06

객체

목차

배열

- **배열**
  - 배열에는 여러 자료형을 요소로 저장 가능
  - 자료형은 객체
  - 자동으로 멤버 속성 length 생성
  - [] 또는 new Array()로 생성
  - 인덱스로 배열 요소를 다룹니다
  - 인덱스는 0부터 시작

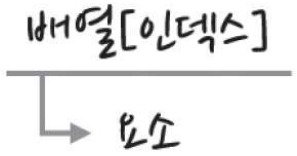


그림 4-2 배열의 요소

```
const arr1 = [1, 2, 3]; // 배열 리터럴 방식
const arr2 = new Array(4, 5, 6); // Array 생성자 사용
const arr3 = Array.of(7, 8, 9); // ES6 - Array.of() 사용
const arr4 = Array(3); // 크기 3의 빈 배열 생성

console.log(arr1);
console.log(arr2);
console.log(arr3);
console.log(arr4);
```

배열

배열

- 배열은 요소에 접근할 때 인덱스를 사용하고, 객체는 키를 사용함

```
const numbers = [10, 20, 30];

for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}
```

```
numbers.forEach((num, index) => {
  console.log(`Index: ${index}, Value: ${num}`);
});
```

```
for (const num of numbers) {
  console.log(num);
}
```

```
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [20, 40, 60]
```

배열

- **배열 요소 처리**

기능	메서드	설명
요소 추가	push(), unshift()	마지막 또는 처음에 추가
요소 제거	pop(), shift(), splice()	마지막, 처음, 특정 요소 삭제
요소 변경	splice(), map()	특정 요소 변경
배열 복사	slice(), [...arr]	배열 일부 또는 전체 복사
배열 정렬	sort(), reverse()	정렬 및 역순
검색	find(), includes()	특정 요소 찾기

배열

스프레드 연산자

- 세 개의 점(...)으로 표현 - ES6(ECMAScript 2015)에서 도입
- 배열이나 객체의 얕은 복사, 병합
- 배열이나 객체를 개별 요소로 분리함수 호출 시 인수로 배열 전달

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1]; // arr1의 복사본 생성
console.log(arr2); // [1, 2, 3]
```

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5];
const merged = [...arr1, ...arr2]; //배열 병합
console.log(merged); // [1, 2, 3, 4, 5]
```

```
const arr = [1, 2, 3];
// source 배열에 새로운 요소 추가, 새 배열 생성
const newArr = [...arr, 4, 5];
console.log(newArr); // [1, 2, 3, 4, 5]
```

속성과 메소드

스프레드 연산자

- 배열을 개별 인수로 분리하여 함수에 전달

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];
console.log(sum(...numbers)); // 6
```

속성과 메소드

배열

- Destructuring 배열이나 객체의 구조를 분해하여 변수에 손쉽게 할당하는 문법입니다.

```
const [a, b, c] = [1, 2, 3];
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

```
const [a, , c] = [1, 2, 3];
console.log(a); // 1
console.log(c); // 3
```

```
//배열의 값이 없으면 기본값을 설정할 수 있습니다.
const [a = 10, b = 20] = [1];
console.log(a); // 1
console.log(b); // 20
```

속성과 메소드

배열 Destructuring

- 배열이나 객체가 중첩된 경우에도 Destructuring 을 사용할 수 있습니다

```
// 나머지 요소 수집
const [a, ...rest] = [1, 2, 3, 4];
console.log(a);    // 1
console.log(rest); // [2, 3, 4]
```

```
const numbers = [1, [2, 3]];
const [a, [b, c]] = numbers;
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- 함수의 매개변수로 전달된 객체나 배열을 Destructuring 하여 간편하게 사용할 수 있습니다.

```
function sum([a, b]) {
  return a + b;
}

console.log(sum([1, 2]));
```

JSON 객체

JSON으로 객체 정의 & 생성

- JSON 객체는 중첩된 구조를 가질 수 있으며, 속성값으로 문자열, 숫자, 배열, 또 다른 객체 등이 정의 될 수 있습니다.

```
const person = {
  name: “HGKim",
  age: 25,
  address: {
    city: "Seoul",
    country: "Korea"
  }
};

console.log(person);
```

- 점(.) 또는 대괄호([]) 표기법을 사용하여 속성 추가

```
person.email = "HGKim@example.com";  // 점 표기법
person["phone"] = "010-1234-5678";   // 대괄호 표기법

console.log(person);
```

- JSON으로 객체를 생성하면 동일한 기능의 메서드를 객체의 속성으로 메모리에 생성함

객체 기본

JSON으로 객체 정의 & 생성

- Object.assign()을 사용하면 기존 객체를 변경할 수도 있고, 새로운 객체를 생성할 수도 있습니다.

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const merged = Object.assign({}, obj1, obj2);
console.log(merged); // { a: 1, b: 2 }
```

- Spread 연산자 (...)를 사용하여 기존 객체를 변경하지 않고 새로운 객체를 만들 수 있습니다

```
const newPerson = { ...person, nationality: "Korean" };

console.log(newPerson);
```

객체 기본

JSON으로 객체의 속성 변경

```
person.age = 26; // 점 표기법
person["address"]["city"] = "Busan"; // 대괄호 표기법

console.log(person);
```

- Object.keys()와 map()을 사용하여 특정 조건에 따라 속성 값 변경
- Object.entries()를 사용하여 객체를 배열로 변환 후 수정할 수 있습니다.

```
const updatedPerson = Object.fromEntries(
  Object.entries(person).map(([key, value]) =>
    key === "company" ? [key, "NewCorp"] : [key, value]
  )
);

console.log(updatedPerson);
```

객체 기본

JSON으로 객체의 속성 삭제

- delete 연산자로 속성을 제거할 수 있지만, 객체의 메모리 공간을 유지합니다.

```
delete person.phone;
console.log(person);
```

- Object.keys()와 reduce()를 사용하여 속성 삭제

```
const { company, ...newObj } = person; // company 속성 제거
console.log(newObj);
```

- Spread 연산자를 사용하면 특정 속성을 삭제하면서 새로운 객체를 만들 수 있습니다.
- Object.entries()와 filter()를 조합하면 특정 속성을 제외하는 방식으로 객체를 정리할 수 있습니다.

```
const filteredPerson = Object.fromEntries(
  Object.entries(person).filter(([key]) => key !== "email")
);

console.log(filteredPerson);
```

객체 기본

- **JSON으로 객체 순회**
  - 객체에 for in 반복문을 적용

```
const user = {
  name: "Na",
  age: 30,
  address: {
    city: "Seoul",
    country: "Korea"
  }
};

for (let key in user) {
  if (typeof user[key] === "object") {
    console.log(`${key}:`);
    for (let subKey in user[key]) {
      console.log(`  ${subKey}: ${user[key][subKey]}`);
    }
  } else {
    console.log(`${key}: ${user[key]}`);
  }
}
```

객체 기본

- **for...in vs Object.keys() vs Object.entries()**
  - Object.keys()는 객체의 키 배열을 반환하며, forEach()와 함께 사용

```
Object.keys(person).forEach(key => {
  console.log(`${key}: ${person[key]}`);
});
```

  - Object.entries()는 [key, value] 쌍의 배열을 반환하여 forEach()와 함께 사용

```
Object.entries(person).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

  - for...in 사용 시 프로토타입 상속된 속성도 포함됨, hasOwnProperty()를 사용하여 객체의 자체 속성만 순회하도록 필터링

```
const obj = Object.create({ inherited: "yes" });
obj.ownProp = "hello";
for (let key in obj) {
  console.log(key);
}
for (let key in obj) {
  if (obj.hasOwnProperty(key))  console.log(key); // "ownProp"만 출력됨
  }
}
```

객체 기본

- **this**
  - 현재 실행 문맥(execution context)을 참조하는 키워드
  - 실행되는 방법과 위치에 따라 this가 참조하는 값이 달라집니다.
    - 현재 실행 중인 함수가 속한 객체를 참조
    - 브라우저 환경의 전역에서 전역 객체인 window를 참조
    - Node.js 환경에서는 global 객체를 참조
    - 일반 함수에서 this는 전역 객체를 참조
    - 클래스의 메서드에서는 this가 클래스의 인스턴스를 참조
    - 생성자 함수에서는 this가 새로 생성된 객체를 참조
    - 화살표 함수는 상위 스코프의 this를 그대로 사용합니다.
    - 화살표 함수는 자신만의 this를 가지지 않으므로, this는 정의된 위치의 문맥에 따라 결정됩니다.

객체 기본

- **this**

```
console.log(this); // 브라우저: window 객체, Node.js: {}
```

```
function showThis() {
  console.log(this); // 브라우저: window, strict mode: undefined
}
showThis();
```

```
const obj = {
  name: "Korea",
  greet() {
    console.log(this.name);
  },
};
obj.greet();
```

```
function Person(name) {
  this.name = name;
}
const person = new Person("Korea");
console.log(person.name);
```

객체 기본

- **this**

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}
const alice = new Person("Korea");
alice.greet();
```

```
const obj = {
  name: "Korea",
  greet: () => {
    console.log(this.name); // undefined (전역 객체의 this를 참조)
  },
};
obj.greet();
```

객체 기본

- **this**

```
function Outer() {
  this.name = "Outer";
  const inner = () => {
    console.log(this.name); // Outer (상위 스코프의 this 참조)
  };
  inner();
}
new Outer();
```

객체 기본

- **this**
  - call()과 apply()는 함수를 즉시 실행하며, 주어진 객체를 this로 사용합니다.
  - 함수의 인수는 콤마로 나열하여 전달합니다.

```
const numbers = [10, 20, 30, 5];

const max1 = Math.max.apply(null, numbers);
const max2 = Math.max.call(null, ...numbers); // spread 연산자 사용

console.log(max1);
console.log(max2);
```

  - call()과 apply()는 첫 번째 인자로 this를 지정할 객체를 받습니다.
  - call(thisArg, arg1, arg2, ...) → 인수를 개별적으로 전달
  - apply(thisArg, [arg1, arg2, ...]) → 인수를 배열로 전달

객체 기본

- **this**
  - bind()는 새로운 함수를 반환하여 나중에 실행할 수 있도록 합니다.
  - 인수를 개별적으로 전달

```
const user = { name: “Korea" };

function sayHi() {
  console.log(`Hi, ${this.name}`);
}

const boundFunction = sayHi.bind(user);
boundFunction();
```

객체 기본

- **this**
  - 생성자 함수에서 상속 구현
  - call()을 사용하여 부모 생성자(Person)의 속성을 자식(Student) 객체에 상속

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

function Student(name, age, grade) {
  Person.call(this, name, age);
  this.grade = grade;
}

const student1 = new Student("KoreaKim", 20, "A");
console.log(student1);
```

객체 기본

객체의 복사 & 병합

- 스프레드 연산자를 사용하여 객체를 개별 요소로 분리, 객체들을 병합하거나 복사하는 데 사용할 수 있습니다. (ES6(ECMAScript 2015)에서 도입)

```
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1 };     // obj1의 복사본 생성
console.log(obj2);            // { a: 1, b: 2 }
```

```
const obj1 = { a: 1, b: 2 };
const obj2 = { b: 3, c: 4 };
const merged = { ...obj1, ...obj2 };
console.log(merged); // { a: 1, b: 3, c: 4 } ,  obj2의 b가 obj1의 b를 덮어씀
```

```
const obj = { a: 1, b: 2 };
const newObj = { ...obj, c: 3 };  //새 속성 추가
console.log(newObj);              // { a: 1, b: 2, c: 3 }
```

객체 기본

객체의 복사 & 병합

- Destructuring

```
const person = { name: “Akasha", age: 25 };
const { name, age } = person;
console.log(name);
console.log(age);
```

```
//객체 속성 이름변경
const person = { name: " Akasha", age: 25 };
const { name: userName, age: userAge } = person;
console.log(userName);
console.log(userAge);
```

```
//객체에 없는 속성을 기본값으로 설정
const person = { name: " Akasha" };
const { name, age = 30 } = person;
console.log(name);
console.log(age);
```

객체 기본

객체의 복사 & 병합

- 배열이나 객체가 중첩된 경우에도 Destructuring 을 사용할 수 있습니다.
- 함수의 매개변수로 전달된 객체나 배열을 Destructuring 하여 간편하게 사용할 수 있습니다.

```
const person = {
  name: "Akasha",
  address: {
    city: "New York",
    zip: 10001
  }
};
const { address: { city, zip } } = person;
console.log(city); // New York
console.log(zip);  // 10001
```

객체 기본

객체의 복사 & 병합

- 함수의 매개변수로 전달된 객체나 배열을 Destructuring 하여 간편하게 사용

```
function greet({ name, age }) {
  console.log(`Hello, ${name}. You are ${age} years old.`);
}

const person = { name: "Alice", age: 25 };
greet(person); // Hello, Alice. You are 25 years old.
```

생성자 함수와 프로토타입

생성자 함수

- 객체를 만드는 함수, 대문자로 시작하는 이름 사용
- 생성자 함수로 만든 객체는 prototype 공간(공유 영역)에 메소드를 지정해서 생성자 함수로부터 생성된 모든 객체가 공유 하도록 함 (prototype 속성으로 참조)

```
// 생성자 함수
function Product(name, price) {
    this.name = name ;
    this.price = price ;
}

Product.prototype.print = function(){
    console.log(`${product.name}의 가격은 ${product.price}원입니다.`)
};
// 객체 생성
let  product = new Product("바나나" , 1200);

console.log(product);
product.print()
```

생성자 함수와 프로토타입

프로토타입(prototype)

- javascript는 객체를 복사하여 새로운 객체를 생성하는 prototype 기반의 언어.
- 모든 객체는 다른 객체로부터 속성과 메서드를 상속받을 수 있습니다
- 생성된 객체는 또 다른 객체의 원형이 될 수도 있다.
- JavaScript의 객체는 생성될 때 자동으로 숨겨진 링크([[Prototype]]으로 표현)를 가집니다.
- [[Prototype]] 링크는 다른 객체를 참조하며, 이 참조를 통해 상속을 구현합니다.
- 함수 객체는 특별히 prototype 속성을 가지며, [[Prototype]]을 통해 생성된 객체와 연결된다.
- 객체가 특정 속성이나 메서드를 찾으려고 할 때, 해당 객체에 직접 존재하지 않으면 프로토타입 체인을 따라가며 찾습니다
- 체인의 끝에 도달했는데도 속성을 찾지 못하면 undefined를 반환한다.
- 모든 함수의 프로토타입은 'constructor' 프로퍼티를 기본으로 가집니다

prototype-based 언어 : 객체 원형인 프로토타입을 이용해 새로운 객체를 만들어준다

null 자료형

null 자료형

- '값이 없는 상태'를 구분할 때 null을 활용

```
console.log(null);
console.log(typeof(null));

let  zeroNumber = 0;
let  falseBoolean = '';
let  emptyString = '';
let  undefinedValue ;
let  nullValue = null

if ( zeroNumber == null)  console.log('0은 존재하지 않는 값입니다');
if ( falseBoolean == null)  console.log('false는 존재하지 않는 값입니다');
if ( emptyString == null)  console.log('빈 문자열은 존재하지 않는 값입니다');
if ( undefinedValue== null)  console.log('undefined은 존재하지 않는 값입니다');
if ( nullValue  == null)  console.log('null은 존재하지 않는 값입니다');
```

class

- **class**
  - ES6 new feature
  - 생성자를 통해 인스턴스가 만들어지고 메서드나 프로퍼티는 인스턴스를 통해 접근해서 사용

```
class User {
 constructor(name, age, gender) {
    this.name = name;
    this.age =age;
    this.gender = gender;
  }
  getName() {
    return this.name;
  }
  getAge() {
    return this.age;
  }
  getGender() {
    return this.gender;
  }
}
```

```
const user1 = new User('kang', 33, 'M');
console.log(user1.getName());
console.log(user1.getAge());
console.log(user1.getGender());
```

class

생성자 함수

- 순수함수로 객체를 정의 : 같은 입력에 대해 항상 같은 결과를 반환한다.
- 외부 상태를 변경하지 않는다(부작용이 없다).

```
const User = function (name, age, gender) {
   this.name = name;
   this.age = age;
   this.gender = gender;
}
User.prototype.getName = function () {
    return this.name;
}
User.prototype.getAge = function () {
    return this.age;
}
User.prototype.getGender = function () {
     return this.gender;
}
```

class

- ## 상속 with 생성자 함수
  - child프로토타입이름 .prototype = new 부모();

```
function Animal(name){
    this.speed = 0;
    this.name = name;
}
Animal.prototype.run = function(speed){
     this.speed = speed;
     console.log(`${this.name} 은/는 속도 ${this.speed}로 달립니다.`);
  }
Animal.prototype.stop = function(){
    this.speed = 0;
    console.log(`${this.name} 이/가 멈췄습니다.`);
}
```

```
function Cat(age, type, name) {
    this.age = age;
    this.type = type;
}

Cat.prototype = new Animal('나비');
```

class

class

- class로 만든 함수엔 특수 내부 프로퍼티인 [[IsClassConstructor]] : true 가 존재
- 클래스 생성자를 new와 함께 호출하지 않으면 에러가
- 클래스 생성자를 문자열로 형변환 하면 'class'로 시작하는 문자열이 됩니다
- 클래스에 정의된 메서드들은 열거할 수 없다(non-enumerable).
  - 클래스의 prototype 프로퍼티에 추가된 메서드의 enumerable 플래그는 false이다.
  - for-in으로 객체를 순회할 때 메서드는 순회의 대상에서 제외됩니다
- 클래스는 항상 엄격모드로 실행된다.(use strict)

```
class Person {
  constructor(name) {
    this.name = name;
  }

  greet() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

class

- **class**

```
try {
            Person("Korea"); // TypeError
      } catch (error) {
        console.error(error);
}
console.log(String(Person));

//클래스 메서드는 열거할 수 없음 (non-enumerable)
console.log(Object.keys(Person.prototype));
console.log(Object.getOwnPropertyNames(Person.prototype));

// prototype에 추가된 메서드는 enumerable이 false
const descriptor = Object.getOwnPropertyDescriptor(Person.prototype,
"greet");
console.log(descriptor.enumerable);

// for-in 순회에서 클래스 메서드는 제외됨
const person = new Person("Bob");
for (let key in person) {
        console.log(key);
}
```

class

- **Prototype Based Language**
  - 클래스 기반 언어는 클래스 내부에 모든 속성과 메소드를 정의합니다.
  - 클래스를 기반으로 생성된 인스턴스는 클래스 내부에 정의 되어 있는 속성과 메소드에 접근하여 사용할 수 있는 형태이다
  - 모든 인스턴스들은 메소드와 속성들을 상속받기 위한 명세로 프로토타입 객체를 가진다
  - 클래스의 인스턴스는 클래스에 정의된 속성과 메소드에 접근할 수 있으며 프로토타입의 속성과 메소드에도 접근할 수 있다

```
const Hello = function(name) {
  this.name = name;
}
```

```
▼ Hello {name: 'hello'} ⓘ
    name: "hello"
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ (name)
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
      ▶ __proto__: Object
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
```

생성자 함수와 프로토타입

프로토타입(prototype)

- 모든 객체는 기본적으로 Object.prototype 을 상속 받습니다.

```
//Object.create(proto1)는  인수로 전달된 객체를 프로토타입으로 하는 객체를 생성합니다.
const proto = {
  greet: function() {
      console.log('Hello!');
 }
};
const obj = Object.create(proto);
obj.greet();      // "Hello!"
```

```
//Object.setPrototypeOf(obj, proto2)로 obj의 프로토타입을 proto2로 변경합니다
const proto2 = {
 greet: function() {
  console.log("Hello from proto2!");
 }
};

Object.setPrototypeOf(obj, proto2);
obj.greet();
```

생성자 함수와 프로토타입

- **프로토타입(prototype)**

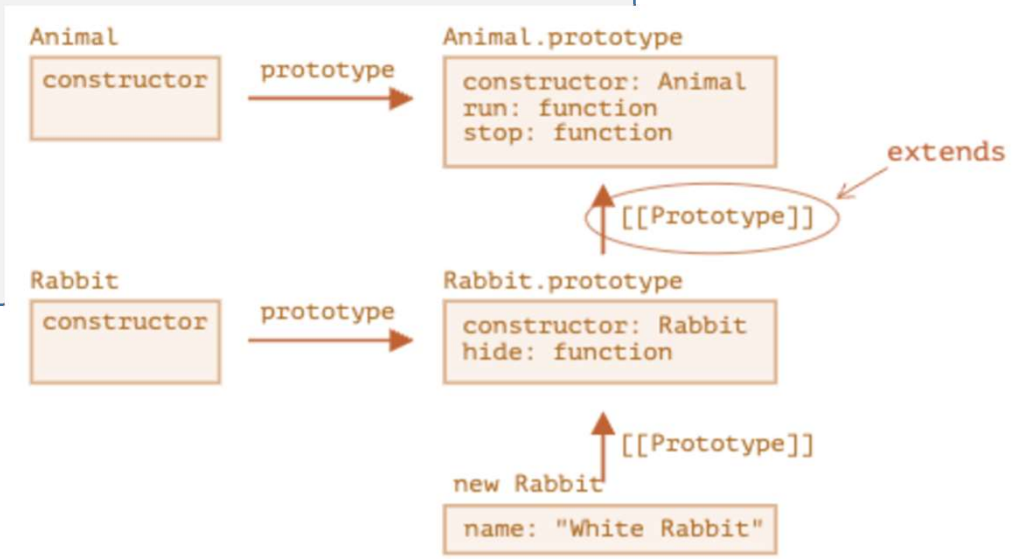
```
console.log(Object.getPrototypeOf(obj) === proto2); // true
console.log(Object.getPrototypeOf(obj) === proto1); // false
```

class

상속

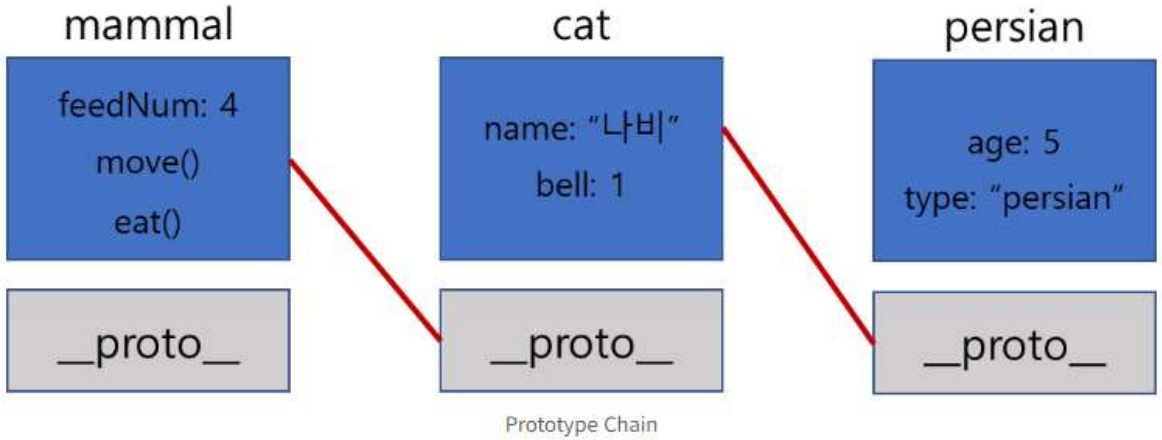
- extends는 프로토타입을 기반으로 동작한다.
- extends는 Rabbit.prototype.[[Prototype]]을 Animal.prototype으로 설정한다.
  Rabbit.prototype에서 메서드를 찾지 못하면 Animal.Prototype에서 메서드를 가져온다.
- 자식클래스의 메서드는 부모클래스의 메서드와 Prototype Chain으로 연결됨

```
Class  Rabbit  extends  Animal {
  hide() {
    alert(`${this.name} 이/가 숨었습니다!`);
  }
}

let rabbit = new Rabbit("흰 토끼");

rabbit.run(5);
rabbit.hide();
```



class

- **상속**
  - prototype을 이용해서 상속



Prototype Chain

```
const persian = {
  age: 5,
  type: "persian",
};

persian.__proto__ = cat;
```

class

- **상속**
  - prototype을 이용해서 상속

```
const mammal = {
  feetNum: 4,
  move() {
    console.log("움직이다");
  },
  eat() {
    console.log("먹다");
  },
}

const cat = {
  name: "나비",
  bell: 1,
};

const dog = {
  name: "하루",
}
```

```
const cow = {
  name: "아지",
}

cat.__proto__ = mammal;
dog.__proto__ = mammal;
cow.__proto__ = mammal;

cat.move();
cat.eat();
console.log(cat.feetNum);

dog.move();
dog.eat();
console.log(dog.feetNum);

cow.move();
cow.eat();
console.log(cow.feetNum);
```

class

- **override**
  - 상속 관계에 있는 부모 클래스에서 이미 정의된 메소드를 자식 클래스에서 동일 시그니쳐를 갖는 메소드로 다시 정의하는 것
  - 메소드 오버라이딩을 할때는 super 라는 키워드를 사용
  - super.method(...) : 부모 클래스에 정의된 method를 호출
  - super(...) : 부모 생성자를 호출하는데, 자식 생성자 내부에서만 사용

```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name}가 숨었습니다!`);
  }

  stop() {
    super.stop();
    this.hide();
  }
}
```

class

- **생성자 override**
  - 다른 클래스를 상속받고 constructor가 없는 경우에는 constructor가 만들어진다

```
class Animal {            // 부모 클래스
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {

}

const dog = new Dog("Buddy");
console.log(dog.name);
dog.speak();

// 기본 생성자가 자동으로 만들어진 것 확인
console.log(Dog.prototype.constructor === Dog); // true
console.log(Object.getOwnPropertyNames(Dog.prototype)); // ["constructor"]
```

class

생성자 override

- 상속 클래스의 생성자에서는 반드시 super(...)를 먼저 호출해야 합니다.
- super(...)는 this를 사용하기 전에 반드시 호출해야 합니다.
- 상속 클래스의 생성자 함수(derived constructor)'와 그렇지 않은 생성자 함수를 구분한다.
- 상속 클래스의 생성자 함수엔 특수 내부 프로퍼티인 [[ConstructorKind]]: " drived " 가 생성됩니다
- 일반 클래스가 new와 함께 실행되면 빈 객체가 만들어지고 this에 이 객체를 바인딩된다
- 상속 클래스의 생성자 함수는 빈 객체를 만들고 this에 이 객체를 할당하는 일을 부모의 클래스 생성자가 처리한다

class

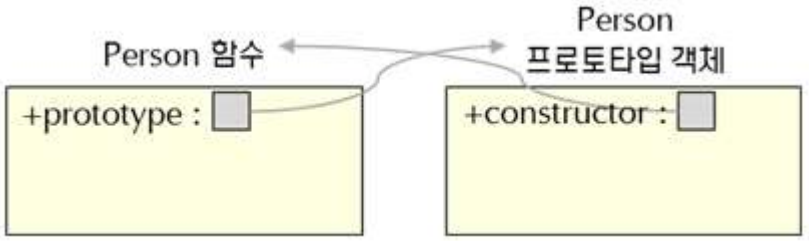
- **생성자 override**

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  // ...
}

class Rabbit extends Animal {
  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }
  // ...
}

let rabbit = new Rabbit("흰 토끼", 10);
```

class

- **Prototype**
  - JavaScript는 기존의 객체를 복사하여(closing) 새로운 객체를 생성하는 프로토타입 기반의 언어입니다.
  - 프로토타입 기반 언어는 객체 원형인 프로토타입을 이용하여 새로운 객체를 만들어냅니다
  - 생성된 객체는 또 다른 객체의 원형이 될 수 있습니다
  - 프로토타입은 객체를 확장하고 객체 지향적인 프로그래밍을 할 수 있게 해줍니다.
  - 프로토타입 객체를 참조하는 prototype 속성과 객체 멤버인 proto 속성이 참조하는 숨은 링크가 있습니다.

  - JavaScript에서 함수를 정의하면 파싱단계에서 함수 멤버로 prototype 속성에 다른 곳에서 생성된 함수 이름의 프로토타입 객체를 참조합니다
  - 프로토타입 객체의 멤버인 constructor 속성은 함수를 참조하는 내부구조를 가집니다

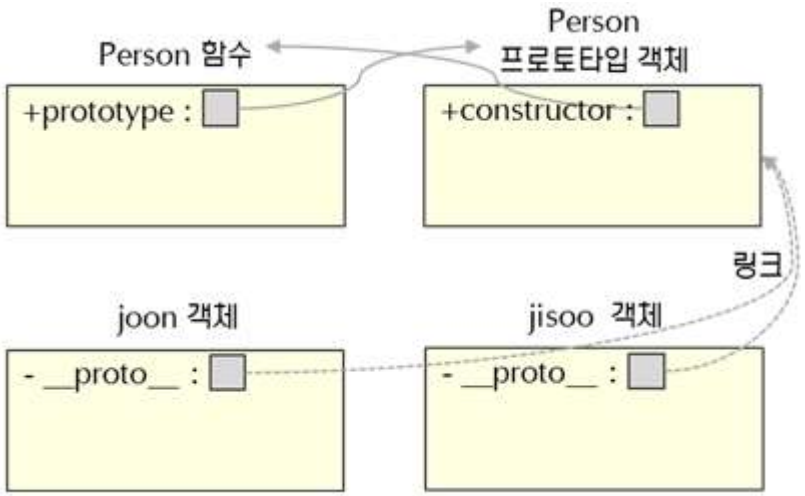


class

Prototype

- 속성이 없는 Person 함수를 정의하면 Person함수 Prototype 속성은 프로토타입 객체를 참조합니다
- 프로토타입 객체 멤버인 constructor 속성은 Person 함수를 참조하는 구조를 가집니ㅏㄷ.
- Person함수의 prototype 속성이 참조하는 프로토타입 객체는 new라는 연산자와 Person 함수를 통해 생성된 모든 객체의 원형이 되는 객체입니다.
- 생성된 모든 객체가 참조합니다
- 객체 안에 proto 속성은 원형인 프로토타입 객체를 숨은 링크로 참조하는 역할을 합니다.

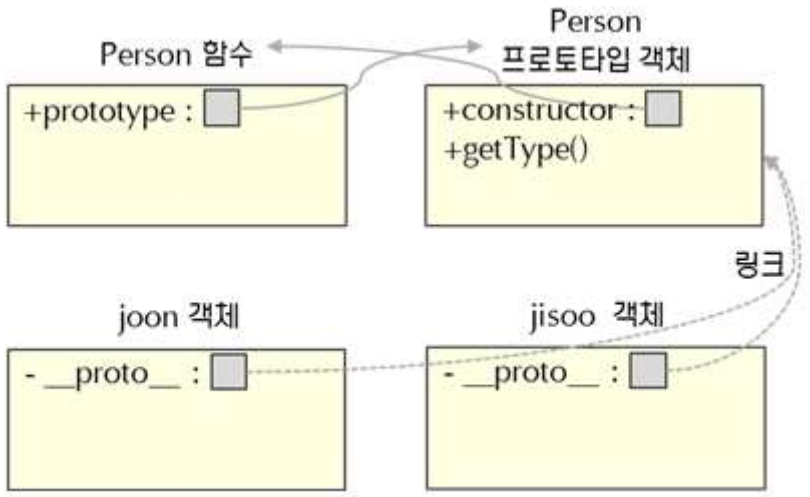
```
function Person(){}

var Joon = new Person();
var jisoo = new Person();
```



class

- **Prototype**
  - 프로토타입 객체에 멤버를 추가, 수정, 삭제할 때는 함수 안의 prototype 속성을 사용해야 합니다.
  - 프로토타입 멤버를 읽을 때는 함수 안의 prototype 속성 또는 객체 이름으로 접근합니다

```
joon.getType = function(){
  return "사람"
}

console.log(joon.getType());
console.log(jisoo.getType());

jisoo.age = 25;
console.log(joon.age);
console.log(jisoo.age);
```

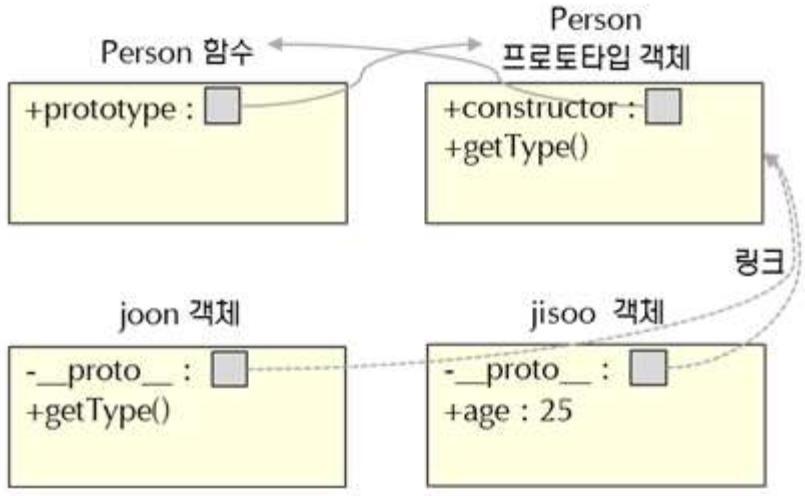


class

Prototype

- 생성된 객체를 이용하여 프로토타입 객체의 멤버를 수정하면 프로토타입 객체에 있는 멤버를 수정하는 것이 아니라 자신의 객체에 멤버를 추가하는 것입니다
- 프로토타입 객체의 멤버를 수정할 경우는 멤버 추가와 함수의 prototype 속성을 이용하여 수정합니다.

```
Person.prototype.getType=function() {
  return "인간"
};
console.log(joon.getType());
console.log(jisoo.getType());
```

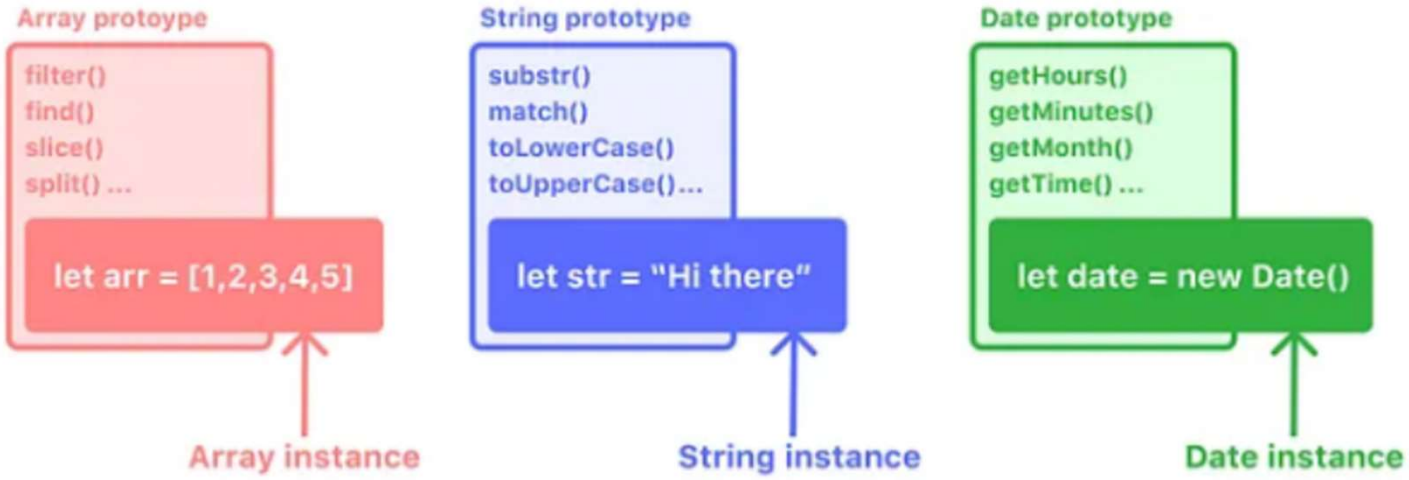


프로토타입 객체는 새로운 객체가 생성되기 위한 원형이 되는 객체입니다
같은 원형으로 생성된 객체가 공통으로 참조하는 공간힙니다.
프로토타입 객체의 멤버를 읽는 경우에는 객체 또는 함수의 prototype 속성을 통해 접그할 수 있습니다.
추가 수정 삭제는 함수의 prototype 속성을 통해 접근해야 합니다.

class

네이티브 객체 프로토타입

- 자바스크립트의 Array, String 등의 내장 객체 역시 자신의 프로토타입을 가지고 있다.




