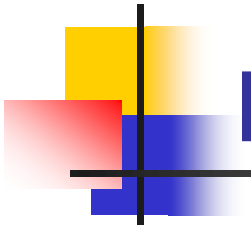
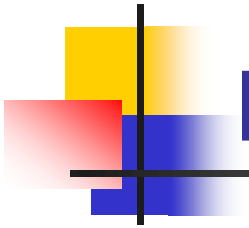
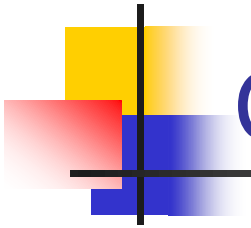


ECMAScript 6 교재



ECMAScript 6

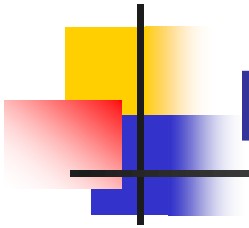
- New Features
  - Constants
  - Block-Scoping
  - Arrow Functions
  - Extended Parameter Handling
  - Template Literals
  - Destructuring Assignment
  - Modules
  - Classess
  - Symbol Type
  - Iterators
  - Generators
  - Typed Arrays
  - Promise



Constants

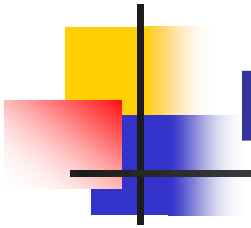
- Constants
  - Support for constants(also knowns as “immutable variables”)
  - Variables which cannot be re-assigned new content
  - This only makes the variable itself immutable, not its assigned content
  - 상수를 선언 할 때는 ‘const’ 로 선언한다

```
const PI = 3.141593
```



Block-Scoping

- Block Scope
  - 일반적으로 블록 스코프란 if, switch, for, while 문 내의 영역
  - { Block Scope }
  - This only makes the variable itself immutable, not its assigned content
  - ECMAScript 6 부터는 식별자(변수, 함수)의 Block-Scoping을 지원한다*
  - let, const로 선언한 변수들은 모두 Block-Scoped가 되고, var로 선언한 변수는 Function-Scoped 임
  - ECMAScript5의 Execution Context의 Scope Chain을 Block으로 확장



Block-Scoping

```
function foo() {
    if ( true ) {
        var i = 1; // exist in function scope
        let j = 2; // exist in block scope
        const SIZE = 100; // exist in block scope
    }
    console.log(i);
    console.log(j);
    console.log(SIZE);
}
```

Block-Scoping

```
(function(){
    let funArr = [];
    for(var i=0; i<10; i++) {
        funArr.push(function(){ return i; });
    }
    for(const f of funArr) {
        console.log(f());
    }
})();
```

```
(function(){
    let funArr = [];
    for(let i=0; i<10; i++) {
        funArr.push(function(){ return i; });
    }
    for(const f of funArr) {
        console.log(f());
    }
})();
```

Arrow Functions

- Arrow functions
  - 화살표 함수 표현(arrow function expression)은 function expression 비해 구문이 짧고 자신의 this 객체를 바인딩 하지 않음
  - this는 상위 스코프의 this (lexical scope)
  - 생성자로 사용할 수 없고, 메소드 함수가 아닌 곳에서 사용

  - 기본 구조

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// 다음과 동일함:  => { return expression; }

// 매개변수가 하나뿐인 경우 괄호는 선택사항:
(singleParam) => { statements }
singleParam => { statements }

// 매개변수가 없는 함수는 괄호가 필요:
() => { statements }
```

Arrow Functions

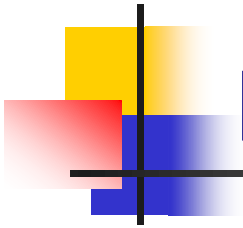
- 고급 구조

```
// 객체 리터럴 표현을 반환하기 위해서는 함수 본문(body)을 괄호 속에
 넣음:
params => ({foo: bar})

// 나머지 매개변수 및 기본 매개변수를 지원함
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) =>
 { statements }

