

# JavaScript

Collection – 김근형 강사

# Iterator

- Iterator

- Iterator는 반복 처리를 나타내며 이를 위한 프로토콜(Protocol)을 갖고 있다.
- 이터러블 프로토콜은 오브젝트의 반복 처리 규약을 정의한다.
- 빌트인 String, Array, Map, Set, TypedArray, Argument 오브젝트는 디폴트로 이터러블 프로토콜을 갖고 있다.
- 그래서 위의 오브젝트들은 사전 처리를 하지 않아도 반복 처리를 할 수 있으며 이러한 오브젝트들을 이터러블 오브젝트라고 한다.

# Iterator

- Iterator

- 오브젝트나 배열에서의 이터레이터의 존재 검증은 다음과 같이 할 수 있다.

```
let arrayObj = [];  
let result = arrayObj[Symbol.iterator];  
console.log(result);
```

```
f values() { [native code] }
```

```
let objectObj = {};  
let result = objectObj[Symbol.iterator];  
console.log(result);
```

```
undefined
```

# Iterator

- Iterator protocol

- 오브젝트의 값을 차례대로 처리하기 위해서는 next() 메서드를 이용한다.

```
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());
```

```
1: object  
2: ▼Object i  
   done: false  
   value: 1  
   ▶ __proto__: Object  
3: ▼Object i  
   done: false  
   value: 2  
   ▶ __proto__: Object  
4: ▼Object i  
   done: true  
   value: undefined  
   ▶ __proto__: Object
```

# Iterator

- 사용자 지정 Iterator
  - 사용자가 직접 Iterator 함수를 만들어 사용할 수 있다.
  - 그러기 위해서는 [Symbol.iterator]의 호출을 통해 만들어야만 한다.
  - Iterator는 for ~ of와 궁합이 잘 맞는다.

```
const iterable = {
  [Symbol.iterator]() {
    let i = 3;
    return {
      next() {
        return i == 0 ? {done: true} : {value: i--, done: false};
      },
      [Symbol.iterator]() {
        return this;
      }
    }
  }
};

let iterator = iterable[Symbol.iterator]();
console.log(iterator.next()); // 3
console.log(iterator.next()); // 2
for (const a of iterator) console.log(a); // 1
```

▶ {value: 3, done: false}

▶ {value: 2, done: false}

1

# Iterator

## ○ DOM과 iterator를 이용한 예제

```
<h1>안녕하세요</h1>
<table>
  <tr>
    <td></td>
    <td></td>
  </tr>
</table>
<ul>
  <li></li>
  <li></li>
  <li></li>
</ul>
```

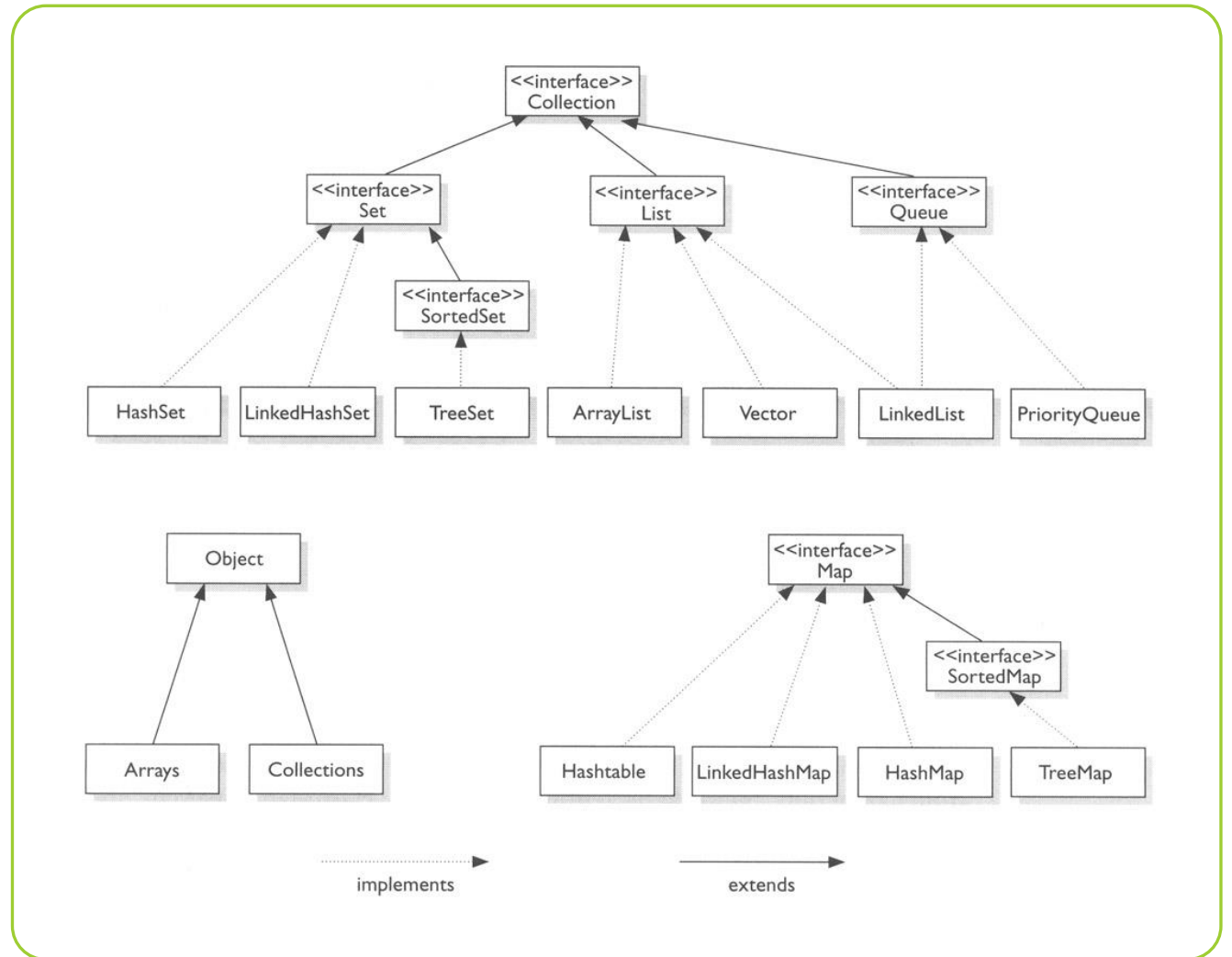
```
for (const a of document.querySelectorAll('*')) console.log(a);
const all = document.querySelectorAll('*');
let iter3 = all[Symbol.iterator]();
console.log(iter3.next());
console.log(iter3.next());
console.log(iter3.next());
```

```
<html lang="en">
  ><head>...</head>
  ><body>...</body>
</html>

><head>...</head>
  <meta charset="UTF-8">
  <title>Document</title>
><body>...</body>
  <h1>안녕하세요</h1>
><table>...</table>
><tbody>...</tbody>
><tr>...</tr>
  <td></td>
  <td></td>
><ul>...</ul>
  <li></li>
  <li></li>
  <li></li>
><script type="text/javascript">...</script>
>{value: html, done: false}
>{value: head, done: false}
>{value: meta, done: false}
```

# Collection

## ○ Collections



# Collection

- Collection
  - 컬렉션(Collection)이란 데이터를 담을 수 있는 일종의 집합을 의미한다
  - JavaScript에서는 데이터를 담기 위해 Array나 Object와 같은 구조를 사용했지만 사용에 기능이 한정적이었다.
  - 또한 이번에 새로 나온 컬렉션과 같은 구조를 구현하기 위해서 추가적인 개발이 들어가는 번거로운 일이 생겼었다.
  - ES2015에서는 위와같은 문제를 해결하기 위해 컬렉션이 등장했으며 기존 Array와 Object 외에 Map/WeakMap/Set/WeakSet/TypedArray가 등장했다.



# Collection

- JavaScript에서는 다음과 같은 컬렉션들이 존재한다.
  - Indexed Collection - Arrays, Typed Array
  - Keyed Collection - Objects, Map, Set, Weak Map, Weak Set
- Indexed Collection은 인덱스를 통해 해당 컬렉션의 특정 부분으로 접근할 수 있는 컬렉션이다.
- Keyed Collection은 해당 컬렉션에서 지정한 키를 통해 특정 부분으로 접근할 수 있는 컬렉션이다.

# Set

- Set
  - Set은 value를 키 값으로 갖는 컬렉션이다.
  - Set은 수정 가능하며, 프로그램이 실행되는 동안 값의 추가나 삭제를 할 수 있다.
  - 여기까지는 Array와 같지만 Set과 Array에는 차이점이 존재한다.

# Set

## ○ Set 특징

- Set 오브젝트에 추가한 순서로 인덱스를 부여하기 때문에 추가한 순서대로 읽히는 것을 보장한다.
- Set 오브젝트는 key 개념을 갖고 있으며 value 자체가 키 값이 된다.
- Set은 동일한 value값이 존재하지 않으며 동일한 값을 넣을 경우 무시된다.
- value에는 string, number, symbol 등의 프리미티브 데이터 타입과 Object, function과 같은 오브젝트 작성도 가능하다.

# Set

- new Set()
  - Set 인스턴스를 생성하여 반환한다. 배열이나 String 안에 같은 값이 있을 경우 무시된다.

구분	타입	데이터(값)
형태		new Set()
파라미터	any	(선택), 값
반환	Set	생성한 Set 인스턴스

```
const setObj = new Set();

const newSet = new Set([1, 2, 1, 2, "스포츠"]);
console.log(newSet.size);

for (let element of newSet){
  console.log(element);
};
```

```
3
1
2
스포츠
```

# Set

- add()

- Set 인스턴스 끝에 value 값을 추가한다. 동일한 값을 넣을 경우 무시된다.