```
const arr = [1,2,3,4,5];
console.log(Object.getPrototypeOf(arr))  //Array prototype
const str = "Hello world!";
console.log(Object.getPrototypeOf(str)) //String prototype
const date = new Date();
console.log(Object.getPrototypeOf(date)) //Date prototype
```

class

Prototype Chain

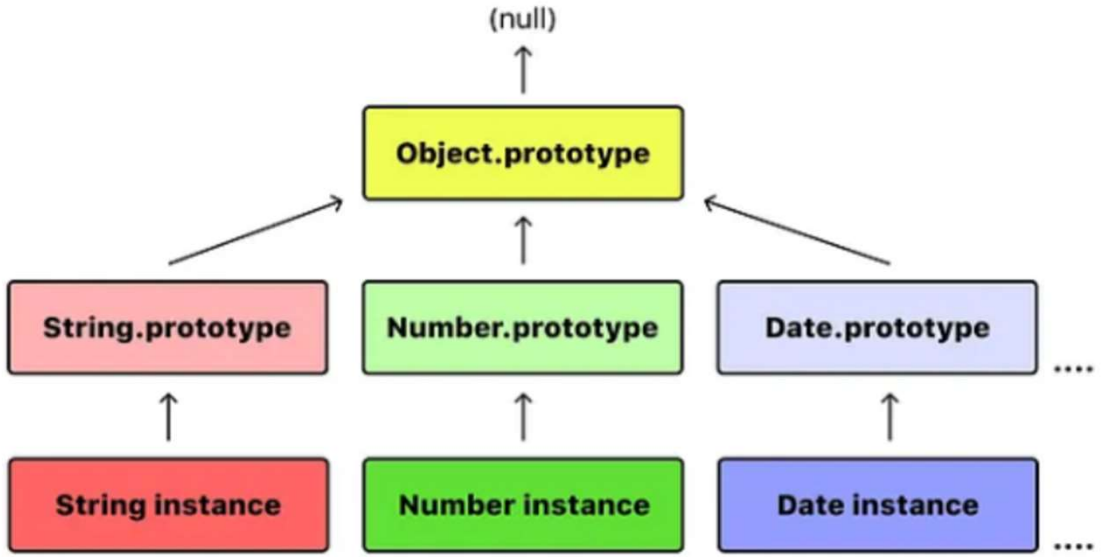
- Prototype 은 객체 생성자 함수에 의해 생성되는 객체들이 공유하는 속성과 메소드를 저장하는 특수 객체이다.
- 상속받는 멤버들은 prototype 속성에 정의되어 있고 상속된다.
- 프로토타입에 정의되지 않은 멤버들은 상속되지 않는다
- Javascript에서 모든 객체는 프로토타입을 공유한다.
- 객체 자체가 스스로 메소드와 속성을 모두 가지는 대신 여러 객체가 동일한 Prototype 을 공유하도록 하면 메모리를 효율적으로 사용할 수 있다.
- JavaScript의 객체는 프로토타입 체인(prototype chain) 을 통해 상속을 구현합니다.
- 객체가 특정 프로퍼티나 메서드를 찾을 때 자신의 프로퍼티에서 먼저 찾고, 없으면 프로토타입을 따라 상위 객체로 탐색합니다.
- 깊은 체인은 프로퍼티 검색 시 여러 단계를 탐색하므로 성능 저하 발생 가능
- 프로토타입 수정 시 모든 하위 객체에 영향을 미치므로 예상치 못한 버그 발생 가능

```
const arr = [1, 2, 3];

console.log(arr.__proto__ === Array.prototype);
console.log(Array.prototype.__proto__ === Object.prototype);
console.log(Object.prototype.__proto__ === null);
```

class

Prototype Chain

- 객체 자신의 것 뿐 아니라 [[Prototype]]가 가리키는 링크를 따라 부모 역할을 하는 모든 프로토 타입 객체의 속성이나 메소드에 접근할 수 있다.
- Prototype Chain 을 기반으로 다른 객체에 정의된 메소드와 속성을 한 객체에서 사용할 수 있게 해준다.




```
const obj = {hello: 'world'};
const str = 'hello'
Object.prototype.hi = function() {console.log('hi')};
obj.hi(); // hi
str.hi(); // hi
```

class

Prototype Chain

- 특정 객체에서 특정 속성이나 메소드에 접근할 때 자바스크립트 엔진은 객체의 속성과 메서드를 탐색하고 없는 경우 Prototype Chain 이 일어나며
- 부모 역할이 되는 상위 객체의 속성이나 메소드를 탐색해 나가고 마지막에는 Object.prototype까지 탐색합니다.
- 최상위의 Object.prototype에서도 찾지 못하게 된다면 undefined를 리턴한다.

```
const num = 55;
num.push // undefined;
num.push() // Uncaught TypeError: num.push is not a function
Object.prototype.push = function() {return this + 1};
num.push() // 56;
```

static property와 method

static property

- 인스턴스가 아닌 클래스 자체에서 관리해야 하는 데이터나 기능을 정의할 때 사용
- "prototype"이 아닌 클래스 자체에 속하는 프로퍼티나 메서드를 정의
- 인스턴스를 생성하지 않아도 static property와 method 에 접근할 수 있습니다.
- 클래스 내부의 모든 인스턴스가 공통적으로 사용하는 데이터는 static property로 선언하면 불필요한 메모리 낭비를 줄일 수 있습니다.

```
class AppConfig {
  static appName = "My Application";
  static version = "1.0.0";
}

console.log( AppConfig.appName );
console.log( AppConfig.version );
```

static property와 method

static method

- 클래스의 인스턴스 상태에 의존하지 않는 범용적인 기능을 제공할 때 사용
- 특정 작업(예: 포매팅, 변환)을 수행하는 유틸리티 메서드를 정적으로 정의합니다.
- 클래스의 모든 인스턴스가 공유합니다.

```
class MathUtil {
  static square(num) {
    return num * num;
  }

  static cube(num) {
    return num * num * num;
  }
}

console.log( MathUtil.square(4) );
console.log( MathUtil.cube(3) );
```

static property와 method

- **static property와 method 활용**
  - static property를 사용하여 하나의 클래스에서 하나의 인스턴스만 존재하도록 보장하는 싱글톤 패턴을 구현할 수 있습니다
  - 데이터베이스 연결 관리, API 요청 관리, App 전역 상태 관리를 할 수 있습니다.

```
class AppState {
  static theme = "light";

  static setTheme(newTheme) {
    AppState.theme = newTheme;
  }

  static getTheme() {
    return AppState.theme;
  }
}

console.log(AppState.getTheme());
AppState.setTheme("dark");
console.log(AppState.getTheme());
```

static property와 method

- **static property와 method 활용**

```
class Singleton {
  static instance = null;

  static getInstance() {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }

  showMessage() {
    console.log("Hello, Singleton!");
  }
}

const s1 = Singleton.getInstance();
const s2 = Singleton.getInstance();

