

3. 자바스크립트를 활용한 함수형 프로그래밍

Prof. Seunghyun Park (sp@hansung.ac.kr)

Division of Computer Engineering

학습 목표: 3장. 자바스크립트를 활용한 함수형 프로그래밍

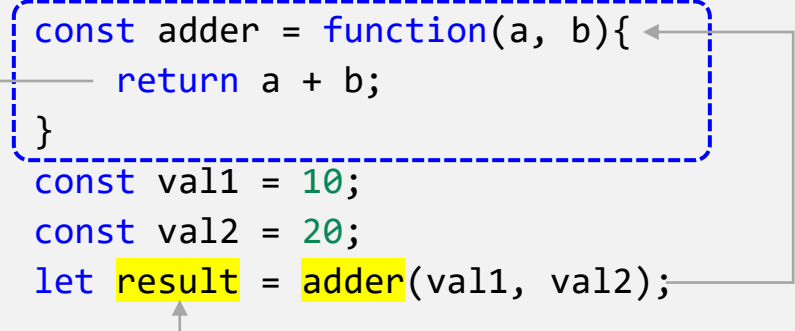
- 함수형 프로그래밍
 - 함수
 - 1급 객체와 고차 함수
 - 함수형 프로그래밍 특징
 - 명령형 프로그래밍과 선언적 프로그래밍 비교
 - 불변성
 - 순수 함수
 - 데이터 변환
 - 고차 함수
 - 재귀
 - 합성

함수

- 함수 ¹⁾: 연산자를 적용하여 평가할 수 있는 모든 호출 가능한 표현식을 의미

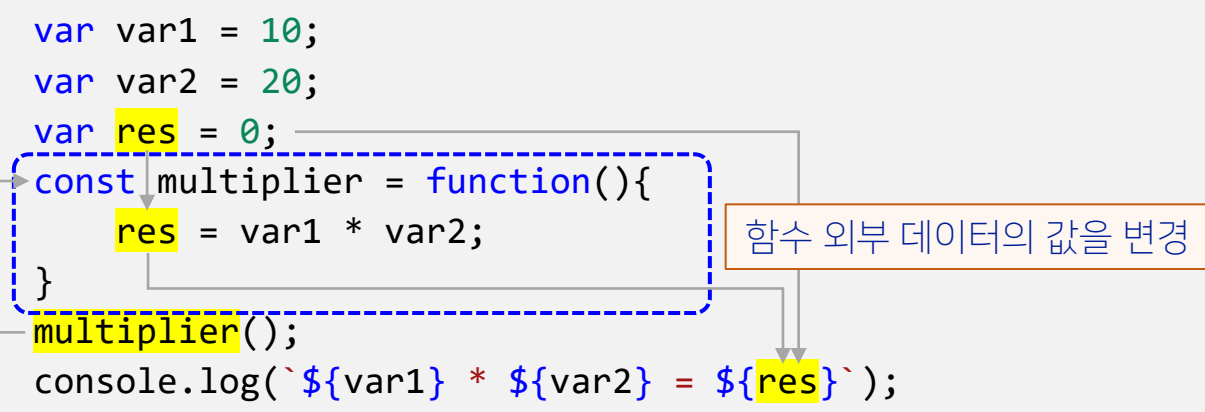
- 작업/연산 결과의 반환

```
const adder = function(a, b){  
  return a + b;  
}  
const val1 = 10;  
const val2 = 20;  
let result = adder(val1, val2);  
  
console.log(`${val1} + ${val2} = ${result}`);
```



- 내/외부 데이터의 변경

```
var var1 = 10;  
var var2 = 20;  
var res = 0;  
const multiplier = function(){  
  res = var1 * var2;  
}  
multiplier();  
console.log(`${var1} * ${var2} = ${res}`);
```



1) 출처: 함수형 자바 스크립트 (루이스 아텐시오 저, 2018, 한빛미디어)

함수

- 1급 객체 (first class object): 다른 객체들에 일반적으로 적용 가능한 연산을 모두 지원하는 객체

- 변수 (variables) 또는 데이터 구조에 할당 가능

- 매개변수 (parameters)로 전달 가능

- 반환 값 (return value)으로 전달 가능

} first class citizen 요건

→ 고차 함수 (high order function)

