04

04-4. 다양한 OOP 기법

JavaScript Object Oriented Programming

typeof, instanceof

- typeof 연산자는 타입을 확인하기 위한 연산자입니다.

typeof 로 타입 확인

```
1   function User(){}
2   let user1 = new User()
3
4   console.log(typeof 10)//number
5   console.log(typeof "hello")//string
6   console.log(typeof true)//boolean
7   console.log(typeof User)//function
8   console.log(typeof [10, 20])//object
9   console.log(typeof user1)//object
```

typeof, instanceof

- instanceof 는 객체의 타입이 특정 타입인지를 판단하기 위한 연산자입니다.
- instanceof 왼쪽에 객체를, 오른쪽에 타입을 명시해서 왼쪽 객체의 타입이 오른쪽에 명시한 타입인지를 판단하는 연산자입니다.
- 연산의 최종 결과는 true/false 입니다.

- instanceof 는 생성자를 가지고 판단하는 연산자입니다.

instanceof 로 타입 확인

```
1   console.log(10 instanceof Number)//false
2   console.log("hello" instanceof String)//false
3   console.log(true instanceof Boolean)//false
```

객체로 생성해서 타입 체크

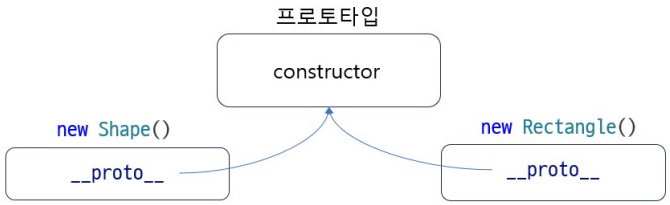
```
1   console.log(new Number(10) instanceof Number)//true
2   console.log(new String("hello") instanceof String)//true
3   console.log(new Boolean(true) instanceof Boolean)//true
4   console.log(new Number(10) instanceof String)//false
```

typeof, instanceof

- 다른 함수의 프로토타입을 그대로 자신의 프로토타입으로 지정하는 경우의 instanceof

다른 함수의 프로토타입을 지정하는 경우

```
1   function Shape(){}
2   function Rectangle(){}
3   Rectangle.prototype = Shape.prototype
4
5   let shape1 = new Shape()
6   let rect1 = new Rectangle()
7
8   console.log(shape1 instanceof Shape)//true
9   console.log(shape1 instanceof Rectangle)//true
10  console.log(rect1 instanceof Shape)//true
11  console.log(rect1 instanceof Rectangle)//true
```

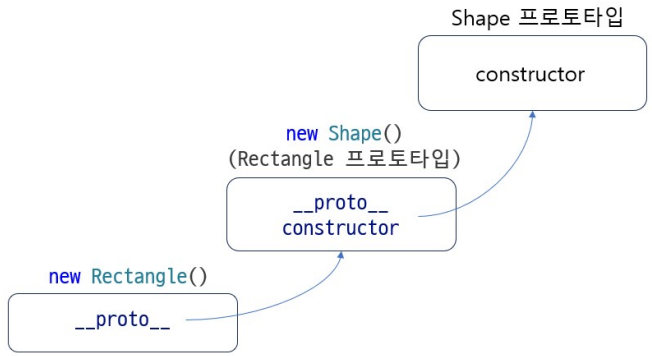


typeof, instanceof

- 상위 객체를 생성해서 하위 프로토타입으로 지정하는 경우의 instanceof

상위 객체를 프로토타입으로 지정하는 경우

```
1   function Shape(){}
2   function Rectangle(){}
3   Rectangle.prototype = new Shape()
4
5   let shape1 = new Shape()
6   let rect1 = new Rectangle()
7
8   console.log(shape1 instanceof Shape)//true
9   console.log(shape1 instanceof Rectangle)//false
10  console.log(rect1 instanceof Shape)//true
11  console.log(rect1 instanceof Rectangle)//true
```



프로퍼티 Descriptor

- 객체에 프로퍼티에는 설명자(Descriptor)라는 정보가 있습니다.
- 설명자를 이용해 객체의 프로퍼티 값이 변경되지 않게 하거나 열거로 사용되지 않게 하는 등 원하는데로 사용되게 할 수 있습니다.

- 객체를 선언하면서 프로퍼티의 설명자를 따로 지정하지 않는다고 하더라도 기본으로 프로퍼티에 설명자가 추가되게 됩니다.
- 프로퍼티의 설명자를 확인하려면 Object.getOwnPropertyDescriptor() 를 이용하면 됩니다.

```
console.log(Object.getOwnPropertyDescriptor(obj, 'name'))
//{value: '홍길동', writable: true, enumerable: true, configurable:
true}
```

프로퍼티 Descriptor

- 설명자의 각 속성의 의미는 아래와 같습니다.

