OOP

Object

- Object(객체)
  - ✓ 데이터를 저장하고 사용할 수 있는 속성과 기능을 수행하는 메서드를 모아놓은 것
  - ✓ 객체의 종류
    - 사용자 정의 객체 – 직접 생성하는 객체
    - 내장 객체 – 제공되는 객체
      - 일반 객체
      - BOM(브라우저 객체 모델) – 브라우저에서 제공하는 객체
      - DOM(문서 객체 모델) – html 문서에서 제공하는 객체
    - 3rd Party 객체 – 외부 라이브러리에서 제공하는 객체

Object

- 사용자 정의 객체
  - ✓ 객체 생성
    - 사용자 정의 객체 생성 - var(let, const) 객체명 = {속성이름:데이터, 속성이름:데이터....}
    - Built-in Object 나 직접 생성자를 만든 경우의 객체 생성 - var 객체명 = 생성자메서드(매개변수)
    - var 대신에 let 이나 const 사용 가능
  - ✓ 객체의 요소 접근

    객체명[속성이름] 또는 객체명.속성이름
  - ✓ function - method
    - 속성 안에 메서드를 작성하거나 대입하는 것이 가능
    - 객체를 통한 내부 메서드 호출

      객체.메서드(매개변수)

Object

❖ 사용자 정의 객체

✓ object1.html

```
<script>
        var object = {
            name : "박문석",
            phone : "01037901997",
            email : "ggangpae1@gmail.com"
        };
        document.write("name:" + object["name"] + "<br>");
        document.write("phone:" + object.phone);
</script>
```

name:박문석
phone:01037901997

Object

- 사용자 정의 객체
  - ✓ object2.html

```
<script>
    var obj = {
        name : "박문석",
        phone : "01037901997",
        email : "ggangpae1@gmail.com",
        disp : function() {
            document.write(this.name + "<br>");
        }
    };

    obj.disp();
    document.write("====================" + "<br>");

    let item = {
        ["one" + "two"]: 12
    };
    document.write(item.onetwo);
</script>
```

```
박문석
====================
12
```

Object

- 사용자 정의 객체
  - ✓ for 임시변수 in 객체{ 수행할 내용 }
    - 객체의 모든 속성을 순서대로 임시 변수에 대입
  - ✓ 속성 in 객체
    - 객체에 속성이 있으면 true를 리턴하고 없으면 false를 리턴
  - ✓ with
    - with(객체명) { }를 이용하면 { }안에서는 객체 이름 생략 가능
  - ✓ 객체의 속성 추가 및 수정 – 속성이 없으면 추가
    - 객체.속성 = 값
  - ✓ 객체의 속성 제거
    - delete(객체.속성)
  - ✓ 객체 내의 메서드 안에서 this 를 이용하면 자신의 객체에 속한 속성을 호출해서 사용

Object

- 사용자 정의 객체
  - ✓ object3.html

```
<script>
        var object = {
                name : "박문석",
                phone : "01037901997"
        };

        object.email = "ggangpae1@gmail.com";
        delete (object.phone);
        for(var key in object)
                document.write(key + ":" + object[key] + "<br>");
</script>
```

name:박문석
email:ggangpae1@gmail.com

Object

- 사용자 정의 객체
  - ✓ 생성자 함수
    - 객체를 생성할 수 있도록 해주는 메서드
    - 객체의 속성을 만들 때는 생성자 메서드 안에서 this를 이용해서 생성
    - this를 붙이지 않으면 지역 변수로 생성
    - 형식

```
function 생성자메서드명(매개변수){
    내용;
}
```

    - 생성자를 이용한 객체 생성

```
var 객체명 = new 생성자메서드(매개변수);
```

Object

- 사용자 정의 객체
  - ✓ constructor.html

```
<script>
        function person(name, phone) {
                this.이름 = name;
                this.전화번호 = phone;
        }

        var object = new person("박문석", "01037901997");

        for ( var key in object)
                document.write(key + ":" + object[key] + "<br>");
</script>
```

이름:박문석
전화번호:01037901997

Object

- 사용자 정의 객체
  - ✓ let_this.html

```
<script type="text/javascript">
    var music = "음악";
    console.log(this.music);
    let sports = "축구";
    console.log(this.sports);
</script>
```

음악
undefined

Object

- 사용자 정의 객체
  - ✓ this
    - var 키워드로 music 변수를 선언하고 값(음악)을 할당
    - 현재 스코프는 글로벌 오브젝트이고 this가 글로벌 오브젝트를 참조
    - console.log(this.music)을 실행하면 글로벌 오브젝트의 music 변수 값인 음악을 출력
    - 글로벌 오브젝트가 window 오브젝트는 아니지만 window 오브젝트로 글로벌 변수에 접근할 수 있음
    - 논리적으로 접근하면 자바스크립트에 글로벌 오브젝트는 있지만 window 오브젝트는 없음
    - window 오브젝트에는 DOM(Document Object Model) 관련 메서드도 있고 window 오브젝트 자체의 프로퍼티도 있음
    - window 오브젝트에 글로벌 오브젝트가 존재하는 개념
    - let 키워드로 sports 변수를 선언하고 축구 를 할당한 후 this.sports로 sports 변수 값을 출력하면 undefined가 출력되는데 this가 window 오브젝트를 참조하는데 window 오브젝트에 let 변수가 없다는 것은 window 오브젝트에 let 변수가 설정되지 않는다는 의미가 되는데 이 점이 var 변수 와 let 변수의 차이

