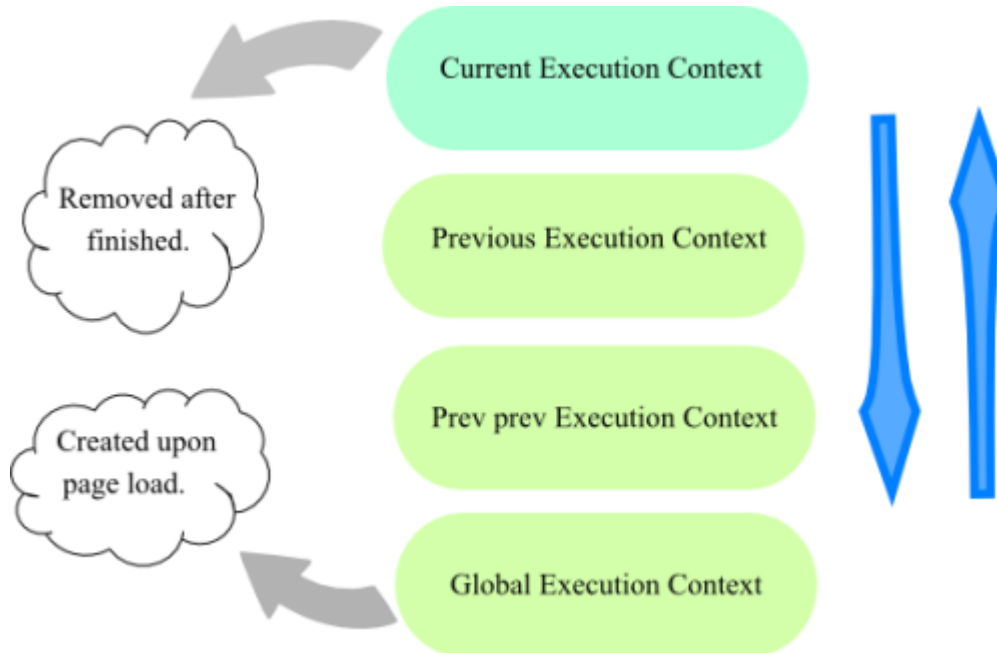
Ref.

<http://ryanmorr.com/understanding-scope-and-context-in-javascript/>



Default EC (Global Context)

프로그램이 처음 초기화 될때 EC Stack 최상위에 들어가는 객체
모든 실행문맥(EC)에서 참조 가능한 전역객체를 의미 (window)





```
> // Globale EC
var msg = 'global';
var g_msg = 'g_msg';

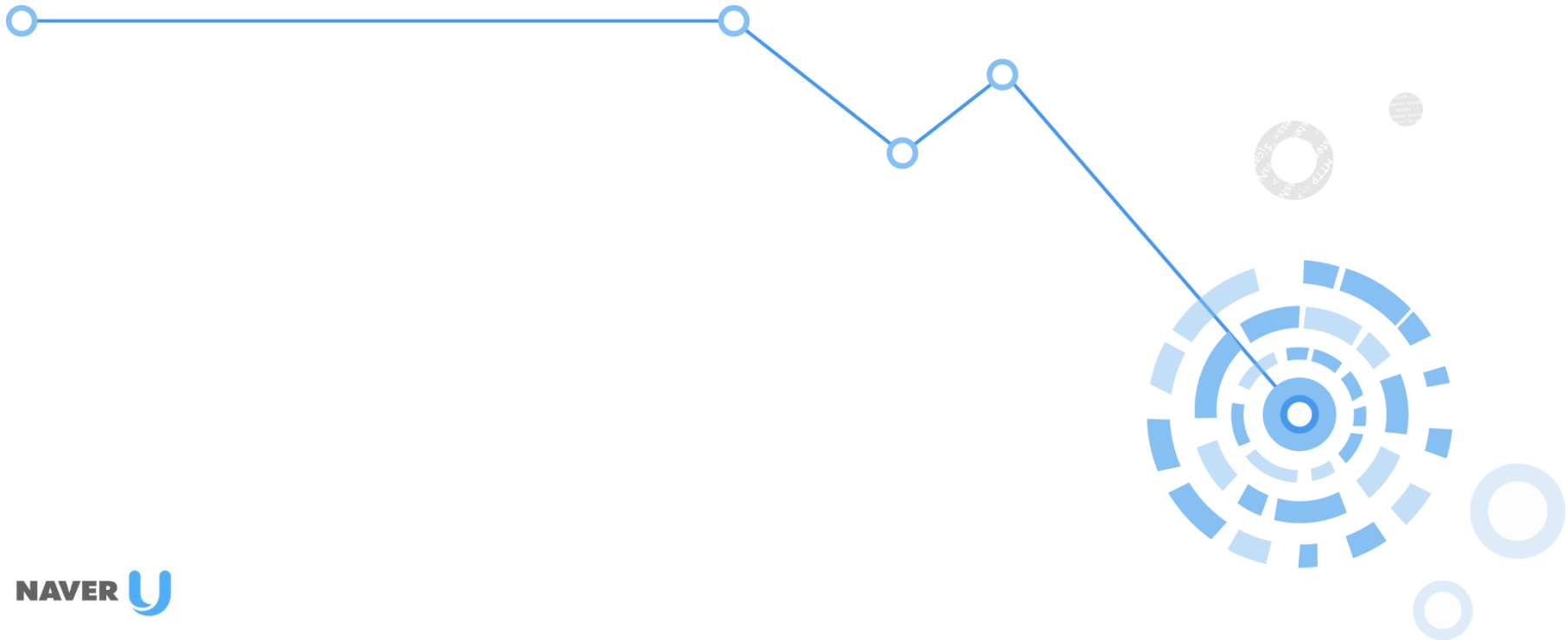
// EC1: 함수 실행시점 EC구성(즉시실행함수)
(function () {

    var msg = 'inner';

    // EC2: 함수 실행시점 EC구성(즉시실행함수)
    (function () {
        var msg = 'inner2';
        console.log(msg);    // EC2 로컬 find 끝
        console.log(g_msg); // EC2 -> EC1 -> Global EC
    })();

})();
```

8. 클로저





Closer ?

- EC는 함수가 실행시 생성, 종료시 GC
- 단, EC 참조 유지시 GC 대상 제외

이렇게 유지된 EC가 바로 클로저!!!



