JAVASCRIPT VERSION

ES5 (2009)

This is the baseline version of JS which you can generally assume all run-times (except really old ones!) will support.

ES6 / ES2015

Standard Modules — import and export

Standardised Promises

Classes & Inheritance

Block-scoped variables — letand const

Template Literals

Object destructing into variables

Generator functions

Map and Set data structures

Internationalisation for Strings, Numbers and Dates via Intl API

ES7 / ES2016

Array.includes()

Numeric exponent (power of) operator

ES8 / ES2017

Async Functions

Object.entries

String padding functions

ES9 / ES2018

Object Rest/Spread const obj = { ...props };

Asynchronous Iteration for await (...) {

Promise finally() function

Regular expression enhancements (lookbehind, named groups)

ES10 / ES2019

Array.flat

Array.flatMap

String.trimStart/trimEnd

Object.fromEntries

Optional Catch Binding

Function toString Revision

Symbol Description Accessor

ES11 / ES2020

staging

1. DOM에 대한 이해

웹페이지 개발의 가장 기본적이고 유일한 방법은 DOM API를 이용하여 개발하는 것입니다. 요즘은 jquery와 같은 라이브러리를 사용하면 DOM API를 직접적으로 알지 못하더라도 쉽고 간단하게 웹페이지를 개발할 수도 있습니다. 하지만 jquery도 내부적으로는 DOM API를 사용하고 있습니다. 결국 DOM API를 사용하지 않고서는 어떠한 것도 할수가 없습니다. 개발자가 DOM에 대해 얼마나 이해하고 있는가는 웹페이지의 품질을 결정하는 중요한 요소가 될 것입니다.

DOM에 대한 자세한 내용은 아래 링크에서 학습하시길 바랍니다.

> https://developer.mozilla.org/ko/docs/Web/API/Document_Object_Model/%EC%86%8C%EA%B0%9C

DOM 의 핵심 Interfaces

- `document.getElementById(id)`
- `document.getElementsByTagName(name)`
- `document.createElement(name)`
- `parentNode.appendChild(node)`
- `element.innerHTML`
- `element.style.left`
- `element.setAttribute`
- `element.getAttribute`
- `element.addEventListener`

[Practice.1] 아래 markup으로 작성된 페이지를 DOM API를 이용하여 작성하시오.

```
[sample.html]
<html>
    <body>
        <div id="page">
            <div id="header">
```

```
                Sample Page
            </div>
            <div id="contents">
                <table>
                    <tr>
                        <td>이름</td>
                        <td><input type="text" id="name" name="name"></td>
                    </tr>
                    <tr>
                        <td>성별</td>
                        <td>
                            <input type="radio" id="gender" name="gender" value="m">남
                            <input type="radio" id="gender" name="gender" value="w">여
                        </td>
                    </tr>
                    <tr>
                        <td>취미</td>
                        <td>
                            <select id="hobby" name="hobby">
                                <option value="reading">독서</option>
                                <option value="trip">여행</option>
                            </select>
                        </td>
                    </tr>
                </table>
            </div>
            <div id="footer">
                <button id="save">저장</button>
                <button id="cancel">초기화</button>
            </div>
        </div>
    </body>
</html>
```

2. Javascript 기본

2.1. 자료형

자바스크립트의 기본 타입은 크게 원시 타입과 객체 타입으로 구분할 수 있습니다.

원시 타입(primitive type)

1. 숫자(number)
2. 문자열(string)

3. 참/거짓 값 (boolean)
4. 심볼(symbol) : ECMAScript 6부터 제공됨
5. undefined

객체 타입(object type)

1. 객체(object)

   함수 (Function)

   배열 (Array)

   날짜 (Date)

   정규식 (RegExp)

2.2. Object

Object는 아래와 같이 3가지 방법으로 생성할 수 있습니다.

1. 리터럴 표기(literal notation)를 이용한 방법

```
var 객체이름 = {
    프로퍼티1이름 : 프로퍼티1의값,
    프로퍼티2이름 : 프로퍼티2의값,
    ...
};


var kitty = {
    name: "나비",
    family: "코리안 숏 헤어",
    age: 1,
    weight: 0.1
};
document.write("우리 집 새끼 고양이의 이름은 " + kitty.name + "이고, 종은 " + kitty.family + "이다.");
```

2. 생성자 함수(constructor function)를 이용한 방법

```
var day = new Date(); // new 연산자를 사용하여 Date 타입의 객체를 생성함.
document.write("올해는 " + day.getFullYear() + "년이다.");
```

3. Object.create() 메소드를 이용한 방법

```
Object.create(프로토타입객체[, 새로운객체의프로퍼티1, 새로운객체의프로퍼티2, ...]);


var obj = Object.create(null, {            // null 프로토타입을 사용하여 새로운 객체를 만들고
    x: { value: 100, enumerable: true },    // x좌표를 나타내는 열거할 수 있는 프로퍼티와
    y: { value: 200, enumerable: true }     // y좌표를 나타내는 열거할 수 있는 프로퍼티를 추가
함.
});
obj.x;                      // x좌표
obj.y;                      // y좌표
Object.getPrototypeOf(obj); // 객체의 프로토타입을 반환해 줌.
```

4. Object의 프로퍼티는 아래와 같이 참조 할 수 있습니다.

```
객체이름.프로퍼티이름
또는
객체이름["프로퍼티이름"]


var person = {
    name: "홍길동",      // 이름 프로퍼티를 정의함.
    birthday: "030219",  // 생년월일 프로퍼티를 정의함.
    pId: "1234567",      // 개인 id 프로퍼티를 정의함.
    fullId: function() { // 생년월일과 개인 id를 합쳐서 주민등록번호를 반환함.
        return this.birthday + this.pId;
    }
};
person.name    // 홍길동
person["name"] // 홍길동
```

> 참조: http://www.devkuma.com/books/pages/162

2.3. Function

자바스크립트에서 함수는 아주 중요한 의미를 가집니다. 자바스크립트의 많은 특징들은 함수에 기인한 것들이 대부분이며, 이런 함수를 올바로 이해하지 못하면 제대로된 자바스크립트 코드를 작성할 수 없습니다.

