3주차 스터디

Ch. 19 - 26

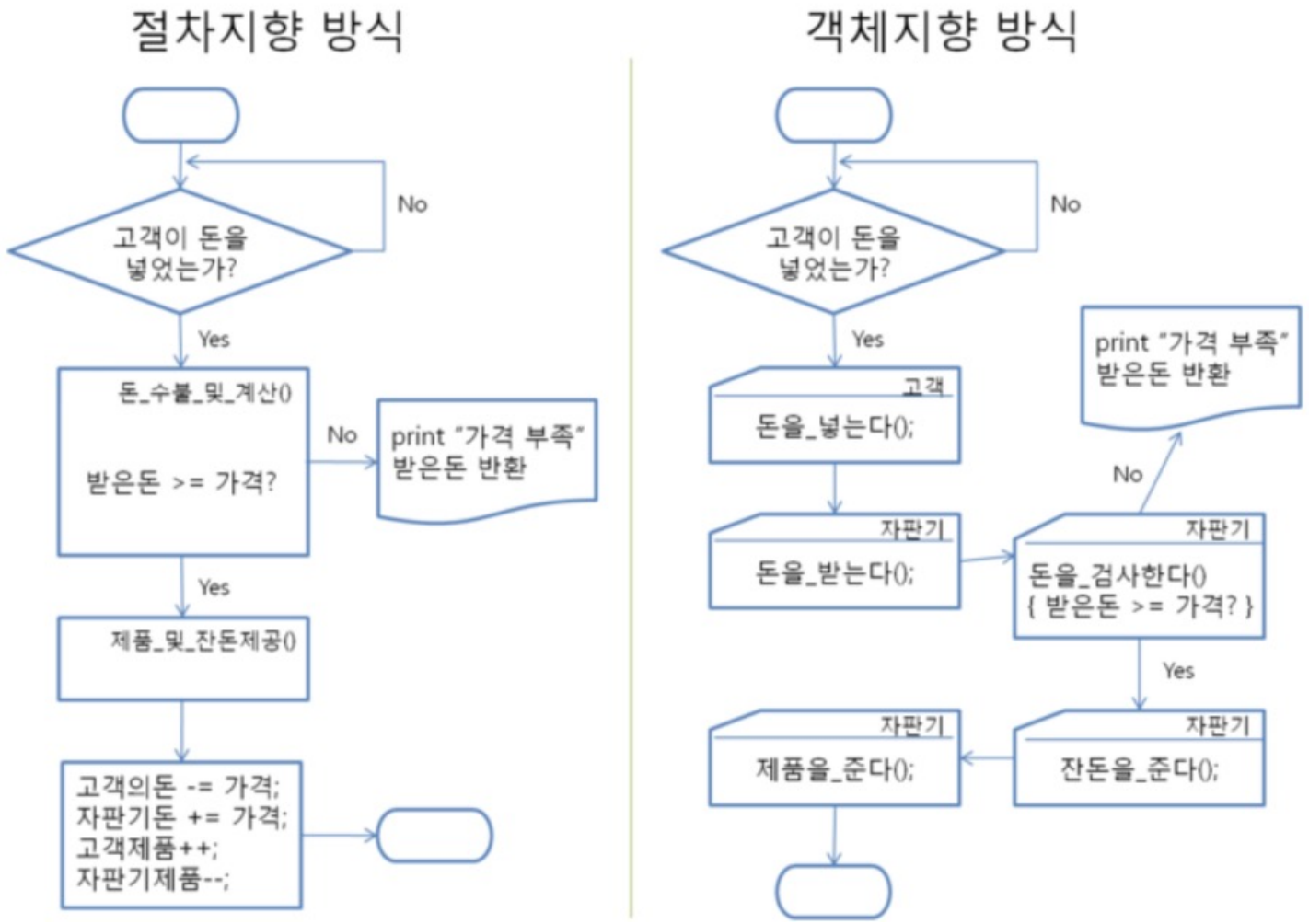
Ch.19
자바스크립트 객체지향 프로그래밍
(Object Oriented Programming)

#1. 객체지향 프로그래밍이란?

- **절차적 프로그래밍 [Procedural Programming]**
  - 순차적 처리를 중요시하여 프로그램 전체가 유기적으로 연결되도록 만드는 기법
  - 👍 : 실행속도 빠름
  - 👎🏻 : 코드순서 바꾸면 결과값 보장 못함. 대형 프로젝트에 부적합. 유지보수 어려움.
  - Ex. C
- **객체지향 프로그래밍 [Object-Oriented Programming]**
  - 프로그래밍에서 필요한 데이터를 추상화시켜 상태와 행위를 가진 객체로 만들고, 객체들 간의 상호작용을 통해 로직을 구현하는 기법
  - 👍 : 유연하고 유지보수 쉬움. 확장성 좋음. 코드 재사용 용이.
  - 👎🏻 : 처리속도 상대적으로 느림. 설계에 많은 시간 필요.
  - Ex. Java, C++, Javascript

✓ 이 둘은 반대의 개념이 아님.
단지 절차지향은 '순차적 실행', 객체지향은 '관계 및 조직'에 초점을 맞추고 있다는 차이점만 존재.

#1. 객체지향 프로그래밍이란?

