객체:기본

ps. 프로퍼티와 속성이 같은 의미를 가리킴을 유의 (자주 혼용해서 사용)

객체

- 객체형은 원시형과 달리 **다양한 데이터**를 함께 담을 수 있음
- 중괄호 안에는 **키(key): 값(value) 쌍으로 구성된 프로퍼티(property)**를 여러 개 넣을 수 있는데, 키엔 **문자형**, 값엔 **모든 자료형**이 허용됨
- 객체를 생성하는 두 가지 방법

```
// '객체 생성자' 문법
let user = new Object();

// '객체 리터럴' 문법 (중괄호 이용)
// 객체를 선언할 땐 주로 객체 리터럴 방법을 사용
let user = {};
```

- 중괄호 {...} 안에는 **키:값 쌍**으로 구성된 프로퍼티(=속성)가 들어감
    - **콜론(:)**을 기준으로 **왼쪽엔 키가, 오른쪽엔 값이 위치**

```
// '객체 리터럴' 문법으로 객체 생성
let user = {
    // 키: "name", 값: "John"
    name: "John",
    // 키: "age", 값: 30
    age: 30
};
```

- **점 표기법(dot notation)**을 이용하여 프로퍼티 값에 접근 가능

```
// 점 왼쪽에 객체의 이름이, 오른쪽에는 속성 이름이 위치
alert( user.name ); // John
alert( user.age ); // 30
```

- 객체에 **동적으로 속성을 추가 가능**
    - 클래스에 미리 정의해놓은 속성(ex: 자바의 클래스 필드)만 사용 가능한 타 언어와 다른 점

```
user.isAdmin = true;
```

- **delete 연산자**를 사용하여 프로퍼티 삭제 가능
    - delete 연산자 사용시 **프로퍼티 삭제가 성공했으면 true, 실패했으면 false를 반환**

```
// age 프로퍼티 삭제
delete user.age;
console.log(user.age); // undefined

// 반환값이 필요한 경우
let result = delete user.age;
console.log(result); // true

// Math 객체의 PI 상수는 삭제 불가
console.log(delete Math.PI); // false
console.log(Math.PI); // 3.14159...
```

- 스페이스나 하이픈(-)과 같은 기호를 조합해서 프로퍼티 이름을 만들어야 할 경우, 프로퍼티 이름을 **따옴표로 감싸야 함** (비권장)

```
let user = {
    name: "John",
    age: 30,
    // 스페이스나 하이픈이 포함된 프로퍼티 이름을 사용해야 할 경우 따옴표로 묶어주기
    "likes birds": true
};
```

- **상수로 선언된 객체도 객체의 내용은 수정 가능함**을 유의
    - 상수 => **새로운 값의 대입(값의 변경)이 불가능**하다는 의미

```
const user = {
    name: "John"
};

// 상수에는 새로운 값의 "대입"이 불가능함! (Uncaught TypeError: Assignment to
constant variable.)
user = {}

// 객체를 통해 접근하여 객체의 속성값을 수정하는 것은 가능함
user.name = "Pete";
```

- **대괄호**를 이용하여 프로퍼티 접근, 수정(추가), 삭제 가능
    - 스페이스나 하이픈(-)과 같은 기호를 조합해서 프로퍼티 이름을 만든 경우, **대괄호와 문자열을 통해서만 접근 가능** => 점 표기법으로는 접근 불가

```
let user = {};

// 수정 (추가)
user["likes birds"] = true;
// 점 표기법으로는 접근 불가능!
// user.likes birds = true;
// user.likes-birds = true;

// 접근
alert(user["likes birds"]);

// 삭제
delete user["likes birds"];
```

- 만약에 **코드 실행 중에 변하는 값(ex: 변수)을 이용하여 프로퍼티에 접근**하고 싶은 경우 대괄호를 이용하여 접근

```
let user = {
    name: "John",
    age: 30
};

let key = prompt("사용자의 어떤 정보를 얻고 싶으신가요?", "name");

// 변수를 이용하여 프로퍼티 접근 (변수값이 평가되고 이후 평가된 값을 키로 이용하여 속성값 접근)
alert( user[key] );
// 밑의 방법은 "key라는 이름의 속성"에 접근하는 것임을 유의
// alert( user.key );
```