구분	타입	데이터(값)
형태		Set.prototype.add()
파라미터	any	value
반환	Set	Value가 추가된 Set 인스턴스

```
const newSet = new Set();
```

축구

```
newSet.add("축구").add("농구");
```

농구

```
newSet.add("축구");
```

```
for (let element of newSet) {  
  console.log(element);  
};
```

```
const newSet = new Set();
```

```
let music = () => {};
```

() => {}

```
newSet.add(music);
```

```
newSet.add(music);
```

```
for (let element of newSet) {  
  console.log(element);  
};
```

# Set

- has()

- Set 인스턴스에서 value의 존재 여부를 반환한다.

구분	타입	데이터(값)
형태		Set.prototype.has()
파라미터	any	value
반환	boolean	Value가 존재하면 true, 아니면 false

```
const newSet = new Set();  
newSet.add("sports");  
  
console.log(newSet.has("sports"));
```

true

# Set

## ○ entries()

- 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Set.prototype.entries()
파라미터		없음
반환	Iterator	Value를 반환하는 이터레이터 오브젝트

```
const newSet = new Set(["one", () => {}]);  
let iteratorObj = newSet.entries();  
  
console.log(iteratorObj.next());  
console.log(iteratorObj.next());
```

```
▼ Object ⓘ  
  done: false  
  ▼ value: Array(2)  
    0: "one"  
    1: "one"  
    length: 2  
    ▶ __proto__: Array(0)  
  ▶ __proto__: Object
```

```
▼ Object ⓘ  
  done: false  
  ▼ value: Array(2)  
    ▶ 0: () => {}  
    ▶ 1: () => {}  
    length: 2  
    ▶ __proto__: Array(0)  
  ▶ __proto__: Object
```

# Set

- values()

- value 값을 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Set.prototype.values()
파라미터		없음
반환	Iterator	Value를 반환하는 이터레이터 오브젝트

```
const newSet = new Set(["one", () => {}]);
let iteratorObj = newSet.values();

console.log(iteratorObj.next());
console.log(iteratorObj.next());
```

```
▼ Object i
  done: false
  value: "one"
  ▶ __proto__: Object
```

```
▼ Object i
  done: false
  ▶ value: () => {}
  ▶ __proto__: Object
```



# Set

## ○ keys()

- key 값을 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Set.prototype.keys()
파라미터		없음
반환	Iterator	Key 값을 반환하는 이터레이터 오브젝트

```
const newSet = new Set(["one", () => {}]);  
let iteratorObj = newSet.keys();  
  
console.log(iteratorObj.next());  
console.log(iteratorObj.next());
```

```
▼ Object 1  
  done: false  
  value: "one"  
  ▶ __proto__: Object  
  
▼ Object 1  
  done: false  
  ▶ value: () => {}  
  ▶ __proto__: Object
```

# Set

- Set의 key, value, entries 에 대한 내용
  - Set은 key가 곧 value이기 때문에 entries를 통해 출력하게 되면 key값과 value가 동일하게 출력이 된다.
  - 그래서 Set을 통해서 entries를 통해 출력하기 보단 key와 value를 통해 출력하는 것이 좋다.
  - key와 value는 값이 동일하므로 어떤 형태로 출력하든 관계가 없다.

# Set

- symbol.iterator

- Set의 Symbol.iterator를 통해 위의 이터레이터 호출이 가능하다.

구분	타입	데이터(값)
형태		Set[Symbol.iterator]()
파라미터		없음
반환	Iterator	이터레이터 오브젝트

```
const newSet = new Set([1, "스포츠"]);  
let iteratorObj = newSet[Symbol.iterator]();  
  
console.log(iteratorObj.next());  
console.log(iteratorObj.next());
```

```
▼ {value: 1, done: false} ⓘ  
  value: 1  
  done: false  
  ▶ __proto__: Object  
▼ {value: "스포츠", done: false} ⓘ  
  value: "스포츠"  
  done: false  
  ▶ __proto__: Object
```

# Set

- forEach()
  - Set 인스턴스에 작성된 순서로 반복하면서 콜백 함수를 호출한다.

구분	타입	데이터(값)
형태		Set.prototype.forEach()
파라미터	Function	반복할 때 마다 호출할 callback 함수
	Object	(선택) callback 함수에서 this로 참조할 오브젝트
반환		Undefined

```
const newSet = new Set(["one", "two"]);

newSet.forEach(function(value, key, obj) {
  console.log(value, this.member);
}, {member: 10});
```

```
one 10
two 10
```

# Set

- delete()
  - Set 인스턴스에서 value 값이 같은 엘리먼트를 삭제한다

구분	타입	데이터(값)
형태		Set.prototype.delete()
파라미터	any	Value, 삭제할 value
반환	boolean	삭제 성공 true, 아닐 시 false

```
const newSet = new Set(["one"]);  
  
console.log(newSet.delete("one"));  
console.log(newSet.size);
```

true

0

# Set

- clear()

- Set 인스턴스의 모든 value를 지운다

구분	타입	데이터(값)
형태		Set.prototype.clear()
파라미터		없음
반환		삭제 성공 true, 아닐 시 false

```
const newSet = new Set(["one"]);  
  
console.log(newSet.delete("one"));  
console.log(newSet.size);
```

```
true
```

```
0
```

# Map

- Map
  - Map Object 는 key와 value로 구성된다.
  - Object 오브젝트의 key 타입이 String 또는 Symbol 이지만 Map 오브젝트는 String, Symbol 이외에도 Object, Function 과 같이 오브젝트를 사용할 수 있다.
  - Map 오브젝트가 key, value 로 구성되지만 {key:value} 형태로 작성되지 않고 {"key","value"} 와 같이 이터러블 형태로 작성한다.
  - Map 오브젝트는 key 값이 같으면 추가되지 않고 value 값이 대체되며 추가한 순서로 읽힌다.(엔진에서 key, value 외에 별도로 일련번호를 부여하기 때문)
  - Map의 메서드는 Set과 동일하며 구현되는 메서드의 결과가 차이가 있다.

# Map

- Map의 등장 배경
  - 객체의 키로 내장 메소드의 이름을 사용할 시 이름 충돌이 일어날 수 있다.
  - 속성의 Key는 항상 문자열 이다. (ES6 에서는 심볼도 가능하다.)
  - 객체에 얼마나 많은 속성이 존재하는지 알아낼 수 있는 효과적인 방법이 없다.(size나 length 같은 메서드가 없다.)
  - 일반 객체를 반복하려면 많은 비용이 소모된다.



# Map

- new Map()
  - Map 인스턴스를 생성하여 반환한다.

구분	타입	데이터(값)
형태		new Map()
파라미터	Array	(선택) 이터러블 오브젝트
반환	Map	생성한 Map 인스턴스

# Map

○ new Map()

```
let emptyMap = new Map();
console.log(emptyMap);
let newMap = new Map([
  ["key1", "value1"],
  ["key2", "value2"],
  ["key1", "sports"]
]);
console.log(newMap);
for (var element of newMap){
  console.log(element);
};
```

```
▼ Map(0) ⓘ
  size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array(0)
    length: 0

▼ Map(2) ⓘ
  size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array(2)
    ▼ 0: {"key1" => "sports"}
      key: "key1"
      value: "sports"
    ▼ 1: {"key2" => "value2"}
      key: "key2"
      value: "value2"
    length: 2

▼ Array(2) ⓘ
  0: "key1"
  1: "sports"
  length: 2
  ▶ __proto__: Array(0)

▼ Array(2) ⓘ
  0: "key2"
  1: "value2"
  length: 2
  ▶ __proto__: Array(0)
```

# Map

## ○ new Map()

```
let newMap = new Map([
  ["key1", "value1"],
  ["key2", "value2"]
]);

for (var element of newMap){
  element.forEach((keyValue, index) => {
    console.log(index, keyValue);
  });
};

for (var [key, value] of newMap){
  console.log(key, value);
};
```

0	"key1"
1	"value1"
0	"key2"
1	"value2"
key1 value1	
key2 value2	

# Map

## ○ new Map() : 잘못된 예

```
try {
  new Map(["one", 1]);
}catch(e){
  console.log("[one, 1]");
};

try {
  new Map({one: 1});
}catch(e){
  console.log("{one: 1}");
};

let newMap = new Map([{one: 1}]);
console.log(newMap);
```

```
[one, 1]
{one: 1}
▼ Map(1) ⓘ
  size: (...)
  ▶ __proto__: Map
  ▼ [[Entries]]: Array(1)
    ▶ 0: {undefined => undefined}
    length: 1
```

# Map

- set()
  - Map 인스턴스에 key와 value를 설정한다

구분	타입	데이터(값)
형태		Map.prototype.set()
파라미터	any	Key
	any	value
반환	Map	Key, value 가 추가된 인스턴스

# Map

## ○ set()

```
const newMap = new Map();
newMap.set("one", 100);
console.log(newMap.size);

newMap.set({}, "오브젝트");
newMap.set(function() {}, "Function");

newMap.set(new Number("123"), "인스턴스");
newMap.set(NaN, "Not a Number");

for (var [key, value] of newMap) {
  console.log(key, value);
};
```

```
1
one 100
▼ Object ⓘ "오브젝트"
  ► __proto__: Object
f () {} "Function"
▼ Number ⓘ "인스턴스"
  ► __proto__: Number
    [[PrimitiveValue]]: 123
NaN "Not a Number"
```

# Map

## ○ set()

```
const newMap = new Map();
newMap.set("one", 100);
newMap.set("one", 123);

for (var [key, value] of newMap) {
  console.log(key, value);
};
console.log("-----");
let sportsObj = {sports: "스포츠"};
newMap.set(sportsObj, "Sports Object");
newMap.set(sportsObj, "Sports Object-변경");

for (var [key, value] of newMap) {
  console.log(key, value);
};
console.log("-----");
newMap.set({}, "Object-1");
newMap.set({}, "Object-2");

for (var [key, value] of newMap) {
  console.log(key, value);
};
```

```
one 123
-----
one 123
▶ {sports: "스포츠"} "Sports Object-변경"
-----
one 123
▶ {sports: "스포츠"} "Sports Object-변경"
▶ {} "Object-1"
▶ {} "Object-2"
```

# Map

- get()
  - Map 인스턴스에서 key값이 같은 value를 반환한다

구분	타입	데이터(값)
형태		Map.prototype.get()
파라미터	any	key
반환	any	[key, value]에서 value, key가 존재하지 않으면 undefined



# Map

## ○ get()

```
const newMap = new Map();
newMap.set("one", 100);
console.log(newMap.get("one"));
console.log(newMap.get("two"));

let sportsObj = {sports: "스포츠"};
newMap.set(sportsObj, "Sports Object");
console.log(newMap.get(sportsObj));
```

100

undefined

Sports Object

# Map

## ○ get()

```
const newMap = new Map();

newMap.set({}, "Object");
console.log(newMap.get({}));

newMap.set(123, "값 123");
console.log(newMap.get(123));

console.log(newMap.get("123"));

newMap.set(NaN, "Not a Number");
console.log(newMap.get(NaN));
```

undefined
값 123
undefined
Not a Number

# Map

- has()

- Map 인스턴스에서 key의 존재 여부를 반환한다.