- value : 프로퍼티에 대입된 값
- writable: 프로퍼티 값을 수정할 수 있는지에 대한 여부
- enumerable : 프로퍼티가 열거형으로 이용이 가능한지에 대한 여부
- configurable : 프로퍼티의 설명자를 변경할 수 있는지에 대한 여부

프로퍼티 Descriptor

writable

- writable 은 프로퍼티의 값을 변경하지 못하게 할 때 이용하면 됩니다.
- 객체를 선언할 때 지정한 값으로만 사용되어야 하거나 특정 업무가 진행되는 동안 값 변경이 불가능하게 하고자 할 때 이용하면 됩니다.

- 프로퍼티의 설명자 값을 변경하고 싶다면 Object.defineProperty() 를 이용하면 됩니다.

```
Object.defineProperty(obj, 'age', {writable: false})
```

프로퍼티 Descriptor

enumerable

- enumerable 은 프로퍼티가 열거로 이용이 가능한지를 설정하기 위해서 사용됩니다.
- 열거라는 것은 프로퍼티 명을 명시하지 않고 어떤 객체에 등록된 프로퍼티들을 순서대로 나열해서 사용하는 것을 의미하며 특정 프로퍼티가 이 열거로 이용되지 않게 하고자 할 때 enumerable 을 false 로 지정합니다.

```
Object.defineProperty(obj, 'age', {enumerable: false})
```

프로퍼티 Descriptor

configurable

- configurable 은 프로퍼티의 설명자를 재설정 할 수 있는지에 대한 정보입니다.
- false 로 지정하면 재설정이 불가해 집니다.

```
Object.defineProperty(obj, 'age', {writable: false, configurable:
false})
```

new Object()

- 함수나 클래스 같은 객체의 모형을 선언하고 이 모형을 이용해 객체를 생성하지 않고 간단하게 객체를 선언하는 방법은 객체 리터럴 입니다.
- 그런데 객체 리터럴 방법을 이용하지 않고 new Object() 로 객체를 선언할 수도 있습니다.

new Object() 로 객체 생성

```
1   let obj = new Object()
2
3   obj.name = '홍길동'
4   obj.age = 20
5   obj.sayHello = function(){
6     console.log(`Hello ${this.name}, age : ${this.age}`)
7   }
8
9   obj.sayHello()//Hello 홍길동, age : 20
```

Object.create()

- 자바스크립트의 모든 객체는 그 객체를 만드는 프로토타입이 있어야 합니다.
- 객체 리터럴로 객체를 생성하는 것은 new Object() 방법으로 객체를 생성하는 것과 동일함으로 객체 리터럴로 만든 객체의 프로토타입은 Object 의 프로토타입입니다.

- 그런데 경우에 따라 객체를 생성하면서 Object의 프로토타입이 아닌 다른 프로토타입을 지정하고 싶은 경우가 있습니다.
- 이렇게 하고자 하는 주된 이유는 어떤 프로토타입을 지정해 객체를 생성해서 그 프로토타입에 등록된 멤버를 객체에서 이용하고자 하는 이유입니다.
- 일종의 상속개념으로 프로토타입을 지정해 그 프로토타입의 코드를 재사용하는 개념이며 이때 Object.create() 로 객체를 생성하면 됩니다.

Object.create()

Object.create(proto [, propertiesObject])

- Object.create() 로 객체를 생성할 때 첫번째 매개변수로 생성되는 객체의 프로토타입을 지정해 주어야 합니다.
- 그리고 두번째 매개변수는 옵셔널로 생략가능한데 등록한다면 생성되는 객체의 프로퍼티입니다.
- 객체의 프로퍼티를 지정하고 있는데 name: { } 형태로 프로퍼티를 등록해야 합니다.
- name 은 프로퍼티 명이며 { } 은 프로퍼티의 설명자입니다. 즉 value, writable, enumerable, configurable 등의 정보로 프로퍼티의 설명자를 등록해야 합니다.

```
let user2 = Object.create(Object.prototype, {
  name: {value: '홍길동'},
  age: {value: 20}
})
```

Object.create()

- Object.create() 를 이용하는 것은 등록하는 프로퍼티에 설명자를 객체를 생성하면서 지정하고자 하는 경우입니다.
- 또다른 경우는 특정 프로토타입을 지정해서 코드 재사용을 하겠다는 이유입니다.

```
let rect1 = Object.create(Shape.prototype, {
  name: {value: 'rect1'},
  width: {value: 10},
  height: {value: 10}
})
```

this

- this 예약어는 코드를 실행시키는 객체를 의미합니다.
- 주로 함수 내에서 이용되며 함수를 호출한객체를 지칭하기 위해서 this 가 사용됩니다.
- 그런데 자바스크립트에서는 함수를 호출한 객체가 정적이지 않고 동적입니다.

