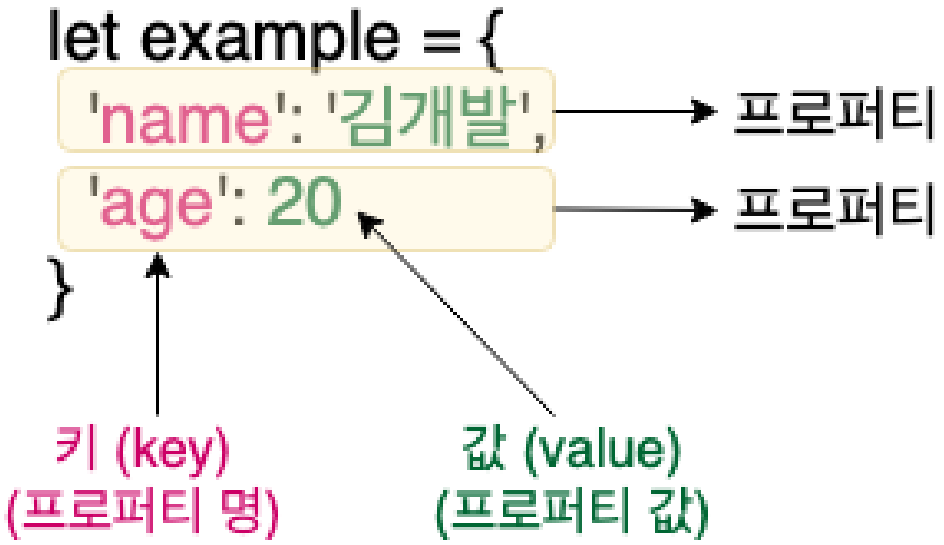
Chap10 객체 리터럴

10-1 객체란?



- 0개 이상의 프로퍼티로 구성된 집합(프로퍼티는 키와 값으 구성되어 있음)
- 프로퍼티 값은 자바스크립트에서 사용할 수 있는 모든 값은 다 사용할 수 있다.

10-2 객체 리터럴에 의한 객체 생성

- 객체 생성 방법
  - 객체 리터럴 → 가장 일반적이고 간단한 방법
  - Object 생성자 함수
  - 생성자 함수
  - Object.create 메서드
  - 클래스(ES6)

  ▼ 객체리터럴

  - 중괄호 안에 0개 이상의 프로퍼티를 정의하는 형식

```
var person={
 name:'Lee',
 sayHello: function(){
   console.log(`Hello! My name is ${this.name}.`);
   }
  };

  console.log(typeof person); //obhect
  console.log(person); //{name:'Lee',sayHeelo:f}
```

  - 만약 중괄호 내에 프로퍼티를 정의하지 않으면 빈 객체가 생성된다.

```
var empry={}; //빈객체
console.log(typeof empty); //object
```

  원시값을 만드는 것과유사하게 리터럴로 객체를 생성한다. 객체 리터럴에 프로퍼티를 포함시켜 객체를 생서함과 동시에 프로퍼티를 만들 수 있고 객체를 생성한 이후에 프로퍼티를 동적으로 추가할 수 있다.

10-3 프로퍼티

- 키와 값을 구성
- 여러 프로퍼티를 나열할때 쉼표를 사용하여 구분한다. 마지막 프로퍼티 뒤에는 쉼표를 사용하지 않으나 사용해도 상관없다.
- 프로퍼티 키

- 키 이름은 식별자 네이밍 규칙을 따를 경우 따옴표 생략해도 되는데 아닐 경우에는 따옴표를 반드시 사용해야 한다.
- 일반적으로 문자열 사용. 심벌 값도 가능. 문자열, 심벌 값 외의 값을 사용하면 암묵적 타입 변환을 통해 문자열이 된다.
- 빈 문자열을 프로퍼티 키로 사용해도 에러가 나지 않는다. 하짐나 키로서 의미를 갖지 못하므로 권장하지 않는다.
- 프로퍼티 키에 표현식을 사용해 동적으로 생성할 수 있지만 대괄호로 묶어야 한다.

```
var obj={};
var key='hello';

obj[key]='world';

console.log(obj); //{hello:'world}
```