값 단축 구문 사용

- **값 단축 구문(property value shorthand)**을 이용한 객체 생성 간소화
    - 프로퍼티 값을 변수에서 받아온 값으로 설정할 때 유용하게 사용 가능

```
let name = "John";
let age = 20;

// 값 단축 구문없이 객체 정의
let user = {
    name: name,
    age: age
}

console.log( user ); // { name: "John", age: 20 }

// "변수와 속성 이름이 같은 경우" 다음과 같이 축약하여 객체 정의 가능
user = {
    // name: name과 같은 의미
    name,
    // age: age와 같은 의미
    age
}

console.log( user ); // { name: "John", age: 20 }

function makeUser(name, age) {
    return { name, age };
}

console.log( makeUser("John", 20) ); // { name: "John", age: 20 }
```

- 다음과 같이 일반 프로퍼티와 단축 프로퍼티를 함께 사용하는 것도 가능

```
let name = "John";

let user = {
    // name에만 값 단축 구문 사용
    name,
    age: 30
};
```

프로퍼티 이름의 제약사항

- 변수 이름엔 for, let, return 같은 예약어를 사용할 수 없지만, 객체 프로퍼티엔 이런 제약이 적용되지 않음

```
// 예약어를 키로 사용해도 무방함
let obj = {
    for: 1,
    let: 2,
    return: 3
};

alert( obj.for + obj.let + obj.return ); // 6
```

- 객체의 키로 문자 자료형(+ 심볼)만 사용 가능하므로, **문자형이나 심볼형에 속하지 않은 값은 문자열로 자동 형 변환됨**

```
let obj = {
    // 숫자형이므로 문자열로 변환 (0 => "0")
    // 즉, "0": "test"와 동일
    0: "test"
};

alert( obj["0"] ); // "test"
// 여기서 전달된 숫자 0은 문자열 "0"으로 변환됨을 유의
alert( obj[0] ); // "test"
```

- 단, 예외적으로 __proto__ 는 특수한 용도(프로토타입 객체의 게터, 세터)로 사용되는 이름이므로 사용 불가

in 연산자를 사용한 속성 존재 여부 확인

- 자바스크립트 객체의 중요한 특징 => **존재하지 않는 프로퍼티에 접근하려 해도 에러가 발생하지 않고 undefined를 반환**함

```
let user = {};

// noSuchProperty 프로퍼티가 존재하지 않으므로 true
alert( user.noSuchProperty === undefined );
```

- **in 연산자**를 사용하여 특정 속성이 객체에 존재하는지 여부를 파악 가능

```
let user = {
    name: "John"
};

// name 속성이 존재하므로 true 출력
alert( "name" in user );
// noSuchProperty 속성이 존재하지 않기 때문에 false 출력
alert( "noSuchProperty" in user );
```

- 존재하지 않는 속성값과 undefined를 비교하여 속성값 존재 여부를 파악하는 것도 가능하지만, **속성값이 undefined인 경우 문제 발생 가능**

```
let user = {
    name: undefined
};

// true
alert( "name" in user );
// name 속성은 존재하지만 undefined이므로 true 출력
alert( user.name === undefined );
```

for .. in 반복문

- for .. in 반복문을 사용하면 **객체의 모든 키를 순회** 가능
    - 단, 배열과 같은 자료구조에 저장된 요소들을 순회하기 위해서 for .. in 반복문을 사용하면 안됨, 그런 상황에는 **for .. of 반복문**을 사용

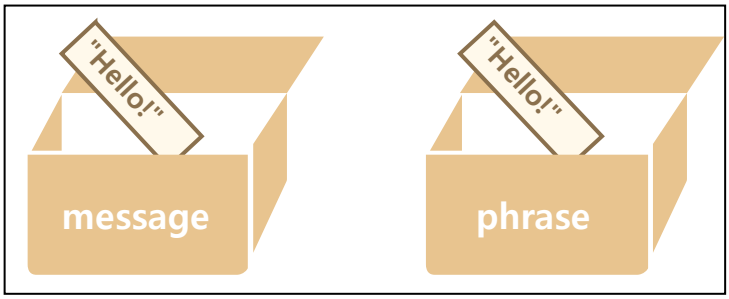
```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};

// name : John
// age : 30
// isAdmin : true
for (let key in user) {
    // key 변수에는 "name", "age", "isAdmin"이 순차적으로 대입됨
    alert(key + " : " + user[key]);
}
```