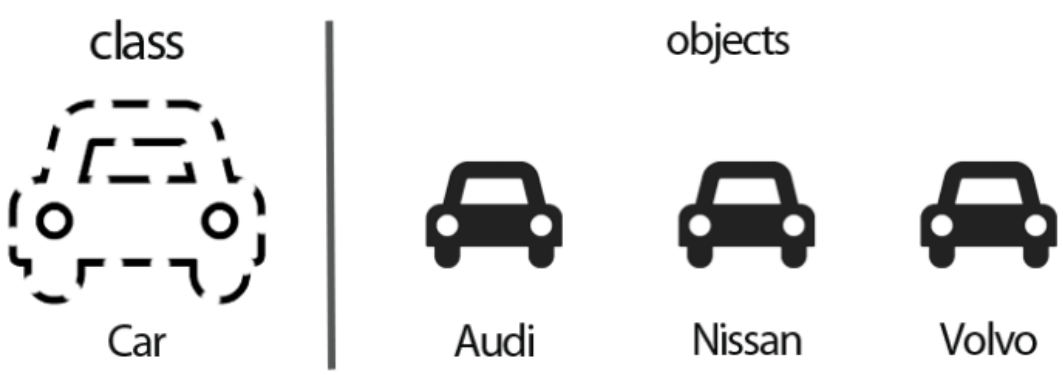


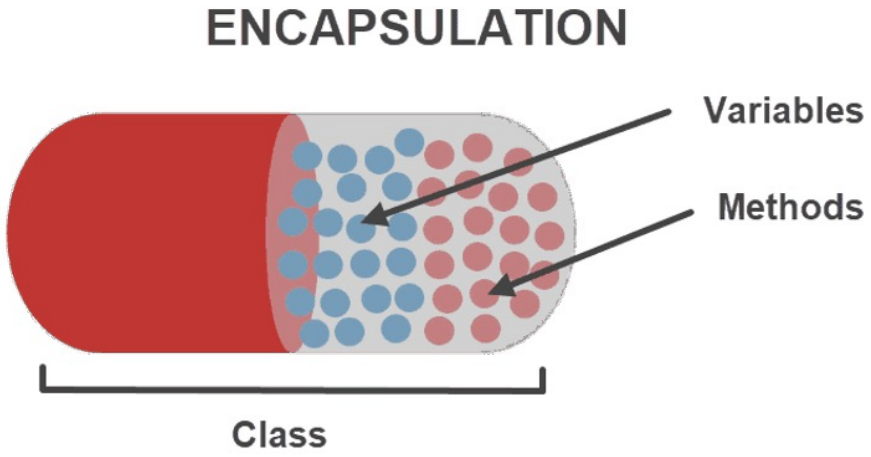
데이터 중심으로 함수 구현

기능 중심으로 메서드 구현

#1. 객체지향 프로그래밍이란?

- 객체지향 프로그래밍의 특징
  1. 추상화 - 객체에서 공통의 속성과 행위를 추출하는 것
  2. 캡슐화 - 변수와 함수를 하나의 클래스로 묶는 것
  3. 상속 - 클래스의 속성/행위를 하위 클래스에 물려주는 것
  4. 다형성 - 하나의 변수명/함수명이 상황에 따라 다른 의미로 해석 가능한 것





#2. 클래스 기반 vs 프로토타입 기반

- **클래스 기반 언어** [Java, C++, C#, Python, PHP …]
  - 클래스로 객체의 자료구조/기능 정의, 생성자를 통해 인스턴스 생성
  - 모든 인스턴스는 클래스에서 정의된 범위 내에서만 작동. 구조변경 불가.

- **프로토타입 기반 언어** [Javascript …]
  - 클래스 개념 없이 별도의 객체 생성 방법이 있다(3가지)
  - 이미 생성된 인스턴스의 자료구조/기능은 동적으로 변경 가능.
  - JS에는 클래스가 없으나, 함수 객체로 클래스/생성자/메소드 등을 구현할 수 있다.
    객체지향의 상속, 캡슐화 등의 개념은 프로토타입 체인과 클로저로 구현할 수 있다.

cf. 자바스크립트 객체 생성 방법 [3가지]

```
// 객체 리터럴
var obj1 = {};
obj1.name = 'Lee';

// Object() 생성자 함수
var obj2 = new Object();
obj2.name = 'Lee';

// 생성자 함수
function F(){};
var obj3 = new F();
obj3.name = 'Lee';
```

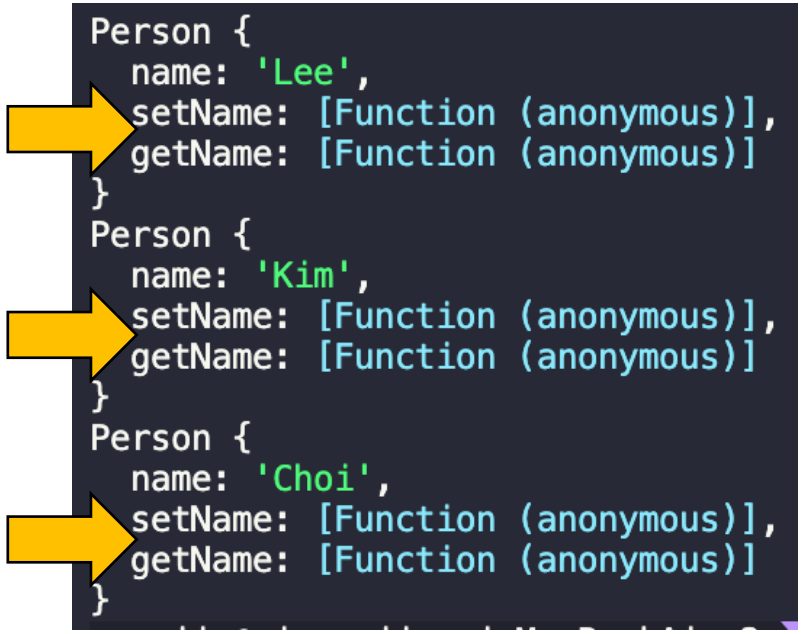
#3. 생성자 함수와 인스턴스의 생성

- Javascript는 생성자함수와 new연산자의 사용을 통해 인스턴스를 생성할 수 있다.
- 이때, 생성자함수는 클래스이자 생성자의 역할을 한다.


```
// 생성자 함수(Constructor)
function Person(name){
    // 프로퍼티
    this.name = name;

    // 메소드
    this.setName = function(name){
        this.name = name;
    };

    // 메소드
    this.getName = function(){
        return this.name;
    };
}


//인스턴스의 생성
var me = new Person('Lee');
console.log(me.getName()); // Lee

//메소드 호출
me.setName('Kim');
console.log(me.getName()); // Kim
```

#3. 생성자 함수와 인스턴스의 생성

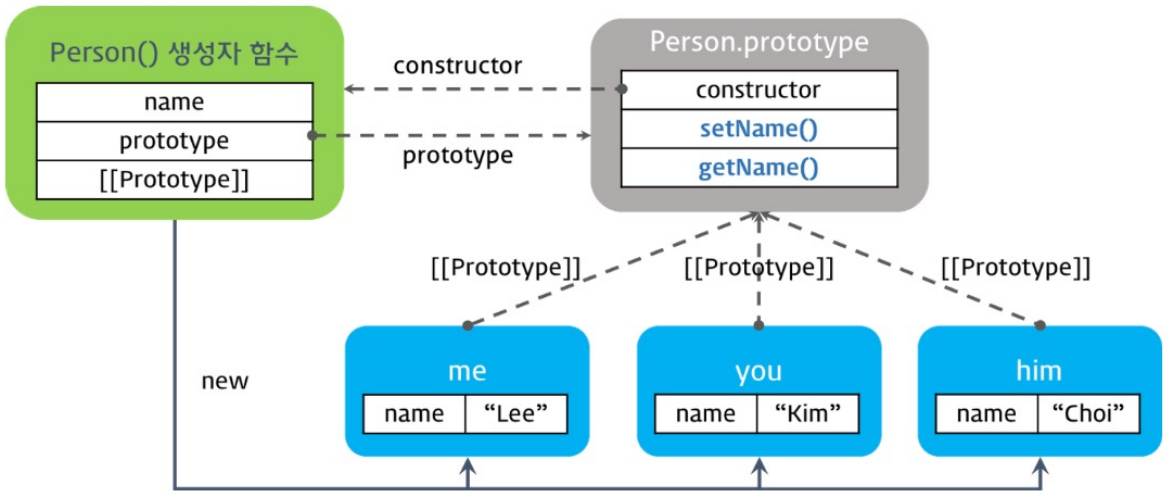
```
var per1 = new Person('Lee');
var per2 = new Person('Kim');
var per3 = new Person('Choi');

console.log(per1);
console.log(per2);
console.log(per3);
```



각 인스턴스에 내용이 동일한 메소드 setName과 getName이 중복 생성된다.
메모리 낭비 !!
이는 프로토타입으로 해결이 가능하다.