구분	타입	데이터(값)
형태		Map.prototype.has()
파라미터	any	Key
반환	Boolean	Key가 존재하면 true, 아니면 false

```
const newMap = new Map();  
newMap.set("one", 100);  
  
console.log(newMap.has("one"));
```

true

# Map

- entries()

- [key, value]를 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Map.prototype.entries()
파라미터		없음
반환	Iterator	[key, value]를 반환하는 이터레이터 오브젝트

# Map

## ○ entries()

```
const newMap = new Map([
  ["key1", "value1"],
  ["key2", "value2"]
]);

let iteratorObj = newMap.entries();

let result = iteratorObj.next();
console.log(result);

console.log(iteratorObj.next());
console.log(iteratorObj.next());
```

```
▼ Object ⓘ
  done: false
  ▶ value: (2) ["key1", "value1"]
  ▶ __proto__: Object

▼ Object ⓘ
  done: false
  ▶ value: (2) ["key2", "value2"]
  ▶ __proto__: Object

▼ Object ⓘ
  done: true
  value: undefined
  ▶ __proto__: Object
```

# Map

- keys()

- key 값을 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Map.prototype.keys()
파라미터		없음
반환	Iterator	Key 값을 반환하는 이터레이터 오브젝트

```
const newMap = new Map([
  ["key1", "value1"]
]);
newMap.set({}, "오브젝트");

let iteratorObj = newMap.keys();
console.log(iteratorObj.next());
console.log(iteratorObj.next());
```

```
▼ Object i
  done: false
  value: "key1"
  ▶ __proto__: Object

▼ Object i
  done: false
  ▶ value: {}
  ▶ __proto__: Object
```

# Map

- values()

- value를 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Map.prototype.values()
파라미터		없음
반환	Iterator	value를 반환하는 이터레이터 오브젝트

```
const newMap = new Map([
  ["key1", "value1"]
]);
newMap.set({}, "오브젝트");

let iteratorObj = newMap.values();
console.log(iteratorObj.next());
console.log(iteratorObj.next());
```

```
▼ Object ⓘ
  done: false
  value: "value1"
  ▶ __proto__: Object

▼ Object ⓘ
  done: false
  value: "오브젝트"
  ▶ __proto__: Object
```

# Map

- Symbol.iterator

- [key, value]를 반환하는 이터레이터 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Map[Symbol.iterator]()
파라미터		없음
반환	Iterator	[key, value]를 반환하는 이터레이터 오브젝트

```
let newMap = new Map([
  ["1", "music"],
  ["2", "sports"]
]);

let iteratorObj = newMap[Symbol.iterator]();

console.log(iteratorObj.next());
console.log(iteratorObj.next());
```

```
▼ {value: Array(2), done: false} ⓘ
  ► value: (2) ["1", "music"]
  done: false
  ► __proto__: Object

▼ {value: Array(2), done: false} ⓘ
  ► value: (2) ["2", "sports"]
  done: false
  ► __proto__: Object
```



# Map

- forEach()
  - Map 인스턴스를 반복할 때마다 callback 함수를 호출한다.

구분	타입	데이터(값)
형태		Map.prototype.forEach()
파라미터	Function	반복할 때마다 호출할 callback 함수
	Object	(선택) callback 함수에서 this로 참조할 오브젝트
반환		undefined

- 실행할 때 마다 세 개의 파라미터를 넘겨주는데 첫번째 파라미터가 value고 두번째 파라미터가 key 이며 세번째 파라미터는 실행중인 map 인스턴스이다.

# Map

## ○ forEach()

```
const newMap = new Map([
  ["key1", "value1"],
  [{}, "오브젝트"]
]);

newMap.forEach((value, key, map) => {
  console.log(key, value);
});
```

key1 value1

▼ Object ⓘ "오브젝트"  
 ► \_\_proto\_\_: Object

# Map

- delete()
  - Map 인스턴스에서 key 값이 같은 엘리먼트를 삭제한다

구분	타입	데이터(값)
형태		Map.prototype.delete()
파라미터	Any	Key, 삭제할 키
반환	boolean	삭제 성공 : true, 삭제 실패 : false

# Map

## ○ delete()

```
const newMap = new Map([
  ["key1", "value1"],
  [{}, "오브젝트"]
]);
let sportsObj = {};
newMap.set(sportsObj, "추가");

console.log(newMap.delete("key1"));
console.log(newMap.delete({}));
console.log(newMap.delete(sportsObj));
```

true

false

true

# Map

- clear()
  - Map 인스턴스의 모든 [key, value]를 지운다

구분	타입	데이터(값)
형태		Map.prototype.clear()
파라미터		없음
반환		없음

```
const newMap = new Map([
  ["key1", "value1"],
  [{}, "오브젝트"]
]);

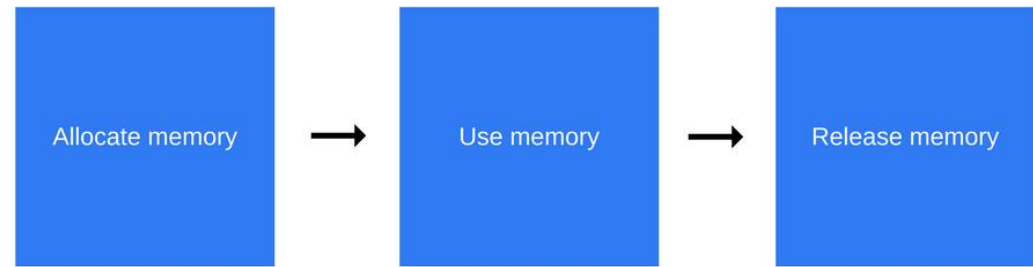
console.log(newMap.size);
newMap.clear();
console.log(newMap.size);
```

2

0

# GC(Garbage Collector)

- 메모리 생명 주기



# GC(Garbage Collector)

## ○ 메모리 생명주기

- 메모리 할당(allocate): 프로그램이 사용할 수 있도록 운영체제가 메모리를 할당한다. 저수준 언어(예를 들어 C)에서는 이를 개발자가 명시적으로 처리해줘야 한다. 그러나 고수준 언어에서는 개발자가 신경 쓸 필요가 없다.
- 메모리 사용(use): 할당된 메모리를 실제로 프로그램이 사용한다. 개발자가 코드 상에서 할당된 변수를 사용함으로써 읽기와 쓰기 작업이 이루어진다.
- 메모리 해제(release): 프로그램에서 필요하지 않은 메모리 전체를 되돌려주어 다시 사용 가능하게 만드는 단계이다. 메모리 할당 작업과 마찬가지로 저수준 언어에서는 이를 명시적으로 처리해야 한다.

# GC(Garbage Collector)

- GC(Garbage Collector)
  - 메모리 해제 시 대부분의 메모리 관리 문제가 일어난다.
  - 여기서 가장 어려운 작업은 할당된 메모리가 언제 더 이상 필요 없는지 알아내는 것이다. 프로그램 상에서 일정한 메모리가 더 이상 사용되지 않고 있으며 이를 반환해야 하는지를 프로그래머가 판단해야 할 때도 있다.
  - 고수준의 언어에는 GC(garbage collector)라는 소프트웨어가 내장되어 있는데 이것의 역할은 메모리 할당을 추적하고 언제 할당된 메모리가 더 이상 사용되지 않는지 파악해서 자동으로 반환한다.
  - 불행하게도 이러한 과정은 추정에 기반하는데 왜냐하면 일반적으로 메모리의 일부가 필요할지를 알아내는 문제는 결정불가능(undecidable), 즉 알고리즘으로 풀 수 없는 문제이기 때문이다.
  - 대부분의 GC는 어떤 메모리를 가리키는 모든 변수가 스코프를 벗어나게 됐을 때 처럼 더 이상 접근 불가능한 메모리를 수집한다. 이는 메모리 공간에서 수집할 수 있는 것들에 대해 보수적인 입장에서 추측하는 것이인데 왜냐하면 어떤 메모리 영역을 가리키는 변수는 아직 존재하지만 그것을 그 이후로 절대로 접근하지 않는 경우는 언제라도 생길 수 있기 때문이다.



# GC(Garbage Collector)

- GC와 Collection
  - 기존의 Collection은 데이터를 저장하고 접근하고 열거한다는 점에서는 최적의 저장 공간이다.
  - 하지만 데이터를 잘못 삭제하거나 수정하게 되면 불필요한 공간이 남게 되고 이 과정에서 메모리 누수가 발생한다.
  - 나중에는 웹에 상당한 부하를 가져오게 하는 주범이 된다.

# WeakMap, WeakSet

- WeakMap, WeakSet
  - WeakSet과 WeakMap은 기존의 Set과 Map과는 달리 [약한 참조]를 지원한다
  - 즉, 객체키에 대한 참조가 존재하지 않을 경우 언제든지 GC 대상이 된다.
  - 또한 위의 두 객체는 열거형(iterator) 형태를 지원하지 않는다
  - 필요한 엔트리를 분명하게 명시하지 않으면 WeakSet, WeakMap 안의 엔트리를 얻을 수 없다. 필요한 엔트리를 얻으려면 필요한 `key`를 전달해야 한다. 이러한 제한은 보안 프로퍼티를 가능케 한다.
  - 또한 약한 참조를 통해 메모리 관리를 좀 더 원활하게 한다는 장점이 있다.
  - 단 기능이 제한적이고 활용이 어려워 쓰기 힘들다.