- 함수를 매개변수로 받거나,
- 함수를 결과로 반환하는 함수

함수: 1급 객체의 활용 예 (계속)

```
/* ch03-01-01-functional.html */  
var log = function(message) {  
    console.log(message)  
}  
  
log("함수를 변수에 할당")
```

함수를 변수에 할당

함수를 변수에 할당

```
/* ch03-01-02-functional.html */  
const log = message => console.log(message)  
  
log("화살표 함수를 상수에 할당")
```

화살표 함수를 상수에 할당

화살표 함수를 상수에 할당

```
/* ch03-01-03-functional.html */  
const obj = {  
    message: "함수를 객체에 추가",  
    log(message) {  
        console.log(message)  
    }  
}  
  
obj.log(obj.message)
```

함수를 객체에 포함

함수를 객체에 추가

```
/* ch03-01-04-functional.html */  
const messages = [  
    "함수를 배열에 추가",  
    msg => console.log(msg),  
    "변수와 동일하게 취급",  
    msg => console.log(msg)  
]  
  
messages[1](messages[0])  
messages[3](messages[2])
```

함수를 배열의 원소로 활용

함수를 배열에 추가
변수와 동일하게 취급

함수: 1급 객체의 활용 예 (계속)

```
/* ch03-01-05-functional.html */  
const insideFn = logger => logger("함수를 다른 함수에 매개변수로 전달")
```

```
insideFn(msg => console.log(msg))
```

함수를 매개변수로 활용

함수를 다른 함수에 매개변수로 전달

```
insideFn : (logger) => logger("함수를... ")
```

insideFn()의 매개변수로 msg => console.log(msg)를 전달

insideFn() 함수 정의

👍 insideFn(logger)의 결과는 logger("함수를... ")를 호출

👍 msg => console.log(msg)를 우선 함수 temp()로 정의했다고 가정

```
temp() : msg => console.log(msg)
```

```
insideFn(temp) === temp("함수를...") === console.log("함수를...")
```

```
/* ch03-01-05-functional-1.html */  
const insideFn = (logger) => {  
  logger("함수를 다른 함수에 매개변수로 전달");  
};
```

```
const temp = (msg) => {  
  console.log(msg);  
};
```

```
insideFn(temp);
```

함수: 1급 객체의 활용 예

```
/* ch03-01-06-functional.html */
var createScream_06 = function(logger) {
  return function(message) {
    logger(message.toUpperCase() + "!!!")
  }
}

const scream_06 = createScream_06(message => console.log(message))
scream_06('createScream은 함수를 반환')
```

함수의 결과로 함수를 반환

```
/* ch03-01-07-functional.html */
const createScream_07 = logger => message =>
  logger(message.toUpperCase() + "!!!")

const scream_07 = createScream_07(message => console.log(message))
scream_07('ES6에서는 더 간편하게 createScream을 만들 수 있음')
```

```
// function() 을 => 함수로 변환 ✓
const createScream_1 = (logger) => {
  return (message) => {
    logger(message.toUpperCase() + "!!!")
  }
}
```

```
// 1줄짜리 함수는 {}와 return 생략
const createScream_2 = (logger) =>
  (message) => {
    logger(message.toUpperCase() + "!!!")
  }
```

```
// 1줄짜리 함수는 {} 생략 ✓
const createScream_3 = (logger) => (message) =>
  logger(message.toUpperCase() + "!!!")
```

```
// 매개변수가 1개 이면 ()도 생략
const createScream_4 = logger => message =>
  logger(message.toUpperCase() + "!!!")
```

함수형 프로그래밍

- 함수형 프로그래밍: 함수 사용을 강조하는 소프트웨어 개발 스타일
 - 데이터의 제어흐름과 연산을 추상화 => side effect 방지, 상태 변이를 최소화 하기 위함
 - 명령형 프로그래밍과 선언형 프로그래밍 비교
- 특징
 - 불변성
 - 순수성
 - 데이터 변환
 - 고차함수
 - 재귀