#4. 프로토타입 체인과 메소드의 정의

생성자 함수 내부의 메소드를 생성자함수의 prototype 프로퍼티가 가리키는 프로토타입 객체로 이동시키면,
생성자 함수에 의해 생성되는 모든 인스턴스는 *프로토타입 체인*을 통해 프로토타입 객체의 메소드를 참조할 수 있다.



cf. prototype : 내가 원형일 때 존재. 함수 객체만 가짐.
__proto__ : 나의 원형을 가리킴. 모든 객체가 가짐.

```
function Person(name){
    this.name = name;
}

//프로토타입 객체에 메소드 정의
Person.prototype.setName = function(name){
    this.name = name;
}
//프로토타입 객체에 메소드 정의
Person.prototype.getName = function(){
    return this.name;
};

var per1 = new Person('Lee');
var per2 = new Person('Kim');
var per3 = new Person('Choi');

console.log(Person.prototype);
// Person { setName: [Function], getName: [Function] }

console.log(per1); // Person { name: 'Lee' }
console.log(per2); // Person { name: 'Kim' }
console.log(per3); // Person { name: 'choi' }
```

#5. 상속

- Java같은 클래스 기반 언어에서 상속은 코드 재사용의 관점에서 매우 유용하다.
  객체는 클래스의 인스턴스이며, 클래스는 다른 클래스로 상속 가능하다.
- 클래스 개념이 없는 Javascript는 프로토타입을 이용해 상속을 구현하는데, 2가지 방법이 있다.

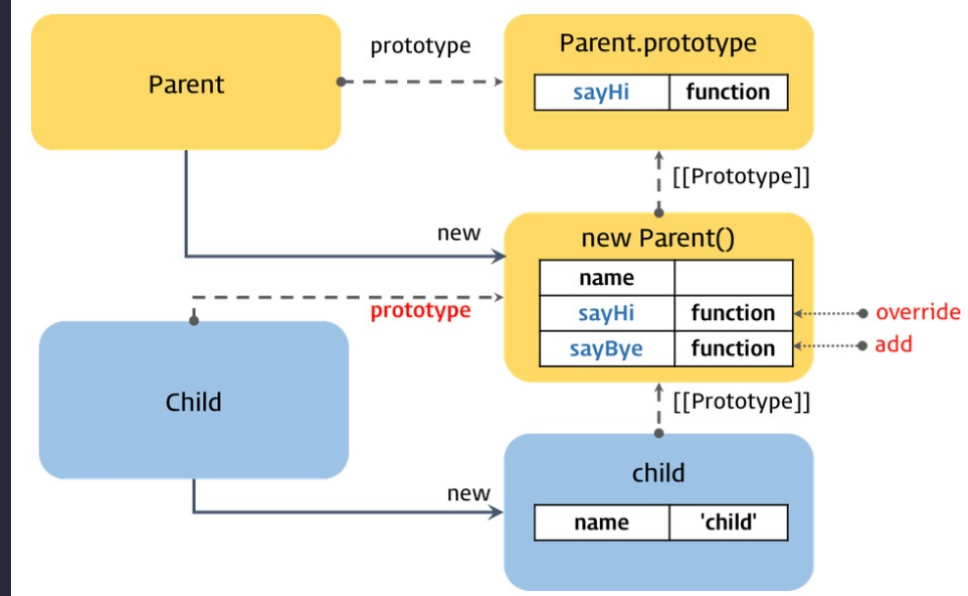
  1. 의사 클래스 패턴 상속 [Pseudo-classical Inheritance]

  2. 프로토타입 패턴 상속 [Prototypal Inheritance]

#5. 상속_의사 클래스 패턴 상속

- 클래스 기반 언어의 상속 방식을 흉내내는 방식.
- 자식 생성자 함수의 prototype 프로퍼티를 부모 생성자 함수의 인스턴스로 교체하여 상속 구현.
  따라서 부모와 자식 모두 생성자 함수 정의 필요.
- 문제점
  1. new 연산자를 통한 인스턴스의 생성
  2. 생성자 링크의 파괴
  3. 객체리터럴 패턴에는 부적합

```js
// 부모 생성자 함수
var Parent = (function(){
    // Constructor
    function Parent(name) {
        this.name = name;
    }
    // Method
    Parent.prototype.sayHi = function(){
        console.log('Hi! ' + this.name);
    };
    // return constructor
    return Parent;
}());

// 자식 생성자 함수
var Child = (function(){
    // Constructor
    function Child(name){
        this.name = name;
    }
    // 자식 생성자 함수의 프로토타입 객체를 부모 생성자 함수의 인스턴스로 교체
    Child.prototype = new Parent(); // ②
    // 메소드 오버라이트
    Child.prototype.sayHi = function(){
        console.log('안녕하세요! ' + this.name);
    };
    // sayBye 메소드는 Parent 생성자함수의 인스턴스에 위치된다
    Child.prototype.sayBye = function(){
        console.log('안녕히가세요! ' + this.name);
    };
    // return constructor
    return Child;
}());

var child = new Child('child'); // ①
console.log(child);  // Parent { name: 'child' }

console.log(Child.prototype); // Parent { name: undefined, sayHi: [Function], sayBye: [Function] }

child.sayHi();  // 안녕하세요! child
child.sayBye(); // 안녕히가세요! child

console.log(child instanceof Parent); // true
console.log(child instanceof Child);  // true
```

