객체지향 프로그래밍



**정의**

*객체지향 프로그래밍(Object Oriented Programming)은 문제를 여러 개의 객체 단위로 나눠 작업하는 방식을 말합니다.*

*이 방식은 오늘날 가장 많이 사용하는 대표적인 프로그래밍 방식이고 JAVA, C# 등이 대표적인 객체지향 프로그래밍 언어입니다.*

**객체지향 프로그래밍 특징 및 장점**

*객체지향 프로그래밍의 가장 큰 특징은 클래스를 이용해 연관 있는 처리부분(함수)과 데이터 부분(변수)를 하나로 묶어 객체(인스턴스)를 생성해 사용한다는 점입니다.*

*그리고 객체지향 프로그래밍에서는 프로젝트를 독립적인 객체 단위로 분리해서 작업할 수 있기 때문에  여러 개발자와 협업해 규모가 큰 프로젝트를 진행할 수 있으며 유지보수측면도 뛰어나다는 장점이 있습니다.*

객체지향 프로그래밍 VS 클래스 기반 프로그래밍

클래스 단위 프로그래밍과 객체지향 프로그래밍이 똑같은 내용 아닌가 하고 생각할 수도 있습니다.

결론부터 말씀드리자면 일부는 맞고 일부는 맞지 않습니다.

엄격한 의미에서 해석

일반적으로 객체지향 프로그래밍은 객체지향 프로그래밍의 네 가지 특징(추상화, 캡슐화, 상속, 다형성)을 골고루 사용해 코딩 하는 프로그래밍을 객체지향이라고 합니다. 클래스 단위 프로그래밍은 이런 객체지향 특징을 전혀 사용하지 않고 오직 클래스만을 이용해 코딩 하는 방식을 말합니다. 이와같이 이 둘은 전혀 다른 프로그래밍 방법입니다.

유연한 의미에서 해석

클래스 단위 코딩을 객체지향 프로그래밍이라고 부를 수 있는 이유는 클래스를 만들어 코딩 하는 자체에 이미 객체지향 프로그래밍 네 가지 특징 중 캡슐화와 은닉성을 사용하고 있기 때문입니다. 이처럼 클래스 기반 프로그래밍은 객체지향 프로그래밍에 포함되는 구조를 가지고 있습니다.

그리고 객체지향 프로그래밍을 배우려면 우선 클래스 기반 프로그래밍에 익숙해져 있어야 한다는 점입니다. 이유는 객체를 생성하는 수단이 클래스(생성자함수)이기 때문입니다.

|  |  |  |
| --- | --- | --- |
| 구   분 | 절차 지향 프로그래밍 | 객체 지향 프로그래밍 |
| 처리 방식 | 문제를 여러 개의 함수로 나누어 순차적으로    호출하여  처리하는 방식 | 문제를 여러 개의 객체 단위로 나누어 처리하는 방식 |
| 단 점 | 오래된 방식이며 협업해서 진행하는 큰    프로젝트에는 적합하지 않음. | 학습 난이도가 높음.     개발자의 활용 능력이 중요함. |
| 장 점 | 간단하고 소규모 프로젝트에 용이함.     비교적 배우기 쉬음. | 최근 가장 많이 사용하는 방식임.     협업이 중요한 대형 프로젝트에 적합함. |

객체란?

물리적으로 존재하거나(자동차, 책, 사람), 추상적인 것(회사, 날짜) 중에서 자신의 속성과 동작을 가지는 모든 것을 말합니다.

객체는 속성(필드)과 메소드(동작)로 구성된 자바 객체로 모델링할 수 있다.

[객체간의 관계]

객체 지향 프로그램에서는 객체는 다른 객체와 다른 객체와 관계를 맺고 있다.

집합 관계 : 완성품과 부품의 관계

사용 관계 : 객체가 다른 객체를 사용하는 관계

상속 관계 : 종류 객체와 국체적인 사물 객체 관계

[객체지향 프로그래밍]

부품객체를 만들고 이것들은 하나씩 조립해서 완성된 프로그래을 만드는 기법

ex) 슬라이딩 배너 (next,prev 버튼, auto 기능, 정지버튼 기능등의 부품 객체를 모두 조립하여 하나의 슬라이딩 객체가 완성

출처: <http://webclub.tistory.com/155> [Web Club]

자바스크립트 객체지향 프로그래밍

자바스크립트를 객체지향 프로그래밍이라고 부를 수 있는 것과 관계된 것 중에 생성자, 프로토타입 인스턴스, 프로퍼티와 메서드, 클래스 프로퍼티와 메서드, 비공개 멤버에 대해 간단히 알아보는 시간을 가져보겠습니다.



자바스크립트 생성자란?

여러 객체지향 프로그래밍 언어에서는 객체의 클래스를 정의한 다음 해당 클래스의 인스턴스에 해당하는 개별 객체를 생성하는 것이 가능합니다.

자바스크립트에서는 진정한 클래스를 지원하지는 않지만 자바스크립트 객체를 생성하는 생성자 함수를 제공하고 있습니다.

자바스크립트 생성자는 자바스크립트 객체를 생성하는 데 사용되는 함수를 말합니다.

자바스크립트 생성자는 new 연산자와 함께 사용되는데, 먼저 프로퍼티가 없는 새 객체를 생성한 후 생성자 함수를 호출하고 새 객체를 this 키워드의 값으로 전달하며, 객체가 사용되기 전에 설정돼 있어야 할 프로퍼티를 설정하는 역할을 합니다.

생성자 함수는 직접 정의할 수 있는데, 단지 this에 프로퍼티를 추가하는 함수를 작성하기만 하면 됩니다.

다음 예제를 살펴봅니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

// 보통 생성자 함수는 반환값이 없다.

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(JSON.stringify(point1));

console.log(JSON.stringify(point2));

자바스크립트 프로토타입(prototype)

모든 자바스크립트 객체는 그것의 프로토타입 객체에 대한 내부 참조를 가지고 있으며, 프로토타입 객체로부터 프로퍼티를 상속받습니다.

모든 함수는 함수가 정의됐을 때 자동으로 생성되고 초기화되는 prototype 프로퍼티를 가지고 있습니다.

이러한 프로토타입 객체에 추가하는 프로퍼티는 생성자로 초기화된 객체 프로퍼티로 나타나게 됩니다.

자바스크립트 프로토타입 예제는 다음과 같습니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

Point.prototype.add = function(){

return this.x + this.y;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(point1.add()); // 10

console.log(point2.add()); // 16

메서드와 프로퍼티는 객체가 생성된 이후에도 동적으로 프로토타입에 추가될 수 있습니다.

다음 예제를 살펴보도록 합니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 객체를 먼저 생성

point1 = new Point(5,5);

point2 = new Point(8,8);

// 프로토타입에 메서드를 추가

Point.prototype.add = function(){

return this.x + this.y;

}

console.log(point1.add()); // 10

console.log(point2.add()); // 16

자바스크립트 프로토타입을 쓰는 이유

자바스크립트 프로토타입의 이점은 다음과 같습니다.

* 객체지향적이고 상속을 사용할 수 있습니다.
* 프로토타입 객체를 사용하면 객체가 프로토타입의 프로퍼티를 상당수 상속할 수 있어서 각 객체에 필요한 메모리의 양을 대폭 줄일 수 있습니다.

인스턴스 프로퍼티와 메서드

자바스크립트는 객체지향 언어이고, 같은 클래스(생성자 함수)의 객체가 여러 개 있을 수 있으며, 각 객체는 해당 클래스의 인스턴스에 해당됩니다.

인스턴스 프로퍼티

모든 객체는 인스턴스 프로퍼티의 사본을 자체적으로 가지고 있습니다.

다음 예제를 살펴보도록 합니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(point1.x); // 5

console.log(point2.y); // 8

인스턴스 프로퍼티에는 개별 객체(이를테면 point1.x)를 통해 접근할 수 있습니다.

인스턴스 메서드

인스턴스 메서드는 특정 인스턴스나 객체를 대상으로 호출됩니다.

인스턴스 메서드를 구현할 때는 this 키워드를 이용해 메서드를 호출할 인스턴스나 객체를 참조할 수 있습니다.

다음 예제를 살려봅니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 객체를 먼저 생성

point1 = new Point(5,5);

point2 = new Point(8,8);

// 프로토타입에 메서드를 추가

Point.prototype.add = function(){

return this.x + this.y;

}

console.log(point1.add()); // 10

console.log(point2.add()); // 16

인스턴스 메서드는 클래스의 모든 인스턴스가 공유합니다.

클래스 프로퍼티와 메서드

위에서는 인스턴스 프로퍼티와 메서드에 알아보았다면 여기서는 클래스 프로퍼티와 메서드에 대해 간략히 알아봅니다.

클래스 프로퍼티

자바스크립트에서 클래스 프로퍼티는 클래스의 각 인스턴스와 연관된 것이 아닌 클래스 자체와 연관된 프로퍼티를 말합니다.

클래스의 인스턴스가 얼마나 생성됐느냐와 관계없이 각 클래스 프로퍼티는 각각 하나씩만 존재합니다.

자바스크립트에서 클래스 프로퍼티를 정의하려면 생성자 함수 자체의 프로퍼티로 정의하기만 하면 됩니다.

이는 자바스크립트 함수 역시 객체이기 때문입니다.

다음 예제를 살려봅니다.

javascript

function Circle(r){

this.radius = r;

}

// 클래스 프로퍼티를 정의

Circle.PI = 3.14159265;

Circle.prototype.area = function(){

return Circle.PI \* this.radius \* this.radius;

}

c1 = new Circle(10);

console.log(c1.area()); // 314.159265

클래스 메서드

클래스 메서드는 클래스의 인스턴스가 아닌 클래스와 연관된 메서드입니다.

자바스크립트에서 클래스 메서드를 정의하려면 적절한 함수를 생성자의 프로퍼티로 만들기만 하면 됩니다.

클래스 메서드는 생성자 함수를 통해 호출되기 때문에 this 키워드는 클래스의 특정 인스턴스를 참조하지 않습니다.

그래서 보통 클래스 메서드에서는 this를 전혀 사용하지 않습니다.

 클래스 메서드의 예는 다음과 같습니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 클래스 메서드를 정의

Point.getHigherPoint = function(a,b){

if (a.x > b.x) return a;

else return b;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(Point.getHigherPoint(point1, point2).y); // 8

비공개 멤버와 메서드

자바스크립트에서 비공개 멤버는 생성자로 만드는데, var 키워드를 이용해 정의한 변수와 생성자의 매개변수는 비공개 멤버가 됩니다.

비공개 멤버는 생성자 함수 밖에서 접근할 수 없으며, 오직 생성자 함수 안에서만 접근할 수 있습니다.

이를 데이터 캡슐화(data encapsulation)라고 합니다.

예제는 다음과 같습니다.

javascript

function Circle(r){

// 비공개 프로퍼티

var radius = r;

// 비공개 메서드

var getArea = function(){

return Circle.PI \* radius \* radius;

}

// 공개 메서드

this.area = function(){return getArea()}

}

// 클래스 프로퍼티를 정의

Circle.PI = 3.14159265;

c1 = new Circle(10);

console.log(c1.area()); // 314.159265

출처: <http://webclub.tistory.com/374> [Web Club]

자바스크립트 객체지향 프로그래밍

자바스크립트를 객체지향 프로그래밍이라고 부를 수 있는 것과 관계된 것 중에 생성자, 프로토타입 인스턴스, 프로퍼티와 메서드, 클래스 프로퍼티와 메서드, 비공개 멤버에 대해 간단히 알아보는 시간을 가져보겠습니다.



자바스크립트 생성자란?

여러 객체지향 프로그래밍 언어에서는 객체의 클래스를 정의한 다음 해당 클래스의 인스턴스에 해당하는 개별 객체를 생성하는 것이 가능합니다.

자바스크립트에서는 진정한 클래스를 지원하지는 않지만 자바스크립트 객체를 생성하는 생성자 함수를 제공하고 있습니다.

자바스크립트 생성자는 자바스크립트 객체를 생성하는 데 사용되는 함수를 말합니다.

자바스크립트 생성자는 new 연산자와 함께 사용되는데, 먼저 프로퍼티가 없는 새 객체를 생성한 후 생성자 함수를 호출하고 새 객체를 this 키워드의 값으로 전달하며, 객체가 사용되기 전에 설정돼 있어야 할 프로퍼티를 설정하는 역할을 합니다.

생성자 함수는 직접 정의할 수 있는데, 단지 this에 프로퍼티를 추가하는 함수를 작성하기만 하면 됩니다.

다음 예제를 살펴봅니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

// 보통 생성자 함수는 반환값이 없다.

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(JSON.stringify(point1));

console.log(JSON.stringify(point2));

자바스크립트 프로토타입(prototype)

모든 자바스크립트 객체는 그것의 프로토타입 객체에 대한 내부 참조를 가지고 있으며, 프로토타입 객체로부터 프로퍼티를 상속받습니다.

모든 함수는 함수가 정의됐을 때 자동으로 생성되고 초기화되는 prototype 프로퍼티를 가지고 있습니다.

이러한 프로토타입 객체에 추가하는 프로퍼티는 생성자로 초기화된 객체 프로퍼티로 나타나게 됩니다.

자바스크립트 프로토타입 예제는 다음과 같습니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

Point.prototype.add = function(){

return this.x + this.y;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(point1.add()); // 10

console.log(point2.add()); // 16

메서드와 프로퍼티는 객체가 생성된 이후에도 동적으로 프로토타입에 추가될 수 있습니다.

다음 예제를 살펴보도록 합니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 객체를 먼저 생성

point1 = new Point(5,5);

point2 = new Point(8,8);

// 프로토타입에 메서드를 추가

Point.prototype.add = function(){

return this.x + this.y;

}

console.log(point1.add()); // 10

console.log(point2.add()); // 16

자바스크립트 프로토타입을 쓰는 이유

자바스크립트 프로토타입의 이점은 다음과 같습니다.

* 객체지향적이고 상속을 사용할 수 있습니다.
* 프로토타입 객체를 사용하면 객체가 프로토타입의 프로퍼티를 상당수 상속할 수 있어서 각 객체에 필요한 메모리의 양을 대폭 줄일 수 있습니다.

인스턴스 프로퍼티와 메서드

자바스크립트는 객체지향 언어이고, 같은 클래스(생성자 함수)의 객체가 여러 개 있을 수 있으며, 각 객체는 해당 클래스의 인스턴스에 해당됩니다.

인스턴스 프로퍼티

모든 객체는 인스턴스 프로퍼티의 사본을 자체적으로 가지고 있습니다.

다음 예제를 살펴보도록 합니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(point1.x); // 5

console.log(point2.y); // 8

인스턴스 프로퍼티에는 개별 객체(이를테면 point1.x)를 통해 접근할 수 있습니다.

인스턴스 메서드

인스턴스 메서드는 특정 인스턴스나 객체를 대상으로 호출됩니다.

인스턴스 메서드를 구현할 때는 this 키워드를 이용해 메서드를 호출할 인스턴스나 객체를 참조할 수 있습니다.

다음 예제를 살려봅니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 객체를 먼저 생성

point1 = new Point(5,5);

point2 = new Point(8,8);

// 프로토타입에 메서드를 추가

Point.prototype.add = function(){

return this.x + this.y;

}

console.log(point1.add()); // 10

console.log(point2.add()); // 16

인스턴스 메서드는 클래스의 모든 인스턴스가 공유합니다.

클래스 프로퍼티와 메서드

위에서는 인스턴스 프로퍼티와 메서드에 알아보았다면 여기서는 클래스 프로퍼티와 메서드에 대해 간략히 알아봅니다.

클래스 프로퍼티

자바스크립트에서 클래스 프로퍼티는 클래스의 각 인스턴스와 연관된 것이 아닌 클래스 자체와 연관된 프로퍼티를 말합니다.

클래스의 인스턴스가 얼마나 생성됐느냐와 관계없이 각 클래스 프로퍼티는 각각 하나씩만 존재합니다.

자바스크립트에서 클래스 프로퍼티를 정의하려면 생성자 함수 자체의 프로퍼티로 정의하기만 하면 됩니다.

이는 자바스크립트 함수 역시 객체이기 때문입니다.

다음 예제를 살려봅니다.

javascript

function Circle(r){

this.radius = r;

}

// 클래스 프로퍼티를 정의

Circle.PI = 3.14159265;

Circle.prototype.area = function(){

return Circle.PI \* this.radius \* this.radius;

}

c1 = new Circle(10);

console.log(c1.area()); // 314.159265

클래스 메서드

클래스 메서드는 클래스의 인스턴스가 아닌 클래스와 연관된 메서드입니다.

자바스크립트에서 클래스 메서드를 정의하려면 적절한 함수를 생성자의 프로퍼티로 만들기만 하면 됩니다.

클래스 메서드는 생성자 함수를 통해 호출되기 때문에 this 키워드는 클래스의 특정 인스턴스를 참조하지 않습니다.

그래서 보통 클래스 메서드에서는 this를 전혀 사용하지 않습니다.

 클래스 메서드의 예는 다음과 같습니다.

javascript

function Point(x,y){

this.x = x;

this.y = y;

}

// 클래스 메서드를 정의

Point.getHigherPoint = function(a,b){

if (a.x > b.x) return a;

else return b;

}

point1 = new Point(5,5);

point2 = new Point(8,8);

console.log(Point.getHigherPoint(point1, point2).y); // 8

비공개 멤버와 메서드

자바스크립트에서 비공개 멤버는 생성자로 만드는데, var 키워드를 이용해 정의한 변수와 생성자의 매개변수는 비공개 멤버가 됩니다.

비공개 멤버는 생성자 함수 밖에서 접근할 수 없으며, 오직 생성자 함수 안에서만 접근할 수 있습니다.

이를 데이터 캡슐화(data encapsulation)라고 합니다.

예제는 다음과 같습니다.

javascript

function Circle(r){

// 비공개 프로퍼티

var radius = r;

// 비공개 메서드

var getArea = function(){

return Circle.PI \* radius \* radius;

}

// 공개 메서드

this.area = function(){return getArea()}

}

// 클래스 프로퍼티를 정의

Circle.PI = 3.14159265;

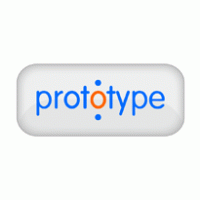
c1 = new Circle(10);

console.log(c1.area()); // 314.159265

출처: <http://webclub.tistory.com/374> [Web Club]

프로토타입 기반 프로그래밍

자바스크립트를 객체지향적인 언어라고 말할 수 있는 가장 중요한 이유 중 하나는 프로토타입 상속(prototypal inheritance) 이라는 확장과 재사용을 가능하게 해주기 때문입니다.



프로토타입 기반 프로그래밍은 객체의 원형인 프로토타입을 이용하여 새로운 객체를 만들어내는 프로그래밍 기법입니다. 이렇게 새롭게 만들어진 객체 역시 자기자신의 프로토타입을 가지고 있으며 이 새로운 객체의 원형을 이용하면 또 다른 새로운 객체를 만들어 낼 수도 있으며 이런 방식,구조로 객체를 확장하는 방식을 프로토타입 기반의 프로그래밍 언어라고 말합니다.

자바스크립트에서 사용되는 프로토타입이란 용어는 크게 두가지로 나누어지고 있기 때문에 이 두가지를 구분하는 것이 중요합니다.

프로토타입 프로퍼티(Prototype Property)가 가리키고 있는 프로토타입 객체(Prototype Object)와 자기 자신을 만들어낸 객체의 원형을 의미하는 프로토타입 링크(Prototype Link)로 나뉘어 집니다.

그리고 ECMAScript에서는 이러한 프로토타입 링크를 암묵적 프로토타입 링크(implicit prototype link)라고 부릅니다.

자바스크립트의 프로토타입(Prototype)

자바스크립트의 모든 객체는 자기 자신을 생성한 객체 원형에 대한 숨겨진 연결고리를 가지고 있습니다. 이때 자기 자신을 생성하기 위해서 사용된 객체의 원형을 프로토타입이라고 합니다.

그리고 자바스크립트의 모든 객체는 Object 객체의 프로토타입을 기반으로 하여 확장되었기 때문에 이 연결고리의 끝은  Object 객체의 프로토타입 Object 입니다.

프로토타입이란 사전적 의미가 원본,원형으로써 프로그래밍 언어에서 프로토타입이란 의미도 크게 다르지 않습니다.

다시 말해, 어떠한 객체가 만들어지기 위해 그 객체의 부모(모태)가 되는 녀석을 프로토타입이라고 합니다.

Java, C++ 과 같은 언어에서 Class 가 바로 자바스크립트의 프로토타입과 동일한 의미로 해석될 수 있습니다.

출처: <http://webclub.tistory.com/158> [Web Club]

[**자바스크립트 클래스(생성자) 만드는 방법**](http://webclub.tistory.com/136)

10개월 전 by 하기실음 관두등가 재야재희

javaScript Class



자바스크립트에서 클래스란...생성자 함수이다!!!

함수가 특정기능을 하는 구문(알고리즘, 로직)을 묶을 때 사용하는 문법이라면, 클래스는 이렇게 만들어진 수많은 변수와 함수 중 연관 있는 변수와 함수만을 선별해 포장하는 기술입니다.

다시말해서, 연관있는 변수와 함수를 하나로 묶을 때 사용하는 문법입니다.

이렇게 클래스로 포장하는 이유는 객체 단위로 코드를 그룹화 할 수 있으며 코드를 재사용하기 위해서 입니다.

클래스로 포장하는 이유는?

*객체 단위로 코드 그룹화 및 코드 재사용성을 사용하기 위함입니다.*

*함수와 비교해서 설명하면 다음과 같이 비교 할 수 있습니다.*

**1) 함수기반 단위**

포장내용 :  특정 기능을 하는 변수 + 구문(알고리즘)등을 감싸는 것입니다.

기능 :

*- 기능 단위의 코드 그룹화*

*- 기능 단위의 중복 코드 제거 및 코드 재사용성*

**2) 클래스기반 단위**

포장내용: 연관 있는 변수와 함수를 감싸는 것입니다.

기능:

*- 객체 단위의 변수와 함수*

*- 객체 단위의 중복 코드 제거 및 코드 재사용.*

다시 말해서  UI로 슬라이더가 있는 경우 로직을 짤 때 슬라이더의 경우 indicate 기능, next, prev 버튼등의 여러 기능들이 있을 경우에 대개는 함수 기반 단위의 코딩은 슬라이더의 필요한 기능들을 각각 로직을 구현하는데 이런 방식으로 구현하게 되면 전역스코프도 오염될 뿐더러 소스 가독성도 떨어지며 그로 인한 유지보수도 힘들어집니다.

하지만 클래스 단위로 코딩을 하게 되면 슬라이더에 대한 기능들을 모두 그룹화하여 관리할 수 있게 됩니다.

그래서 클래스의 대표적인 기능으로 연관 있는 변수와 함수를 그룹화할 수 있습니다.

함수를 그룹화한다는 것이 중요합니다.

이렇게 클래스를 사용함으로써 얻는 이점은 겍체 단위의 중복 코드를 제거 할 수 있으며 코드 재사용성이 좋아집니다.

코드의 재사용성이라 함은 예를 들어서 탭패널 3개가 들어 있는 웹 페이지를 만들어야 하는 경우 클래스 하나로 독립적으로 동작하는 3개의 탭패널을 모두 동작하게 할 수 있다는 것입니다.

함수 기반 단위의 코딩은 코드가 구분없이 작성되어 있기 때문에 이슈가 발생할 경우 해당 이슈되는 코드를 쉽게 찾을 수 없으며, 큰 프로젝트에서 다수의 사람이 협업을 이뤄 진행하게 될 때 개발자간에 같은 이름의 변수와 함수를 만들어 사용할 확률이 많아지기 때문에 그로 인한 이슈,충돌의 위험성이 내포되어 있습니다.

하지만 클래스란 포장기술을 이용하면 산발적으로 흩어져 있는 코드들을 한 곳에 부품별로 포장하여 그룹화시켜 관리할 수 있도록 해줍니다.

클래스(생성자) VS 일반함수

|  |  |  |
| --- | --- | --- |
|  | 클래스(생성자함수) | 일반 함수 |
| 포장 내용 | 연관있는 변수와 함수 | 특정기능을 하는 변수 + 구문 |
| 기능 | 객체 단위의 코드를 그룹화  객체 단위의 중복 코드 제거 및 코드의 재사용성 | 기능 단위의 코드를 그룹화  기능 단위의 중복 코드 제거 및 코드의 재사용성 |

자바스크립트에서 클래스를 만드는 방법 3가지

사실 자바스크립에는 객체지향 프로그래밍 언어에서 기본적으로 제공하는 클래스라는 개념을 제공하지 않습니다.

생성자 함수가 다른 언어에서의 클래스(Class)와 유사한 개념이기 때문에 클래스를 사용할 수 있는 방법이 있습니다.

그렇기 때문에 자바스크립트에서는 클래스는 없지만 OOP 와 같이 비슷하게 사용할 수 있는 문법이 존재합니다.

다음은 그 세 가지 방법입니다.

*1) 리터럴 방식*

*2) 함수 방식*

*3) 프로토타입 방식*

**1) 리터럴 방식의 클래스**

var 인스턴스 = {  
 프로퍼티1 : 초기값,  
 프로퍼티2 : 초기값,  
 메서드1 : function() {  
  
 },  
 메서드2 : function() {  
  
 }  
  
}

**2) 함수 방식의 클래스**

function 클래스이름() {  
 this.프로퍼티1 = 초기값;  
 this.프로퍼티2 = 초기값;  
   
 this.메서드1 = function() {  
   
 }  
 this.메서드2 = function() {  
   
 }  
  
}  
  
var 인스턴스 = new 클래스이름();

**3) 프로토타입(prototype) 방식의 클래스**

function 클래스이름() {  
 this.프로퍼티1 = 초기값;  
 this.프로퍼티2 = 초기값;  
}  
  
클래스이름.prototype.메서드1 = function() {  
  
}  
  
클래스이름.prototype.메서드2 = function() {  
  
}

위와 같이 세 가지 패턴으로 클래스를 사용할 수 있습니다.

하지만 세 가지 패턴마다 장,단점이 있기 때문에 상황에 맞게 선택적으로 사용해야 합니다.

한가지 단점을 말씀드려 보자면 단점이 없어 보이는 함수방식의 클래스는 아주 치명적인 단점을 가지고 있습니다.

바로 인스턴스를 생성할 때마다 내부의 모든 메서드가 독립적으로 만들어진다는 것입니다.

예를 들어 탭메뉴를 제작할 때 하나의 클래스로 여러 개의 탭메뉴를 개별적으로 동작한다고 했을 때 클래스는 하나이지만 각각의 탭메뉴에 대한 인스턴스를 생성하게 되면 그 인스턴스마다 클래스 내부에 정의된 변수나 함수들이 독립적으로 각각 생성되어 메모리 누수가 생길 여지가 많습니다.

즉, 인스턴스마다 중복 변수와 중복 메서드들이 생성됩니다.

이번 포스팅에서는 간단하게 클래스를 사용할 때 알아야할 몇 가지 개념들만 짚고 넘어가 보도록 하겠습니다.

인스턴스(instance)

일반적인 함수의 경우는 함수를 선언한 후 함수를 호출해줘야 동작하듯이 클래스의 경우는 클래스 인스턴스를 생성해야만 클래스에 들어 있는 함수들의 내부 기능들을 사용할 수 있습니다.

다시말해서, 클래스가 설계 도면이라면 인스턴스는 설계에 따른 실체가 담겨 지는 곳입니다.

**클래스(생성자 함수)는 설계도, 인스턴스는 실체이다.**

예) 붕어빵을 제작한다고 가정했을 때 : 붕어빵틀 = 클래스, 붕어빵 = 인스턴스

그리고 일반적으로 인스턴스 생성 방법은 다음과 같습니다.

var 인스턴스 = new 클래스이름(); *// new 키워드를 사용*

객체(Object)란?

객체는 인스턴스의 또 다른 이름이고 그때 그때 상황에 따라 인스턴스 또는 객체라고 부르고 있습니다.

인스턴스는 주로 new 연산자를 이용하여 클래스의 실체를 생성할 때 사용하고 이 외에 클래스에서 제공하는 프로퍼티와 메서드를 사용한다고 했을 때의 경우를 객체라고 말합니다.

그리고 프로퍼티는 클래스 내부에 만드는(포장하는) 변수를 프로퍼티티라고 부르며 멤버 변수라고도 합니다.

메서드는 클래스에 만드는(포장하는) 함수들을 메서드라고 부르며 멤버 함수라고도 부릅니다.

메서드는 주로 객체의 프로퍼티 값을 변경하거나 알아내는 기능과 클래스를 대표하는 기능이 담기게 됩니다.

지금까지 간단히 클래스에 대해 알아보았으며 아래의 링크에서는 같은 내용일 수는 있으나 조금 다른 접근방식의 설명,내용들을 보실 수 있습니다.

[자바스크립트 생성자 기초 알아보기](http://e-rooms.tistory.com/entry/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8-%EC%83%9D%EC%84%B1%EC%9E%90)

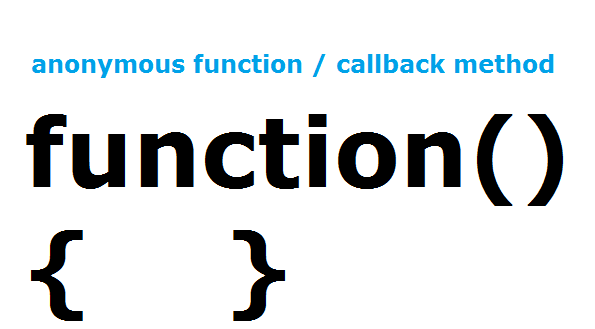
[자바스크립트 생성자를 클래스라 부르는 이유](http://e-rooms.tistory.com/entry/%ED%81%B4%EB%9E%98%EC%8A%A4-%EC%83%9D%EC%84%B1%EC%9E%90-%EB%A9%94%EC%84%9C%EB%93%9C)

출처: <http://webclub.tistory.com/136> [Web Club]

[**함수는 일급 객체이다.**](http://webclub.tistory.com/12)

10개월 전 by 하기실음 관두등가 재야재희

함수 : First Class



자바스크립트에서 함수는 값으로 취급된다.

그리고 함수는 객체다.

이는 함수도 일반 객체처럼 취급될 수 있다는 것을 말한다.

또한 다음과 같은 특징으로 인해 자바스크립트에서는 함수를 일급 객체(First Class)라고 부른다.

1. 러터럴에 의해 생성

2. 변수나 배열의 요소(원소), 객체의 프로퍼티 등에 할당 가능

3. 함수의 인자로 전달 가능

4. 함수의 러턴값으로 리턴 가능

5. 동적으로 프로퍼티를 생성 및 할당 가능

여기서 일급객체라는 말은 컴퓨터 프로그래밍 언어분야에서 쓰이는 용어로서, 앞에서 나열한 기능이 모두 가능한 객체를 일급객체라고 부른다.

이렇게 자바스크립트 함수가 가지는 이러한 일급 객체의 특성으로 함수형 프로그래밍이 가능한 것이다.

그럼 위에 나열된 항목들을 살펴보자.

1. 리터럴에 의해 생성

// 함수 리터럴을 통한 sum()함수 정의  
  
// 함수 표현식 방법  
**var** sum = **function**(x, y) {  
 **return** x + y;  
}  
//생성된 함수를 변수에 할당하여 함수를 생성하는 것을 함수 표현식  
  
// 함수 선언문 방법  
**function** sum(x, y) {  
 **return** x + y;  
}  
// 함수 생성시 함수명이 정의되어 있는 것을 함수 선언문 (즉, 함수 이름이 정의되어 있음)

위 코드와 같이 함수 리터럴을 이용해 함수를 생성할 수 있다.

실제로 함수 선언문이나 함수 표현식 방법 모두는 이런 함수 리터럴 방식으로 함수를 생성하는 것이다.

2. 변수나 프로퍼티의 값으로 할당이 가능

// 변수에 함수 할당  
**var** bar = **function**() { **return** 'jaehee'; };  
console.log(bar()); // jaehee  
  
// 프로퍼티에 함수 할당  
**var** obj = {};  
obj.bar2 = **function**() { **return** 'javaScript'; };  
console.log(obj.bar2()); // javaScript

위 코드의 bar2처럼 배열의 원소 등에도 할당이 가능하다.

3. 함수의 인자로 전달 가능

// 함수를 다른 함수의 인자로 넘긴 코드  
// 함수 표현식으로 test() 함수를 생성  
**var** test = **function**(func) {  
 func(); // 인자로 넘겨받은 func()함수를 호출  
}  
  
//test() 함수를 호출,실행  
test(**function**(){  
 console.log('함수자체는 매개변수(인자)로 사용될 수 있다.');  
});

test() 함수를 호출할 때, 함수 리터럴 방식으로 생성한 익명함수를 func 인자로 넘겼다.

따라서 test() 함수 내부에서는 func매개변수로 인자에 넘겨진 함수를 호출할 수 있는 것이다.

4. 함수의 리턴값으로 리턴 가능

// 함수를 다른 함수의 리턴값으로 활용한 코드  
  
// 함수를 리턴하는 foo() 함수 정의  
**var** foo = **function**() {  
 // 함수를 리턴  
 **return function**() {  
 console.log('이 함수는 함수를 반환한다.');  
 };  
};  
  
// foo() 함수가 호출되면, 리턴값으로 전달되는 익명함수가 bar 변수에 저장된다.  
**var** bar = foo();  
  
// 익명함수를 호출하기 위해서 ()인 함수호출 연산자를 이용해  
// bar()로 익명함수를 호출,실행한다.  
bar();

위 코드는 간단한 익명함수를 리턴하는 역할을 한다. 이것이 가능한 이유 또한 함수 자체가 값으로 취급되기 때문이다.

5. 동적으로 프로퍼티를 생성 및 할당 가능

계속해서 강조하지만 자바스크립트에서는 함수도 객체다.

즉, 함수의 기본 기능인 코드 실행뿐만 아니라, 함수 자체가 일반 객체처럼 프로퍼티를 생성하고 할당이 가능하다는 것이다.

// 함수 선언문 방식으로 add() 함수 정의  
**function** add(x,y) {  
 **return** x + y;  
}  
  
// add()함수도 객체처럼 프로퍼티를 가질 수 있다.  
// add()함수 객체에 result, status 프로퍼티를 추가해 본다.  
add.result = add(3,4);  
add.status = 'complete';  
  
console.log(add.result); // 7  
console.log(add.status); // 'complete'

add()함수는 마치 일반 객체처럼 result 프로퍼티를 동적으로 생성하고 add()함수를 호출한 결과를 저장한 것을 확인 할 수가 있다.

이렇게 일반객체의 접근방식처럼 add.result, add.status를 이용해 접근이 가능하다.

다시말해서 자바스크립트에서 함수는 특정기능의 코드를 수행할 뿐만 아니라, 일반 객체처럼 자신의 프로퍼티를 가질 수 있는 특별한 객체라고 볼 수 있다.

출처: <http://webclub.tistory.com/12> [Web Club]

## [함수의 프로토타입 - prototype](http://webclub.tistory.com/301)

10개월 전 by 하기실음 관두등가 재야재희

프로토타입 체인(Prototype Chain)

자바스크립트는 Function() 인스턴스에 자동으로 prototype 이라는 속성을 만듭니다.

구체적으로 말하자면 prototype 속성은 new 키워드와 생성자 함수를 같이 사용해서 만든 객체에 연결됩니다.

인스턴스들은 생성자 함수의 prototype 속성을 통해 공통의 메소드와 속성을 공유하고 상속합니다.

중요한 사실은 이러한 공유가 속성을 검색하는 동안 일어난다는 것입니다.

즉, 우리가 어떤 객체의 속성을 찾거나 접근하면 자바스크립트는 그때마다 해당 객체는 물론 프로토타입 체인에서도 그 속성을 찾습니다.



함수는 모두 **prototype 객체**를 포함하고 있다.

심지어 생성자 함수로 사용할 생각이 전혀없는 함수도 prototype 객체를 자동으로 포함한다.

다음은 Array() 생성자를 사용하여 배열을 하나 만든 후 배열의 join() 메소드를 호출하는 코드입니다.

javascript

var myArray = new Array('foo', 'bar');

console.log(myArray.join()); // 'foo, bar' 가 기록된다.

join() 메소드는 myArray 객체 인스턴스에 정의되어 있지 않은 속성입니다.