// 매개변수 목록 내 구조분해할당도 지원됨
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f();  // 6
```

객체 구조분해의 새로운 변수 이름으로 할당



Extended Parameter Handling

- Overview
  - 매개변수 기본값
  - Rest 파라미터
  - Spread 문법
  - Rest/Spread 프로퍼티

Extended Parameter Handling

- 매개변수 기본값(Default Parameter Value)
  - 함수를 호출 할 때 매개변수의 개수만큼 인수를 넘겨 주는게 일반적임
  - 매개변수에 해당하는 인자를 넘겨주지 않으면 undefined
  - 인자가 넘어 오지 않을 때 설정되는 기본 값을 매개변수 기본값이라 부른다

```
function sum(x = 1, y = 1) {
    return x + y;
}

foo();
```

Extended Parameter Handling

- Rest 파라미터
  - Rest 파라미터(Rest Parameter, 나머지 매개변수)는 매개변수 이름 앞에 세개의 점 ...을 붙여서 정의
  - 함수에 전달된 인수들의 목록을 배열로 전달받는다
  - Rest 파라미터는 반드시 마지막 파라미터여야 한다

```
function foo(a, b, ...rest) {
    console.log(rest);
}

foo(1, 2, 3, 4, 5);
```

Extended Parameter Handling

- Spread 문법
  - 스프레드 연산자를 사용하면 배열, 문자열, 객체 등 반복 가능한 객체(Iterable Object)를 개별 요소로 분리할 수 있다
  - 스프레드 연산자의 대상은 반드시 이터러블이어야 한다

```
function foo(x, y, z) {
    console.log(x, y, z);
}
foo(...[1,2,3]);

console.log(..."hello");
```

Extended Parameter Handling

- 함수의 인수로 사용하는 경우
  - 일반적으로 배열의 요소를 함수의 인수로 사용하고자 할 때

```
//es5
function foo(x, y, z) { }
var args = [0, 1, 2];
foo.apply(null, args);
```

```
//es6
function foo(x, y, z) { }
var args = [0, 1, 2];
foo(...args);
```

Extended Parameter Handling

- 배열에서 사용하는 경우
  - Spread 문법을 배열에서 사용하면 보다 간결하고 가독성 좋게 표현
  - concat, push, splice, slice 등의 함수를 대체 할 수 있다

1. concat – 기존 배열의 요소를 새로운 배열 요소의 일부로 만들고 싶은 경우

```
//es5
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];
var newArr = arr1.concat(arr2);

//es6
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];
var newArr = [...arr1, ...arr2];
```

Extended Parameter Handling

2. push – 기존 배열에 다른 배열의 개별 요소를 각각 push하기

```
//es5
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

Array.prototype.push.apply(arr1, arr2);

//es6
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];

arr1.push(...arr2); // == arr1.push(4, 5, 6);
```

Extended Parameter Handling

3. splice – 기존 배열에 다른 배열의 개별 요소를 삽입

```
// ES5
var arr1 = [1, 2, 3, 6];
var arr2 = [4, 5];

Array.prototype.splice.apply(arr1, [3, 0].concat(arr2));


// ES6
const arr1 = [1, 2, 3, 6];
const arr2 = [4, 5];

arr1.splice(3, 0, ...arr2);
```

Extended Parameter Handling

- Rest/Spread 프로퍼티
  - 객체 리터럴을 분해하고 병합하는 syntax 제공

```
// 객체 리터럴 Rest/Spread 프로퍼티
// Spread 프로퍼티
const n = { x: 1, y: 2, ...{ a: 3, b: 4 } };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }

// Rest 프로퍼티
const { x, y, ...z } = n;
console.log(x, y, z); // 1 2 { a: 3, b: 4 }
```

Extended Parameter Handling

```
// 객체의 병합
const merged = { ...{ x: 1, y: 2 }, ...{ y: 10, z: 3 } };
console.log(merged); // { x: 1, y: 10, z: 3 }

// 특정 프로퍼티 변경
const changed = { ...{ x: 1, y: 2 }, y: 100 };
// changed = { ...{ x: 1, y: 2 }, ...{ y: 100 } }
console.log(changed); // { x: 1, y: 100 }