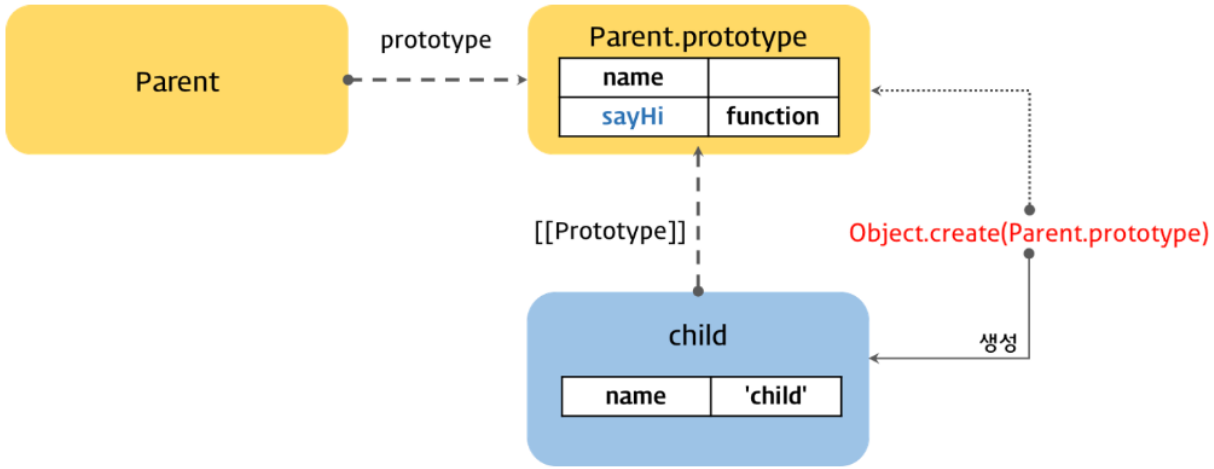


Child 생성자함수가 생성한 인스턴스 child(1)의 프로토타입 객체
= Parent 생성자함수가 생성한 인스턴스(2)

#5. 상속_프로토타입 패턴 상속

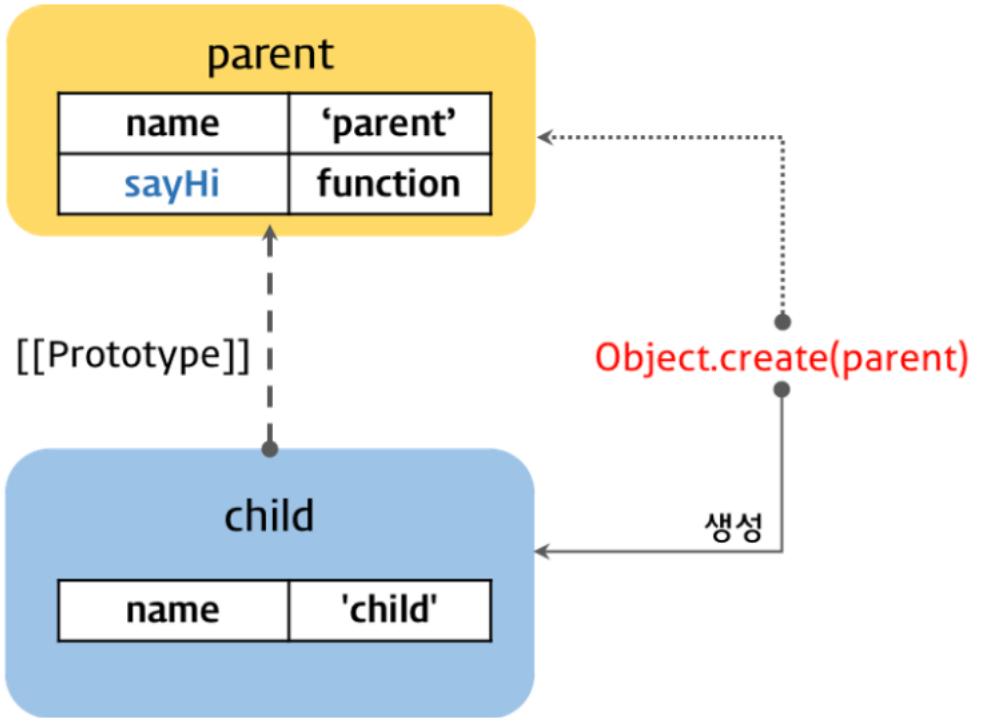
- 프로토타입으로 상속을 구현하는 방식.
- Object.create 함수를 이용하여 객체에서 다른 객체로 '직접' 상속을 구현하는 방식.

```
// 부모 생성자 함수
var Parent = (function(){
    // Constructor
    function Parent(name) {
        this.name = name;
    }
    // method
    Parent.prototype.sayHi = function(){
        console.log('Hi! ' + this.name);
    };
    // return constructor
    return Parent;
}());

// create 함수의 인수 = 프로토타입
var child = Object.create(Parent.prototype);
child.name = 'child';

child.sayHi(); // Hi! child
console.log(child instanceof Parent); // true
```



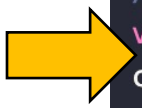
- 의사 클래스 패턴 상속에서는 불가능했던 객체리터럴 패턴으로도 상속 구현이 가능하다

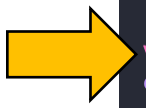
```
// 객체리터럴 패턴으로 생성한 객체의 상속
var parent = {
    name: 'parent',
    sayHi: function(){
        console.log('Hi! ' + this.name);
    }
};

// create 함수의 인자 = 객체
var child = Object.create(parent);
child.name = 'child';
//var child = Object.create(parent,{name: {value: 'child'}});

parent.sayHi(); // Hi! parent
child.sayHi();  // Hi! child

console.log(parent.isPrototypeOf(child)); // true
```



#5. 상속_[전격 비교!!]

```
// 부모 생성자 함수
var Parent = (function(){
    // Constructor
    function Parent(name) {
        this.name = name;
    }
    // method
    Parent.prototype.sayHi = function(){
        console.log('Hi! ' + this.name);
    };
    // return constructor
    return Parent;
}());

// create 함수의 인수 = 프로토타입
var child = Object.create(Parent.prototype);
child.name = 'child';

child.sayHi(); // Hi! child
console.log(child instanceof Parent); // true
```





```
// 부모 생성자 함수
var Parent = (function(){
    // Constructor
    function Parent(name) {
        this.name = name;
    }
    // Method
    Parent.prototype.sayHi = function(){
        console.log('Hi! ' + this.name);
    };
    // return constructor
    return Parent;
}());

// 자식 생성자 함수
var Child = (function(){
    // Constructor
    function Child(name){
        this.name = name;
    }
    // 자식 생성자 함수의 프로토타입 객체를 부모 생성자 함수의 인스턴스로 교체
    Child.prototype = new Parent(); // ②
    // 메소드 오버라이트
    Child.prototype.sayHi = function(){
        console.log('안녕하세요! ' + this.name);
    };
    // sayBye 메소드는 Parent 생성자함수의 인스턴스에 위치된다
    Child.prototype.sayBye = function(){
        console.log('안녕히가세요! ' + this.name);
    };
    // return constructor
    return Child;
}());

var child = new Child('child'); // ①
console.log(child);  // Parent { name: 'child' }

console.log(Child.prototype); // Parent { name: undefined, sayHi: [Function], sayBye: [Function] }

child.sayHi();  // 안녕하세요! child
child.sayBye(); // 안녕히가세요! child

console.log(child instanceof Parent); // true
console.log(child instanceof Child);  // true
```