# WeakSet

- WeakSet Object
  - Set의 약한 버전 타입
  - WeakSet 오브젝트는 value 오브젝트만 사용할 수 있으며, string, number, symbol과 같은 값을 사용할 수 없다.
  - key는 사용하지 않는다.

# WeakSet

- new WeakSet()
  - WeakSet 인스턴스를 생성하여 반환한다.

구분	타입	데이터(값)
형태		new WeakSet()
파라미터	object	(선택), object, 이터러블 오브젝트 안에 작성
반환	WeakSet	생성한 WeakSet 인스턴스

```
let newString = new String("문자열");
let newNumber = new Number(12345);
const newWeakSet = new WeakSet([newString, newNumber]);

try {
  new WeakSet(["ABC", 345]);
} catch (e) {
  console.log("object 이외 지정 불가");
};
```

object 이외 지정 불가

# WeakSet

- add()
  - WeakSet 인스턴스에 value를 추가한다.

구분	타입	데이터(값)
형태		WeakSet.prototype.add()
파라미터	object	Value, Object/Function 등
반환	WeakSet	Value가 추가된 WeakSet 인스턴스

```
const newWeakSet = new WeakSet();

let newString = new String("문자열");
newWeakSet.add(newString);

let obj = {sports: "스포츠"};
newWeakSet.add(obj);
console.log(newWeakSet);
```

```
▼ WeakSet 1
  ▶ __proto__: WeakSet
  ▼ [[Entries]]: Array(2)
    ▶ 0: String
    ▶ 1: Object
    length: 2
```

# WeakSet

- has()
  - WeakSet 인스턴스에서 value의 존재 여부를 반환한다.

구분	타입	데이터(값)
형태		WeakSet.prototype.has()
파라미터	object	Value, 오브젝트
반환	boolean	존재하면 true, 아니면 false

```
let newString = new String("문자열");  
const newWeakSet = new WeakSet([newString]);  
  
console.log(newWeakSet.has(newString));
```

true

# WeakSet

- delete()
  - WeakSet 인스턴스에서 value가 같은 엘리먼트를 삭제한다.

구분	타입	데이터(값)
형태		WeakSet.prototype.delete()
파라미터	object	Value, 오브젝트
반환	boolean	삭제 성공하면 true, 아니면 false

```
let newString = new String("문자열");
const newWeakSet = new WeakSet([newString]);

console.log(newWeakSet.delete(newString));
console.log(newWeakSet.has(newString));
```

true

false

# WeakMap

## ○ WeakMap Object

- WeakMap 오브젝트의 key에 오브젝트만 지정할 수 있으며 string, number, symbol과 같은 값을 작성할 수 없다. 하지만 value는 타입에 제한이 없다.
- WeakMap 오브젝트에서 제공하는 메서드는 set(), get(), has(), delete()만 있다.
- WeakMap 오브젝트에 forEach(), entries() 메서드는 존재하지 않는다.
- WeakMap 오브젝트는 size 프로퍼티가 없어서 현재의 [key, value]수를 알 수 없다.



# WeakMap

- new WeakMap()
  - WeakMap 인스턴스를 생성하여 반환한다.

구분	타입	데이터(값)
형태		new WeakMap()
파라미터	object	(선택), object, 이터러블 오브젝트 안에 작성
반환	WeakMap	생성한 WeakMap 인스턴스

```
const emptyWeakMap = new WeakMap();

let obj = {};
const newWeakMap = new WeakMap([
  [obj, "오브젝트"]
]);
```

# WeakMap

- set()
  - WeakMap 인스턴스에 key와 value를 설정한다.

구분	타입	데이터(값)
형태		WeakMap.prototype.set()
파라미터	object	Key, Object/Function 등의 오브젝트
	any	value
반환	WeakMap	Key, value가 추가된 WeakMap 인스턴스

# WeakMap

## ○ set()

```
const newWeakMap = new WeakMap();

(function(){
  var obj = {item: "weakmap"};
  newWeakMap.set(obj, "GC");
})();

const newMap = new Map();
(function(){
  var obj = {item: "map"};
  newMap.set(obj, "Keep");
})();

setTimeout(function() {
  console.log("1:", newWeakMap);
  console.log("2:", newMap);
}, 1000);
```

```
1: ▼ WeakMap { {... } => "GC" } ⓘ
  ▼ [[Entries]]
    No properties
  ▶ __proto__: WeakMap

2: ▼ Map(1) { {... } => "Keep" } ⓘ
  ▼ [[Entries]]
    ▼ 0: {Object => "Keep"}
      ▶ key: {item: "map"}
        value: "Keep"
      size: (...)
  ▶ __proto__: Map
```

# WeakMap

## ○ set()

```
const newWeakMap = new WeakMap();

let sportsObj = {};
newWeakMap.set(sportsObj, "Object-1");

sportsObj = {};
newWeakMap.set(sportsObj, "Object-2");

setTimeout(function() {
  console.log(newWeakMap);
}, 1000);
```

```
▼ WeakMap {{...} => "Object-2", {...} => "Object-1"} ⓘ
  ▼ [[Entries]]
    ▼ 0: {Object => "Object-2"}
      ► key: {}
      value: "Object-2"
    ► __proto__: WeakMap
```

# WeakMap

## ○ set()

```
const newWeakMap = new WeakMap();

let fn = function(){};
newWeakMap.set(fn, "함수");

newWeakMap.set(fn, "value 변경");
console.log(newWeakMap);
```

```
▼ WeakMap ⓘ
  ▶ __proto__: WeakMap
  ▼ [[Entries]]: Array(1)
    ▼ 0: {function(){} => "value 변경"}
      ▶ key: f ()
        value: "value 변경"
      length: 1
```

# WeakMap

- get()
  - WeakMap 인스턴스에서 key가 같은 value를 설정한다.

구분	타입	데이터(값)
형태		WeakMap.prototype.get()
파라미터	object	Key, 오브젝트
반환	any	Key가 같은 value 값 반환

```
const newWeakMap = new WeakMap();

let obj = {};
newWeakMap.set(obj, "오브젝트");

console.log(newWeakMap.get(obj));
```

오브젝트

# WeakMap

- has()
  - WeakMap 인스턴스에서 key의 존재여부를 반환한다.

구분	타입	데이터(값)
형태		WeakMap.prototype.has()
파라미터	object	Key, 오브젝트
반환	boolean	존재하면 true, 존재하지 않으면 false

```
const newWeakMap = new WeakMap();

let obj = {};
newWeakMap.set(obj, "오브젝트");

console.log(newWeakMap.has(obj));
```

true

# WeakMap

- delete()
  - WeakMap 인스턴스에서 key가 같은 엘리먼트를 삭제한다.

구분	타입	데이터(값)
형태		WeakMap.prototype.delete()
파라미터	object	Key, 삭제할 키 값
반환	boolean	삭제하면 true, 삭제실패는 false

```
const newWeakMap = new WeakMap();

let obj = {};
newWeakMap.set(obj, "오브젝트");

console.log(newWeakMap.delete(obj));
```

true



# Collection

- 각 Collection별 for ~ of 로 iterator 순회하기

```
// array
console.log('Arr -----');
const arr = [1, 2, 3];
let iter1 = arr[Symbol.iterator]();
for (const a of iter1) console.log(a);

// set
console.log('Set -----');
const set = new Set([1, 2, 3]);
for (const a of set) console.log(a);

//map
console.log('Map -----');
const map = new Map([[ 'a', 1 ], [ 'b', 2 ], [ 'c', 3 ]]);
for (const a of map.keys()) console.log(a);
for (const a of map.values()) console.log(a);
for (const a of map.entries()) console.log(a);
```