하지만 사용자는 join() 메소드가 마치 원래 있던 것처럼 사용할 수 있었습니다.

분명한 것은 이 메소드는 어디엔가 정의되어 있을 것입니다.

하지만 어디에 정의되어 있을까?!

이 메소드는 Array() 생성자의 prototye 속성에 속성으로서 정의되어 있습니다.

배열 객체 인스턴스에서는 join() 메소드가 없기 때문에 자바스크립트는 프로토타입 체인에서 join() 이라는 메소드가 있는지 검색합니다.

왜 이런 식으로 동작하도록 만들었을까?

사실 이는 효율성과 재사용성에 대한 문제입니다.

배열 생성자 함수가 만드는 배열 인스턴스마다 언제나 똑같은 방식으로 동작하는 join() 메소드를 굳이 일일이 추가할 필요가 있을까?

모든 배열 인스턴스마다 함수를 새로 만드는 것보다는 하나의 join() 함수를 모든 배열에서 가져다 쓰는 편이 더 합리적일 것입니다.

자바스크립트에서는 prototype 속성, 프로토타입 결합(prototype linkage), 프로토타입 체인 검색을 통해 이런 효율성을 만들었습니다.

하지만 솔직히 말해 프로토타입 체인 상속이 실제로 어떻게 동작하는지 원리를 그냥 외우는 편이 더 나을지도 모릅니다.

prototype 속성이 왜 중요한가?!

prototype 속성은 네 가지 측면에서 중요합니다.

1. 네이티브 생성자 함수(예: Object(), Array(), Function() 등)는 prototype 속성을 사용해 생성자 인스턴스가 메소드와 속성을 상속받도록 하고 있기 때문이다. 자바스크립트는 이러한 매커니즘을 사용해 객체 인스턴스가 생성자 함수의 prototype 속성을 상속받을 수 있도록 지원해 준다. 따라서 자바스크립트를 더 잘 이해하려면 자바스크립트가 prototype 객체를 어떻게 사용하는지 잘 알아야 한다.
2. 사용자 정의 생성자 함수를 만들 때 자바스크립트 네이티브 객체와 동일한 방식을 통해 프로토타입 상속을 구현할 수 있다. 그러나 상속이 이루어지는 방식에 대한 이해가 반드시 수반되어야 한다.
3. 프로토타입 상속을 선호하지 않거나 다른 상속 패턴을 더 좋아할 수도 있지만 현실적으로 볼 때 아마도 누군가 프로토타입 상속을 사용해 구현해놓은 코드를 수정하거나 조작해야 할 일이 있을 것이다. 이때 프로토타입 상속이 어떻게 동작하는지 알아야 다른 개발자가 만든 생성자 함수의 기능을 그대로 복제할 수 있을 것이다.
4. 프로토타입 상속을 사용하면 동일한 메소드를 공유하는 여러 개의 효율적인 객체 인스턴스를 만들 수 있다. 앞서 언급했던 것과 같이 Array() 생성자의 인스턴스인 배열 객체는 굳이 인스턴스마다 join() 메소드를 추가해주지 않아도 된다. 모든 배열 인스턴스의 프로토타입 체인에는 join() 메소드가 저장되어 있으므로 모든 배열 인스턴스는 동일한 join() 메소드를 사용할 수 있다.

모든 Funtion() 인스턴스에는 prototype 속성을 가지고 있다.

자바스크립트에서 함수는 Function() 생성자를 직접 호출하여 만들었든(예: var add = new Function('x', 'y', 'return x + y'); ) 함수 리터럴 표기법을 사용해 만들었든(예: var add = function(x, y) { return x + y }; ) 예외없이 모두 Function() 생성자로부터 만들어집니다.

함수 인스턴스를 만들면 그 인스턴스에는 항상 prototype 속성이 추가됩니다.

이때, prototype 속성은 그 자체로는 빈 객체와 같습니다.

다음 코드에서 myFn 이라는 함수를 정의한 후, 이 함수의 prototype 속성에 접근해 봅니다.

javascript

var myFn = function() {};

console.log(myFn.prototype); // Object() 가 기록된다.

console.log(typeof myFn.prototype); // 'object' 가 기록된다.

비록 위 코드에서는 생성자로 사용한 사용자 정의 함수가 하나밖에 없지만, Function() 생성자는 모든 함수 인스턴스에 prototype 속성을 부여합니다.

prototype 속성은 Object() 객체다

프로토타입에 대한 내용은 조금 어렵습니다.

엄밀히 말해, prototype은 Function() 인스턴스를 만들 때 자바스크립트가 **인스턴스에 부여하는 이름이 "prototype"** 이고 기본값이 빈 객체인 속성일 뿐입니다.

만약 이를 사용자가 직접 코드로 만든다면 다음과 같이 작성할 수 있을 것입니다.

javascript

var myFn = function() {};

// prototype 속성을 추가하고 속성값으로 빈 객체를 할당한다.

myFn.prototype = {};

console.log(myFn.prototype); // 빈 객체가 기록된다.

사실 자바스크립트가 이미 내부적으로 수행했던 일을 다시 코드로 작성한 것이고, 코드로 직접 작성한 위의 예제도 실행해보면 문제없이 동작합니다.

사실 prototype 속성의 값으로는 어떤 복합 객체(E.g : 배열)든 사용할 수 있다.

prototype의 속성값으로 원시값을 설정하려 하면 자바스크립트에서는 해당 코드를 무시한다.

생성자 함수를 통해 만든 인스턴스는 생성자 함수의 prototype 속성과 연결되어 있다

생성자 함수의 prototype 속성은 그 자체로만 놓고보면 객체일 뿐이지만 프로토타입 체인을 통해 인스턴스와 연결되는 특이한 속성입니다.

다시 말해, new 키워드와 생성자 함수를 사용해 객체를 만들면 생성자 함수의 prototype 속성과 새롭게 만들어진 객체 인스턴스 사이에는 일종의 숨겨진 연결고리가 생깁니다.

일부 브라우저에서는 이 연결고리가 인스턴스의 \_\_proto\_\_ 속성으로 나타나기도 합니다.

현재는 Firefox2+, Safari, Chrome, Android 브라우저에서만 지원되고 있는 속성입니다.

자바스크립트는 생성자 함수를 사용해 인스턴스를 만들 때 인스턴스 객체와 생성자 함수를 자동으로 연결해 두며, 이러한 연결 덕분에 프로토타입 체인이 형성되는 것입니다.

다음은 Array() 생성자의 prototype에 속성을 하나 추가한 후 배열 인스턴스의 \_\_proto\_\_ 속성을 사용해 추가한 속성에 접근하는 코드입니다.

javascript

// 이 코드는 \_\_proto\_\_ 를 사용할 수 있는 브라우저에서만 동작한다.

// 파이어폭스 2이상, 사파리, 크롬, 안드로이드에서만 동작한다.

Array.prototype.foo = 'foo';

var myArray = new Array();

console.log(myArray.\_\_proto\_\_.foo); // foo 가 기록된다.

// myArray.\_\_proto == Array.prototype 이다

사실 \_\_proto\_\_ 속성은 ECMA 공식 표준이 아니기 때문에 다른 방법을 사용해야 합니다.

객체 인스턴스에서 constructor 속성을 사용하면 생성자를 구할 수 있고 이를 통해 인스턴스가 상속받은 프로토타입 객체도 구할 수 있습니다.

다음의 코드를 살펴봅니다.

javascript

Array.prototype.foo = 'foo'; // 이제 Array() 의 인스턴스는 모두 foo 속성을 상속받는다.

var myArray = new Array();

// .constructor.prototype을 사용하면 조금 번잡하지만 foo 속성을 구할 수 있다.

console.log(myArray.constructor.prototype.foo); // foo 가 기록된다.

// 물론 체인을 사용해도 된다.

console.log(myArray.foo); // foo 가 기록된다.

// 프로토타입 체이닝을 통해 검색된다. Array.prototype.foo 가 검색된다.

이 코드에서 foo 속성은 prototype 객체에서 찾을 수 있었습니다.

이는 Array() 인스턴스와 Array() 생성자의 프로토타입 객체(=Array.prototype) 간에 연결고리가 있었기 때문에 가능한 것입니다.

간단히 말해서 myArray.\_\_proto\_\_(또는 myArray.constructor.prototype)는 Array.prototype 을 참조합니다.

프로토타입 체인의 끝은 Object.prototype 이다

prototype 속성은 객체이기 때문에 프로토타입 체인 또는 프로토타입 검색의 종점은 Object.prototype 입니다.

다음 코드를 살펴봅니다.

javascript

var myArray = [];

console.log(myArray.foo); // undefined 가 기록된다.

/\*\*

\* myArray.foo, Array.prototype.foo, Object.prototype.foo에서 foo를 찾지 못하여

\* foo 속성은 undefined 가 된다.

\*-----------------------------------------------------------------------------

\*/

이 코드에서는 myArray 라는 빈 배열을 만든 후 프로토타입 체인을 검색하도록 myArray에 정의되지 않은 속성에 접근해 보았습니다.

자바스크립트는 먼저 myArray 객체에서 foo 속성을 찾아보지만, 찾지 못하기 때문에 Array.prototype 에서 다시 검색을 시도할 것입니다.

하지만 여기서도 찾을 수 없으므로 마지막 종착지점인 Object.prototype에서 foo 속성을 찾아보는 것입니다.

검색의 끝 지점인 여기에서도 찾을 수 없으므로 결국 foo 속성은 undefined가 되는 것입니다.

정리해 보면, 프로토타입 체인은 Object.prototype 에서 끝납니다. foo 속성이 있는지 가장 마지막에 찾아본 곳은 Object.prototype이었습니다.

Object.prototype에 속성을 추가하면 추가한 속성이 for in 반복문에 나타난다.

프로토타입 체인은 체인에서 제일 먼저 찾은 속성을 반환한다

스코프 체인과 마찬가지로 프로토타입 체인은 체인을 검색하다가 가장 먼저 발견한 값을 사용합니다.

바로 앞에서 사용했던 코드를 조금 수정하여 Object.prototype과 Array.prototype 객체에 똑같은 속성을 추가한 후 배열 인스턴스에서 그 값에 접근해 보면 Array.prototype 객체의 값이 반환될 것입니다.

javascript

Object.prototype.foo = 'object-foo';

Array.prototype.foo = 'arry-foo';

var myArray = [];

console.log(myArray.foo); // Array.prototype.foo 에서 찾은 'array-foo' 가 기록된다.

myArray.foo = 'bar';

console.log(myArray.foo); // myArray.foo 에서 찾은 'bar' 가 기록된다.

이 코드에서 Array.prototype.foo의 foo 값은 Object.prototype.foo의 foo 값을 가려버린 결과를 가져온 것입니다.

이렇게 프로토타입에서 속성을 검색할 때는 가장 먼저 발견한 속성을 사용하고 검색을 바로 종료합니다. 그리고 체인에 똑같은 이름의 속성이 몇 개나 더 있었는지는 상관없습니다.

prototype 속성을 새 객체로 대체하면 기본 constructor 속성이 삭제된다

prototype 속성의 기본값은 다른 값으로 대체할 수 있습니다.

하지만 prototype 속성을 바꾸면 원래의 prototype 객체에서 볼 수 있었던 기본 constructor 속성이 사라지게 됩니다.

다음은 Foo 생성자 함수를 만들고 Foo의 prototype 속성을 빈 객체로 대체한 후 인스턴스의 constructor 속성이 사라졌는지 확인해 보는 코드입니다.

javascript

var Foo = function() {};

Foo.prototype = {}; // 빈 객체로 prototype 속성을 대체한다.

var FooInstance = new Foo();

console.log(FooInstance.constructor === Foo); // false 가 기록된다. 즉 참조가 사라진 것이다.

console.log(FooInstance.constructor); // Foo() 가 아닌 Object()가 기록된다.

위 코드는 이제 constructor 속성은 덜 유용한 Object() 생성자를 참조하게 됩니다.

아래 코드는 prototype 값을 대체하지 않은 경우입니다.

javascript

var Bar = function() {};

var BarInstance = new Bar();

console.log(BarInstance.constructor === Bar); // true 가 기록된다.

console.log(BarInstance.constructor); // Bar() 가 기록된다.

만약 자바스크립트가 설정한 기본 prototype 속성을 대체할 생각(자바스크립트 객체지향 패턴에서 종종 사용되는 방식)이라면, 생성자 함수를 참조하는 constructor 속성을 원래대로 복원해주어야 할 것입니다.

다음은 앞의 코드를 조금 수정하여 constructor 속성이 원래의 생성자 함수를 올바르게 참조하도록 한 코드입니다.

javascript

var Foo = function Foo(){};

Foo.prototype = {constructor:Foo};

var FooInstance = new Foo();

console.log(FooInstance.constructor === Foo); // true가 기록된다.

console.log(FooInstance.constructor); // Foo()가 기록된다.

프로토타입에서 상속한 속성은 가장 최근의 값을 사용한다

인스턴스가 프로토타입에서 상속한 속성은 그 속성이 어떻게 만들어지고, 변경되고, 추가되었든 상관없이 항상 최근의 값을 사용합니다.

다음 코드는 Foo 생성자를 만들고 x라는 속성을 prototype에 추가한 후 Foo() 생성자의 인스턴스를 FooInstance라는 이름으로 만듭니다.

여기서 한번 x의 값을 기록한 후 생성자의 prototype 속성에 포함된 x의 값을 수정한 후 다시 한번 x의 값을 기록해 봅니다.

이를 통해 우리는 인스턴스가 prototype 객체에서 가져오는 값은 가장 마지막 값이라는 사실을 알 수 있을 것입니다.

javascript

var Foo = function Foo(){};

Foo.prototype.x = 1;

var FooInstance = new Foo();

console.log(FooInstance.x); // 1이 기록된다.

Foo.prototype.x = 2;

console.log(FooInstance.x); // 2가 기록된다. FooInstance 도 갱신되었다.

프로토타입 체인이 어떻게 동작하는지 이해했다면 이 같은 동작이 그리 놀랍진 않을 것입니다.

이런 현상은 기본 prototype 객체를 사용했든 또는 prototyp 객체를 새로운 객체로 대체했든 상관없이 똑같이 볼 수 있습니다.

다음 코드에서는 기본 prototype 객체 대신 새로운 객체를 사용해 봅니다.

javascript

var Foo = function Foo(){};

Foo.prototype = {x:1}; // 아래의 코드는 이전과 똑같이 동작한다.

var FooInstance = new Foo();

console.log(FooInstance.x); // 1이 기록된다.

Foo.prototype.x = 2;

console.log(FooInstance.x); // 2가 기록된다. FooInstance 도 갱신되었다.

prototype 속성을 새 객체로 대체하면 이전에 만든 인스턴스는 갱신되지 않는다

지금까지 알아본 바에 의하면 아마도 prototype 속성은 언제든 완전히 대체할 수 있고 그렇게 하면 모든 인스턴스가 갱신될 것이라 생각할 수 있습니다.

하지만 안타깝게도 이는 틀린 말입니다.

인스턴스를 만들면 인스턴스를 만들 때의 prototype과 인스턴스가 서로 묶여버리기 때문에 prototype 속성에 새 객체를 정의하면 이미 만들어진 인스턴스와 새로운 prototype 간의 연결고리가 끊어져 버리게 됩니다.

하지만 앞서 설명한 방법을 사용하면 인스턴스를 만들었던 시점의 prototype 객체를 찾아서 값을 수정하거나 추가할 수 있으며 이를 통해 이 prototype과 연결된 인스턴스를 갱신할 수 있습니다.

javascript

var Foo = function Foo(){};

Foo.prototype.x = 1;

var FooInstance = new Foo();

console.log(FooInstance.x); // 잠작한 대로 1이 기록된다.

// prototype 객체를 새로 만든 Object() object로 대체/재정의 해보자.

Foo.prototype = {x:2};

console.log(FooInstance.x); /\* 1이 기록된다. 엥? 우리가 방금 prototype을 갱신했으니 2가 되어야하지 않나? \*/

/\* FooInstance는 여전히 인스턴스로 만들어지던 시점의 prototype을 참조하고 있다. \*/

// Foo()의 인스턴스를 새로 만든다.

var NewFooInstance = new Foo();

// 새로 만든 인스턴스는 새로운 prototype 객체(={x:2};)와 묶여있다.

console.log(NewFooInstance.x); // 2가 기록된다.

여기서 알 수 있는 사실은 인스턴스를 만든 뒤에는 객체의 prototype 속성을 새 객체로 대체하면 안된다는 것입니다.

만약 새 객체로 대체해버리면 같은 생성자에서 만든 인스턴스라 해도 서로 다른 prototype을 참조하게 될 것입니다.

사용자 정의 생성자도 네이티브 생성자처럼 프로토타입을 상속할 수 있다

지금까지 자바스크립트에서 prototype 속성을 사용해 상속하는 법(E.g : Array.prototype)을 살펴보았습니다.

그리고 이와 같은 패턴을 네이티브가 아닌 사용자가 직접 만든 생성자에도 적용할 수 있습니다.

다음은 Person 객체를 가지고 자바스크립트의 상속 패턴을 흉내 내는 코드입니다.

javascript

var Person = function() {};

// 모든 Person 인스턴스는 legs, arms, countLimbs 속성을 상속한다.

Person.prototype.legs = 2;

Person.prototype.arms = 2;

Person.prototype.countLimbs = function() {return this.legs + this.arms;};

var chuck = new Person();

console.log(chuck.countLimbs()); // 4가 기록된다.

위 코드에서 Person() 생성자 함수를 만든 후 Person()의 prototype 속성에 몇 개의 속성을 추가하여 모든 인스턴스가 상속받도록 하였습니다.

이 코드는 자바스크립트 네이티브 객체를 상속할 때 사용했던 것과 똑같은 방법으로 프로토타입 체인을 사용한 것입니다.

전달된 매개변수가 없을 때 프로토타입에서 속성을 상속받는 생성자 함수를 만들어보면 이를 조금 더 잘 이해할 수 있을 것입니다.

다음 코드에서 Person() 생성자는 전달된 매개변수가 있으면 이를 사용해 인스턴스 속성을 추가하지만 전달된 값이 아예 없거나 한 개만 있으면 프로토타입에서 상속받은 값을 사용하도록 합니다. 인스턴스 속성이 있으면 상속된 속성이 사용되지 않는 것입니다.

따라서 어느 경우에든 속성을 문제없이 사용할 수 있을 것입니다.

javascript

var Person = function(legs, arms) {

// 프로토타입에서 상속받은 값을 가린다.

if (legs !== undefined) {this.legs = legs;}

if (arms !== undefined) {this.arms = arms;}

};

Person.prototype.legs = 2;

Person.prototype.arms = 2;

Person.prototype.countLimbs = function() {return this.legs + this.arms;};

var chuck = new Person(0, 0);

console.log(chuck.countLimbs()); // 0이 기록된다.

상속 체인 만들기

프로토타입 상속은 전통적인 객체 지향 프로그래밍 언어에서 볼 수 있던 상속 패턴을 흉내 내기 위해 만들어졌습니다.

자바스크립트에서 인스턴스란 간단히 말해 다른 객체의 속성에 접근할 수 있는 객체입니다.

이를 위해 먼저 상속받고자 하는 부모 객체의 인스턴스를 만든 후 생성자의 prototype에 할당하면 부모 객체를 상속받을 수 있습니다.

prototype 속성에 부모 객체의 인스턴스를 할당하고 나면 부모 객체 생성자의 prototype과 상속받은 객체 사이에는 연결고리(즉, \_\_proto\_\_)가 생깁니다.

다음 코드에서 Chef 객체(=cody)는 Person()을 상속받습니다.

다시 말해, 어떤 속성을 Chef 객체에서 발견하지 못하면 그 다음에는 Person() 생성자 함수의 prototype에서 속성을 찾는다는 뜻입니다.

이렇게 상속 관계를 만들기 위해 할 일은 Chef.prototype의 값으로 Person() 객체의 인스턴스를 할당해 주는 것이 전부입니다.

즉, Chef.prototype = new Person()

javascript

var Person = function() { this.bar = 'bar' };

Person.prototype.foo = 'foo';

var Chef = function() { this.goo = 'goo' };

Chef.prototype = new Person();

var cody = new Chef();

console.log(cody.foo); // 'foo'가 기록된다.

console.log(cody.goo); // 'goo'가 기록된다.

console.log(cody.bar); // 'bar'가 기록된다.

위 코드에서 한 일은 네이티브 객체에서 원래 사용하고 있던 시스템을 빌려온 것 뿐입니다.

prototype 속성에 있어서 원래 사용하던 Object() 값이나 Person() 값은 다른 점이 없습니다.

다시 말해, 객체의 상속된 속성에 접근하면 객체를 만든 생성자 함수의 prototype(=Object.prototype) 속성에서 그 값을 찾을 것이라는 의미입니다.

출처: <http://webclub.tistory.com/301> [Web Club]

## [함수 객체의 기본 프로퍼티](http://webclub.tistory.com/13)

10개월 전 by 하기실음 관두등가 재야재희

자바스크립트에서 함수는 객체다.

이것은 함수 역시 일반적인 객체의 기능은 물론이고 추가로 호출됐을 때 정의된 코드를 실행하는 기능을 가지고 있다는 것이다.

(나와 같은 초보에게 함수는 코드를 실행하는 기능으로 많이 사용하지만...)

또한, 일반 객체와는 다르게 추가로 함수 객체만의 표준 프로퍼티를 가지고 있다.

다음의 예제 코드로 함수 객체의 기본 프로퍼티들을 살펴 보자.

// add() 함수 정의  
  
**function** add(x,y) {  
 **return** x + y;  
}  
  
console.dir(add);

함수의 기본 프로퍼티를 알아보기에 앞서 console.dir()에 말하자면 console.log(), info(), warning()등 처럼 단순 로그를 보여주는 것과는 달리 프로퍼티가 있다면 객체의 프로퍼티들을 모두 보여준다.

이제 console.dir(add)의 결과를 살펴보면 add()함수는 arguemts, caller, length, name 등과 같은 다양한 프로퍼티가 기본적으로 자동 생성된 것을 확인할 수가 있다.

이러한 프로퍼티들이 함수를 생성할 때 생기는 표준 프로퍼티다.

name, caller, arguments

먼저 name, caller, arguments, \_\_proto\_\_ 프로퍼티를 살펴보자.

참고로 이 프로퍼티들은  ECMA 표준이 아님을 기억하자.

***name*** *프로퍼티는 함수의 이름을 나타내고 만약 이름이 없는 익명함수라면 빈 문자열이 나타낸다.*

***caller*** *프로퍼티는 자신을 호출한 함수를 나타내는데 위 코드에서는 add() 함수를 어떤 곳에서도 호출하지 않았기 때문에 null 값이 나왔다.*

***arguments*** *프로퍼티는 함수를 호출할 때 전달된 인자값을 나타내는데  add()함수가 호출된 상태가 아니라서 null값이 출력됐다.*

arguments 프로퍼티와 같은 이름으로 ES 표준에서는 arguments 객체를 정의하고 있는데 이는 arguments 객체를 공부할 때 다시 알아보자.

\_\_proto\_\_

다음으로 \_\_proto\_\_ 프로퍼티를 알아보자.

프로토타입에서 모든 자바스크립트 객체는 자신의 프로토타입을 가리키는 [prototype] 내부프로퍼티를 가지고 있다.

그리고 크롬브라우저에서 [prototype] 내부프로퍼티를 \_\_proto\_\_ 프로퍼티로 구현되어 있다.

add() 함수 역시 자바스크립트 객체이므로 [prototype] 내부프로퍼티인 \_\_proto\_\_ 프로퍼티를 가지고 있고 이 \_\_proto\_\_ 프로퍼티는 자신의 부모역할을 하는 프로토타입 객체를 가리키게 된다.

ECMA 표준에서는 add()함수와 같이 함수 객체의 부모역할을 하는 프로토타입 객체를 Function.prototype 객체라고 명명하고 있으며이 Function.prototype 객체 역시 함수 객체라고 정의하고 있다.

그리고 크롬 브라우저에서는 Function.prototype 객체를 Empty() 함수라고 정의하고 있다.

Empty()함수 역시 함수 객체이므로 name, caller, arguments 등과 같은 함수 객체의 프로퍼티들을 가지고 있다.

length, prototype

마지막으로 length와 prototype 프로퍼티에 대해 알아보자.

ECMA5 스크립트 명세서에는 모든 함수는 length, prototype 프로퍼티를 가져야 한다고 정의하고 있다.

여기서 주목할 점은 prototype 프로퍼티가 앞서 언급한 \_\_proto\_\_과는 다르므로 혼동하지 말아야 한다.

모든 함수는 객체로서 prototype 프로퍼티를 가지고 있다.

이 prototype 프로퍼티는 사용자가 함수를 정의하면 자동으로 prototype 프로퍼티가 만들어지며  constructor 프로퍼티 하나만 있는 객체를 가리킨다.

그리고 prototype 프로퍼티가 가리키는 프로토타입 객체의 유일한 constructor 프로퍼티는 자신과 연결된 함수를 가리킨다.

이 말인 즉슨, 자바스크립트에서 함수를 생성하면 함수 자신과 연결된 프로토타입 객체를 동시에 생성하며, 이 prototype 프로퍼티와 constructor 프로퍼티는 서로를 참조하게 된다.

prototype 프로퍼티와 \_\_proto\_\_ 의 차이점

두 프로퍼티는 모두 프로토타입 객체를 가리키는 공통점을 가지고 있으나 차이점은 있고 구분해야 한다는 점이다.

내부 프로퍼티인 \_\_proto\_\_(객체의 숨은 프로퍼티:암묵적 프로토타입 링크)는 객체 입장에서 자신의 부모인 프로토타입 객체를 가리키는 반면에, 함수 객체 자체가 가지는 명시적인 prototype 프로퍼티는 함수가 생성자로 사용될때 이 함수를 사용하여 만들어진 객체의 부모역할을 하는 프로토타입을 가리킨다는 점이다.

이 둘의 차이점을 좀더 자세히 알려면 자바스크립트의 객체를 생성하는 규칙을 알아야 한다.

우선 여기까지만 하고 prototype에 대한 것은 따로 챕터를 구성해 좀 더 보안하도록 하겠다.

그럼 마지막으로 length 프로퍼티에 대해 알아보고 끝내자.

함수 객체의 length 프로퍼티는 모든 함수가 가져야하는 표준 프로퍼티라고 언급했다.

이 length 프로퍼티는 함수가 실행될 때 기대되는 인자의 개수를 나타낸다.

다음의 예제 코드를 살펴보자.

// 함수 세 개를 정의  
**function** fn0() {  
  
}  
  
**function** fn1(a) {  
 **return** a;  
}  
  
**function** fn2(a,b) {  
 **return** a + b;  
}  
  
console.log(fn0.length, fn1.length, fn2.length)  
// 0, 1, 2

이렇게 출력값을 보면 함수 객체의 lengh 프로퍼티에는 함수 작성시 정의한 인자 개수를 나타내고 있다.

[\*출처 인사이드 자바스크립트\*]

출처: <http://webclub.tistory.com/13> [Web Club]

## [클래스 & 생성자, 메서드](http://webclub.tistory.com/157)

10개월 전 by 하기실음 관두등가 재야재희

클래스(=생성자) 메서드 정리

C++ 이나 Java와 같은 경우 class 라는 키워드를 제공하여 프로그래머는 클래스를 만들 수 있습니다.

클래스와 같은 이름의 메서드로 생성자를 구현해 내는 것입니다. 하지만 자바스크립트에는 이러한 개념이 없습니다.

하지만 자바스크립트의 거의 모든 것이 객체이고, 특히 함수 객체롤 많은 것을 구현해 내고 있기 때문에 클래스, 생성자, 메스드도 모두 함수로 구현이 가능합니다.

생성자

생성자는 new 연산자에 의해 호출되어 객체의 초기화를 담당합니다.

new 생성자함수() => 자바에서는 new 클래스이름()

다시 말해서, 자바스크립트에서는 클래스 키워드가 없기 때문에 생성자함수가 클래스이자 생성자의 역활을 수행합니다.

자바스크립트에서의 생성자의 역할

데이터의 값을 설정(변수 설정)하거나 메소드를 호출해서 객체를 사용할 수 있도록 준비하는 역할을 수행

다음의 예제를 살펴 보겠습니다.

javascript

function Person(arg) {

this.name = arg;

this.getName = function () {

return this.name;

}

this.setName = function (value) {

this.name = value;

}

}

var me = new Person('jaehee');

/\*\*

\* new 키워드로 새로운 객체 me 를 만들었음.

\* 함수 Person 이 클래스이자 생성자의 역할을 한다.

\*/

console.log(me.getName()); // jaehee

me.setName('JavaScript');

console.log(me.getName()); // JavaScript

위의 코드 형태는 기존 객체지향 프로그래밍 언어에서 한 클래스의 인스턴스를 생성하는 코드와 매우 유사합니다.

함수 Person 이 클래스이자 생성자의 역할을 동시에 하고 있으며 자바스크립트에서 클래스 기반의 객체지향 프로그래밍의 기본적인 형태가 이와 같습니다.

다시 말해서, 클래스 및 생성자의 역할을 하는 함수가 있고, 사용자는 new 키워드를 이용하여 인스턴스를 생성하여 사용할 수 있습니다. 예제 코드에서 생성된 me 는 Person 의 인스턴스로서 name 변수가 있고, getName() 과 setName() 함수가 있는 구조입니다.

하지만 이 예제에는 문제가 있습니다.

예를 들어서 이 Person 을 생성자로 하여 여러 개의 인스턴스인 객체를 생성한다고 가정해보겠습니다.

javascript

var me = new Person('me');

var you = new Person('you');

var her = new Person('her');

위와 같이 여러 개의 인스턴스를 생성하여 사용하면 겉으로는 별 문제없이 작동되어 질 것입니다. 하자만 각 객체(인스턴스)는 자기 영역에서 공통적으로 사용할 수 있는 setName() 함수와 getName() 함수를 각각의 객체마다 따로 생성하고 있습니다. 이는 불필요하게 중복되는 영역을 메모리에 올려놓고 사용하게 됨을 의미하고 자원 낭비와 성능을 떨어뜨리는 원인이 됩니다.

따라서 위와 같은 문제를 해결하려면 다른 방식을 접근이 필요한데, 여기서 활용할 수 있는 자바스크립트의 특성인 함수 객체의 프로토타입을 이용하는 것입니다.

다음의 코드를 살펴 보겠습니다.

javascript

function Person(arg) {

this.name = arg;

}

Person.prototype.getName = function () {

return this.name;

};

Person.prototype.setName = function (value) {

this.name = value;

};

var me = new Person('jaehee');

var you = new Person('you');

console.log(me.getName()); // jaehee

console.log(you.getName()); // you

위의 코드는 Person 함수객체의 prototype 프로퍼티에 getName() 과 setName() 함수를 정의하였습니다.

이 Person 으로 객체를 생성한다면 각 객체를 각자 따로  getName() 과 setName() 함수객체를 생성할 필요 없이 객체가 공통으로 가지는 공간인 prototype 객체의 공간인 한군데에서 setName() 과 getName() 함수에 접근하여 사용할 수 있습니다. 정확히는 프로토타입 체인을 통해 접근하는 것입니다.

이와 같이 자바스크립트에서 클래스 안의 메서드를 정의할 때는 프로토타입 객체에 정의한 후,  new 로 생성한 객체에서 접근할 수 있게 하는 것이 좋습니다.

**@ 객체(Object)와 클래스(Class)**

현실세계: 설계도 → 객체 (아파트를 짓기 위해 설계도면 필요, 설계도를 가지고 아파트가 완성)

자바: 클래스 → 객체

클래스에는 객체를 생성하기 위한 필드와 메소드가 정의되어 있다.

클래스로부터 만들어진 객체를 해당 클래스의 인스턴스(instance)라고 한다. ( 보통 객체를 인스턴스라고 부르기도 함)

객체는 독립적으로 사용하는 어떤 대상을 일컫는다면 인스턴스는 어떤 설계도를 바탕으로 만들어진 것인지에 포커스를 맞춘다

하나의 클래스로부터 여러 개의 인스턴스를 만들 수 있다.



자바스크립트에는 클래스가 없다.

클래스가 없으면 프로그램이 더 짧아진다. 객체를 생성하기 위해 클래스를 만들 필요가 없기 때문이다.

자바스크립트에서는 빈 객체를 필요한 시점에 생성하고 그 이후에 필요한 멤버를 추가할 수 있다.

객체는 원시 데이터 타입이나 함수, 다른 객체를 추가하여 객체의 프로퍼티를 구성한다.

빈 객체는 사실 완전히 비어있는 것이 아니다.

빈 객체는 몇몇 내장 프로퍼티를 가지고 있지만 자신이 직접 소유한 프로퍼티가 없을 뿐이다.

JAVA 의 객체 생성방법

JAVA

// JAVA 의 객체 생성

HelloOO hello\_oo = new HelloOO

출처: <http://webclub.tistory.com/157> [Web Club]

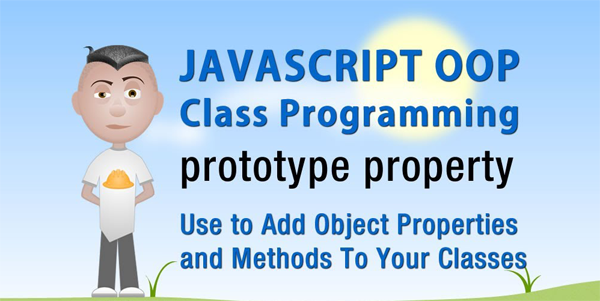
## [자바스크립트 프로토타입(Prototype)](http://webclub.tistory.com/15)

10개월 전 by 하기실음 관두등가 재야재희

Prototype

자바스크립트는 C++,자바와 같은 클래스를 정의하고 이를 통해 객체를 생성하여 객체지향 프로그래밍을 하는 것과는 달리 클래스란 개념이 없다.

대신에 객체 리터럴이나 생성자 함수로 객체를 생성하여 객체지향 프로그래밍을 지원하는데 이렇게 객체리터럴, 생성자 함수로 생성된 객체의 부모 객체가 바로 '프로토타입 객체'이다.



즉, 상속개념과 마찬가지로 자식 객체는 부모 객체가 가진 프로퍼티에 접근하거나 메소드를 상속받아서 호출하는 것이 가능하도록 해준다.

그래서 자바스크립트는 클래스 기반이 아닌 프로토타입 기반의 객체지향 프로그래밍을 지원가능하도록 해준다.

자바스크립트의 모든 객체는 객체 자신의 부모인 프로토타입 객체를 가리키는 숨겨진 프로퍼티가 있는데 이 숨겨진 프로퍼티가  prototype 프로퍼티이며 크롬브라우저에서는 \_\_proto\_\_란 형식으로 구현되어 있다.

여기서 구분해야될 점이 있는데 함수 객체의 기본 프로퍼티에서 서술했듯이 함수 객체 자신의 prototype 프로퍼티와숨겨진 protype 프로퍼티인 \_\_proto\_\_을 구분해야 한다는 점이다.

**이 둘의 차이점을 알아보도록 하자.**

prototype 프로퍼티(함수 객체) vs \_\_proto\_\_

자바스크립트는 객체를 생성하는 방법이 객체리터럴 방식과 생성자 함수를 이용한 방식이 있다.

우선 이 두가지 방식으로 객체를 생성했을 때의 규칙을 알아 볼 필요가 있다.

자바스크립트에서 모든 객체는 자신을 생성한 생성자 함수의 prototype 프로퍼티가 가리키는 프로토타입 객체를자신의 부모 객체로 나타내주는 \_\_proto\_\_로 연결되어 있다.

다음의 예제 코드를 살펴보자.

**생성자 함수 방식으로 객체 생성**

// Jaehee 생성자 함수 정의  
**function** Person(name) {  
 **this**.name = name;  
}  
  
// Person() 생성자 함수를 이용하여 jaehee란 객체를 생성  
**var** jaehee = **new** Person('jaehee!!!');  
  
// 프로퍼티를 알아보기 위해 dir 사용  
console.dir(Person);  
console.dir(jaehee);

크롬 브라우저의 콘솔창의 결과 화면을 보면  첫번째 콘솔 항목에 prototype: Person 과 두번째 콘솔 항목에 \_\_proto\_\_: Person을 확인할 수가 있다.