#6. 캡슐화와 모듈패턴

- **캡슐화[Encapsulation]**
  - 관련있는 멤버 변수와 메소드를 하나의 틀 안에 담고 외부에 공개될 필요가 없는 정보는 숨기는 것
  - Java에서는 private/public 키워드 이용
  - Javascript에서는?

var를 쓰면 private변수가 되고,
this를 쓰면 public멤버가 된다.

```
var Person = function(arg) {
    var name = arg ? arg : '';

    this.getName = function(){
        return name;
    };

    this.setName = function(arg){
        name = arg;
    };
}

var me = new Person('Lee');
var me_name = me.getName();
console.log(me.name); //undefined
console.log(me_name);    //Lee

me.setName('Kim');
me_name = me.getName();
console.log(me_name);    //Kim
```

#6. 캡슐화와 모듈패턴

- 모듈 패턴 [Module Pattern]
  - 캡슐화 구현을 위해 사용되는 디자인 패턴
  - 클로저를 기반으로 동작

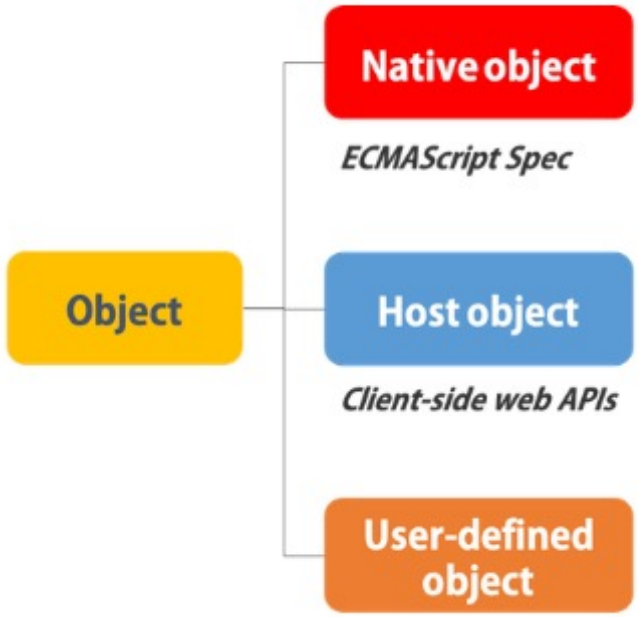
person함수는 객체를 반환한다.
이 객체 내의 메소드 getName, setName은
클로저로서 private변수인 name에 접근할 수 있다.

  - 주의할 점
    1. private 멤버가 객체나 배열일 경우, 반환된 멤버의 변경이 가능하다.
    2. person 함수가 반환한 객체는 person 함수 객체의 프로토타입에 접근할 수 없다. 이는 상속을 구현할 수 없음을 의미한다.

```
var person = function(arg) {
    var name = arg ? arg : '';

    return {
      getName: function() {
        return name;
      },
      setName: function(arg) {
        name = arg;
      }
    }
  }

  var me = person('Lee'); /* or var me = new person('Lee'); */
  var me_name = me.getName();

  console.log(me_name); // Lee

  me.setName('Kim');
  me_name = me.getName();

  console.log(me_name); // Kim
```

Ch.20
빌트인 객체
(Built-in Object)

#0. 자바스크립트 객체의 분류



- Object, String, Number, Boolean, Symbol, Function, Array, RegExp, Date, Math, Promise…

- Global object: window, global(Node.js)
- DOM: Document, Event, HTMLElement, HTMLCollection, NodeList…
- BOM: Location, History, Navigator, Screen
- Ajax: XMLHttpRequest, fetch
- HTML5 APIs: Blob, File, Canvas, Geolocation, Drag&Drop, Web storage, Web worker…

#1. 네이티브 객체

- Native Object / Built-in objects / Global objects
- ECMAScript 명세에 정의된 객체로, 어플리케이션의 환경과 관계없이 언제나 사용 가능.
- 함수 객체 + 메소드로 구성됨.
- 전역객체(Global object)와 다른개념임! 혼동주의

#2. 호스트 객체

- 호스트 환경에 정의된 객체
- 브라우저에서 동작하는 환경과 브라우저 외부에서 동작하는 환경의 자바스크립트(Node.js)는 다른 호스트 객체를 사용할 수 있다
- 이하 생략

Ch.21
전역 객체
(Global Object)

#0. 전역객체[Global Object]란?

- 모든 객체의 유일한 최상위 객체
- Javascript에 미리 정의된 객체로, 전역 프로퍼티나 전역 함수를 담는 공간 (개발자는 전역객체 생성 X)
- 전역객체는 전역 스코프(Global Scope)를 가짐
- 일반적으로 전역객체의 기술은 생략
- 전역변수 = 전역객체의 프로퍼티
  전역함수 = 전역객체의 메소드
- Javascript의 모든 객체는 전역객체의 프로퍼티가 된다
- Browser-side -> window객체
  Server-side(Node.js) -> global객체

```
//in browser console
this === window

//in Terminal
node
this === global
```

#1. 전역 프로퍼티 [Global Property]

- 전역객체의 프로퍼티로, 애플리케이션 전역에서 사용되는 값들을 나타내기 위해 사용
- 간단한 값이 대부분이며 다른 프로퍼티나 메소드를 가지고 있지 않음
- Infinity, NaN, undefined

```
console.log(window.Infinity); //Infinity
console.log(3/0);             //Infinity
console.log(-3/0);            //-Infinity
console.log(typeof Infinity); //number

console.log(window.NaN);      //NaN
console.log(Number('xyz'));   //NaN
console.log(1*'string');      //NaN
console.log(typeof NaN);      //number

console.log(window.undefined); //undefined
var foo;
console.log(foo);              //undefined
console.log(typeof undefined); //undefined
```

#2. 전역 함수 [Global Functions/Methods]

- 애플리케이션 전역에서 호출 가능한 함수로, 전역 객체의 메소드이다.
- **isFinite(testvalue) => boolean**

  : 주어진 값이 유한한지 판별하는 메소드 .
  testvalue를 인자로 받아 NaN, Infinity, undefined이면 false, 아니면 true 반환
- **isNaN(testvalue) => boolean**

  : 주어진 값이 NaN인지 판별하는 메소드.

  testvalue를 인자로 받아 NaN이면 true, 아니면 false 반환
- **parseInt(string, radix) => number**

  : 주어진 문자열을 정수형 숫자로 변환하여 반환하는 메소드.

  string과 radix를 인자로 받아 string을 radix진법의 정수로 변환하여 반환한다.