console.log(s1 === s2);
s1.showMessage();
```

static property와 method

static property와 method 상속

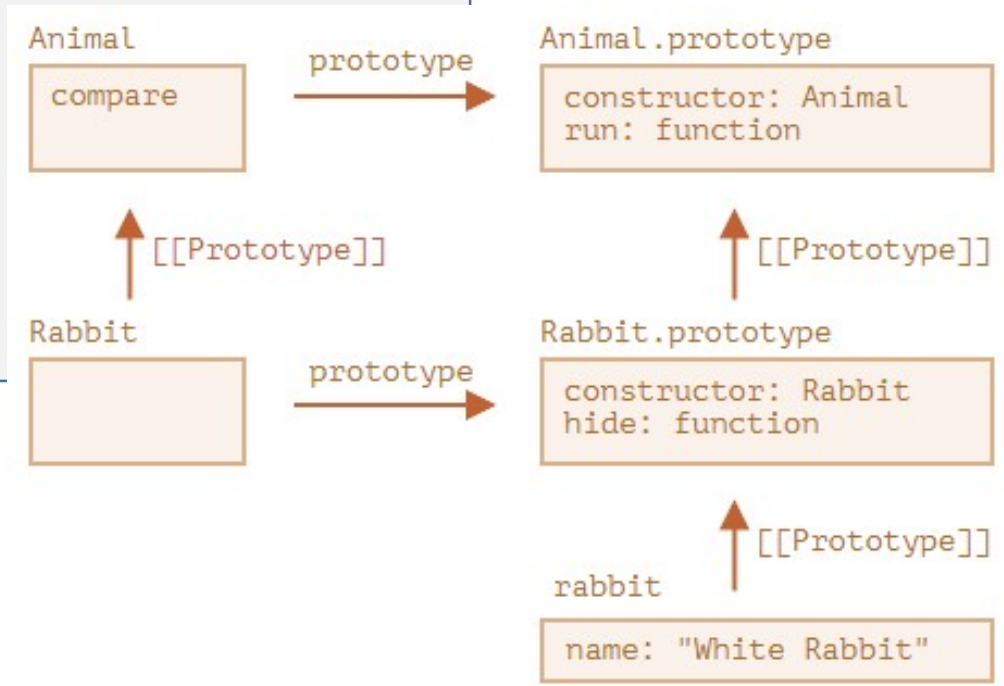
- static property와 method 는 상속됩니다.

```
class Animal {
 static planet = "지구";

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
 console.log(`${this.name}가 속도 ${this.speed}로 달립니다.`);
  }

  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}
```

static property와 method

- **static property와 method 상속**

```
class Rabbit extends Animal {
  hide() {
 console.log(`${this.name}가 숨었습니다!`);
  }
}

let rabbits = [
  new Rabbit("흰 토끼", 10),
  new Rabbit("검은 토끼", 5)
];

rabbits.sort( Rabbit.compare );
rabbits[0].run

alert(Rabbit.planet);
```



static property와 method

- **static property와 method 상속**

```
class Animal {}
class Rabbit extends Animal {}
// 정적 메서드
alert(Rabbit.__proto__ === Animal); // true
// 일반 메서드
alert(Rabbit.prototype.__proto__ === Animal.prototype); // true
```

객체 생성 실습

객체 생성 및 속성 다루기

- user라는 객체를 생성합니다.
- name, age, email이라는 속성을 추가합니다.
- greet라는 메서드를 추가하여 "안녕하세요, 제 이름은 [name]입니다."를 출력하도록 합니다.

```
const user = {
   // 여기에 속성과 메서드를 추가하세요
};

console.log(user.name);  // 예상 출력: (설정한 이름)
user.greet();    // 예상 출력: 안녕하세요, 제 이름은 [name]입니다.
```

객체 생성 실습

객체 속성 관리

- 객체 book을 생성합니다.
- title, author, price 속성을 추가합니다.
- discount 메서드를 추가하여, price를 입력된 할인율만큼 줄입니다.
- price는 0 이하로 내려가지 않도록 설정합니다.

```
const book = {
   // 여기에 코드 작성
};

// 사용 예
book.price = 100; // 책 가격 설정
book.discount(20); // 20% 할인
console.log(book.price); // 예상 출력: 80
```

객체 생성 실습

- **클래스 생성**
  - Rectangle이라는 클래스를 생성합니다.
  - 생성자는 width와 height를 입력받아 초기화합니다.
  - area() 메서드를 추가하여 넓이를 반환합니다.
  - perimeter() 메서드를 추가하여 둘레를 반환합니다.

```
class Rectangle {
   // 여기에 코드 작성
}

// 사용 예
const rect = new Rectangle(10, 5);
console.log(rect.area()); // 예상 출력: 50
console.log(rect.perimeter()); // 예상 출력: 30
```

객체 생성 실습

클래스 상속

- Animal 클래스를 생성하고 name과 sound를 속성으로 받는 생성자를 만듭니다.
- speak() 메서드를 추가하여 "[name]가 [sound] 소리를 냅니다."를 출력합니다.
- Dog 클래스를 Animal 클래스를 상속받아 만듭니다.
- Dog 클래스에 fetch() 메서드를 추가하여 "개가 물건을 가져옵니다."를 출력합니다.

```
class Animal {
   // 여기에 코드 작성
}

class Dog  {
   // 여기에 코드 작성
}

// 사용 예
const dog = new Dog("뽀삐", "멍멍");
dog.speak(); // 예상 출력: 뽀삐가 멍멍 소리를 냅니다.
dog.fetch(); // 예상 출력: 개가 물건을 가져옵니다.
```

객체 생성 실습

- **클래스와 접근 제어**
  - BankAccount 클래스를 생성합니다.
  - balance 속성은 private으로 선언합니다.
  - deposit(amount) 메서드를 통해 예금합니다.
  - withdraw(amount) 메서드를 통해 출금합니다.
  - getBalance() 메서드를 통해 현재 잔액을 반환합니다.

```
class BankAccount {
   // 여기에 코드 작성
}

// 사용 예
const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // 예상 출력: 100
account.withdraw(50);
console.log(account.getBalance()); // 예상 출력: 50
```

객체 생성 실습

- **정적 메서드와 속성**
  - MathUtils라는 클래스를 생성합니다.
  - static 메서드 add(a, b), subtract(a, b)를 추가합니다.
  - 정적 속성 PI를 추가하여 3.14159를 저장합니다.

```
class MathUtils {
   // 여기에 코드 작성
}

// 사용 예
console.log(MathUtils.add(10, 5)); // 예상 출력: 15
console.log(MathUtils.subtract(10, 5)); // 예상 출력: 5
console.log(MathUtils.PI); // 예상 출력: 3.14159
```

객체 생성 실습

- **클래스와 다형성**
  - Shape라는 클래스를 생성하고 area() 메서드를 정의합니다.
  - Circle 클래스와 Square 클래스를 Shape 클래스를 상속받아 구현합니다.
  - Circle은 반지름(radius)을 기반으로 넓이를 계산합니다.
  - Square는 한 변의 길이(side)를 기반으로 넓이를 계산합니다.
  - 각 클래스의 area() 메서드는 해당 도형의 넓이를 반환합니다.

```
class Shape {
   // 여기에 코드 작성
}
class Circle  {
   // 여기에 코드 작성
}
class Square  {
   // 여기에 코드 작성
}
// 사용 예
const circle = new Circle(5);
console.log(circle.area()); // 예상 출력: 78.53975 (PI * r^2)
const square = new Square(4);
console.log(square.area()); // 예상 출력: 16 (side^2)
```