실행시점의 실행문맥을 유지하는 기법

무분별 하게 사용하면, 메모리 누수의 원인이 된다.



```
> // 일반적인 로컬 변수
(function () {
    var str = 'local'; // 로컬 변수 함수 종료시 GC
    console.log(str);
})();
```

```
> str;
```

```
✖ ▶ Uncaught ReferenceError: str is not defined(...)
```

```
> |
```



```
> function Module() {  
    var str = 'closer';  
  
    return {  
        func: function() {  
            if (str) {  
                str = str + str;  
            }  
            console.log(str);  
        }  
    };  
}  
  
// 리턴객체 변수 저장  
var module = Module();  
  
// 함수의 실행은 끝났지만, 내부에서 로컬변수 str 접근가능  
module.func();  
closercloser
```



Module ?

→ 내부의 상태나 구현내용을 숨기고 인터페이스만 제공하는 함수 or 객체



JS객체는 내부상태를 숨길 수 있을까?



> // 일반적인 객체 : 모든 속성 접근 가능

```
var obj = {  
  name: "홍길동",  
  getName: function () {  
    return this.name;  
  }  
};
```

```
obj.getName();
```

< "홍길동"

> console.dir(obj);

```
▼ Object ⓘ  
  ▶ getName: function ()  
    name: "홍길동"  
  ▶ __proto__: Object
```



```
> var obj = (function () {  
    var name = "홍길동";  
  
    return {  
        getName: function() {  
            return name;  
        }  
    }  
})();  
  
obj.getName(); // 동일하게 동작  
◀ "홍길동"
```

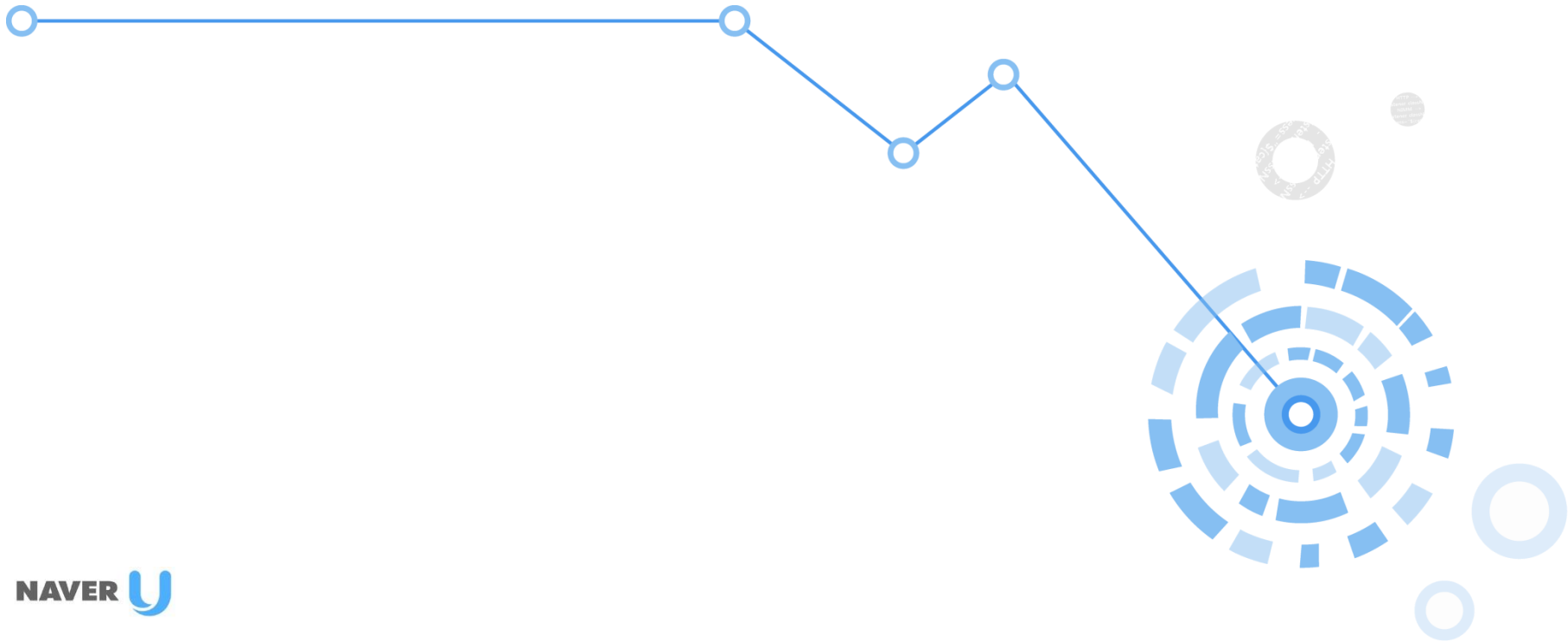
```
> console.dir(obj);  
▼ Object ⓘ  
  ▶ getName: function ()  
  ▶ __proto__: Object
```



Module Pattern

→ 내부구현이 숨겨진 객체를
즉시실행 함수를 이용해 반환하는 패턴

9. 이벤트 루프





Event Loop ? (evnet driven)

→ 동작요청 후 처리 결과를 callback 으로 전달하는 방법



동기

→ JS는 싱글 스레드로 동작

→ 현재 실행중인 함수의 실행이 끝나야 다음 함수를 실행



비동기

→ 실행순서가 아닌 이벤트 발생 순서로 콜백함수 실행

→ 싱글 스레드이므로 실행 순서는 보장되지만,

완료(callback) 시점은 보장되지 않는다.