Arr -----

1

2

3

Set -----

1

2

3

Map -----

a

b

c

1

2

3

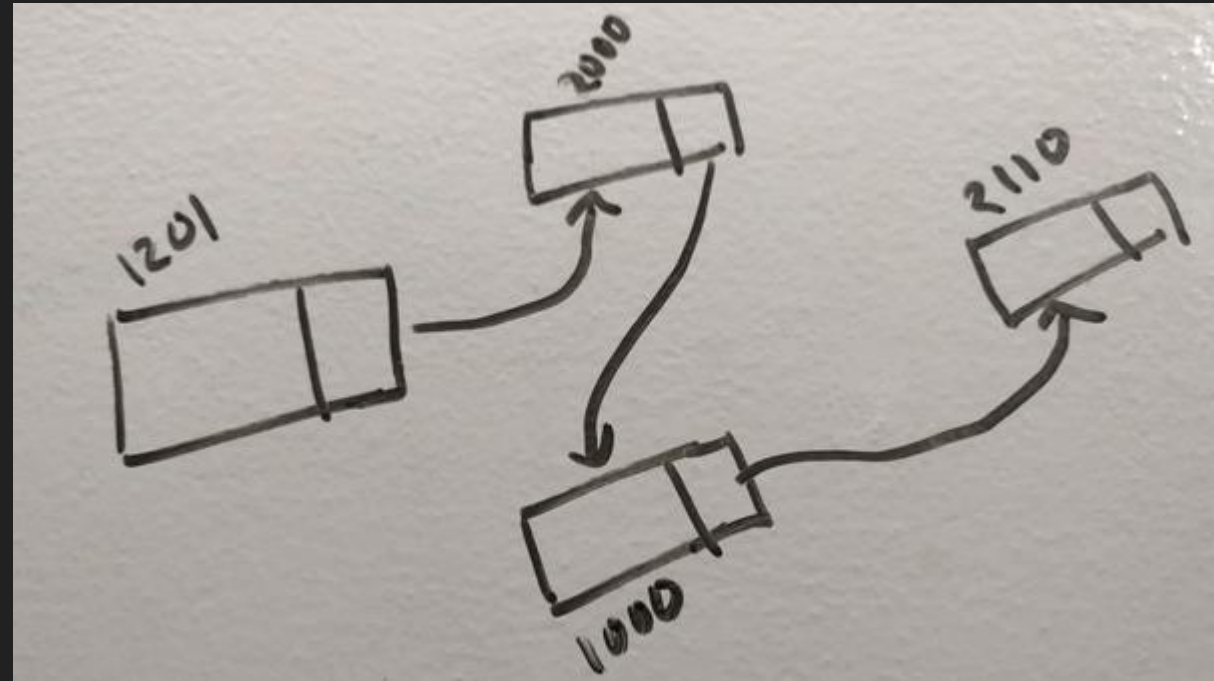
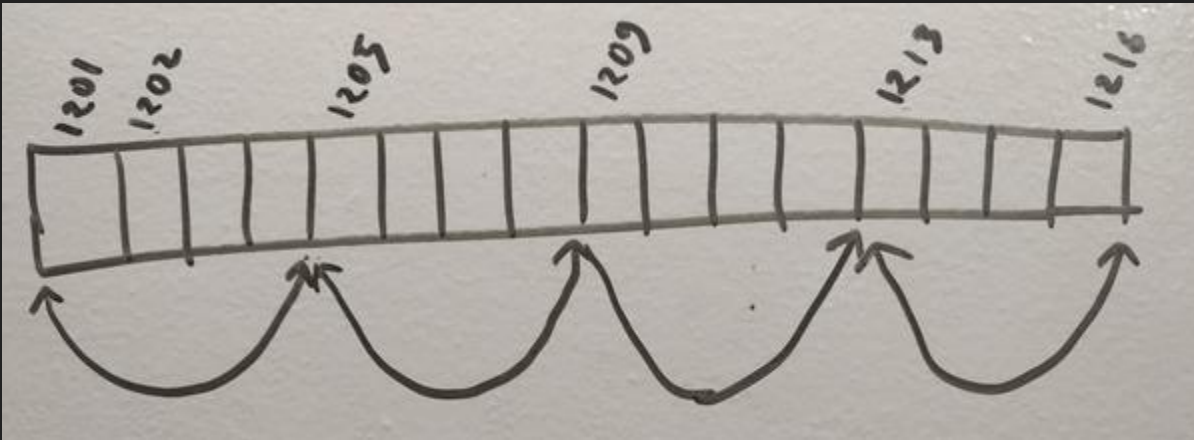
▶ (2) [ "a", 1 ]

▶ (2) [ "b", 2 ]

▶ (2) [ "c", 3 ]

# Typed Array

- JavaScript의 배열이 실제로는 배열이 아닌 이유



# Typed Array

- JavaScript의 배열이 실제로는 배열이 아닌 이유
  - 원래의 JavaScript Array는 인접하지 않고 연속적이지 않다.
  - Array의 정체는 Hash Map으로 구성되어 있으며 Linked List 형태를 통해 다음 데이터가 있는 주소를 가리킬 수 있다.
  - 그래서 각 주소를 순회 하며 조회를 하게 되면 다른 언어의 Array List보다 성능이 매우 떨어지게 된다.
  - 이 점을 고려해 일부 웹 컴파일러에서는 이러한 배열을 연속된 배열로 자동으로 바꿔주기도 한다.
  - 하지만 타입이 전혀 다른 데이터를 삽입할 경우 순회 시 속도는 극심한 차이를 내게 된다.

# Typed Array

- JavaScript의 동일 타입 Array 순회 테스트
  - 동일 타입의 경우 컴파일러가 연속적인 데이터라는 것을 인지하고 바꿔준다.

```
var LIMIT = 10000000;  
var arr = new Array(LIMIT);  
console.time("Array insertion time");  
for (var i = 0; i < LIMIT; i++) {  
    arr[i] = i;  
}  
console.timeEnd("Array insertion time");
```

Array insertion time: 30.305908203125ms

```
var LIMIT = 10000000;  
var buffer = new ArrayBuffer(LIMIT * 4);  
var arr = new Int32Array(buffer);  
console.time("ArrayBuffer insertion time");  
for (var i = 0; i < LIMIT; i++) {  
    arr[i] = i;  
}  
console.timeEnd("ArrayBuffer insertion time");
```

ArrayBuffer insertion time: 35.873779296875ms

# Typed Array

## ○ JavaScript의 동일하지 않는 타입 Array 순회 테스트

```
console.time("Array read time");
var LIMIT = 10000000;
var arr = new Array(LIMIT);
arr.push({a: 22});
for (var i = 0; i < LIMIT; i++) {
    arr[i] = i;
}
var p;
for (var i = 0; i < LIMIT; i++) {
    //arr[i] = i;
    p = arr[i];
}
console.timeEnd("Array read time");
Array read time: 1414.826171875ms
```

```
console.time("ArrayBuffer read time");
var LIMIT = 10000000;
var buffer = new ArrayBuffer(LIMIT * 4);
var arr = new Int32Array(buffer);
for (var i = 0; i < LIMIT; i++) {
    arr[i] = i;
}
for (var i = 0; i < LIMIT; i++) {
    var p = arr[i];
}
console.timeEnd("ArrayBuffer read time");
ArrayBuffer read time: 81.008056640625ms
```

# Typed Array

- Typed Array 구현 요소

- Buffer : 메모리에 바이너리 데이터를 저장하는 영역을 의미한다. 영역이 가변적이지 않으므로 처리 속도가 향상된다
- View : 데이터를 저장하고, 읽고, 변경하고, 삭제하는 것을 의미한다.

```
let bufferObj = new ArrayBuffer(20);  
let int32View = new Int32Array(bufferObj);
```

# Typed Array(Array Buffer)

- ArrayBuffer
  - 바이트 길이가 고정된 바이너리 버퍼
  - 지정한 바이트 변경이 불가능하다.
  - 또한 직접 view가 불가능하다.
  - 대신 TypedArray나 DataView 오브젝트를 사용하여 view를 할 수 있다.
  - ArrayBuffer는 처음부터 끝까지 데이터 타입 또는 바이트 수가 같지 않아도 된다.
  - 예를들어 100바이트로 길이를 정의했을 때 1부터 50까지는 1바이트 단위로 사용하고 51부터 100까지는 2바이트 단위로 사용할 수 있다. 이를 엘리먼트 구조 또는 구조체라고 한다.

# Typed Array(Array Buffer)

- new ArrayBuffer()

- ArrayBuffer 인스턴스를 생성하여 반환한다

구분	타입	데이터(값)
형태		new ArrayBuffer()
파라미터	Number	Length, ArrayBuffer 바이트 수
반환	ArrayBuffer	생성한 ArrayBuffer 인스턴스

- 바이트는 평균적으로 3억 바이트 정도 지원한다.



# Typed Array(Array Buffer)

## ○ new ArrayBuffer()

```
let obj1 = new ArrayBuffer(20);  
console.log(obj1.byteLength);  
  
let obj2 = new ArrayBuffer();  
console.log(obj2.byteLength);  
  
let obj3 = new ArrayBuffer("12");  
console.log(obj3.byteLength);
```

20

0

12

# Typed Array(Array Buffer)

- slice()
  - ArrayBuffer 인스턴스에서 지정한 범위를 복사하여 반환한다.

구분	타입	데이터(값)
형태		ArrayBuffer.prototype.slice
파라미터	Number	Start, 시작 인덱스
	Number	End, 끝 인덱스
반환	ArrayBuffer	복사한 ArrayBuffer 인스턴스

# Typed Array(Array Buffer)

## ○ slice()

```
let newBuffer = new ArrayBuffer(20);  
  
let oneObj = newBuffer.slice(0);  
console.log(oneObj.byteLength);  
  
let twoObj = newBuffer.slice(3, 7);  
console.log(twoObj.byteLength);
```

20

4

# Typed Array(Array Buffer)

- `isView()`

- `TypedArray` 오브젝트 또는 `DataView` 오브젝트 여부를 반환한다.

구분	타입	데이터(값)
형태		<code>ArrayBuffer.prototype.isView()</code>
파라미터	<code>Any</code>	<code>Object</code> , 체크 대상 오브젝트
반환	<code>Boolean</code>	<code>TypedArray</code> 또는 <code>DataView</code> 이면 <code>true</code> 아니면 <code>false</code>

```
let bufferObj = new ArrayBuffer(10);
console.log(ArrayBuffer.isView(bufferObj));

let typedObj = new Int16Array();
console.log(ArrayBuffer.isView(typedObj));

let viewObj = new DataView(bufferObj);
console.log(ArrayBuffer.isView(viewObj));
```

false

true

true

# Typed Array(Array Buffer)

- Symbol.species()
- ArrayBuffer constructor를 반환한다

구분	타입	데이터(값)
형태		Symbol.species
파라미터		없음
반환	ArrayBuffer	생성한 인스턴스

```
class ExtendBuffer extends ArrayBuffer {  
  static get [Symbol.species]() {  
    return ArrayBuffer;  
  }  
}  
  
let newBuffer = new ExtendBuffer(20);  
  
let bufferObj = newBuffer.slice(3, 7);  
console.log(bufferObj.byteLength);
```

# Typed Array

- TypedArray

- 아래 표에 열거된 9개 오브젝트를 대표하는 스펙상의 오브젝트이다

생성자 이름	엘리먼트타입	바이트	개요
Int8Array	Int8	1	8비트 2's 보수법 signed integer
Uint8Array	Uint8	1	8비트 unsigned integer
Uint8ClampedArray	Uint8C	1	8비트 unsigned integer(clamped)
Int16Array	Int16	2	16비트 2's 보수법 signed integer
Uint16Array	Uint16	2	16비트 unsigned integer
Int32Array	Int32	4	32비트 2's 보수법 signed integer
Uint32Array	Uint32	4	32비트 unsigned integer
Float32Array	Float32	4	32비트 IEEE 부동 소수점
Float64Array	Float64	8	64비트 IEEE 부동 소수점

# Typed Array

- TypedArray

## ○ 엘리먼트 타입과 바이트

엘리먼트 타입	2 바이트															
UInt8	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
UInt16	0		1		2		3		4		5		6		7	
UInt32	0				1				2				3			
Float64	0								1							

# Typed Array

- TypedArray
  - 비트값 구성
    - 1바이트는 8비트로 구성되며 0과 1 값을 가질 수 있다.
    - 사인 비트를 갖는 엘리먼트 타입은 2의 보수법으로 음수를 표현한다.
    - 사인 비트를 갖지 않는 엘리먼트 타입은 타입의 첫 문자가 대문자 U로 시작한다.