- 자바스크립트에서는 실행단계에서 동적으로 this 가 결정이 됩니다.
- 동일한 함수의 this 라고 하더라도 함수를 어떻게 호출했는지에 따라 다른 객체를지칭하게 됩니다.

this

전역위치에 선언된 함수에서의 this

- 전역위치 함수 호출의 경우, 자바스크립트 코드를 엄격모드에서 작성하고 있는지에 따라 this 가 다르게 나올 수 있습니다.

- 일반모드에서는 전역위치를 실행시키는 객체는 별도로 명시하지 않아도 window 가 지정됩니다.

```
console.log(this == window)//true

function myFun1(){
  console.log(this == window)
}

myFun1()//true
window.myFun1()//true
this.myFun1()//true
```

this

전역위치에 선언된 함수에서의 this

- 엄격모드에서 전역위치 함수를 객체를 지정하지 않고 호출하게 되면 this 는 window 가 아니라 undefined 상태입니다.

엄격모드

```
1   "use strict";
2
3   console.log(this == window)//true
4
5   function myFun1(){
6     console.log(this == window)
7   }
8
9   myFun1()//false
10  window.myFun1()//true
11  this.myFun1()//true
```

this

함수 내에 선언된 함수에서의 this

- 함수 내에 선언된 함수를 호출하는 경우도 전역 위치의 함수 호출과 동일합니다.

함수 내부의 함수 호출

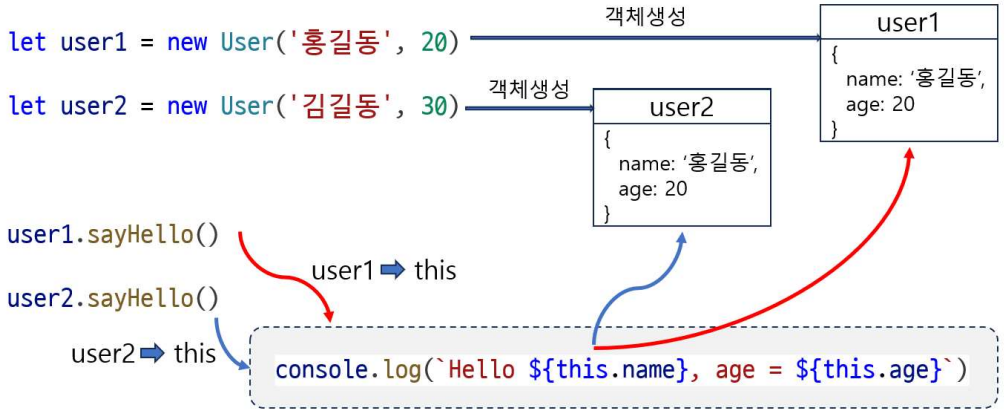
```
1   function outerFun(){
2     function innerFun1(){
3       console.log(this == window)//false
4       console.log(this)//undefined
5     }
6     function innerFun2(){
7       innerFun1()
8       console.log(this == window)//false
9     }
10    innerFun2()
11  }
12
13  outerFun()
```

this

객체 메서드 호출시의 this

- 객체를 생성하고 그 객체의 메서드를 호출했을 때 메서드를 실행시키는 this 는 메서드를 호출한 객체가 됩니다.

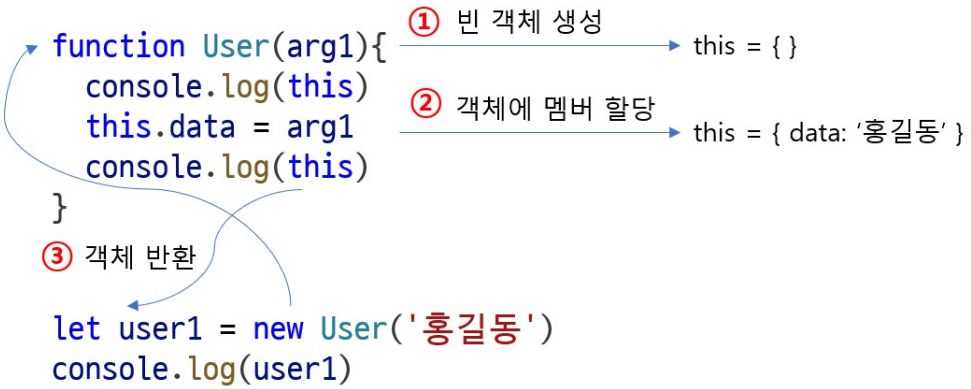
```
function User(arg1, arg2){
  this.name = arg1
  this.age = arg2

  this.sayHello = function(){
    console.log(`Hello ${this.name}, age = ${this.age}`)
  }
}
```