Event Loop



```
> function a() { console.log('a'); }  
function b() { console.log('b'); }
```

```
a();  
b(); // a의 실행이 끝난 뒤에 실행 가능
```

```
a
```

```
b
```

```
> setTimeout(a, 300); // b를 300ms 뒤에 수행  
b(); // a의 실행과 무관하게 먼저 실행
```

```
b
```

```
< undefined
```

```
a
```

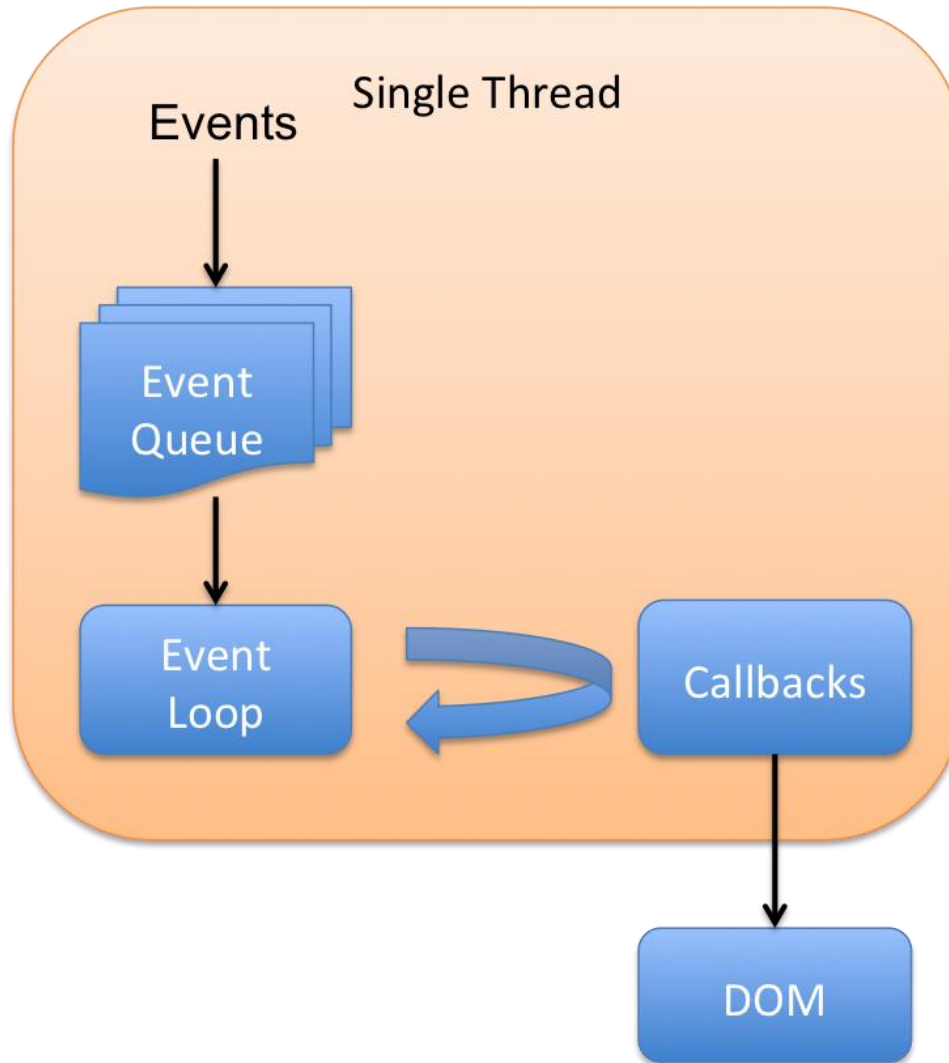
```
> |
```



싱글스레드 지만

- 실행순서는 보장되지만, 나중에 실행된 함수가 먼저 수행될 수 있다.
ex. 네트워크 응답처리: ajax call, 마우스 클릭 이벤트, 타이머 이벤트 등등..

Event Loop : Queue (FIFO)



이벤트 Queue	함수실행
setTimeout	타임 이벤트 등록 (콜백함수 a)
	b(); // 함수 실행
	a(); // fire callback



콜백시점이 겹치는 경우
먼저 들어온 이벤트 콜백이 수행되고,

기존에 함수가 실행중이라면,
해당 함수가 종료될때까지 Waiting...
(JS는 싱글스레드)



```
> function a() { console.log('a'); }  
function b() { console.log('b'); }
```

```
var nThread1 = setTimeout(a, 300); // 300ms 뒤에 a를 한번만 수행
```

```
var nThread2 = setInterval(b, 100); // 100ms 마다 b 호출
```

```
< undefined
```

2 b

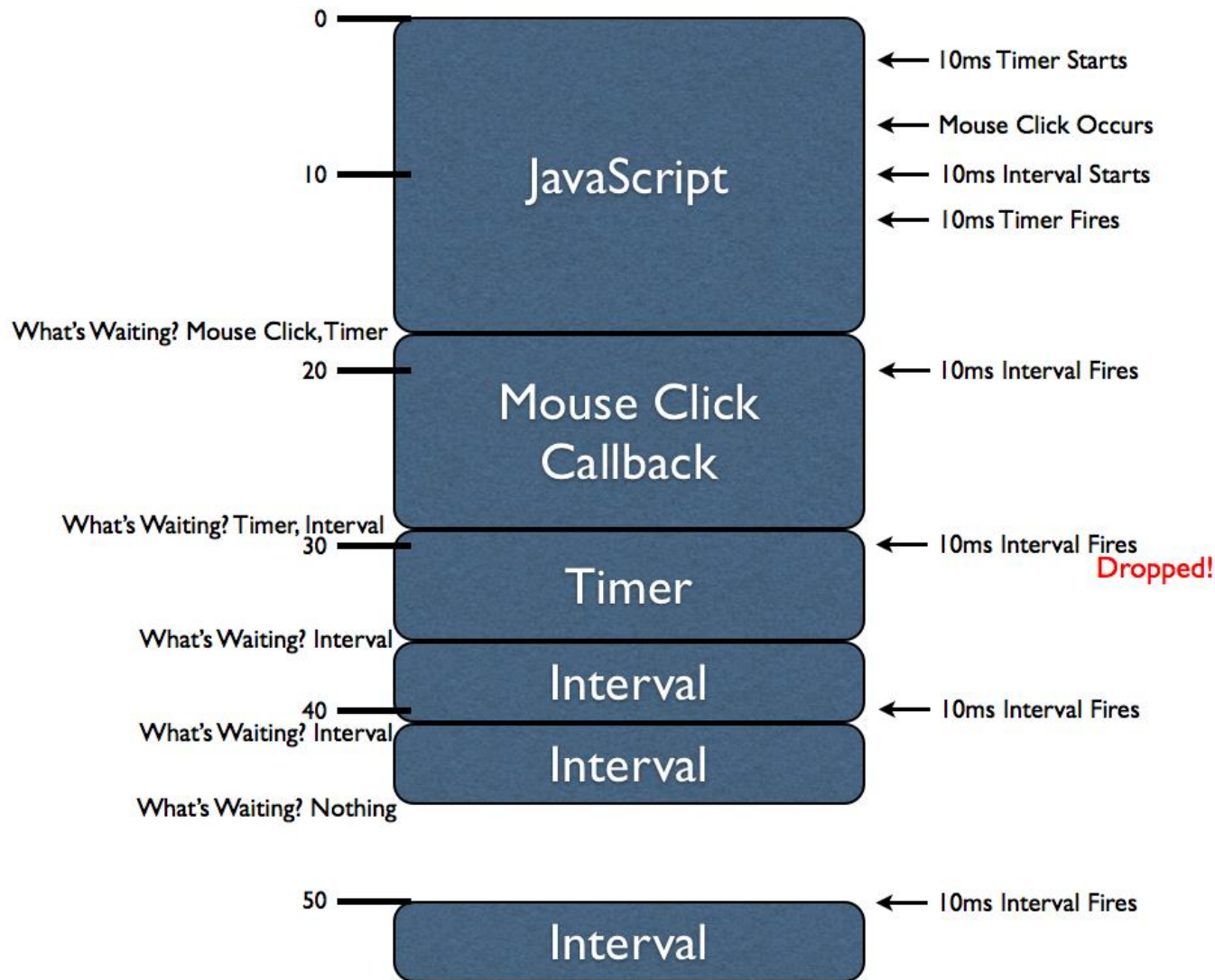
a

658 b

```
> clearTimeout(nThread1); // setTimeout 요청을 제거
```

```
> clearInterval(nThread2); // setInterval 요청을 제거
```

Event Loop





```
> /**
 * 짧은 시간에 연속적인 비동기 요청이 있을때
 * 가장 마지막 요청만 수행되도록 하기
 */
function runTest(nDelay) {
    // 앞의 함수를 이용해 이부분을 변경해본다.
    console.log(nDelay);
}

runTest(100);
runTest(200);
runTest(300); // 이 요청만 수행되어야 한다.
```



```
> function runTest(nDelay) {  
    clearTimeout(window.nThread);  
  
    window.nThread = setTimeout(function () {  
        console.log(nDelay);  
    }, nDelay);  
  
}  
  
runTest(100);  
runTest(200);  
runTest(300);  
  
< undefined
```

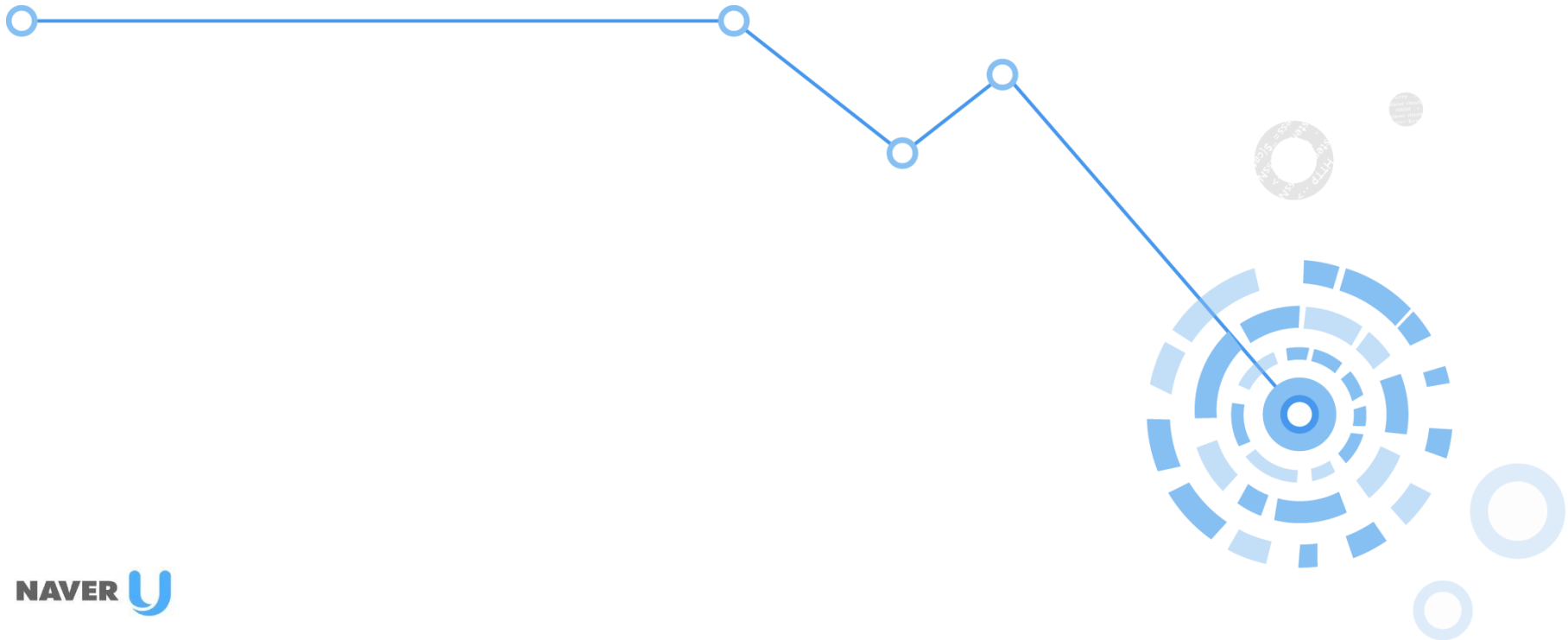
300

Ref.