\* Person에서

prototype: Person의 prototype 프로퍼티는 함수가 생성될 때 만들어지며 constructor 프로퍼티 하나만 있는 객체를 가리킨다.

즉, 함수의 입장에서 자신과 링크된 프로토타입 객체를 가리키므로 Person.prototype과  Person()은 서로 참조하고 있다.

\*jaehee에서

\_\_proto\_\_: Person은 객체의 입장에서 자신의 부모 객체인 프로토타입 객체를 내부의 숨겨진 링크인 \_\_proto\_\_로 가리키고 있으며생성자 함수를 통해 생성된 객체이므로 위에 언급했듯이 자신의 부모 객체로 나타내주는  \_\_proto\_\_가 Person을 가리키고 있는 것으로 보아 Person.prototype이 jaehee의 프로토타입 객체인 것이다.

결국, 자바스크립트에서 객체를 생성하는 건 생성자 함수의 역할이지만, 생성된 객체의 실제 부모 역할을 하는 건 생성된 객체 자신이 아닌 생성자의 prototype 프로퍼티가 가리키고 있는 프로토타입 객체다.

다음으로 **객체 리터럴 방식**으로 객체를 생성해 보자.

// 객체 리터럴 방식으로 foo 객체 생성  
**var** foo = {  
 name : 'jaehee',  
 gender : 'man',  
 year : 2015  
}

크롬 브라우저의 개발자 도구에서 객체를 출력해보면 \_\_proto\_\_:  Object 인 것을 확인할 수 있다.

이렇게 foo객체의 프로토타입객체(\_\_proto\_\_)은 Object(실제로 Object.prototype)이다.

위의 출력 결과를 보면 알겠지만, 객체 리터럴 방식과 생성자 함수 방식의 차이가 프로토타입 객체(\_\_proto\_\_)에서 나타나고 있음을 알 수 있다. 예제 코드 출력상에서 객체리터럴 방식의 경우는 자신의 프로토타입 객체가 Object, 생성자 함수의 경우는  Person으로

서로 다름을 확인할 수 있다. 이렇게 차이가 발생하는 이유는 자바스크립트가 객체를 생성하는 규칙때문인 것이다.

자바스크립트 객체는 자신을 생성한 생성자 함수의 prototype 프로퍼티가 가리키는 객체를 자신의 프로토타입 객체로 설정한다.

객체 리터럴 방식에서는 객체를 생성하는 생성자 함수는 빌트인 Object()이며, 생성자 함수 방식의 경우는 생성자 함수 자체인 것이다.

그렇게 때문에 예제 코드에서 객체 생성 방법이 다르기 때문에 서로 다른 프로토타입 객체를 참조하고 있는 것이다.

자바스크립트에서는 일반 함수와 생성자 함수의 차이

일반함수와 생성자 함수는 별도의 차이가 없지만 new연산자를 붙여서 함수를 호출하면 생성자 함수로서 동작하는 것이다.

객체 생성을 목적으로 작성한 생성자 함수를 new 없이 호출하거나 일반 함수 생성을 목적으로 작성한 코드에 new를 붙여서 호출할 경우 코드에서 오류가 발생할 수가 있다.

그 이유는 생성자 함수 호출과 일반 함수를 호출할 때 this 바인딩 방식이 달라지기 때문이다.

일반 함수 호출의 경우는 this가 window 전역객체에 바인딩되는 반면에, 생성자 함수 호출에서 this는 new 생성자함수()를 통해 생성되는 객체에 바인딩되기 때문이다.

그래서 자바스크립트에서는 생성자 함수 작성시에 생성자 함수로서 코드가 작성됐음을 강조하기 위해서 함수명의 첫글자를 대문자로 사용하기를 권장하고 있다.

출처: <http://webclub.tistory.com/15> [Web Club]

## [객체 리터럴 방식의 클래스](http://webclub.tistory.com/139)

10개월 전 by 하기실음 관두등가 재야재희

객체(Object) 리터럴 방식의 클래스



문법

**var** 인스턴스 = {  
 프로퍼티 : 초기값,  
 . . .  
 메서드 : **function**(){  
 . . .  
 }  
}

특징 (객체 리터럴일 경우를 일컫는다.)

1) 인스턴스를 만드는 구문을 작성하지 않아도 되는 장점이 있습니다.

인스턴스를 만드는 구문

var 인스턴스 = new 클래스이름();

객체 리터럴 방식은 정의함과 동시에 인스턴스가 자동으로 만들어지기 때문에 인스턴스를 만드는 구문을 작성하지 않아도 되는 장점이 있습니다.

2) 이에 반면 인스턴스를 여러 개 만들 수 없는 단점이 있습니다.

예를 들어 여러 개의 탭 메뉴가 있을 경우

객체 리터럴 방식의 클래스 방법을 사용할 경우 인스턴스가 자동으로 생성되기 때문에 코드의 재사용성이 떨어지는 단점을 가지고 있습니다.

주 용도는 여러 개의 데이터 포장용

객체 안의 데이터를 만들어서 인스턴스로 전달하는 역할만 한다.

실무에서 객체 리터럴은 다음과 같이 사용되고 있습니다.

**var** $ch=$("#ch1");  
$ch.css("position","absolute");  
$ch.css("left",100);  
$ch.css("top",100);  
  
/\* 객체 리터럴 방식 \*/  
**var** $ch=$("#ch1");  
$ch.css({  
 position:"absolute",  
 left:100,  
 top:100  
  
})

다음의 예제는 사용자 이름, 아이디, 별명, 나이, 주소를 매개변수 값으로 받아서 콘솔창에 출력하는 함수입니다.

**1) 방법  - 객체 리터럴 사용 전**

/\* 객체 리터럴 사용 전 \*/  
**function** showInfo(userName, id, nickName, age, address) {  
 console.log("name = " + userName, "\nid = " + id, "\nnickName = " + nickName, "\nage = "  
 + age, "\naddress = " + address);  
}  
  
showInfo("김재희", "jaehee", "재야", 20, "잠실");

**2) 방법  - 객체 리터럴 사용 후**

/\* 객체 리터럴 사용 후 \*/  
**var** userInfo = {  
 userName : "재희",  
 id : "jaehee",  
 address : "잠실",  
 age : 20,  
 nickName : "재야"  
};  
  
// 함수에서 객체 리터털의 데이터를 받도록 합니다.  
**function** showInfo(info) {  
 console.log("name = " + info.userName, "\nid = " + info.id, "\nnickName = " + info.nickName,  
  
 "\nage = " + info.age, "\naddress = " + info.address);  
}  
  
// 함수 호출에 객체 리터럴 데이터를 넘겨줍니다.  
showInfo(userInfo);

**3)  방법 - 객체 리터럴 방식에 extend 를 적용**

/\* 객체 리터럴방식에 extend 적용 \*/  
  
// 기본 사용자 정보 값을 설정합니다.  
**var** defaults = {  
 userName : '미정',  
 id : '아이디??',  
 age : 0,  
 address : "어디에 사세요?",  
 nickName : ""  
}  
  
// 사용자 정의의 정보 갑을 설정합니다.  
**var** userInfo = {  
 userName : "김재희",  
 id : "jaehee",  
 address : '잠실'  
}  
  
// 함수에서 데이터 사용을 위한 로직  
**function** showInfo(info) {  
 // 기본 사용자 정보와 사용자 정의된 정보를 합치기(확장)  
 info = $.extend({}, defaults, userInfo);  
 console.log("name = " + info.userName, " id = " + info.id, "nickName = " + info.nickName,  
  
 " age = " + info.age, " address = " + info.address);  
}  
  
// 함수 호출(사용자 정의 정보값 전달)  
  
showInfo(userInfo);

출처: <http://webclub.tistory.com/139> [Web Club]

## [추상 클래스](http://webclub.tistory.com/176)

10개월 전 by 하기실음 관두등가 재야재희

추상클래스(Abstract Class)



추상 클래스 개념

추상(abstract)

*실체들 간에 공통되는 특성을 추출한 것을 말합니다.*

*예를 들어 새, 곤충, 물고기를 동물이라고 일컫는 것과 같이 공통되는 특성을 추출하여 의미론적으로 접근한 것입니다.*

*삼성, 현대, LG => 회사(추상)*

추상 클래스

*실체 클래스들의 공통되는 필드와 메소드를 정의한 클래스를 말합니다.*

*추상 클래스는 실체 클래스의 부모 클래스 역할을 할 수 있습니다.*

*추상 클래스는 단독으로 객체 생성을 할 수 없고, 부모 클래스로만 사용됩니다.*

추상 클래스의 용도

실체 클래스의 공통된 필드와 메소드의 이름을 통일할 목적

*실체 클래스를 설계하는 사람이 여러 사람일 경우,*

*실체 클래스마다 필드와 메소드가 제각기 다른 이름을 가질 수 있습니다.*

실체 클래스를 작성할 때 시간을 절약

*실체 클래스는 추가적인 필드와 메소드만 선언하면 됩니다*

실체 클래스 설계 규격을 만들고자 할 때

*실체 클래스가 가져야할 필드와 메소드를 추상 클래스에 미리 정의해 놓고 실체 클래스는 추상 클래스를 무조건 상속 받아 작성토록 합니다.*

출처: <http://webclub.tistory.com/176> [Web Club]

## [객체 지향 프로그래밍의 상속과 다형성](http://webclub.tistory.com/169)

10개월 전 by 하기실음 관두등가 재야재희

상속(Inheritance)

현실 세계에서 상속이란 부모가 가지고 있는 재산을 자식에게 물려주는 행위를 말합니다.



**@ 객체지향 프로그래밍에서 상속**

*상위(부모) 객체의 필드(속성)와 메소드를 하위(자식) 객체에게 물려주는 행위*

*하위 객체는 상위 객체를 확장해서 추가적인 필드(속성)와 메소드를 가질 수 있음*

*상속 대상: 필드(속성)와 메소드*

**@ 상속의 효과**

*상위 객체를 재사용해서 하위 객체를 빨리 개발할 수 있도록 한다.*

*반복된 코드의 중복을 줄여준다*

*유지 보수의 편리성을 제공해 준다.*

*객체의 다형성을 구현할 수 있다.*

즉, 하위 객체에서 상위객체의 필드와 메소드를 따로 또 만들어서 사용할 필요가 없으며 상위 객체의 메소드에 문제가 있을 경우 상위 객체에서의 수정으로 인해 하위객체에서 문제가 있었던 메소드를 따로 수정할 필요없이 메소드를 사용할 수 있습니다.

다형성(Polymorphism)

**@ 같은 타입이지만 실행 결과가 다양한 객체를 대입할 수 있는 성질을 말합니다.**

*부모 타입에는 모든 자식 객체가 대입*

*인터페이스 타입에는 모든 구현 객체가 대입*

**@ 효과**

*객체를 부품화시킬 수 있다.*

*유지보수가 용이하다.*

자동차는 타이어 타입 설계를 했다면 타이어를 부품화 시킨다고 봅니다. 그래서 타이어 타입으로 한국 타이어(부품)와 금호 타이어(부품)를 사용할 수 있지만 각 타이어의 성능을 다르게 나온다는 것이 다형성 측면입니다.

출처: <http://webclub.tistory.com/169> [Web Club]

## [객체지향 프로그래밍의 캡슐화, 상속, 다형성](http://webclub.tistory.com/156)

10개월 전 by 하기실음 관두등가 재야재희

캡슐화란? 은닉성이다!!!

캡슐화는 **중요한 데이터를 보존,보호하는 것**입니다.

캡슐화(encapsulation)는 **일반적으로 연관 있는 변수와 함수를 클래스로 묶는 작업**을 말합니다. 그런데 이 작업은 클래스 만드는 작업과 비슷하다고 여길 수도 있습니다. 하지만 캡슐화에는 은닉성이란게 있어서 클래스에 담는 내용 중 중요한 데이터나 기능을 외부에서 접근하지 못하게 할 수 있습니다.

일반 OOP에서 지원하는 캡슐화

일반 OOP 언어에서는 접근지정자를 제공

* public
* protected
* private

자바스크립트에서 캡슐화

* 기본 public
* private,protected 에 \_ 붙여 선언

JavaScript

function MyClass() {

//public 프로퍼티

this.프로퍼티이름 = 값;

//private/protected 프로퍼티

this.\_프로퍼티이름 = 값;

}

// public 메서드

MyClass.prototype.메서드이름 = function() {

}

// private/protected 메서드

MyClass.prototype.\_메서드이름 = function() {

}

\_(underBar) 를 붙이는 것은 외부에서 언더바를 붙힌 프로퍼티나 메서드에 접근하여 사용하지 말라는 암묵적인 약속입니다.

클래스(생성자함수)명의 시작문자를 대문자로 사용하는 것과 마찬가지입니다.

캡슐회는 만일의 상황(타인이 외부에서 조작)을 대비해서 외부에서 특정 속성이나 메서드를 시용자가 사용할 수 없도록 숨겨놓은 것입니다.

**캡슐화**

* 객체의 필드(속성), 메소드를 하나로 묶고, 실제 구현 내용을 외부에 감추는 것을 말한다.
* 외부 객체는 객체 내부의 구조를 얻지 못하며 객체가 노출해서 제공하는 필드와 메소드만 이용할 수 있다.
* 필드와 메소드를 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록 하는데 있다.
* 자바 언어는 캡슐화된 멤버를 노출시킬 것인지 숨길 것인지를 결정하기 위해 접근 제한자(Access Modifier)를 사용한다.

상속(Inheritance)이란

현실 세계에서 상속이란 부모가 자식에게 물려주는 행위, 부모가 자식을 선택해서 물려주는 행위이지만 객체지향 프로그래밍에서의 상속은 현실 세계와 반대로 자식이 부모를 선택해서 물려받는 것을 말합니다.

* 자식(하위,파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
* 자식이 부모를 선택해서 물려 받는 것
* 상속 대상 : 부모의 필드와 메소드

**상속의 효과**

* 부모 클래스를 재사용해서 자식 클래스를 빨리 개발할 수 있다.
* 반복된 코드의 중복을 줄여준다.
* 유지 보수의 편리성을 제공해 준다. (부모 클래스를 한 번만 수정함으로써 자식클래스를 수정할 필요가 없음)
* 객체의 다형성을 구현할 수 있다.

객체 지향 프로그래밍의 다형성(polymorphism)

다형성은 **같은 타입이지만 실행 결과가 다양한 객체를 대입(이용)할 수 있는 성질**을 말합니다.

부모타입에는 모든 자식 객체가 대입될 수 있으며 자식 타입은 부모 타입으로 자동 타입 변환이 됩니다.

다형성은 객체를 부품화시킬 수 있습니다.

다시말해서, 자동차란 객체가 있을 경우자동차를 설계를 할 때 타이어 부분에 사용된 부분을 타이어 타입이라고 가정한다면  자동차의 타이어 타입으로 한국 타이어와 금호 타이어가 있다고 한다면 어떤 타이어 타입을 사용하느냐에 따라 각 타이어의 성능은 다르게 나올 수 있다는 것이 다형성이라는 측면입니다.

그래서 한국, 금호 타이어는 자식 객체로써 객체를 부품화시킬 수 있다는 것입니다.

****

출처: <http://webclub.tistory.com/156> [Web Club]

[**객체 지향 프로그래밍의 추상화**](http://webclub.tistory.com/137)

10개월 전 by 하기실음 관두등가 재야재희

객체 지향 프로그래밍 - OOP

객체지향 프로그래밍이란 캡슐화, 다형성, 상속을 이용하여 코드 재사용을 증가시키고, 유지보수를 감소시키는 장점을 얻기 위해서 객체들을 연결시켜 프로그래밍 하는 것 입니다.

이 내용은 책, 검색을 통해서 무수히 나오는 개념들입니다. 하지만 퍼블리셔란 직군에서 스타트한 제게는 이런 개념이 생소하기만 하며 어렵게 느껴집니다. 이 포스팅에서는 간단히 퍼블리셔 직군에서 자바스크립트를 사용하여 OOP 에 좀 더 가깝게 다가가 보자란 의미로 접근해 보도록 하겠습니다.



먼저 객체란 무엇인가?

객체는 실제 세계(real world) 를 모델링 하는 것입니다.

OOP 는 실세계르 모델링 하기 위해 추상화(Abstraction)을 사용하는 프로그래밍 패러다임입니다.

추상화란 말 그대로 상세한 정보는 무시하고 필요성에 의해 있어야할 정보들만 간추려서 구성하는 것입니다.

마우스를 예를 들어보겠습니다.

지금은 실제 셰계에 마우스가 모델링되어 있지만 마우스가 만약 없다면 그것을 모델링 하고 싶다고 가정해보겠습니다.

컴퓨터를 좀 더 편하게 다루기 위해 클릭을 했으면 하고 스크롤도 있으면 좋겠다란 생각으로 정의를 시작할 수 있습니다.

좀 더 추상화를 해보자면 왼쪽 클릭영역, 오른쪽 클릭영역, 스크롤영역등... 이런 식으로 설계를 하는 것이 추상화 작업입니다.

단순히 있어야할 정보들을 간추려서 구성하는 것입니다.

이렇게 마우스의 추상화 작업으로 인해 왼,오른쪽 영역, 스크롤 등을 프로그래밍에서는 속성,프로퍼티라고 부릅니다.

다른 예로 사람을 추상화 작업한다고 하면

**■ 사람 객체**

1. 심장

2. 눈, 코, 입

3. 이름

4. 성별

5. 신장(키)

6. 생각한다

7. 뛴다

8. 잔다.

등등 사람을 객체화하여 추상화를 할 수 있습니다. 하지만 사람은 날 수 없으므로 '날다' 라고 추상화를 할 수는 없겠죠!

이렇게 공통적으로 가질 수 있는 정보들을 추상화한다고 하며 프로그래밍의 세계로 접근,접목 시킨 것이 객체지향 프로그래밍에서의 추상화입니다.

위의 사람 객체에서 정의한 리스트들을 프로그래밍적으로 프로퍼티(속성)이리고 하며 프로퍼티 중에 '생각하다', '뛴다', '잔다' 라는 동사적인 행위들을 프로그래밍에서는 메서드라고 부르는 것입니다. ( method(방법) - 뛰게하는 방법, 생각하게 하는 방법 )

이렇듯 추상화란 모든 객체에 공통적인 성질들만 담고 있어야 합니다!!

추상화란 공통점을 뽑아내는 것입니다.

- 객체 지향 프로그래밍에서 추상화(Abstraction)는 객체들의 공통적인 프로퍼티와 메서드를 뽑아내는 작업

- 코드상에서 구현(로직) 부분을 제외한 오직 선언 부분만을 설계하는 것을 말합니다.

자바스크립트에서의 추상화란

자바스크립트에서는 추상 클래스와 인터페이스를 제공하지 않기 때문에 오직 클래스 만을 이용해서 추상화 작업을 해야 합니다.  클래스만을 사용해야 하기 때문에 인터페이스의 기능인 클래스가 구현하고 있어야 하는 규약 등을 적용할 수 없습니다.

자바스크립트는 클래스와 비슷하게 만들어서 사용하는 방법 세 가지를 제공합니다.

1) 리터럴 방식

2) 함수활용 방식

3) 프로토타입 방식

출처: <http://webclub.tistory.com/137> [Web Club]

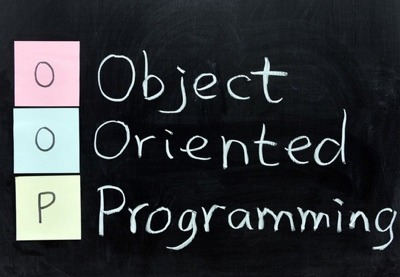
## [자바스크립트 객체지향 프로그래밍의 다형성](http://webclub.tistory.com/406)

10개월 전 by 하기실음 관두등가 재야재희

다형성 알아보기

다형성은 객체지향 프로그래밍 특징 중 가장 핵심적인 기능이라고 할 수 있습니다.

다형성의 개념을 학습하고 자바스크립트에서 다형성을 사용하는 방법에 대해 알아봅니다.



다형성이란?

다형성(polymorphism)은 특정 기능을 **선언(설계)부분**과 **구현(동작)부분**으로 분리한 후 구현부분을 다양한 방법으로 만들어 선택해서 사용할 수 있게 하는 기능힙니다.

여기서 선언부분은 **인터페이스**라고도 합니다.

좀더 자세히 설명하면 선언부분은 구현코드가 전혀 없는 텅 빈 상태이며 일종의 지켜야 할 약속(규약)으로 가득 찬 일종의 규약 문서입니다.

개발자는 문제를 해결하는 구현 코드를 선언부분에 맞게 구현하기만 하면 됩니다.

즉, 선언부분과 구현 부분은 1 : N 의 다형성 관계가 형성됩니다.

이해를 돕기 위해 다형성에 대한 예제를 들어보도록 하겠습니다.

멀티탭과 전원 케이블

다형성의 예는 현실에서도 쉽게 찾아 볼 수 있습니다.

주변에 가장 많이 사용되는 멀티탭을 생각해 봅니다.



멀티탭을 다형성 개념으로 접근한다면 선언부분은 구멍이 두 개 뚫린 부분이라고 할 수 있습니다.

구현 부분은 멀티탭에 연결하는 전원 케이블입니다.

이 멀티탭에 연결하려면 전원케이블(e.g 모니터전원케이블)은 반드시 연결 부분이 두 개 튀어 나와 있어야 하며, 튀어 나와 있는 부분이 한 개거나 세 개여서는 절대 안됩니다.

그리고 튀어 나온 부분이 사각형이어도 안되고 반드시 둥그런 원형이어야 합니다.

참고로 선언부분을 실제 코드로 만들 때 객체지향 프로그래밍의 기본 기능인 인터페이스(interface)와 추상 클래스가 사용되며 구현 부분은 일반 클래스와 추상 클래스가 사용됩니다.

이 내용은 이후 다뤄보도록 하겠습니다.

USB와 USB 기기들

다형성은 여러분이 즐겨 사용하는 USB 기기에서도 찾을 수 있습니다.

모든 USB 기기는 USB 규격에 맞춰 만들어져 있습니다. 또한 USB를 연결해서 사용하는 컴퓨터 역시 USB 규격에 맞춰 만들어져 있습니다.

USB 규격에 맞춰 만들어져 있는 기기라면 그 어떤 기기라도 연결해서 사용할 수 있습니다.

심지어 아직 출시되지 않은 USB 제품까지 말입니다.

여기서 USB 규격은 설계 부분인 인터페이스에 해당하며 USB기기들은 구현 부분을 담당하게 됩니다.

다형성과 데이터 타입과의 관계

이번에는 다형성과 데이터 타입과의 관계에 대해 알아봅니다.

자바스크립트에서는 데이터 타입이 중요하게 사용되진 않지만 일반 프로그래밍에서 데이터 타입은 매우 중요합니다.

일반 객체지향 프로프래밍에서의 데이터 타입의 의미

먼저 자바스크립트에서 변수를 선언할 때 데이터 타입을 작성하진 않지만 일반 프로그래밍에서는 대부분 변수에 저장할 데이터 타입을 미리 작성해야 합니다.

그리고 이와 같은 개념은 함수와 메서드에서도 그대로 사용됩니다.

만약 함수를 만든 후 매개변수에 문자열 타입으로 선언하는 경우에도 문자열 데이터 값으로 메서드를 호출해야 하지 다른 값으로 하는 경우 에러가 발생하게 됩니다.

이처럼 일반 프로그래밍에서의 데이터 타입은 일종의 반드시 지켜야할 규약처럼 사용됩니다.

일반 객체지향 프로그래밍에서 데이터 타입은 다음과 같은 규칙을 가지고 있습니다.

* 변수 및 매개변수 선언 시 특정 인터페이스 타입인 경우 반드시 특정 인터페이스를 구현한 클래스의 인스턴스만을 저장하거나 매개변수 값으로 넘길 수 있습니다.
* 변수 및 매개변수 선언 시 특정 클래스 타입인 경우 반드시 특정 클래스 또는 특정 클래스를 상속받은 자식 클래스의 인스턴스만을 저장하거나 매개변수 값으로 넘길 수 있습니다.

자바스크립트에서 다형성

자바스크립트에서 다형성을 구현하는 방법에 대해 간단하게 알아보겠습니다.

먼저 일반 객체지향 프로그래밍의 다형성 관련 문법을 살펴보면 선언부분과 구현부분으로 나눠 구성되며 다음과 같이 사용됩니다.

* 다형성 **선언**부분 : 인터페이스(Interface)와 추상클래스(abstract class)
* 다형성 **구현**부분 : 클래스(class)

자바스크립트에서 다형성 지원 유무

자바스크립트는 인터페이스와 추상 클래스 그리고 엄격한 데이터 타입 체크 등의 다형성과 관련된 문법을 젼혀 지원하지 않습니다.

그렇기 때문에 그저 자바스크립트에는 다형성을 지원한다는 가정하에 다형성스럽게 코드를 만든다는 것입니다.

문법만 제공하지 않을 뿐 다형성 개념을 사용해 코드를 만들게 되면 좀더 월등한 프로그래밍을 할 수 있을 것입니다.

자바스크립트에서 다형성 개념을 적용하여 사용하면 일반적으로 다음과 같은 구조를 갖게 됩니다.

javascript

// 자바스크립트는 선언부분이 없습니다.

// 그냥 선언부분이 있다가 생각하고 작성한다.

// 합성을 이용해 구현부분을 사용하는 클래스

function 이미지정렬(선언부분) { // 이 부분을 선언부분이라고 가정하고 사용한다.

선언부분.기능1();

선언부분.기능2();

}

/\*\*

\* -----------------------------------------------------

\* 구현 부분

\* 구현도 선언부분이 있다는 가정하에 선언부분에 맞춰 구현한다.

\* -----------------------------------------------------

\*/

// 구현 부분 클래스 첫번째 묶음

function 기능1() {

// 구현 코드

}

function 기능1() {

// 구현 코드

}

// 구현 부분 클래스 두번째 묶음

function 기능2() {

// 구현 코드

}

function 기능2() {

// 구현 코드

}

// 구현 부분 클래스 세번째 묶음

function 기능3() {

// 구현 코드

}

function 기능3() {

// 구현 코드

}

다형성을 이용하여 심플 갤러리 만들어보기

위에서 알아본 다형성을 간단한 예제를 통해 검증하는 시간을 가져보도록 합니다.

다형성의 장점을 확인하기 위해 다형성을 적용하지 않은 경우와 다형성을 적용한 경우로 나누어 단계별로 진행해 보도록 하겠습니다.

**심플 갤러리 요구사항**

* 버튼을 누르면 버튼에 맞는 이미지 정렬 기능을 구현하려고 한다.
* 버튼은 4개로 가로정렬, 세로정렬, 랜덤정렬, 바둑판정렬 기능 동작을 요구한다.

기본 마크업과 스타일을 다음과 같습니다.

html

<div class="btn-gruop">

<button id="btnHorizontal">가로정렬</button>

<button id="btnVertical">세로정렬</button>

<button id="btnRandom">랜덤정렬</button>

<button id="btnGrid">바둑판정렬</button>

</div>

<div class="image-container" id="container1">

<img src="src/images/1.jpg">

<img src="src/images/2.jpg">

<img src="src/images/3.jpg">

<img src="src/images/4.jpg">

<img src="src/images/5.jpg">

<img src="src/images/6.jpg">

<img src="src/images/7.jpg">

<img src="src/images/8.jpg">

</div>

css

.btn-gruop {

margin-bottom: 10px;

}

.btn-gruop button {

background-color: #222;

color: #fff;

border-radius: 4px;

border: 1px solid #ccc;

line-height: 1;

padding: 5px 10px;

}

div.image-container {

position: relative;

border: 1px solid #000;

}

div.image-container img {

position: absolute;

left: 0;

top: 0;

width: 120px;

}

다음의 코드는 이미지 가로 정렬 기본 기능입니다.

javascript

$(function(){

var gallery = new Gallery('#container1 img')

var $btnHorizn = $('#btnHorizontal');

$btnHorizn.on('click', function () {

gallery.show();

});

});

function Gallery(selector) {

this.$images = null;

this.init(selector)

}

Gallery.prototype.init = function (selector) {

this.$images = $(selector);

};

// 이미지를 정렬 기본 기능

Gallery.prototype.show = function () {

// 이미지 개수 구하기

var length = this.$images.length;

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

var $img = this.$images.eq(i);

var x = i \* 150;

$img.css({

left : x

})

}

};

위에서 요구한 4가지 정렬 기능을 다형성을 적용하지 않은 각각의 메서드를 만들어 구현한 코드입니다.

javascript

$(function(){

var gallery = new Gallery('#container1 img');

var $btnHorizn = $('#btnHorizontal'),

$btnRandom = $('#btnRandom'),

$btnVertical = $('#btnVertical'),

$btnGrid = $('#btnGrid');

$btnHorizn.on('click', function () {

gallery.show('horizontal');

});

$btnVertical.on('click', function () {

gallery.show('vertical');

});

$btnRandom.on('click', function () {

gallery.show('random');

});

$btnGrid.on('click', function () {

gallery.show('grid');

});

});

function Gallery(selector) {

this.$images = null;

this.init(selector)

}

Gallery.prototype.init = function (selector) {

this.$images = $(selector);

};

// 이미지 정렬 방향 처리

Gallery.prototype.show = function (type) {

switch (type) {

case 'horizontal' :

this.horizontal();

break;

case 'vertical' :

this.vertical();

break;

case 'random' :

this.random();

break;

case 'grid' :

this.grid();

break;

default :

this.horizontal();

}

};

// 이미지 가로 정렬 처리

Gallery.prototype.horizontal = function () {

// 이미지 개수 구하기

var length = this.$images.length;

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

var $img = this.$images.eq(i);

var x = i \* 150;

$img.css({

left : x

})

}

};

// 이미지 세로 정렬 처리

Gallery.prototype.vertical = function () {

// 이미지 개수 구하기

var length = this.$images.length;

// 이미지 배열하기

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

var $img = this.$images.eq(i);

var y = i \* 150;

$img.css({

left: 0,

top : y

})

}

};

// 이미지 랜덤 정렬 처리

Gallery.prototype.random = function () {

// 이미지 개수 구하기

var length = this.$images.length;

// 이미지 배열하기

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

var $img = this.$images.eq(i);

var x = 200 \* Math.random();

var y = 200 \* Math.random();

$img.css({

left: x,

top : y

})

}

};

// 이미지 바둑판 정렬 처리

Gallery.prototype.grid = function () {

// 이미지 개수 구하기

var length = this.$images.length,

count = 3;

// 이미지 배열하기

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

var $img = this.$images.eq(i);

var x = (i % count) \* 150;

var y = parseInt(i / count) \* 150;

$img.css({

left: x,

top : y

})

}

}

위 코드를 자세히 살펴보면 네 개의 이미지 정렬기능이 모두 Gallery 생성자(클래스) 내부에 구현되어 있습니다.

이는 다음과 같은 문제점이 있습니다.

**사용하지 않는 코드가 많다.**

* 세로, 가로, 바둑판식, 랜덤 이미지 정렬까지 네 가지의 이미지 정렬 기능이 모두 Gallery 클래스에 구현되어 있기 때문에 기능 하나를 선택하면 세 개의 기능은 사용하고 있지 않게 되어 불필요한 코드가 됩니다.

**코드 재사용성이 떨어진다.**

* Gallery 내부에 이미지 정렬 기능이 모두 구현되어 있기 때문에 만약 이미지 정렬 기능을 독립적으로 사용하고 싶어도 사용할 수가 없습니다. 원하는 정렬 기능을 사용하려면 반드시 Gallery 클래스의 인스턴스를 만들어야 합니다.

**유지보수가 어렵다.**

* 네 가지 정렬 기능 이외에 새로운 이미지 정렬 기능을 추가해야 하는 경우 Gallery 클래스에 추가해서 구현해야 하기 때문에 클래스 기능이 점점 거대해져 유지보수가 어려워집니다.

다형성을 적용한 코드

참고로 일반적으로 다형성 적용 전 코드와 같이 switch를 활용해 여러 조건을 처리하는 코드는 다형성으로 바꿀 수 있는 전형적인 경우입니다.

위 코드를 가지고 다형성 적용을 위해 5단계로 나눠보도록 합니다.

* 단계 01 : 다형성 선언부분 만들기
* 단계 02 : 다형성 구현부분 만들기(정렬기능 구현)
* 단계 03 : 다형성과 합성을 활용한 정렬 기능 연결
* 단계 04 : 정렬 기능 호출
* 단계 05 : 독립적으로 정렬 기능을 사용할 수 있는지 확인하기
* 단계 06 : 정리

단계 01 : **다형성 선언부분 만들기**

다형성을 만드는 첫 번째 작업은 구현 부분에 공통적으로 구현해야 하는 기능을 선별해 선언부분으로 만드는 작업입니다.

지금 이미지 정렬 갤러리의 경우는 정렬 기능을 선언부분으로 만들어주면 좋을 것입니다.

javascript

function align($images) {

// 이곳에 정렬 기능을 구현한다

}

정리하자면 선언부분은 앞에서 알아본 것처럼 반드시 지켜야할 약속으로써 구현부분에서는 반드시 선언부분에 정의되어 있는 메서드의 이름과 매개변수 개수까지 동일하게 메서드를 정의해서 구현해야 합니다.

또한 자바스크립트의 경우 다형성의 선언부분을 정의하는 문법을 제공하지 않기 때문에 선언부분을 만들었다는 가정을 하고 진행하도록 합니다.

다시 말해, 위 코드는 실제 만들었다는 가정하에 진행하기 때문에 실제 코드에는 존재하지 않습니다.

단계 02 : **다형성 구현부분 만들기**

이번에는 선언부분에 맞게 코드를 구현하는 단계입니다.

아래 코드와 같이 네 가지의 정렬 기능 클래스를 생성한 후 선언부분에 작성된 메서드와 동일하게 메서드를 추가해 정렬 기능을 구현해 주도록 합니다.

인스턴스는 여러 개 생성할 필요없이 하나만 생성하면 되니 프로토타입 방식보다 리터럴 방식을 이용해 클래스를 만들어 보도록 하겠습니다.

javascript

// 가로 정렬 기능

var horizontal = {

align : function ($images) {

// 이미지 개수 구하기

var length = $images.length;

// 이미지 배열하기

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

var $img = $images.eq(i);

var x = i \* 150;

$img.css({

left: x,

top : 0

});

}

}

};

// 세로 정렬 기능

var vertical = {

align : function ($images) {

// 이미지 개수 구하기

var length = $images.length;

// 이미지 배열하기

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

var $img = $images.eq(i),

y = i \* 150;

$img.css({

left: 0,

top : y

})

}

}

};

// 랜덤 정렬 기능

var random = {

align : function ($images) {

// 이미지 개수 구하기

var length = $images.length;

// 이미지 배열하기

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

var $img = $images.eq(i),

x = 200 \* Math.random(),

y = 200 \* Math.random();

$img.css({

left: x,

top : y

})

}

}

};

// 바둑판 정렬 기능

var grid = {

align : function ($images) {

// 이미지 개수 구하기

var length = $images.length,

count = 3;

// 이미지 배열하기

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

var $img = $images.eq(i),

x = (i % count) \* 150,

y = parseInt(i / count) \* 150;

$img.css({

left: x,

top : y

})

}

}

}

단계 03 : **다형성과 합성을 활용한 정렬 기능 연결**

이제 Gallery 클래스의 show() 메서드에 다형성을 적용해 주기 위해 정렬 기능을 구현한 클래스의 인스턴스를 받을 수 있는 매개변수를 추가해 주도록 합니다.

javascript

// 이미지 정렬 기능 실행

Gallery.prototype.show = function (alignable) {

alignable.align(this.$images);

};

정렬 기능 버튼을 누를 때 선택한 정렬 기능이 alignable 매개변수로 넘어와 사용될 것입니다.

즉 합성을 사용해 정렬 기능을 빌려 사용하게 되는 것입니다.