2.3.1. 함수 선언

함수는 아래와 같이 2가지 방식으로 선언할 수 있습니다.

선언식 (Function Declaration)

```
function funcDeclarations() {
  return 'A function declaration';
}
funcDeclarations(); // 'A function declaration'
```

표현식 (Function Expressions)

```
var funcExpression = function () {
    return 'A function expression';
}
funcExpression(); // 'A function expression'
```

위 2가지 방식은 거의 비슷합니다. 차이점은 함수가 생성되는 시점에 있습니다.

선언식의 경우 해당 코드가 로드될 때 생성되고, 표현식의 경우 함수가 실행될 때 생성 됩니다.

```
funcDeclarations(); // 'A function declaration'

function funcDeclarations() {
  return 'A function declaration';
}

funcExpression(); // 'A function expression'

var funcExpression = function () {
    return 'A function expression';
}
```

위 코드를 보면 funcDeclarations()는 제대로 실행되지만, funcExpression() 코드는 아래와 같은 에라가 발생함을 확인 할 수 있습니다.

> Uncaught TypeError : funcExpression is not a function

이는 표현식으로 선언된 funcExpression함수는 7번째 라인이 실행되는 시점에 아직 생성 되지 않아서 발생한 것입니다.

2.3.2. First-Class

함수를 다른 변수와 동일하게 다루는 언어는 **일급 함수**를 가졌다고 표현합니다. 이는 함수를 변수나 배열과 같은 데이터 구조에 에 할당할수 있으며, 다른 함수의 파라미터로 전달 할 수 있으며, 함수를 반환할수 있다는 말입니다. 이는 자바스크립트 프로그래밍의 핵심적인 특징이라고 할수 있습니다.

1. 변수에 함수 할당

```
const foo = function() {
   console.log("foobar");
}
// 변수를 사용해 호출
foo();

const reslut = [];
result.push(foo);   //배열에 저장
```

2. 함수를 인자로 전달

```
function sayHello() {
   return "Hello, ";
}
function greeting(helloMessage, name) {
  console.log(helloMessage() + name);
}
// `sayHello`를 `greeting` 함수에 인자로 전달
greeting(sayHello, "JavaScript!");
```

3. 함수 반환

```
function sayHello() {
   return function() {
      console.log("Hello!");
   }
}
```

> 참조: https://developer.mozilla.org/ko/docs/Glossary/First-class_Function

2.3.3. this

자바스크립트에서 `this`를 이해하는 것은 매우 중요합니다. `this`는 함수가 호출되는 시점에 결정됩니다. 또한 함수를 호출하는 방법에 따라서 `this`는 달라집니다.

자바스크립트에서 함수를 호출하는 방법은 아래와 같이 4가지가 존재합니다.

1. 일반 함수로의 호출
2. 멤버 함수로의 호출
3. call 함수를 이용한 호출
4. apply 함수를 이용한 호출

1. 일반 함수로의 호출

```
function say(something) {
    alert(something);
}
say("ho"); // #1
```

2. 멤버 함수로의 호출

```
var unikys = {
    say: function (something) {
        alert(something);
    }
}
unikys.say("hello"); // #2
```

3. call과 apply로 호출

```
function say(something) {
    alert(something);
}
say.call(undefined, "call ho"); // #3
say.apply(undefined, ["apply ho"]); // #4
```

호출 방법에 따라 this가 결정되는 방법

```
function whatsThis() {
    return this.toString();
}
var unikys = {
    what: whatsThis,
    toString: function () {
        return "[object unikys]";
    }
};
whatsThis(); // #1
unikys.what(); // #2
whatsThis.call(); // #3
whatsThis.apply(unikys); // #4
unikys.what.call(undefined); // #5
unikys.what.apply(unikys); // #6
```

> 출처: https://unikys.tistory.com/306

2.3.4. arguments

자바스크립트에서 함수가 실행될때 `arguments` 객체가 함수 내부로 전달됩니다. 이는 실행된 함수에 전달된 파라미터들을 배열형태로 저장하고 있는 객체입니다. arguments는 배열은 이니지만 배열과 유사한다고 해서 유사 배열 객체(like array)로 부릅니다. 이는 함수에 전달되는 파라미터들의 갯수가 가변 적일때 아주 유용하게 사용할수 있습니다.

```
function print(a){
    console.log(arguments[0]);
}
print("hi");

//arguments는 배열은 아니지만 length 프로퍼티를 가지고 있습니다.
function print_anythings(){
    for(var k=0, len= arguments.length; k < len; k){
        console.log(arguments[k]);
    }
}
print_anythings("apple", "banana", "orange");
```