명령형 프로그래밍과 선언적 프로그래밍 비교

예제) 문자열을 읽어와 공백은 '-' 문자로, 영문자는 소문자로 변환하는 코드 구현

- 입력: This is the mid day show with Cheryl Waters
- 출력: this-is-the-mid-day-show-with-cheryl-waters

명령형 프로그래밍

- 코드로 원하는 결과를 달성하는 과정에 집중

```
/* ch03-02-01-imperative-declarative.html */
```

```
var string = "This is the mid day show with Cheryl Waters"
```

```
var urlFriendly = ""
```

```
for (var i=0; i<string.length; i++) {
```

```
  if (string[i] === " ") {
```

```
    urlFriendly += "-"
```

```
  } else {
```

```
    urlFriendly += string[i]
```

```
  }
```

```
}
```

```
urlFriendly = urlFriendly.toLowerCase()
```

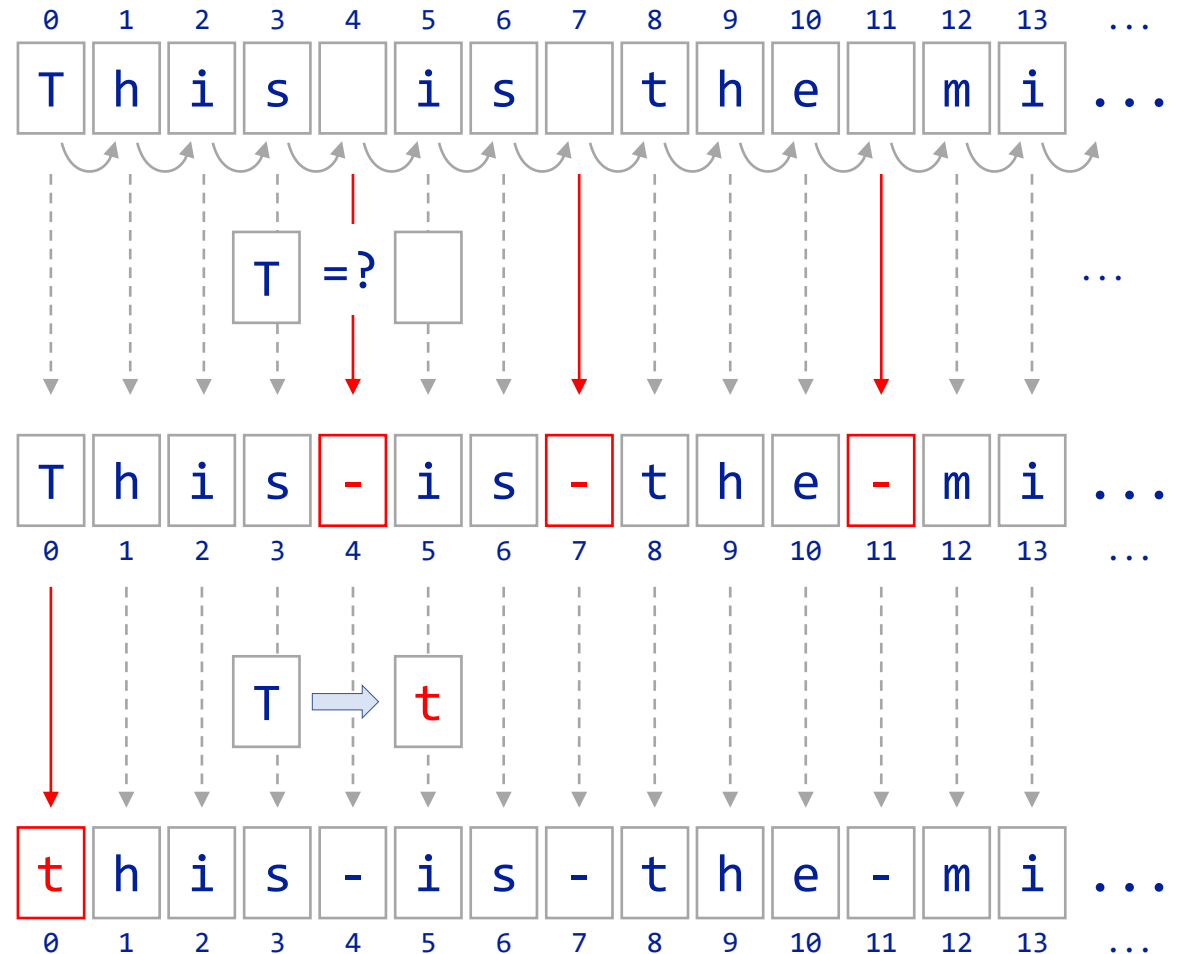
```
console.log(urlFriendly)
```

- 1) 새로운 문자열을 생성하고
- 2) 원본 문자열 길이만큼 루프를 돌면서
 각 문자를 탐색
- 3) 문자가 공백(" ")이면 -으로 변경
- 4) 그렇지 않으면 원래의 문자를 복제

- 5) 새롭게 생성한 문자열을 소문자로 변환

urlFriendly:

string:



- 필요한 결과물에 집중

```
/* ch03-02-02-imperative-declarative.html */
```

```
const string = "This is the mid day show with Cheryl Waters"
```

```
const urlFriendly = string.replace(/ /g, "-").toLowerCase()
```

```
console.log(urlFriendly)
