자바스크립트 함수

자바스크립트에서 함수는 일급 객체로 취급됩니다. 이는 함수가 변수에 할당되거나, 다른 함수의 매개변수로 전달되거나, 함수에서 반환될 수 있다는 것을 의미합니다. 함수는 코드 블록을 나타내며, 특정 작업을 수행하는 JavaScript의 핵심 구성 요소 중 하나입니다.

특징	설명
재사용성 (Reusability)	함수는 특정 작업을 수행하는 코드 블록으로, 여러 번 호출될 수 있습니다. 이를 통해 동일한 작업을 여러 곳에서 재사용할 수 있습니다.
모듈화 (Modularity)	함수를 사용하여 코드를 모듈화할 수 있습니다. 관련된 작업을 수행하는 함수를 그룹화하여 코드의 가독성을 높이고 유지 관리를 용이하게 할 수 있습니다.
매개변수 (Parameters)와 반환값 (Return Value)	함수는 매개변수를 통해 입력값을 받아들일 수 있으며, 이를 처리한 후 결과를 반환할 수 있습니다.
스코프 (Scope)	함수는 자신만의 스코프를 가집니다. 이는 함수 내에서 선언된 변수는 함수 외부에서 접근할 수 없으며, 함수 외부에서 선언된 변수는 함수 내에서 접근할 수 없다는 것을 의미합니다.
재귀 (Recursion)	함수는 자신 내부에서 자기 자신을 호출할 수 있습니다. 이를 통해 재귀적인 알고리즘을 구현할 수 있습니다.

함수 종류

종류	설명	예시
함수 선언식 (Function Declaration)	함수를 정의하고 이름을 지정하는 방식으로, `function` 키워드를 사용하여 함수를 선언합니다.	`function bread() {}`
함수 표현식 (Function Expression)	변수에 함수를 할당하는 방식으로, 익명 함수를 만들고 변수에 할당하여 사용합니다.	`const bread = function() {}`
화살표 함수 (Arrow Function)	ES6에서 도입된 간결한 문법으로, `=>` 기호를 사용하여 함수를 선언합니다.	`const bread = () => {}`
익명 함수 (Anonymous Function)	이름이 없는 함수로, 함수 표현식이나 콜백 함수로 주로 사용됩니다.	`setTimeout(function() { console.log('Hello') }, 1000)`
즉시 실행 함수 (Immediately Invoked Function Expression, IIFE)	정의되자마자 즉시 실행되는 함수로, 함수를 선언하고 즉시 괄호로 둘러싸서 호출합니다.	`(function() { console.log('I am invoked immediately') })()`

```
// 함수 선언식 (Function Declaration)
function bread() {
  console.log("This is a bread function declaration");
}
```

```
// 함수 표현식 (Function Expression)
const breadFunc = function() {
  console.log("This is a bread function expression");
};

// 화살표 함수 (Arrow Function)
const breadArrow = () => {
  console.log("This is a bread arrow function");
};

// 익명 함수 (Anonymous Function)
setTimeout(function() {
  console.log('Hello');
}, 1000);

// 즉시 실행 함수 (Immediately Invoked Function Expression, IIFE)
(function() {
  console.log('I am invoked immediately');
})();
```

함수를 사용하면서 자주 등장하는 용어

용어	설명
지역변수	함수 내부에서 선언된 변수로, 해당 함수 내부에서만 접근할 수 있습니다.
전역변수	함수 외부에서 선언된 변수로, 스크립트 전체에서 접근할 수 있습니다.
메서드	객체에 속한 함수를 의미하며, 해당 객체의 속성으로써 동작합니다.
매개변수 (parameter)	함수 안에서의 정의 및 사용에 나열되어 있는 변수들을 의미합니다.
전달 인자 (argument)	함수를 호출할 때 전달되는 실제 값을 의미합니다.
this	현재 실행 중인 코드에서 사용되는 객체를 가리키는 키워드로, 주로 메서드 내부에서 사용됩니다. 화살표 함수에서는 자체적인 this 바인딩을 제공하지 않습니다.

지역변수 (Local Variable)

지역변수(local variable)는 컴퓨터 프로그래밍에서 특정한 함수나 블록 내에서만 정의되고 접근할 수 있는 변수를 말합니다. 이 변수는 그 함수나 블록이 실행될 때 생성되며, 해당 함수나 블록의 실행이 종료되면 소멸합니다.

