기존 js vs ES6

태그 이론

첫번째, 템플릿 리터럴이 추가 되었습니다.

기존 js에서는 문자와 데이터를 섞어서 사용시 더하기연산자를 사용하여 데이터를 문자열사이에 추가해주어야 했지만 템플릿 리터럴의 추가로 백틱과 달러표시로 간결하게 사용가능하게 되었습니다.

두번째는 구조분해 할당입니다.

배열이나 객체의 속성 혹은 값을 해체해서 그 값을 변수에 각각 담아 사용하는 것을 말합니다. 할당하려는 변수명과 구조화된 데이터의 속성명이 같으면 기존의 배열 값을 변수에 한꺼번에 할당 가능하고 스프레드 연산자 사용시 값 자체를 복사하고 분리하여 원하는 값만 선택할 수 있게 되었습니다.

세번째는 변수키워드 let, const가 추가 되었습니다.

let, const 변수키워드의 추가로 할당된 값을 변하지 못하게 함으로써 할당된 값을 변화시켜야할때, 변화시키면 안될때를 명확하게 구분지어 변수를 선언할 수 있게 되었습니다.

네번째는 화살표 함수를 사용할 수 있게 되었습니다.

화살표 함수를 사용함으로써, 기존에 함수선언식, 표현식에 비해 많은 부분이 함수를 선언할 때 썻던 코드가 간결하게 쓸 수 있게 되었습니다.

ES6 (ES2015) 의 특징들

모든 특징들을 나열하진 않고 중요하다고 생각하는 특징들만 나열한다.

화살표 함수(Arrow Function)

`=>` 로 사용할 수 있으며 함수와 달리 `this` 가 함수 스코프에 바인딩 되지 않고 렉시컬 스코프를 가진다. 즉, 자신을 감싸는 코드와 동일한 `this` 를 공유한다. 또한 표현식과 문에서도 사용할 수 있다.

```
// 표현식(expression)
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// 문(Statement)
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// 렉시컬 this
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

클래스(Classes)

프로토타입 기반의 객체지향 패턴을 쉽게 만든 장치로, 상속과 생성자 및 인스턴스와 정적 메서드 등을 지원한다.

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);
```

```
    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  get boneCount() {
    return this.bones.length;
  }
  set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

향상된 객체 리터럴

객체 리터럴로 객체를 만들 때 프로퍼티 지정을 좀 더 유연하게 할 수 있도록 기능이 확장되었다.

```
var obj = {
    // 프로토타입 객체 지정
    __proto__: theProtoObj,
    // 프로퍼티이름과 값이 동일할 경우 줄일 수 있음
    handler,
    // 메서드
    toString() {
     // super 호출
     return "d " + super.toString();
    },
    // 계산된 프로퍼티
    [ 'prop_' + (() => 42)() ]: 42
};
```

템플릿 문자열(Template String)

복잡한 문자열을 쉽게 만들어주는 장치로 문자열 안에 문자열 및 변수를 넣을 수 있고 여러 줄의 문자열이 가능하다.

```
// 문자열 안에 문자열 사용하기
`In JavaScript '\\n' is a line-feed.`

// 여러 줄 문자열
`In JavaScript this is
 not legal.`

// 문자열 보간(interpolation)
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`
```

비구조화(Destructuring)

배열과 객체의 패턴 매칭을 통해서 바인딩을 하는 기법이다.

```
// 배열 매칭
var [a, , b] = [1,2,3];

// 객체 매칭
var { op: a, lhs: { op: b }, rhs: c }
       = getASTNode()

// 객체 매칭을 단축해서 사용
// op, lhs, rhs가 스코프 내에서 바인딩됨
var {op, lhs, rhs} = getASTNode()

// 매개변수 위치에도 사용 가능
function g({name: x}) {
  console.log(x);
}
g({name: 5})

// Fail-soft 비구조화 (Fail-soft는 고장이 나도 작동하도록 짠 프로그램을 말한다)
var [a] = [];
a === undefined;
```

```
// 기본 값이 있는 Fail-soft 비구조화
var [a = 1] = [];
a === 1;
```

기본 값 + Rest + Spread

기본 값은 주어지는 값이 없을 때 초기화시키는 값이고 rest 문법은 명시한 변수 외에 나머지를 배열로 가져오는 것이다. Spread 문법은 배열을 반대로 펼치는 역할이다.