# Typed Array

- TypedArray

- 사인 비트 값이 0인 8비트 양수

비트	8	7	6	5	4	3	2	1
On/Off	0	1	1	1	1	1	1	1
비트 값	0	64	32	16	8	4	2	1
누적 값	127	127	63	31	15	7	3	1

비트	8	7	6	5	4	3	2	1
On/Off	1	1	1	1	1	1	1	1
비트 값	1	64	32	16	8	4	2	1
누적 값	-128	-127	-63	-31	-15	-7	-3	-1

# Typed Array

- TypedArray

- 사인 비트가 없는 8비트

비트	8	7	6	5	4	3	2	1
On/Off	1	1	1	1	1	1	1	1
비트 값	128	64	32	16	8	4	2	1
누적 값	255	127	63	31	15	7	3	1

# Typed Array

- `new TypedArray(length)`
  - 파라미터의 엘리먼트 수로 `TypedArray` 인스턴스를 생성한다.

구분	타입	데이터(값)
형태		<code>new TypedArray()</code>
파라미터	Number	Length, 엘리먼트 수
반환	TypedArray	생성한 <code>TypedArray</code> 인스턴스

- 여기서 `TypedArray`는 9개의 타입을 대표하는 이름이다.

# Typed Array

- new TypedArray(length)
- new Int16Array(3) 작성방법
  - new Int16Array(3) : Int16 타입의 엘리먼트 3개를 설정한다

```
let int16obj = new Int16Array(3);
console.log("1:", int16obj);

int16obj[0] = 123;
int16obj[1] = 456;
console.log("2:", int16obj);

console.log("3:", int16obj[0]);

console.log("4:", new Int16Array());
```

```
1: ▼ Int16Array(3) ⓘ
  0: 123
  1: 456
  2: 0
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray

2: ▼ Int16Array(3) ⓘ
  0: 123
  1: 456
  2: 0
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray

3: 123

4: ▼ Int16Array(0) ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray
```

# Typed Array

- new TypedArray(length)
- new Int16Array(3) 작성방법 : String type

```
let oneObj = new Int16Array("123");  
console.log("1:", oneObj.length);  
  
let twoObj = new Int16Array("ABC");  
console.log("2:", twoObj.length);
```

1:	123
2:	0

# Typed Array

- `new TypedArray(TypedArray)`
  - 파라미터의 `TypedArray` 인스턴스로 `TypedArray` 인스턴스를 생성한다

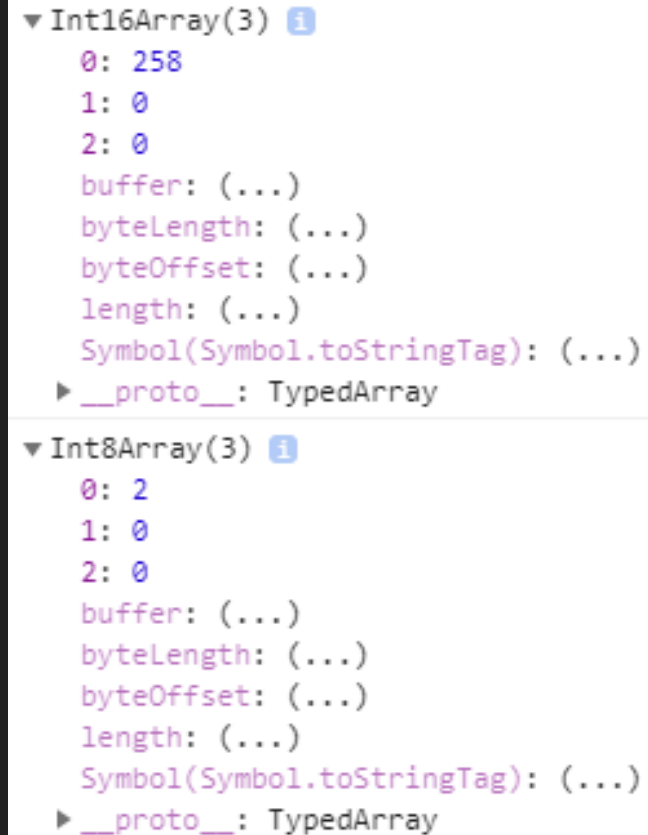
구분	타입	데이터(값)
형태		<code>new TypedArray()</code>
파라미터	<code>TypedArray</code>	<code>TypedArray</code> 인스턴스
반환	<code>TypedArray</code>	생성한 <code>TypedArray</code> 인스턴스

- 복사해주는 `TypedArray` 인스턴스의 바이트 수보다 생성한 `TypedArray` 인스턴스의 바이트 수가 작으면 값이 잘린다.

# Typed Array

## ○ new TypedArray(TypedArray)

```
let int16obj = new Int16Array(3);  
int16obj[0] = 258;  
  
console.log(new Int16Array(int16obj));  
  
console.log(new Int8Array(int16obj));
```



```
▼ Int16Array(3) ⓘ  
  0: 258  
  1: 0  
  2: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray  
  
▼ Int8Array(3) ⓘ  
  0: 2  
  1: 0  
  2: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

# Typed Array

- new TypedArray(object)
  - 파라미터의 오브젝트로 TypedArray 인스턴스를 생성한다.

구분	타입	데이터(값)
형태		new TypedArray()
파라미터	object	object, Array 등의 인스턴스
반환	TypedArray	생성한 TypedArray 인스턴스



# Typed Array

## ○ new TypedArray(object)

```
let oneObj = new Int16Array([12, 34, 56]);  
console.log("1:", oneObj.length);  
  
let twoObj = new Int16Array({0: 12, 1: 34});  
console.log("2:", twoObj.length);  
  
let threeObj = new Int16Array({0: 12, 1: 34, length:2});  
console.log("3:", threeObj.length);
```

1:	3
2:	0
3:	2

# Typed Array

- new TypedArray(ArrayBuffer)
  - 파라미터의 ArrayBuffer로 TypedArray 인스턴스를 생성한다.

구분	타입	데이터(값)
형태		new TypedArray()
파라미터	ArrayBuffer	buffer, ArrayBuffer 인스턴스
	Number	(선택), offset, offset 값을 바이트 수로 지정
	Number	(선택), length, ArrayBuffer에서 사용할 엘리먼트 수
반환	TypedArray	생성한 TypedArray 인스턴스

# Typed Array

## ○ new TypedArray(ArrayBuffer)

```
let bufferObj = new ArrayBuffer(32);  
  
let oneObj = new Int16Array(bufferObj);  
console.log(oneObj.length);  
  
console.log(oneObj.byteLength);  
  
let twoObj = new Int32Array(bufferObj);  
console.log(twoObj.length);
```

16

32

8

```
let bufferObj = new ArrayBuffer(10);  
let oneObj = new Int16Array(bufferObj, 4);  
  
console.log(oneObj.byteLength);  
console.log(oneObj.length);  
  
oneObj[1] = 22;  
console.log(oneObj);
```

```
6  
3  
▼ Int16Array(3) ⓘ  
  0: 0  
  1: 22  
  2: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ► __proto__: TypedArray
```

# Typed Array

## ○ new TypedArray(ArrayBuffer)

```
let bufferObj = new ArrayBuffer(10);  
let oneObj = new Int16Array(bufferObj, 4, 2);  
  
oneObj[0] = 22;  
console.log(oneObj);  
  
let twoObj = new Int16Array(bufferObj);  
console.log(twoObj);
```

```
▼ Int16Array(2) ⓘ  
  0: 22  
  1: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

```
▼ Int16Array(5) ⓘ  
  0: 0  
  1: 0  
  2: 22  
  3: 0  
  4: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

# Typed Array

## ○ new TypedArray(ArrayBuffer)

```
let bufferObj = new ArrayBuffer(1);  
  
let int8Obj = new Int8Array(bufferObj);  
  
[127, 128, 129].forEach(function(value){  
  int8Obj[0] = value;  
  console.log(value, ":", int8Obj[0]);  
});
```

127	:	127
128	:	-128
129	:	-127

```
let bufferObj = new ArrayBuffer(1);  
  
let uint8Obj = new Uint8Array(bufferObj);  
[255, 256, 257].forEach(function(value){  
  uint8Obj[0] = value;  
  console.log(value, ":", uint8Obj[0]);  
});
```

255	:	255
256	:	0
257	:	1

# Typed Array

## ○ new TypedArray(ArrayBuffer)

```
let bufferObj = new ArrayBuffer(1);

let clampedObj = new Uint8ClampedArray(bufferObj);
[255, 256, 257].forEach(function(value){
  clampedObj[0] = value;
  console.log(value, ":", clampedObj[0]);
});
```

255	:	255
256	:	255
257	:	255

```
let buffer64Obj = new ArrayBuffer(8);

let float64Obj = new Float64Array(buffer64Obj);

buffer64Obj[0] = Number.MAX_VALUE;
console.log(buffer64Obj[0] == Number.MAX_VALUE);
```

true

# Typed Array

- BYTES\_PER\_ELEMENT

- TypedArray 인스턴스에서 엘리먼트 하나의 바이트 수를 반환한다.

구분	타입	데이터(값)
형태		TypedArray.BYTES_PER_ELEMENT
파라미터		프로퍼티 이므로 파라미터가 없음
반환	Number	9개 타입 인스턴스의 엘리먼트 바이트 수