특성	설명
범위	지역변수는 선언된 함수나 블록 내에서만 접근할 수 있습니다. 함수나 블록 외부에서는 이 변수에 접근할 수 없습니다.
생명주기	지역변수는 해당 함수나 블록이 실행될 때 메모리에 할당되고, 그 실행이 끝나면 메모리에서 해제됩니다.

```
function bakery() {
  const cheeseBread = "치즈빵"; // 함수 내부에서만 접근할 수 있는 지역 변수
  console.log(cheeseBread);
}

bakery(); // 출력: 치즈빵
// console.log(cheeseBread); // 오류: cheeseBread is not defined (지역 변수이므로 함수 외부에서 접근 불가)
```

전역변수 (Global Variable)

전역변수(global variable)는 프로그래밍에서 프로그램 전체에서 접근할 수 있는 변수를 말합니다. 이러한 변수는 프로그램의 어느 곳에서나, 어느 함수 내에서나 접근하고 수정할 수 있습니다.

특성	설명
범위 (Scope)	전역변수는 프로그램 전체에서 접근 가능합니다. 함수나 블록에 상관없이 어디서나 접근할 수 있으며, 프로그램의 어느 부분에서도 해당 변수의 값을 변경하거나 참조할 수 있습니다.
생명주기 (Lifecycle)	전역변수는 프로그램이 시작될 때 생성되고 프로그램이 종료될 때까지 유지됩니다. 따라서, 전역변수는 프로그램의 실행 기간 동안 계속 메모리를 차지합니다.
공유 및 변경 가능성	모든 함수와 블록이 전역변수를 공유하기 때문에, 한 곳에서의 변경이 전체 프로그램에 영향을 미칩니다. 이는 프로그램의 상태 관리를 복잡하게 만들 수 있으며, 예기치 못한 버그의 원인이 될 수 있습니다.

```
const saltBread = "소금빵";

function cafe() {
  console.log(saltBread); // 함수 내부에서 전역 변수에 접근 가능
}

cafe(); // 출력: 소금빵
console.log(saltBread); // 출력: 소금빵 (전역 변수라서 함수 외부에서도 접근 가능)
```

매개변수(parameter)와 전달 인자(argument)

자바스크립트의 독특한 점 중 하나는 매개변수의 개수와 전달 인자의 개수가 일치하지 않아도 된다는 것입니다. 예를 들어, 매개변수가 두 개인 함수에 한 개의 전달 인자만 제공할 수 있으며, 이 경우 누락된 매개변수는 undefined로 처리됩니다. 반대로 매개변수보다 더 많은 전달 인자를 제공할 수도 있는데, 이 경우 함수 내에서 arguments 객체를 통해 접근할 수 있습니다.

종류	설명
매개변수 (Parameter)	자바스크립트 함수를 정의할 때 사용되는 변수들입니다. 이 변수들은 함수가 실행될 때 전달될 데이터의 '형식'과 '이름'을 정의합니다. 함수 내부에서 이 매개변수들은 해당 함수의 로컬 변수처럼 작동합니다. 예를 들어, `function add(x, y) { ... }`에서 `x`와 `y`는 매개변수로, `add` 함수의 정의 부분에 나타납니다.
전달 인자 (Argument)	자바스크립트 함수가 호출될 때 실제로 전달되는 값들입니다. 이 값들은 함수가 실행될 때 매개변수에 할당되어, 함수 내에서 사용됩니다. 예를 들어, `add(5, 10)`에서 `5`와 `10`은 전달 인자로, `add` 함수를 호출할 때 제공됩니다.

```
function shop(bread) {
  // 함수 내에서 사용되는 변수 'bread'는 매개변수(parameter)입니다.
  console.log("맛있는 " + bread + "!");
}

const saltBread = "소금빵";
const chocoBread = "초코빵";
// shop(saltBread)에서 saltBread가 인자입니다.
shop(saltBread); // 맛있는 소금빵!
```