```

동작의 결과물을 예측하기 쉬움 (가독성)
구현 과정은 추상화

string:

This is the mid day show with Cheryl Waters

- 1) 정규표현식으로 문자열에서 모든 공백 찾기: `/ /g`
- 2) 공백을 - 문자로 변환하기: `replace(A, B)`

This-is-the-mid-day-show-with-Cheryl-Waters

- 3) 모든 문자열을 소문자로 변환하기: `toLowerCase()`

urlFriendly:

this-is-the-mid-day-show-with-cheryl-waters

명령형 프로그래밍과 선언적 프로그래밍 비교: DOM 구성 예

• 명령형 프로그래밍

```
/* 03-1-dom.js */
```

DOM을 구축하는 과정에 집중

```
var target = document.getElementById("target");  
var wrapper = document.createElement("div");  
var headline = document.createElement("h1");
```

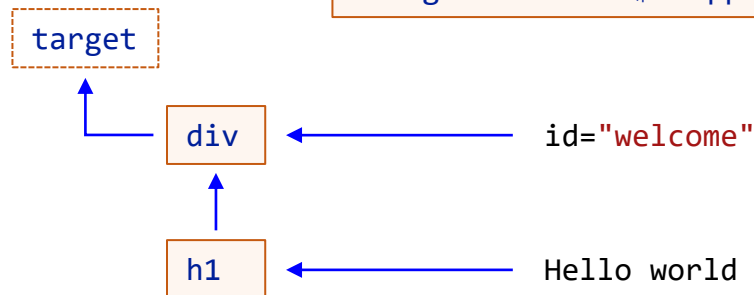
<div> element를 생성하고, id 속성 지정

```
wrapper.id = "welcome";  
headline.innerText = "Hello world";
```

```
wrapper.appendChild(headline);  
target.appendChild(wrapper);
```

<h1> element 생성
innerText 값을 지정
<div> 하위 태그로 연결

<target> element에 wrapper 객체 연결



• 선언적 프로그래밍

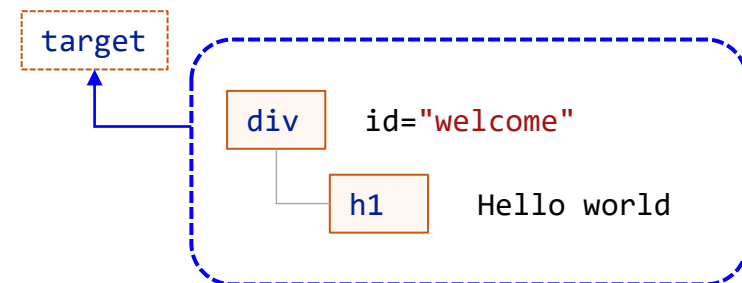
```
/* 03-2-dom.js */
```

```
const { render } = ReactDOM;  
const Welcome = () => (  
  <div id="welcome">  
    <h1>Hello world</h1>  
  </div>  
)
```