`arguments`는 배열이 아니기 때문에 `push()`, `concat()`, `join()`등 Array 객체가 가지는 메소드를 가지고 있지 않습니다. 따라서 실무에서는 arugments를 배열로 변환해야 하는 경우가 많습니다. 이를 위한 가장 간단한 방법으로는 slice() 함수를 이용하여 새로운 Array 객체를 반환 받는 방법입니다.

```
//arguments를 배열로 변경하기
function print(){
    arguments.push("mango");    //Error!!
    console.log(arguments.join(","))    //Error!!


    var list = [].slice.call(arguments);    //새로운 배열 반환
    list.push("mango");

    console.log(list.join(","))
}
print("apple", "banana", "orange");
```

2.3.5. bind

`bind()` 함수는 `this`를 고정시키는 함수입니다. 위에서 언급했듯이 자바스크립트에서 `this`는 함수 실행시점에 결정되고, 이는 해당 함수가 어떻게 호출되냐에 따라 달라지게 됩니다. 하지만 `bind()` 함수를 통해 생성된 함수는 언제나 동일한 `this`를 가지게 됩니다.

`bind()` 함수는 첫번째 인자로 전달된 값을 `this`로 바인딩한 새로운 함수를 반환합니다.

```
function print(){
    console.log(this.name);
}
print.call({ name: "apple" });  //apple

var print_banana = print.bind({ name: "banana" });
print_banana.call({ name: "apple" });   //banana
```

이는 주로 콜백함수를 생성할때 사용합니다. 콜백함수는 주로 Event Handler, setTimeout, ajax등 비동기 처리 로직과 함께 사용하게 됩니다. 이때 `bind()` 함수를 이용하게 된다면 콜백 함수가 어떻게 호출 되더라도 `this` 를 원하는대로 지정 할수 있습니다.

```
function openAlert(){
    alert(this.message);
}

//event listener
var errorAlert = openAlert.bind({message: "error"});
document.getElementById("clickDiv1").addEventListener("click", errorAlert);


//ajax callback function
$.ajax({
    url: "/ECERP/API",
    success: openAlert.bind({message: "Success!!"}),
    error: openAlert.bind({message: "Error!!"}),
});
```

2.4. Scope

출처: https://unikys.tistory.com/295

2.4.1 scope의 생성

scope의 생성은 특정 구문이 실행될 때나 객체들이 생성될 때 새롭게 scope가 하나 생성하게 된다. 그 구문들은 다음과 같다.

- function
- with
- catch

```
<div id="divScope0">Click me! DIV 0</div>
<div id="divScope1">Click me! DIV 1</div>
<div id="divScope2">Click me! DIV 2</div>

<script>
    var i, len = 3;
    for (i = 0; i < len; i++) {
        document.getElementById("div" + i).addEventListener("click", function () {
alert("You clicked div #" + i);
        }, false);
    }
</script>
```

2.4.2 scope의 지속성

사실 scope의 생성되는 방식이 기존의 언어와 가장 다른 점은 아니다. 하지만 scope가 지속되는 것은 다른 언어와는 다른 자바스크립트만의 강점 중 하나이다. 이러한 지속성이 자바스크립트에서 필요한 이유는 바로 자바스크립트에서 새로운 scope가 생성되는 '함수'를 변수에 넣을수도 있고, 다른 함수의 인자로 넘겨줄수도 있으며, 함수의 return 값으로도 활용할 수 있기 때문에 필요했던 개념이다. 즉, 지금 함수가 선언된 곳이 아닌 전혀 다른 곳에서도 함수가 호출될 수 있기 때문에, 그 함수의 scope는 지속될 필요가 있었던 것이다.

함수가 생성될때

```
<div id="divScope0">Click me! DIV 0</div>
<div id="divScope1">Click me! DIV 1</div>
<div id="divScope2">Click me! DIV 2</div>

<script>
    function setDivClick(index) {
        document.getElementById("divScope" + index).addEventListener("click", function () {
            alert("You clicked div #" + index);
        }, false);
    }
    var i, len = 3; for (i = 0; i < len; i++) {
        setDivClick(i);
    }
</script>
```

closure와 익명함수

```
<div id="divScope0">Click me! DIV 0</div>
<div id="divScope1">Click me! DIV 1</div>
<div id="divScope2">Click me! DIV 2</div>

<script>
    var i, len = 3;
    for (i = 0; i < len; i++) {
        document.getElementById("divScope" + i).addEventListener("click", (function (index)
{           return function () {
                alert("You clicked div #" + index);
            };
        }(i)), false);
    }
</script>
```

with

```
<div id="divScope0">Click me! DIV 0</div>
<div id="divScope1">Click me! DIV 1</div>
<div id="divScope2">Click me! DIV 2</div>

<script>
    var i, len = 3;
    for (i = 0; i < len; i++) {
        with ({num: i}) {
            document.getElementById("divWith" + num).addEventListener("click", function () {
alert("You clicked div #" + num);
            }, false);
        }
    }
</script>
```

2.5. Prototype

출처: https://mygumi.tistory.com/312 [마이구미의 HelloWorld]

"자바스크립트의 프로토타입(Prototype)은 무엇인가?"

일반적으로 프로토타입(prototype)이란 시제품, 견본 등과 같은 의미로 일반적으로 **원형**이라는 뜻을 가진다.

질문은 "C++, JAVA 에서 Class 란 무엇인가?" 와 같은 맥락이라고 볼 수 있다.