메서드 (Method), this

용어	설명
메서드 (Method)	자바스크립트의 메서드는 객체의 속성이며, 함수 형태로 정의됩니다. 객체에 속한 함수는 해당 객체의 상태에 접근하고, 객체의 행동을 정의할 수 있습니다. 메서드는 주로 객체의 내부 데이터를 처리하거나 객체 상태를 변경하는 데 사용됩니다. 예를 들어, `let obj = { value: 1, increment: function() { this.value++; } };`에서 `increment`는 `obj` 객체의 메서드로, `obj`의 `value` 속성을 증가시킵니다.
this 키워드	`this`는 자바스크립트에서 매우 중요한 키워드로, 현재 실행 컨텍스트의 객체를 참조합니다. 메서드 내에서 `this`를 사용하면 해당 메서드가 속한 객체에 접근할 수 있습니다. 메서드가 호출될 때, `this`는 메서드를 호출한 객체에 자동으로 바인딩됩니다. 예를 들어, `obj.increment()`를 호출하면, `increment` 메서드 내의 `this`는 `obj`를 가리킵니다.

```
const order = {
  drink: "아메리카노",
  size: "Tall",
  makeDrink: function() {
    // makeDrink 메서드 내부에서 this.drink와 this.size를 사용하면 현재 order 객체의 속성에 접근할 수 있습니다. this는 메서드가 호출될 때 호출된 객체에 바인딩되므로, makeDrink 메서드가 order 객체에서 호출될 때 this는 order 객체를 가리키게 됩니다.
    // 객체 내에 함수로 정의된 속성은 키(key)로도 부를 수 있고, 메서드로도 부를 수 있습니다.
    console.log("종류 : " + this.drink + ", 사이즈 : " + this.size);
  }
};

order.makeDrink(); // 종류 : 아메리카노, 사이즈 : Tall
```

생성자 함수

자바스크립트에서 생성자 함수(Constructor Function)는 객체를 생성하는 데 사용되는 특별한 함수입니다. 주로 객체 지향 프로그래밍 방식에서 사용되며, 새 객체의 초기 속성을 설정하고, 필요한 초기화 작업을 수행하는 데 사용됩니다.

생성자 함수는 new 키워드와 함께 호출되어야 합니다. new를 사용하여 함수를 호출하면, 함수 내부에서 새로운 객체가 생성되고, 이 객체는 함수 내의 this로 참조됩니다. 함수의 실행이 끝날 때, 이 this 객체가 함수 호출의 결과로 반환됩니다.

목적	생성자 함수	일반 함수
호출 방식	`new` 키워드를 사용하여 객체 인스턴스를 생성합니다.	호출하는 동안 `new` 키워드를 사용하지 않습니다.
반환 값	일반적으로 명시적인 반환문이 없는 경우에도 객체를 반환합니다.	명시적으로 반환된 값이 있거나 아무 값도 반환하지 않을 수 있습니다.
이름 관례	일반적으로 첫 글자를 대문자로 시작하여 이름을 짓습니다. 예: `new Date()`	소문자 또는 대문자로 이름을 짓습니다.
사용 패턴	동일한 구조의 여러 객체를 생성할 때 사용됩니다.	특정 작업을 수행하기 위해 호출됩니다.

---

new Function()

자바스크립트의 new Function()은 동적으로 새로운 함수를 생성하는 방법 중 하나입니다. 이 생성자는 문자열로 된 함수의 본문과 매개변수를 받아서 새로운 함수 객체를 만듭니다.

항목	설명

항목	설명
기본 구조	`new Function([arg1[, arg2[, ...argN]],] functionBody)`
매개변수	(arg1, arg2, ... argN) 함수의 매개변수를 나타내는 문자열들입니다.
동적 함수 생성	`Function` 생성자는 실행 시점에 함수를 정의할 수 있게 해주고, 코드를 동적으로 만들고 실행합니다.
범위 (Scope)	`new Function()`으로 생성된 함수는 글로벌 스코프에서 실행됩니다. 따라서, 이러한 방식으로 생성된 함수 내부에서는 로컬 스코프에 접근할 수 없으며, 오직 전역 변수와 함수에만 접근할 수 있습니다.

```
// new Function을 사용하여 함수 생성
let sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 3)); // 5

// 여러 매개변수와 복잡한 로직을 포함하는 함수
let complexFunction = new Function('a', 'b', 'if(a > b) { return a * b; }
else { return a + b; }');

console.log(complexFunction(3, 2)); // 6 (3 * 2)
console.log(complexFunction(2, 3)); // 5 (2 + 3)
```

자바스크립트 프로토타입

자바스크립트의 프로토타입(Prototype)은 객체지향 프로그래밍의 중요한 특징 중 하나로, 자바스크립트가 객체 간의 상속을 구현하는 메커니즘입니다.