```
function f(x, y=12) {
  // y가 없거나 undefined이면 12이다.
  return x + y;
}
f(3) == 15
```

```
function f(x, ...y) {
  // y는 배열이다.
  return x * y.length;
}
f(3, "hello", true) == 6
```

```
function f(x, y, z) {
  return x + y + z;
}
// 각 배열의 원소를 인자로 넘긴다.
f(...[1,2,3]) == 6
```

Let + Const

블록 스코프를 갖고 재선언이 불가능하며 선언 이전에 사용할 수 없다.

```
function f() {
  {
    let x;
    {
      // 블록 스코프를 갖기 때문에 허용!
      const x = "sneaky";
      // const는 재할당 불가, 에러!
      x = "foo";
    }
    // 해당 블록에 이미 선언됨, 재선언 불가이므로 에러!
    let x = "inner";
  }
}
```

반복자(Iterator) + For...Of

반복자는 자신만의 반복을 정의하는 규약이고 이는 `for...of` 를 통해 순회할 수 있다. `[Symbol.iterator]` 라는 이름의 메서드를 정의해야하며 그 메서드는 반드시 `next()` 메서드를 가진 객체를 반환해야 한다.

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}

for (var n of fibonacci) {
  if (n > 1000)
    break;
  console.log(n);
}
```

제너레이터(Generator)

반복자를 쉽게 생성해주는 것으로 `function*` 과 `yield` 를 사용한다. 반복자의 하위 타입으로, `next` 와 `throw` 를 포함한다. 또한 ES7의 `await` 과 같이 사용할 수 있다.

```
var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  if (n > 1000)
    break;
  console.log(n);
}
```

모듈(Module)

- **import 로 모듈 불러오기**

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```

- **export 로 모듈 내보내기**

```
// lib/mathplusplus.js
export * from "lib/math";
export var e = 2.71828182846;
export default function(x) {
    return Math.log(x);
}
```

Map + Set + WeakMap + WeakSet

자주 쓰이는 자료구조로, Weak이 붙은 것은 가비지 컬렉션을 허용하며 `size` 프로퍼티를 가지지 않는다.

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;

// Weak Maps
var wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.size === undefined
```

```
// Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });
// WeakSet에 들어간 객체가 어떠한 참조도 가지지 않으므로 가비지 컬렉션이 된다.
```

심볼(Symbol)

새로 추가된 **원시타입** 으로 유일한 값을 가지며 객체의 접근제어를 가능하게 한다. `description` 매개변수를 이용해 디버깅이 가능하며 `Object.getOwnPropertySymbols` 를 통해 객체의 심볼 프로퍼티들을 볼 수 있다.

```
var MyClass = (function() {

  // IIFE 안의 심볼, 모듈화 된 것.
  // 즉, 심볼로 private 데이터 만듦
  var key = Symbol("key");

  function MyClass(privateData) {
    this[key] = privateData;
  }

  MyClass.prototype = {
    doStuff: function() {
      ... this[key] ...
    }
  };

  return MyClass;
})();

var c = new MyClass("hello");
// "key"와 Symbol("key")는 다르다.
c["key"] === undefined
```

Number + String + Array + Object 에 추가된 것들

```
Number.isInteger(Infinity) // false
Number.isNaN("NaN") // false

"abcde".includes("cd") // true
"abc".repeat(3) // "abcabcabc"

Array.from(document.querySelectorAll('*')) // 유사배열객체를 배열로 변환
Array.of(1, 2, 3) // [1,2,3]
[0, 0, 0].fill(7, 1) // [0,7,7]
[1, 2, 3].find(x => x == 3) // 3
[1, 2, 3].findIndex(x => x == 2) // 1
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
["a", "b", "c"].entries() // 반복자 [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // 반복자 0, 1, 2
["a", "b", "c"].values() // 반복자 "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) }) // 얕은 복사
```

프라미스(Promise)

비동기 작업이 맞이할 미래의 완료/실패와 결과 값을 나타내는 객체이다.

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, duration);
    })
}

var p = timeout(1000).then(() => {
    return timeout(2000);
}).then(() => {
    throw new Error("hmm");
}).catch(err => {
    return Promise.all([timeout(100), timeout(200)]);
})
```


참고

- lukehoban, es6features github repo
- 각 특징들의 MDN 관련 문서들