Chapter 03. Node의 자바스크립트와 친해지기

* 자바스크립트의 객체
* 자바스크립트에서 객체를 생성할 때는 { } 중괄호를 사용하며 🡪 { } 객체 안에는 ‘속성’을 추가할 수 있다.
* 객체 안의 속성은 하나의 ‘변수’이며, name / value로 구분된다.
* 각 속성은 .name 으로 접근하거나 [ ] 대괄호를 이용해 접근한다.

[ch03\_test2.js]

* var Person = { }; Person[‘age’] = 20; Person.mobile = ‘0102132’;
* 자바스크립트의 함수
* JS의 경우 JAVA와 달리 리턴 값 자료형을 표시하지 않고, **변수에 함수를 할당이 가능**

1. 일반 정의(선언문)

function add(a,b){ return a+b; }

1. 익명함수 정의(표현식이므로 ; 붙여준다. 변수에 함수를 할당하는 방식)

var add = function(a,b){ return a+b; };

* 변수에 함수를 할당 가능하므로, 결국 **변수에 할당된 함수를 속성에 할당할 수 있다**

[ch03\_test5.js]

var Person = {};

Person['age'] = 20;

Person['name'] = '아이즈원';

Person.add = function(a, b){

return a + b;

};

console.log('더하기 : %d', Person.add(10, 11));

* 또는, 함수를 변수에 먼저 할당한 후🡪해당 변수를 객체 속성으로 할당도 가능

[ch03\_test6.js]

var Person = {};

Person['age'] = 20;

Person['name'] = '프로미스나인';

var oper = function(a, b){

return a + b;

};

Person['add'] = oper;

console.log('더하기 : %d', Person.add(10, 50));

* 더불어, 객체를 만들 때 { } 안에 속성 바로 할당하는 것도 가능하다. 이 때 name과 value는 콤마( : )를 통해 구분한다.

[ch03\_test7.js]

var Person = {

age : 20,

name : '아이즈원',

add : function(a, b){

return a + b;

}

};

console.log('더하기 : %d', Person.add(10, 53));

* 배열
* 배열은 여러 데이터를 ‘하나의 변수’에 담아 둘 수 있는 구조로 각 Item(요소)는 [ ]를 통해 접근이 가능
* [ ]를 이용해 배열을 이용해 생성하고, push( )메소드로 Item을 추가할 수 있다.

var Users = [

{name : '아이즈원', age : 20},

{name : '트와이스', age : 21}

];

Users.push({name : '프로미스9', age : 22});

console.log('그룹의 수 : %d', Users.length);

console.log('첫 번째 그룹의 이름 : %s', Users[0].name);

* ‘변수에는 함수를 할당’할 수 있으므로, 배열의 Item으로 함수를 할당할 수 있다.

var Users = [

{name : '아이즈원', age : 20},

{name : '트와이스', age : 21}

];

var add = function(a, b){

return a + b;

};

Users.push(add);

console.log('배열 요소의 수 : %d', Users.length);

console.log('세 번째 요소 함수 실행 : %d', Users[2](11, 20));

* 단, C나 JAVA에서는 배열에 함수를 넣어두는 방식 불가. 주의!
* 배열의 모든 요소 출력(for, forEach)

var Users = [

{name : '아이즈원', age : 20},

{name : '트와이스', age : 21},

{name : '프로미스9', age : 22}

];

console.log('배열 요수의 수 : %d', Users.length);

for(var i = 0; i < Users.length; i++){

console.log('배열 요소 #' + i + ' : %s', Users[i].name );

}

console.log('\nforEach 구문 사용');

Users.forEach(function(item, index){

console.log('배열 요소 #' + index + ': %s', item.name);

});

* 94~99 page는 향후 배열 연산 사용 시 다시 학습
* Callback Function 이해하기
* ‘함수’를 ‘parameter’로 전달하는 방식은 대부분 Non-Blocking Programming 방식에서 코드 작성시 사용한다. **Parameter로 전달되는 함수를 Callback Function이라 칭한다.**

1. **Blocking 방식 과 Non-Blocking 방식 비교**

// Blocking 방식

var content = file.read('a.txt');

// 대기

doShow(content);

// 다음 작업

var result = doAdd(10, 10);

// Non-Blocking 방식

file.read('a.txt', function(content){

doShow(content);

});

// 다음 작업

var result = doAdd(10,10);

위와 같이, 콜백 함수를 사용해 doAdd(10, 10)을 file.read( ) 메소드 작업 완료와 관계없이 바로 실행할 수 있도록 하고, file.read( )작업이 완료되면 doShow(content)를 실행할 수 있다. Callback Function을 사용하는 메소드일 경우, 해당 callback을 파라미터로 사용할 때 해당 메소드에 사전에 callback function에서 사용하는 parameter에 대한 정의가 사전에 선언 되어 있다고 이해하면 좋다.