개념	설명
프로토타입 객체	자바스크립트에서 모든 객체는 `prototype`이라는 숨겨진 속성을 가지고 있습니다. 이 속성은 다른 객체를 참조하며, 이를 '프로토타입 객체'라고 합니다. 객체의 프로토타입은 해당 객체에 없는 속성이나 메서드에 접근할 때 사용됩니다.
상속 메커니즘	프로토타입을 통해 객체는 다른 객체의 속성과 메서드를 상속받을 수 있습니다. 객체에서 특정 속성이나 메서드를 찾을 때, 자바스크립트는 먼저 그 객체 내부를 검색하고, 없을 경우 프로토타입 체인을 따라 상위 객체의 속성과 메서드를 찾습니다.
프로토타입 체인	한 객체의 프로토타입에서 또 다른 객체의 프로토타입으로 연결되는 이 체인을 통해, 자바스크립트는 여러 객체 간의 상속 관계를 형성합니다. 이 체인의 끝에는 대부분 `Object.prototype`이 위치하며, 이는 모든 일반 자바스크립트 객체의 근본이 됩니다.

개념	설명
동적 상속	객체의 프로토타입은 동적으로 변경될 수 있으며, 이는 실행 중인 프로그램에서 객체의 행동을 변경할 수 있도록 해줍니다.

프로토타입 예제

JavaScript의 배열은 Array.prototype에서 메서드와 속성을 상속받습니다. 따라서 모든 배열은 Array.prototype에 정의된 메서드(push, pop, forEach 등)를 사용할 수 있습니다.

```
let myArray = [1, 2, 3];
myArray.push(4); // myArray는 Array.prototype의 메서드를 상속받음
```

프로토타입 장점과 단점

장점	설명
메모리 효율	같은 생성자로 생성된 모든 객체가 메서드를 공유하기 때문에, 각 객체가 메서드의 복사본을 가지고 있을 필요가 없어 메모리를 절약할 수 있습니다.
재사용성	공통된 특성과 행위를 가진 객체들을 쉽게 만들 수 있어, 코드의 재사용성과 관리 용이성이 증가합니다.

단점	설명
오버라이딩의 복잡성	하위 객체에서 상위 객체의 메서드를 오버라이드(재정의)할 때 예상치 못한 결과가 발생할 수 있습니다.
속성 공유의 문제	프로토타입을 통해 상속받은 속성이 참조 타입일 경우, 한 객체에서 이 속성을 변경하면 다른 모든 객체에도 영향을 미칩니다.

Getter와 Setter

개념	설명
함수의 Prototype 프로퍼티	모든 함수는 `prototype` 속성을 가지며, 이는 함수가 생성자로 사용될 때 생성된 객체의 프로토타입으로 설정됩니다. 이를 통해 생성자 함수의 인스턴스들이 메서드와 속성을 공유할 수 있습니다.
프로퍼티 Getter와 Setter	객체의 특정 프로퍼티에 접근하거나 수정할 때 호출되는 특별한 메서드입니다. `get`과 `set`을 사용하여 정의되며, 데이터 검증이나 계산된 속성을 제공하는 데 유용합니다.

```
// 프로퍼티 Getter와 Setter
let person = {
  firstName: 'John',
  lastName: 'Doe',

  get fullName() {
```

```
    return `${this.firstName} ${this.lastName}`;
  },

  set fullName(name) {
    [this.firstName, this.lastName] = name.split(' ');
  }
};

console.log(person.fullName); // "John Doe"
person.fullName = 'Jane Smith';
console.log(person.fullName); // "Jane Smith"


// 함수의 prototype 프로퍼티
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a noise.`);
};

const dog = new Animal('Dog');
dog.speak(); // "Dog makes a noise."
```

자바스크립트에서 객체지향 프로그래밍

자바스크립트에서 객체지향 프로그래밍을 구현하기 위해 생성자 함수와 프로토타입을 사용하는 과정은 다음과 같이 정리할 수 있습니다: 먼저, 생성자 함수를 정의하여 객체의 기본 구조를 설정합니다. 이후, 이 생성자 함수의 프로토타입 속성에 메서드와 속성을 추가하여 객체의 공통 기능을 정의합니다. 그리고 new 키워드를 사용해 생성자 함수의 인스턴스를 생성할 때, 이 인스턴스는 생성자 함수의 프로토타입 속성에 정의된 메서드와 속성을 상속받게 됩니다. 이 방식을 통해, 프로토타입을 기반으로 하는 객체지향 패턴을 자바스크립트에서 구현할 수 있습니다.

```
// prototype 을 new 생성자로 생성하면 새로운 객체가 생성되는 예제
function Bread() {}

Bread.name = 'choco';
console.log(Bread.name) // 'choco'

Bread.prototype.name = 'salt';

// Bread.name 와 myBread.name는 다름
let myBread = new Bread();
console.log(myBread.name) // 'salt'

let yourBread = new Bread();
console.log(yourBread.name) // 'salt'
```

```
yourBread.name = 'coffee';
console.log(yourBread.name) // 'coffee'
```