단계 04 : **정렬 기능 호출**

정렬 버튼 클릭시 버튼에 맞는 이미지 정렬 기능(align)의 인스턴스를 매개변수 값으로 show() 메서드를 호출해 줍니다.

javascript

$(function(){

var alignImg = new Gallery('#container1 img');

var $btnHorizn = $('#btnHorizontal'),

$btnRandom = $('#btnRandom'),

$btnVertical = $('#btnVertical'),

$btnGrid = $('#btnGrid');

$btnHorizn.on('click', function () {

alignImg.show(horizontal); // 가로정렬 인스턴스 전달

});

$btnVertical.on('click', function () {

alignImg.show(vertical); // 세로정렬 인스턴스 전달

});

$btnRandom.on('click', function () {

alignImg.show(random); // 랜덤정렬 인스턴스 전달

});

$btnGrid.on('click', function () {

alignImg.show(grid); // 바둑판 정렬 인스턴스 전달

});

});

단계 05 : **독립적으로 정렬 기능을 사용할 수 있는지 확인하기**

마지막으로 테스트 하나를 해보겠습니다.

정렬 기능이 Gallery 내부에 구현되어 있지 않고 외부에 독립적으로 구현되어 있기 때문에 랜덤 정렬 기능을 다음과 같이 독립적으로 사용할 수 있을 것입니다.

javascript

$(function(){

var $btnRandom = $('#btnRandom');

$btnRandom.on('click', function () {

// 독립적으로 랜덤 정렬 기능만 사용하기

random.align($('#container1 img'));

});

});

단계 06 : **정리**

마지막으로 지금까지 작성한 코드를 요약해서 살펴보겠습니다.

이미지 정렬 기능은 Gallery 클래스 내부에 구현되어 있지 않고 각각 독립적인 클래스로 구현되어 있다는 점입니다.

이로써 다형성을 적용해 구현한 갤러리는 다음과 같은 장점이 있습니다.

가벼워진 코드

Gallery 클래스 내부에 이미지 정렬 기능을 직접 구현하지 않고 모두 외부에 구현되어 있기 때문에 많이 가벼워집니다. 이와 동시에 코드를 간결하게 만들 수 있습니다.

코드 재사용성이 높아진다.

이미지 정렬 기능이 각각 독립적으로 구현되어 있어서 Gallery 도움없이 이미지 정렬 기능을 사용할 수 있습니다.

유지보수가 쉬워짐

새로운 이미지 정렬 기능을 추가해야 하는 경우 이미지 정렬 기능 인터페이스에 맞게 클래스를 만들어 Gallery에 연결해서 사용하면 됩니다. Gallery 클래스를 전혀 수정하지 않고 말입니다.

Result View

See the Pen &amp;amp;lt;a href='http://codepen.io/jaehee/pen/JKYbxB/'&amp;amp;gt;다형성&amp;amp;lt;/a&amp;amp;gt; by jaeheekim (&amp;amp;lt;a href='http://codepen.io/jaehee'&amp;amp;gt;@jaehee&amp;amp;lt;/a&amp;amp;gt;) on &amp;amp;lt;a href='http://codepen.io'&amp;amp;gt;CodePen&amp;amp;lt;/a&amp;amp;gt;.

출처: <http://webclub.tistory.com/406> [Web Club]

## [생성자 함수 상속- inheritance](http://webclub.tistory.com/402)

10개월 전 by 하기실음 관두등가 재야재희

자바스크립트 - 생성자 함수 상속 기능

이 포스팅에서는 자바스크립트에서 생성자 함수를 이용하여 상속 기능을 어떤 방법으로 구현하는가를 간단한 예제를 보면서 정리해 봅니다.

상속은 다음과 같이 세 가지 기능을 가지고 있습니다.

* 코드의 재사용
* 중복코드 제거
* 확장

위의 기능을 어떻게 구현하는지에 대한 과정을 예제를 통해 알아봅니다.



생성자 함수의 상속

코드의 재사용과 상속을 하기에 앞서 다음과 같은 코드가 있다가 가정해 봅니다.

javascript

// ConstructorA

function ConstructorA() {

this.num01 = 1234;

}

ConstructorA.prototype.func01 = function () {

console.log('this.num01 은 ' + this.num01);

};

ConstructorA.prototype.func02 = function () {

console.log('func02 는 ConstructorA 에서만 정의한 메소드');

};

// ConstructorB

function ConstructorB() {

this.num01 = 1234;

}

ConstructorB.prototype.func01 = function () {

console.log('this.num01 은 ' + this.num01);

};

ConstructorB.prototype.func02 = function () {

console.log('func02 는 ConstructorB 에서만 정의한 메소드');

};

// 인스턴스 생성

var ConstA = new ConstructorA();

// 프로퍼티와 메서드 접근

ConstA.func01(); // this.num01 = 1234

ConstA.func02(); // func02 는 ConstructorA 에서만 정의한 메소드

// 인스턴스 생성

var ConstB = new ConstructorB();

// 프로퍼티와 메서드 접근

ConstB.func01(); // this.num01 = 1234

ConstB.func02(); // func02 는 ConstructorB 에서만 정의한 메소드

위 코드를 살펴보면 중복코드가 발생하고 있는 것을 알 수 있습니다.

이 중복코드를 제거하고 상속하는 것을 다음의 코드를 통해 알아봅니다.

javascript

// 부모로 사용할 생성자 함수를 정의한다(이곳에 중복되는 코드를 담는다)

function ParentConstructor() {

this.num01 = 10;

}

ParentConstructor.prototype.func01 = function () {

console.log('this.num01 = ' + this.num01);

};

// firstConstructor

function FirstConstructor() {

}

// 상속

FirstConstructor.prototype = new ParentConstructor();

FirstConstructor.prototype.func02 = function () {

console.log('func02 는 ConstructorA 에서만 정의한 메소드');

};

// SecondConstructor

function SecondConstructor() {

}

// 상속

SecondConstructor.prototype = new ParentConstructor();

SecondConstructor.prototype.func02 = function () {

console.log('이 기능은 MyClassB 에 있는 고유 기능입니다');

};

// 인스턴스 생성

var firstConst = new FirstConstructor();

// 프로퍼티와 메서드 접근

firstConst.func01(); // this.num01 = 10

firstConst.func02(); // func02 는 ConstructorA 에서만 정의한 메소드

// 인스턴스 생성

var SecondConst = new SecondConstructor();

// 프로퍼티와 메서드 접근

SecondConst.func01(); // this.num01 = 10

SecondConst.func02(); // func02 는 ConstructorB 에서만 정의한 메소드

중복 코드가 있는 곳을 리팩토링하여 ParentConstructor 와 ParentConstructor.prototype.func01 에 재정의하였고 new ParentConstructor() 객체를 생성하면서 각각의 FirstConstructor, SecondConstructor 생성자 함수의 프로토타입 객체에 할당하였습니다.

위와 같은 패턴으로 중복 발생하는 코드를 제거하면서 상속할 수 있습니다.

### Related links

1. [객체 확장 알아보기](http://e-rooms.tistory.com/entry/%EA%B0%9D%EC%B2%B4-%ED%99%95%EC%9E%A5)

출처: <http://webclub.tistory.com/402> [Web Club]

## [Function 상속과 call,apply](http://webclub.tistory.com/394)

10개월 전 by 하기실음 관두등가 재야재희

Function prototype member

Object 처럼 Function도 function 또는 new Function 으로 생성된 모든 함수 객체가 상속할 프로토타입 멤버를 정의하고 있습니다.

Object의 프로토타입 멤버를 모든 객체에서 상속하듯이 Function의 프로토타입 멤버는 모든 함수에서 상속합니다.

Function 생성자도 다른 함수처럼 프로토타입 객체를 갖는데, 이 프로토타입 객체의 실체는 Object 타입의 인스턴스입니다.

따라서 Function 인스턴스는 모두 Object의 프로토타입 멤버를 상속하게 됩니다.



Function 프로토타입 멤버

Function의 프로토타입 객체에서 정의한 멤버를 정리하면 다음과 같습니다.

Function 프로토타입의 속성 멤버

**속성 : prototype**

* + 모든 함수가 메모리에 정의될 때 자동으로 갖게 되는 속성.
  + 함수에 대한 프로토타입 객체에 참조값으로 초기화된다.

Function 프로토타입의 메서드 멤버

**메서드 : func.call(thisObj, arg#1, arg#2, ... arg#n)**

내부에서 this를 사용하는 함수가 있다고 가정해 봅니다.

자바스크립트에서는 함수 내부에서 사용하는 this를 외부에서 전달한 객체로 할당할 수 있는 방법이 있습니다.

call 또는 apply 함수가 그 방법입니다.

위 코드에 있는 함수 func 가 내부 코드에서 this를 사용하고 있는 함수라고 했을 경우 func를 단순히 func() 처럼 호출하는 것이 아니라 func.call() 처럼 호출하면 func 내부에서 사용하는 this 의 값을 변경할 수 있다는 것입니다.

위 코드에서 call()의 첫 번째 인자로 thisObj 객체를 전달하고 있는데 이 객체가 함수의 내부에서 사용되고 있는 this에 할당됩니다.

만약 thisObj가 null 이라면 현재 프로그램 실행 환경의 루트 객체(Window라면 Window 객체에)가 this에 할당될 것입니다.

첫 번째 인자 뒤에 오는 arg#1 ~ arg#n 은 func을 호출할 때 사용하는 인자입니다.

call 의 반환값은 func 가 반환하는 값입니다.

내부에서 this를 사용하는 함수는 대부분 생성자이거나 객체의 메서드일 것입니다.

먼저 생성자를 call을 이용해서 호출하는 다음과 같은 경우를 생각해 보자.

javascript

function Constructor01() {

Constructor02.call(this);

this.method01 = function () {

// code here ...

}

}

function Constructor02() {

this.method02 = function () {

// code here ...

}

}

Constructor01 생성자 내부에서 call을 이용해서 다른 생성자 Constructor02를 호출하고 있습니다.

그리고 call을 통해서 전달되는 this 는 현재 생성자 Constructor01로 생성되는 객체입니다.

call을 이용해서 호출하는 Constructor02 내부에서의 this는 결국 Constructor01 객체에 대한 참조가 됩니다.

결국 method02도 Constructor01 객체의 메서드로 정의됩니다.

즉, Constructor02가 부모 객체의 생성자이고 Constructor01을 자식 객체의 생성자로 간주하면 결과적으로 Constructor01이 Constructor02를 상속하는 것처럼 보이게 됩니다.

call을 사용하면 다른 생성자에 정의된 멤버를 가져올 수 있다.

그리고 call을 이용하면 다른 곳에 정의되어 있는 메서드를 호출하면 인자로 전달되는 객체의 메서드처럼 사용할 수도 있습니다.

다음은 다른 객체에 정의되어 있는 메서드를 호출하는 예제입니다.

보통 객체 타입을 판별하기 위해서 현재 주어진 객체에 대해서 Object가 정의하고 있는 toString 메서드를 호출할 필요가 있습니다.

javascript

Object.prototype.toString.call(obj);

// "[object 생성자명]" 을 반환

이 결과값은 **"[object 생성자명]"** 같은 포맷의 문자열을 반환합니다.

위 코드의 obj가 Object로 생성된 객체라면 결과는 "[object Object]" 가 될 것입니다.

obj.toString() 을 직접 사용하지 않은 이유는 다른 타입의 생성자는 toString을 각자 원하는 대로 오버라이딩해서 사용하고 있기 때문에 결과값이 우리가 원하는 포맷으로 반환되지 않기 때문입니다.

이런 이유로 Object에 정의돼 있는 원래의 toString 로직을 현재 객체에 적용해야 하는데 이때 위와 같이 call을 사용할 수 있습니다.

call을 사용하면 오버로딩 이전의 기본 메서드를 호출할 수 있다.

call/apply 가 생성자에서 사용될 경우

모든 함수는 Function을 상속받아 프로토타입 멤버로 정의돼 있는 call/apply를 다음과 같이 호출할 수 있습니다.

javascript

func.call(객체, 인자값);

func.apply(객체, 인자값 배열);

call/apply 내부에서는 전달받은 인자를 이용해 다시 함수 func를 호출해 줍니다.

func 함수 내부에서 this를 사용하고 있다면 call/apply의 첫 번째 인자는 func의 내부에 있는 this에 할당되고 두 번째 인자부터는 func을 호출하는 인자로 사용합니다.

호출하는 모양은 이상하지만 결국 함수 func가 호출됩니다.

call/apply가 생성자에 사용되면 다른 생성자에 정의된 인스턴스 멤버를 가져와서 정의할 수 있습니다.

다음과 같은 Person, Korea 생성자가 있습니다.

javascript

function Person(name) {

this.name = name;

}

function Korean(city) {

this.city = city;

}

Korean에서 Person에 정의돼 있는 인스턴스 멤버를 가져오고 싶은 경우에 call/apply를 사용할 수 있다는 것입니다.

javascript

function Korean(name,city) {

// 생성자 Person에 대해 call/apply를 호출한다.

Person.call(this, name);

this.city = city;

}

Korean 생성자 내부에서 Person.call(this, name) 을 호출하면 이때의 this는 현재 생성된 Korean 인스턴스가 됩니다.

결국 Person 생성자 내부에서 this를 통해 멤버를 정의하면 결국 그 멤버는 현재 생성된 Korean 인스턴스의 멤버가 되는 것입니다.

Korean의 생성자가 앞에서처럼 정의된 상태에서 Korean 인스턴스를 생성하는 것을 상상해 봅니다.

new Korean을 실행하면 Korean 인스턴스를 생성하고 나서 Korean 생성자를 호출하는 과정을 거칩니다.

이때 생성된 인스턴스를 Korean의 this에 할당합니다.

Korean 생성자 내부에서는 다시 Person.call/apply를 호출하면 생성된 Korean 인스턴스를 Person 생성자의 this로 전달합니다.

결국 Person 생성자 내부의 this.name = name 은 생성된 Korean 인스턴스에 name 속성멤버를 추가하는 결과를 가져옵니다.

이렇게 추가된 name은 실질적인 Korean 인스턴스 멤버입니다.

따라서 다음의 코드는 true를 반환합니다.

javascript

function Person(name) {

this.name = name;

}

function Korean(name,city) {

// 생성자 Person에 대해 call/apply를 호출한다.

Person.call(this, name);

this.city = city;

}

var mySon = new Korean();

console.log(mySon.hasOwnProperty('name')); // true 반환

hasOwnProperty 함수는 인자로 전달한 속성이 인스턴스 자신의 멤버이면 true를 그렇지 않고 상속을 통해 물려받은 멤버라면 false를 반환합니다.

앞의 결과는 call/apply를 통해 Korean에 추가한 name 속성은 정말로 Korean 인스턴스의 속성임을 말해줍니다.

다시 말해, Person 생성자에 정의된 name 속성은 전적으로 자식 Korean 인스턴스의 멤버로 정의된 것입니다.

그럼 아래의 Person 인스턴스를 생성하는 두 표현을 비교해 봅니다.

javascript

function Person(name) {

this.name = name;

}

// new 와 생성자 Person을 이용해 인스턴스를 생성

var mySon1 = new Person('jaehee');

// Object 인스턴스와 call/apply를 이용해 인스턴스 멤버를 구성

var mySon2 = {}; // Object 인스턴스 생성

Person.call(mySon2, 'jaehee');

두 표현 모두 최종적으로 name이라는 속성을 지닌 객체를 생성합니다.

그러나 new를 이용해 생성한 인스턴스인 mySon1.constructor 값은 Person 입니다.

그리고 Person.prototype에 멤버를 추가하면 mySon1을 통해 접근할 수 있습니다.

그러나 mySon2.constructor 는 Object를 반환합니다. 그리고 Person.prototype 에 멤버를 추가하더라도 mySon2를 통해 접근할 수 없습니다.

mySon2의 프로토타입 체인은 Object 프로토타입 객체가 연결돼 있기 때문입니다.

출처: <http://webclub.tistory.com/394> [Web Club]

[**자바스크립트 상속 사용 예제**](http://webclub.tistory.com/450)

10개월 전 by 하기실음 관두등가 재야재희

javaScript 상속

자바스크립트의 상속은 기존의 객체나 생성자 함수를 기반으로 새로운 객체나 생성자 함수를 생성하는 것을 말합니다.

기존의 객체를 바탕으로 만들어지므로 상속을 통해 새로 만들어지는 객체에는 기존 객체의 특성을 모두 가지고 있습니다.

이를 기존의 객체에서 유산(속성과 메서드)을 물려받는 것과 비슷하다고 하여 상속이라는 말을 사용합니다.

다시 말해, 상속을 사용하면 이전에 만들었던 객체와 비슷한 객체를 쉽게 생성할 수 있습니다.



상속 활용 예제 코드

javascript

function Rectangle(w , h) {

var width = w;

var height = h;

this.getWidth = function () {

return width;

};

this.getHeight = function () {

return height;

};

this.setWidth = function (value) {

if (value < 0) {

throw '길이는 음수일 수 없다';

} else {

width = value;

}

};

this.setHeight = function (value) {

if (value < 0) {

throw '길이는 음수일 수 없다';

} else {

height = value;

}

}

}

Rectangle.prototype.getArea = function () {

return this.getWidth() \* this.getHeight();

};

function Square(length) {

this.base = Rectangle;

this.base(length, length);

}

Square.prototype = Rectangle.prototype;

var r = new Rectangle(5, 7);

var s = new Square(6);

console.log('Rectangle(5, 7) : ' + r.getArea() + ', \nnew Square(6) : ' + s.getArea());

/\* 콘솔창 결과 \*/

// Rectangle(5, 7) : 35,

// new Square(6) : 36

call() 메서드를 사용한 상속

javascript

function Rectangle(w , h) {

var width = w;

var height = h;

this.getWidth = function () {

return width;

};

this.getHeight = function () {

return height;

};

this.setWidth = function (value) {

if (value < 0) {

throw '길이는 음수일 수 없다';

} else {

width = value;

}

};

this.setHeight = function (value) {

if (value < 0) {

throw '길이는 음수일 수 없다';

} else {

height = value;

}

}

}

// call() 메서드를 사용한 상속

Rectangle.prototype.getArea = function () {

return this.getWidth() \* this.getHeight();

};

function Square(length) {

Rectangle.call(this, length, length);

}

Square.prototype = new Rectangle();

Square.prototype.constructor = Square;

var r = new Rectangle(2, 5);

var s = new Square(7);

console.log('Rectangle(2, 5) : ' + r.getArea() + ', \nnew Square(7) : ' + s.getArea());

/\* 콘솔창 결과 \*/

// Rectangle(2, 5) : 10,

// new Square(7) : 49

Jaehee's WebClub

**[발췌] 모던웹을 위한 javaScript, jQuery (윤인성 지음)**

출처: <http://webclub.tistory.com/450> [Web Club]

## [객체 확장](http://webclub.tistory.com/400)

10개월 전 by 하기실음 관두등가 재야재희

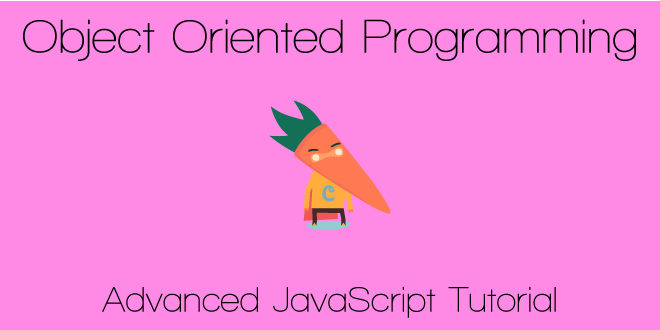
멤버 확장

확장이라 함은 부모 객체의 멤버를 모두 자식 객체로 전달하고 자식만의 멤버를 추가하는 것을 말합니다.

일반 객체지향 언어에서는 확장의 방법으로 대개 상속을 이용하는데 이 글에서 소개하는 방법은 실용적이면서도 간단한 자바스크립트만의 확장 방법에 대해 알아봅니다.

다음과 같은 순서로 상속을 구현하는 방법에 대해 소개합니다.

* 프로토타입 멤버 상속
* 인스턴스 멤버 상속
* 멤버 확장



프로토타입 멤버 상속 구현 - prototype

여기서 소개하는 프로토타입 멤버 상속은 자바스크립트에서 흔히 이용되는 방법입니다.

Person을 상속하는 Korean을 정의해 보도록 합니다.

다음은 먼저 Person 생성자의 코드입니다.

javascript

function Person() {

// 인스턴스 멤버 정의는 없음

}

// Person의 프로토타입 멤버를 정의

Person.prototype.species = 'human';

다음은 Korean 생성자를 정의해 봅니다.

Korean에서 nationality 프로토타입 멤버를 추가적으로 정의하도록 합니다.

javascript

// Korean 정의

function Korean() {

// 인스턴스 멤버 정의는 없음

}

// Korean의 프로토타입 멤버를 정의

Korean.prototype.nationality = 'korea';

이제 Korean이 Person의 프로토타입 멤버를 상속하도록 코드를 작성해 봅니다.

javascript

Korean.prototye = new Person();

Korean.prototype.constructor = Korean;

프로토타입 멤버의 상속은 Person 객체를 정의할 때 내부적으로 다음과 같은 코드가 실행되어 Object 프로토타입 객체를 상속받습니다.

javascript

Person.prototype = new Object();

Person.prototype.constructor = Person

위의 코드는 Person의 프로토타입 객체로 Object 인스턴스를 사용하고 Person 생성자를 가리키는 constructor를 방금 추가한 프로토타입 객체의 멤버로 추가하는 작업입니다.

Object를 상속하는 Person을 정의할 때는 앞의 두 작업을 자바스크립트가 자동으로 알아서 처리해 줍니다.

그러나 여기서 하려는 것은 Object 가 아닌 Person을 상속받아 다른 사용자 정의 자식 타입을 정의하려고 하는 것입니다.

이 경우에는 자바스크립트가 자동으로 해줬던 내부 작업을 사용자가 직접 구현해야 합니다.

javascript

function Person() {

// 인스턴스 멤버 정의는 없음

}

// Person의 프로토타입 멤버를 정의

Person.prototype.species = 'human';

// Korean 정의

function Korean() {

// 인스턴스 멤버 정의는 없음

}

// Korean의 프로토타입 멤버를 정의

Korean.prototype.nationality = 'korea';

Korean.prototype = new Person();

Korean.prototype.constructor = Korean;

위 코드에서는 Korean 프로토타입 객체로 Person의 인스턴스를 사용하고 있고 Korean의 프로토타입 객체의 constructor 속성값을 Korean 생성자에 대한 참조로 변경했습니다.

이 코드는 결구 Korean 프로토타입 객체를 Person 인스턴스로 대체하는 작업을 하는 것으로서 하단의 두줄 코드가 실행되고 나면 Korean의 프로토타입 체인에 Person의 프로토타입 객체가 추가되는 결과가 나타나게 될 것입니다.

Person 프로토타입 멤버의 상속이란 결국 프로토타입 체인상에서 Korean의 프로토타입 객체의 상위에 Person의 프로토타입 객체를 만들어 끼워넣는 작업이다.

이제 Korean 인스턴스를 생성해서 사용하면 Korean에서 정의하지 않은 species를 사용할 수 있습니다.

다음은 프로토타입 멤버의 상속이 완성된 Korean의 인스턴스를 생성해서 사용하는 코드입니다.

javascript

function Person() {

// 인스턴스 멤버 정의는 없음

}

// Person의 프로토타입 멤버를 정의

Person.prototype.species = 'human';

// Korean 정의

function Korean() {

// 인스턴스 멤버 정의는 없음

}

Korean.prototype = new Person();

Korean.prototype.constructor = Korean;

// Korean의 프로토타입 멤버를 정의

Korean.prototype.nationality = 'korea';

var obj = new Korean();

var species = obj.species;

console.log(species); // human 반환

var nationality = obj.nationality;

console.log(nationality); // korea 반환

다음은 Korean 인스턴스에 대해 instanceof 연산의 결과를 보여주고 있습니다.

javascript

var p = new Korean();

console.log(p instanceof Korean); // true 반환

console.log(p instanceof Person); // true 반환

지금까지 본 프로토타입 멤버를 상속하는 방법을 다른 문서에서는 "프로토타입 상속"이라는 말로 표현하기도 합니다.

인스턴스 멤버의 상속 구현 - call/apply

자바스크립트 개발을 하다 보면 프로토타입 멤버뿐 아니라 부모 객체에 있는 인스턴스 멤버도 사용하고 싶을 때가 있습니다.

이제 부모에 정의된 인스턴스 멤버를 자식의 인스턴스에서도 사용할 수 있게 만드는 방법을 알아봅니다.

어떻게 부모 객체에 정의된 인스턴스 멤버를 자식 객체에서 사용하게 할 수 있을까?

일반 객체지향 언어에서는 base()나 super() 같은 메서드를 제공해 주고 있습니다. 그래서 자식 객체를 생성할 때 이것을 호출하게 해서 부모 객체도 생성하게 만들 수 있습니다.

일반 객체지향 언어에서는 부모 객체와 자식 객체가 별도로 생성되어서 자동으로 상속되는 구조입니다.

그러나 자바스크립트에서는 base()나 super() 같은 함수나 메서드를 제공하지 않습니다.

그럼 어떻게 인스턴스 상속을 구현할까?

지금 설명하려는 상황을 정리해 봅니다.

다음과 같이 Person에 인스턴스 멤버인 name이 정의되어 있다고 가정해 봅니다.

그리고 Korean에는 다음과 같이 city라는 인스턴스 속성이 정의해 봅니다.

javascript

function Person(name) {

this.name = name; // 인스턴스 멤버 정의

}

function Korean(name, city) {

this.city = city; // 인스턴스 멤버 정의

}

인스턴스 멤버 상속

위와 같은 상황에서 Korean으로 생성된 인스턴스에서 Person에 정의된 인스턴스 멤버인 name을 사용하고 싶다고 가정해 봅니다.

Korean 생성자에서 Person을 직접 호출하면 될까?

javascript

function Korean(name, city) {

Person(name);

// Korean의 인스턴스 멤버 정의

this.city = city;

}

이렇게 하고 Korean의 인스턴스를 생성하고 name에 접근해 봅니다.

javascript

function Person(name) {

// Person의 인스턴스 멤버 정의

this.name = name;

}

function Korean(name, city) {

Person(name);

// Korean의 인스턴스 멤버 정의

this.city = city;

}

var mySon = new Korean('재희', '서울');

console.log(mySon.name); // undefined 를 반환

보다시피 name이 Korean의 인스턴스에 정의되지 않았다고 undefined가 출력됩니다.

앞에서처럼 직접 Korean 생성자에서 Person을 호출할 때 Person 멤버를 소유하는 객체를 지정하지 않았습니다.

따라서 Person을 루트 객체의 멤버로 간주하고 Person 내부에서 사용되는 this는 루트 객체를 가리키게 됩니다.

브라우저에서는 Window 객체를 가리킨다.

따라서 위와 같은 경우는 전역 변수 스코프의 루트 객체에 name 속성을 추가하는 셈입니다.

이러한 경우 Function의 call 또는 apply를 사용해 Person 내부에서 사용되는 this를 원하는 객체를 바라보게 할 수 있습니다.

다른 말로 하면 Person의 실행 컨텍스트를 Korean의 실행 컨텍스트로 설정할 수 있다는 것입니다.

Function의 call,apply를 사용하면 함수를 다른 함수의 컨텍스트에서 실행할 수 있다.

수정된 Korean 생성자 코드는 다음과 같습니다.

javascript

function Person(name) {

// Person의 인스턴스 멤버 정의

this.name = name;

}

function Korean(name, city) {

// 부모 생성자를 호출한다.

// 이때 부모 생성자 내의 this에는 call 또는 apply 함수의 첫 번째 인자를 할당한다.

// 다음 코드는 Korean 타입의 인스턴스 this를 Person 생성자의 this에 할당한다.

// 결국, 부모 생성자 Person에서는 인자로 전달 받은 Korean 인스턴스에 name 속성을 추가하는 셈이다.

Person.apply(this, [name]); // 또는 Person.call(this, name);

// Korean의 인스턴스 멤버 정의

this.city = city;

}

var mySon = new Korean('재희', '서울');

console.log(mySon.name); // '재희' 를 반환

new Korean을 실행하면 Korean 인스턴스를 생성하고 나서 Korean 생성자를 호출합니다.

이때 생성된 인스턴스를 Korean의 this에 할당합니다.

Korean 생성자 내부에서 Person.call/apply를 호출하면 방금 생성된 Korean 인스턴스를 Person 생성자의 this로 전달합니다.

결국 Person 생성자 내부의 this.name = name 은 생성된 Korean 인스턴스에 name 속성 멤버를 추가하는 결과를 가져오게 되는 것입니다.

상속 구현 통합

이제 프로토타입 상속과 인스턴스 멤버 상속를  함께 구현하는 경우를 생각해 보도록 하겠습니다.

앞에서 본 프로토타입 상속을 구현하는 코드를 다시 살펴봅니다.

javascript

Korean.prototype = new Person();

Korean.prototype.constructor = Korean;

첫 번째 코드에 의해 Korean의 프로토타입 객체는 Person 인스턴스 객체로 대체됩니다.

그런데 프로토타입 객체에 있는 name은 인스턴스별로 존재하는 것이기에 프로토타입 멤버로는 대부분 필요없을 것입니다.

따라서 다음과 같이 프로토타입 멤버에서 name 속성을 제거하면 상속이 완성됩니다.

javascript

delete Korean.prototype.name;

프로토타입 멤버에서 name 속성을 반드시 제거해야만 하는 것은 아닙니다.

인스턴스 멤버가 프로토타입 멤버로 전활될 때 일반적으로 불필요하기 때문에 제거하는 것이 일반적이지만 그대로 두어도 상관은 없습니다.

이런 작업은 선택적입니다.

인스턴스 멤버 상속과 프로토타입 멤버 상속을 구현하는 코드를 합쳐보도록 합니다.

Person과 Korean에는 프로토타입 멤버인 species, nationality가 포함돼 있습니다.

javascript

// Person 생성자

function Person(name) {

this.name = name;

}

// Person의 프로토타입 멤버 정의

Person.prototype.species = 'human';

// Korean 생성자

function Korean(name, city) {

// 인스턴스 멤버 상속

Person.call(this, name); // 또는 Person.apply(this, [name]);

// Korean의 인스턴스 멤버

this.city = city;

}

Korean.prototype.nationality = 'korean';

// 프로토타입 멤버 상속

Korean.prototype = new Person();

Korean.prototype.constructor = Korean;

// 다음 코드는 상속 구현에서 선택적이다

delete Korean.prototype.name;

이제 Korean 타입의 인스턴스를 mySon을 생성하고 모든 멤버를 출력해 보도록 합니다.

javascript

var mySon = new Korean('재희님', 'seoul');

// 멤버 출력

// 속성을 담을 배열 선언

var arr = [];

for(var propertyName in mySon) {

arr.push(propertyName);

}

// 배열 출력

console.log(arr); // ["name", "city", "constructor", "species"]

멤버 확장

앞에서는 생성자의 prototype 속성을 이용해 프로토타입 멤버를 상속하는 것과 Function의 call/apply 함수를 이용해 인스턴스 멤버를 상속하는 방법에 대해 알아보았습니다.

그러나 자바스크립트에서는 상속을 흉내낼 수 있는 다른 간단한 방법도 있습니다.

상속이란 결국 부모 객체의 멤버를 모두 자식 객체에서 사용할 수 있게 하는 것입니다.

따라서 부모 객체의 모든 멤버를 자식 객체로 복사하는 것도 좀 단순해 보이긴 하지만 자바스크립트에서 간단하게 상속을 구현하는 방법이라고 할 수 있습니다.

부모 객체를 자식 객체로 복사하는 작업에는 for/in 을 이용할 수 있습니다.

다음과 같은 유틸리티 메서드를 Object의 프로토타입 멤버로 만들어두면 편하게 멤버 복사를 할 수 있습니다.

javascript

Object.prototype.extend = function (parent) {

for(var property in parent) {

this[property] = parent[property];

}

};

for/in을 이용하면 사용자가 정의한 모든 프로토타입 멤버와 인스턴스 멤버(속성,메서드)에 접근할 수 있습니다.

위에서 정의한 메서드는 다음과 같이 사용할 수 있습니다.

javascript

Object.prototype.extend = function (parent) {

for(var property in parent) {

this[property] = parent[property];

}

};

function Person(name) {

this.name = name;

}

Person.prototype.setNewName = function (newName) {

this.name = newName;

};

// 부모 객체 생성

var parent = new Person('재희야');

// 자식 객체 생성

var child = {};

// 멤버 상속

child.extend(parent)

child.extend(parent)를 실행할 때 extend 메서드 내부에서 사용되는 this는 child 객체를 가리킨다는 것을 알 수 있을 것입니다.

이런 식으로 child 객체의 멤버를 확장하면 extend에서 사용하는 for/in의 특징 때문에 parent의 모든 인스턴스 멤버와 프로토타입 멤버는 child의 인스턴스 멤버로 추가됩니다.

child에 추가된 멤버를 출력해 보면 다음과 같습니다.

javascript

var arr = [];

for (var property in child) {

if(child.hasOwnProperty(property)) { // 인스턴스 멤버인지 확인

arr.push(property);

}

}

console.log(arr); // ["name", "setNewName", "extend"] 반환

### Related links

1. [클래스 상속 기능 샘플 코드](http://e-rooms.tistory.com/entry/%ED%81%B4%EB%9E%98%EC%8A%A4-%EC%83%81%EC%86%8D-%EA%B8%B0%EB%8A%A5-%EC%83%98%ED%94%8C-%EC%BD%94%EB%93%9C)

출처: <http://webclub.tistory.com/400> [Web Club]

## [자바스크립트의 클래스 상속 #1](http://webclub.tistory.com/344)

10개월 전 by 하기실음 관두등가 재야재희

클래스 상속 소개

클래스 상속을 사용하면 기존 코드를 변경하지 않고도 기능을 추가하거나 수정할 수 있습니다.

이 글에서는 클래스 상속 개념에 대해 간략히 알아봅니다.

클래스 상속이란?

객체지향 프로그래밍에서 상속의 개념은 현실에서 사용되는 상속이란 단어의 뜻과 유사합니다.

예를 들어, 자식이 부모로부터 유산을 상속받았다면 자식은 상속받은 재산을 사용할 수 있게 됩니다.

부모의 재산이 집, 땅, 현금 1억이 있다면 그 자식은 이를 물려 받을 수 있으며 자식은 부모로부터 상속받은 재산과 직접 만든 재산(자동차, 주식등..)을 가지고 있을 수 있습니다.

이러한 사전적인 상속의 개념은 프로그래밍에서도 상속이라고도 부르고 확장이라고도 부릅니다.

프로그래밍적으로 적용해 보면 특정 클래스(부모 클래스)의 속성과 메서드를 하위(자식) 클래스가 물려받는 것을 의미합니다.

상속을 받는 자식 클래스에서는 부모의 소스코드를 복사할 필요없이 부모 클래스의 기능과 속성을 모두 사용할 수 있을 뿐 아니라 필요한 기능을 추가해 확장할 수도 있습니다.

지금까지의 설명을 자바스크립트 코드로 표현하면 다음과 같습니다.

JavaScript

// MyParent 클래스

function MyParent() {

this.property1 = 5;

}

MyParent.prototype.method1 = function () {

console.log(this.property1);

};

// MyChild 클래스

function MyChild() {

// MyChild 에는 method1() 이 존재하지 않는다.

}

// MyParent 클래스를 상속받는 문법

MyChild.prototype = new MyParent();

// 인스턴스 생성 및 호출

var child1 = new MyChild();

// 부모의 기능에 있는 method1() 메소드를 호출하도록 한다.

child1.method1();

// MyChild 에는 method1()이 없지만 MyParent를 상속받았기 때문에 코드 복사없이 재사용할 수 있다.

MyChild에는 method1()이라는 메서드를 만들지 않았지만 MyChild는 MyParent를 상속받았기 때문에 Myparent의 기능인 method1()를 사용할 수 있는 것입니다.

바로 이러한 기능이 상속(확장)입니다.

클래스 상속 기능

상속은 다음과 같이 세 가지 기능을 가지고 있습니다.