1. **Callback Function 직접 세부 정의해보기**

function add(a, b, callback){

var result = a + b;

callback(result);

}

add(10, 10, function(result){

console.log('파라미터로 전달된 callback 호출');

console.log('add(10, 10)의 결과 : %d', result);

});

**위와 같이 메소드 add는 인수 a, b와 callback 함수를 parameter로 사용하는 함수이다. 해당 함수를 실행하면 a와 b를 더한 값을 변수 result에 대입한 후 result를 parameter로 사용하는 callback function을 실행하게 된다.**

* **Add(10, 10, function(result){….와 같은 형태로 사용하게 되는 것!!!!**

1. **함수 안에서 값을 반환할 때, 새로운 함수를 만들어 반환해보기**

function add(a, b, callback){

var result = a + b;

callback(result);

var history = function(){

return a + '+' + b + '=' + result;

};

return history;

}

// 즉, add 함수를 실행하면 result를 parameter로 사용하는 callback이 실행되고

// history라는 변수를 return한다

var add\_history = add(11, 11, function(result){

console.log('파라미터로 전달된 콜백 함수 호출됨!');

console.log('11더하기 11의 값 : %d', result);

});

console.log('결과 값으로 받은 함수 실행 결과 : ' + add\_history());

* 실행결과   
  파라미터로 전달된 콜백 함수 호출됨!  
  11더하기 11의 값 : 22  
  add메소드 실행한 후 return 받은 값 : 11+11=22
* 현 단계에서 이해 안되는 부분  
  > 마지막 줄에서 add\_history( ) 를 출력하면 11+11=22가 출력되고  
   add\_histrory로 출력하면 a+ ‘+’…. result; 텍스트만 그대로 출력.  
  > 아마 add\_history는 단순 변수가 아닌 메소드이기 때문에 그런 듯 한데..

1. **함수가 반환된 이후에도 접근 가능한 경우**

function add(a, b, callback){

var result = a + b;

callback(result);

var count = 0;

var history = function(){

count++;

return count + ':' + a + '+' + b + '=' + result;

};

return history;

}

var add\_history = add(10, 10, function(result){

console.log('파라미터로 전달된 콜백함수 호출됨!');

console.log('더하기(10,10)의 결과 : %d', result);

});

console.log('결과 값으로 받은 함수 실행 결과 : ' + add\_history());

console.log('결과 값으로 받은 함수 실행 결과 : ' + add\_history());

console.log('결과 값으로 받은 함수 실행 결과 : ' + add\_history());

* 실행 결과

파라미터로 전달된 콜백함수 호출됨!

더하기(10,10)의 결과 : 20

결과 값으로 받은 함수 실행 결과 : 1:10+10=20

결과 값으로 받은 함수 실행 결과 : 2:10+10=20

결과 값으로 받은 함수 실행 결과 : 3:10+10=20

* **주의! 반환된 history함수가 실행될 때는 이미 add 함수가 메모리에서 접근할 수 없으며, count 변수에도 접근할 수 없어야 한다. 하지만 함수 안에서 새로운 함수를 만들어 반환하는 경우 예외적으로 접근을 허용하며, 이를 클로저(Closure)라고 한다.**
* 프로토타입 객체 만들기
* **함수도 객체이기 때문에, 함수를 이용해 객체를 만들어 낼 수도 있다.**Person = { }; 이 형태 뿐만 아니라, 함수 형태로도 객체 생성이 가능하다.
* JS에서도 객체 프로토 타입 생성하여 인스턴스 생성이 가능함

// Person 함수(객체)의 constructor. 속성 name과 age를 부여한다.

function Person(name, age){

this.name = name;

this.age = age;

}

// Person 함수(객체)에서 자동으로 생성되는 prototype속성을 사용해

// walk 함수를 정의한다.

Person.prototype.walk = function(speed){

console.log(speed + 'km의 속도로 걸어간다.');

}

var person01 = new Person('아이즈원', 20);

var person02 = new Person('트와이스', 21);

console.log(person01.name + '객체의 walk(10)을 호출');

person01.walk(10);

* 결과  
  아이즈원객체의 walk(10)을 호출  
  10km의 속도로 걸어간다.
* walk 함수 정의 시, Person.prototype.walk 와 Person.walk로 각각 정의해도 결과는 같지만 프로토타이핑 한 함수가 메모리 관리에 효율적이다.
* 즉, Person이라는 프로토타입 객체는 속성 name, age, constructor(생성자), prototype(프로토타입 + walk 함수) 을 갖는다.