- 이미 존재한느 프로퍼티 키를 중복 선언하면 나중에 선언한 프로퍼티가 먼저 선언한 프로퍼티를 덮어쓴다. 에러 발생하지 않는다는 점에 주의하자.

```
var foo={
 name:'Lee',
 name:'Kim'
};
console.log(foo); //{name:'Kim'}
```

10-4 메서드

- 자바스크립트에서 사용할 수 있는 모든 값은 프로퍼티 값으로 사용할 수 있듯이 함수도 프로퍼티 값으로 사용할 수 있다.
- 프로퍼티 값이 함수일 경우 `메서드` 라 부른다.

```
var circle={
 redius:5, // <- 프로퍼티
 getDiameter:function(){ // <- 메서드
  return 2*this.redius;
 }
```

```
};
console.log(circle.getDiameter()); //10
```

▼ this 키워드

객체 자신을 가리키는 참조 변수

10-5 프로퍼티 접근

▼ 방법 1. 마침표 프로퍼티 접근 연산자(.)를 사용하는 마침표 표기법(dot notation)

```
var person={
 name:'Lee'
}
console.log(person.name); //Lee
```

▼ 방법 2. 대괄호 프로퍼티 접근 연산자([...])를 사용하는 대괄호 표기법(bracket notation)

- 대괄호 프로퍼티 접근 연산자 내부에 지정하는 프로퍼티 키는 반드시 따옴표로 감싼 문자열이어야 한다. 안감싸면 자바스크립트 엔진이 식별자로 해석하여 ReferenceError가 발생할 수 있다.
  - 단 프로퍼티 키가 숫자로 된 문자열인 경우 따옴표 생략 가능.
- 그러나 아예 객체에 선언조차 하지않은 프로퍼티에 접근하면 에러가 아닌 undefined를 반환한다.
- 식별자 네이밍 규칙을 준하지 않는 이름을 사용시 사용

```
var person={
 name:'Lee'
};

console.log(peerson['name']); //<- 올바른 표기법. 결과값 : Lee
```

💡 프로퍼티 키가 식별자 네이밍 규칙을 준하는 일므일 경우 마침표 표기법과 대괄호 표기법 모두 사용할 수 있다.

▼ 퀴즈

```
var person={
 'last-name':'Lee',
  1:10
 };
person.'last-name'; //SyntacError. 식별자 네이밍 규칙에 어긋나는
person.last-name;
```

person-last-name을 실행할 때 자바스크립트 엔진은 먼저 person.last를 평가한다. person객체에는 프로퍼티가 last인 프로퍼티가 없기 때문에 undefiend로 평가된다. 그리고 그 후 name을 평가하게 되는데 undefined-name과 같은 형태로 평가를 하게 된다. 이때 name은 프로퍼티 키가 아니라 **식별자로** 해석된다.

- Node.js환경

name이라는 식별자 선언이 없으므로 ReferenceError:name is not defined라는 에러 발생

- 브라우저

브라우저 환경에서는 name이라는 전역변수(전역 객체 window의 프로퍼티)가 암묵적으로 존재.

즉, 브라우저 환경에서는 `window.name`이 빈 문자열 `""`이므로 숫자 `0`으로 변환된다.

따라서 undefined - 0이므로 NaN을 반환한다.

10-6 프로퍼티 값 갱신

- 이미 존재하는 프로퍼티에 값을 할당하면 프로퍼티 값이 갱신된다.

```
var person={
 name:'Lee'
};

//person 객체에 name 프로퍼티가 존재하므로 name 프로퍼티의 값이 갱신된다
person.name='Kim';
```

```
console.log(person); //{name:'kim'}
```

10-7 프로퍼티 동적 생성

존재하지 않는 프로퍼티에 값을 할당하면 프로퍼티가 동적으로 생성되어 추가되고 프로퍼티 값이 할당된다.

```
var person={
 name:'Lee'
};

//person 객체에는 age프로퍼티 존재x
//따라서 person 객체에 age 프로퍼티가 동적으로 생성되고 값이 할당된다.
person.age=20;

console.log(person); //{name:'Lee',age:20}
```