// 프로퍼티 추가
const added = { ...{ x: 1, y: 2 }, z: 0 };
// added = { ...{ x: 1, y: 2 }, ...{ z: 0 } }
console.log(added); // { x: 1, y: 2, z: 0 }
```

Template literals

- Template literals
  - 템플릿 리터럴은 내장된 표현식을 허용하는 문자열
  - 복수행 및 문자보간 기능
  - `` 백틱을 사용
  - syntactic sugar

  - 기본 구조

```
`string text`

`string text line 1
 string text line 2`

`string text ${expression} string text`
```

Destructuring assignment

- 구조 분해 할당
  - 배열이나 객체의 속성을 해체하여 그 값을 개별 변수에 담을 수 있게 하는 표현식

  - 기본 구조

```
var a, b, rest;

[a, b] = [10, 20];

[a, b, ...rest] = [10, 20, 30, 40, 50];

({ a, b } = { a: 10, b: 20 });

// Stage 4(finished) proposal
({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});
```

Destructuring assignment

- Array Destructuring
  - 배열 구조 분해

```
// 객체 리터럴 Rest/Spread 프로퍼티
// Spread 프로퍼티
const n = { x: 1, y: 2, ...{ a: 3, b: 4 } };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }

// Rest 프로퍼티
const { x, y, ...z } = n;
console.log(x, y, z); // 1 2 { a: 3, b: 4 }
```

Module

- Module
  - 초기에 사용된 자바스크립트는 작은 단위로 작성됨
  - 현재는 자바스크립트 라이브러리가 많아짐
  - Javascript 생태계 스펙트럼이 매우 넓음
  - 모듈화의 필요성

- 기본 구조 – 모듈 정의, module.mjs

```
export default function foo() {
    console.log('foo');
}

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

export const version = '1.0.0';

const a = 1;
const b = 2;
const c = 3;

export {a, b, c};
```

Module

- 모듈 가져와서 사용하기 – main.mjs

```
// export default에 정의된 함수
import foo from ‘./module.mjs';
// 객체 분할 문법 형태로 모듈 가져와서 사용
import {a, b, c} from ‘./module.mjs';

foo();
console.log(a, b, c);
```

Class

- class
  - **Class**는 객체를 생성하기 위한 템플릿
  - Class 이전에는 생성자 함수를 사용하여 객체를 생성
  - Synthetic sugar를 제공

  - 클래스 정의

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getInfo() {
        return `Name: ${this.name} Age: ${this.age}`;
    }
}
```

Class

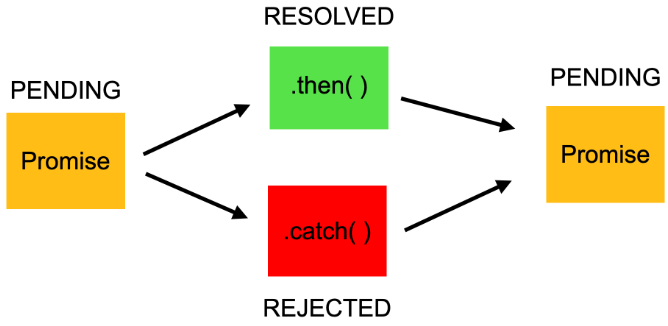
- 클래스 상속

```
class Policeman extends Person {
    constructor(name, age, area) {
        super(name, age);
        this.area = area;
    }
    getInfo() {
        return `Name: ${this.name} Age: ${this.age} Area: ${this.area}`;
    }
}
```

Promise

- 개념
  - 전통적인 비동기 함수의 처리결과는 **callback** 처리
  - The promise object represents the eventual completion(or failure) of an asynchronous operation and its resulting value

- States
  - Pending : initial state, neither fulfilled nor rejected.
  - Fulfilled : meaning that the operation was completed successfully.
  - Rejected : meaning that the operation failed.



Promise

- then
  - 정상적으로 실행되었을 경우 실행, resolved

```
const op = new Promise((resolve, reject)=>{
    setTimeout(() => {
        resolve('done');
    }, 2000);
});

op.then((result) => {
    console.log(result);
});
```

- catch
  - rejected, throw new Exception 발생했을 때 실행되는 구문

```
new Promise((resolve, reject) => {
    setTimeout(() => {
        reject('error occured here');
    }, 2000);
})
.then((result) => {
    console.log(result);
})
.catch(err=>console.log(err));
```