Object

- 사용자 정의 객체
  - ✓ Computed Property Name
    - 문자열을 연결해서 프로퍼티의 키로 사용 가능
    - 문자열과 변수를 조합하여 오브젝트의 프로퍼티 이름으로 사용하는 것도 가능

Object

- ❖ 사용자 정의 객체
  - ✓ Computed Property Name
    - property_key.html

```
<script type="text/javascript">
    let item = {
        ["one" + "two"]: 12
    };
    console.log(item.onetwo);

    item = "tennis";
    let sports = {
        [item]: 1,
        [item + "Game"]: "윔블던",
        [item + "Method"](){
            return this[item];
        }
    };
    console.log(sports.tennis);
    console.log(sports.tennisGame);
    console.log(sports.tennisMethod());
```

Object

- ❖ 사용자 정의 객체
  - ✓ Computed Property Name
    - property_key.html

```
let one = "sports";
    let {[one]: value} = {sports: "농구"};
    console.log(value);
</script>
```

12

1

윔블던

1

농구

OOP

- class
  - ✓ ES6 부터 class가 도입되었으며 class 문법은 메서드와 유사하게 class 표현식과 class 선언 두 가지 방법을 제공
  - ✓ 선언

    class 클래스이름 { } // Class 선언

    class 클래스이름 extends 상위클래스이름 { }
  - ✓ class는 Hoisting의 영향을 받지 않기 때문에 class를 사용하기 위해서는 class를 먼저 선언해야 하며 그렇지 않으면 에러가 발생
  - ✓ 클래스 안의 메서드 나 속성은 객체.메서드명() 또는 객체.속성명 으로 호출

OOP

- class
  - ✓ class_decalartion.html

```
class Member {
        getName() {
            return "이름";
        }
};

let obj = new Member();

console.log(obj.getName());
```

이름

OOP

- class
  - ✓ 표현식

    let name = class { }

    let name = class inner_name { }

    let name = class extends super_name { }

    let name = class inner_name extends super_name { }

    - 할당 연산자(=) 왼쪽에 클래스 이름을 작성하고 오른쪽에 class 키워드를 작성하고 블록 { } 안에 클래스 코드를 작성할 수 있는데 이를 클래스 표현식이라고 함
    - function 키워드 메서드의 메서드 표현식과 작성 방법이 같음
    - class와 extends 키워드 사이의 inner_name은 클래스 안에서 자신을 호출할 때 사용
    - function 키워드 메서드에도 inner_name을 직성할 수 있지만 사용하지 않듯이 클래스도 사용하지 않음

Class

- class
  - ✓ 표현식
    - class_expression.html

```
<script type="text/javascript">
    let Member = class {
        getName() {
            return "이름";
        }
    };

    let obj = new Member();
    console.log(obj.getName());
</script>
```

이름

Class

- class
  - ✓ 특징
    - 클래스 안에 메서드를 작성할 때 function 키워드 생략 가능
    - prototype에 프로퍼티 연결 가능
    - class_function.html

```
<script type="text/javascript">
    let Member = class {
        setName(name) {
            this.name = name;
        }
    };
    Member.prototype.getName = function(){return this.name;}
    let obj = new Member();
    obj.setName('아담')
    console.log(obj.getName());
</script>
```

아담

Class

- class
  - ✓ constructor
    - constructor는 클래스 인스턴스를 생성하고 생성한 인스턴스를 초기화하는 역할을 수행하는 메서드
    - new 생성자( )를 실행하면 클래스.prototype.constructor가 호출됨
    - 클래스에 constructor를 작성하지 않으면 prototype의 constructor가 호출되는데 이를 default constructor 라고 하며 constructor가 없으면 인스턴스를 생성할 수 없음
    - 자바스크립트에서는 생성자가 클래스 자신의 인스턴스 이외의 인스턴스를 리턴할 수 있음

Class

- class
  - ✓ constructor
    - class_constructor.html

```
<script type="text/javascript">
    let Member = class {
        constructor(name){
            this.name = name
        }

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

    let obj = new Member("카리나");
    console.log(obj.getName());
```

Class

- ❖ class
  - ✓ constructor
    - ● class_constructor.html

```
class Person {
        constructor(){
            return {name: "홍길동"};
        }
        getName(){
            return "이름";
        }
    }
    let personObj = new Person();

    console.log(personObj.name);
    console.log(personObj.getName);
</script>
```

카리나
홍길동
undefined

Class

- class
  - ✓ getter, setter
    - 클래스 내부의 메서드 이름앞에 get 이나 set을 추가하면 getter 와 setter 가 되는데 이렇게 되면 메서드를 속성처럼 사용하는 것이 가능
    - class_property.html

```
<script type="text/javascript">
    let Member = class {
        get getName() {
            return this.name;
        }

        set setName(name) {
            this.name = name;
        }
    };

    let obj = new Member();
    obj.setName = '아담';
    console.log(obj.getName);

</script>
```

아담

Class

- class
  - ✓ static 메서드
    - 클래스의 prototype에 연결되지 않고 클래스에 직접 연결하기 위한 메서드
    - 인스턴스로는 호출할 수 없고 클래스 이름으로만 호출 가능
    - 메서드를 호출하기 위해서는 메서드가 Function 오브젝트이어야 함
    - 엔진이 class 키워드를 만나면 클래스 안에 정적 메서드 작성 여부를 체크하고 정적 메서드가 있으면 이를 Function 오브젝트로 생성하는데 이렇게 하는 것은 선언된 클래스 바로 아래에서 정적 메서드를 호출할 수 있기 때문
    - class_static.html

```
<script type="text/javascript">
    class Sports {
        static getGround() {
            return "상암구장";
        }
    };
    console.log(Sports.getGround());
</script>
```

상암구장

Class

- class
  - ✓ computed name
    - 클래스의 메서드 이름을 조합하여 생성 가능
    - class_computed.html

```
<script type="text/javascript">
    let type = "Type";
    class Sports {
        static ["get" + type](kind){
            return kind ? "스포츠" : "음악";
        }
    }
    console.log(Sports["get" + type](1));
</script>
```

스포츠

Class

- class
  - ✓ this
    - static 메서드 안에서 this는 클래스 오브젝트를 참조
    - constructor 안에서 this.constructor().static메서드()를 이용하면 static 메서드 호출 가능
    - class_this.html

```
<script type="text/javascript">
    class Sports{
        constructor(){
            console.log(Sports.getGround());
            console.log(this.constructor.getGround());
        }
        static getGround(){
            return "상암구장";
        }
    };
    let obj = new Sports();
</script>
```

상암구장
상암구장

Class

- class
  - ✓ new.target
    - new.target은 메타(meta) 프로퍼티로 생성자 메서드와 클래스에서 constructor를 참조
    - new 연산자로 인스턴스를 생성하지 않으면 new.target 값은 undefined

Class

- class
  - ✓ new.target
    - class_newtarget.html

```
<script type="text/javascript">
    let sports = function(){
        console.log(new.target);
    }
    sports();
    new sports();
    class Sports {
        constructor(){
            console.log("Sports:", new.target.name);
        }
    };
    class Soccer extends Sports {
        constructor(){
            super();
            console.log("Soccer:", new.target.name);
        }
    };
    let sportsObj = new Sports();
    let soccerObj = new Soccer();
</script>
```

Class

- class
  - ✓ new.target
    - class_newtarget.html

```
undefined
ƒ (){
        console.log(new.target);
    }
Sports: Sports
Sports: Soccer
Soccer: Soccer
```

Class

- ❖ Inheritance
  - ✓ 상속
    - 객체지향 프로그래밍에서 상속은 주요한 기능 중 하나
    - 클래스를 상속받으면 상속받은 클래스의 메서드와 프로퍼티를 사용할 수 있음
    - 형식

      class subClass extends superclass { }

Class

- Inheritance
  - ✓ 상속
    - class_extends.html

```
<script type="text/javascript">
    class Sports {
        constructor(member){
            this.member = member;
        }
        getMember(){
            return this.member;
        }
    };
    class Soccer extends Sports {
        setGround(ground){
            this.ground = ground;
        }
    };

    let obj = new Soccer('손흥민');
    console.log(obj.getMember());
</script>
```

손흥민

Class

- Inheritance
  - ✓ super
    - 서브 클래스와 슈퍼 클래스는 obj.__proto__ 구조이기 때문에 서브 클래스와 슈퍼 클래스에 같은 이름의 메서드가 존재하면 슈퍼 클래스의 메서드가 호출되지 않음
    - super 키워드를 사용하여 슈퍼 클래스의 메서드를 호출할 수 있음
    - 서브 클래스의 constructor에 super()를 작성하면 슈퍼 클래스의 constructor가 호출됨
    - 슈퍼 클래스의 메서드를 호출하려면 super.name()과 같이 super 키워드에 이어서 호출하려는 메서드 이름을 작성
    - Method Overriding
      - 서브 클래스와 슈퍼 클래스에 같은 이름의 메서드가 있는 경우를 메서드 오버라이딩(Overriding)이라고 함
      - 서브 클래스와 슈퍼 클래스의 메서드가 같은 목적을 가진 것을 나타내면서 서브 클래스의 목적에 맞도록 보완할 때 사용
      - 슈퍼 클래스의 메서드 기능을 사용하면서 서브 클래스에서 기능을 추가, 변경할 때 사용
      - 슈퍼 클래스와 서브 클래스의 메서드 기능/목적이 다를 때는 같은 이름을 사용하지 않음

Class

- ❖ Inheritance
  - ✓ super
    - ● class_super.html

```
<script type="text/javascript">
    class Sports {
        constructor(member){
            this.member = member;
            console.log(this.member);
        }

        method(){
            console.log('상위 클래스의 메서드');
        }
    };
```

Class

- ❖ Inheritance
  - ✓ super
    - ● class_super.html

```
class Soccer extends Sports {
        constructor(member){
            super(member);
            this.member = '차범근';
            console.log(this.member);
        }

        method(){
            super.method()
            console.log('하위 클래스의 메서드');
        }
    };

    let obj = new Soccer('손흥민');
    obj.method()
</script>
```

Class

- ❖ Inheritance
  - ✓ super
    - ● class_super.html

손흥민
class_super.html:21 차범근
class_super.html:14 상위 클래스의 메서드
class_super.html:26 하위 클래스의 메서드

Class

- Inheritance
  - ✓ built_in Object 상속
    - class_extends_builtin.html

```
<script type="text/javascript">
    class ExtendArray extends Array {
        constructor(){
            super();
        }

        getTotal(){
            let total = 0;
            for (var value of this){
            total += value;
            };
            return total;
        }
    };

    let obj = new ExtendArray();
    obj.push(10, 20);

    console.log(obj.getTotal());
</script>
```

Class

- ❖ Inheritance
  - ✓ built_in Object 상속
    - ● class_extends_builtin.html

      30

Iteration

- Iteration
  - ✓ Iteration의 사전적 의미는 되풀이, 반복
  - ✓ Iteration은 반복 처리를 나타내며 이를 위한 프로토콜(Protocol)을 갖고 있음
  - ✓ Iteration을 위한 규약이 있으며 이를 지켜야 반복 처리를 할 수 있음
  - ✓ 프로그래밍에서 반복 처리는 Array 오브젝트를 연상 할 수 있는데 Array(배열)를 반복하여 처리하기 위해서는 배열이 반복할 수 있는 오브젝트이어야 하고 오브젝트에 반복 처리를 할 수 있는 메서드가 있어야 하는데 이것이 Iteration 프로토콜(규약)
  - ✓ Iteration 프로토콜은 Iterable 프로토콜과 iterator 프로토콜로 구성

Iteration

- Iterable
  - ✓ 오브젝트의 반복 처리 규약을 정의한 프로토콜
  - ✓ Built In 객체인 String, Array, Map, Set, TypedArray, Argument 오브젝트 와 DOM 의 NodeList는 Iterable 프로토콜을 구현한 객체인데 자바스크립트 엔진이 랜더링 될 때 Iterable 프로토콜이 설정되기 때문에 사전 처리를 하지 않아도 반복 처리를 할 수 있음
  - ✓ Iterable 프로토콜이 설정된 오브젝트를 Iterable 오브젝트라고 부름
  - ✓ 자바스크립트는 Iterable 오브젝트에 Symbol.iterator가 있어야 하는데 이것이 Iterable 프로토콜
  - ✓ Symbol.iterator가 있으면 Iterable 오브젝트인데 자체 오브젝트에는 없지만 상속받은 prototype chain에 있어도 Iterable 오브젝트가 되는데 Built In Array 오브젝트를 상속받은 오브젝트는 Iterable 오브젝트가 됨
  - ✓ 프로토콜이 규약이므로 Iterable이 아닌 오브젝트에 프로토콜을 준수해서 개발자 코드를 추가하면 Iterable 오브젝트가 되는데 Iterable 오브젝트가 아닌 오브젝트에 Symbol.iterator를 개발자 코드로 추가하면 Iterable 프로토콜을 준수하게 되므로 Iterable 오브젝트가 됨
  - ✓ Symbol.iterator에 반복 처리할 수 있도록 코드를 작성하면 됨

Iteration

- Iterable
  - ✓ iteration_1.html

```
<script type="text/javascript">
    let arrayObj = [100, 200, 300];
    //iterable 프로토콜을 준수했는지 확인 - 준수함
    let result = arrayObj[Symbol.iterator];
    console.log(result);

    let mapObj = {"name":"adam"};
    //iterable 프로토콜을 준수했는지 확인 - 준수하지 않음
    result = mapObj[Symbol.iterator];
    console.log(result);
</script>
```

```
ƒ values() { [native code] }
undefined
```

Iteration

- Iterator
  - ✓ Iterator 프로토콜은 오브젝트의 값을 차례대로 처리할 수 있는 방법을 제공
  - ✓ 오브젝트에 있는 next() 메서드로 구현
  - ✓ 오브젝트에 next() 메서드가 있으면 Iterator 프로토콜이 적용된 것
  - ✓ 자바스크립트에서 {key: value} 형태의 Object 오브젝트는 작성한 순서대로 열거되는 것을 보장하지 않는데 이는 Object 오브젝트에 next()가 없다는 의미이기도 함
  - ✓ ES6 에서 Object 오브젝트에 추가한 순서대로 key, value가 열거되는 Map 오브젝트를 제공

Iteration

- Iterator
  - ✓ iteration_2.html

```
<script type="text/javascript">
    let arrayObj = [1, 2];
    let iteratorObj = arrayObj[Symbol.iterator]();

    console.log("1:", typeof iteratorObj);

    console.log("2:", iteratorObj.next());
    console.log("3:", iteratorObj.next());
    console.log("4:", iteratorObj.next());
</script>
```

Iteration

- Array-like
  - ✓ Array는 아니지만 Array처럼 사용할 수 있는 Object 오브젝트가 Array-like

    let values = {0: " zero " z 1: " one " , 2: " two " , length: 3};
  - ✓ 배열은 인덱스(Index)를 갖고 있기 때문에 작성된 순서대로 읽을 수 있으며 인덱스 번째의 엘리먼트 값을 수정, 삭제할 수 있으며 배열의 엘리먼트 수를 나타내는 length 프로퍼티가 있음
  - ✓ {key: value} 형태의 Object 오브젝트 특징을 유지하면서 배열의 특징을 가미한 것이 Array-like
  - ✓ {key: value} 형태에서 key에 0 부터 1씩 증가하면서 값을 부여
  - ✓ 배열의 인덱스 값을 프로퍼티 key 값으로 사용하는 것과 같은데 values["0"] 형태로 프로퍼티를 읽을 수 있음
  - ✓ 프로퍼티 값(value)에 배열의 엘리먼트에 해당하는 값을 작성하고 {length: 3}과 같이 Object 오브젝트의 전체 프로퍼티 수를 작성

Iteration

- Array-like
  - ✓ array_like.html

```
<script type="text/javascript">
    let values = {0: "zero", 1: "one", 2: "two", length: 3}

    for (var key in values){
        console.log(key, ':', values[key]);
    }

    for (var k = 0; k < values.length; k++){
        console.log(values[k]);
    }
</script>
```

```
0 : zero
1 : one
2 : two
length : 3
zero
one
two
```

Iteration

- for ~ of 문

  for(임시변수 of 컬렉션)

  - ✓ 반복 가능한 객체(Array, Map, Set, String, TypedArray, arguments 객체 등을 포함)에 대해서 반복하고 각 개별 속성값에 대해 실행되는 문이 있는 사용자 정의 반복 후크를 호출하는 루프를 생성
  - ✓ Iterable Object를 반복하여 처리하는 구문으로 for – in 과 유사
  - ✓ for - of 문과 for - in 문 모두 반복하는 것은 같지만 차이가 있는데 for-in 문의 대상은 Object 오브젝트이며 열거 가능한 프로퍼티가 대상이므로 프로퍼티의 enumerable 속성 값이 false이면 반복에서 제외되며 for-of 문의 대상은 iterable 오브젝트이며 prototype에 연결된 프로퍼티는 대상이 아님

Iteration

- for ~ of 문
  - ✓ forof.html

```
<script type="text/javascript">
        const array1 = ['a', 'b', 'c'];
        for (const element of array1) {
            console.log(element);
        }

        let iterable = "boo";
        for (let value of iterable) {
            console.log(value);
        }

        let hashMap = new Map([["a", 1], ["b", 2], ["c", 3]]);
        for (let entry of hashMap) {
            console.log(entry);
        }
</script>
```

```
a                          forof.html:10
b                          forof.html:10
c                          forof.html:10
b                          forof.html:15
2 o                        forof.html:15
▸ (2) ['a', 1]             forof.html:20
▸ (2) ['b', 2]             forof.html:20
▸ (2) ['c', 3]             forof.html:20
```

Destructuring

- Destructuring
  - ✓ 구조 분해 할당 또는 비구조화 할당이라고도 함
  - ✓ 배열이나 객체의 속성을 분해하여 그 값을 개별 변수에 담을 수 있게 하는 표현식

```
let one, two;

[one, two] = [1, 2]
```

  - one 변수에 1이 할당되고 two 변수에 2가 할당됨
  - 배열 [1, 2]를 엘리먼트로 분할하여 one과 two 변수에 할당하는데 ES6 스펙에서 이를 Destructuring Assignment로 표기하고 있음
  - one 변수에 배열의 1을 할당한다는 것보다 배열의 1을 one 변수에 할당한다는 표현이 더 정확한데 배열의 엘리먼트 값을 변수에 할당하기 위해서는 우선 배열의 엘리먼트를 분할해야 하며 분할된 엘리먼트 값을 변수에 할당하기 때문

Destructuring

- Destructuring
  - ✓ Array 분해 할당 – dest_1.html

```
<script type="text/javascript">
    let one, two, three, four, five;
    const values = [1, 2, 3];

    [one, two, three] = values;
    console.log("A:", one, two, three);

    [one, two] = values;
    console.log("B:", one, two);

    [one, two, three, four] = values;
    console.log("C:", one, two, three, four);

    [one, two, [three, four]] = [1, 2, [73, 74]];
    console.log("D:", one, two, three, four);
 </script>
```

A: 1 2 3
B: 1 2
C: 1 2 3 undefined
D: 1 2 73 74

Destructuring

- Destructuring
  - ✓ Object 분해 할당 – dest_2.html

```
<script type="text/javascript">
    //오브젝트를 분해해서 할당할 때는 변수 이름 과 속성이 같을 때  대입됨
    let {one, two} = {one: 1, nine: 9};
    console.log(one, two);

    //속성의 값을 직접 할당할 때는 이름이 같을 필요가 없음
    var obj1 = {
      title : '원본 글',
      reply : {
            title : '댓글'
      }
    };

    var title1 = obj1.title;
    var reply1 = obj1.reply.title;
    console.log(title1, reply1);
```

Destructuring

- Destructuring
  - ✓ Object 분해 할당 – dest_2.html

```
const obj2 = {
            title: '제목',
            reply: {
              replytitle: '댓글 제목'
            }
          }
  const { title, reply:{replytitle}, k } = obj2;
  console.log(title, replytitle, k); // '제목', '댓글 제목', undefined
</script>
```

1 undefined
원본 글 댓글
제목 댓글 제목 undefined

Destructuring

- Destructuring
  - ✓ Parameter 분해 할당 – dest_3.html

```
<script type="text/javascript">
    function total({one, plus: {two, five}}){
    console.log(one + two + five);
    };

    total({one: 1, plus: {two: 2, five: 5}});
</script>
```

8

Destructuring

- spread 연산자
  - ✓ Iterable Object 의 Element 를 하나씩 분리하여 전개 – 배열 과 객체
  - ✓ 전개한 결과를 변수에 할당하는 것이 가능
  - ✓ 구문
    - [...iterableObject]
    - function(...iterableObject)
  - ✓ spread 연산자는 ... 과 같이 점(.) 세 개를 연속해서 작성하고 이어서 Iterable Object를 작성하는데 [ ] 안에 작성하거나 { } 안에 작성하는 것이 가능
  - ✓ 문자열도 Iterable Object 이므로 글자 단위로 분리 가능

Destructuring

- spread 연산자
  - ✓ spread.html

```
<script type="text/javascript">
        //배열의 spread 연산
        let aespa = ["카리나", "지젤", "윈터", "닝닝"];
        console.log(aespa);
        let [leader, ...etc] = aespa;
        console.log(leader);
        console.log(etc);
```

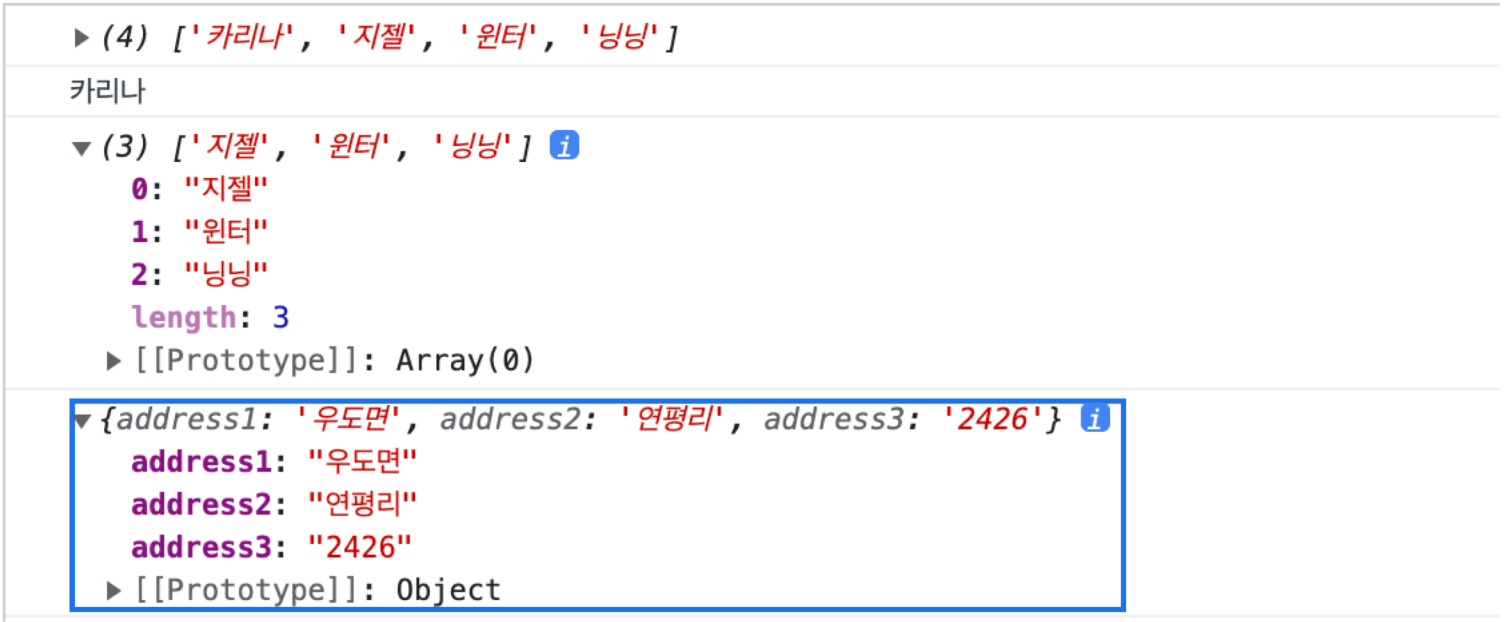
Destructuring

- spread 연산자
  - ✓ spread.html

```
        //객체의 spread 연산
        let address = {
            country: '제주특별자치도',
            city: '제주',
            address1: '우도면',
            address2: '연평리',
            address3: '2426'
        }
        const {
            country,
            city,
            ...detail
        } = address
        console.log(detail)
    </script>
```

Destructuring

- spread 연산자
  - ✓ spread.html



Destructuring

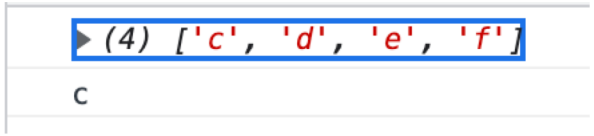
- Rest Parameter
  - ✓ 함수의 매개변수를 spread 연산자를 이용해서 작성한 형태
  - ✓ parameter앞에 ...을 추가해서 설정

    function(param, paramN, ...rest);
  - ✓ rest parameter는 항상 제일 마지막 parameter로 있어야 함

Destructuring

- Rest Parameter
  - ✓ restparam.html

```
<script>
    function func1 (a, b, ...c) {
        console.log(c);
    }
    function func2 (a, b, c) {
        console.log(c);
    }
    func1('a', 'b', 'c', 'd', 'e', 'f');
    func2('a', 'b', 'c', 'd', 'e', 'f');
</script>
```



Destructuring

- Default Value
  - ✓ 함수에 파라미터 값을 대입하지 않았을 때 설정되는 값
  - ✓ default_value.html

```
<script type="text/javascript">
    let [one, two, five = 5] = [1, 2];
    console.log(five);

    [one, two, five = 5] = [1, 2, 77];
    console.log(five);

    let {six=6, seven} = {seven: 7};
    console.log(six, seven);

    [one, two = one + 1, five = two + 3] = [1];
    console.log(one, two, five);
 </script>
```

```
5
77
6 7
1 2 5
```

Exception Handling

- 예외 처리(Exception Handling)
  - ✓ 프로그램 실행 중에 오류가 발생해서 프로그램이 중단되는 경우를 중단되지 않도록 하기 위한 처리하는 것
  - ✓ 예외 처리 기본 형식

```
try{
    //실행 코드;
}
catch(예외처리 변수){
  //예외가 발생했을 때 처리할 내용;
}
finally{
    무조건 수행되는 코드;
}
```

  - finally 나 catch 둘 중 하나는 생략할 수 있음

Exception Handling

- 예외 객체의 멤버
  - ✓ message: 예외 메시지
  - ✓ description: 예외 설명
  - ✓ name: 예외 이름

Exception Handling

- 예외 처리
  - ✓ try1.html – 예외 처리를 하지 않음

```
<script>
    var array = new Array(4294967296);
    // 배열 maxlength(4,294,967,295) 보다 크게 생성하기 때문에 에러 발생
    //이 문장은 수행되지 않음
    alert("메시지");
</script>
```

Exception Handling

- 예외 처리
  - ✓ try1.html – 예외 처리 수행

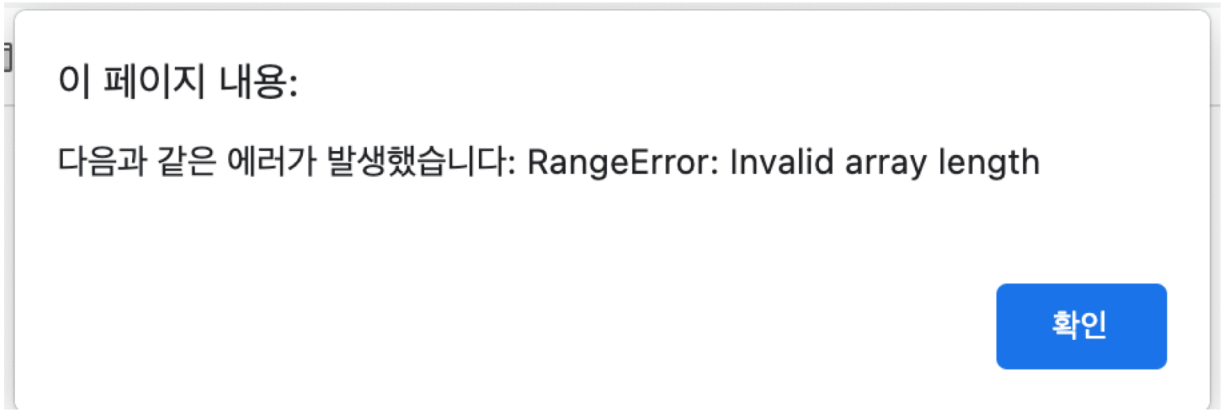
```
<script>
    try{
        var array = new Array(4294967296);
        // 배열 maxlength(4,294,967,295) 보다 크게 생성하기 때문에 에러 발생
    }catch{

    }
    //이 문장이 수행됨
    alert("메시지");
</script>
```

Exception Handling

- 예외 처리
  - ✓ try2.html – 예외 객체의 멤버 사용

```
<script>
    try {
      console.log('에러가 나기 직전까지의 코드는 잘 실행됩니다.');
      new Array(-1); // RangeError: Invalid array length
      console.log('에러가 난 이후의 코드는 실행되지 않습니다.');
    } catch (e) {
      console.log('코드의 실행 흐름이 catch 블록으로 옮겨집니다.');
      alert(`다음과 같은 에러가 발생했습니다: ${e.name}: ${e.message}`);
    }
</script>
```



Exception Handling

- 예외 처리
  - ✓ try3.html – finally 사용

```
<script>
    try {
        console.log('try');
        new Array(-1); // RangeError: Invalid array length
    } catch (e) {
        console.log('catch');
    } finally {
        console.log('finally');
    }
</script>
```

Exception Handling

- 예외 처리
  - ✓ try ~ catch 문에서 try 블록 { } 기준으로 블록 스코프를 가짐
    - let_try.html

```
<script type="text/javascript">
    let sports = "축구";
    try {
        let sports = "농구";
        console.log(sports);
    } catch (e) {

    };
    console.log(sports);
</script>
```

Exception Handling

- ❖ 예외 처리
  - ✓ 예외 강제 발생: throw 에러객체
    - ● throw.html

```
<script>
    function sum(x, y) {
        if (typeof x !== 'number' || typeof y !== 'number') {
            throw "숫자를 입력하세요"
        }
        return x + y;
    }

    console.log(sum("abc", 1))
</script>
```

Exception Handling

- ❖ 예외 처리
  - ✓ 예외 강제 발생: throw 에러객체
    - throw.html

```
<script>
    function sum(x, y) {
        if (typeof x !== 'number' || typeof y !== 'number') {
            throw new Error("숫자를 입력하세요");
        }
        return x + y;
    }

    console.log(sum("abc", 1))
</script>
```

Module Programming

- Module
  - ✓ 개발하는 애플리케이션의 크기가 커지면 파일을 여러 개로 분리해야 하는 시점이 오게 되는데 이때 분리된 파일 각각을 모듈(module) 이라고 부르는데 모듈은 대개 클래스 하나 혹은 특정한 목적을 가진 복수의 함수로 구성된 라이브러리 하나로 구성됨
  - ✓ export
    - ✓ 파일에 작성된 데이터를 다른 파일에서 사용할 수 있도록 내보내는 기능
    - ✓ 기본적으로는 export 데이터 형태로 작성
    - ✓ export default 라는 특별한 문법을 지원하는데 export default를 사용하면 해당 모듈에는 개체가 하나만 있다 는 사실을 명확히 나타낼 수 있는데 내보내고자 하는 객체 앞에 export default를 붙이면 되는데 하나의 모듈에서는 한 번 만 사용해야 함

Module Programming

- Module
  - ✓ export

```
// 하나씩 내보내기
export let name1, name2, …, nameN; // var, const도 동일
export let name1 = …, name2 = …, …, nameN; // var, const도 동일
export function functionName(){...}
export class ClassName {...}

// 목록으로 내보내기
export { name1, name2, …, nameN };

// 내보내면서 이름 바꾸기
export { variable1 as name1, variable2 as name2, …, nameN };

// 비구조화로 내보내기
export const { name1, name2: bar } = o;

// 기본 내보내기
export default expression;
export default function (…) { … } // also class, function*
export default function name1(…) { … } // also class, function*
export { name1 as default, … };
```

Module Programming

- Module
  - ✓ export

```
// 모듈 조합
export * from ...; // does not set the default export
export * as name1 from ...;
export { name1, name2, ..., nameN } from ...;
export { import1 as name1, import2 as name2, ..., nameN } from ...;
export { default } from ...;
```

Module Programming

- Module
  - ✓ import
    - 다른 모듈에서 export 된 데이터를 받아오는 기능

```
import name from "module-name";
//export default로 export한 멤버를 name에 받음.

import * as name from "module-name";
// export되는 모든 멤버를 name에 받음.

import { member } from "module-name";
// export된 멤버 member를 member로 받음.

import { member as alias } from "module-name";
// export된 멤버 member를 alias로 받음.

import "module-name";
// import만 하면 되는 경우 ex) import "main.css";
```

Module Programming

- Module
  - ✓ export.js

```
function cube(x) {
    return x * x * x;
}

const foo = Math.PI + Math.SQRT2;

var graph = {
    options: {
        color:'white',
        thickness:'2px'
    },
    draw: function() {
        console.log('From graph draw function');
    }
}

export { cube, foo, graph };
```

Module Programming

- Module
  - ✓ import.js

```
import { cube, foo, graph } from './export.js';

graph.options = {
    color:'blue',
    thickness:'3px'
};

graph.draw();
console.log(cube(3)); // 27
console.log(foo);     // 4.555806215962888
```

Module Programming

- Module
  - ✓ import.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Generator</title>
<script type="module" src="./import.js">

</script>
</head>

<body>
</body>
</html>
```