<http://ejohn.org/blog/how-javascript-timers-work/>

<http://www.altitudelabs.com/blog/what-is-the-javascript-event-loop/>

10. DOM API 맛보기...





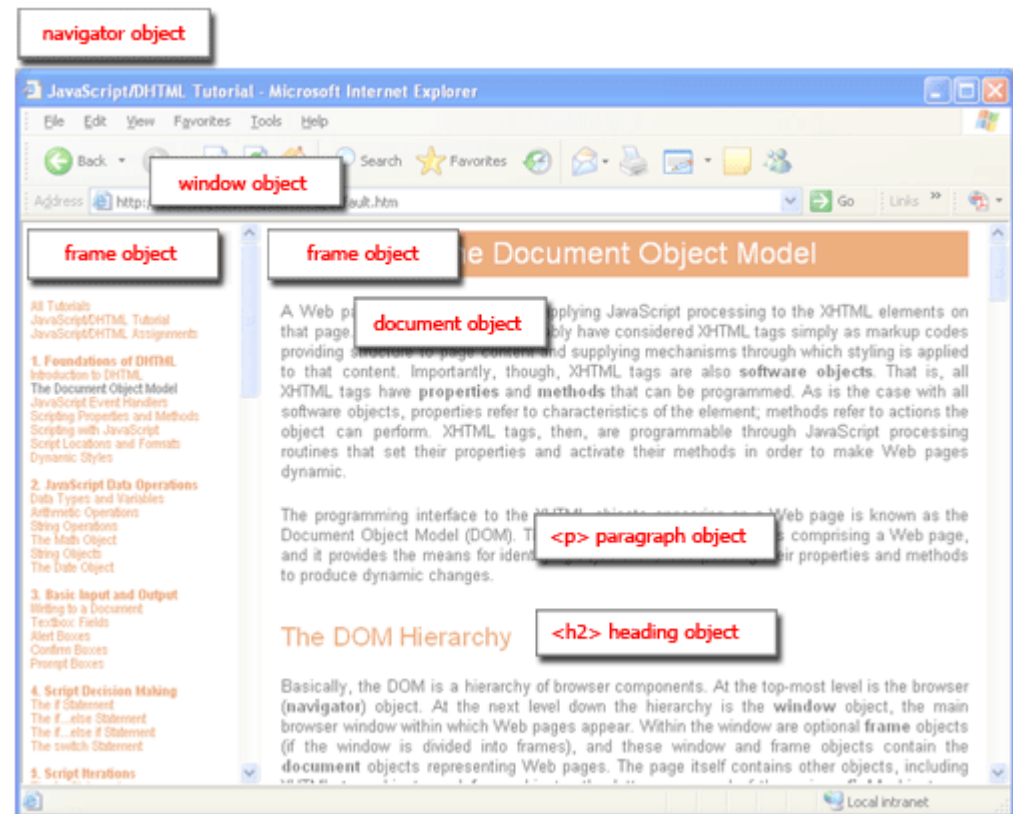
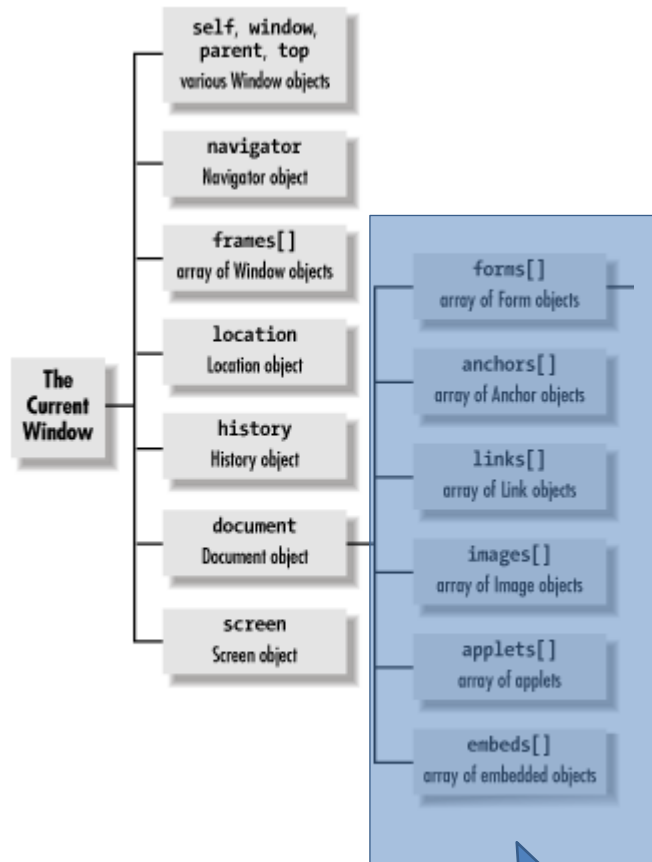
BOM (Browser Object Model)

→ 브라우저와의 인터랙션을 위해 제공되는 JS 인터페이스 객체



DOM (Document Object Model)

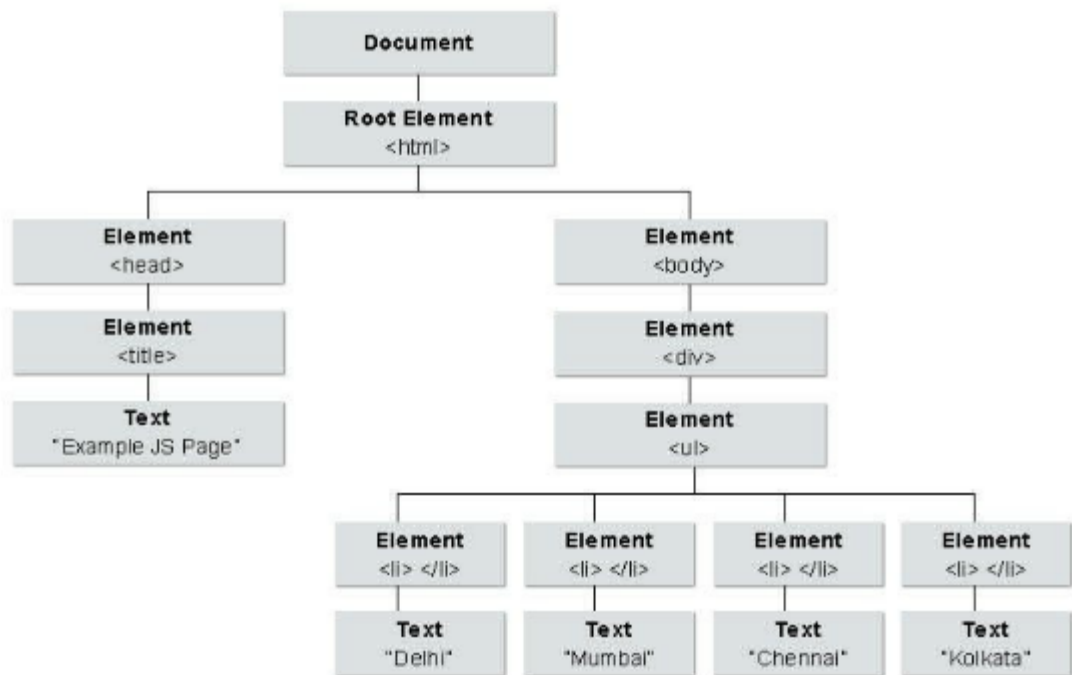
- 문서(html, xml)와의 인터랙션을 위한 JS 인터페이스 객체
- 렌더링을 위한 정보를 담고있는 트리 구조의 객체
- 이 객체의 속성을 변경하면, 브라우저 화면에 반영된다.