* 코드 재사용성
* 중복 코드의 제거
* 확장

코드의 재사용성

JavaScript

// 부모 클래스

function Parent() {

this.prop = 'value1';

}

Parent.prototype.method01 = function () {

console.log('Parent.method01() ' + this.prop);

};

// 자식 클래스

function Child() {

}

// 상속하기

Child.prototype = new Parent();

// 인스턴스 생성

var child01 = new Child();

// 부모의 프로퍼티와 메서드를 호출

console.log('child01.prop = ' + child01.prop);

child01.method01();

위의 코드에서 확인할 수 있는 것처럼 상속을 사용하면 코드 복사없이 Child 클래스에서 Parent 클래스 요소인 prop과 method01()을 사용할 수 있는 것을 확인할 수 있습니다.

바로 이러한 경우를 코드의 재사용이라고 합니다.

중복 코드의 제거

클래스 상속은 중복 코드의 제거 기능을 가지고 있습니다.

다음의 예제를 통해 좀더 알아봅니다. 먼저 다음과 같은 코드가 있습니다.

JavaScript

// 첫 번째 : MyClassA

function MyClassA() {

this.prop1 = 10;

}

MyClassA.prototype.method1 = function () {

console.log('this.prop1 = ' + this.prop1);

};

MyClassA.prototype.mothod2 = function () {

console.log('이 기능은 MyClassA 에 있는 기능입니다.');

};

// 두 번째 : MyClassB

function MyClassB() {

this.prop1 = 10;

}

MyClassB.prototype.method1 = function () {

console.log('this.prop1 = ' + this.prop1);

};

MyClassB.prototype.mothod2 = function () {

console.log('이 기능은 MyClassB 에 있는 기능입니다.');

};

// 인스턴스 생성

var classA = new MyClassA();

// 프로퍼티와 메서드 접근(호출)

console.log('classA.prop1 = ' + classA.prop1); // classA.prop1 = 10

classA.method1(); // this.prop1 = 10

classA.mothod2(); // 이 기능은 MyClassA 에 있는 기능입니다.

var classB = new MyClassB();

// 프로퍼티와 메서드 접근(호출)

console.log('classB.prop1 = ' + classB.prop1); // classB.prop1 = 10

classB.method1(); // this.prop1 = 10

classB.mothod2(); // 이 기능은 MyClassB 에 있는 기능입니다.

위의 코드를 살펴보면 두개의 클래스에 중복 코드가 발생하고 있는 것을 확인할 수 있습니다.

위와 같은 중복 코드는 클래스 상속을 활용하면 중복 코드가 없는 좀 더 간결하고 깔끔한 코드를 작성할 수 있습니다.

상속을 활용해 중복 코드를 제거하는 방법은 간단합니다.

**중복 코드를 담을 부모 클래스를 생성한다.**

먼저 다음과 같이 신규 클래스(Myparent)를 만든 후 MyClassA와 MyClassB 클래스의 중복 코드를 신규 클래스에 담아주도록 합니다.

JavaScript

// 중복 코드를 담을 클래스

function MyParent() {

this.prop1 = 10;

}

MyParent.prototype.method1 = function () {

console.log('this.prop1 = ' + this.prop1 );

};

**중복 코드의 제거와 상속관계 만들기**

중복 코드를 MyParent 클래스에 담았다면 MyClassA와 MyClassB 클래스에서 사용되던 중복 코드를 제거해 주도록 합니다.

그리고 다음의 코드와 같이 상속으로 연결해 주도록 합니다.

JavaScript

// 중복 코드를 담을 클래스

function MyParent() {

this.prop1 = 10;

}

MyParent.prototype.method1 = function () {

console.log('this.prop1 = ' + this.prop1 );

};

// 첫 번째 : MyClassA

function MyClassA() {

}

// 상속

MyClassA.prototype = new MyParent();

MyClassA.prototype.mothod2 = function () {

console.log('이 기능은 MyClassA 에 있는 기능입니다.');

};

// 두 번째 : MyClassB

function MyClassB() {

}

// 상속

MyClassB.prototype = new MyParent();

MyClassB.prototype.mothod2 = function () {

console.log('이 기능은 MyClassB 에 있는 기능입니다.');

};

// 인스턴스 생성

var classA = new MyClassA();

// 프로퍼티와 메서드 접근(호출)

console.log('classA.prop1 = ' + classA.prop1); // classA.prop1 = 10

classA.method1(); // this.prop1 = 10

classA.mothod2(); // 이 기능은 MyClassA 에 있는 기능입니다.

var classB = new MyClassB();

// 프로퍼티와 메서드 접근(호출)

console.log('classB.prop1 = ' + classB.prop1); // classB.prop1 = 10

classB.method1(); // this.prop1 = 10

classB.mothod2(); // 이 기능은 MyClassB 에 있는 기능입니다.

중복 코드는 부모 클래스인 MyParent에 모두 담겨 있습니다.

그리고 자식 클래스에서는 부모 클래스의 기능을 사용할 수 있게 됩니다.

이렇게 클래스 상속을 활용하면 중복 코드를 없앨 수 있을 뿐 아니라 코드를 재사용할 수 있습니다.

출처: <http://webclub.tistory.com/344> [Web Club]

[**멤버 접근 제어 구조**](http://webclub.tistory.com/393)

10개월 전 by 하기실음 관두등가 재야재희

객체지향 언어에서의 멤버에 대한 접근 제어

일반 객체지향 언어에서는 멤버에 대한 접근을 제어하기 위해 **public, private 같은 한정자**를 제공합니다.

그러나 자바스크립트에서는 멤버에 대한 접근 제어 및 정보 은닉을 위한 특별한 키워드를 언어 차원에서 제공하지 않습니다.

자바스크립트에서의 공개 속성은 **"값을 직접 가지고 있는 변수"**와 같습니다.

따라서 객체의 내부 상태를 속성을 통해 그대로 노출하는 것입니다.

이것은 **"정보 은닉"**이 되지 않는다는 의미입니다.

때로는 변수의 값을 외부에 노출시키기도 하지만 값을 외부에서 직접 접근하지 못하도록 **비공개 멤버(private member)**를 만들고 싶은 경우도 있습니다.

하지만 자바스크립트에서는 이러한 비공개 멤버를 만드는 방법을 지원하지 않으므로 개발자가 이를 직접 구현해야 합니다.

여기서는 접근 제어를 위한 특별한 패턴 2가지를 소개합니다.



비공개 멤버 구현

비공개 멤버를 구현하는 첫 번째 방법을 소개합니다.

함수의 지역 변수는 외부에서 접근할 수 없는 비공개 변수이므로 이러한 지역 변수가 객체의 상태값을 가지고 있으면 됩니다.

문제는 공개 멤버가 비공개 변수에 직접 접근할 수 없다는 것입니다.

따라서 공개 멤버는 내부 변수에 접근할 수 있는 내부 함수를 두고서 이 함수를 통해 내부 상태값에 접근해야 합니다.

이를 정리하면 이렇습니다.

객체의 상태값으로 내부의 지역 변수를 사용하고 외부의 접근 메서드로는 공개 메서드를 사용하되, 공개 메서드는 지역 변수에 접근하는 내부 함수를 정의해서 사용한다는 것입니다.

즉, 상태값과 공개된 외부 메서드 사이에 내부 함수가 존재하는 구조가 됩니다.

내부 변수(상태값) ↔ 내부 함수 ↔ 공개 메서드

다음은 공개된 속성을 이렇게 내부 변수와 그에 대응하는 전용 함수를 사용하는 방식으로 작성한 예제입니다.

javascript

function A() {

// 내부 지역 변수(상태값)

var \_localX = 7;

// 공개 접근 메서드

this.getX = function() {

return \_localX;

};

// 10보다 작은 수만 속성값으로 입력받는다.

this.setX = function (x) {

if( x < 10 ) {

\_localX = x;

}

return \_localX;

}

}

지역 변수 \_localX 의 읽기, 쓰기 전용의 공개 메서드인 getX, setX 를 정의했습니다.

getX, setX에서는 다시 지역변수에 접근하기 위해 익명함수를 내부에서 사용하고 있습니다.

이처럼 직접 속성의 상태값에 접근하지 않고 내부의 정의된 함수를 통해 접근하는 구조가 되면 내부 상태값을 반환할 때는 그 값을 필요한 대로 가공해서 건네줄 수 있고, 값을 할당할 때도 입력받은 값을 가공하거나 유효성 검사 등을 수행한 후 상태값을 변경할 수 있습니다.

위 코드에서는 내부 상태 변수에 할당될 값은 10보다 작아야 한다는 조건을 지정했습니다.

이제 다음과 같이 A의 객체를 생성한 후 공개 메서드를 통해 값에 접근할 수 있습니다.

javascript

function A() {

// 내부 지역 변수(상태값)

var \_localX = 7;

// 공개 접근 메서드

this.getX = function() {

return \_localX;

};

// 10보다 작은 수만 속성값으로 입력받는다.

this.setX = function (x) {

if( x < 10 ) {

\_localX = x;

}

return \_localX;

};

}

var obj = new A();

var x = obj.getX();

console.log(x); // \_localX의 값인 7을 출력

console.log(obj.setX(8)); // localX 에 8을 설정하여 8을 출력함

console.log(obj.setX(11)); // 8을 출력

obj.setX(11) 은 내부 조건에 부합되지 않아 11이 할당되지 않고 앞서 할당된 obj.setX(8)의 값인 8이 반환됩니다.

클로저 인스턴스

이제 비공개 멤버와 공개 멤버를 가진 객체를 구현하는 **패턴화된 기법**을 소개합니다.

이 기법을 이용하면 자바스크립트에서도 일반 객체지향 언어에서의 객체와 유사한 구조의 객체를 만들어낼 수 있습니다.

자바스크립트 클로저는 "비공개 변수를 가질 수 있는 함수 인스턴스의 생성자"입니다.

만약 클로저가 익숙치 않다면 클로저를 이해한 후 읽어보시길 바랍니다.

여기서는 비공개 멤버를 가진 객체를 생성하는 데 클로저를 사용할 것입니다.

**비공개 객체 정의는 비공개 멤버, 공개 멤버로 설계**할 수 있습니다.

다음과 같이 공개, 비공개 멤버 구조를 지닌 객체를 만들어 낼 수 있는 outer 라는 클로저를 정의해 봅니다.

javascript

function outer() {

// 비공개 멤버

var \_x = 0;

function \_private01() {

return ++\_x;

}

function \_private02() {

return (\_x += 2);

}

// 공개 멤버(반환된 객체를 통해 외부에 공개된다)

return {

public01 : \_private01,

public02 : \_private02

};

}

\_private01(), \_private02()는 내부 변수 \_x를 각각 1,2씩 증가시키는 내부 함수이고, 이 내부 함수를 각각 호출하는 함수 public01, public02 는 return 문에서 반환한 객체를 통해 외부에 공개하고 있습니다.

return 문으로 반환된 객체가 외부에 나가서 사라지지 않는 한 내부의 \_x 값도 계속 유지됩니다.

그리고 클로저인 outer를 호출할 때마다 반환되는 객체별로 내부 변수 \_x가 초기화됩니다.

다음은 앞에서 정의한 outer를 호출해 객체를 생성하고 실행하는 예제 코드입니다.

javascript

var obj1 = outer(); // 인스턴스 obj1 생성

console.log(obj1.public01()); // 1 출력

console.log(obj1.public02()); // 3 출력

var obj2 = outer(); // 인스턴스 obj2 생성 (새롭게 반환되는 인스턴스로 \_x가 초기화된다)

console.log(obj2.public01()); // 1 출력

console.log(obj2.public02()); // 3 출력

outer()를 호출해서 생성된 객체를 obj1에 할당합니다.

public01을 호출해서 값을 1 증가시키면 public02가 호출될 때 이전에 증가된 값이 유지되어 두 번째 호출의 시작값이 됩니다.

outer 함수 호출이 종료되더라도 내부 변수 스코프 객체는 그대로 유지되는 클로저의 속성을 그대로 이용하고 있습니다.

다시 한번 outer()를 호출해서 새로운 인스턴스를 obj2에 할당한 후 public01, public02를 호출해서 결과를 보면 내부 변수 \_x가 새롭게 초기화됐다는 사실로 앞의 클로저와는 다른 닫힌 환경을 가진 **새로운 인스턴스가 생성**됐음을 알 수 있습니다.

클로저를 호출하면 단순히 객체가 반환되는 것이 아닙니다. 객체와 함께 그것과 연결된 닫힌 공간이 함께 반환되는 것입니다.

그리고 그 닫힌 공간에 내부 변수가 존재합니다.

클래스와 new를 사용해 여러 개의 닫혀진 공간을 가진 인스턴스를 만들어 내듯이 클로저와 ()를 이용(함수 호출)하면 닫힌 공간을 가진 인스턴스를 여러 개 반복해서 만들어 낼 수 있습니다.

즉, 클로저는 클래스와 유사한 존재로 볼 수 있다는 것입니다.

위의 outer 함수를 이용한 클로저 인스턴스는 obj1, obj2에 독립적인 변수 공간을 가진 인스턴스를 생성해 낸 것입니다.

그래서 클로저를 통한 생성은 개념적으로는 클래스를 이용한 생성과 거의 유사합니다.

클로저를 호출할 때마다 새로운 인스턴스가 생성된다.

새로운 클로저 인스턴스는 내부 상태가 초기화된 새로운 닫힌 환경을 가진다.

출처: <http://webclub.tistory.com/393> [Web Club]

[**자바스크립트 비공개 프로퍼티와 메서드**](http://webclub.tistory.com/312)

10개월 전 by 하기실음 관두등가 재야재희

JS 비공개(private) 멤버 및 특권(privileged) 메서드

자바 등 다른 언어와는 달리 자바스크립트에는 **private, protected, public 프로퍼티와 메서드**를 나타내는 별도의 문법이 없습니다.

객체의 모든 멤버는 public, 즉 공개되어 있습니다.



자바스크립 객체의 모든 멤버는 공개되어 있다

위에 언급했듯이 다른 언어와 달리 별도의 문법이 없기 때문에 일반적으로 객체의 모든 멤버는 공개되어 있습니다.

다음의 예제를 살펴봅니다.

javascript

var myObj = {

myProp : 1,

getProp: function () {

return this.myProp;

}

};

console.log(myObj.myProp); // myProp 에 공개적으로 접근할 수 있다.

console.log(myObj.getProp()); // getProp() 역시 공개되어 있다.

객체 리터럴 방식을 사용할 때와 마찬가지로 생성자 함수를 사용해 객체를 생성할 때도 마찬가지로 모든 멤버가 공개되어 있습니다.

javascript

function Gadget() {

this.name = 'iPod';

this.stretch = function () {

return 'iPad';

};

}

var toy = new Gadget();

console.log(toy.name); // name 은 공개되어 있다.

console.log(toy.stretch()); // stretch() 메서드도 공개되어 있다.

비공개(private) 멤버

위 예제에서는 모두 외부에서 객체의 멤버에 접근할 수 있도록 공개되어 있습니다.

그렇다면 객체의 멤버에 접근하지 못하도록 공개하지 않을 수는 없을까?

비공개 멤버에 대한 별도의 문법은 없지만 클로저를 사용해서 구현할 수 있습니다.

생성자 함수 안에서 클로저를 만들면, 클로저 유효범위 안의 변수는 생성자 함수 외부에 노출되지 않지만 객체의 공개 메서드 안에서는 쓸 수 있습니다.

즉, 생성자에서 객체을 반환할 때 객체의 메서드를 정의하면, 이 메서드 안에서는 비공개 변수에 접근할 수 있는 것입니다.

다음의 예제를 봅니다.

javascript

function Gadget() {

// 비공개 멤버

var name = 'iPod';

this.getName = function () {

return name;

};

}

var toy = new Gadget();

// name 은 비공개이므로 undefined 가 출력된다.

console.log(toy.name);

// 공개 메서드에서는 name 에 접근할 수 있다.

console.log(toy.getName()); // iPod

보다시피 자바스크립트에서도 쉽게 비공개 멤버를 구현할 수가 있습니다.

비공개로 유지할 데이터를 함수로 감싸기만 하면 되는 것입니다. 이 데이터들을 함수의 지역 변수로 만들면, 함수 외부에서는 접근할 수 없습니다.

특권(privileged) 메서드

특권 메서드라는 개념은 특정한 문법과는 관련이 없습니다.

단지 비공개 멤버에 접근권한을 가진(즉 일종의 특권을 부여 받은)공개 메서드를 가리키는 이름일 뿐입니다.

앞선 예제에서 getName()은 비공개 프로퍼티인 name에 **'특별한'** 접근 권한을 가지고 있기 때문에 특권 메서드라고 할 수 있습니다.

비공개 멤버의 허점

특권 메서드에서 비공개 변수의 값을 바로 반환할 경우 이 변수가 객체나 배열이라면 값이 아닌 참조가 반환되기 때문에, 외부 코드에서 비공개 변수 값을 수정할 수가 있습니다.

다음에 나오는 Gadget 구현은 별 문제가 없어보일지 모릅니다.

javascript

function Gadget() {

// 비공개 멤버

var specs = {

screen\_width : 320,

screen\_height : 480,

color : 'white'

};

// 공개 함수

this.getSpecs = function () {

return specs;

};

}

여기서 getSpecs() 메서드가 specs 객체에 대한 참조를 반환하고 있다는 것이 문제입니다.

specs는 감춰진 비공개 멤버처럼 보이지만 Gadget 사용자에 의해 변경될 소지가 있습니다.

javascript

function Gadget() {

// 비공개 멤버

var specs = {

screen\_width : 320,

screen\_height : 480,

color : 'white'

};

// 공개 함수

this.getSpecc = function () {

return specs;

};

}

var toy = new Gadget(),

specs = toy.getSpecc();

specs.color = 'black';

specs.price = 'free';

console.dir(toy.getSpecc());

// 비공개 객체가 아래와 같이 수정된다.

/\*\*

\* -------------------

\* color: "black"

\* price: "free"

\* screen\_height: 480

\* screen\_width: 320

\* --------------------

\*/

이와 같은 예상치 않은 문제를 해결하기 위해서는 비공개로 유지해야 하는 객체나 배열에 대한 참조를 전달할 때 주의를 기울이는 수밖에 없습니다.

하나의 방법은 getSpecs()에서 아예 새로운 객체를 만들어 사용자에게 쓸모있을 만한 데이터 일부만 담아 반환하는 것입니다.

이것을 **'최소 권한의 원칙(Principle of Least Authority, POLA)'**이라고도 합니다. 필요 이상으로 권한을 주지 말아야 한다는 의미입니다.

위 예제에서는 Gadget 사용자가 Gadget이 어떤 상자에 들어맞을지를 알아보고 싶어하는 거라면 Gadget의 면적만 알려주면 됩니다.

그렇다면 모든 정보를 내주는 대신 getDimensions()라는 메서드를 만들어 width와 height만을 담은 객체를 반환하면 될 것입니다.

이렇게 구현한다면 getSpecs() 메서드는 아예 구현할 필요조차 없었을지도 모릅니다.

객체 리터럴과 비공개 멤버

지금까지는 비공개 멤버를 만드는데 생성자를 사용하는 방법들만 살펴보았습니다.

그렇다면 객체 리터럴로 객체를 생성한 경우에 어떻게 해야 할까? 이 경우에도 비공개 멤버를 구현할 수 있을까?

여태까지 보아왔듯이 비공개 데이터를 함수로 감싸기만 됩니다.

따라서 객체 리터럴에서는 **익명 즉시 실행 함수**를 추가하여 클로저를 만듭니다.

다음 예제를 살펴봅니다.

javascript

var myObj; // 이 변수에 객체를 할당할 것이다.

(function(){

// 비공개 멤버

var name = 'My name is jaehee';

// 공개될 부분을 구현한다.

// var를 사용하지 않았다는 데 주의하라.

myObj = {

// 특권(privileged) 메서드

getName : function () {

}

}

}());

myObj.getName(); // 'My name is jaehee'

다음 예제는 기본 개념은 동일하지만 약간 다르게 구현해 본 것입니다.

javascript

var myObj = (function () {

// 비공개 멤버

var name = 'My name is jaehee';

// 공개될 부분을 구현한다.

return {

getName : function () {

return name;

}

}

}());

myObj.getName(); // My name is jaehee

프로토타입과 비공개 멤버

생성자를 사용하여 비공개 멤버를 만들 경우, 생성자를 호출하여 새로운 객체를 만들 때마다 비공개 멤버가 매번 재생성된다는 단점이 있습니다.

사실 생성자 내부에서 this에 멤버를 추가하면 항상 이런 문제가 발생합니다.

이러한 중복을 없애고 메모리를 절약하려면 공통 프로퍼티와 메서드를 생성자의 prototype 프로퍼티에 추가해야 합니다.

이렇게 하면 동일한 생성자로 생성한 모든 인스턴스가 공통된 부분을 공유하게 됩니다.  감춰진 비공개 멤버들도 모든 인스턴스가 함께 쓸 수 있습니다.

이를 위해서는 두 가지 패턴, 즉 생성자 함수 내부에 비공개 멤버를 만드는 패턴과 객체 리터럴로 비공개 멤버를 만드는 패턴을 함께 사용해야 합니다.

왜냐하면 prototype 프로퍼티도 결국 객체라서, 객체 리터럴로 생성할 수 있기 때문입니다.

다음 예제를 통해 알아봅니다.

javascript

function Gadget() {

// 비공개 멤버

var name = 'iPod';

// 공개 함수

this.getName = function () {

return name;

};

}

Gadget.prototype = (function () {

// 비공개 멤버

var browser = 'Mobile Webkit';

// 공개된 프로토타입 멤버

return {

getBrowser : function () {

return browser;

}

}

}());

var toy = new Gadget();

console.log(toy.getName()); // 객체 인스턴스의 특권 메서드

console.log(toy.getBrowser()); // 프로토타입의 특권 메서드

비공개 함수를 공개 메서드로 노출시키는 방법

노출 패턴(revelation pattern)은 비공개 메서드를 구현하면서 동시에 공개 메서드로도 노출하는 것을 말합니다.

객체의 모든 기능이 객체가 수행하는 작업에 필수불가결한 것들이라서 최대한의 보호가 필요한데, 동시에 이 기능들의 유용성 때문에 공개적인 접근도 허용하고 싶은 경우가 있을 수 있습니다.

노출 패턴은 이런한 경우에 유용하게 사용할 수 있습니다. 그리고 메서드가 공개되어 있다는 것은 결국 이 메서드가 위험에 노출되어 있다는 말과도 같습니다.

공개 API 사용사자 어쩌면 본의 아니게 메서드를 수정할 수 있기 때문입니다.

ECMAScript 5에서는 객체를 고정(freeze)시킬 수 있는 선택지가 있지만, 하위 버전에서는 그렇지 않습니다.

이제 노출 패턴에 대해 알아봅니다.

이 용어는 크리스천 헤일먼(Christian Heimann)이 만들어냈으며 처음에는 '모듈 노출 패턴(revealing module pattern)'이라고 했습니다.

먼저 예제를 살펴 봅니다.

이 예제는 객체 리터럴 안에서 비공개 멤버를 만드는 패턴에 기반하고 있습니다.

javascript

var myArray;

(function (){

var arrStr = '[object Array]',

toString = Object.prototype.toString;

function isArray(a) {

return toString.call(a) === arrStr;

}

function indexOf(haystack, needle) {

var i = 0,

max = haystack.length;

for (; i < max; i += 1) {

if(haystack[i] === needle) {

return i;

}

}

return -1;

}

myArray = {

isArray : isArray,

indexOf : indexOf,

inArray : indexOf

}

}());

여기에는 비공개 변수 두 개와 비공개 함수 두 개, isArray()와 indexOf()가 존재합니다.

즉시 실행 함수의 마지막 부분을 보면, 공개적인 접근을 허용해도 괜찮겠다고 결정한 기능들이 myArray 객체에 채워집니다.

비공개 함수 indexOf()는 ECMAScript 5식의 이름인 indexOf와 PHP에서영향을 받은 이름인 inArray라는 두 개의 이름으로 노출되었습니다.

이제 새로운 myArray 객체를 테스트해 봅니다.

javascript

console.log(myArray.isArray([1, 2])); // true

console.log(myArray.isArray({0: 1})); // false

console.log(myArray.indexOf(['a', 'b', 'z'], 'z')); // 2

console.log(myArray.inArray(['a', 'b', 'z'], 'z')); // 2

이제 공개된 메서드인 indexOf()에 예기치 못한 일이 일어나더라도, 비공개 함수인 indexOf()는 안전하게 보호되기 때문에 inArray()는 계속해서 잘 동작할 것입니다.

javascript

myArray.indexOf = null;

console.log(myArray.inArray(['a', 'b', 'z'], 'z')); // 2

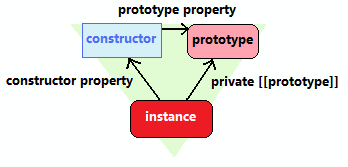
출처: <http://webclub.tistory.com/312> [Web Club]

[**prototype, constructor, 인스턴스**](http://webclub.tistory.com/392)

10개월 전 by 하기실음 관두등가 재야재희

프로토타입, 생성자, 인스턴스

이 글에서는 인스턴스와 생성자의 prototype 속성, 프로토타입 객체의 constructor 속성에 대해 알아봅니다.



생성자, 프로토타입 객체, 인스턴스 관계도

함수 모델

일반적으로 모든 함수를 정의하면 다음과 같은 모델이 구성됩니다.

javascript

function fn() {

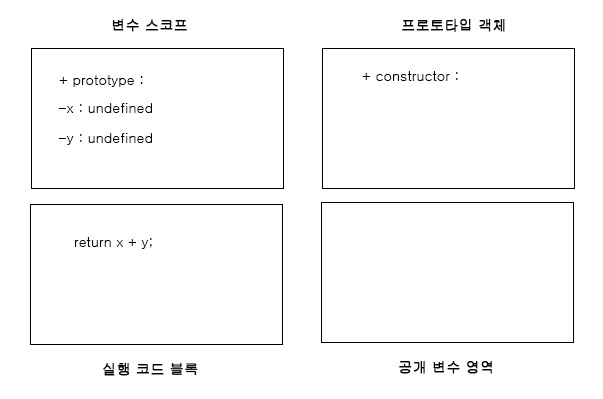
var x = 10;

var y = 20;

return x + y;

}

fn();



위의 그림에서는 함수가 정의될 때 그 함수와 연결된 프로토타입이라는 객체가 함께 정의되는 것을 보여주고 있습니다.

생성자도 함수로서 이와 동일한 구조로 생성됩니다.

생성자 모델

이제 다음과 같이 정의된 생성자 Person을 봅니다.

javascript

function Person(name) {

this.name = name;

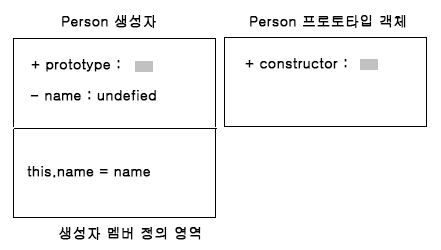
}

Person을 파싱해서 정의하면 최종적으로 생성자 객체와 프로토타입 객체가 함께 정의됩니다.

그렇다고 두 존재가 같은 객체에 포함되는 것은 아닙니다.

생성자와 프로토타입 객체는 다음 그림처럼 별도로 존재하면서 서로에 대한 참조를 가지고 연결되어 있습니다.

그리고 공개 변수 스코프는 생성자의 멤버가 정의되는 곳으로서 현재 Person에는 생성자 멤버가 없는 관계로 생략해서 그림을 다음과 같이 간소해서 표현할 수 있을 것입니다.



Person 생성자와 그것의 프로토타입 객체가 별도의 객체라는 점을 강조하기 위해 생성자와 프로토타입 객체를 분리해서 표현되었습니다.

분리된 두 객체가 생성자의 prototype 속성과 프로토타입 객체의 constructor라는 속성으로 연결되어 있습니다.

생성자 객체와 프로토타입 객체가 생성되면 각각 prototype, constructor 공개 속성이 추가되고 서로에 대한 참조가 할당됩니다.

생성자와 프로토타입 객체는 별도의 객체로서 서로의 참조값을 prototype, constructor에 가지고 있다.

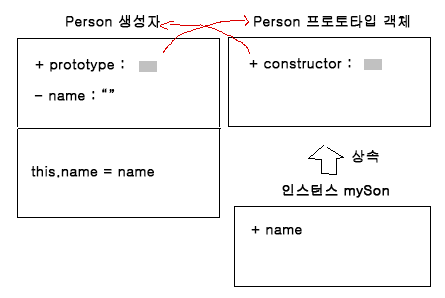
이제 new Person 으로 객체를 생성해 봅니다.

javascript

// Person 인스턴스 생성

var mySon = new Person('jaehee');

이 코드가 실행되고 나서 생성되는 인스턴스와 prototype, constructor의 관계를 그려보면 다음과 같습니다.



그림을 보면 mySon 인스턴스는 Person의 프로토타입 멤버를 상속한다고 표시되어 있습니다.

이렇게 생성자 객체, 프로토타입 객체 그리고 인스턴스는 구분돼야 합니다.

생성자, 프로토타입 객체는 함수를 정의하면 함께 정의되는 객체이고 인스턴스는 생성을 해서 얻게 되는 객체입니다.

프로토타입 객체

함수를 정의하면 함께 정의된다고 하는 프로토타입 객체에 대해 좀더 알아봅니다.

prototype 속성

위 예제 코드인 Person 생성자에는 prototype이라는 공개 속성이 있는데, 이 속성을 통해 프로토타입 객체에 접근할 수 있습니다.

javascript

Person.prototype

생성자의 프로토타입 객체에 접근해 멤버를 추가하거나 삭제하려면 반드시 생성자의 prototype 속성을 이용해야 합니다.

프로토타입 객체는 생성자별로 하나만 정의되므로 프로토타입 객체에 접근하는 데 생성자를 이용하고 있습니다.

하지만 인스턴스에서 직접 프로토타입 객체에 접근하는 방법은 없습니다.

따라서 다음과 같은 코드는 올바르지 않습니다.

javascript

mySon.prototype // 잘못된 코드

중요한 것은 직접 프로토타입 객체에 접근은 못해도 인스턴스를 통해서 생성자별로 정의되는 프로토타입 **멤버**에는 접근할 수 있습니다.

프로토타입 객체의 특징

프로토타입 객체도 자바스크립트 객체입니다.

즉, 프로토타입 객체도 (키, 값) 쌍으로 된 멤버를 추가할 수 있는 연관 배열 구조라는 것입니다.

따라서 프로토타입 객체에도 런타임에 멤버를 동적으로 추가할 수 있습니다.

참고로 프로토타입 객체에 포함된 멤버를 프로토타입 멤버라고 하며, 프로토타입 객체가 생성되면 constructor 라는 속성이 기본적으로 추가됩니다.

다음은 prototype 을 통해 Person의 프로토타입 메서드 멤버를 추가하는 코드입니다.

javascript

Person.prototype.getName = function () {

return this.name;

}

이렇게 Person.prototype 을 통해 추가하면 그 멤버는 Person의 모든 인스턴스에서 사용할 수 있게 된다는 점입니다.

다시 말해, 모든 인스턴스는 해당 생성자가 가지고 있는 프로토타입 객체의 멤버를 상속받는다라고 할 수 있습니다.

이제 Person 인스턴스는 모두 getName에 접근할 수 있습니다.

Person 생성자의 코드를 정리하면 다음과 같습니다.

javascript

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

}

var mySon = new Person('jaehee');

var yourSon = new Person('jeongin');

console.log(mySon.getName()); // jaehee 반환

console.log(yourSon.getName()); // jeongin 반환

위와 같이 프로토타입 멤버에서 this를 사용한다면 이 this는 현재 인스턴스를 나타닙니다.

프로토타입 멤버에서의 this 는 현재 인스턴스를 나타낸다.

위 코드에서 보다시피 getName에 접근하는 데Person.prototype.getName 처럼 생성자의 prototype을 통해 접근할 수 있을 뿐더러 mySon.getName 처럼 인스턴스를 통해서도 접근할 수 있습니다.

그러나 생성자의 prototype을 통해 접근하느냐 인스턴스를 통해 접근하느냐에 따라 다른 행동을 할 수 있습니다.

이와 관련된 설명은 뒤에서 알아보도로 하겠습니다.

프로토타입 멤버와 인스턴스 멤버 비교

각 인스턴스도 연관 배열 구조이므로 멤버를 동적으로 추가할 수 있습니다.

앞에서 생성된 인스턴스인 mySon, yourSon 에 멤버를 추가해 봅니다.

javascript

mySon.nickName = '재희 별명';

yourSon.nickName = '정진 별명';

위처럼 인스턴스에 새롭게 추가된 nickName 멤버는 인스턴스별로 관리됩니다.

자바스크립트에서는 객체의 멤버를 언제든지 조작할 수 있게 공개해 놓고 있습니다.

그래서 프로토타입 객체의 멤버도 언제든지 멤버를 추가, 제거, 대체할 수 있습니다.

프로토타입 객체는 함수를 메모리에 정의할 때 한 번만 생성됩니다.

프로토타입 멤버는 인스턴스별로 복사복이 존재하는 것이 아니라 해당 생성자에 하나만 존재하면서 그 생성자의 모든 인스턴스가 함께 공유하게 되는 것입니다.

프로토타입 멤버 편집의 비대칭

여기서 설명하는 프로토타입 멤버의 특징을 이해하지 못하면 자바스크립트 코드를 작성하다가 자칫 디버깅하는 데 많은 시간을 허비할 수 있습니다.

구조상으로는 분명한데 우리가 의도한 대로 되지 않을 수 있는 부분이 있기 때문입니다.

먼저 다음 코드를 예로 들어보겠습니다.

javascript

// 생성자 정의

function Person() {

}

// 인스턴스 생성

var mySon = new Person();

var yourSon = new Person();

// prototype을 통해 age 값 쓰기

Person.prototype.age = 6;

위에서 생성자 Person이 정의됐고 Person 인스턴스인 mySon, yourSon을 생성했습니다.

그리고 나서 Person의 프로토타입 객체에 age 라는 속성을 추가했습니다.

**프로토타입 멤버 읽기**

프로토타입 멤버를 다음과 같이 읽는 경우 그 결과는 다음과 같이 명백합니다.

javascript

// age 값 읽기

console.log(Person.prototype.age); // 6 출력

console.log(mySon.age); // 6 출력

console.log(yourSon.age); // 6 출력

mySon, yourSon에는 age가 없습니다.

따라서 mySon에서 age 속성 검색을 마치고 나면 프로토타입 객체로 이동해 age를 검색하고 그렇게 검색한 결과를 출력하기 때문에 모두 동일한 값인 6이 출력됩니다.

Person 프로토타입 객체에 정의된 멤버(이 경우 age)의 값을 "읽을"때는 다음의 두 표현을 모두 사용할 수 있습니다.

javascript

var age = Person.prototype.age;

var age = mySon.age;

**프로토타입 멤버 추가하기**

이제 다음과 같이 인스턴스를 통해 age에 값을 쓰는 코드를 보도록 합니다.

javascript

mySon.age = 7; // mySon의 age에 7를 쓴다.

yourSon.age = 8; // yourSon의 age에 8을 쓴다.

age는 인스턴스 멤버가 아니므로 mySon에서 age를 찾지 못합니다.

그러면 age를 mySon 인스턴스에 속성으로 추가할까, 아니면 프로토타입 객체의 멤버를 검색할까?

이를 알아보고자 앞에서 할당한 값을 다음과 같이 출력해 봅니다.

javascript

// 생성자 정의

function Person() {

}

// 인스턴스 생성

var mySon = new Person();

var yourSon = new Person();

// prototype을 통해 age 값 쓰기

Person.prototype.age = 6;

mySon.age = 7; // mySon의 age에 7를 쓴다.

yourSon.age = 8; // yourSon의 age에 8을 쓴다.

// age 값 읽기

console.log(Person.prototype.age); // 6 출력

console.log(mySon.age); // 7 출력

console.log(yourSon.age);// 8 출력

출력 결과를 보면 인스턴스를 통해 속성에 값을 쓸 때는 프로토타입 객체를 검색하지 않고 인스턴스에서만 속성을 검색한다는 것을 알 수 있습니다.

즉, mySon.age에 값을 할당하면 Person.prototype.age 의 값을 업데이트하는 것이 아니라 mySon에 age 속성을 추가해서 값을 할당합니다.

이제 코드처럼 인스턴스를 통해 age의 값을 출력하면 각 인스턴스의 age 값을 출력하게 됩니다.

이러한 메커니즘으로 설계된 이유는 하나의 생성자를 이용해 여러 인스턴스가 생성될 텐데, 인스턴스 하나에서 값을 수정하더라도 다른 객체에 영향을 미치는 것을 막기 위해서입니다.

인스턴스를 통해 멤버에 값을 쓰는 경우 멤버 검색 시 프로토타입 객체로까지 거슬러 올라가지 않는다.

그러나 어떤 상황에서는 모든 인스턴스의 값을 동시에 수정하고 싶은 경우가 있을 텐데, 이런 경우는 어떻게 할까?

그럴 때는 다음처럼 인스턴스를 통한 접근 대신 생성자의 prototype을 이용해 프로토타입 객체에 직접 접근해서 값을 변경하면 됩니다.

아래와 같이 prototype을 통해 속성의 값을 변경하면 이제 모든 인스턴스를 통해 age값을 읽어도 동일한 값이 출력됩니다.

javascript

function Person() {

}

// 인스턴스 생성

var mySon = new Person();

var yourSon = new Person();

// prototype을 통해 age 값 쓰기

Person.prototype.age = 10;

// age 값 읽기

console.log(Person.prototype.age); // 10 출력

console.log(mySon.age); // 10 출력

console.log(yourSon.age);// 10 출력

프로토타입 멤버에 값을 쓰고 싶다면 인스턴스가 아니라 생성자의 prototype 속성을 이용해야 한다.

생성자(constructor)

함수를 정의하면 프로토타입 객체가 정의되고 그 객체의 멤버로 constructor 속성을 가지고 있다고 설명했습니다.

constructor 속성은 프로토타입 객체를 생성했던 함수에 대한 참조를 나타냅니다.

이 프로토타입 객체의 constructor 속성은 상속을 통해 인스턴스에서 다음과 접근할 수 있습니다.

위의 Person 예제의 경우 mySon.construcctor 값은 Person을 가리키므로 다음 코드는 참이 될 것입니다.

javascript

// 생성자 정의

function Person() {

}

// 인스턴스 생성

var mySon = new Person();

var yourSon = new Person();

console.log(mySon.constructor == Person); // true 반환

앞에서 본 생성자, 프로토타입 객체, 그리고 인스턴스 관계도를 보면서 따져보면 다음과 같은 객체 비교는 모두 참이 됩니다.

javascript

console.log(mySon.constructor == Person.prototype.constructor); // true 반환

console.log(mySon.constructor == Person); // true 반환

여기서 주목할 점은 constructor 속성은 단순히 함수명을 나타내는 것이 아니라 **함수 객체에 대한 참조**라는 것입니다.

constructor 프로토타입 속성이 생성자 자체를 가리키는 참조이므로 이 값을 통해 함수를 호출할 수도 있고 따라서 다음과 같이 객체를 생성할 수 도 있습니다.

javascript

// 생성자 정의

function Person(name) {

this.name = name;

}

var mySon = new Person('jaehee');

// constructor 속성을 통해 객체를 생성

var myGrandSon = new mySon.constructor('손자');

console.log(myGrandSon.name);

mySon.constructor는 생성자 객체를 가리키는 참조값이므로 () 연산자를 이용해 직접 호출할 수 도 있고 코드처럼 new와 함께 사용되어 객체를 생성할 수도 있습니다.

프로토타입 객체의 멤버인 constructor 속성은 해당 프로토타입 객체를 생성한 생성자 객체를 가리킨다.

prototype 속성처럼 constructor에 할당된 기본값도 런타임에 언제든지 변결될 수도 있습니다.

다음과 같이 mySon의 constructor에 Object 생성자를 할당할 수도 있습니다.

javascript

mySon.constructor = Object;

생성자, 프로토타입 객체, 그리고 인스턴스 관계도를 보면 인스턴스의 constructor를 통해서도 프로토타입 객체에 접근하는 방법이 있습니다.

만약 mySon 인스턴스를 통해 프로토타입 객체에 접근하려면 다음과 같이 할 수 있습니다.

javascript

mySon.constructor.prototype; // 인스턴스를 통해 프로토타입 객체에 접근

constructor는 앞에서 본 것처럼 상속을 통해 mySon에서 사용할 수 있게 됩니다.

따라서 다음과 같은 비교는 모두 프로토타입 객체를 나타내는 표현이 됩니다.

javascript

function Person(name) {

this.name = name;

}

var mySon = new Person('jaehee');

console.log(mySon.constructor.prototype == Person.prototype); // true 반환

출처: <http://webclub.tistory.com/392> [Web Club]

## [자바스크립트 네임스페이스 패턴 - namespace pattern](http://webclub.tistory.com/311)

10개월 전 by 하기실음 관두등가 재야재희

자바스크립트 설계 패턴 -  Namespace Pattern

네임스페이스는 프로그램에서 필요로 하는 전역 변수의 개수를 줄이는 동시에 과도한 접두어를 사용하지 않고도 이름이 겹치지 않게 해줍니다.

자바스크립트의 언어 문법에 내장되어 있지는 않지만, 네임스페이스는 꽤 쉽게 구현할 수 있는 기능입니다.



네임스페이스 패턴

대개 애플리케이션이나 라이브러리를 위해  전역 유효 범위에 수많은 함수, 객체, 변수들로 어지럽히지 않도록 하기 위해 전역 객체를 하나 만들고(단 하나만 만드는 것이 이상적이다) 모든 기능을 이 객체에 추가하는 패턴을 네임스페이스 패턴이라고 합니다.

다음의 예제는 안티 패턴입니다.

JavaScript

// 수정 전 : 전역 변수 5개

// 경고 : 안티 패턴

// 생성자 함수 2개

function Parent() {}

function Child() {}

// 변수 1개

var some\_var = 1;

// 객체 2개

var module1 = {};

module.data = {

a : 1,

b : 2

};

var module2 = {}

위와 같은 코드를 리팩터링하기 위해서는 먼저 애플리케이션 전용 전역 객체, 이를 테면   MYAPP을 생성합니다.

그런다음 모든 함수와 변수들을 이 전역 객체의 프로퍼티로 변경합니다.

다음의 예제는 리팩터링한 결과입니다.

JavaScript

// 수정 후 : 전역 변수 1개

// 전역 객체

var MYAPP = {};

// 생성자

MYAPP.Parent = function() {};

MYAPP.Child = function() {};

// 변수

MYAPP.some\_var = 1;

// 객체 컨테이너

MYAPP.modules = {};

// 객체들을 컨테이너 안에 추가한다.

MYAPP.modules.module1 = {};

MYAPP.modules.module1.data = { a : 1, b : 2 };

MYAPP.modules.module2 = {};

전역 네임스페이스 객체의 이름은 애플리케이션 이름이나 라이브러리의 이름, 도메인명, 회사 이름 중에서 선택할 수도 있습니다.

흔히 코드를 읽는 사람 눈에 띄도록 전역 객체 이름은 모두 대문자로 쓰는 명명 규칙을 사용하기도 합니다.(이 규칙은 상수를 쓸 때도 사용된다는 점에 주의하라.)

이 패턴은 코드에 네임스페이스를 지정해주며, 코드 내의 이름 충돌 뿐 아니라 이 코드와 같은 페이지에 존재하는 자바스크립트 라이브러리나 위젯 등 서드 파티 코드와의 이름 충돌도 방지해 줄 수 있습니다.

다양한 작업에 응용할 수 있으며, 매우 권장하는 패턴입니다. 그러나 다음과 같은 단점도 존재합니다.

* 모든 변수와 함수에 접두어를 붙여야 하기 때문에 전체적으로 코드량이 약간 더 많아지고 따라서 다운로드해야 하는 파일 크기도 늘어난다.
* 전역 인스턴스가 단 하나뿐이기 때문에 코드의 어느 한 부분이 수정되어도 전역 인스턴스를 수정하게 된다. 즉 나머지 기능들도 갱신된 상태를 물려 받는다.
* 이름이 중첩되고 길어지므로 프로퍼티를 판별하기 위한 검색 작업도 길고 느려진다.

범용 네이스페이스 함수

프로그램의 복잡도가 증가하고 코드의 각 부분들이 별개의 파일로 분리되어 선택적으로 포함하게 되면, 어떤 코드가 특정 네이스페이스나 그 내부의 프로퍼티를 처음으로 정의한다고 가정하기가 위험합니다.

네임스페이스에 추가하려는 프로퍼티가 이미 존재할 수도 있고 따라서 내용을 덮어쓰게 될 지도 모릅니다.

그러므로 네임스페이스를 생성하거나 프로퍼티를 추가하기 전에 먼저 이미 존재하는지 여부를 확인하는 것이 최선입니다.

다음의 예제를 살펴봅니다.

JavaScript

// 위험한 패턴

var MYAPP = {};

// 개선된 패턴

if(typeof MYAPP === 'undefined') {

var MYAPP = {};

}

// 또는 더 짧게 쓸 수 있다.

var MYAPP = MYAPP || {};

이렇게 추가된 확인 작업 때문에 상당량의 중복 코드가 생겨날 수 있습니다.

예를 들어 MYAPP.modules.module2 를 정의하려면, 각 단계의 객체와 프로퍼티를 정의할 때마다 확인 작업을 거쳐야 하므로 코드가 세 번 중복될 것입니다.

따라서 네임스페이스 생성의 실제 작업을 맡아 줄 재사용 가능한 함수를 만들어두면 편리합니다.

이러한 재사용이 가능한 네임스페이스 생성 함수를 namespace()라 만들고 다음과 같이 사용할 수 있습니다.

JavaScript

// 네임스페이스 생성 함수를 사용한다.

MYAPP.namespace('MYAPP.modules.module2');

// 위 네이스페이스 함수는 다음과 같은 결과를 반환할 것이다.

var MYAPP = {

modules : {

module2 : {}

}

};

이제 위와 같은 결과를 반환할 수 있는 네임스페이스를 구현해 보도록 합니다.

다음과 같은 방식은 해당 네임스페이스가 존재하면 덮어쓰지 않기 때문에 기존의 코드를 망가뜨리지 않는다.

JavaScript

ar MYAPP = MYAPP || {};

MYAPP.namespace = function (ns\_string) {

var parts = ns\_string.split('.'),

parent = MYAPP,

i;

// 처음에 중복되는 전역 객체명은 제거한다.

if(parts[0] === MYAPP) {

parts = parts.slice(1);

}

for (i = 0; i < parts.length; i += 1) {

// 프로퍼티가 존재하지 않는다면 생성한다.

if(typeof parent[parts[i]] === 'undefined') {

parent[parts[i]] = {};

}

parent = parent[parts[i]];

}

return parent;

};

위 코드는 다음과 같이 사용할 수 있습니다.

JavaScript

// 반환 값을 지역 변수에 할당한다.

var module2 = MYAPP.namespace('MYAPP.modules.module2');

module2 === MYAPP.modules.module2; // true

**namespace**는 자바스크립트에서 향후 사용하고자, 일반적인 사용을 금지한 예약어에 포함되어 있다.

따라서 이 단어 그대로를 프로퍼티명으로 쓰는 것은 권하지 않는다.

출처: <http://webclub.tistory.com/311> [Web Club]

[**네임스페이스 패턴(namespace pattern)**](http://webclub.tistory.com/81)

10개월 전 by 하기실음 관두등가 재야재희

네임스페이스 패턴

자바스크립트 코드의 재사용과 생산성을 향상시키고자 많은 라이브러리가 나오고 있다. 이러한 라이브러리를 제작할 때 일반적으로 가장 중요한 것은 다른 모듈로부터 독립성을 보장하고 상호 간섭하지 않게끔 네임스페이스(namespace) 를 사용하는 것이다.

라이브러리를 사용하다 보면 언제든지 한쪽에서 사용한 타입명이 다른 쪽에서도 사용될 수도 있는데, 어디서 사용될 지 모르는 라이브러리를 만드는 경우라면 반드시 네임스페이스를 정의해 객체를 안전하게 분리/관리하는 것을 필수입니다.



네임스페이스는 프로그램에서 필요로 하는 전역 변수의 개수를 줄이는 동시에 과도한 접두어를 사용하지 않고도 이름이 겹치지 않게 해준다.

수많은 함수, 객체, 변수들로 전역 유효범위를 어지럽히지 않는 대신, 애플리케이션이나 라이브러리를 위한 전역 객체를 하나 만들고 (단 하나만 만드는 것이 이상적이다.) 모든 기능을 이 객체에 추가하면 됩니다.

// 안티 패턴 : 전역 변수 5개  
  
// 생성자 함수 2개  
**function** Parent(){}  
**function** Child(){}  
  
// 변수 1개  
**var** some\_var = 1;  
  
// 객체 2개  
**var** module1 = {};  
module1.data = {a: 1, b: 2};  
**var** module2 = {};

위 코드를 살펴보면 전역 변수가 5개가 있는 것을 확인할 수 있다.

이러한 코드는 전역 스코프를 어지럽히기 때문에 바람직하지 않은 안티 패턴이다.

위와 같은 코드를 리팩터링하기 위해서는 먼저 애플리케이션 전용 전역 객체, 이를 테면 MYAPP을 생성한다.

그런 다음 모든 함수와 변수들을 이 전역 객체의 프로퍼티로 변경한다.

// 리팩토링 후 : 전역변수는 1개만 존재  
  
// 전역 객체  
**var** MYAPP = {};  
  
// 생성자  
MYAPP.Parent = **function**(){};  
MYAPP.Child = **function**(){};  
  
// 변수  
MYAPP.some\_var = 1;  
  
// 객체 컨테이너  
MYAPP.module = {};  
  
// 객체들을 컨테이너 안에 추가한다.  
MYAPP.modules.module1 = {};  
MYAPP.modules.module1.data = {a: 1, b: 2};  
MYAPP.modules.module2 = {};

위 코드와 같이 전역 스코프에는 하나의 전역변수만 둔다.

전역 네임스페이스 객체의 이름은 애플리케이션 이름이나 라이브러리의 이름, 도메인명, 회사 이름중에서 선택할 수 있다.

범용 네임스페이스 함수

프로그램의 복잡도가 증가하고 코드의 각 부분들이 별개의 파일로 분리되어 선택적으로 문서에 포함되게 되면, 어떤 코드가 특정 네임스페이스나 그 내부의 프로퍼티를 처음으로 정의한다고 가정하기가 위험하다. 네임스페이스에 추가하려는 프로퍼티가 이미 존재할 수도 있고 따라서내용을 덮어쓰게 될지도 모른다.

그러므로 네임스페이스를 생성하거나 프로퍼티를 추가하기 전에 먼저 이미 존재하는지 여부를 확인하는 것이 최선이다.

// 안티 패턴  
**var** MYAPP = {};  
  
// 개선된 패턴  
**if**(**typeof** MYAPP === "undefined"){  
 **var** MYAPP = {};  
}  
  
//또는 더 짧게 쓸 수 있다.  
**var** MYAPP = MYAPP || {};

이렇게 네임스페이스와 네임스페이스의 프로퍼티가 존재여부를 확인하는 작업으로 인해 많은 양의 중복 코드가 생겨 날 수가 있디.

따라서 네임스페이스 생성의 실제 작업을 맡아 줄 재사용 가능한 함수를 만들어 두면 코드의 재사용성이 높아진다.

// 네임스페이스 함수를 사용하여 네임스페이스와 네임스페이스의 프로퍼티를 확인한다  
MYAPP.namespace('MYAPP.modules.module2');  
  
// namespace함수를 사용하면 다음과 같은 결과를 반환한다.  
**var** MYAPP = {  
 modules : {  
 module2: {}  
 }  
};

**다음은 네임스페이스 함수를 구현한 예제다.**

다음과 같은 방식은 네임스페이스가 존재하면 덮어쓰지 않기 때문에 기존 코드를 망가뜨리지 않는다.

**var** MYAPP = MYAPP || {};  
  
MYAPP.namespace = **function**(ns\_string){  
 **var** parts = ns\_string.split('.'),  
 parent = MYAPP,  
 i;  
  
 //처음에 중복되는 전역 객체명은 제거한다.  
 **if**(parts[0] === 'MYAPP'){  
 parts = parts.slice(1);  
 }  
  
 **for**(i=0; i<parts.length; i++){  
 //프로퍼티가 존재하지 않으면 생성한다.  
 **if**(**typeof** parent[parts[i]] === 'undefined'){  
 parent[parts[i]] = {};  
 }  
 parent = parent[parts[i]];  
 }  
 **return** parent;  
};

이 코드는 다음과 같이 사용할 수 있다.

// 반환 값을 지역변수에 할당한다.  
**var** module2 = MYAPP.namespace('MYAPP.modules.module2');  
module2 ==== MYAPP.modules.module2; // true  
  
// 첫부분의 'MYAPP'을 생략하고도 쓸 수 있다.  
MYAPP.namespace('modules.module51');

출처: <http://webclub.tistory.com/81> [Web Club]

[**싱글톤(singleton) 패턴**](http://webclub.tistory.com/150)

10개월 전 by 하기실음 관두등가 재야재희

싱글톤

싱글톤(singleton) 패턴은 전체 시스템에서 하나의 인스턴스만 존재하도록 보장하는 객체 생성패턴을 말합니다.

따라서 객체 리터럴도 싱글톤 패턴이라고 할 수 있습니다.

javascript

var singletonObj = {

a : '값',

b : function () {

}

}

그러나 객체 리터럴로는 비공개 상태 및 함수를 정의할 수는 없습니다. 규모가 큰 라이브러리에서는 흔히 외부에서 접근할 수 없는 비공개 멤버를 가지고 있습니다. 자바스크립트에서 비공개 멤버가 필요하다면 먼저 클로저(closure) 를 염두에 두어야 합니다.

즉, 비공개 멤버를 가진 싱글톤 객체를 생성하려면 클로저를 이용해야 합니다.



다음은 클로저를 이용한 싱글톤 패턴을 구현하는 예제입니다.

javascript

var Singleton = (function () {

/\*\*

\* --------------------------------

\* 싱글톤 패턴 구현 코드

\* --------------------------------

\*/

// 비공개 변수, 메서드 정의

var instantiaed;

function init() {

// 싱글톤 객체 정의

return {

// 공개 메서드 정의

publicMethod : function () {

return 'hello Singleton Pattern!!!';

},

// 공개 프로퍼티 정의

publicProp : 'single value'

}

}

// 공개 메서드인 getInstance() 를 정의한 객체.

// 렉시컬 특성으로 인해 비공개 변수, 메서드에 접근 가능(클로저)

return {

getInstance : function () {

if (!instantiaed) {

instantiaed = init();

}

return instantiaed;

}

}

})();

// 싱글톤 객체 생성하여 publicMethod 호출 가능해짐

var first = Singleton.getInstance();

first.publicMethod();

console.log(first.publicMethod()); // hello Singleton Pattern!!!

var second = Singleton.getInstance();

second.publicMethod();

console.log(second.publicMethod()); // hello Singleton Pattern!!!

console.log(first === second); // true

위의 예제 코드에서 먼저 비공개 함수인 init() 의 return 문에서 객체 리터럴로 정의되는 객체가 싱글톤 객체입니다.

이 객체는 프로그램 전체에서 하나만 존재하게 됩니다. 외부에 공개되는 익명함수의 return 문에서는 싱글톤 객체를 구하는 공개 메서드인 getInstance 가 포함된 객체를 반환합니다. getInstance 메서드에서는 내부 변수로 정의된 instantiaed 변수의 값을 확인해서 아직 싱글톤 객체가 생성되지 않았다고 판단하면 내부 함수인 init 를 호출하여 싱글톤 객체를 생성해서 반환하게 됩니다. 그런 다음 instantiaed 번수에 싱글톤 객체를 할당하게 됩니다.

이렇게 일반적으로 싱글톤 패턴에서는 이미 객체가 생성됐는지 여부를 알려주는 instantiaed 와 같은 내부 변수가 필요합니다.

그리고 싱글톤 패턴에서는 내부 변수에 접근할 수 있는 객체를 반환하는 클로저를 이용해야 합니다.

위 코드에서는 변수의 렉시컬한 특성으로 인해 내부의 getInstance 함수에서 비공개 변수인 instantiaed 에 접근할 수 있다는 것과 getInstane() 호출이 끝나더라도 instantiaed 값은 계속 유지되는 특성(클로저)을 이용해 publicMethod(), publicProp 이 포함된 객체를 유일하게 생성하게 됩니다.

그래서 singleton.getInstance 를 몇 번 호출하더라도 결과로 얻는 객체는 모두 동일한 싱글톤 객체를 가리키게 됩니다.

출처: <http://webclub.tistory.com/150> [Web Club]

[**팩토리 함수를 활용한 Object.create**](http://webclub.tistory.com/163)

10개월 전 by 하기실음 관두등가 재야재희

팩토리 함수를 활용한 Object.create



JavaScript

/\*\*

\* -------------------------------------------------------

\* Object.create() 를 지원하지 않는

\* 브라우저를 위한 크로스 브라우징 메서드 정의

\* -------------------------------------------------------

\*/

var objectCreate = function ( arg ) {

if(!arg) { return {}; }

function obj() {}

obj.prototype = arg;

return new obj;

};

Object.create = Object.create || objectCreate;

var proto = {

sentence : 1,

probation : 2

};

var makePrisoner = function (name, id) {

var prisoner = Object.create(proto);

prisoner.name = name;

prisoner.id = id;

return prisoner;

}

var first = makePrisoner('jaehee', 'sksk');

var second = makePrisoner('aion', 'ckckck');

출처: <http://webclub.tistory.com/163> [Web Club]

[**절차 지향 방식에서 객체지향 방식(함수기반 클래스,프로토타입방식 클래스) 으로 리팩토링하기**](http://webclub.tistory.com/154)

10개월 전 by 하기실음 관두등가 재야재희

함수기반 클래스, 프로토타입 방식 클래스



절차 지향 방식 코드

jQuery(**function**($){  
  
 console.time('성능');  
 /\* Radio \*/  
 **var** radioBox = $('.ctm-radio');  
 **var** addClassRadio = **function**($input){  
  
 $input.each(**function**(){  
 **if** ($(**this**).prop('checked')){  
 **if** ($(**this**).parent().hasClass('ctm-radio')){  
 **var** radioName = $(**this**).attr('name');  
 $('input[name='+radioName+']').parent('.ctm-radio').removeClass('checked');  
 }  
 $(**this**).parent().addClass('checked').siblings().removeClass('checked');  
 }  
 });  
 };  
 radioBox.on('change', 'input', **function**(){  
 addClassRadio($(**this**));  
 })  
 console.timeEnd('성능');  
});

객체 지향 방식 (함수기반의 클래스)

(**function** (global, $) {  
  
 console.time('성능');  
 **var** ctmRadio = customRadio();  
 ctmRadio.init('.ctm-radio')  
  
 **function** customRadio() {  
 **if** (**this** === window) {  
 **return new** customRadio;  
 }  
 // 요소 정의  
 **this**.radioBox = **null**;  
 **this**.chk = **null**;  
  
 // 요소 초기화  
 **this**.init = **function** (el) {  
 **this**.radioBox = $(el);  
 **this**.chk = 'checked';  
  
 // 초기화 이벤트 호출출  
 **this**.initEvent();  
 };  
  
 **this**.addClassRadio = **function** (ipt) {  
 **var** $input = $(ipt),  
 \_self = **this**;  
 $.each($input, **function**(index, el){  
 **var** \_$this = $input.eq(index);  
 **if** (\_$this.prop('checked')){  
 **if** (\_$this.parent().hasClass('ctm-radio')){  
 **var** radioName = \_$this.attr('name');  
 $('input[name='+radioName+']').parent('.ctm-radio').removeClass(\_self.chk);  
 }  
 \_$this.parent().addClass(\_self.chk).siblings().removeClass(\_self.chk);  
 }  
 });  
 };  
  
 **this**.initEvent = **function** () {  
 **var** \_self = **this**;  
 **this**.radioBox.on('change','input', **function** () {  
 \_self.addClassRadio(**this**);  
 })  
 }  
  
 }  
 console.timeEnd('성능');  
  
})(window, window.jQuery)

객체 지향 방식(프로토타입방식의 클래스)

(**function**(global, $){  
  
 console.time('성능');  
 $(runRadio);  
  
 **function** runRadio() {  
 **var** radio = RadioCustom();  
 radio.init('.ctm-radio');  
  
 }  
 // CLASS RadioCustom Definition  
 **function** RadioCustom() {  
  
 **if**(**this** === window) **return new** RadioCustom;  
  
 // Element definition  
 **this**.$radioBox = **null**;  
 **this**.chk = **null**;  
  
 }  
 // RadioCustom 초기화  
 RadioCustom.prototype.init = **function** (el) {  
  
 **this**.$radioBox = $(el);  
 **this**.chk = 'checked';  
 **this**.initEvent();  
  
 };  
 // RadioCustom Call Event  
 RadioCustom.prototype.initEvent = **function** () {  
 **var** objThis = **this**;  
 **this**.$radioBox.on('change', 'input', **function** () {  
 objThis.addClassRadio(**this**);  
 })  
 };  
 // RadioCustom class toggle  
 RadioCustom.prototype.addClassRadio = **function** (ipt) {  
 **var** $input = $(ipt),  
 \_self = **this**;  
 $.each($input, **function** (index, el) {  
 **var** \_$this = $input.eq(index);  
 **if** (\_$this.prop('checked')) {  
 **if** (\_$this.parent().hasClass('ctm-radio')){  
 **var** radioName = \_$this.attr('name');  
 $('input[name='+radioName+']').parent('.ctm-radio').removeClass(\_self.chk);  
 }  
 \_$this.parent().addClass(\_self.chk).siblings().removeClass(\_self.chk);  
 }  
 })  
 };  
 console.timeEnd('성능');  
})(window, window.jQuery);

지금까지 절차지향 방식을 두가지 타입의 객체 지향방식으로 리팩토링해 보았습니다.

혹시 이런 의문점이 생길 수도 있습니다. 객체지향방식 코드는 더 길어지고 가독성이 떨어지는거 아닌가 하고 말입니다.

절차지향방식으로 지금껏 코딩하시는 분들이 많이 있을 것입니다. 저 역시도 얼마전까지는 함수단위 기반의 절차지향이 익숙했습니다.

하지만 객체지향은 모듈별 관리를 할 수 있다는 점에서 유지보수 측면이나 가독성은 월등히 좋습니다. 다만 익숙하지 않을 뿐입니다.

제이쿼리 코드 또한 프로토타입 기반의 코드로 집약되어 있습니다. 그리고 위의 코드를 실행하면 콘솔창에서 성능시간을 체크하도록 해 놓았는데 확인해보면 프로토타입기반의 클래스가 성능은 훨씬 빠릅니다. 코드는 길지만 더 빠르다는 것입니다.

성능최적화와 유지보수 측면을 고려한다면 당연히 프로토타입을 이용해야 합니다. 물론 모든 기능을 프로토타입을 이용한다는 것이 아니라 재사용성 있는 코드들은 적어도 함수기반의 클래스나 좀 더 나아간다면 프로토타입을 이용하는 것이 더 낫다고 판단되어집니다.

출처: <http://webclub.tistory.com/154> [Web Club]

[**클래스 기반(일반 언어)과 프로토타입(자바스크립트) 기반 비교**](http://webclub.tistory.com/162)

10개월 전 by 하기실음 관두등가 재야재희

클래스 기반(일반 언어)과 프로토타입 기반 비교



클래스 기반(일반 언어)

Java

/\* 1단계 : 클래스 정의 \*/

public class Prisoner {

public int setence = 4;

public int probation = 2;

public string name;

public string id;

/\* 2단계 : 클래스 생성자 정의\*/

public Prisoner(string name, string id) {

this.name = name;

this.id = id;

}

}

/\* 3단계 : 객체 생성 \*/

Prisoner firstPrisoner = new Prisoner("Joe", "12A");

Prisoner secondPrisoner = new Prisoner("Sam", "2BC");

프로토타입 기반(자바스크립트)

JavaScript

/\* 1단계 : 프로토타입 객체 정의 \*/

var proto = {

sentence : 2,

probation : 2

};

/\* 2단계 : 객체 생성자 정의 \*/

var Prisoner = function(name, id) {

this.name = name;

this.id = id;

}

/\* 3단계 : 생성자와 프로토타입 객체 연결 \*/

Prisoner.prototype = proto;

/\* 4단계 : 객체 생성 \*/

var firstPrisoner = new Prisoner("Joe", "12A");

var secondPrisoner = new Prisoner("Joe", "12A");

두가지 방식 모두 먼저 객체 템플릿을 만들었습니다. 클래스 기반 프로그래밍에서는 이런 템플릿을 클래스라 부르고 프로토타입 기반 프로그래밍에서는 이를 프로토타입 객체라고 부르지만 두 템플릿의 목적은 모두 같습니다.

바로 객체를 생성할 수 있는 프레임워크 역할을 하는 것입니다.

두번재로 생성자를 만들었고 클래스 기반 언어에서는 객체를 생성할 때 생성자가 어떤 클래스에 속하는지 명확히 알 수 있도록 생성자를 클래스 내부에 정의합니다. 자바스크립트에서는 객체 생성자를 프로토타입 외부에서 설정하고 있기 때문에 둘을 서로 연결해 주는 단계(3단계)가 필요합니다. 마지막으로 객체를 생성합니다.

출처: <http://webclub.tistory.com/162> [Web Club]

[**함수의 프로토타입(prototype) 핵심 정리**](http://webclub.tistory.com/509)

10개월 전 by 하기실음 관두등가 재야재희

프로토타입 체인

자바스크립트는 Function() 인스턴스에 자동으로 prototype 이라는 속성을 만듭니다.

구체적으로 말하자면 prototype 속성은 new 키워드와 생성자 함수를 같이 사용해서 만든 객체에 연결됩니다.

인스턴스들은 생성자 함수의 prototype 속성을 통해 공통의 메소드와 속성을 공유하고 상속합니다.

중요한 사실은 이러한 공유가 속성을 검색하는 동안 일어난다는 것입니다.

즉, 어떤 객체의 속성을 찾거나 접근하면 자바스크립트는 그때마다 해당 객체는 물론 프로토타입 체인에서도 그 속성을 찾는다는 것입니다.



함수는 모두 prototype 객체를 포함하고 있다.

심지어 생성자 함수로 사용할 생각이 전혀 없는 함수도 prototype 객체를 자동으로 포함합니다.

다음은 Array() 생성자를 이용해 배열을 하나 만든 후 배열의 join() 메소드를 호출하는 코드입니다.

javascript

var myArr = new Array('foo', 'bar');

console.log(myArr.join()); // foo,bar 가 기록된다.

join() 메소드는 myArr 객체 인스턴스에 정의되어 있지 않은 속성입니다.

하지만 사용자는 join() 메소드가 마치 원래 있던 것처럼 사용할 수 있었습니다.

분명히 이 메소드는 어디엔가 정의되어 있을 것입니다.

하지만 어디일까? 이 메소드는 Array() 생성자의 prototype 속성에 속성으로서 정의되어 있습니다.

배열 객체 인스턴스에서는 join() 메소드가 없기 때문에 자바스크립트는 프로토타입 체인에서 join() 이라는 메소드가 있는지 검색합니다.

왜 이런 식으로 동작하도록 만들었을까?

사실 이는 효율성과 재사용성에 대한 문제입니다.

배열 생성자 함수가 만드는 배열 인스턴스마다 언제나 똑같은 방식으로 동작하는 join() 메소드를 굳이 일일이 추가할 필요가 있을까?

모든 배열 인스턴스마다 함수를 새로 만드는 것보다 하나의 join() 메소드를 모든 배열에서 가져다 쓰는 편이 더 합리적일 것입니다.

자바스크립트에서는 prototype 속성, 프로토타입 결합(prototype linkage), 프로토타입 체인 검색을 통해 이러한 효율성을 이루고 있습니다.

prototype 속성은 왜 중요한가?!

네이티브 생성자 함수(예: Object(), Array(), Function() 등)는 prototype 속성을 사용해 생성자 인스턴스가 메소드와 속성을 상속받도록 하고 있기 때문입니다.

자바스크립트는 이러한 메커니즘을 사용해 객체 인스턴스가 생성자 함수의 prototype 속성을 상속받을 수 있도록 지원해 주고 있습니다.

따라서, 자바스크립트를 더 잘 이해하려면 자바스크립트가 prototype 객체를 어떻게 사용하는지 잘 알아야 할 것입니다.

사용자 정의 생성자 함수를 만들 때 자바스크립트 네이티브 객체와 동일한 방식을 통해 프로토타입 상속을 구현할 수 있습니다.

그러나 상속이 이루어지는 방식에 대한 이해가 반드시 수반되어야 합니다.

프로토타입 상속을 싫어하거나 다른 상속 패턴을 더 선호할 수도 있겠지만 현실적으로 생각해 볼 때 아마도 누군가 프로토타입 상속을 사용해 구현해 놓은 코드를 수정하거나 조작해야 할 일이 생길 것입니다.

이때 프로토타입 상속이 어떻게 동작하는지 알아야 다른 개발자가 만든 생성자 함수의 기능을 그대로 복제할 수 있을 것입니다.

프로토타입 상속을 사용하면 동일한 메소드를 공유하는 여러 개의 효율적인 객체 인스턴스를 만들 수 있습니다.

앞서 말한 것과 같이 Array() 생성자의 인스턴스인 배열 객체는 굳이 인스턴스마다 join() 메소드를 추가해주지 않아도 됩니다.

모든 배열 인스턴스의 프로토타입 체인에는 join() 메소드가 저장되어 있으므로 모든 배열 인스턴스는 동일한 join() 메소드르 사용할수 있습니다.

모든 Function() 인스턴스에는 prototype 속성이 있다.

자바스크립트에서 함수는 Function() 생성자를 직접 호출하여 만들었든 리터럴 표기법을 사용해 만들었든 예외없이 모드 Function() 생성자로부터 만들어집니다.

함수 인스턴스를 만들면 인스턴스에는 항상 prototype 속성이 추가됩니다.

이때 prototype 속성은 그 자체로는 빈 객체와 같습니다.

다음 코드에 myFn 이라는 함수를 정의한 후, 이 함수의 prototype 속성에 접근해 보도록 해봅니다.

javascript

var myFn = function () {

};

console.log(myFn.prototype); // Object() 가 기록된다.

console.log(typeof myFn.prototype); // 'object' 가 기록된다.

Function() 생성자를 사용할 때문 언제나 자동으로 설정되는 prototype 속성은 중요한 속성이므로 잘 이해하도록 해야합니다.

비록 위 코드에서는 생성자로 사용한 사용저 정의 함수가 하나밖에 없지만, Function() 생성자는 모든 함수 인스턴스에 prototype 속성을 부여합니다.

prototype 속성은 Object() 객체다.

프로토타입에 대한 내용은 조금 어렵습니다.

엄밀히 말해 prototype 은 Function() 인스턴스를 만들 때 자바스크립트가 인스턴스에 부여하는 이름이 'prototype' 이고 기본값이 빈 객체인 속성일 뿐입니다.

만약 이를 사용자가 직접 코드로 만든다면 다음과 같이 작성할 수 있을 것입니다.

javascript

var myFunc = function () {

};

myFunc.prototype = {}; // 시용자가 직접 prototype 속성을 추가하고 속성값으로 빈 객체를 할당

console.log(myFunc.prototype); // 빈 객체인 Object {} 가 기록된다.

사실 prototype 속성의 값으로는 어떤 복합 객체(예: 배열등..)든 사용할 수 있습니다.

prototype 에 원시값을 설정하려 하면 자바스크립트에서는 해당 코드를 무시하게 된다.

생성자 함수를 통해 만든 인스턴스는 생성자 함수의 prototype 속성과 연결되어 있다.

생성자 함수의 prototype 속성은 그 자체만 놓고 보면 객체일 뿐이지만 프로토타입 체인을 통해 인스턴스와 연결되는 특이한 속성입니다.

다시말해, new 키워드로 생성자 함수를 사용해 객체를 만들면 생성자 함수의 prototype 속성과 새롭게 만들어진 객체 인스턴스 사이에는 일종의 숨겨진 연결고리가 생깁니다.

일부 브라우저에서는 이 연결고리가 인스턴스의 \_\_proto\_\_속성으로 나타나기도 합니다.