Welcome 컴포넌트: 렌더링 할 DOM 명세
※ DOM을 구성하는 과정은 추상화

```
render(<Welcome />, document.getElementById('target'));
```

Welcome 컴포넌트를 <target> element에 렌더링



불변성 (immutable, 계속)

```
/* ch03-04-01-immutability.html */
```

```
let color_lawn = {  
  title: "잔디",  
  color: "#00FF00",  
  rating: 0  
}
```

```
function rateColor(obj, rating) {  
  obj.rating = rating  
  return obj  
}
```

객체를 매개변수로 전달하고,
해당 객체에 값을 변경
변경된 객체를 반환

```
console.log(color_lawn.rating)
```

```
// rateColor는 원래의 색을 변경한다.
```

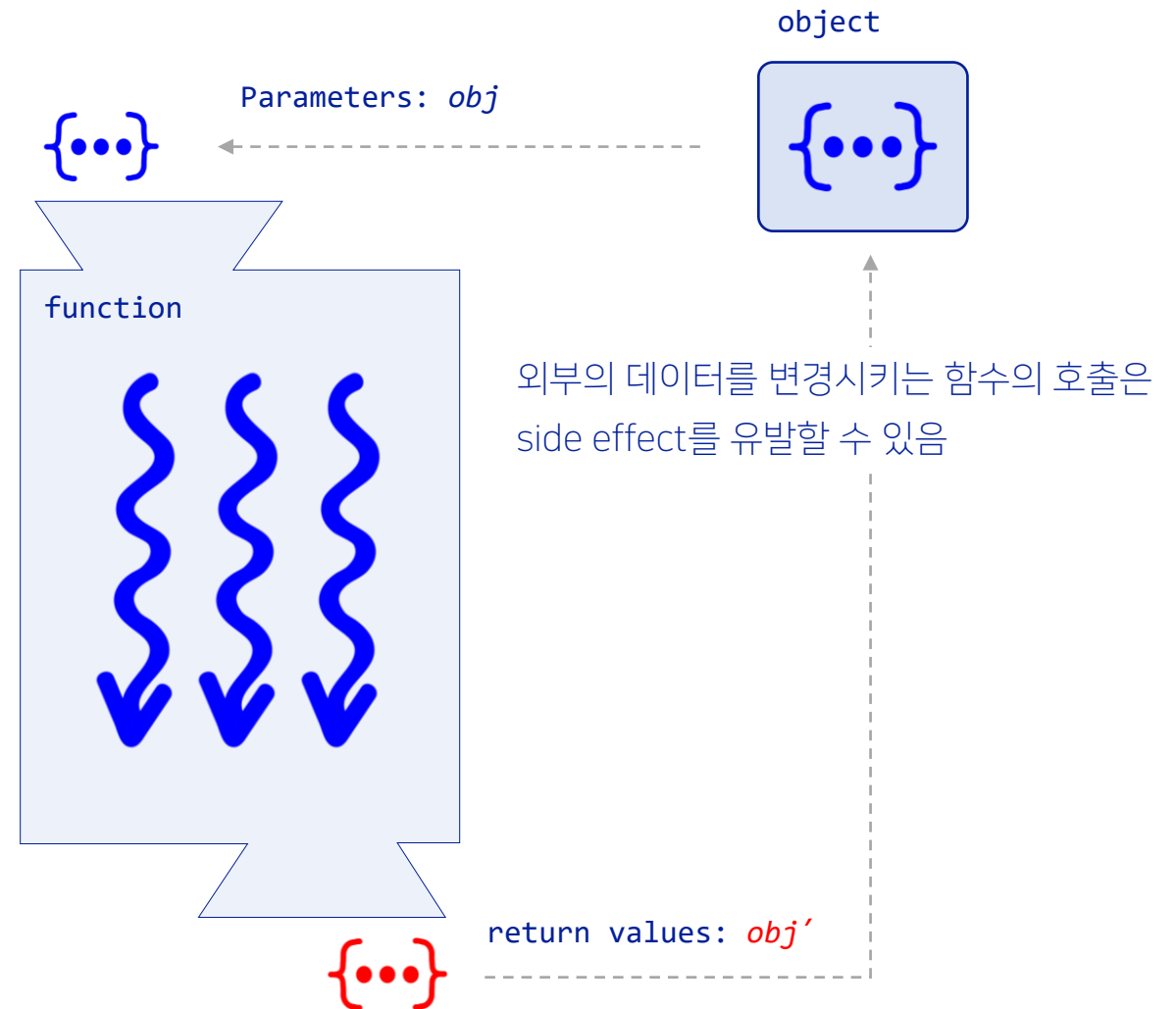
```
console.log(rateColor(color_lawn, 5).rating)
```

```
console.log(color_lawn.rating)
```