이러한 원천적인 질문의 답은 실질적으로는 "원형" 이라는 일반적인 의미를 벗어나지 않는다.

즉, 자바스크립트에서 프로토타입은 **자신을 만들어낸 객체의 원형**을 뜻한다.

만약 반대로 프로토타입은 자신을 통해 만들어질 객체의 원형이라고 생각하고 있다면, 이건 다음 질문에 대한 답이 된다.

"자바스크립트에서 프로토타입(prototype)은 어떻게 사용되는가?"

실질적인 자바스크립트 사용 질문으로써, 여기서 나타내는 프로토타입은 **자신을 통해 만들어질 객체의 원형**을 뜻한다.

정확히는 용어로는 프로토타입보다는 프로토타입 프로퍼티(prototype property) 를 의미한다.

prototype 프로퍼티란 아래와 같이 실제 코드에서 볼 수 있는 것을 의미한다.

```
function person(){}
person.prototype  //--> Prototype property
```

결과적으로 질문에 따른 프로토타입의 의미는 2 가지로 구분했다.

이러한 이유는 자바스크립트 자체의 프로토타입을 표현하는 용어는 두가지로 나뉘어있기 때문이다.

프로토타입의 2 가지 용어는 다음과 같다.

- Prototype Link - 자신을 만들어낸 객체의 원형
- Prototype Object - 자신을 통해 만들어질 객체의 원형

위와 같은 용어들은 예제 코드를 통해 확인해보자.

```
function person() {
}
console.dir(person);    //콘솔 확인
```

샘플 작성

ecount.control.js

ecount.control.input.js

ecount.control.radio.js

3. 공통함수 만들기 - Module Pattern

일반적으로 모듈이란 기능별로 나뉘어지는 프로그램을 이야기 합니다. 조금더 자세하게 설명하자면 '기능 구현의 세부사항은 숨겨두고(encapsuration), 특정 API를 노출해(public) 다른 프로그램에서 쉽게 재사용 할수 있게 설계된 코드'를 이야기 합니다. 자바스크립트에서 모듈은 주로 `namespace`, `즉시 실행 함수`와 `클로저(Clouser)`를 이용하여 작성됩니다.

3.1. namespace

자바스크립트에서는 java의 package나 c#의 namespace와 같은 스펙이 없습니다. 하지만 전역변수를 기초로 하는 javascript의 언어적 한계(전역 변수의 오염) 때문에 위와 같은 기능은 자바스크립트 개발에서 꼭 필요한 부분입니다. 이를 해결하기 위한 방법은 여러가지가 있지만 가장 일반적으로 사용되는 방법은 Object Literal을 이용한 NameSpacing 방법입니다.

이는 하나의 전역 객체를 생성한후 모든 변수, 객체, 함수 등을 이 전역 객체에 추가하는 방법입니다.

```
var ecount = ecount || {};
ecount.version = "1.0"
ecount.common.api = function(){}
```

위와 같은 방법으로 여러 스크립트가 한 페이지 안에 함께 로드 되더라도 전역 변수가 오염되는 것을 방지 할수 있습니다. **물론 어디에선가 ecount = {};와 같은 코드를 작성한다면 이 모든게 의미가 없게 됩니다.** 이런 상황은 다수의 개발자가 함께 개발하다 보면 언제든지 발생할수 있는 상황이기 때문에 항상 주의해야 합니다.

3.2. 즉시 실행 함수(IIFE)

자바스크립트의 고질적인 문제는 글로벌 영역에 정의된 것은 어디서든지 접근/변경이 가능하는 것입니다. 이는 다수의 인원이 함께 개발해야 하는 상황에서는 자칫 예기치 못한 오류가 발생하는 원인이 되곤 합니다.

이때 `즉시 실행 함수(Immediately-invoked function expression)`를 사용하게 되면 외부에서는 접근하지 못하는 변수를 할당 할수 있습니다.

`즉시 실행 함수`는 말 그대로 생성된 즉시 실행되는 함수를 말합니다. 이때 `즉시 실행 함수` 내에 선언된 변수는 외부에서 접근 할수 없습니다. 이런 특징을 이용해서 특정 변수를 외부 접근으로 부터 보호할 수 있습니다.

```
//()는 함수를 실행하는 연산자
var test = function(){ console.log("test") }; //선언
test();  //실행

(function(){ console.log("test") })(); // 선언 -> 실행

(function(){
    var version = 1;    //함수 외부에서는 접근이 불가능
})();
```

`즉시 실행 함수` 내에 정의된 변수에 접근 할수 있는 방법은 바로 `클로저`를 이용하는 것입니다.

3.3. 클로저(Clouser)

클로저란, 이미 생명 주기가 끝난 외부 함수의 변수를 참조하는 함수를 말합니다.

 # sample1

```
function outer () {
  var count = 0;
  var inner = function () {
    return ++count;
  };
  return inner;
}
var increase = outer(); //8번째 라인이 실행되면 count 변수는 메모리에서 해제되어야 한다.

console, log(increase()); // === 1 //inner함수가 count 참조하고 있어서 해제 되지 않는다.
console, log(increase()); // === 2
```

 # sample2

```
function outer () {
  var count = 0;
  return {
    increase: function(){
      return ++count;
    },
    decrease: function(){
      return --count;
    }
  };
}

var counter = outer();
console, log(counter.increase()); // === 1
console, log(counter.increase()); // === 2
console, log(counter.decrease()); // === 1

var counter2 = outer();
console.log(counter2.increase()); // === ?
```

sample3

```
var countFactory = (function () (
  var staticCount = 0;
  return function () {
    var localCount = 0;
    return {
      increase: function () {
        return {
          static: ++staticCount,
          local: ++localCount
        };
      },
      decrease: function () {
        return {
          static: --staticCount,
          local: --localCount
        };
      }
    }
  }
}()) ;

var counter = countFactory(), counter2 = countFactory();
console.log(counter.increase());
console.log(counter.increase());
console.log(counter2.decrease());
console.log(counter.increase());
```

> 참조: 도서 인사이트 자바스크립트

[Practice.2] 아래 코드가 동작하도록 ecBind 함수를 구현하시오.

```
Function.prototype.ecBind = function(){
    // 코드를 구현하시오.
}

var log = function(){
    console.log([].slice(arguments).join(" , "));
}

var ecLog = log.ecBind("a", "b", "c");
ecLog(1,2,3);   //a , b , c , 1 , 2 , 3
ecLog("A", "B", "C", "D", "E"); //a , b , c , A , B , C , D , E
```

[Practice.3] 아래 코드가 동작하도록 ecount.lib.js 코드를 작성하시오.

```
<html>
    <body>
        <div id="page">
            <div id="header"></div>
            <div id="contents"></div>
            <div id="footer"></div>
        </div>
        <script src="./ecount.lib.js"></script>
        <script>
            //#header
            var input = ecount.lib.createElement({
                type: "text",
                text: "Sample Page"
            });
            ecount.lib.render("#header", input);

            //#contents
            var table = ecount.lib.createElement({
                type: "table",
                items: [
                    {
                        type: "input",
                        title: "이름",
                        id: "name",
                        name: "name"
                    },{
                        type: "radio",
                        title: "성별",
                        items: [{id: "gender", name: "gender", value: "m", text: "남"},
                                {id: "gender", name: "gender", value: "w", text: "여"}]
                    },{
                        type: "select",
                        title: "취미",
                        id: "hobby",
                        name: "hobby",
                        option: [{value: "reading", text: "독서"},
                                 {value: "trip", text: "여행"}]
                    }
                ]
            });
            ecount.lib.render("#contents", table);

            //#footer
            var save = ecount.lib.createElement({
                type: "button",
                id: "save",
                text: "저장",
                click: function(){
                    alert(this.text);   //저장
                }
```

```
            });
            ecount.lib.render("#footer", save);

            var cancel = ecount.lib.createElement({
                type: "button",
                id: "cancel",
                text: "초기화",
                click: function(){
                    alert(this.text);   //초기화
                }
            });
            ecount.lib.render("#footer", cancel);

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

4. Class

클래스(class)를 만드는 목적은 여러 가지가 있습니다. 그중 중요한 것 중 하나가 **데이터**를 다루기 입니다. 동일한 클래스로 생성된 여러 객체(instance)들은 동일한 기능(method)을 가지고, 각기 다른 데이터(data, state)를 취급하게 됩니다. 소프트웨어란 결국 `데이터`를 다루는 것입니다. 최근에 널리 사용되는 ReactJS나 VueJS등과 같은 UI라이브러리들도 결국 순수한 UI 요소와 UI의 상태나 값을 나타내는 `데이터`를 나누고, `데이터`를 간단하게 조작함으로써 복잡한 UI 로직을 제어하기 위한 목적으로 만들어진 것입니다. 이도 곧 `데이터`를 다루는 것으로 볼수 있습니다. 더구나 이카운트와 같은 ERP 프로그램은 결국 고객들이 입력하는 `데이터`를 취급하는데 그 목적이 있습니다. 따라서

자바스크립트에서 말하는 `class`는 일반적으로 **함수**를 의미합니다. 최근 ES6 문법에 추가된 `class` 도 사실 우리가 일반적으로 알고있는 class 라기 보다는 **조금 더 특별함 함수**에 지나지 않습니다. 일반적으로 자바스크립트는 `prototype`을 이용하여 클래스를 정의합니다만, 지금 부터는 개발의 편의성을 위해 ES6 문법을 사용하여 코드를 작성하도록 하겠습니다.

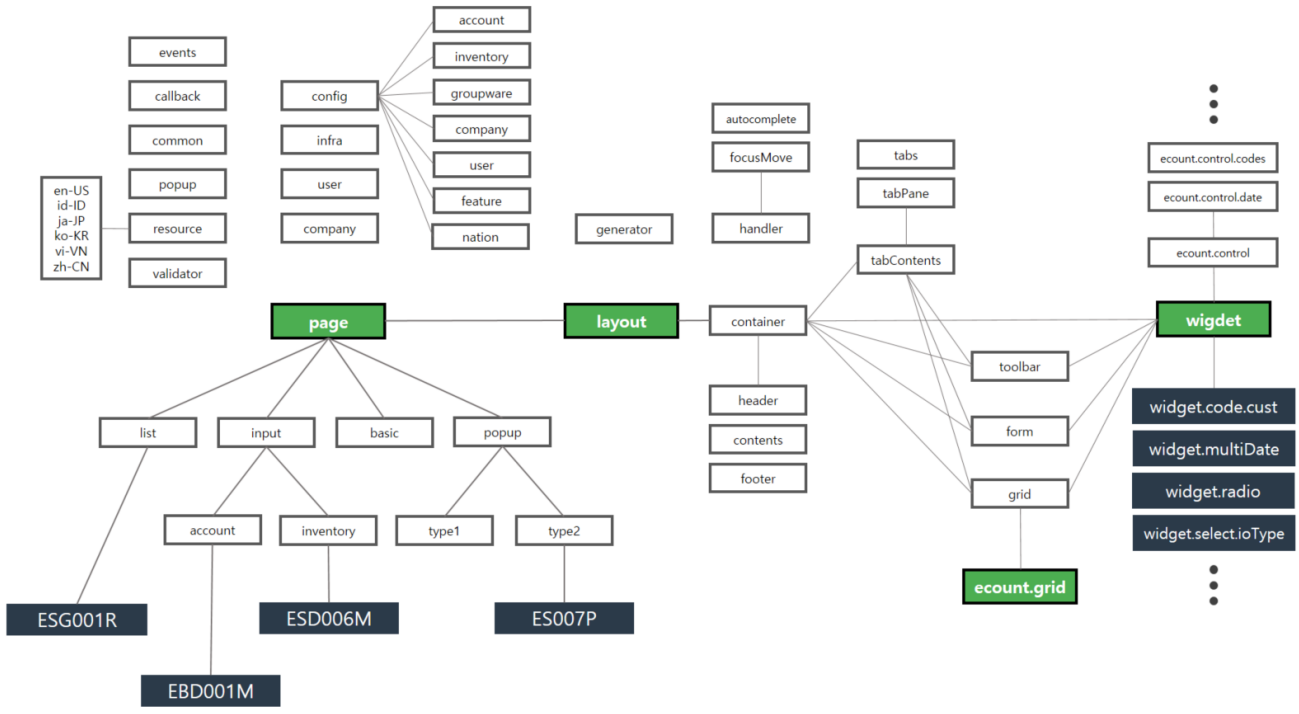
```
class Person {