자바스크립트는 생성자 함수를 사용해 인스턴스를 만들 때 인스턴스 객체와 생성자 함수를 자동으로 연결해 두며, 이러한 연결 덕분에 프로토타입 체인이 형성됩니다.

다음은 네이티브 Array() 생성자의 prototype 에 속성을 하나 추가한 후 배열 인스턴스의 \_\_proto\_\_ 속성을 사용해 추가한 속성에 접근하는 코드입니다.

javascript

// 해당 코드는 \_\_proto\_\_ 를 구현해 놓은 브라우저(FF, Chrome, 사파리, 안드로이드)에서만 동작한다.

Array.prototype.foo = 'foo';

var myArr = new Array();

console.log(myArr.\_\_proto\_\_.foo);

// myArr.\_\_proto\_\_ == Array.prototype 이므로 foo 가 기록될 것이다.

사실 \_\_proto\_\_ 속성은 ECMA 공식 표준이 아니기 때문에 이보다 다른 방법을 사용할 수 있습니다.

객체 인스턴스에서 constructor 속성을 사용하면 생성자를 구할 수 있고 이를 통해 인스턴스가 상속받은 프로토타입 객체를 구할 수 있습니다.

다음 코드를 살펴봅니다.

javascript

Array.prototype.foo = 'foo'; // Array() 인스턴스는 모두 foo 속성을 상속받게 된다.

var myArray = new Array();

// .constructor.prototype 을 사용하면 조금 복잡하지만 foo 속성을 구할 수 있다.

console.log(myArray.constructor.prototype.foo); // foo 가 기록된다.

// 물론 프로토타입 체인을 사용해도 된다.

console.log(myArray.foo); // 프로토타입 체인을 사용한 Array.prototype.foo 를 검색하게 된다.

이 코드에서 foo 속성은 prototype 객체에서 찾을 수 있었습니다.

이는 Array() 인스턴스와 Array() 생성자의 프로토타입 객체(==Array.prototype) 간에 연결고리가 있었기 때문에 가능한 것이었습니다.

간단히 말해서 myArray.\_\_proto\_\_(또는 myArray.constructor.prototype) 은 Array.prototype 을 참조합니다.

프로토타입 체인의 끝은 Object.prototype 이다.

prototype 속성은 객체 이기 때문에 프로토타입 체인 또는 프로토타입 검색의 종점은 Object.prototype 이다.

javascript

var meArray = [];

console.log(meArray.foo); // undefined 가 기록된다.

/\*

meArray.foo, Array.prototype.foo, Object.prototype.foo 에서 foo 를 찾지 못했기 때문에

foo 는 undefined 를 기록하게 된다.

\*/

위 코드에서 meArray 라는 빈 배열을 만든 후 프로토타입 체인을 검색하도록 meArray 에 정의되지 않은 속성에 접근해 보았습니다.

자바스크립트는 먼저 meArray 객체에서 foo 속성을 찾아보지만, 찾지 못하기 때문에 Array.prototype 에서 다시 검색을 시도할 것입니다.

하지만 여기서도 찾을 수 없으므로 마지막 종착지인 Object.prototype 에서 foo 속성을 찾아볼 것입니다.

하지만 검색의 끝 지점인 여기서에서 찾을 수 없으므로 결국 foo 속성은 undefined 가 될 것입니다.

프로토타입 체인은 Object.prototype 에서 끝난다.

foo 속성이 있는지 가장 마지막에 찾아본 곳은 Object.prototype 이었다.

Object.prototype 에 속성을 추가하면 추가한 속성이 for in 반복문에 나타난다.

프로토타입 체인은 체인에서 제일 먼저 찾은 속성을 반환한다.

스코프 체인과 마찬가지로 프로토타입 체인은 체인을 검색하다가 가장 먼저 발견한 값을 사용합니다.

바로 앞에서 사용했던 코드를 조금 수정하여 Object.prototype 과 Array.prototype 객체에 똑같은 속성을 추가한 후 배열 인스턴스에서 그 값에 접근해 보도록 하겠습니다.

Array.prototype 객체의 값이 반환될 것입니다.

javascript

Object.prototype.foo = 'object-foo';

Array.prototype.foo = 'array-foo';

var arr = [];

console.log(arr.foo); // Array.prototype.foo 에서 찾은 'array-foo' 가 기록된다.

arr.foo = 'bar';

console.log(arr.foo); // arr.foo 에서 찾은 'bar' 가 기록된다.

이 코드에서 Array.prototype.foo 의 foo 값은 Object.prototype.foo 의 foo 값을 가리게 됩니다.

프로토타입에서 속성을 검색할 때는 가장 먼저 발견한 속성을 사용하고 검색을 종료합니다.

체인에 똑같은 이름의 속성이 몇 개나 더 있었는지는 상관없습니다.

prototype 속성을 새 객체로 대체하면 기본 constructor 속성이 삭제된다.

prototype 속성의 기본값은 다른 값으로 대체할 수 있습니다.

하지만 prototype 속성을 바꾸면 원래의 prototype 객체에서 볼 수 있었던 기본 constructor 속성도 사라지게 됩니다.

다음은 Foo 생성자 함수를 만들고 Foo 의 prototype 속성을 빈 객체로 대체한 후 인스턴스의 constructor 속성이 사라졌는지 확인해 보는 코드입니다.

이제 이 코드에서는 constructor 속성은 Object() 생성자를 참조하게 될 것입니다.

javascript

var Foo = function Foo() {

};

Foo.prototype = {}; // 빈 객체로 prototype 속성을 대체한다.

var fooInstance = new Foo();

console.log(fooInstance.constructor == Foo); // false 가 기록된다.

// 참조가 망가졌다. 즉, 연결고리가 끊어진 것이다.

console.log(fooInstance.constructor); // Foo 생성자 함수 가 아닌 Object() 가 기록된다.

// prototype 값을 대체하지 않은 경우와 비교해 본다.

var Bar = function Bar() {

};

var barInstance = new Bar();

console.log(barInstance.constructor == Bar); // true 가 기록된다.

console.log(barInstance.constructor); // Bar 생성자 함수가 기록된다.

만약 자바스크립트가 설정한 기본 prototype 속성을 대체할 생각이라면 즉, 자바스크립트 객체지향 패턴에서 종종 사용되는 방식등을 사용할 경우에는 생성자 함수를 참조하는 constructor 속성을 원래대로 복원해주어야 합니다.

다음은 앞의 코드를 조금 수정하여 constructor 속성이 원래의 생성자 함수를 올바르게 참조하도록 해봅니다.

javascript

var Foo = function Foo() {

};

Foo.prototype = { constructor : Foo };

var fooInstance = new Foo();

console.log(fooInstance.constructor == Foo); // true 가 기록된다.

console.log(fooInstance.constructor); // Foo 생성자 함수가 기록된다.

프로토타입에서 상속한 속성은 가장 최근의 값을 사용한다.

인스턴스가 프로토타입에서 상속한 속성은 그 속성이 어떻게 만들어지고, 변경되고, 추가되었든 상관없이 항상 가장 최근의 값을 사용합니다.

다음 코드는 Foo 생성자를 만들고 x 라는 속성을 prototype 에 추가한 후 Foo() 생성자의 인스턴스를 fooInstance 라는 이름으로 만들었습니다.

여기서 한번 x 의 값을 기록한 후 생성자의 prototype 속성에 포함된 x 의 값을 수정한 후 다시 한번 x 의 값을 기록했습니다.

이를 통해 우리는 인스턴스가 prototype 객체에서 가져오는 값은 가장 마지막 값이라는 사실을 알 수 있을 것입니다.

javascript

var Foo = function Foo() {

};

Foo.prototype.x = 10;

var FooInstance = new Foo();

console.log(FooInstance.x); // 10 이 기록된다.

Foo.prototype.x = 20;

console.log(FooInstance.x); // 20 이 기록된다. FooInstance 도 갱신된 것이다.

이러한 현상은 기본 prototype 객체를 사용했든 또는 prototype 객체를 새로운 객체로 대체했든 상관없이 똑같이 볼 수 있습니다.

다음 코드에서는 기본 prototype 객체 대신에 새로운 객체를 사용해 이 같은 사실을 증명해 보도록 합니다.

javascript

var Foo = function Foo() {

};

Foo.prototype = { x: 10};

var FooInstance = new Foo();

console.log(FooInstance.x); // 10 이 기록된다.

Foo.prototype.x = 20;

console.log(FooInstance.x); // 20 이 기록된다. FooInstance 도 갱신된 것이다.

prototype 속성을 새 객체로 대체하면 이전에 만든 인스턴스는 갱신되지 않는다.

지금까지 살펴본 바에 따르면 아마도 prototype 속성을 언제든 완전히 대체할 수 있고 그렇게 하면 모든 인스턴스가 갱신될 것이라고 생각할 수도 있습니다.

하지만 이는 잘못된 생각입니다.

인스턴스를 만들면 인스턴스를 만들 때의 prototype 과 인스턴스가 서로 묶여 버리기 때문에 prototype 속성에 새 객체를 설정하면 이미 만들어진 인스턴스와 새로운 prototype 간의 연결고리가 끊어져버리게 됩니다.

하지만 앞서 살펴본 방법을 사용하면 인스턴스를 만들었던 시점의 prototype 객체를 찾아서 값을 수정하거나 추가할 수 있으며 이를 통해 이 prototype 과 연결괸 인스턴스를 갱신할 수 있습니다.

javascript

var Foo = function Foo() {

};

Foo.prototype.x = 10;

var FooInstance = new Foo();

console.log(FooInstance.x); // 예상한 대로 10 이 기록된다.

// prototype 객체를 새로 만든 Object() 객체로 대체/정의해 보도록 한다.

Foo.prototype = {x: 20};

console.log(FooInstance.x); // 10 이 기록된다.

/\*\*

\* 위에서 prototype 을 새로운 객체로 갱신했으니까 20 이 되어야 하지 않을까? 라고 생각할 수 있다.

\* FooInstance 는 여전이 처음 인스턴스로 만들어진 시점의 prototype 객체를 참조하고 있다.

\*\*/

// Foo() 의 인스턴스를 새로 만들어 본다.

var NewFooInstance = new Foo();

// 새로 만든 인스턴스는 새로운 prototype 객체인 { x : 20 } 와 묶여있게 된다.

console.log(NewFooInstance.x); // 20 이 기록된다.

여기서 알 수 있는 사실은 인스턴스를 만든 뒤에는 객체의 prototype 속성을 새 객체로 대체하면 안된다는 것입니다.

만약 새 객체로 대체해버리면 같은 생성자에서 만든 인스턴스라 해도 서로 다른 prototype 객체를 참조하게 될 것입니다.

사용자 정의 생성자도 네이티브 생성자처럼 프로토타입을 상속할 수 있다.

지금까지 자바스크립트에서 prototype 속성을 사용해 상속하는 법(예: Array.prototype)을 살펴보았습니다.

같은 패턴을 네이티브가 아닌 사용자가 직접 만든 생성자에도 적용할 수 있습니다.

다음은 Person 사용자 생성자함수를 가지고 객체를 생성하여 자바스크립트 상속 패턴을 흉내내는 코드입니다.

javascript

var Person = function () {

};

// 모든 Person 인스턴스는 legs, arms, countLimbs 속성을 상속하도록 정의한다.

Person.prototype.legs = 2;

Person.prototype.arms = 2;

Person.prototype.countLimbs = function () {

return this.legs + this.arms;

};

var chuck = new Person();

console.log(chuck.countLimbs()); // 4 가 기록된다.

위의 코드에서 Person() 생성자 함수를 만든 후 Person() 의 prototype 속성에 몇 개의 속성을 추가하여 모든 인스턴스가 상속받도록 정의했습니다.

이 코드는 자바스크립트가 네이티브 객체를 상속할 대 사용했던 것과 똑같은 방법으로 프로토타입 체인을 사용한 것입니다.

전달된 매개변수가 없을 때 프로토타입에서 속성을 상속받은 생성자 함수를 만들어보면 이를 조금 더 잘 이해할 수 있을 것입니다.

다음 코드에서 Person() 생성자는 전달된 매개변수가 있으면 이를 사용해 인스턴스 속성을 추가하지만 전달된 값이 아예 없거나 한 개만 있으면 프로토타입에서 상속받은 값을 사용합니다.

인스턴스 속성이 있으면 상속된 속성이 사용되지 않습니다.

따라서 어느 경우에든 속성을 문제없이 사용할 수 있을 것입니다.

javascript

var Person = function (legs, arms) {

// 프로토타입에서 상속받은 값을 가린다.

if (legs !== undefined) {

this.legs = legs;

}

if (arms !== undefined) {

this.arms = arms;

}

};

Person.prototype.legs = 2;

Person.prototype.arms = 2;

Person.prototype.countLimbs = function () {

return this.legs + this.arms;

};

var chuck = new Person(4,4);

console.log(chuck.countLimbs()); // 8 이 기록된다.

// 매개변수를 전달하지 않을 경우

var chuck2 = new Person();

console.log(chuck2.countLimbs()); // 4 가 기록된다.

// 전달된 매개변수가 없기 때문에 Person 에는 속성을 가지고 있지 않다.

// 하지만 프로토타입 체인을 통해 검색한 결과를 통해 4 를 기록하게 되는 것이다.

상속 체인 만들기

프로토타입 상속은 전통적인 객체 지향 프로그래밍 언어에서 볼 수 있던 상속 패턴을 흉내내기 위해 만들어진 것입니다.

자바스크립트에서 인스턴스란 간단히 말해 다른 객체의 속성에 접근할 수 있는 객체입니다.

이를 위해 먼저 상속받고자 하는 부모 객체의 인스턴스를 만든 후 생성자의 prototype 에 할당하면 부모 객체를 상속받을 수 있습니다.

prototype 속성에 부모 객체의 인스턴스를 할당하고 나면 부모 객체 생성자의 prototype 과 상속받는 객체 사이에는 연결고리(\_\_proto)가 생기게 됩니다.

다음의 코드를 살펴봅니다.

javascript

var Person = function () {

this.bar = 'bar';

};

Person.prototype.foo = 'foo';

var Chef = function () {

this.goo = 'goo';

};

Chef.prototype = new Person(); // Person() 객체의 인스턴스를 할당

var cody = new Chef();

위 코드에서 Chef 객체(=cody) 는 Person 을 상속받습니다.

다시 말해, 어떤 속성을 Chef 객체에서 발견하지 못하면 그 다음에는 Person() 생성자 함수의 prototype 에서 속성을 찾는다는 뜻입니다.

이렇게 상속 관계를 만들기 위해 해야 할 일은 Chef.prototype 의 값으로 Person() 객체의 인스턴스를 할당 해주는 것이 전부입니다.

위 코드에서 한 일은 네이티브 객체에서 원래 사용하고 있던 시스템을 빌려온 것 뿐입니다.

prototype 속성에 있어서 원래 사용하던 Object() 값이나 Person() 값은 다른 점이 없습니다.

다시 말해, 객체의 상속된 속성에 접근하면 객체를 만든 생성자 함수의 prototype 속성에서 그 값을 찾을 것이라는 의미입니다.

출처: <http://webclub.tistory.com/509> [Web Club]

[**new Object() 의 문제점**](http://webclub.tistory.com/513)

10개월 전 by 하기실음 관두등가 재야재희

객체 생성자의 함정

객체 리터럴을 사용할 수 있는 상황에서는 new Object() 생성자를 쓸 이유가 없지만, 다른 사람이 작성한 레거시 코드를 물려받을 수도 있기 때문에 이 생성자의 '기능'하나를 알아둘 필요가 있습니다.

결국 이는 생성자를 써서는 안되는 이유이기도 합니다.

문제의 기능은 Object() 생성자가 인자를 받을 수 있다는 점입니다.

인자로 전달되는 값에 따라 생성자 함수가 다른 내장 생성자에 객체 생성을 위임할 수 있고, 따라서 기대한 것과는 다른 객체가 반환되기도 합니다.



다음은 new Object() 에 숫자, 문자열, 불린 값을 전달한 몇 가지 예입니다.

예상한 바와 다른 생성자로 생성된 객체가 반환됩니다.

JavaScript

// 경고 : 모두 안티 패턴이다.

// 빈 객체

var obj = new Object();

console.log(obj.constructor === Object); // true 가 기록

// 숫자 객체

var obj = new Object(10);

console.log(obj.constructor === Number); // true 가 기록

obj.toFixed(2); // 1.00 이 기록

// 문자열 객체

var obj = new Object('I am a string');

console.log(obj.constructor === String); // true 가 기록

// 일반적인 객체에는 substring() 이라는 메서드가 없지만 문자열 객체에는 있다.

console.log(typeof obj.substring); // function 이 기록

// 불린 객체

var obj = new Object(true);

console.log(obj.constructor === Boolean); // true 가 기록

Object() 생성자의 이 같은 동작 방식 때문에, 런타임에 결정하는 동적인 값이 생성자에 인자로 전달될 경우 예기치 않은 결과가 반환될 수도 있습니다.

거듭 말하자만 결론적으로, new Object() 를 사용하지 마라.

더 간단하고 안정적인 객체 리터럴을 사용하는 것을 권장합니다.

출처: <http://webclub.tistory.com/513> [Web Club]

[**module pattern (모듈패턴) #1**](http://webclub.tistory.com/5)

10개월 전 by 하기실음 관두등가 재야재희

Module Pattern

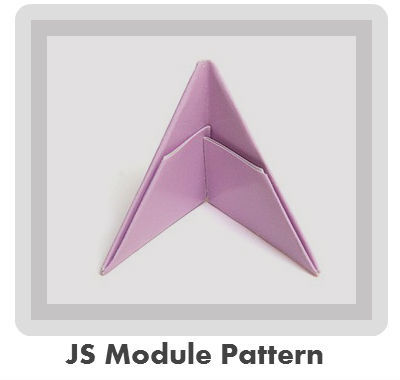
모듈 패턴을 알아보기 전에 네임스페이스란 용어부터 짚고 넘어가 보겠습니다.

네임스페이스란 수많은 함수, 객체, 변수들로 이루어진 코드가 전역 유효범위를 어지럽히지 않고, 애플리케이션이나 라이브러리를 위한 하나의 전역 객체를 만들고 모든 기능을 이 객체에 추가하는 것을 말합니다.

예를 들어 jQuery와 같은 라이브러리에서 jQuery가 네임스페이스를 가지며 jQuery 이름 내에 모든 기능을 추가하는 것을 말합니다.

다시 말해서, 코드에 네임스페이스를 지정해주며, 코드 내의 이름 충돌뿐만 아니라 이 코드와 같은 페이지에 존재하는 또 다른 자바스크립트 라이브러리나 위젯등 서드파티 코드와의 이름 충돌도 미연에 방지해 주는 것입니다.

우리는 이러한 네이스페이스 패턴을 작은 기능들 만을 모아 놓은 패턴으로 사용할 수가 있는데 모듈 패턴으로 이를 흉내낼 수가 있습니다.



모듈(module) 이라는 것은 전체 애플리케이션의 일부를 독립된 코드로 분리해서 만들어 놓은 것을 말합니다. 관련된 유용한 기능을 모아둔 모듈이라 할 수 있으며 크게는 jQuery 와 같은 라이브러리도 모듈이라 할 수 있습니다. 그리고 jQuery API 카테고리 분류를 보면 css, core, ajax 등등으로 API 를 제공하고 있는데 이 또한 카테고리 섹션을 모듈별로 관리를 하고 최종 빌드버전을 배포하고 있습니다. 물론 jQuery를 모듈별로 사용자가 다운받아 사용할 수도 있습니다.

자바스크립트에서는 모듈을 구현하는 가장 쉬운 방법은 객체 리터럴을 사용하는 방법입니다.

javascript

var module = {

key : 'value',

pulicMethod : function () {

}

}

전의 포스팅에서 객체 리터럴을 싱글톤이라고 언급한 바 있습니다. 객체 리터럴 표현이 싱글톤 패턴이라고 해서 다른 패턴이 되지 말라는 법은 없습니다.

즉, 동일한 코드를 어떤 관점에서 보느냐에 따라 다양한 패턴이 될 수 있습니다. 객체 리터럴이 하나의 객체라는 점에서 싱글톤 패턴이라고 할 수도 있고 독립된 모듈이라는 점에서 모듈 패턴의 하나라고도 할 수 있습니다.

리고 독립된 모듈은 자체적으로 필요한 내부 변수 및 내부 함수를 모두 갖고 있어야 합니다. 이와 같이 내부 변수와 내부 함수를 가지고 있는 객체를 생성한다고 했을 때 클로저를 이용해야 합니다.

다음은 클로저를 이용해 모듈 패턴을 구현한 코드입니다.

javascript

var module = (function () {

/\*\*

\* --------------------------------

\* 모듈 패턴을 구현한 클로저 코드

\* --------------------------------

\*/

// 은닉될 멤버 정의

var privateKey = 0;

function privateMethod() {

return ++privateKey;

}

// 공개될 멤버 (특권 메소드) 정의

return {

publickey : privateKey,

publicMethod : function () {

return privateMethod();

}

}

})();

console.log(module.publicMethod()); // 1

console.log(module.publicMethod()); // 2 (클로저로 인한 결과)

모듈 패턴은 반환값이 함수가 아니라 객체이며, 자동 호출된다는 점만 제외하고 클로저와 유사합니다. 그리고 인스턴스를 여러 개 만들어 낼 수 있는 구조라는 점에서 싱글톤 패턴과 차이가 있습니다.

위의 코드를 실행하면 익명함수가 자동으로 호출되어 익명함수가 반환하는 객체가 Module 변수에 할당되게 됩니다. /p>

따라서 위와 같이 module.publicMethod() 호출할 수 있습니다. 위의 코드는 하나의 인스턴스 객체만을 생성하고 있어서 싱글톤과 유사합니다.

하지만 아래와 같이 자동으로 호출(self-invoking) 되는  구조를 없애면 여러 개의 인스턴스를 생성하여 사용할 수 있습니다.<

자동 호출 구조를 없앤 코드입니다.

javascript

var Module = function () {

/\*\*

\* --------------------------------

\* 모듈 패턴을 구현한 클로저 코드

\* --------------------------------

\*/

// 은닉될 멤버 정의

var privateKey = 0;

function privateMethod() {

return ++privateKey;

}

// 공개될 멤버 (특권 메소드) 정의

return {

publickey : privateKey,

publicMethod : function () {

return privateMethod();

}

}

};

// 두 개의 인스턴스 생성

var obj1 = Module();

obj1.publicMethod(); // 1 출력

obj1.publicMethod(); // 2 출력

var obj2 = Module();

obj2.publicMethod(); // 1 출력

obj2.publicMethod(); // 2 출력

위와 같이 Module 함수를 정의(즉시실행 X) 하여 함수를 호출하면 여러 개의 인스턴스인 객체를 생성하여 사용할 수 있습니다.

클로저 인스턴스와 유사하지만 한가지 차이점은 내부의 익명함수에서 반환값이 함수가 아니라 객체를 반환한다는 점입니다.

모듈 패턴의 장,단점

단점

- 전체적으로 코드량이 약간 더 많아지고 따라서 다운로드해야 하는 파일크기도 늘어난다.

- 전역 인스턴스가 단 하나뿐이기 때문에 코드의 어느 한 부분이 수정되어도 전역 인스턴스를  수정하게 된다.

즉, 나머지 기능들도 갱신된 상태를 물려받게 된다.

장점

- 점점 더 늘어만 가는 코드를 정리할때 널리 사용되며 자바스크립트 코딩패턴에서 널리 권장되는 방법이기도 하다.

모듈 기본패턴 정의하기

javascript

// 1. 네임스페이스를 설정하고 모듈을 정의

var MyApp = {} // 전역 객체

MyApp.modules = {}

/\*

2. 공개범위(특권메소드 등..)와 비공개 유효범위를 만든다

==> 즉시 실행함수로 모듈이 될 객체를 반환하고

모듈 사용자에게 제공할 공개 인터페이스가 담기게 된다.

\*/

MyApp.modules.libs = (function() {

// 비공개 프로퍼티

// var 선언 및 비공개 메소드등의 유효범위 (private 멤버)

// 공개 API (public, previlege 멤버)

return {

};

}());

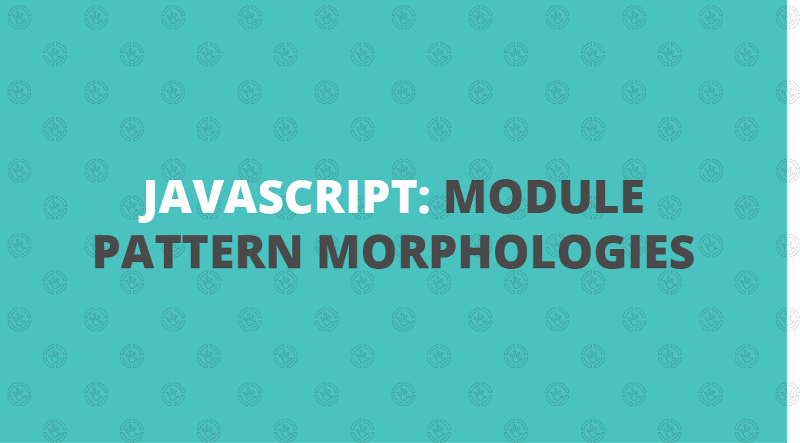
출처: <http://webclub.tistory.com/5> [Web Club]

## [모듈화와 모듈 패턴(module pattern) #2](http://webclub.tistory.com/78)

10개월 전 by 하기실음 관두등가 재야재희

Module Pattern

모듈화와 모듈 패턴에 대해 알아봅니다.



먼저 이 개념에 대해 알아보기 전에 고유변수(private member)와 특권메소드(privileged member)에 대해 간략히 짚고 넘어가자.

자바 등 다른 언어와는 달리 자바스크립트에는 private, protected, public 프로퍼티와 메서드를 나타내는 별도의 문법이 없다.

객체의 모든 멤버는 public, 즉 공개되어 있다.

다시 말해서,자바스크립트에는 고유 구성원(private member)이란 개념이 없으며 객체의 프로퍼티는 모두 공용(public)이다.

하지만 '고유변수'란 개념은 존재한다.

예를 들어 함수 안에서 정의한 변수는 함수 밖에서 접근할 수가 없으므로 모두 고유변수(private member)라고 간주한다.

이 고유변수에는 함수의 매개변수,지역 변수, 내부 함수 등이 포함된다

아래에서는 먼저 모듈 패턴 #1 에서 알아본 네임스페이스 패턴에 대해 추가적으로 알아보고 위에서 언급한 내용에 대해 계속 알아보도록 하겠습니다.

네임스페이스 패턴 생성시 유용한 범용적인 네임스페이스 함수

프로그램의 복잡도가 증가하고 코드의 각 부분들이 별개의 파일로 분리되어 선택적으로 문서에 포함하게 되면, 어떤 코드가 특정 네임스페이스나 그 내부의 프로퍼티를 처음으로 정의한다고 가정하기가 위험합니다.

네임스페이스에 추가하려는 프로퍼티가 이미 존재할 수도 있고 따라서 내용을 덮어쓰게 될 지도 모릅니다.

그러므로 네임스페이스를 생성하거나 프로퍼티를 추가하기 전에 먼저 이미 존재하는지 여부를 확인하는 것이 최선입니다.

다음의 예제를 살펴보도록 합니다.

javascript

// 위험하다.

var MYAPP = {};

// 개선안

if (typeof MYAPP === 'undefined') {

MYAPP = {};

}

// 또는 더 짧게 작성할 수 있다.

var MYAPP = MYAPP || {};

이렇게 추가되는 확인 작업 때문에 상당량의 중복 코드가 생겨날 수 있습니다.

예를 들어 MYAPP.modules.module2 를 정의하려면, 각 단계의 객체와 프로퍼티를 정의할 때마다 확인 작업을 거쳐야 하므로 코드가 세 번 중복됩니다.

따라서 네임스페이스 생성의 실제 작업을 맡아 줄 재사용 가능한 함수를 만들어두면 편리합니다.

다음은 네임스페이스 함수를 구현한 예제입니다.

다음과 같은 방식은 해당 네임스페이스가 존재하면 덮어쓰지 않기 때문에 기존 코드를 망가뜨리지 않습니다.

var MYAPP = MYAPP || {};

MYAPP.namespace = function (ns\_string) {

var parts = ns\_string.split('.'),

parent = MYAPP,

i;

// 처음에 중복되는 전역 객체명은 제거한다.

if (parts[0] === 'MYAPP') {

parts = parts.slice(1);

}

for (i = 0; i < parts.length; i += 1) {

if (typeof parent[parts[i]] === 'undefined') {

parent[parts[i]] = {};

}

parent = parent[parts[i]];

}

return parent;

};

이 코드는 다음 모든 예에서 사용할 수 있습니다.

javascript

var module2 = MYAPP.namespace('MYAPP.modules.module2');

console.log(module2);

console.log(module2 === MYAPP.modules.module2); // true 가 기록

// 첫 부분의 'MYAPP' 을 생략하고도 사용할 수 있다.

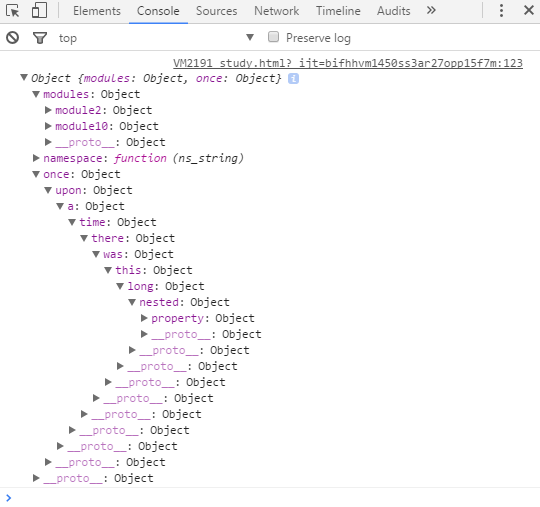
MYAPP.namespace('modules.module10');

// 매우 긴 네임스페이스를 만들어 본다.

MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');

console.log(MYAPP);

다음의 그림은 위 예제를 요소 검사기로 MYAPP 을 실제로 검사했을 때의 모습입니다.



의존 관계 선언

자바스크립트 라이브러리들은 대개 네임스페이스를 지정하여 모듈화되어 있기 때문에, 필요한 모듈만 골라서 사용할 수 있습니다.

예를 들어, YUI2 에는 네임스페이스 역할을 하는 YAHOO 라는 전역 변수가 있고, 이 전역 변수의 프로퍼티로 YAHOO.util.DOM(DOM 모듈)이나 YAHOO.util.Event(이벤트 모듈)와 같은 모듈이 추가되어 있습니다.

이때 함수나 모듈 최상단에, 의존 관계에 있는 모듈을 선언하는 것이 좋습니다.

즉, 지역변수를 만들어 모듈을 가리키도록 선언하는 것입니다.

javascript

var myFunction = function () {

// 의존 관계에 있는 모듈들

var event = YAHOO.util.Event,

dom = YAHOO.util.Dom;

// 이제 event 와 dom 이라는 변수를 사용한다...

};

대단히 간단한 패턴이지만 상당히 많은 장점을 가지고 있습니다.

* 의존 관계가 명시적으로 선언되어 있기 때문에 코드를 사용하는 사람이 페이지 내에서 반드시 포함시켜야 하는 스크립트 파일이 무엇인지 알 수 있습니다.
* 함수의 첫머리에 의존 관계가 선언되기 때문에 의존 관계를 찾아내고 이해하기 쉽습니다.
* dom 과 같은 지역 변수는 YAHOO 와 같은 전역 변수보다 언제나 더 빠르며 YAHOO.util.Dom 처럼 전역 변수의 중첩 프로퍼티와 비교하면 더 말할 것도 없습니다. 의존 관계 선언 패턴을 잘 지키면 함수 안에서 전역 객체 판별을 단 한번만 수행하고 이 다음부터는 지역 변수를 사용하기 때문에 훨씬 빠릅니다.

비공개 멤버

javascript

function add(num1, num2) {

var sum = num1 + num2;

return sum;

}

위 코드의 함수에는 num1, num2, sum이라는 세 가지의 고유변수가 있다. 이들 변수는 함수 내부에서는 접근이 가능하지만 함수 외부에서는 접근이 불가능하다.

비공개 멤버에 대한 별도의 문법은 없지만 클로저를 사용해서 구현할 수 있다.

생성자 함수 안에서 클로저를 만들면, 클로저 유효범위 안의 변수는 생성자 함수 외부에 노출되지 않지만 객체의 공개 메서드 안에서는 쓸 수 있다. 클로저를 함수 안에 만들어 사용하면 스코프 체인을 통해 이들 변수에 접근이 가능하게 된다.

즉, 생성자에서 객체를 반환할 때 객체의 메서드를 정의하면, 이 메서드안에서는 비공개 변수에 접근할 수 있는 것이다.

javascript

function Gadget(){

// 비공개 멤버(private member)

var name = 'iPod';

// 공개된 함수(privieged member)

this.getName = function(){

return name;

};

}

var toy = new Gadget();

// 'name'은 비공개이므로 undefined가 출력된다.

console.log(toy.name);

// 공개 메서드에서는 'name'에 접근할 수 있다.

console.log(toy.getName());

보시다시피 자바스크립에서도 쉽게 비공개 멤버를 구현할 수 있다.

비공개로 유지할 데이터를 함수로 감싸기만 하면 된다.

이 데이터들을 함수의 지역 변수로 만들면, 함수 외부에서는 접근할 수 없다.

특권(privileged) 메서드

위와 같은  코드 기법을 활용하여 함수 외부에서 고유변수에 접근이 가능하도록 공용 메소드를 만들 수가 있는데 이를 특권(privileged) 메소드라고 한다.

다시 말해, 특권 메소드는 고유변수/함수에 접근이 가능한 공용메소드인 것이다.

특권 메서드는 단지 비공개 멤버에 접근권한을 가진 공개 메서드를 가리키는 이름일 뿐이다.

앞선 예제에서는 getName()은 비공개 프로퍼티인 name에 '특별한' 접근 권한을 가지고 있기 때문에 특권 메서드라고 할 수 있다.

비공개 멤버의 허점

특권 메서드에서 비공개 변수의 값을 바로 반환할 경우 이 변수가 객체나 배열이라면 값이 아닌 참조가 반환되기 때문에, 외부 코드에서 비공개 변수 값을 수정할 수 있다.

javascript

function Gadget(){

// 비공개 멤버

var specs = {

screen\_width : 320,

screen\_height : 480,

color : "white"

};

// 공개 함수

this.getSpecs = function(){

return specs;

};

}

var toy = new Gadget(),

specs = toy.getSpecs();

specs.color = "black";

specs.price = "free";

console.log(toy.getSpecs());

//Object {screen\_width: 320, screen\_height: 480, color: "black", price: "free"}

얼핏 보기엔 별 문제 없어 보이나 여기서 getSpec() 메서드가 specs 객체에 대한 참조를 반환한다는게 문제다.

specs는 감춰진 비공개 멤버처럼 보이지만 Gadget 사용자에 의해 변경될 소지가 있다.

이와 같은 예기치 않은 문제를 해결하기 위해서는 비공개로 유지해야 하는 객체나 배열에 대한 참조를 전달할 때 주의를 기울여야 하며 주어진 객체의 최상위 프로퍼티만을 복사하는 extend() 함수와 모든 중첩 프로퍼티를 재귀적으로 복사하는 extendDeep() 함수로 해결할 수 있다.

객체에 특권 메소드를 만드는 방법은 여러가지가 있지만 이 포스팅에서는 모듈화,모듈 패턴화를 알아보면서 어떤 방식으로 특권 메소드를 사용하는지 알아보도록 하자.

모듈화(캡슐화)

javascript

// 전역 스코프(gloval scope)

// 변수(멤버 변수)

// var sayHi = "안녕";

// 함수 영역

/\*

var moduleFunc = function() {

// 함수 스코프(새로운 영역이 생성됨)

};

\*/

// 위의 멤버변수와 함수 영역을 하나의 코드블럭을 생성한다

// 즉, 위의 코드는 전역에 정의되어 있지만 멤버변수와 함수 영역을 전역 스코프와 상관없이

// 하나의 새로운 범위를 생성,관리하기위해 모듈화시킨다

// 새로운 모듈화(캡슐화)

// 자가 실행 함수

(function(){

var sayHi = "안녕";

var moduleFunc = function() {

return sayHi;

};

console.log(moduleFunc()); // 같은 스코프안에서 함수 호출함

}()); // 즉시 실행 함수

// 외부에서 호출

// console.log(sayHi); // Uncaught ReferenceError: sayHi is not defined

// console.log(moduleFunc()) // Uncaught ReferenceError: moduleFunc is not defined

모듈 패턴

더글라스 크록포드가 고안한 모듈 패턴은 싱글톤에서 같은 일을 한다.