원본 수정

0
5
5

같은 객체의 값을 출력하나, 값이 변경됨
: 연산 과정에서 객체가 수정되었음을 의미



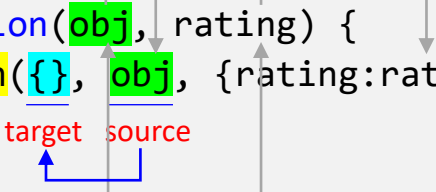
- 원본 대신 데이터 구조의 **복사본**을 만들어서 수정 및 활용
> 데이터 원본은 수정되지 않음

```
/* ch03-04-02-immutability.html */
```

```
let color_lawn = {  
  (생략)  
}
```

```
var rateColor = function(obj, rating) {  
  return Object.assign({}, obj, {rating:rating})  
}
```

원본 유지, 복사본 수정



```
console.log(color_lawn.rating)
```

```
// Object.assign으로 복사본을 만들어서 평점을 부여한다  
console.log(rateColor(color_lawn, 5).rating)  
console.log(color_lawn.rating)
```

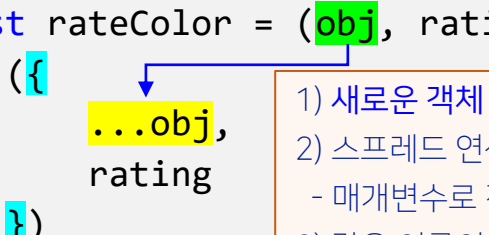
0
5
0

객체의 속성 값을 수정
: 객체의 복사본에 작업하였으므로, 원본은 유지

```
/* ch03-04-03-immutability.html */
```

```
let color_lawn = {  
  title: "잔디",  
  color: "#00FF00",  
  rating: 0  
}
```

```
const rateColor = (obj, rating) =>  
  ({  
    ...obj,  
    rating  
  })
```



- 1) 새로운 객체 생성: 원본 유지, 복사본에 작업
- 2) 스프레드 연산자 ... 활용
- 매개변수로 전달받은 객체의 모든 요소 나열
- 3) 같은 이름의 element 값 수정

```
console.log(color_lawn.rating)
```

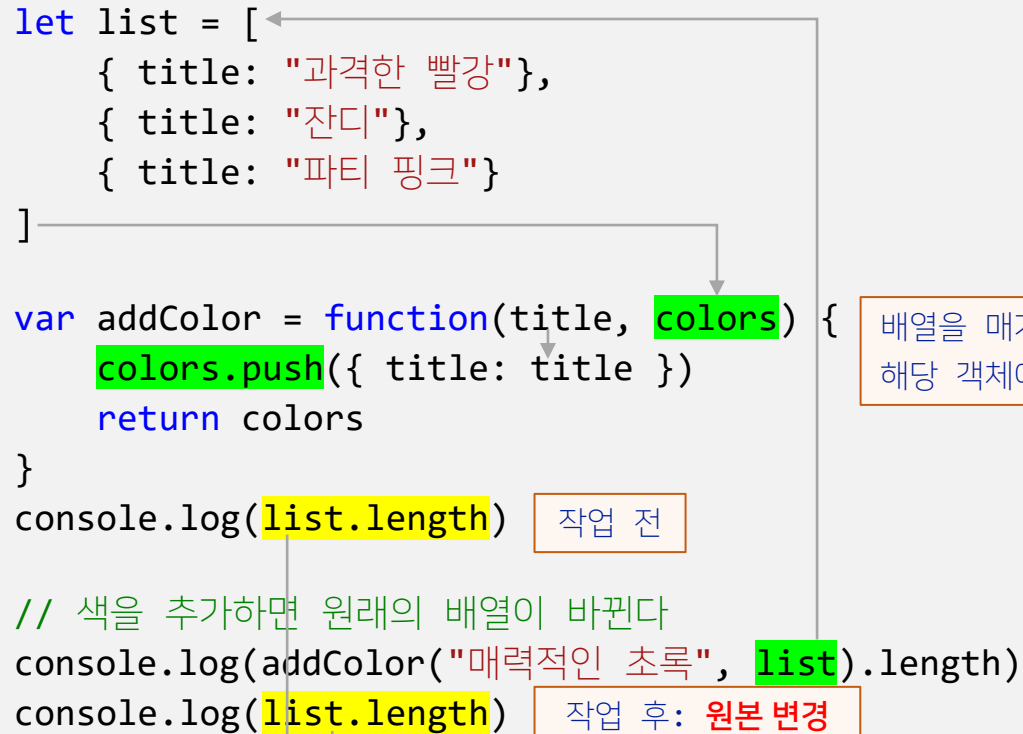
```
// 객체 스프레드 ...를 사용  
console.log(rateColor(color_lawn, 5).rating)  
console.log(color_lawn.rating)
```