Class

- JavaScript에서 클래스(class)는 객체 생성을 위한 템플릿으로 사용됩니다.
- 객체 지향 프로그래밍의 중요한 구성 요소인 클래스는 데이터(속성)와 이를 조작하는 메서드를 하나의 단위로 묶어 관리합니다.
- 클래스는 특정 데이터 구조와 관련된 기능을 함께 캡슐화하고 구조화하는 중요한 역할을 수행합니다.
- 자바스크립트에서, 클래스는 기술적으로 함수의 한 종류로 간주됩니다.

개념	설명
생성자 (Constructor)	클래스를 통해 객체가 생성될 때 자동으로 호출되는 특별한 메서드입니다. 일반적으로 객체의 초기 상태를 설정하는 데 사용됩니다.
속성 (Properties)	클래스에 의해 생성된 객체들이 가질 수 있는 변수나 데이터입니다.
메서드 (Methods)	클래스 내에서 정의된 함수로, 클래스가 가지고 있는 데이터를 조작하거나 계산하는 데 사용됩니다.
상속 (Inheritance)	한 클래스가 다른 클래스의 특성을 상속받아 사용할 수 있습니다. 이를 통해 코드의 재사용성과 유지보수성이 향상됩니다.
캡슐화 (Encapsulation)	데이터(속성)와 그 데이터를 조작하는 메서드를 하나로 묶음으로써 정보 은닉을 가능하게 합니다.

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

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

const person = new Person("Alice");
console.log(typeof Person);  // "function"
console.log(person.greet()); // "Hello, my name is Alice"
```

클래스 선언과 표현식

클래스 정의는 클래스 선언(class MyClass {}) 또는 클래스 표현식(const MyClass = class {})을 통해 이루어지며, 이는 함수를 정의하는 방법과 유사합니다. 자바스크립트의 클래스는 객체 지향적인 프로그래밍 접근을 가능하게 하는 함수 기반의 구조로 볼 수 있습니다.

구분	방법	설명
클래스 선언	`class MyClass {}`	클래스 선언은 `class` 키워드 다음에 클래스 이름을 명시하고, 중괄호(`{}`) 안에 클래스의 속성과 메서드를 정의하는 방식입니다.
클래스 표현식	`const MyClass = class {}`	클래스 표현식은 클래스를 변수에 할당하는 방식입니다. 클래스에 이름을 주거나 이름을 생략할 수 있으며, 이는 함수 표현식과 유사합니다.

constructor

constructor 메서드는 클래스에서 객체를 생성하고 초기화하는데 사용되지만, 모든 클래스에 반드시 있어야 하는 것은 아닙니다. 예를 들어, constructor가 없는 간단한 클래스는 다음과 같이 정의할 수 있습니다.

구분	설명
초기화 작업	`constructor`를 사용하는 주된 이유는 새로 생성된 객체의 초기 상태를 설정하는 것입니다. 속성 값을 설정하거나 필요한 초기화 작업을 수행합니다.
선택 사항	클래스에서 `constructor`를 생략할 수 있습니다. 생략할 경우, 자바스크립트 엔진이 기본 `constructor`를 자동으로 제공합니다.
자동 생성	`constructor`가 명시적으로 정의되지 않은 경우, 클래스는 빈 `constructor`를 가지고 있는 것으로 간주됩니다. 즉, 인스턴스를 생성할 때 아무런 작업도 수행하지 않는 기본 생성자가 존재하는 것과 동일합니다.

```
class MyClass {
  myMethod() {
    console.log('Hello, World!');
  }
}

const myInstance = new MyClass();
myInstance.myMethod(); // "Hello, World!" 출력
```

Private과 Protected 프로퍼티와 메서드

유형	설명
private 프로퍼티/메서드	클래스 내부에서만 접근 및 사용이 가능한 프로퍼티나 메서드입니다. JavaScript에서는 private 필드를 `#` 기호를 사용하여 선언합니다. 클래스 외부에서 접근할 수 없어 데이터 캡슐화와 보안성을 향상시킵니다.
protected 프로퍼티/메서드	클래스 자체와 그 자식 클래스에서만 접근할 수 있는 프로퍼티나 메서드입니다. JavaScript에서는 공식적인 `protected` 지원이 없지만, 관례적으로 `_` (언더스코어) 접두사를 사용하여 표시하는 경우가 많습니다.

```
class MyClass {
  #privateField;   // private 프로퍼티 선언. 클래스 내부에서만 접근 가능합니다.
  _protectedField; // protected 프로퍼티 선언. 클래스 내부와 상속받은 자식 클래스에서 접근 가능합니다.

  constructor() {
    this.#privateField = "private";     // private 프로퍼티 초기화
    this._protectedField = "protected"; // protected 프로퍼티 초기화
  }

  #privateMethod() {
    console.log('이것은 private 메서드입니다.'); // private 메서드 정의. 클래스 내부에서만 호출 가능합니다.
  }

  _protectedMethod() {
    console.log('이것은 protected 메서드입니다.'); // protected 메서드 정의. 클래스 내부와 상속받은 자식 클래스에서 호출 가능합니다.
  }
}

const myInstance = new MyClass(); // MyClass의 인스턴스 생성
// myInstance.#privateField;  // SyntaxError. Private 필드에 외부에서 접근 시도, 접근할 수 없습니다.
// myInstance.#privateMethod();  // SyntaxError. Private 메서드에 외부에서 호출 시도, 호출할 수 없습니다.
console.log(myInstance._protectedField); // protected 필드 접근 가능. 출력: "protected"
myInstance._protectedMethod(); // protected 메서드 호출 가능. 출력: "이것은 protected 메서드입니다."
```

---

call, apply, bind 메서드

자바스크립트에서 call, apply, 그리고 bind는 함수의 컨텍스트(this 값)를 명시적으로 설정할 때 사용하는 매우 중요한 메서드들입니다. 이들은 함수를 다양한 컨텍스트에서 재사용하고, 함수의 호출 방식을 더욱 유연하게 만들어 줍니다.

메서드	설명
call	`call` 메서드는 함수를 호출하면서 첫 번째 인자로 지정된 객체를 해당 함수의 `this`로 설정합니다. 이 메서드를 사용하면 다른 객체의 메서드를 현재의 컨텍스트에서 실행할 수 있으며, 추가 인자들은 함수에 직접 전달됩니다.
apply	`apply` 메서드는 `call`과 유사하지만, 함수에 전달되는 추가 인자들을 배열 형태로 받습니다. 이 메서드는 특히 배열 형태의 인자를 함수에 전달할 때 유용합니다.

메서드	설명
bind	`bind` 메서드는 새로운 함수를 반환하고, 이 반환된 함수의 `this`는 `bind`를 호출할 때 첫 번째 인자로 전달된 객체로 고정됩니다. `bind`는 함수의 `this` 값을 미리 설정하고 새로운 함수를 생성하고 싶을 때 사용됩니다.

```
// 빵 가게 정의
const bakery = {
  breadType: "바게트",
  flavor: "허니버터",
};

// 빵 굽기 함수
function bakeBread() {
  console.log("🥖 " + this.breadType + " 빵이 구워졌습니다. 맛은 " + this.flavor
+ "입니다.");
}

// 빵 굽기 - call 방식
bakeBread.call(bakery); // 출력: 🥖 바게트 빵이 구워졌습니다. 맛은 허니버터입니다.

// 빵 굽기 - apply 방식
bakeBread.apply(bakery); // 출력: 🥖 바게트 빵이 구워졌습니다. 맛은 허니버터입니다.

// 빵 굽기 - bind 방식
const boundBreadFunction = bakeBread.bind(bakery);
boundBreadFunction(); // 출력: 🥖 바게트 빵이 구워졌습니다. 맛은 허니버터입니다.

// 화살표 함수로 빵 굽기
const bakeBreadArrow = () => {
  console.log("🥖 " + this.breadType + " 빵이 구워졌습니다. 맛은 " + this.flavor
+ "입니다.");
};

bakeBreadArrow.call(bakery); // 출력: 🥖 undefined 빵이 구워졌습니다. 맛은
undefined입니다.
bakeBreadArrow.apply(bakery); // 출력: 🥖 undefined 빵이 구워졌습니다. 맛은
undefined입니다.
const boundBreadFunctionArrow = bakeBreadArrow.bind(bakery);
boundBreadFunctionArrow(); // 출력: 🥖 undefined 빵이 구워졌습니다. 맛은
undefined입니다.
```

> Function.prototype.call() :
> https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Function/call
> Function.prototype.apply() :
> https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

> Function.prototype.bind() :
> https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Function/bind