```
let bufferObj = new ArrayBuffer(10);  
let oneObj = new Int16Array(bufferObj);  
  
console.log(oneObj.BYTES_PER_ELEMENT);
```

2

# Typed Array

## ○ buffer

- TypedArray 인스턴스를 생성할 때 사용한 ArrayBuffer 인스턴스를 반환한다.

구분	타입	데이터(값)
형태		TypedArray.prototype.buffer
파라미터		프로퍼티 이므로 파라미터가 없음
반환	ArrayBuffer	TypedArray 인스턴스를 생성할 때 사용한 ArrayBuffer

```
let bufferObj = new ArrayBuffer(10);
let oneObj = new Int16Array(bufferObj, 4);

console.log(oneObj.buffer);
console.log(oneObj.buffer.byteLength);
```

```
▼ ArrayBuffer(10) ⓘ
  ▶ [[Int8Array]]: Int8Array(10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
  ▶ [[Int16Array]]: Int16Array(5) [0, 0, 0, 0, 0]
  ▶ [[Uint8Array]]: Uint8Array(10) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    byteLength: (...)
  ▶ __proto__: ArrayBuffer
10
```



# Typed Array

- `byteOffset`
  - `ArrayBuffer` 인스턴스의 오프셋 값을 반환한다.

구분	타입	데이터(값)
형태		<code>TypedArray.prototype.byteOffset</code>
파라미터		파라미터가 없음
반환	<code>Number</code>	오프셋 값

```
let bufferObj = new ArrayBuffer(10);  
let oneObj = new Int16Array(bufferObj, 4);  
  
console.log(oneObj.byteOffset);
```

4

# Typed Array

- from()

- Array-like 또는 이터러블 오브젝트로 TypedArray 인스턴스를 생성한다.

구분	타입	데이터(값)
형태		TypedArray.from()
파라미터	Object	Source, Array-like 또는 이터러블 오브젝트
	Function	(선택) fn, 생성하는 엘리먼트마다 호출할 함수
	Object	(선택) this, 호출한 함수에서 this로 참조할 오브젝트
반환	TypedArray	생성한 TypedArray 인스턴스

# Typed Array

## ○ from()

```
console.log(Uint8Array.from([12, 34]));
console.log(Uint8Array.from("12"));

let threeObj = Uint8Array.from("56", function(value){
  console.log(value);
  return value;
}, this);
console.log(threeObj);

let fourObj = Uint8Array.from({length: 3}, function(value, key){
  return key;
});
console.log(fourObj);
```

```
▼ Uint8Array(2) [12, 34] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 12
  1: 34
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray

▼ Uint8Array(2) [1, 2] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 1
  1: 2
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray

5
6

▼ Uint8Array(2) [5, 6] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 5
  1: 6
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray

▼ Uint8Array(3) [0, 1, 2] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 0
  1: 1
  2: 2
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray
```

# Typed Array

- of()
- TypedArray 인스턴스를 생성하고 파라미터 값을 설정한다

구분	타입	데이터(값)
형태		TypedArray.of()
파라미터	Number	Item0[, item1[, item2...]]
반환	TypedArray	생성한 TypedArray 인스턴스

```
console.log(Uint8Array.of(10));  
console.log(Uint8Array.of(20, 30, 40));  
  
console.log(Uint8Array.of(null));  
console.log(Uint8Array.of("4", "A", "6"));
```

```
▼ Uint8Array(1) ⓘ  
  0: 10  
▼ Uint8Array(3) ⓘ  
  0: 20  
  1: 30  
  2: 40  
▼ Uint8Array(1) ⓘ  
  0: 0  
▼ Uint8Array(3) ⓘ  
  0: 4  
  1: 0  
  2: 6
```

# Typed Array

- set()
  - 생성한 TypedArray 인스턴스에 값을 설정한다

구분	타입	데이터(값)
형태		TypedArray.prototype.set()
파라미터	Array	배열 또는 TypedArray
	Number	(선택) offset, 오프셋
반환	TypedArray	값을 설정한 TypedArray 인스턴스

# Typed Array

## ○ set()

```
let bufferObj = new ArrayBuffer(6);  
let uint8Obj = new Uint8Array(bufferObj);  
  
uint8Obj.set([10, 20, 30], 2);  
console.log(uint8Obj);
```

```
let oneObj = new ArrayBuffer(7);  
let twoObj = new Uint8Array(oneObj);
```

```
twoObj.set(uint8Obj, 1);  
console.log(twoObj);
```

```
▼ Uint8Array(6) ⓘ  
  0: 0  
  1: 0  
  2: 10  
  3: 20  
  4: 30  
  5: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

```
▼ Uint8Array(7) ⓘ  
  0: 0  
  1: 0  
  2: 0  
  3: 10  
  4: 20  
  5: 30  
  6: 0  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

# Typed Array

- subarray()
  - TypedArray 인스턴스를 생성하고 여기에 ArrayBuffer 데이터를 설정하여 반환한다.

구분	타입	데이터(값)
형태		TypedArray.prototype.subarray()
파라미터	Number	(선택) begin 오프셋
	Number	(선택) end 오프셋
반환	TypedArray	생성한 TypedArray 인스턴스

```
let bufferObj = new ArrayBuffer(6);
let uint8Obj = new Uint8Array(bufferObj);

uint8Obj.set([10, 20, 30], 1);
console.log(uint8Obj);

console.log(uint8Obj.subarray(1, 4));
```

```
▼ Uint8Array(6) ⓘ
  0: 0
  1: 10
  2: 20
  3: 30
  4: 0
  5: 0
```

```
▼ Uint8Array(3) ⓘ
  0: 10
  1: 20
  2: 30
```

# Typed Array

- Symbol.iterator()
- iterator 오브젝트를 생성하여 반환한다.

구분	타입	데이터(값)
형태		Symbol.iterator()
파라미터		없음
반환	Iterator	생성한 이터레이터 오브젝트



# Typed Array

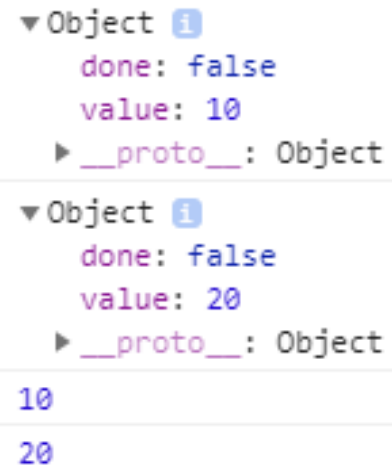
## ○ Symbol.iterator()

```
let uint8obj = new Uint8Array([10, 20]);
let iteratorObj = uint8obj[Symbol.iterator]();

console.log(iteratorObj.next());

console.log(iteratorObj.next());

for (let value of uint8obj) {
  console.log(value);
};
```



```
▼ Object ⓘ
  done: false
  value: 10
  ► __proto__: Object

▼ Object ⓘ
  done: false
  value: 20
  ► __proto__: Object

10
20
```

# Typed Array

- Symbol.species()
- constructor를 반환한다

구분	타입	데이터(값)
형태		Symbol.species
파라미터		없음
반환	Instance	생성한 인스턴스

```
class extendUint8 extends Uint8Array{
  static get [Symbol.species]() {
    return Uint8Array;
  }
}

let uint8ExtendObj = new extendUint8([10, 20, 30]);

let filterObj = uint8ExtendObj.filter(value => value > 10);
console.log(filterObj);
```

```
▼ Uint8Array(2) ⓘ
  0: 20
  1: 30
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  Symbol(Symbol.toStringTag): (...)
  ▶ __proto__: TypedArray
```

# Typed Array

- copyWithin()

- 같은 TypedArray에서 지정한 범위의 값을 복사하여 지정한 위치에 설정한다

구분	타입	데이터(값)
형태		TypedArray.copyWithin()
파라미터	Number	Target, 복사한 값을 할당할 인덱스
	Number	(선택) 시작 인덱스
	Number	(선택) 끝 인덱스
반환	TypedArray	같은 TypedArray 인스턴스

# Typed Array

## ○ copyWithin()

```
var buffer = new ArrayBuffer(8);
var uint8 = new Uint8Array(buffer);
uint8.set([1,2,3]);
console.log(uint8); // Uint8Array [ 1, 2, 3, 0, 0, 0, 0, 0 ]
uint8.copyWithin(3,0,3);
console.log(uint8); // Uint8Array [ 1, 2, 3, 1, 2, 3, 0, 0 ]
```

```
▼ Uint8Array(8) [1, 2, 3, 0, 0, 0, 0, 0] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 1
  1: 2
  2: 3
  3: 1
  4: 2
  5: 3
  6: 0
  7: 0
  Symbol(Symbol.toStringTag): (...)
  ► __proto__: TypedArray

▼ Uint8Array(8) [1, 2, 3, 1, 2, 3, 0, 0] ⓘ
  buffer: (...)
  byteLength: (...)
  byteOffset: (...)
  length: (...)
  0: 1
  1: 2
  2: 3
  3: 1
  4: 2
  5: 3
  6: 0
  7: 0
  Symbol(Symbol.toStringTag): (...)
  ► __proto__: TypedArray
```

# Typed Array(활용)

- 구조체(Structure)

- 구조체란 서로 다른 데이터 타입을 하나로 묶어 놓은 형태를 의미한다.

- [시나리오]

명칭	타입	바이트	엘리먼트수	전체바이트	프로퍼티키	프로퍼티값
품명 코드	Uint8	1	10	10	Code	"Book"
품명	Uint16	2	10	20	Desc	"자바스크립트"
수량	Uint16	2	1	2	Qty	10
단가	Uint16	2	1	2	Price	20
금액	Uint32	4	1	4	amount	200

# Typed Array(활용)

## ○ 구조체(1)