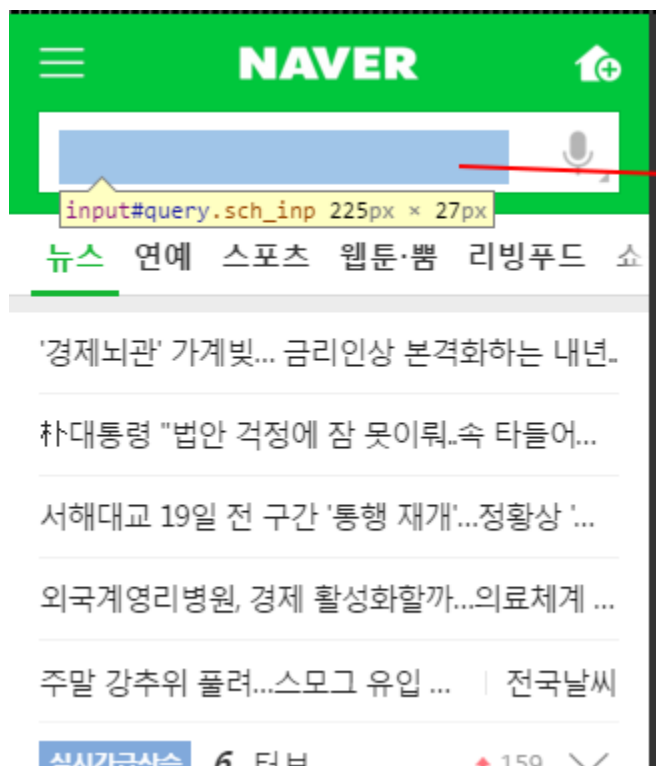


DOM



```
<html>
<head>
<meta charset=utf-8 />
<title>DOM</title>
</head>
<body>
  <div id="Rayyan">
    <ul>
      <li>Delhi</li>
      <li>Mumbai</li>
      <li>Chennai</li>
      <li>Kolkata</li>
    </ul>
  </div>
</body>
</html>
```





```
...<input type="search" id="query" name="
  <span class="sch_inpw_sch_inpw">
    <span class="sch_inpw_in">
      <input type="search" id="query" name="
        "검색어 입력" value class="sch_inp" au
        autocomplete="off" autocapitalize="of
        "255" data-initval onfocus="this.setA
        focus','focus');">
      <input type="hidden" name="where" val
      <input type="hidden" id="sm" name="sm
        "mtp_hy">
      <button id="clear_input" type="button
        "sch_del" style="display:none">...</butt
      <button id="_rs_show_btn" type="butto
        "sch_btn sch_region" onclick=
        "nclk(this,'sch.qbtn','','');">...</butt
```



- > // 유일한 id 속성으로 엘리먼트 탐색 (엘리먼트)

```
document.getElementById('query');
```

- < `<input type="search" id="query" name="query" title=`

- > // 태그명으로 엘리먼트 탐색 (배열)

```
document.getElementsByTagName('input');
```

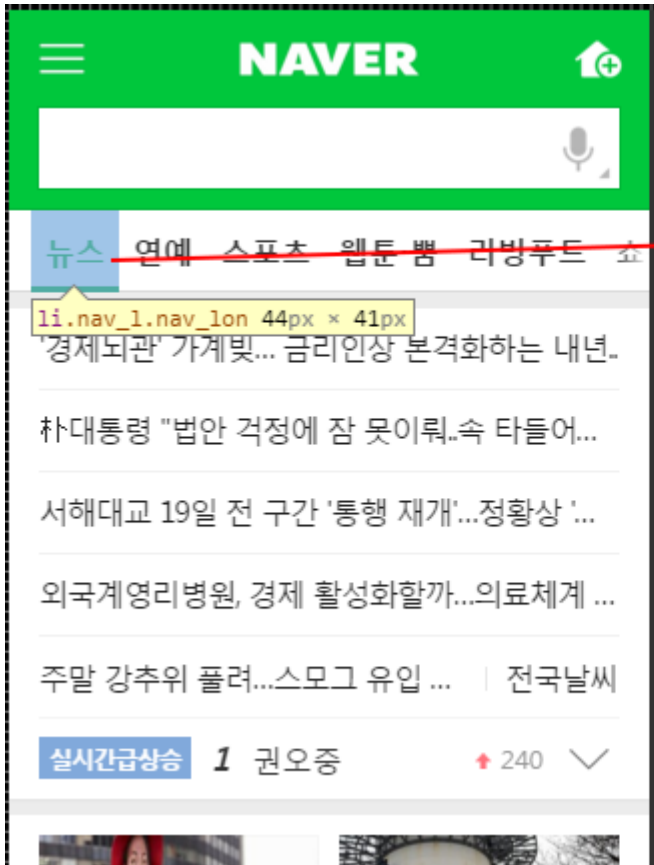
- < [
 `<input type="search" id="query" name="query" title=`

- > // 태그명으로 엘리먼트 탐색 (배열)

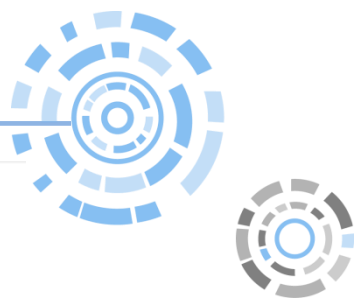
```
document.getElementsByTagName('input');
```