0
5
0

불변성 (immutable, 계속)

```
/* ch03-04-04-immutability.html */
```

```
let list = [  
  { title: "과격한 빨강"},  
  { title: "잔디"},  
  { title: "파티 핑크"}  
]  
  
var addColor = function(title, colors) {  
  colors.push({ title: title })  
  return colors  
}  
  
console.log(list.length) // 작업 전  
  
// 색을 추가하면 원래의 배열이 바뀐다  
console.log(addColor("매력적인 초록", list).length)  
console.log(list.length) // 작업 후: 원본 변경
```



배열을 매개변수로 전달하고,
해당 객체에 요소 추가

작업 전

작업 후: 원본 변경

• `Array.prototype.push()` [🔗](#)

- 배열의 끝에 하나 이상의 **요소를 추가**하고, 배열의 새로운 길이를 반환

3
4
4



불변성 (immutable)

```
/* ch03-04-05-immutability.html */
```

```
let list = [  
  { title: "과격한 빨강"},  
  { title: "잔디"},  
  { title: "파티 핑크"}  
]
```

```
const addColor = (title, array) => array.concat({title})
```

```
console.log(list.length)
```

원본 유지, 복사본 수정

```
// array.concat을 사용하면 원래의 배열이 변경되지 않는다  
console.log(addColor("매력적인 초록", list).length)  
console.log(list.length)
```

3
4
3

배열의 원소를 추가
: 배열의 복사본에 작업하였으므로, 원본은 유지

• `Array.prototype.concat()` [🔗](#)

- 인자로 주어진 배열이나 값들을 기존 배열에 합쳐서 **새 배열을 반환**

```
/* ch03-04-06-immutability.html */
```

```
const addColor = (title, list) => [...list, {title}]
```

```
console.log(list.length)
```

```
// ... 스프레드 연산자로 배열을 복사하면 더 편리하다
```

```
console.log(addColor("매력적인 초록", list).length)
```

```
console.log(list.length)
```

3
4
3

1) 새로운 배열 생성: 원본 유지, 복사본에 작업
2) 스프레드 연산자 ... 활용
- 매개변수로 전달받은 배열의 모든 요소 나열
3) 함께 전달된 요소 추가

순수 함수 (pure functions, 계속)

```
/* ch03-05-01-pure-functions.html */
```

```
var frederick = {  
  name: "Frederick Douglass",  
  canRead: false,  
  canWrite: false  
}
```

```
function selfEducate() {  
  frederick.canRead = true  
  frederick.canWrite = true  
}
```

selfEducate()

순수하지 않은 함수

- 1) 매개변수 없음
- 2) 반환 값 없음
- 3) 함수 밖 객체의 값 (속성)을 변경

side effect 발생

```
console.log( frederick )
```

```
{name: 'Frederick Douglass', canRead: true, canWrite: true}
```

- 매개변수에 의해서만 반환 값이 결정되는 함수
> 데이터 원본은 수정되지 않음

```
/* ch03-05-02-pure-functions.html */
```

```
var frederick = { (생략) }
```

```
const selfEducate = person => {  
  person.canRead = true  
  person.canWrite = true  
  return person  
}
```

순수하지 않은 함수

- 매개변수와 반환 값은 존재하나,
1) 매개변수로 전달된 함수 밖 객체의 (속성) 값을 변경

```
console.log( selfEducate( frederick ) )  
console.log( frederick )
```

side effect 발생

```
{name: 'Frederick Douglass', canRead: true, canWrite: true}  
{name: 'Frederick Douglass', canRead: true, canWrite: true}
```

순수 함수 (pure functions, 계속)

```
/* ch03-05-03-pure-functions.html */
```

```
var frederick = {  
  name: "Frederick Douglass",  
  canRead: false,  
  canWrite: false  
}
```

```
const selfEducate = person =>  
  ({  
    ...person,  
    canRead: true,  
    canWrite: true  
  })
```

```
console.log( selfEducate(frederick) )  
console.log( frederick )
```

- 1) 새로운 객체 생성
- 2) 스프레드 연산자 ... 활용
- 매개변수로 전달받은 객체의 모든 요소 나열
- 3) 같은 이름의 element 값 수정
- 4) 복사본에 작업하고 결과 반환 (원본 유지)

순수 함수

- 1) 매개변수 전달
- 2) 결과 값 반환
- 3) 함수 밖 객체의 값을 변경하지 않음

side effect 발생하지 않음

```
{name: "Frederick Douglass", canRead: true, canWrite: true}  
{name: "Frederick Douglass", canRead: false, canWrite: false}
```

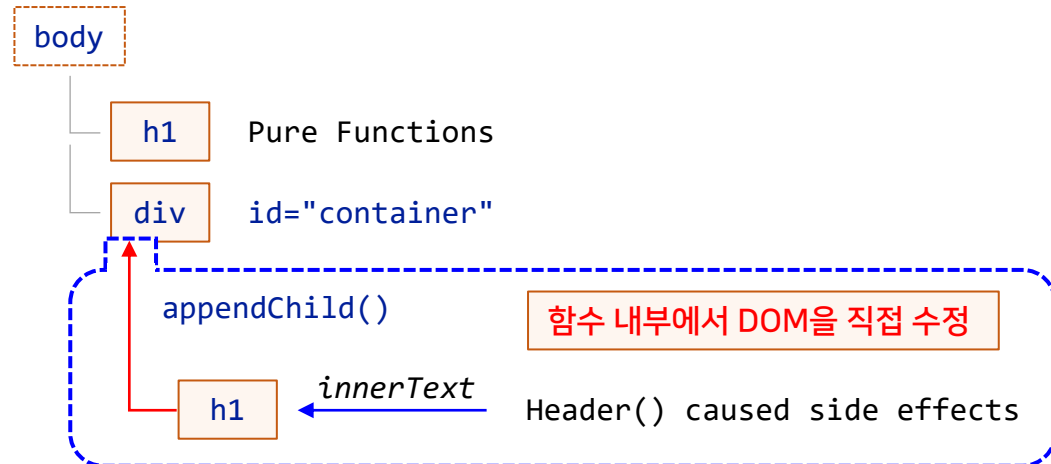
순수 함수 (pure functions)

```
/* ch03-05-04-1-pure-functions.html */
function Header(text) {
  const h1 = document.createElement('h1')
  h1.innerText = text
  const div = document.getElementById("container")
  div.appendChild(h1)
}
```

함수 내부에서 h1 element를 생성하였으나,
DOM을 변화시킴

side effect 발생

Header("Header() caused side effects")



```
/* ch03-05-05-pure-functions.html */
const Header = (props) => <h1>{props.title}</h1>
```

Header():

- props를 매개변수로 받아서 새로운 <h1> 객체를 반환
- DOM에 직접적인 변화시키지 않음

```
ReactDOM.render(
  <Header title="React uses pure functions" />,
  document.getElementById('react-container')
)
```

Header 컴포넌트를 <div> element에 렌더링