  반환이 불가능하면 false, radix가 주어지지 않을 경우 string이 0x로 시작하면 16진수로, 그외에는 10진수로 반환한다.
- parseFloat, encodeURI, decodeURI, …

```
console.log(isFinite(Infinity)); // false
console.log(isFinite(NaN));      // false
console.log(isFinite('Hello'));  // false

console.log(isFinite(0));        // true
console.log(isFinite('10'));     // true
console.log(isFinite(null));     // true

//참고
Number(null)  // 0
Boolean(null) // false
```

```
parseInt(10);
parseInt(10.123);
parseInt('10');
parseInt('10.123'); // 모두 10

parseInt('10',2);   // 2진수 10 -> 10진수 2
parseInt('10',8);   // 8진수 10 -> 10진수 8
parseInt('10',16);  // 16진수 10 -> 10진수 16
```

```
isNaN(NaN)       // true
isNaN(undefined) // true: undefined → NaN
isNaN({})        // true: {} → NaN
isNaN('blabla')  // true: 'blabla' → NaN

isNaN(true)      // false: true → 1
isNaN(null)      // false: null → 0
isNaN(37)        // false

// strings
isNaN('37')      // false: '37' → 37
isNaN('37.37')   // false: '37.37' → 37.37
isNaN('')        // false: '' → 0
isNaN(' ')       // false: ' ' → 0

// dates
isNaN(new Date())              // false: new Date() → Number
isNaN(new Date().toString())   // true:  String → NaN
```

cf. 레퍼 객체 [Wrapper Object]

- 원시 타입은 객체가 아니기 때문에 프로퍼티나 메소드를 직접 추가할 수 없다.
  하지만 아래의 코드를 보면 str.length에서 에러가 나지 않는다. 왜그럴까 ?0?

```
var str = '문자열';
console.log(str.length); // 3
```

- **레퍼 객체**
  - 원시타입값에 대해 표준 빌트인 객체의 메소드를 호출할 때 생성되는 임시 객체
  - 원시타입 number, string, Boolean 데이터 타입에 각각 대응되는 Number, String, Boolean이 제공된다.

```
var str = "문자열"; // 문자열 리터럴 생성
var strObj = new String(str); // 문자열 객체 생성
console.log(str.length);
// 리터럴 값은 내부적으로 레퍼 객체를 생성한 후 length 프로퍼티를 참조함.
// 참조가 끝나면 레퍼 객체는 사라짐.

console.log(str == strObj);  //true
console.log(str === strObj); //false
console.log(typeof str);     //string
console.log(typeof strObj);  //object
```

Pop Quiz

```
var str = 'test';

String.prototype.myMethod = function() {
  return 'myMethod';
}

console.log(str.myMethod());
console.dir(String.prototype);

console.log(str.__proto__ === String.prototype);                  // ① true
console.log(String.prototype.__proto__  === Object.prototype);    // ② true
console.log(String.prototype.constructor === String);             // ③ true
console.log(String.__proto__ === Function.prototype);             // ④ true
console.log(Function.prototype.__proto__  === Object.prototype); // ⑤ true
```

Ch.22-26
네이티브 객체들

#1. Object

- Object() 생성자 함수는 객체를 생성한다.
- 생성자 함수의 인수값에 따라 강제 형변환된 객체를 반환한다. (인수값이 null/undefined이면 빈 객체)

```
// 1. 빈 객체를 반환하는 경우
var obj1 = new Object();
var obj2 = new Object(undefined);
var obj3 = new Object(null);

// 2. 형변환된 객체를 반환하는 경우 (순서대로 String, Number, Boolean 객체 반환)
var obj1 = new Object('String'); //var obj = new String('String');과 동치
var obj2 = new Object(123);      //var obj = new Number(123);와 동치
var obj3 = new Object(true);     //var obj = new Boolean(true);와 동치
```

#2. Function

- Javascript의 모든 함수는 Function 객체이며, new 연산자를 통해 생성이 가능하다.
- Function 생성자함수를 사용하는 방식은 잘 쓰이지 않는다.

```
var adder = new Function('a','b','return a+b');
adder(2,6); // 8
```

#3. Boolean

- 원시타입 boolean을 위한 레퍼 객체로, Boolean() 생성자함수로 Boolean 객체 생성 가능.
- Boolean객체는 true/false를 포함하고 있는 객체로, 원시타입 boolean과의 혼동에 주의하자

```
var foo = new Boolean(true);      // true
var foo = new Boolean('false');

var foo = new Boolean(false);     // false
var foo = new Boolean();
var foo = new Boolean('');
var foo = new Boolean(0);
var foo = new Boolean(null);

var x = new Boolean(false);
if (x) { // x는 객체로서 존재하는 것이기에 참으로 간주됨.
}
```

#4. Number

- 원시타입 number를 위한 레퍼 객체이다.
- Number() 생성자함수로 Number 객체 생성 가능.
  이때 인자가 숫자로 변환될 수 없다면 NaN을 반환하고, new를 붙이지 않아 Number()를 생성자로 사용하지 않으면 Number객체가 아닌 원시타입 숫자를 반환한다.

```
var x = new Number(123);
var y = new Number('123');
var z = new Number('str');

console.log(x); // 123
console.log(y); // 123
console.log(z); // NaN

var x = Number('123');
console.log(typeof x, x); // number 123
console.log(typeof y, y); // object 123
```

#4. Number_Number Property

- Number 객체를 생성할 필요 없이 Number.propertyName의 형태로 사용한다.
  - Number.EPSILON : JS에서 표현 가능한 수 있는 가장 작은 수
  - Number.MAX_VALUE : JS에서 사용 가능한 가장 큰 수
  - Number.MIN_VALUE : JS에서 사용 가능한 가장 작은 수
  - Number.POSITIVE_INFINITY : 양의 무한대 반환
  - Number.NEGATIVE_INFINITY : 음의 무한대 반환
  - Number.NaN : NaN을 나타내는 숫자값으로, window.NaN프로퍼티와 같다

#4. Number_Number Method

- **Number.isFinite(testValue: number) => boolean**

  : 주어진 값이 정상적인 유한수인지 판별하는 메소드. 전역함수 isFinite()는 인수를 숫자로 변환하여 검사하는 반면 Number.isFinite()는 인수를 변환하지 않기에 숫자가 아닌 인수가 주어지면 항상 false를 반환.

- **Number.isInteger(testValue: number) => boolean**

  : 주어진 값이 정수인지 판별하는 메소드. 검사 전에 인수를 숫자로 변환하지 않는다.

- **Number.isNaN(testValue: number) => boolean**

  : 주어진 값이 NaN인지를 판별하는 메소드. 검사 전에 인수를 숫자로 변환하지 않는다.

- **Number.prototype.toExponential(fractionDigits: number) => string**

  : 주어진 값을 지수 표기법으로 변환하여 문자열로 반환하는 메소드.

- **Number.prototype.toFixed(fractionDigits: number) => string**

  : 주어진 값의 소숫점자리를 반올림하여 문자열로 반환하는 메소드.

- **Number.prototype.toString(radix: number) => string**

  : 주어진 진법에 맞게 숫자를 문자열로 변환하여 반환하는 메소드.

#5. Math

- 수학 상수와 함수를 위한 프로퍼티와 메소드를 제공하는 빌트인 객체.
- Math 객체는 생성자 함수가 아니므로 정적 프로퍼티와 메소드만을 제공한다.
- **Math Property**
  - Math.PI : 원주율을 값으로 갖는 속성
  - Math.E : 자연상수 e를 값으로 갖는 속성
  - Math.LN2 : log2를 값으로 갖는 속성
  - Math.SQRT2 : $\sqrt{2}$를 값으로 갖는 속성

```
console.log(Math.PI);    //3.141592653589793
console.log(Math.E);     //2.718281828459045
console.log(Math.LN2);   //0.6931471805599453
console.log(Math.SQRT2); //1.4142135623730951
```

#5. Math_Math Method

- Math.abs(x) => number : 주어진 값의 절댓값(|x|)을 반환.
- Math.ceil(x) => number : 주어진 값보다 큰 가장 작은 정수(올림, [x])를 반환
- Math.floor(x) => number: 주어진 값보다 작은 가장 큰 정수(버림, [x])를 반환
- Math.max([val1, val2, ..., valN]) => number: 주어진 값들 중 가장 큰 값을 반환
- Math.min([val1, val2, ..., valN]) => number: 주어진 값들 중 가장 작은 값을 반환
- Math.pow(x, y) => number: xy 의 값을 반환
- Math.random() => number: 0 이상 1 미만의 랜덤값을 반환
- Math.round(x) => number: 주어진 값에 가장 가까운 정수(반올림, [x + 0.5])를 반환
- Math.sqrt(x) => number: 주어진 값의 square root값($\sqrt{2}$)을 반환

```
Math.abs(-1);         //1
Math.abs('');         //0
Math.abs('string'); //NaN

Math.round(1.4);      //1
Math.round(1.6);      //2

Math.ceil(1.4);       //2
Math.ceil(1.6);       //2

Math.floor(1.4);      //1
Math.floor(1.6);      //1

Math.sqrt(9);         //3

Math.random();        //random_num

Math.pow(2,8);        //256

Math.max(1,2,3);      //3
Math.min(1,2,3);      //1
```

#6. Date

- 날짜와 시간을 위한 메소드를 제공하는 빌트인 객체이자 생성자 함수
- Date생성자함수로 생성한 Date 객체는 내부적으로 숫자값을 가짐
- Date 객체 생성 방법 [4가지]

    1. **new Date()** : 현재 날짜,시간을 가지는 인스턴스 반환
    2. **new Date(msec)** : Epoch 시각으로부터 전달된 밀리초만큼 경과한 날짜,시간을 가지는 인스턴스 반환
    3. **new Date(dateString)** : 날짜,시간을 나타내는 문자열을 전달하면 그에 따른 인스턴스 반환.
    4. **new Date(year, month, day, hrs, min, sec, msec)** : 인수로 년, 월, 일, 시, 분, 초, 밀리초를 의미하는 숫자를 전달하면 그에 따른 인스턴스 반환.

```
//1
let date = new Date();
console.log(date);  // Fri Jan 26 2024 19:07:06 GMT+0900 (한국 표준시)
//2
date = new Date(8640000);
console.log(date);  // Thu Jan 01 1970 11:24:00 GMT+0900 (한국 표준시)
//3
date = new Date('2019/05/16/17:22:10');
console.log(date);  // Thu May 16 2019 17:22:10 GMT+0900 (한국 표준시)
//4
date = new Date(2019,4,16,17,24,30,0);
console.log(date);  // Thu May 16 2019 17:24:30 GMT+0900 (한국 표준시)
```

인수	내용
year	1900년 이후의 년
month	월을 나타내는 0 ~ 11까지의 정수 (주의: 0부터 시작, 0 = 1월)
day	일을 나타내는 1 ~ 31까지의 정수
hour	시를 나타내는 0 ~ 23까지의 정수
minute	분을 나타내는 0 ~ 59까지의 정수
second	초를 나타내는 0 ~ 59까지의 정수
millisecond	밀리초를 나타내는 0 ~ 999까지의 정수

#6. Date_Date Method

- Date.now
- Date.parse
- Date.UTC
- Date.prototype.getFullYear / setFullYear
- Date.prototype.getMonth / setMonth
- Date.prototype.getDate / setDate
- Date.prototype.getDay
- Date.prototype.getHours / setHours …
- Date.prototype.toDateString
- Date.prototype.toTimeString

#7. String

- 원시타입 string을 다룰 때 유용한 프로퍼티, 메소드를 제공하는 레퍼 객체
- String 생성자함수로 생성한 String객체의 전달된 인자는 모두 문자열로 변환된다.
- new연산자를 사용하지 않으면 String객체가 아닌 문자열 리터럴을 반환하며, 형변환이 발생할 수 있다.

```
let strObj = new String('Lee');
console.log(strObj);    // [String: 'Lee']
strObj = new String(123);
console.log(strObj);    // [String: '123']
strObj = new String(undefined);
console.log(strObj);    // [String: 'undefined']


let x = String('Lee');
let y = new String('Lee');
console.log(typeof x, x);   // string Lee
console.log(typeof y, y);   // object [String: 'Lee']
```

#7. String_String Property

- String.length
  : 문자열 내의 문자 개수를 반환한다.

```
const str1 = 'Hello';
console.log(str1.length);   // 5

const str2 = 'Hellllllllo';
console.log(str2.length);   // 11
```

