함수 합성

커링

커링(currying)과 부분 적용(Partial Appication)

- **커링** - 함수의 분해 기법, 다수의 인자를 가지는 함수 대신, 하나의 인자를 가지는 연속된 함수들의 중첩
- **부분 적용** - 함수의 인자 일부를 고정한 새로운 함수를 생성

커링의 동작 원리

```
//Curring : f(a,b,c) -> f(a)(b)(c)

function fn(x, y){
    return x + y
}

fn(1,2);

//커링으로 변환
function curried_fn(x){//첫번째 함수에 인자 x를 전달하고
    return function(y){//바로 함수를 리턴하는데 두번째 인자(y)를 하위함수에 적용시켜주고
        return x + y //최종적으로 x+y를 반환하면 함수를 커링으로 변환하는 문법입니다.
    }
}

//화살표 함수로 표현(문법상 다를 뿐이고 동작원리는 같음)
const curried_fn2 = x => y => x + y;

//첫번째 인자가 x를 담당하는 함수를 나타낸거고 두번째 인자가 y인자를 처리해주는 함수를 나타내는 겁니다.
```

```
console.log(
    curried_fn(1)(2),
    curried_fn2(1)(2)
)
```

커링 함수들을 보면 지금 호출하는 방식이 다릅니다.

`curried_fn(1)(2)`

첫번째 인자를 사용해서 하나의 함수를 실행하고 그리고 또 두 번째 인자를 전달하고 또 하나의 함수를 실행해요

이렇게 함수들이 독립적으로 실행이 되고 인자들을 한 번에 하나씩 전달하게 됩니다

> 💡 **함수에 전달하려는 인자들을 꼭 한꺼번에 제공하지 않아도 된다는 얘기**

인자를 담고 있는 함수의 재사용이 가능하고 하나의 기능에서 다른 기능으로 전달해야 되는 상황에서도 활용을 할 수가 있게 되는 겁니다.

```
//커피를 주문하는 함수
function makeCoffee(roastType){
    return function(sugar){
        return function(cream){
            return console.log(`Coffee, ${roastType}, sugar: ${sugar}, cream: ${cream}`)
        }
    }
}

makeCoffee('Dark Roast')(1)(2) //Coffee, Dark Roast, sugar: 1, cream: 2
```

내부 함수(클로저)에서 외부 함수의 인자값(roastType, sugar)을 읽어 왔다

클로저가 내부적으로 동작을 하였기 때문에 상위 스코프에 있는 변수들을 읽어올 수가 있었던 겁니다.

부분 적용

커피 주문이 몇 개 들어왔다고 가정을 해봅시다

```
//미디엄 로스트라는 함수를 정의해 주면서 첫번째 인자값을 미리 호출하고 값을 할당받았기 때문에 다음 순서들의 함수들을 사용할 때 나머지 인자만 전달해서 커피를 만들수 있게 되는 겁니다.
const mediumRost = makeCoffee("Medium Roast");

//주문이 들어왔다고 가정
const order1 = mediumRost(1)(2) //Coffee, Medium Roast, sugar: 1, cream: 2
const order2 = mediumRost(2)(3) //Coffee, Medium Roast, sugar: 2, cream: 3
```

1. 커링은 다항의 성질을 가진 함수 → 단일의 인자를 사용하는 함수들의 함수열로 변환
2. 클로저의 도움을 통해서 외부 환경에 값들을 기억했다가 사용을 하게 됩니다
3. 커링을 사용을 할 때 인자 일부를 고정시킨 후 새로운 함수로 사용하는 걸 부분 적용이라고 합니다

HOF 패턴

샌드위치를 만드는 함수 예시(HOF)

```
const ingredientAdder = (ingredient) => (input) => console.log(`${input},${ingredient}`);

//bacon, lettuce, cheese를 고정된 인자로 전달
const bacon = ingredientAdder("Bacon");
const lettuce = ingredientAdder("Lettuce");
const cheese = ingredientAdder("Cheese");

//중첩된 고차함수들로 합성된 패턴(가독성이 나쁨, 재사용하기 힘듬)
const makeSandwich = bread => cheese(lettuce(bacon(bread)));
```

```
makeSandwich("White Bread");
```

compose & pipeline 패턴

Compose, Pipeline 패턴을 사용하여 함수들을 연속적으로 적용

각 함수의 결과를 다음 함수의 입력으로 전달한다.

- 명확하고 간결하게 코드 관리를 할 수 있다
- 가독성과 재사용성 면에서 장점들이 많이 부각 됩니다

Compose

```
const compose = (...functions) => input =>
    functions.reduceRight((acc, func) => func(acc), input);

compose(fnC, fnB, fnA)("some input data");
            // <--------- 오른쪽에서 왼쪽 / 안쪽에서 바깥
```

컴포즈 같은 경우에는 인자로 전달 받은 함수들을

오른쪽에서 왼쪽으로 순차적으로 실행을 하고, 그 결과를 반환하는 함수

커링 패턴을 사용해서 인풋을 전달 받고 받은 인풋을 기준으로 시작해서 오른쪽에서 왼쪽 또는 안쪽에서 바깥쪽으로 실행이 됩니다

샌드위치를 만드는 함수 예시(compose)

```
const ingredientAdder = (ingredient) => (input) => console.
log(`${input},${ingredient}`);

//bacon, lettuce, cheese를 고정된 인자로 전달
const bacon = ingredientAdder("Bacon");
const lettuce = ingredientAdder("Lettuce");
const cheese = ingredientAdder("Cheese");
```

```
//compose패턴 활용
const compose = (...functions) => input =>
    functions.reduceRight((acc, func) => func(acc), input);

//makeSandwich함수 호출시 bread를 인자로 받고
const makeSandwich = bread =>
    //컴포즈패턴을 활용하여 bread만 새로운 인자로 전달(부분적용)
  compose(cheese, lettuce, bacon)(bread);

//makeSandwich함수 호출
makeSandwich("White Bread");
```

Pipeline

파이프 라인 패턴의 경우 컴포즈와 같고 실행 순서만 달라집니다

```
const pipe = (...functions) => input =>
    funcitons.reduce((acc, func) => func(acc), input);

pipe(fnA, fnB, fnC)("some input data");
    // -------------> 왼쪽에서 오른쪽 / 바깥쪽에서 안쪽

//우리가 읽기에는 pipeline 패턴이 훨씬 더 자연스럽고 편합니다
```

샌드위치를 만드는 함수 예시(pipe)

```
const ingredientAdder = (ingredient) => (input) => console.
log(`${input},${ingredient}`);

const bacon = ingredientAdder("Bacon");
const lettuce = ingredientAdder("Lettuce");
const cheese = ingredientAdder("Cheese");

const pipe = (...functions) => input =>
    functions.reduce((acc, func) => func(acc), input);
```

```
const makeSandwich = bread =>
    //인자로 전달되는 함수들의 순서가 다름
  pipe(bacon, lettuce, cheese)(bread);

makeSandwich("White Bread");
```

예제

```
const log = console.log;

//문자열 생성
const title = "Learning Function Composition 1";
//문자열을 slug 형태로 변환을 해야 하는 상황이라고 가정
//slug는 웹사이트나 블로그에서 사용되는 url의 일부를 구성하는 문자열입니다.
//검색엔진 최적화같은 부분에서 사용되는 요소중의 하나입니다

//slug -> 'learning-function-composition-1'

//1.문자열  -> 배열
const strToArr = str => str.split(" ");
//2.배열(단어) -> 소문자 변환
const toLower = arr => arr.map(w=>w.toLowerCase());
//3.배열 -> 문자열 , 여백처리("-")
const joinWithDash = arr => arr.join("-");

//중첩고차함수를 사용할 경우
//const slug = joinWithDash(toLower(strToArr(title)));
//log(slug); //learning-function-composition-1

//compose패턴 사용
const compose = (...functions) => input =>
    //안쪽에서 바깥쪽이기 때문에 reduceRight메서드를 사용, input은 초기값
```

```
    functions.reduceRight((acc, fn) => fn(acc), input)
//joinWithDash, toLower, strToArr순으로 전달
const slug = compose(joinWithDash,toLower,strToArr)(title);
log(slug); //learning-function-composition-1

//pipe패턴 사용
const pipe = (...functions) => input =>
    //바깥쪽에서 안쪽이기 때문에 reduce메서드를 사용
    functions.reduce((acc, fn) => fn(acc), input)
//compose와 반대로 함수를 전
const slug = pipe(strToArr, toLower, joinWithDash)(title);
log(slug); //learning-function-composition-1
```