}
```

5. 이카운트 UI 맛보기

앞서 우리가 개발했던 페이지는 현재 이카운트 ERP에서 쉽게 볼수 있는 형태의 화면입니다. 이카운트 ERP 페이지는 `헤더`, `컨텐츠`, `푸터` 영역이 나누어져 있고, 각 영역에는 다양한 `위젯`을 조합하여 여러 형태의 UI를 구성할 수 있습니다.

이카운트 ERP UI는 아래와 같이 구성 되어져 있습니다.



`ESG001R.js` : 판매현황 업무 로직을 개발하는 페이지

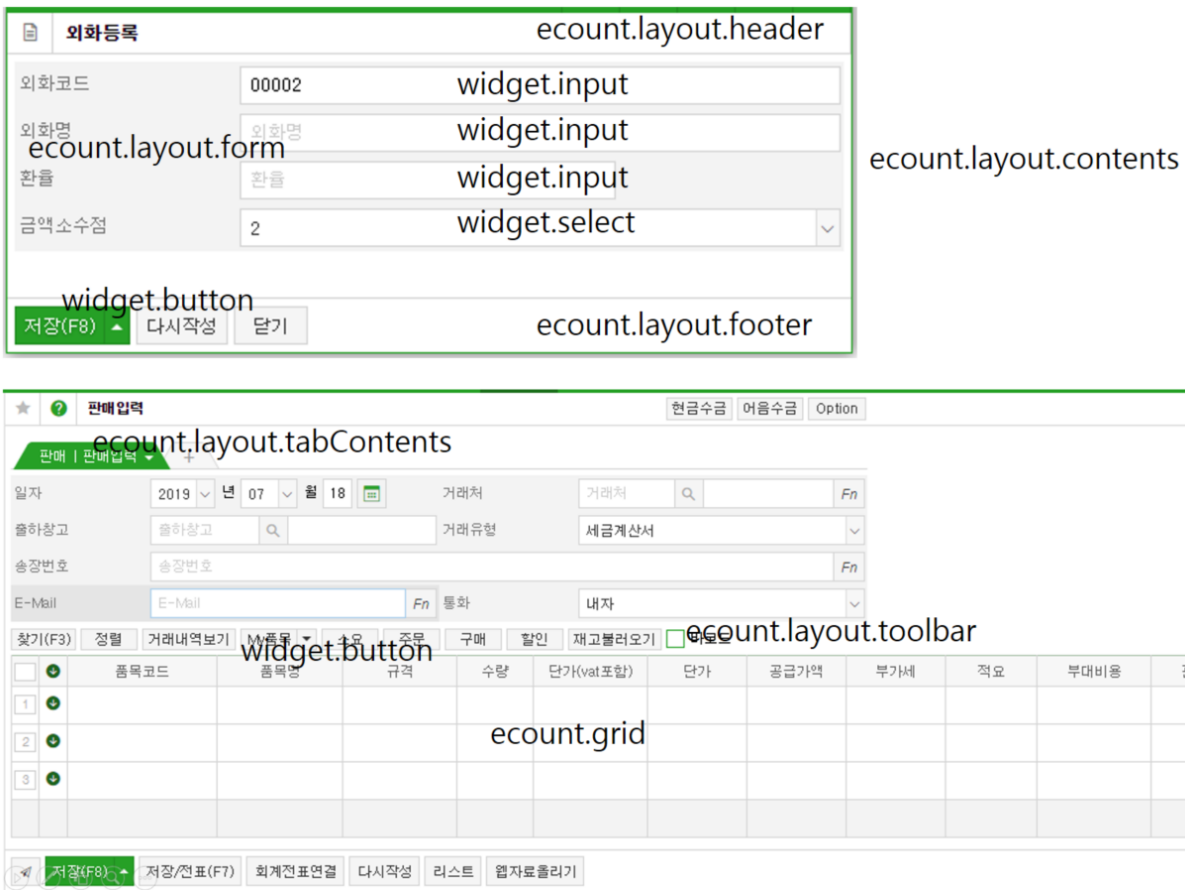
`ecount.page.list.js` : 조회 페이지 공통 로직을 개발하는 페이지

`ecount.page.js` : ERP 모든 페이지의 공통 로직을 개발하는 페이지

`ecount.layout.js` : 컨테이너 역할을 하는 UI 위젯

`ecount.control.js` : 기능을 가지는 가장 작은 단위의 UI 위젯

실제 ERP 화면을 살펴보면 아래와 같이 구성되어 있음을 알수 있습니다.



이카운트 ERP는 수천개의 페이지가 서비스 되고 있으며, 약 백여명의 개발자에 의해 매일 매일 업데이트가 되고 있습니다. 이런 상황 속에서 최상의 서비스를 하기 위해서는 무엇보다 코드의 표준화, 기능의 공통화등이 아주 중요합니다. 이에 따라 이카운트에서는 지속적인 업무 페이지 공통화 작업 및 다양한 위젯 기능 개발을 통하여 코드의 품질을 유지하면서 다양한 변화에 신속하게 대응할 수 있는 시스템을 만들어 가고 있습니다.

지금 부터는 이카운트 ERP UI를 구성하는 기본 `control` , `layout` , `page` 위젯들을 간단하게 개발해 보면서 기본적인 UI 개발 컨셉과 각 페이지별 UI가 동작하는 큰 흐름에 대해 알아 보겠습니다.

5. Control

앞서 우리가 개발한 `ecount.lib` 모듈을 사용하면 동일한 모양을 가지는 Element는 만들수 있겠지만, 각각의 Element가 가지는 상태 정보나 데이터들은 해당 모듈 자체에서 관리 하기가 어렵습니다. 이런 이유로 이카운트에서는 기능과 모양을 가지는 최소한의 UI 단위를 Control이라 정의하고 해당 클래스를 개발하게 되었습니다.

`푸터` 영역을 구성하는 아래의 코드는 `ecount.control.button` 클래스를 이용하여 변경될수 있습니다.

```
//ecount.control 클래스르 이용하여 버튼 생성
var saveInstance = new ecount.control.button({
    type: "button",
    id: "save",
    text: "저장",
    click: function(){
        alert(this.text);   //저장
    }
});
saveInstance.render("#footer");
```

[Practice.4] ecount.control을 상속받는 input, button, radio, select 클래스를 작성하시오.

6. Layout

단순하게 `Control`을 나열 하는 것 만으로는 다양한 UI를 구성할 수 없습니다. 우리에겐 `Control`을 화면 요소 요소에 배치하고, 여러개의 `Control`을 통합 관리 할수 있는 요소가 필요합니다. 이런 컨테이너 역할을 하는 위젯이 바로 `Layout` 입니다.

앞서 우리가 작성했던 코드를 보면 `Table Element`가 `Layout` 위젯과 비슷한 역할을 하고 있습니다. `Table`은 `input`, `radio`등과 같은 Element들을 화면에 배치하는 역할을 합니다. 여기서 더 나아가 우리가 작성하게될 `ecount.layout.form` 위젯은 단순한 `control`의 배치 뿐만 아니라, 해당 `Layout` 위젯에 포함된 `control` 의 인스턴스를 관리하는 역할까지 하게 될것 입니다.

```
var form = new ecount.layout.form({
    items: [
        {
            type: "input",
            title: "이름",
            id: "name",
            name: "name",
             value: "이카운트"
        },{
            type: "radio",
            title: "성별",
            items: [{id: "gender", name: "gender", value: "m", text: "남", seledcted: true},
                    {id: "gender", name: "gender", value: "w", text: "여"}]
        },{
            type: "select",
            title: "취미",
```

```
            id: "hobby",
            name: "hobby",
            option: [{value: "reading", text: "독서"},
                    {value: "trip", text: "여행",  seledcted: true}]
        }
    ]
});
form.render("#contents");
```

[Practice.5] 아래 코드가 동작하도록 getControl(), setValue(), reset() 메소드를 구현하시오.

```
//아이디가 name인 control의 value 값을 ECOUNT로 변경한다.
form.getControl("name").setValue("ECOUNT");

//form에 있는 모든 컨트롤의 값이 초기화 된다(빈값)
form.reset();
```

[Practice.6] ecount.layout을 상속받는 header, contents, footer 클래스를 작성하시오.