- < [
 `<input type="search" id="query" name="query" title=`

DOM 객체를 변경하여 브라우저 반영하기



```
translate3d(0px, 0px, 0px); height: 10px;
▼ <ul class="nav_u" data-ids="
  ['NEWS', 'ENT', 'SPORTS', 'BBOOM', 'LIVI
  ', 'BEAUTY', 'VIDEO', 'CARGAME']">
  ▶ <li class="nav_1 nav_lon">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  ▶ <li class="nav_1">...</li>
  </ul>
  </nav>
</div>
</div>
```



```
> // 화면에서 nav_1 클래스를 가진 엘리먼트 찾기
var items = document.getElementsByClassName('nav_1');
console.dir(items);
```

```
▼ HTMLCollection[9] ⓘ
  ▶ 0: li.nav_1.nav_lon
  ▶ 1: li.nav_1
```

```
> // 첫번째 LI엘리먼트 DOM 객체 속성 살펴보기
console.dir(items[0]);
```

```
▼ li.nav_1.nav_lon ⓘ
  __jindo__id: "e145042833894169160147"
  accessKey: ""
  ▶ attributes: NamedNodeMap
    baseURI: "http://m.naver.com/"
    childElementCount: 1
  ▶ childNodes: NodeList[3]
  ▶ children: HTMLCollection[1]
  ▶ classList: DOMTokenList[2]
  className: "nav_1 nav_lon"
  clientHeight: 41
```

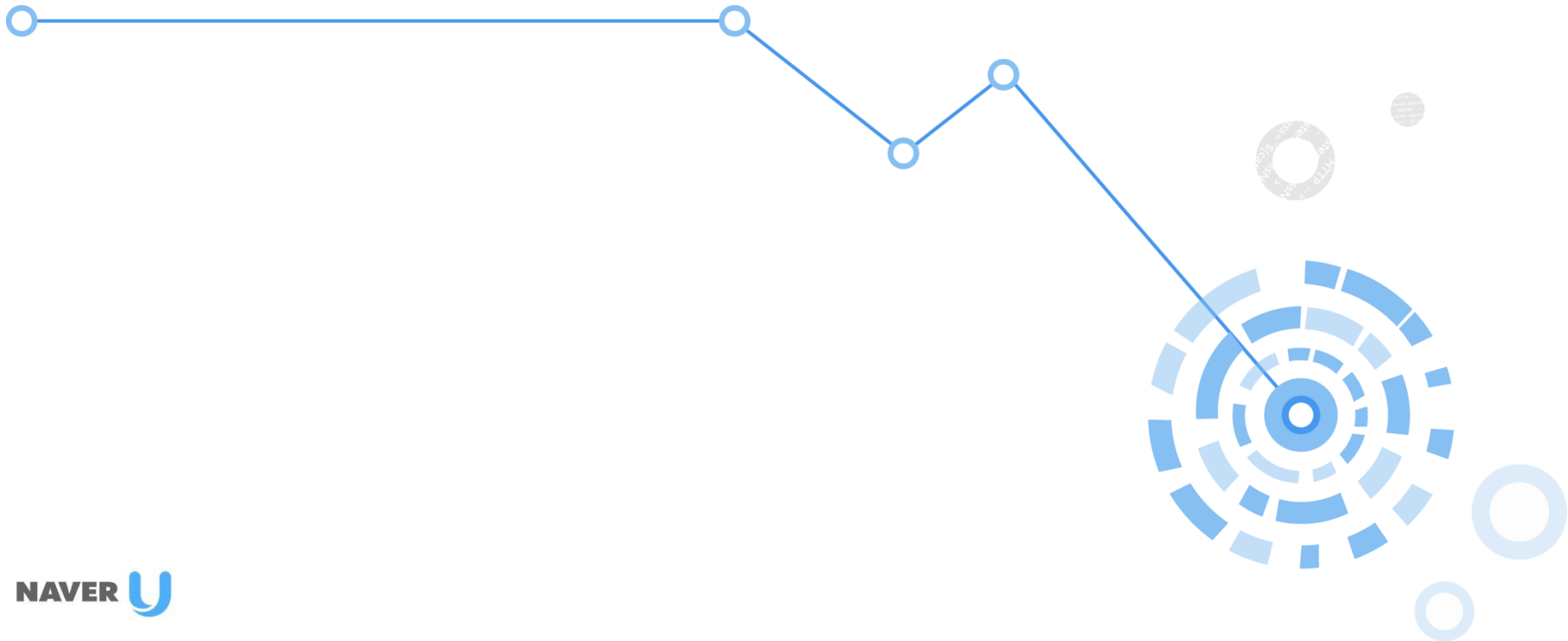


```
> var items = document.getElementsByClassName('nav_1');
```

```
items[0].className = "nav_1";           // css 클래스 변경 (비활성화)
```

```
items[0].className = "nav_1 nav_1on"; // css 클래스 변경 (활성화)
```

기타. 이벤트 처리





이벤트 주도 개발

- 이벤트가 발생하면, 미리 등록된 콜백 함수를 비동기로 실행.
- 이벤트 핸들러: 이벤트 발생시 실행되는 콜백함수



이벤트 핸들러 등록 방식

- DOM0 : 전통적인 연결 방법
- DOM1~3 : W3C 연결 방법

Ref.

https://developer.mozilla.org/ja/docs/DOM_Levels

http://www.w3schools.com/jsref/met_element_addeventlistener.asp

<http://www.w3.org/TR/DOM-Level-3-Events/>

https://developer.mozilla.org/ja/docs/DOM_Levels



DOM 0

- 장점: 모든 브라우저에 일관되게 적용가능
- 단점: 대상 엘리먼트에 콜백함수를 하나만 등록 가능

```
> var links = document.getElementsByClassName('nav_a');  
  
// 클릭 이벤트 핸들러 등록하기  
links[0].onclick = function () {  
    alert('클릭');  
    console.log(this); // 콜백함수의 this는 이벤트가 발생한 element  
};
```



DOM 2

- 장점: 대상 엘리먼트에 하나 이상의 콜백함수를 등록가능
- 단점: 브라우저마다 차이가 있음

```
> var links = document.getElementsByClassName('nav_a');
```

```
// 클릭 이벤트 핸들러 여러개 등록하기
```