this

생성자 함수에서의 this

- new 연산자에 의해 호출되는 시점의 this 는 새로 만들어지는 객체를 의미합니다.



this

화살표 함수

- 자바스크립트에서 함수가 실행될때의 this 는 호출하는 시점에 결정되는 동적 바인딩이지만 화살표 함수에 한해서만 정적 바인딩이 됩니다.
- 화살표 함수내의 this 는 호출하는 시점에 결정되는 것이 아니라 코드를 작성하는 시점, 즉 선언 시점에 결정이 됩니다.
- 이를 전문 용어로 렉시컬(Lexical) this 혹은 렉시컬 바인딩이라 부릅니다.
- 화살표 함수의 this 는 선언되는 시점에 지정되며 화살표 함수의 상위 스코프에 있는 this가 지정되게 됩니다.

this

화살표 함수 – 전역 위치에 선언

- 화살표 함수는 선언된 위치의 상위 스코프의 this 에 바인딩이 됨으로 전역 위치에 선언된 화살표 함수의 상위 스코프는 window 입니다.

```
console.log(this == window)//true

let fun1 = () => {
  let data = 20
  console.log(this == window)//true
  console.log(this.data)//undefined
}
```

this

화살표 함수 – 생성자 함수내에 선언

- 화살표 함수는 선언한 순간 this 가 결정되며 상위 스코프의 this 에 등록됩니다.
- 결국 this.fun3 = () => { } 로 등록했음으로 생성자 함수로 생성되는 객체에 대입됩니다.

```
function User() {
  this.data = 30
  this.fun2 = function(){
    console.log(this.data)//30
  }
  this.fun3 = () => {
    console.log(this.data)//30
  }
}
let user1 = new User()
user1.fun2()
user1.fun3()
```

this

화살표 함수 – 객체 리터럴에 선언

- 객체 리터럴 내의 함수를 화살표 함수로 선언하면 화살표 함수내에서의 this 는 생성되는 객체를 지칭하지 못합니다.

```
let obj = {
  data: 40,
  fun4: function(){
    console.log(this.data)//40
  },
  fun5: () => {
    console.log(this.data)//undefined
  }
}
```

this 동적 바인딩

- 함수에서의 this 는 그 함수를 호출한 객체를 의미합니다.
- 자바스크립트에서는 함수의 this 를 동적 바인딩이 가능합니다.
- 동적 바인딩이란 실행시점에 함수의 this 를 지정할 수 있다는 의미이며 이 동적 바인딩 기법을 이용해 함수의 this 역할을 하는 객체를 바꿔서 이용할 수 있습니다.

bind()

- bind() 함수는 함수에 this 역할을 하는 객체를 바인딩하여 새로운 함수를 반환하는 역할을 합니다.
- bind() 는 함수를 호출하는 것이 아니라 새로운 함수를 만드는 역할입니다.

```
함수명.bind(객체명)
```

this 동적 바인딩

bind()

bind() 로 this 바인딩

```
1  let obj = {
2    name: '홍길동'
3  }
4  let sayHello = function(){
5    console.log(`Hello, ${this.name}`)//Hello, 홍길동
6  }
7  let newSayHello = sayHello.bind(obj)
8  newSayHello()
```

this 동적 바인딩

bind()

- bind 되는 함수에 매개변수를 지정하고 새로 만들어지는 함수를 호출하면서 매개변수 값을 대입할 수 있습니다.

```
let sayHello = function(arg1, arg2){
  console.log(`Hello, ${this.name}, ${arg1}, ${arg2}`)//Hello, 홍길동,
30, 40

}
let newSayHello = sayHello.bind(obj)
newSayHello(30, 40)
```

this 동적 바인딩

bind()