```
<html>
    <body>
        <div id="page">
            <div id="header"></div>
            <div id="contents"></div>
            <div id="footer"></div>
        </div>
        <script src="./ecount.lib.js"></script>
        <script>
            var header = ecount.layout.header({
                title: "Sample Page"
            });
            header.render();

            var contents = ecount.layout.contents({
                type: "table",
                items: [
                    {
                        type: "input",
                        title: "이름",
                        id: "name",
                        name: "name"
                    }, {
```

```
                        type: "radio",
                        title: "성별",
                        items: [{id: "gender", name: "gender", value: "m", text: "남"},
                                {id: "gender", name: "gender", value: "w", text: "여"}]
                    }, {
                        type: "select",
                        title: "취미",
                        id: "hobby",
                        name: "hobby",
                        option: [{value: "reading", text: "독서"},
                                {value: "trip", text: "여행"}]
                    }
                ]
            });
            contents.render();

            var footer = ecount.layout.footer({
                items: [
                    {
                        type: "button",
                        id: "save",
                        text: "저장",
                        click: function(){
                            alert(this.text);   //저장
                        }
                    }, {
                        type: "button",
                        id: "cancel",
                        text: "초기화",
                        click: function(){
                            alert(this.text);   //초기화
                        }
                    }
                ]
            });
            footer.render();
        </script>
    </body>
</html>
```

7. Page

우리는 지금까지 `HTML`로 구성된 페이지를 `DOM API`를 사용해 변경하고, 이를 좀더 `모듈화` 하는 작업을 거쳐 `Control`과 `Layout` 클래스를 만들고 이를 사용하여 UI를 구성하는 작업을 했습니다.

앞서 잠깐 이야기 한대로 이카운트 ERP에는 이런 페이지들이 수천개가 존재하고, 대부분의 페이지는 우리가 작성한 것보다 훨씬더 다양한 UI를 가지고 있습니다. 그 뿐만 아니라 실제 페이지에는 UI 관련 코드와 수많은 업무 로직들이 혼재하여 더욱 복잡 합니다. 이렇게 다양하고 복잡한 각각의 페이지들을 좀더 쉽고 효율적으로 개발할 수 있는 대안이 필요했고 그 결과 `Page` 클래스가 생성 되었습니다. 현재, 이카운트 ERP의 모든 페이지는 `Page` 클래스를 상속 받아 개발하도록 되어 있습니다.

이카운트 ERP UI 코드를 보면 크게 2가지 목적의 코드가 존재합니다. `UI를 구성,조작하는 코드`와 `업무 관련 코드`입니다. 다행이 우리는 이미 `Control`과 `Layout` 클래스를 개발하여 `UI를 구성,조작하는 코드`에 관련된 로직들은 일정부분 표준화를 이루었습니다. 하지만 아직 끝난 것은 아닙니다.

7.1. Lifecycle

`Page` 클래스의 역할중 하나는 `UI`를 구성하고 로드하는 페이지의 `Lifecycle` 을 관리하는 것입니다. `헤더` 영역을 정의하는 코드는 `onInitHeader()` 메소드를 구현하여 정의 할수 있습니다. 마찬가지로 `컨텐츠`, `푸터` 영역에 해당 되는 코드들은 `onInitContents()`, `onInitFooter()` 메소드에 각각 구현하면 됩니다. 이렇게 제각각 산재해 있던 UI 관련 로직들을 특정 메소드에 정의하게 함으로써 모든 페이지의 `UI 관련` 로직들을 표준화 할수 있게 되었습니다. 또한 `init()`, `onLoadComplete()` 처럼 기본적으로 필요한 페이지의 `Lifecycle` 관련 메소드를 정의 하게 함으로써 복잡한 페이지의 로직들을 좀더 통일성 있게 작성할수 있게 되었습니다.

```
var page = new ecount.page({
    init: function(){

    },
    onInitHeader: function(){
        var header = ecount.layout.header({
            title: "Sample Page"
        });
        header.render();
    },
    onInitContents: function(){
        //contents 구성
    },
    onInitFooter: function(){
        //footer 구성
    },
    onLoadComplete: function(){
        alert("Loaded!!")
    }
});
```

[Practice.7] 위 코드가 동작하도록 ecount.page 클래스를 작성하시오.

7.2. Event Handler

이제 한가지 남은 것은 `업무` 관련 로직들을 정리 하는 것입니다. ERP 프로그램은 기본적으로 사용자의 인터렉션에 기반하여 동작합니다. 저장 버튼을 클릭하면 데이터를 저장하고, 엔터를 누르면 포커스가 이동되는 등 각 페이지는 다양한 사용자의 동작에 반응하여 `업무` 로직을 실행해야 합니다. 이에 `Page`는 해당 페이지의 각 `Control`에서 발생하는 다양한 이벤트에 대응할 수 있는 메소드를 제공합니다.

```
var page = new ecount.page({
    ...
    onFooterSave: function(){
        //footer에 있는 Control중 id가 save인 control에서 클릭 이벤트가 발생할때 실행
    },
    onChangeControl: function(){
        //페이지에 있는 Control에서 change 이벤트가 발생할때 실행
    }
});
```

[Practice.8] 위 코드가 동작하도록 ecount.page 클래스를 작성하시오.

7.3. Generator

이제 ERP의 모든 페이지들이 일정한 형식을 갖추어 상당 부분 표준화가 되었습니다. 하지만 각 페이지를 보면 아직 반복적으로 생상되고 있는 코드들이 남아 있습니다.

```
var page = new ecount.page({
    onInitHeader: function(){
        var header = ecount.layout.header({
            title: "Sample Page"
        });
        header.render();
    },
    onInitContents: function(){
        //contents 구성
    },
    onInitFooter: function(){
        var footer = ecount.layout.footer({
            items: [
                {
                    type: "button",
                    id: "save",
                    text: "저장",
                    click: function(){
```

```
                        alert(this.text);   //저장
                    }
                }, {
                    type: "button",
                    id: "cancel",
                    text: "초기화",
                    click: function(){
                        alert(this.text);   //초기화
                    }
                }
            ]
        });
        footer.render();
    }
});
```