10-8 프로퍼티 삭제

- delete 연산자 사용하여 프로퍼티 삭제 가능
- delete 연산자의 피연산자는 프로퍼티 값에 접근할 수 있는 표현식이어야 한다. 만약 존재하지 않는 프로퍼티를 삭제하면 아무런 에러 없이 무시된다.

```
var person={
 name:'Lee'
};
person.age=20;

delete person.age;

delete.person.address;

console.log(person); //{name:'Lee'}
```

10-9 ES6에서 추가된 객체 리터럴의 확장 기능

▼ 프로퍼티 축약 표현

- 프로퍼티 값으로 변수를 사용하는 경우 변수 이름과 프로퍼티 키가 동이한 이름일 때 프로퍼티 키를 생략할 수 있다. 이때 프로퍼티 키는 변수 이름으로 자동 생성된다.

```
let x=1; y=2;

const obj={x,y};
console.log(obj); //{x:1,y=2}
```

▼ 계산된 프로퍼티 이름

문자열 또는 문자열로 타입 변환할 수 있는 값으로 평가되는 표현식을 사용해 프로퍼티 키를 동적으로 생성할 수도 있습니다. 단, 프로퍼티 키로 사용할 표현식을 대괄호([...])로 묶어야 합니다. 이를 **계산된 프로퍼티 이름(computed property name)** 이라 한다.

ES5에서 계산된 프로퍼티 이름으로 키를 동적 생성하려면 객체 리터럴 외부에서 대괄호 표기법을 사용해야 했다.

```
// ES5
var prefix = 'prop';
var i = 0;

var obj = {};

// 계산된 프로퍼티 이름으로 프로퍼티 키 동적 생성
obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;
obj[prefix + '-' + ++i] = i;

console.log(obj); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

ES6에서는 객체 리터럴 내부에서도 계산된 프로퍼티 이름으로 키를 동적 생성할 수 있습니다.

```
// ES6
const prefix = 'prop';
let i = 0;

// 객체 리터럴 내부에서 계산된 프로퍼티 이름으로 키를 동적 생성
const obj = {
  [`${prefix}-${++i}`] : i,
  [`${prefix}-${++i}`] : i,
  [`${prefix}-${++i}`] : i,
};

console.log(obj); // {prop-1: 1, prop-2: 2, prop-3: 3}
```

▼ 메서드 축약 표현

ES6에서는 메서드를 정의할 때 function키워드 생략한 축약 표현 사용 가능

```
// ES6
const obj = {
  name: 'Jo',
  // 메서드 축약 표현
  sayHi() {
    console.log('Hi!' + this.name)
  }
};

obj.sayHi(); // 'Hi! Jo'
```

ES6의 메서드 축약 표현으로 정의한 메서드는 **프로퍼티에 할당한 함수와 다르게 동작**합니다.

즉 function()사용한거랑 생략한거랑 다르게 동작한다는 뜻이다.

💡 ES6 이전 사양에는 메서드에 대한 명확한 정의가 없었습니다. 일반적으로 객체에 바인딩 된 함수를 모두 메서드라 불렀지만, ES6 사양에서는 메서드에 대한 정의가 명확하게 규정되었습니다.

ES6 사양에서 메서드는 메서드 축약 표현으로 정의된 함수만을 의미합니다.

따라서

```
const obj = {
  x: 1,
  // foo는 메서드 입니다.
  foo() {
    return this.x;
  },
  // bar에 바인딩된 함수는 일반 함수입니다.
  bar: function() {
    return this.x;
  }
};

console.log(obj.foo()); // 1
console.log(obj.bar()); // 1
```

ES6 사양에서 정의한 메서드는 인스턴스를 생성할 수 없는 `non-constructor` 입니다.

```
new obj.foo(); // TypeError: obj.foo is not a constructor
new obj.bar(); // bar {}
```

ES6 메서드는 인스턴스를 생성할 수 없으므로 `property 프로퍼티` 가 없고 프로토타입도 생성하지 않습니다.