싱글톤이란 인스턴스를 단 하나만 갖게 의도한 객체이다. 전통적으로 자바스크립트에서 싱글톤을 만들 때는 다음 코드와 같이 객체 리터럴 표기법을 사용한다.

javascript

var singleton = {

name: value,

method : function() {

// 메소드 코드

}

}

객체 리터럴과 비공개 멤버

생성자가 아닌 객체 리터럴로 비공개 멤버를 구현할 수 있다.

객체 리터럴에서는 익명 즉시 실행함수를 추가하여 클로저를 만든다.

모듈 패턴은 위 코드 형식에 따라 기본 싱글톤을 확장하여 고유멤버(private member)와 특권 메소드(privileged member)를 사용할 수가 있다.

javascript

var myobj; // 이 변수에 객체를 할당한다.

(function(){

// 비공개 멤버

var name = "my, oh my";

// 공개될 부분을 구현한다.

// var를 사용하지 않는다.

myobj = {

//특권 메서드

getName : function(){

return name;

}

};

}());

myobj.getName();

또는 다음과 같이 구현할 수 있다.

javascript

var myobj = (function(){

//비공개 멤버

var name = "my, oh my";

//공개될 부분을 구현한다.

return {

//특권 메서드

getName : function(){

return name;

}

};

}());

myobj.getName();

좀 더 코드를 작성하여 복습해 보자.

javascript

var myObj = function(){

// 고유 멤버(private member)가 될 스코프

var sayHi = "안녕하세요!!";

var intCnt = 0;

var hi = function() {

intCnt += 1; // hi 함수를 호출할 때마다 1씩 증가하도록 함

return sayHi;

};

var cnt = function() {

return intCnt; // closure를 통해서 증가된 intCnt값을 반환하도록 해준다.

}

// 특권/공용메소드와 프로퍼티 (공용 인터페이스)=>외부에 공개하기 위한 역할

// 객체 리터럴을 함수 값으로 반환(반환되는 객체리터럴에는 공용이 될 프로퍼티와 메소드만 소유)

return {

getHi : function() {

return sayHi; // 고유멤버에 접근이 가능(특권 메소드)

},

getHi2 : function() {

return '반갑습니다~~!!';

},

getHi3 : hi,

getCnt : intCnt, // intCnt를 직접 외부에 반환시켜주면 증가된 값이 출력되지 않는다.

getCnt2 : cnt

}

}();

// 모듈 패턴 외부에서 함수 호출이 가능한 것은 객체 리터럴을 반환값으로 넘겨줬기 때문이다

// 이렇게 외부에서 사용이 가능하다고 하여 공용 메소드라고 일컫는다.

console.log(myObj.getHi());

console.log(myObj.getHi2());

console.log(myObj.getHi3());

console.log(myObj.getCnt);

console.log(myObj.getCnt2());

console.log(myObj.getHi3());

console.log(myObj.getCnt2());

이렇게 모듈 패턴은 객체를 반환하는 익명 함수를 사용한다.

익명 함수 내부에서는 첫번째로 고유멤버인 함수의 매개변수,지역변수,함수 등을 정의하고 그 다음에는 객체 리터럴을 함수값으로 반환하는 것이다.

반환 된 객체 리터럴에는 공용적으로 사용될 프로퍼티나 메소드만 포함하게 되고 이 객체는 익명함수 내에서 정의되었기 때문에 공용메소드는 고유멤버에 접근하여 사용이 가능하다.

즉, 객체 리터럴이 싱글톤에 대한 공용 인터페이스를 정의하는 것이다.

프로토타입과 비공개 멤버

생성자를 사용하여 비공개 멤버를 만들 경우, 생성자를 호출하여 새로은 객체를 만들 때마다 비공개 멤버가 매번 재생성된다는 단점이 있다.

사실 생성자 내부에서 this 에 멤버를 추가하면 항상 이런 문제가 발생한다.

이러한 중복을 없애고 메모리를 절약하려면 공통 프로퍼티와 메서드를 생성자의 prototype 프로퍼티에 추가해야 합니다.

이렇게 하면 동일한 생성자로 생성한 모든 인스턴스가 공통된 부분을 공유하게 된다.

감춰진 비공개 멤버들도 모든 인스턴스가 함께 쓸 수 있다.

이를 위해서는 두 가지 패턴, 즉 생성자 함수 내부에 비공개 멤버를 만드는 패턴과 객체 리터럴로 비공개 멤버를 만드는 패턴을 함께 써야 한다.

왜냐하면 prototype 프로퍼티도 결국 객체라서 객체 리터럴로 생성할 수 있기 때문이다.

javascript

function Gadget() {

// 비공개 멤버

var name = 'iPod';

// 공개 함수

this.getName = function () {

return name;

}

}

Gadget.prototype = (function () {

// 비공개 멤버

var browser = 'Mobile Webkit';

// 공개된 프로토타입 멤버

return {

getBrowser : function () {

return browser;

}

}

})();

var toy = new Gadget();

console.log(toy.getName()); // 객체 인스턴스의 특권 메서드

console.log(toy.getBrowser()); // 프로토타입의 특권 메서드

비공개 함수를 공개 메서드로 노출시키는 방법

노출 패턴(revelation pattern) 은 비공개 메서드를 구현하면서 동시에 공개 메서드로도 노출하는 것을 말합니다.

객체의 모든 기능이 객체가 수행하는 작업에 필수불가결한 것들이라서 최대한 보호가 필요한데, 동시에 이 기능들의 유용성 때문에 공개적인 접근도 허용하고 싶은 경우가 있을 수 있습니다.

노출 패턴은 이런한 경우에 유용하게 쓸 수 있습니다.

메서드가 공개되어 있다는 것은 결국 이 메서드가 위험에 노출되어 있다는 말과도 같습니다.

공개 API 사용자가 어저면 본의 아니게 메서드를 수정할 수 있기 때문입니다.

ECMAScript 5에서는 객체를 고정(freeze)시킬 수 있는 선택자가 있지만, 이전 버전에서는 그렇지 않습니다.

이제 노출 패턴에 대해 알아보도록 하자.

이 용어는 크리스천 헤일먼(Christian Heilmann)이 만들어냈으며 처음에는 '모듈 노출 패턴(revealing module pattern)'이라고 했습니다.

먼저 예제를 살펴보도록 합니다.

이 예제는 객체 리터럴 안에서 비공개 멤버를 만드는 패턴에 기반하고 있습니다.

javascript

var arr;

(function () {

var astr = '[object Array]';

var toString = Object.prototype.toString;

function isArray(a) {

return toString.call(a) === astr;

}

function indexOf(haystack, needle) {

var i = 0,

max = haystack.length;

for (; i < max; i += 1) {

if (haystack[i] === needle ) {

return i;

}

}

return -1;

}

arr = {

isArray : isArray,

indexOf : indexOf,

inArray : indexOf

}

})();

여기에는 비공개 변수 두 개와 비공개 함수 두 개인 isArray() 와 indexOf() 가 존재하고 있습니다.

즉시 실행 함수의 마지막 부분을 보면, 공개적인 접근을 허용해도 괜찮겠다고 결정한 기능들이 arr 객체에 채울 수 있습니다.

비공개 함수 indexOf() 는 ECMAScript 5 식의 이름인 indexOf 와 PHP 에서 영향을 받은 이름이 inArray 라는 두 개의 이름으로 노출되어 있습니다.

위에서 정의한 코드를 바탕으로 새로운 arr 객체를 태스트해보도록 합니다.

javascript

arr.isArray([1, 2]); // true 가 기록

arr.isArray({0: 1}); // false 가 기록

arr.indexOf(['a', 'b', 's'], 's'); // 2 가 기록

arr.inArray(['a', 'b', 's'], 's'); // 2 가 기록

이제 공개된 메서드인 indexOf() 에 예기치 못한 일이 일어나더라도, 비공개 함수인 indexOf() 는 안전하게 보호되기 때문에 inArray() 는 계속해서 잘 동작할 것입니다.

javascript

arr.indexOf = null;

arr.inArray(['a', 'b', 'z', 'z']); // 2 가 기록

모듈 패턴 정리(Module Pattern)

모듈 패턴은 늘어나는 코드를 구조화하고 정리하는데 도움이 되기 때문에 널리 쓰입니다.

다른 언어와는 달리 자바스크립트에는 패키지를 위한 별도의 문법이 없습니다.

하지만 모듈 패턴을 사용하면 개별적인 코드를 느슨하게 결합시킬 수 있습니다.

따라서 각 기능들을 블랙박스처럼 다루면서도 소프트웨어 개발 중에(끊임없이 변하는) 요구사항에 따라 기능을 추가하거나 교체하거나 삭제하는 것도 자유롭게 할수 있습니다.

모듈 패턴은 지금까지 알아본 다음의 패턴들 여러 개를 조합한 것이다!!

* 네임스페이스 패턴
* 즉시 실행함수
* 비공개 멤버와 특권멤버
* 의존관계 선언

첫 단계는 네임스페이스를 설정하는 것입니다.

위 상단 글에서 알아본 namespace() 함수를 사용해, 유용한 배열 메서드를 제공하는 유틸리티 모듈 예제를 만들어 봅니다.

javascript

MYAPP.namespace('MYAPP.utilities.array');

그 다음 단계는 모듈을 정의하는 것입니다. 공개 여부를 제한해야 한다면 즉시 실행 함수를 사용해 비공개 유효범위를 만들면 됩니다.

즉시 실행 함수는 모듈이 될 객체를 반환합니다.

이 객체에는 모듈 사용자에게 제공할 공개 인터페이스가 담기게 될 것입니다.

javascript

MYAPP.utilities.array = function () {

return {

inArray : function (needle, haystack) {

// ...

},

isArray : function (a) {

// ...

}

}

}();

즉시 실행함수의 비공개 유효범위를 사용하면, 비공개 프로퍼티와 메서드를 마음껏 선언할 수 있습니다.

모듈에 의존 관계가 있다면 즉시 실행 함수 상단에서 정의합니다.

변수를 선언한 다음에는 필요에 따라 모듈을 초기화하는 데 필요한 일회성 초기화 코드를 두어도 좋습니다.

즉시 실행 함수가 반환하는 최종 결과는 모듈의 공개 API 담은 객체입니다.

javascript

MYAPP.namespace('MYAPP.utilities.array');

MYAPP.utilities.array = function () {

// 의존 관계 선언

var uobj = MYAPP.utilities.object,

ulang = MYAPP.utilities.lang;

// 비공개 프로퍼티

var array\_string = '[object Array]',

ops = Object.prototype.toString;

// 비공개 메서들

// ...

// 필요하다면 일회성 초기화 절차를 실행한다.

// ...

// 공개 API

return {

inArray : function (needle, haystack) {

for (var i = 0, max = haystack.length; i < max; i += 1) {

if (haystack[i] === needle) {

return true;

}

}

},

isArray : function (a) {

return ops.call(a) === array\_string;

}

// ... 더 필요한 메서드와 프로퍼티를 여기 추가할 수 있다.

}

}();

모듈 패턴은 특히 점점 늘어만 가는 코드를 정리할 때 널리 사용되면 매우 추천하는 방법입니다.

다음 포스팅에서는 계속해서 모듈 노출 패턴, 생성자를 생성하는 모듈, 그리고 샌드박스 패턴 등에 대해 알아보도록 하겠습니다.

### Related Info

1. [module pattern (모듈패턴) #1](http://webclub.tistory.com/5)

출처: <http://webclub.tistory.com/78> [Web Club]

## [모듈 패턴(기타) 및 샌드박스 패턴 #3](http://webclub.tistory.com/521)

10개월 전 by 하기실음 관두등가 재야재희

모듈 노출 패턴, 생성자를 생성하는 모듈 그리고 샌드박스 패턴

이 포스팅에서는 전 포스팅의 [모듈 패턴 #2](http://webclub.tistory.com/78) 에서 알아본 내용 이외의 패턴들에 대해 알아봅니다.

모듈 노출 패턴

전 포스팅에서 비공개 멤버와 관련된 패턴을 살펴보면서 이미 노출 패턴을 다룬 바 있습니다.

모듈 패턴도 비슷한 방식으로 작성할 수 있습니다.

즉, 모든 메서드를 비공개 상태로 유지하고, 최종적으로 공개 API 를 갖출 때 공개할 메서드만 골라서 노출하는 것입니다.

javascript

var MYAPP = MYAPP || {};

MYAPP.namespace = function (ns\_string) {

var parts = ns\_string.split('.'),

parent = MYAPP,

i;

// 처음에 중복되는 전역 객체명은 제거한다.

if (parts[0] === 'MYAPP') {

parts = parts.slice(1);

}

for (i = 0; i < parts.length; i += 1) {

if (typeof parent[parts[i]] === 'undefined') {

parent[parts[i]] = {};

}

parent = parent[parts[i]];

}

return parent;

};

MYAPP.namespace('MYAPP.utilities.array');

MYAPP.utilities.array = (function () {

// 비공개 프로퍼티

var arr\_string = '[object Array]',

ops = Object.prototype.toString;

// 비공개 메서드

var inArray = function (haystack, needle) {

for (var i = 0, max = haystack.length; i < max; i += 1) {

if (haystack[i] === needle) {

return i;

}

}

return -1;

};

var isArray = function (a) {

return ops.call(a) === arr\_string;

};

// 공개 API 노출

return {

isArray : isArray,

inArray : inArray

}

}());

console.log(MYAPP.utilities.array.inArray(['1',3], 3)); // 1 이 기록

console.log(MYAPP.utilities.array.isArray({'a': 1})); // false 가 기록

생성자를 생성하는 모듈

앞선 예제에서 MYAPP.utilities.array 라는 객체를 만들었습니다.

하지만 생성자 함수를 사용해 객체를 만드는 것이 더 편할 때도 있습니다.

모듈 패턴을 사용하면서도 이렇게 할 수 있습니다.

모듈을 감싼 즉시 실행 함수가 마지막에 객체가 아니라 함수를 반환하게 하면 됩니다.

다음 모듈 패턴 예제는 생성자 함수인 MYAPP.utilities.Array 를 반환합니다.

javascript

MYAPP.namespace('MYAPP.utilities.Array');

MYAPP.utilities.Array = (function(){

// 의존 관계 선언

var uobj = MYAPP.utilities.object,

ulang = MYAPP.utilities.lang,

// 비공개 프로퍼티와 메서드를 선언한 후 ...

Constr;

// var 선언을 마친다.

// 필요하다면 일회성 초기화 절차를 실행한다.

// ...

// 공개 API - 생성자 함수

Constr = function (o) {

this.elements = this.toArray(o);

};

// 공개 API - 프로토타입

Constr.prototype = {

consturctor : MYAPP.utilities.Array,

version : '2.0',

toArray : function (obj) {

for (var i = 0, a = [], len = obj.length; i < len; i += 1) {

a[i] = obj[i];

}

return a;

}

};

// 생성자 함수를 반환한다.

// 이 함수가 새로운 네임스페이스에 할당될 것이다.

return Constr;

}());

// 위 생성자 함수는 다음과 같이 사용할 수 있다.

var arr = new MYAPP.utilities.Array({});

console.log(arr.version);

모듈에 전역변수 가져오기

이 패턴의 흔한 변형 패턴으로는 모듈을 감싼 즉시 실행 함수에 인자를 전달하는 형태가 있습니다.

어떠한 값이라도 가능하지만 보통 전역 변수에 대한 참조 또는 전역 객체 자체를 전달합니다.

이렇게 전역 변수를 전달하면 즉시 실행 함수 내에서 지역 변수로 사용할 수 있게 되기 때문에 탐색 작업이 좀더 빨리집니다.

javascript

MYAPP.utilities.module = (function (app, global) {

// 전역 객체에 대한 참조와

// 전역 애플리케이션 네임스페이스 객체에 대한 참조가 지역 변수화 된다.

}(MYAPP, this));

샌드박스 패턴

샌드박스 패턴은 네임스페이스 패턴의 다음과 같은 단점을 해결합니다.

* 애플리케이션 전역 객체가 단 하나의 전역 변수에 의존한다. 따라서 네임스페이스 패턴으로는 동일한 애플리케이션이나 라이브러리의 두 가지 버전을 한 페이지에서 실행시키는 것이 불가능하다. 여러 버전들이 모두 이를테면 MYAPP 이 라는 동일한 전역 변수명을 쓰기 때문이다.
* MYAPP.utilities.array 와 같이 점으로 연결된 긴 이름을 써야 하고 런타임에는 탐색 작업을 거쳐야한다.

이름을 보고 짐작할 수 있듯이 샌드박스 패턴은 어떤 모듈이 다른 모듈과 그 모듈의 샌드박스에 영향을 미치지 않고 동작할 수 있는 환경을 제공하는 것입니다.

전역 생성자

네임스페이스 패턴에서는 전역 객체가 하나입니다.

샌드박스 패턴의 유일한 전역은 생성자입니다.

이것을 Sandbox() 라고 해보자. 이 생성자를 통해 객체들을 생성할 것입니다.

그리고 이 생성자에 콜백 함수를 전달해 해당 코드를 샌드박스 내부 환경으로 격리시킬 것입니다.

샌드박스 사용법은 다음과 같습니다.

javascript

new Sandbox(function (box) {

// 여기에 코드가 들어간다...

})

box 객체는 네임스페이스 패턴에서의 MYAPP 과 같은 것입니다.

코드가 동작하는데 필요한 모든 라이브러리 기능들이 여기에 들어갑니다.

이 패턴에 두 가지를 추가해 보도록 하자.

* new 를 강제하는 패턴을 활용하여 객체를 생성할 때 new 를 쓰지 않아도 되게 만든다.
* Sandbox() 생성자가 선택적인 인자를 하나 이상 받을 수 있게 한다. 이 인자들은 객체를 생성하는 데 필요한 모듈의 이름을 지정한다. 우리는 코드의 모듈화를 지향하고 있으므로 Sandbox() 가 제공하는 기능 대부분이 실제로는 모듈 안에 담겨지게 될 것이다.

이제 객체를 초기화하는 코드가 어떤 모습인지 예제를 보도록 합니다.

다음과 같이 new 를 쓰지 않고도, 가상의 모듈 'ajax' 와 'event' 를 사용하는 객체를 만들 수 있습니다.

javascript

Sandbox(['ajax','event'], function (box) {

// console.log(box);

})

다음 예제는 앞선 예제와 비슷하지만 모듈 이름을 개별적인 인자로 전달합니다.

javascript

Sandbox('ajax','dom', function (box) {

// console.log(box);

})

"쓸 수 있는 모듈을 모두 사용한다" 는 의미로 와일드카드 \* 인자를 사용하면 어떨까?

편의를 위해 모듈명을 누락시키면 샌드박스가 자동으로 \* 를 가정하도록 해보자.

그렇다면 모든 모듈을 사용하는 방법으로 다음 두 가지가 가능하게 될 것입니다.

javascript

Sandbox('\*', function (box) {

// console.log(box);

});

Sandbox(function (box) {

// console.log(box);

});

마지막으로 샌드박스 객체의 인스턴스를 여러 개 만드는 예제를 살펴보도록 하자.

심지어 한 인스턴스 내부에 다른 인스턴스를 중첩시킬 수도 있습니다.

이 때도 두 인스턴스 간의 간섭 현상은 일어나지 않습니다.

javascript

Sandbox('dom', 'event', function (box) {

// dom 과 event 를 가지고 작업하는 코드

Sandbox('ajax', function () {

// 샌드박스된 box 객체를 또 하나 만든다.

// 이 "box" 객체는 바깥쪽 함수의 "box" 객체와는 다르다.

// ajax 를 사용하는 작업 완료

});

// 더 이상 ajax 모듈의 흔적은 찾아볼 수 없다.

});

이 예제들에서 볼 수 있듯이, 샌드박스 패턴을 사용하면 콜백 함수로 코드를 감싸기 때문에 전역 네임스페이스를 보호할 수 있습니다.

필요하다면 함수가 곧 객체라는 사실을 활용하여 Sandbox() 생성자의 "스태틱" 프로퍼티에 데이터를 저장할 수도 있습니다.

또 원하는 유형별로 모듈의 인스턴스를 여러 개 만들 수도 있습니다.

이 인스턴스들은 각각 독립적으로 동작하게 됩니다.

그럼 이제 이 모든 기능을 지원하는 Sandbox() 생성자와 그 모듈을 구현하는 방법을 살펴보도록 하자.

모듈 추가하기

실제 생성자를 구현하기 전에 모듈을 어떻게 추가할 수 있는지부터 살펴보도록 합니다.

Sandbox() 생성자 함수 역시 객체이므로 modules 이라는 프로퍼티를 추가할 수 있습니다.

이 프로퍼티는 키-값의 쌍을 담은 객체로, 모듈의 이름이 되고 각 모듈을 구현한 함수가 값이 되도록 할 것입니다.

javascript

Sandbox.modules = {};

Sandbox.modules.dom = function (box) {

box.getElement = function () {

};

box.getStyle = function () {

};

box.foo = 'bar';

};

Sandbox.modules.event = function (box) {

// 필요에 따라 다음과 같이 Sandbox 프로토타입에 접근할 수 있다.

// box.constructor.prototype.m = 'mmm';

box.attachEvent = function () {

};

box.detachEvent = function () {

};

};

Sandbox.modules.ajax = function (box) {

box.makeRequest = function () {

};

box.getResponse = function () {

};

};

위 예제에서는 dom, event, ajax 라는 모듈을 추가했습니다.

모든 라이브러리와 복잡한 웹 애플리케이션에서 흔히 사용되는 기능들입니다.

각 모듈을 구현하는 함수들이 현재의 인스턴스 box 를 인자로 받아들인 다음 이 인스턴스에 프로퍼티와 메서드를 추가하게 됩니다.

생성자 구현

이제 Sandbox() 생성자를 구현해 보도록 합니다.(이 생성자의 이름은 여러분의 라이브러리나 애플리케이션에 맞게 바꾸어도 좋다.)

javascript

function Sandbox() {

// arguments 를 배열로 바꾼다.

var args = Array.prototype.slice.call(arguments);

// 마지막 인자는 콜백 함수다.

var callback = args.pop();

// 모듈은 배열로 전달될 수도 있고, 개별 인자로 전달될 수도 있다.

var modules = (args[0] && typeof args[0] === 'string') ? args : args[0];

var i;

// 함수가 생성자로 호출되도록 보장한다.(new 를 강제하는 패턴)

if ((!this instanceof Sandbox)) {

return new Sandbox(modules, callback);

}

// this 에 필요한 프로퍼티를 추가한다.

this.a = 1;

this.b = 2;

// 코어 'this' 객체에 모듈을 추가한다.

// 모듈이 없거나 '\*' 이면 사용 가능한 모든 모듈을 사용한다는 의미이다.

if (!modules || modules === '\*' || modules[0] === '\*') {

modules = [];

for (i in Sandbox.modules) {

if (Sandbox.modules.hasOwnProperty(i)) {

modules.push(i);

}

}

}

// 필요한 모듈들을 초기화한다.

for (i = 0; i < modules.length; i += 1) {

Sandbox.modules[modules[i]](this);

}

// 콜백 함수를 호출한다.

callback(this);

}

// 필요한 프로토타입 프로퍼티들을 추가한다.

Sandbox.prototype = {

name: 'My Application',

version : '1.0',

getName : function () {

return this.name;

}

};

위 구현에서 핵심적인 사항들은 다음과 같습니다.

* this 가 Sandbox 의 인스턴스인지 확인하고, 그렇지 않으면 (즉, Sandbox() 가 new 없이 호출되었다면) 함수를 생성자로 호출한다.
* 생성자 내부에서 this 에 프로퍼티를 추가한다. 생성자의 프로토타입에도 프로퍼티를 추가할 수 있다.
* 필요한 모듈은 배열로도, 개별적인 인자로도 전달할 수 있고, \* 와일드카드를 사용하거나, 쓸 수 있는 모듈을 모드 쓰겠다는 의미로 생략할 수도 있다. 이 예제에서는 필요한 기능을 다른 파일로부터 로딩하는 것까지는 구현하지 않았지만, 이러한 선택지도 확실히 고려해보아야 한다.
* 필요한 모듈을 모두 파악한 다음에는 각 모듈을 초기화한다. 다시 말해 각 모듈을 구현한 함수를 호출한다.
* 생성자의 마지막 인자는 콜백 함수이다. 이 콜백 함수는 맨 마지막에 호출되며, 새로 생성된 인스턴스가 인자로 전달된다. 이 콜백 함수가 실제 사용자의 샌드박스이며 필요한 기능을 모두 갖춘 상태에서 box 객체를 전달받게 된다.

출처: <http://webclub.tistory.com/521> [Web Club]

## [Static member of JavaScript](http://webclub.tistory.com/526)

9개월 전 by 하기실음 관두등가 재야재희

스태틱 멤버

스태틱 프로퍼티와 메서드란 인스턴스에 따라 달라지지 않는 프로퍼티와 메서드 를 말합니다.

클래스 기반 언어에서는 별도의 문법을 통해 스택틱 멤버를 생성하여 클래스 자체의 멤버인 것처럼 사용할 수 있습니다.

예를 들어 MathUtils 클래스에 max() 라는 스태틱 메서드가 있다면MathUtils.max(2,5) 와 같은 식으로 호출할 수 있습니다.

이것은 공개 스태틱 멤버의 예로, 클래스의 인스턴스를 생성하지 않고도 사용할 수 있습니다.

비공개 스태틱 멤버는 클래스 사용자에게는 보이지 않지만 클래스의 인스턴스들은 모두 함께 사용할 수 있습니다.

그럼 자바스크립트에서 공개와 비공개 스태틱 멤버를 구현하는 방법을 살펴보도록 합니다.



공개 스태틱 멤버

자바스크립트에는 스태틱 멤버를 표기하는 별도의 문법이 존재하지 않습니다.

그러나 생성자에 프로퍼티를 추가함으로써 클래스 기반 언어와 동일한 문법을 사용할 수 있습니다.

생성자도 다른 함수와 마찬가지로 객체이고 그 자신의 프로퍼티를 가질 수 있기 때문에 이러한 구현이 가능합니다.

다음 예제는 Gadget 이라는 생성자에 스태틱 메서드인 isShiny() 와 일반적인 인스턴스 메서드인 setPrice() 를 정의한 것입니다.

isShiny() 는 특정 Gadget 객체를 필요로 하지 않기 때문에 스태틱 메서드라 할 수 있습니다.

모든 Gadget 이 빛나는지 알아내는 데는 특정한 하나의 Gadget 이 필요하지 않은 것과 같습니다.

반면 개별 Gadget 들의 가격은 다를 수 있기 때문에 setPrice() 메서드를 쓰려면 객체가 필요합니다.

javascript

// 생성자

var Gadget = function () { };

// 스태틱 메서드

Gadget.isShiny = function () {

return 'you bet.'

};

// 프로토타입에 일반적인 함수를 추가

Gadget.prototype.setPrice = function (price) {

this.price = price;

};

이제 이 메서드를 호출해 보도록 합니다.

스태틱 메서드인 isShiny() 는 생성자를 통해 직접 호출되지만, 일반적인 메서드는 인스턴스를 통해 호출됩니다.

javascript

// 스태틱 메서드를 호출하는 방법

console.log(Gadget.isShiny()); // you bet

// 인스턴스를 생성한 후 메서드를 호출하기

var iphone = new Gadget();

iphone.setPrice(500);

인스턴스 메서드를 스태틱 메서드와 같은 방법으로 호출하면 동작하지 않습니다.

스태틱 메서드 역시 인스턴스 iphone 객체를 이용해 호출하면 동작하지 않습니다.

javascript

console.log(typeof Gadget.setPrice); // undefined

console.log(typeof iphone.isShiny); // undefined

스태틱 메서드가 인스턴스를 통해 호출했을 때도 동작한다면 편리한 경우가 있을 수 있습니다.

이 경우에는 간단하게 프로토타입에 새로운 메서드를 추가하는 것만으로 쉽게 구현할 수 있습니다.

이 새로운 메서드는 원래의 스태틱 메서드를 가리키는 일종의 퍼사드(facade) 역할을 합니다.

javascript

Gadget.prototype.isShiny = Gadget.isShiny;

console.log(iphone.isShiny());

이런 경우에는 스태틱 메서드 안에서 this 를 사용할 때 주의를 기울여야 합니다.

Gadget.isShiny() 호출했을 때 내부의 this 는 Gadget 생성자를 가리키지만, iphone.isShiny() 를 호출했을 때는 this 는 생성자 함수로 생성된 객체인 iphone 을 가리키게 됩니다.

마지막으로 스태틱한 방법으로도, 스태틱하지 않은 방법으로도 호출될 수 있는 어떤 메서드를 호출 방식에 따라 살짝 다르게 동작하게 하는 예제를 살펴보도록 합니다.

메서드가 어떻게 호출되었는지 판별하기 위해서 instanceof 연산자를 활용해 봅니다.

javascript

// 생성자

var Gadget = function (price) {

this.price = price;

};

// 스태틱 메서드

Gadget.isShiny = function () {

// 다음은 항상 동작한다.

var msg = 'you bet';

if (this instanceof Gadget) {

// 다음은 스태틱하지 않은 방식으로 호출되었을 때만 동작하도록 한다.(인스턴스 멤버를 가리킴)

msg += ', it costs $' + this.price + '!!';

}

return msg;

};

// 프로토타입에 일반적인 메서드를 추가한다

Gadget.prototype.isShiny = function () {

return Gadget.isShiny.call(this); // 이 this 는 생성자 함수로 생성된 인스턴스 객체를 가리키게 된다.

};

스태틱 메서드와 인스턴스를 통해 스태틱하지 않은 방법으로 호출해 보면 다음과 같은 결과가 나타납니다.

javascript

// 스태틱 메서드를 호출

console.log(Gadget.isShiny()); // you bet 이 기록된다.

// 인스턴스를 통해 스태틱하지 않은 방법으로 호출

var iphone = new Gadget(345.99);

console.log(iphone.isShiny()); // you bet, it costs $345.99!! 가 기록된다.

비공개 스태틱 멤버

지금까지는 공개 스태틱 멤버를 살펴보았습니다.

이번에는 비공개 스태틱 멤버를 구현하는 방법을 알아봅니다.

비공개 스태틱 멤버란 다음과 같은 의미를 가지고 있습니다.

* 동일한 생성자 함수로 생성된 객체들이 공유하는 멤버입니다.
* 생성자 외부에서는 접근할 수 없습니다.

Gadget 생성자 안에 counter 라는 비공개 스태틱 프로퍼티를 구현하는 예제를 살펴보도록 합니다.

비공개 프로퍼티는 먼저 클로저 함수를 만들고, 비공개 멤버를 이 함수로 감싼 후 이 함수를 즉시 실행한 결과로 새로운 함수를 반환하게 됩니다.

반환되는 함수는 Gadget 변수에 할당되어 새로운 생성자가 될 것입니다.

javascript

var Gadget = function () {

// 스태틱 변수/프로퍼티

var counter = 0;

// 생성자의 새로운 구현 버전을 반환한다.

return function () {

console.log(counter += 1);

};

}(); // 즉시 실행한다.

새로운 Gadget 생성자는 단순히 비공개 counter 값을 증가시켜 출력합니다.

몇 개의 인스턴스를 만들어 테스트해보면 실제로 모든 인스턴스가 동일한 counter 값을 공유하고 있음을 확인할 수 있습니다.

javascript

var g1 = new Gadget(); // 1 이 기록된다

var g2 = new Gadget(); // 2 이 기록된다

var g3 = new Gadget(); // 3 이 기록된다

객체당 1씩 counter 를 증가시키고 있기 때문에 이 스태틱 프로퍼티는 Gadget 생성자를 통해 생성된 개별 객체의 유일성을 식별하는 ID 가 될 수 있다.

유일한 식별자는 쓸모가 많으니 특권 메서드로 노출시켜도 좋지 않을까?

앞선 예제에 덧붙여 비공개 스태틱 프로퍼티에 접근할 수 있는 getLastId() 라는 특권 메서드를 추가해보도록 해봅니다.

javascript

// 생성자

var Gadget = function () {

// 스태틱 변수/프로퍼티

var counter = 0,

NewGadget;

// 이 부분이 생성자를 새롭게 구현한 부분이다.

NewGadget = function () {

counter += 1;

};

// 특권 메서드

NewGadget.prototype.getLastId = function () {

return counter;

};

// Gadgt 생성자를 덮어쓴다.

return NewGadget;

}(); // 즉시 실행한다.

새로운 버전을 아래에서 테스트 해봅니다.

javascript

var iphone = new Gadget();

console.log(iphone.getLastId()); // 1 이 기록

var ipod = new Gadget();

console.log(ipod.getLastId()); // 2 이 기록

var ipad = new Gadget();

console.log(ipad.getLastId()); // 3 이 기록

공개/비공개 스태틱 프로퍼티는 상당히 편리합니다.

특정 인스턴스에 한정되지 않는 메서드와 데이터를 담을 수 있고 인스턴스별로 매번 재생성되지도 않습니다.

출처: <http://webclub.tistory.com/526> [Web Club]

[**Chaining Pattern of JavaScript**](http://webclub.tistory.com/528)

9개월 전 by 하기실음 관두등가 재야재희

체이닝패턴

체이닝 패턴이란 객체에 연쇄적으로 메서드를 호출할 수 있도록 하는 패턴입니다.

여러 가지 동작을 수행할 때, 먼저 수행한 동작의 반환 값을 변수에 할당한 후 다음 작업을 할 필요가 없기 때문에 호출을 여러 줄에 걸쳐 쪼개지 않아도 됩니다.

javascript

myobj.method1('hello').method2().method3('world').method4();

만약 메서드에 의미있는 반환 값이 존재하지 않는다면, 현재 작업중인 객체 인스턴스인 this 를 반환하게 됩니다.

이렇게 하면 객체의 사용자는 앞선 메서드에 이어 다음 메서드를 바로 호출할 수 있습니다.

javascript

var obj = {

value : 1,

increment : function () {

this.value += 1;

return this;

},

add : function (v) {

this.value += v;

return this;

},

shout : function () {

console.log(this.value);

}

};

// 메서드 체이닝 호출

obj.increment().add(2).shout(); // 4 가 기록

// 위와 달리 메서드를 하나씩 호출하려면 다음과 같이 해야한다.

obj.increment();

obj.add(3);

obj.shout(); // 8 가 기록



체이닝 패턴의 장단점

체이닝 패턴을 사용하면 코드량이 줄고 코드가 좀더 간결해져 거의 하나의 문장처럼 읽히게 할 수 있다는 장점이 있습니다.

또 체이닝 패턴을 통해서 함수를 쪼개는 방법을 생각하게 되고, 혼자서 너무 많은 일을 처리하려는 함수보다는 좀더 작고 특화된 함수를 만들게 됩니다.

장기적으로는 이런 방법을 통해 유지보수가 개선됩니다.

그러나 이렇게 작성된 코드는 디버깅하기가 어렵습니다.

코드의 어느 라인에서 에러가 발생했는지 알아내더라도, 그 라인에서 수행하는 일이 너무 많을 수 있기 때문입니다.

여러 개의 메서드 중 하나가 실패해 버린다면, 실패한 메서드가 어느 것인지 알아내기기 어렵습니다.

『Clean Code』 의 저자 로버트 마틴은 이러한 상황을 '열차 사고' 패턴이라 불렀습니다.

어쨋거나 이 패턴을 알아두면 도움이 됩니다.

어떤 메서드가 명백히 의미있는 반환값을 가지지 않는다면 항상 this 를 반환하게 하는 것입니다.

이 패턴은 jQuery 라이브러리 등에서 널리 사용되고 있습니다.

DOM API 를 들여다보면, DOM 의 요소들도 체이닝 패턴을 사용하는 경향이 있음을 알 수 있습니다.

출처: <http://webclub.tistory.com/528> [Web Club]