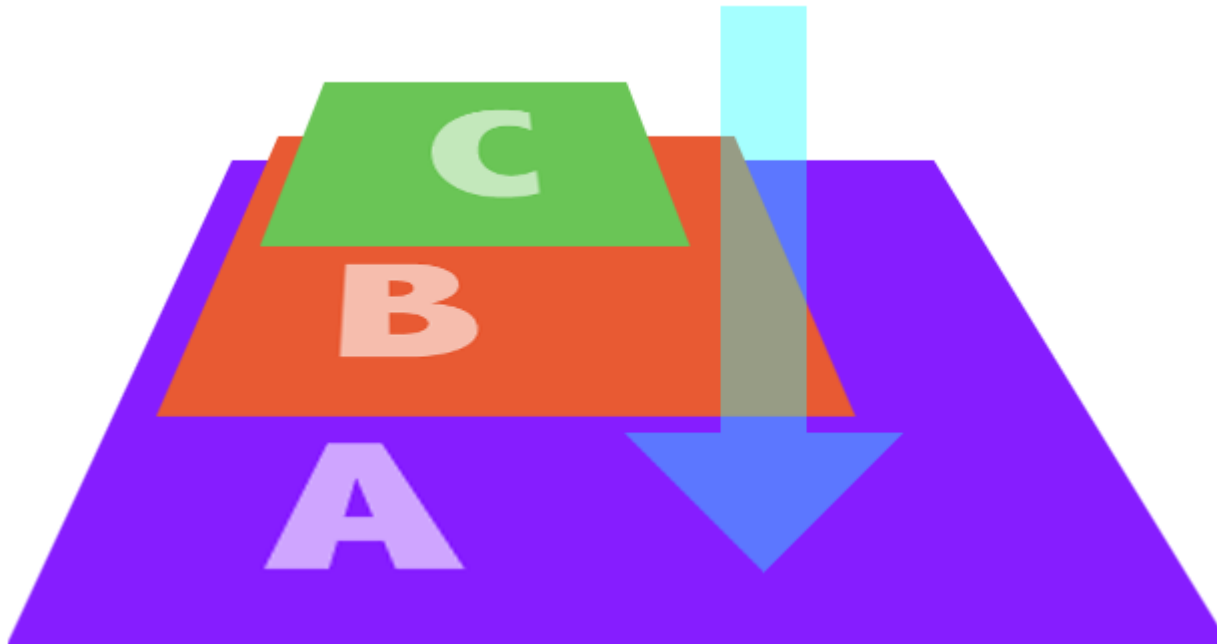
```
links[0].addEventListener('click', function () {  
    console.log(1);  
    console.log(this);  
});
```

```
links[0].addEventListener('click', function () {  
    console.log(2);  
    console.log(this);  
});
```



이벤트 버블링

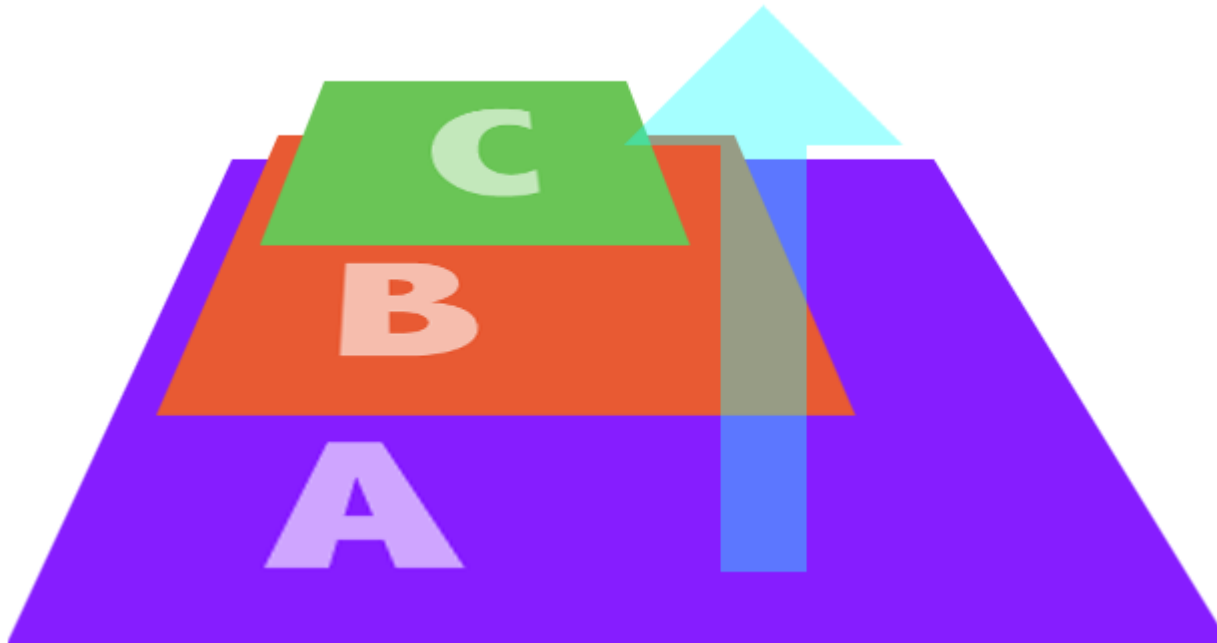
- 이벤트를 DOM 객체에 전파하는 순서
- 자식노드부터 이벤트가 발생하여 부모로 전파





이벤트 캡처링

- 이벤트를 DOM 객체에 전파하는 순서
- 버블링과 반대





```
> var nav = document.getElementById('nav');  
var link = document.getElementsByClassName('nav_a')[0];  
  
// 캡처링  
nav.addEventListener('click', function () {  
    console.log('nav');  
}, true);  
  
link.addEventListener('click', function () {  
    console.log('link');  
}, true);  
  
// 버블링  
nav.addEventListener('click', function () {  
    console.log('nav1');  
}, false);  
  
link.addEventListener('click', function () {  
    console.log('link1');  
}, false);
```

※ 기술직무교육 강의안 작성 가이드



강의안은 발표와 인쇄를 겸하는 문서이므로 배경은 흰색으로!



글씨 크기는 최소 14폰트 이상으로! (18폰트 이상을 권장)



글씨를 작게 해야 한 장에 딱 맞는 소스는 어떻게 표현할까요?

→ 2장으로 나눕니다. 앞장에는 중요한 것만 크게 넣고 생략한 다음에,
다음 장에 전체 소스를 작은 크기로 넣어두면 됩니다. (교재로 볼 수 있도록)