참조에 의한 객체 복사

- 객체와 원시 타입의 근본적인 차이 중 하나는 **객체는 참조에 의해(by reference) 저장되고 복사된다는 점**
- 원시 값을 대입할 경우 말 그대로 **값의 복사(deep copy)**가 일어남

```
let message = "Hello!";
// 원시값 대입으로 인하여 값의 복사가 발생
let phrase = message;
```

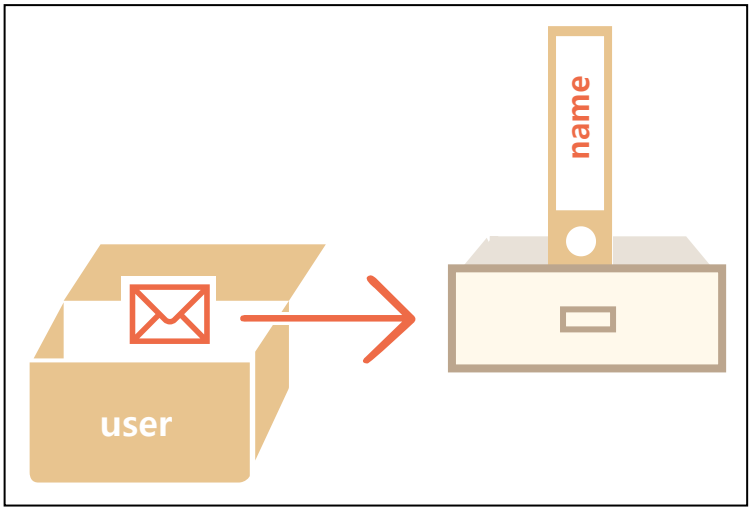
- 위 코드는 다음과 같이 **서로 다른 변수에 각기 독립된 값("Hello")을 저장**



- 객체를 변수에 대입할 경우 **객체에 대한 참조(=메모리 주소, 화살표)가 저장**됨 (C로 치자면 포인터와 비슷한 개념으로 이해)

```
let user = {
    name: "John"
};
```

다음과 같이 값 자체가 아닌, 해당 **값(객체)을 가리키고 있는 화살표를 저장**한다고 이해하기




```
let user = { name: "John" };

// 참조값을 대입(복사)함, 따라서 admin과 user 변수는 "같은 객체"를 가리키게 됨
let admin = user;

// admin 참조에 의해서 값이 변경됨
admin.name = 'Pete';

// (둘이 가리키는 객체가 결국 "같은 참조를 가진 같은 객체"이므로) user 참조 값을 이용해서도 변경사항을 확인 가능
alert(user.name); // Pete 출력
```

- 함수의 인수로 전달받는 과정에서도 참조값의 복사가 진행됨

```
let user = { name: "John" };

function changeName(user) {
    // 원본 객체에 영향을 끼침
    user.name = "Pete";
}

alert(user.name); // "John"
changeName(user);
alert(user.name); // "Pete"
```

- **객체끼리 값 비교시 ==, === 연산자는 동일한 결과를 출력**
    - 어차피 비교 대상이 둘 다 객체이므로 타입까지 검사하는 === 연산자와 차이가 없어짐
    - 비교 후 **피연산자인 두 객체가 동일한 참조를 가진 객체인 경우에 참을 반환**함

```
let a = {};
let b = a; // 참조에 의한 복사

alert( a == b ); // true, 두 변수는 같은 객체를 참조합니다.
alert( a === b ); // true, == 연산자와 결과 같음
```

내용이 같아도 참조가 다르면 false를 반환함을 유의

```
let a = {};
let b = {}; // 독립된 두 객체
alert( a == b ); // false, 같은 객체를 참조하지 않음

let user1 = { name: "John" };
let user2 = { name: "John" };
alert( user1 == user2 ); // false
```