```
let itemObj = {code: "book", desc: "자바스크립트",  
  qty: 10, price: 20, amount: 200};  
  
let bufferObj = new ArrayBuffer(40);  
  
let codeObj = new Uint8Array(bufferObj, 0, 10);  
for (var k = 0; k < itemObj.code.length; k++){  
  codeObj.set([itemObj.code.charCodeAt(k)], k);  
};  
  
console.log(codeObj);
```

```
▼ Uint8Array(10) [98, 111, 111, 107, 0, 0, 0, 0, 0, 0] ⓘ  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  length: (...)  
  0: 98  
  1: 111  
  2: 111  
  3: 107  
  4: 0  
  5: 0  
  6: 0  
  7: 0  
  8: 0  
  9: 0  
  Symbol(Symbol.toStringTag): (...)  
  ▶ __proto__: TypedArray
```

# Typed Array(활용)

## ○ 구조체(2)

```
let descObj = new Uint16Array(bufferObj, 10, 10);

for (var k = 0; k < itemObj.desc.length; k++){
    descObj.set([itemObj.desc.charCodeAt(k)], k);
};

console.log(descObj);

let qtyObj = new Uint16Array(bufferObj, 30, 1);
qtyObj.set([itemObj.qty]);

console.log(qtyObj);

let priceObj = new Uint16Array(bufferObj, 32, 1);
priceObj.set([itemObj.price]);

console.log(priceObj);

let amountObj = new Uint32Array(bufferObj, 36, 1);
amountObj.set([itemObj.amount]);

console.log(amountObj);
```

▶ Uint16Array(10) [51088, 48148, 49828, 53356, 47549, 53944, 0, 0, 0, 0]

▶ Uint16Array [10]

▶ Uint16Array [20]

▶ Uint32Array [200]

# Typed Array(활용)

## ○ 구조체(3)

```
let result = [];  
for (var k = 0; k < itemObj.code.length; k++){  
    result.push(String.fromCharCode(codeObj[k]));  
};  
console.log(result.join(''));  
  
result = [];  
for (var k = 0; k < itemObj.desc.length; k++){  
    result.push(String.fromCharCode(descObj[k]));  
};  
console.log(result.join(''));  
  
console.log(qtyObj[0]);  
console.log(priceObj[0]);  
console.log(amountObj[0]);
```

Book
자바스크립트
10
20
200



# Typed Array(Data View)

- DataView 오브젝트

- ArrayBuffer 데이터를 TypedArray 오브젝트와 DataView 오브젝트로 View 할 수 있으며 공유가 가능하다.

- 인스턴스 생성

- TypedArray는 파라미터에 ArrayBuffer 인스턴스를 지정하여 9개 타입의 인스턴스를 생성하고, 생성한 인스턴스에서 제공하는 set()과 같은 메서드로 ArrayBuffer 인스턴스에 데이터를 저장한다.

- 한편, DataView는 파라미터에 ArrayBuffer 인스턴스를 지정하여 인스턴스를 생성하는 것은 같지만, 타입별로 인스턴스를 생성하지 않고 인스턴스 하나만 생성한다. 대신, 인스턴스에 포함된 타입별 getter와 setter 메서드를 사용하여 ArrayBuffer 메서드를 View한다.

```
let bufferObj = new ArrayBuffer(5);  
let uint8Obj = new Uint8Array(bufferObj);  
uint8Obj.set([10], 1);  
  
let viewObj = new DataView(bufferObj);  
viewObj.setUint8(3, 30);
```

# Typed Array(Data View)

- DataView 오브젝트

- 엔디언 지정 가능

- 엔디언은 메모리에 데이터를 배치하는 기준이다.
    - TypedArray 오브젝트는 엔디언을 지정할 수 없지만, DataView 오브젝트는 엔디언 지정이 가능하다.
    - TypedArray는 그래픽과 같은 청크(Chunk)데이터를 메모리에서 처리하기 위한 것이 목적이다.
    - 고려할 것은 메모리에 데이터를 배치하는 방법이 디바이스에 따라 다르다.
    - TypedArray 오브젝트는 메모리에서 바이너리 데이터를 처리하는 데 사용하고, DataView 오브젝트는 다른 디바이스와 데이터를 송수신할 때 사용한다.

# Typed Array(Data View)

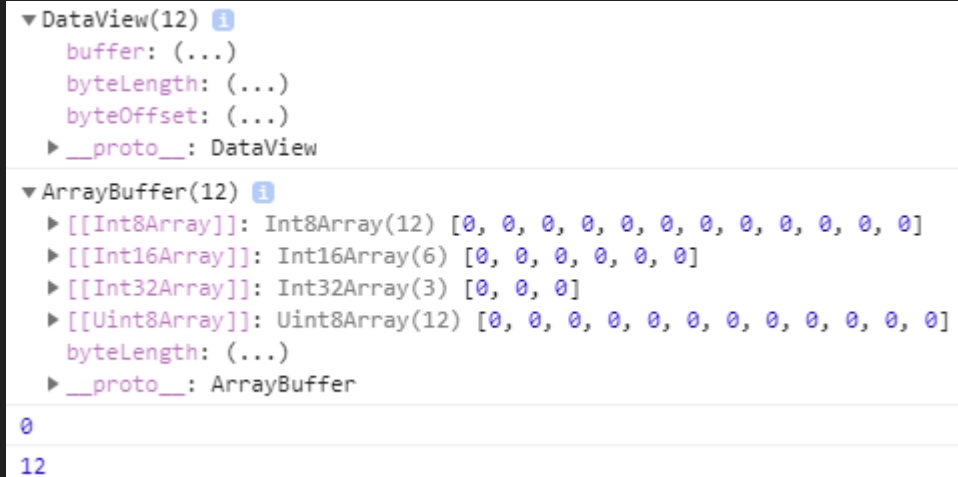
- new DataView()
  - DataView 인스턴스를 생성하여 반환한다.

구분	타입	데이터(값)
형태		New DataView()
파라미터	ArrayBuffer	Buffer, ArrayBuffer 인스턴스
	Number	(선택) BufferOffset
	Number	(선택) ByteLength
반환	DataView	생성한 DataView 인스턴스

# Typed Array(Data View)

## ○ new DataView()

```
let bufferObj = new ArrayBuffer(12);  
let viewObj = new DataView(bufferObj);  
  
console.log(viewObj);  
  
console.log(viewObj.buffer);  
console.log(viewObj.byteOffset);  
console.log(viewObj.byteLength);
```



```
▼ DataView(12) ⓘ  
  buffer: (...)  
  byteLength: (...)  
  byteOffset: (...)  
  ▶ __proto__: DataView  
  
▼ ArrayBuffer(12) ⓘ  
  ▶ [[Int8Array]]: Int8Array(12) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  ▶ [[Int16Array]]: Int16Array(6) [0, 0, 0, 0, 0, 0]  
  ▶ [[Int32Array]]: Int32Array(3) [0, 0, 0]  
  ▶ [[Uint8Array]]: Uint8Array(12) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
  byteLength: (...)  
  ▶ __proto__: ArrayBuffer  
  
0  
12
```

# Typed Array(Data View)

## ○ new DataView()

```
let bufferObj = new ArrayBuffer(12);  
let viewObj = new DataView(bufferObj, 1, 8);  
  
console.log(viewObj.byteOffset);  
console.log(viewObj.byteLength);
```

1

8

# Typed Array(Data View)

- getInt8()
  - 사인 부호가 있는 8비트 값을 반환한다

구분	타입	데이터(값)
형태		DataView.prototype.getInt8()
파라미터	Number	DataOffset, (선택)오프셋 바이트 수
반환	Number	오프셋 위치의 값

```
let bufferObj = new ArrayBuffer(4);

let int8Obj = new Int8Array(bufferObj);
int8Obj.set([10, 20, 30]);

let viewObj = new DataView(bufferObj);

for (var k = 0; k < viewObj.byteLength; k++){
  console.log(viewObj.getInt8(k));
};
```

10

20

30

0

# Typed Array(Data View)

- setInt8()

- 사인 부호가 있는 8비트에 값을 설정한다

구분	타입	데이터(값)
형태		DataView.prototype.setInt8()
파라미터	Number	byteOffset, 오프셋 바이트 수
	Number	Value, 설정할 값
반환		없음

```
let bufferObj = new ArrayBuffer(6);  
let viewObj = new DataView(bufferObj);  
  
viewObj.setInt8(2, 127);  
console.log(viewObj.getInt8(2));
```

127

# Typed Array(Data View)

- setUint8()

- 사인 부호가 없는 8비트에 값을 설정한다

구분	타입	데이터(값)
형태		DataView.prototype.setUint8()
파라미터	Number	byteOffset, 오프셋 바이트 수
	Number	Value, 설정할 값
반환		없음

```
let bufferObj = new ArrayBuffer(4);  
let viewObj = new DataView(bufferObj);  
  
viewObj.setUint8(1, 255);  
console.log(viewObj.getUint8(1));
```

255



# Typed Array(Data View)

- 엔디언(Endian)
  - 메모리에 데이터를 배치하는 방법이며 바이트 오더라고도 불린다.
  - 엔디언은 세 개의 타입이 있다

엔디언	메모리에 값 배치 형태	설명
빅-엔디언	0x12 0x34 0x56 0x78	12345678을 두 바이트씩 앞에서부터 배치
리틀-엔디언	0x78 0x56 0x34 0x12	12345678을 두 바이트씩 뒤에서부터 배치
믹스드-엔디언	0x34 0x12 0x78 0x56	빅-엔디언과 리틀 엔디언을 섞은 형태로 배치

# Typed Array(Data View)

- setUint16()
  - 사인 부호가 있는 16비트에 값을 설정한다

구분	타입	데이터(값)
형태		DataView.prototype.setUint16()
파라미터	Number	byteOffset, 오프셋 바이트 수
	Number	Value, 설정할 값
	Boolean	(선택) 빅-엔디언: false/undefined, 리틀-엔디언 : true
반환		없음

# Typed Array(Data View)

## ○ setUint16()

```
let bufferObj = new ArrayBuffer(6);  
let viewObj = new DataView(bufferObj);  
  
viewObj.setInt16(2, 32767);  
console.log(viewObj.getInt16(2));  
  
viewObj.setInt16(2, 32767, true);  
console.log(viewObj.getInt16(2, true));  
  
console.log(viewObj.getInt16(2));
```

32767

32767

-129