#7. String_String Method

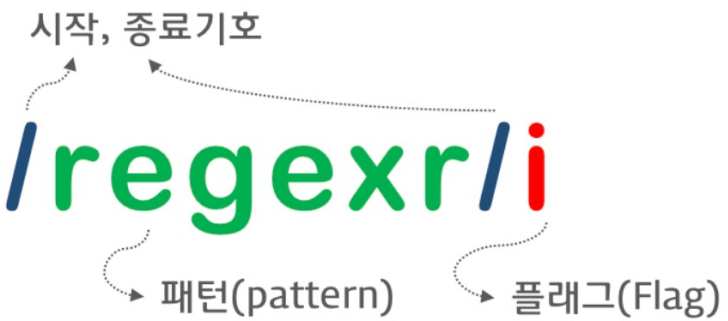
** 문자열은 변경 불가능한 원시값이기 때문에 String객체의 모든 메소드는 항상 새로운 문자열을 반환한다

- String.prototype.**charAt**(number) => string
- String.prototype.**concat**(strings) => string
- String.prototype.**indexOf**(string,index) => number
- String.prototype.**replace**(searchvalue, replacer) => string
- String.prototype.**split**([separator, limit]) => string[]
- String.prototype.**substring**(indexStart,indexEnd) => string
- String.prototype.**toLowerCase**() => string
- String.prototype.**toUpperCase**() => string
- String.prototype.**trim**() => string
- String.prototype.**includes**(string) => boolean

```
let str = 'Hello';

console.log(str.charAt(1)); // e
console.log(str.charAt(5)); // 빈 문자열 반환

str = str.concat(' World');
console.log(str); // Hello World

console.log(str.indexOf('or'));     // 7
console.log(str.indexOf('or',8));   // -1 (8 = 검색시작인덱스)

console.log(str.replace('World','Youjin')); // Hello Youjin
console.log(str.replace(/hello/gi,'안뇽'));  // 안뇽 World

console.log(str.split('o'));    // [ 'Hell', ' W', 'rld' ]
console.log(str.split('o',2));  // [ 'Hell', ' W' ]

console.log(str.substring(1,4));    // ell
console.log(str.substring(2));      // llo World

console.log(str.slice(0,5));    // Hello
console.log(str.slice(-5));     // World
console.log(str.slice(2));      // llo World

console.log(str.toLowerCase()); // hello world
console.log(str.toUpperCase()); // HELLO WORLD

let str2 = '  foo  ';
console.log(str2.trim());   // foo

console.log(str.includes('Wo'));    //true
console.log(str.includes('Wow'));   //false
```

#8. RegExp

- 정규표현식 [Regular Expression]
  - 문자열에서 특정 내용을 찾거나 대체, 발췌할 때 사용
  - 리터럴 표기법



<플래그 종류 (Optional) >
1) i [Ignore Case] : 대소문자 구별하지않고 검색
2) g [Global] : 문자열 내 모든 패턴 검색
3) m [Multi Line] : 문자열 행이 바뀌어도 계속 검색

<패턴에 사용되는 메타문자>
1) . : 임의의 문자 1개
2) + : 앞선 패턴을 최소 1번 반복
3) | : or의 의미
4) [] : [] 내의 문자는 or로 동작
5) - : [] 안에 -를 사용하면 범위 지정 가능

#8. RegExp

- Javascript는 정규표현식을 위해 RegExp를 지원한다.
  RegExp 객체를 생성하려면 리터럴 방식이나 생성자 함수를 사용할 수 있는데, 리터럴 방식이 일반적이다.

```
// 리터럴 방식
const regExp1 = /is/g;
// 생성자 함수
const regExp2 = new RegExp(/is/,'g');
```

- **RegExp Method**
  - RegExp.prototype.exec(target_string) => RegExpExecArray | null
    : 문자열을 검색하여 매칭 결과(배열 or null)를 반환한다.
  - RegExp.prototype.test(target_string) => boolean
    : 문자열을 검색하여 매칭 결과(true or false)를 반환한다.

```
const target = "Is this all there is?";
const regExp = /is/g;

const res = regExp.exec(target);    // exec메소드는 g플래그를 지정해도 첫번째 매칭결과만을 반환한다
console.log(res);   // [ 'is', index: 5, input: 'Is this all there is?', groups: undefined ]

const res2 = regExp.test(target);
console.log(res2);  // true
```

#9. Array

- 다음 시간에 ^0^

#10. Error

- Error 생성자는 error 객체를 생성
- Error객체의 인스턴스를 런타임에러가 발생할 때 throw된다

```
try {
  // foo();
  throw new Error('Whoops!');
} catch (e) {
  console.log(e.name + ': ' + e.message);
}
```

Exercise

Exercise 3.1: Validity of Number

인자로 받은 값이 1 이상 9 이하의 정수인지 판별하여 결과를 반환하는 함수 `isValidNumber`를 구현하여라.

`isValidNumber` 함수는 **Code 3.35**와 같이 동작하여야 한다.

Code 3.35 Exercise 3.1 examples

```
> isValidNumber(9);
true
> isValidNumber('4');
true
> isValidNumber('abc');
false
> isValidNumber(-5);
false
> isValidNumber(3.5);
false
> isValidNumber(3 / 0);
false
```

Exercise 3.2: Divisors

인자로 받은 정수의 모든 양의 약수(約數, divisor) 배열을 작은 순서대로 반환하는 함수 `getDivisors`를 구현하여라. 정수 x의 약수는 $\sqrt{x}$까지만 탐색하여도 모두 구할 수 있음을 이용하고, 배열의 `sort` 메서드를 이용하여라. `getDivisors` 함수는 **Code 3.36**과 같이 동작하여야 하며, 인자로 받은 값이 유효한 값인지 확인할 필요는 없다.

Code 3.36 Exercise 3.2 examples

```
> getDivisors(5);
[ 1, 5 ]
> getDivisors(24);
[ 1, 2, 3, 4, 6, 8, 12, 24 ]
> getDivisors(196);
[ 1, 2, 4, 7, 14, 28, 49, 98, 196 ]
```

Exercise 3.1

Code B.8 Exercise 3.1 answer

```
const isValidNumber = num => {
    const parsedNumber = parseInt(num);
    if (!isFinite(parsedNumber) || isNaN(parsedNumber)) return false;
    if (parsedNumber != num) return false;
    return parsedNumber >= 1 && parsedNumber <= 9;
};
```

Exercise 3.2

Code B.9 Exercise 3.2 answer

```
const getDivisors = num => {
    const divisors = [];
    for (let i = 1; i <= Math.sqrt(num); i++) {
        if (i * i === num) divisors.push(i);
        else if (num % i === 0) divisors.push(i, num / i);
    }
    return divisors.sort((first, second) => first - second);
};
```

- 끄읕 -