- bind() 로 함수에 객체를 바인딩 시키면서 매개변수 값을 지정할 수 있습니다.
- 또한 함수를 호출하면서 지정한 매개변수 값이 모두 같이 이용되게 할 수 있습니다.

```
let sayHello = function(...args){
  console.log(`Hello, ${this.name}, ${args}`)//Hello, 홍길동,
10,20,30,40
}
let newSayHello = sayHello.bind(obj, 10, 20)
newSayHello(30, 40)
```

this 동적 바인딩

call(), apply()

- bind()는 새로운 함수를 만드는 역할이지 함수를 호출하는 역할은 아닙니다.
- call(), apply() 를 이용하면 함수에 객체를 바인딩 시키고 그 함수를 호출까지 해줍니다.
- 결국 call(), apply() 의 반환 값은 새로운 함수가 아니라 함수를 호출한 결과 값입니다.

```
let sayHello = function(){
  console.log(`Hello, ${this.name}`)//Hello, 홍길동
  return 100
}
console.log(sayHello.call(obj))//100
```

this 동적 바인딩

call(), apply()

- call() 로 객체를 바인딩 시켜 함수를 호출할 수 있는데 이때 원한다면 매개변수 값을 전달 할 수 있습니다.

```
let sayHello = function(arg1, arg2){
  console.log(`Hello, ${this.name}, ${arg1}, ${arg2}`)//Hello, 홍길동,
10, 20
  return 100
}
console.log(sayHello.call(obj, 10, 20))//100
```

this 동적 바인딩

call(), apply()

- apply() 함수도 call() 와 마찮가지로 함수에 객체를 바인딩 하면서 함수를 호출하는 역할을 합니다.
- apply() 함수가 call() 과 차이가 있는 것은 함수를 호출하면서 전달하는 매개변수 값을 지정하는 방법입니다.
- apply() 는 전달하는 매개변수를 배열로 지정해야 합니다.

```
let sayHello = function(arg1, arg2){
  console.log(`Hello, ${this.name}, ${arg1}, ${arg2}`)//Hello, 홍길동,
10, 20
  return 100
}
console.log(sayHello.apply(obj, [10, 20]))
```

getter/setter

- 함수 스타일로 프로퍼티를 정의할 수도 있습니다.
- 함수 스타일의 프로퍼티란 객체내에 선언된 함수인데 외부에서는 객체의 변수처럼 사용되는 프로퍼티를 의미합니다.
- 일반적으로 어떤 객체가 가지고 있는 값을 위한 프로퍼티인데 이 프로퍼티에 값이 대입될 때 로직이 실행되어야 하거나 아니면 이 값이 참조될 때 로직이 실행되어야 하는 경우에 이용합니다.
- 로직이 실행되어야 함으로 함수로 작성이 되고 그 함수가 호출이 되어야 하는데 외부에서는 이 함수를 변수처럼 이용한다는 개념입니다.
- 이런 함수들을 흔히 getter/setter 함수라고 부릅니다.

getter/setter

- 소프트웨어 언어에서 흔히 사용하는 용어로 어떤 변수가 있고 그 변수의 값을 획득하기 위한 함수를 getter 함수라고 부르고, 그 변수의 값을 변경하기 위한 함수를 setter 라고 부릅니다.
- 이둘을 합쳐서 흔히 getter/setter 함수라고 부릅니다.
- 소프트웨어 언어별로 getter/setter 함수를 선언하는 문법의 차이는 있으며 자바스크립트에서는 get, set 예약어를 통해 변수의 getter/setter 를 추가합니다.

getter/setter

- 자바스크립트에서 프로퍼티로 이용되는 getter/setter 함수를 만들기 위해서는 get, set 이라는 예약어로 함수가 선언되어 있어야 합니다.
- get 예약어로 선언된 함수를 getter 라고 부르며 프로퍼티 값을 참조할 때 호출이 됩니다.
- set 예약어로 선언된 함수는 매개변수를 가지고 있어야 하며 이 프로퍼티 값을 변경할 때 호출이 됩니다.

get/set 활용

```
1   let obj = {
2     _num: 0,
3     get num() {
4       return this._num
5     },
6     set num(value){
7       this._num = value
8     }
9   }
10
11  obj.num = 10
12  console.log(obj.num)//10
```

getter/setter

- get 함수만 선언하면 함수를 프로퍼티로 활용할 수 있지만 get 만 있는 프로퍼티가 됨으로 값 참조만 되고 변경은 불가능한 프로퍼티가 됩니다.
- set 만 선언된 함수를 만들 수도 있습니다. 이렇게 되면 값 대입만 되는 프로퍼티가 됩니다.

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare