객체지향프로그래밍이란

객체 지향 프로그래밍(OOP, Object-oriented programming)은, 절차 지향 프로그래밍과는 다르게 데이터와 기능을 한데 묶어서 처리한다.

속성과 메소드가 하나의 "객체"라는 개념을 포함되며, 이는 자바스크립트 내장 타입인 object(객체)와는 다르게, 클래스(class)라는 이름으로 부른다.

Action Items

메소드 호출 실습하기

메소드 호출은, 객체.메소드() 과 같이 객체 내에서 메소드를 호출하는 방법을 의미한다.

단순 객체를 사용한 이러한 예제를 흔히 볼 수 있다. 카운터를 구현한 예제입니다.

```
let counter1  =  {
  value: 0,
  increase:  function() {
    this.value++
  },
  decrease:  function() {
    this.value--
  },
  getValue: function() {
    return this.value
  }
}

counter1.increase() //value+=1 value = 1
counter1.increase() //value+=1 value = 2
counter1.decrease() //value-=1 value = 1
counter1.getValue() // 1
```

클로저를 이용해 매번 새로운 객체 생성하기

singleton 패턴은 단 하나의 객체만 만들 수 있다. 만약 똑같은 기능을 하는 카운터를 여러 개가 필요하다면, 똑같은 기능을 하는 카운터를 여러 개 만들지 않고 아래 코드와 같이 클로저 모듈 패턴을 이용할 수 있다.

```
function makeCounter() {

  return {

    value: 0,
    increase: function() {
      this.value++
    },
    decrease: function() {
      this.value--
    },
    getValue: function() {
      return this.value;
    }
  }
}
let counter1 = makeCounter()
counter1.increase()
counter1.getValue() // 1

let counter2 = makeCounter()
counter2.increase()
counter2.increase()
counter2.getValue() // 2
```

객체 지향 프로그래밍

: 하나의 모델이 되는 청사진을 만들고, 그 청사진을 바탕으로 한 객체(object)를 만드는 프로그래밍 패턴

- 하나의 모델이 되는 청사진을 만들고 ⇒ class
- 그 청사진을 바탕으로 한 객체(object)를 만드는 ⇒ instance

new 키위드를 통해 클래스의 인스턴스를 만들어낼 수 있다.

```
class Car {
  constructor(brand, name, color){
    this.brand = brand
    this.name = name
    this.color =  color
  }
  refuel() {
  }
  drive() {
  }
}

let mini = new Car('bmw', 'mini', 'white');
let beetles = new Car('volkswagen', 'beetles', 'red')

mini.refuel() // 미니에 연료를 공급합니다.
mini.brand  // 'bmw'

beetles.refuel() // 비틀에 연료를 공급합니다.
beetles.color // 'red'
```

prototype

: 모델의 청사진을 만들 때 쓰는 원형 객체

constructor

: 인스턴스가 초기화될 때 실행하는 생성자 함수

this

: 함수가 실행될 때, 해당 스코프마다 생성되는 고유한 실행 context(execution context) new 키워드로 인스턴스를 생성했을 때에는, 해당 인스턴스가 바로 this의 값이 됨.

```
function Car(brand, name, color){
   this.brand = brand;
   this.name = name;
   this.color = color;
} //class and constructor(생성자)함수

let avante = new Car('hyundai', 'avante', 'black');// instance
avante.brand //'hyundai'
avante.name //'avante'
avante.color //'black'
```

Action Items

: 클래스 문법을 이용하여 카운터를 만들어 보면,

```
class Counter {
  constructor() {
    this.value = 0; //생성자 호출을 할 경우, this는 new 키워드로 생성한 Counter의 인스턴스
  }
  increase() {
    this.value++;
  }
  decrease() {
    this.value--;
  }
  getValue() {
    return this.value
  }
}

let counter1 = new Counter() //생성자 호출
counter1.increase()
counter.getValue() // 1
```

OOP BASIC CONCEPTS

- Encapsulation (캡슐화)
- Inheritance (상속)
- Abstraction (추상화)
- Polymorphism (다형성)

캡슐화 Encapsulation

- 데이터와 기능을 하나의 다위로 묶는 것
- 은닉: 구현은 숨기고, 동작은 노출시킴
- 느슨한 결합에 유리: 언제든 구현을 수정가능

캡슐화는 외부에서 앞서 말했던 데이터(속성)와 기능(메소드)를 따로 정의하는 것이 아닌, 하나의 객체 안에 넣어서 묶는 것입니다. 데이터(속성)과 기능(메소드)들이 느슨하게 결합되는 것이다.

느슨한 결합은 코드 실행 순서에 따라 절차적으로 코드를 작성하는 것이 아니라, 코드가 상징하는 실제 모습과 닮게 코드를 모아 결합하는 것을 의미

은닉화는 내부 데이터나 내부 구현이 외부로 노출되지 않도록 만드는 것, 따라서, 디테일한 구현이나 데이터는 숨기고, 객체 외부에서 필요한 동작(메소드)만 노출시켜야 합니다. 은닉화의 특징을 살려서 코드를 작성하면 객체 내 메소드의 구현만 수정하고, 노출된 메소드를 사용하는 코드 흐름은 바뀌지 않도록 만들 수 있습니다. 반면 절차적 코드의 경우 데이터의 형태가 바뀔 때에 코드의 흐름에 큰 영향을 미치게 되어 유지보수가 어렵다. 그래서 더 엄격한 클래스는 속성의 직접적인 접근을 막고, 설정하는 함수, 불러오는 함수를 철저하게 나누기도 한다.

추상화 Abstraction

- Simpler Interface
- Reduce the Impact of Change

이러한 추상화를 통해 인터페이스가 단순해진다. 너무 많은 기능들이 노출되지 않은 덕분에 예기치 못한 사용상의 변화가 일어나지 않도록 만들 수 있다.

추상화는 클래스를 사용하는 사람이 필요하지 않은 메소드 등을 노출시키지 않고, 단순한 이름으로 정의하는 것에 포커스가 맞춰져 있다.

상속 Inheritance

상속은 부모 클래스의 특징을 자식 클래스가 물려받는 것.

기본 용어는 기본클래스(base class)의 특징을 파생 클래스(derive class)가 상속받는다로 표현한다.

```
//상위 클라스를 상속 받아오는 방법

// 상위 클라스를 우선 선언
class Cloth {
  constructor(type, color, price) {
    this.type = 'skirt'
    this.color = 'black'
    this.price = 10000
  }
}

//상위 클라스 상속 받아오기
class MyWish extends Cloth {
  constructor() {
    super()
  }
}

let purchase = new MyWish()

purchase.type === 'skirt'
purchase.color === 'white'
purchase.price === 10000
```

다형성 Polymorphism

객체 역시 똑같은 메소드라 하더라도, 다른 방식을 구현이 가능하다.

예를 들어, TextBox, Select, Checkbox의 공통의 부모인 HTML Element라는 클래스에 render라는 메소드를 만들고 상속을 받게 만들수 있다. 그런데 다형성의 핵심은 이 같은 이름의 render라는 메소드가 조금씩 다르게 작동한다는 데 있다. 이처럼 같은 이름을 가진 메소드라도 조금씩 다르게 작동합니다. 이것이 바로 다형성이다.

OOP의 주요 개념에 대한 장점

캡슐화는 코드가 복잡하지 않게 만들고, 재사용성을 높인다.

추상화는 마찬가지로 코드가 복잡하지 않게 만들고, 단순화된 사용으로 인해 변화에 대한 영향을 최소화한다.

상속 역시 불필요한 코드를 줄요 재사용성을 높인다.

다형성으로 인해 동일한 메소드에 대해 if/else if와 같은 조건문 대신 객체의 특성에 맞게 달리 작성하는 것이 가능해진다.

Advance learning

- `addEventListener` 속성은 어떤 클래스의 프로토타입에서 찾을 수 있나요?
- `remove` 메소드는 어떤 클래스의 프로토타입에서 찾을 수 있나요?
- 모든 객체에 `toString()` 메소드가 존재하는 이유가 무엇인가요?