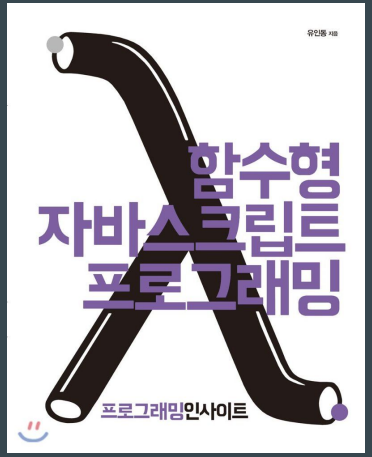
함수형 자바스크립트 #1

•••

ABCD 한성일



함수형 자바스크립트 프로그래밍
http://www.yes24.com/Product/Goods/56885507?Acode=101

책 소스
https://github.com/indongyoo/functional-javascript

이 책을 기반으로 스터디를 진행합니다.

tool

https://runjs.dev/

https://code.visualstudio.com/

1. 함수형 자바스크립트 소개

react hook vs classes

https://medium.com/better-programming/react-hooks-vs-classes-add2676a32f2

소스코드

https://github.com/ABoutCoDing/Functional-Javascript

addMaker

```
function addMaker(a) {
  return function(b) {
    return a + b;
  };
}
addMaker(10)(5); // 15



// addMaker로 만든 함수
var add5 = addMaker(5);
add5(3); // 8
add5(4); // 9
```

값으로서의 함수와 클로저

```
var v1 = 100;
var v2 = function() {};

function f1() {
  return 100;
}
function f2() {
  return function() {};
}
```

```
function addMaker(a) {
  return function(b) {
    return a + b;
  };
}

addMaker(10)(5); // 15
var add5 = addMaker(5);
add5(3); // 8
add5(4); // 9
var add3 = addMaker(3);
add3(3); // 6
add3(4); // 7
```

for 문으로 필터링 하기

```
var users = [
  { id: 1, name: "ID", age: 32 },
  { id: 2, name: "HA", age: 25 },
  { id: 3, name: "BJ", age: 32 },
  { id: 4, name: "PJ", age: 28 },
  { id: 5, name: "JE", age: 27 },
  { id: 6, name: "JM", age: 32 },
  { id: 7, name: "HI", age: 24 }
];
var temp_users = [];
for (var i = 0, len = users.length; i < len; i++) {
  if (users[i].age < 30) temp_users.push(users[i]);
}
console.log(temp_users.length);
// 4
var ages = [];
for (var i = 0, len = temp_users.length; i < len; i++) {
  ages.push(temp_users[i].age);
}
console.log(ages);
// [25, 28, 27, 24]

...
```

검색조건에 따라
코드 중복이 발생 한다.

age, name 순...

for에서 filter로 if 에서 predicate로

```
var temp_users = [];
for (var i = 0, len = users.length; i < len; i++) {
  if (users[i].age < 30) temp_users.push(users[i]);
}
console.log(temp_users.length); // 4
```

```
function filter(list, predicate) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i])) new_list.push(list[i]);
  }
  return new_list;
}
```

filter 사용

```
// predicate
var users_under_30 = filter(users, function(user) {
  return user.age < 30;
});
console.log(users_under_30.length);
// 4
var ages = [];
for (var i = 0, len = users_under_30.length; i < len; i++) {
  ages.push(users_under_30[i].age);
}
console.log(ages);
// [25, 28, 27, 24]
// predicate
var users_over_30 = filter(users, function(user) {
  return user.age >= 30;
});
console.log(users_over_30.length);
// 3
var names = [];
for (var i = 0, len = users_over_30.length; i < len; i++) {
  names.push(users_over_30[i].name);
}
console.log(names);
// ["ID", "BJ", "JM"]
```

함수형 관점에서 필터 다시보기

인자가 들어오면 항상 동일하게 동작
filter 함수의 로직은 외부나 내부의 어떤 상태변화에도 의존하지 않는다.

이전 값의 상태를 변경하지 않고 새로운 값을 리턴한다. (new_list)
    불변의 데이터 구조(immutable data structures)

new_list는 이 함수에서 최초로 만들어졌고 외부의 어떠한 쇙황이나 상태와도 무관하다.
new_list의 결과도 달라질 수 없다. new_list가 완성되고 나면 new_list를 리턴 하고 filter는 완전히 종료된다.

filter는 if의 predicate 의 결과에만 의존한다. 별도의 로직이 없고 단순하고 쉽다.
predicate에서도 값을 변경하지 않는다. true 인지 false인지를 filter의 if 에게 전달하는 일만 한다.
호출 코드에는 for도 없고 if도 없다.

```
filter(users, function(user) {return user.age < 30});
```

함수형 관점에서 필터 다시보기

절차지향 프로그래밍에서는 위에서 아래로 내려가면서 특정 변수의 값을 변경해 나가는 식으로 로직을 만든다.
객체지향 프로그래밍에서는 객체들을 만들어 놓고 객체들 간의 협업을 통해 로직을 만든다. 이벤트 등으로 서로를 연결한 후 상태의 변화를 감지하여 스스로 자신이 가진 값을 변경하거나 상대의 메서드를 직접 실행하여 상태를 변경하는 식으로 프로그래밍 한다.

함수형 프로그래밍에서는 '항상 동일하게 동작하는 함수'를 만들고 보조함수를 조합하는 식으로 로직을 완성한다.
내부에서 관리하고 있는 상태를 따로 두지 않고 넘겨진 인자에만 의존한다.
동일한 인자가 들어오면 항상 동일한 값을 리턴하도록 한다.
보조함수 역시 인자이며, 보조 함수 에서도 상태를 변경하지 않으면 보조함수를 받는 함수는 항상 동일한 결과를 만드는 함수가 된다.

함수형 관점에서 필터 다시보기

객체지향 언어에서도 새로운 객체를 만들어 부수효과를 줄일 수 있다.
그러나 많은 객체를 만드는 것은 운용도 어렵고 객체지향과 어울리지 않는다.

자신의 상태를 메서드를 통해 변경하는 것은 객체지향의 단점이 아니라 객체지향의 방법 그 자체이다.

함수형 프로그래밍은 부수효과를 최소화 하는 것이 목표이다.

함수형 프로그래밍은 객체지향과 함께 동작해야한다.
현대 프로그래밍에서 다루는 값은 대부분 객체
기능 확장을 객체의 확장으로 풀어가느냐 함수의 확장으로 풀어가느냐의 차이이다.

map 함수

```
var ages = [];
for (var i = 0, len = users_under_30.length; i < len; i++) {
  ages.push(users_under_30[i].age);
}
console.log(ages);
var names = [];
for (var i = 0, len = users_over_30.length; i < len; i++) {
  names.push(users_over_30[i].name);
}
console.log(names);
```

```
function map(list, iteratee) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    new_list.push(iteratee(list[i]));
  }
  return new_list;
}
```

map 함수

```
var users_under_30 = filter(users, function(user) {
  return user.age < 30;
});
console.log(users_under_30.length);
// 4
// iteratee
var ages = map(users_under_30, function(user) {
  return user.age;
});
console.log(ages);
// [25, 28, 27, 24]
var users_over_30 = filter(users, function(user) {
  return user.age >= 30;
});
console.log(users_over_30.length);
// 3
// iteratee
var names = map(users_over_30, function(user) {
  return user.name;
});
console.log(names);
// ["ID", "BJ", "JM"]
```

함수 중첩 1

#1

```
var ages = map(
  filter(users, function(user) {
    return user.age < 30;
  }),
  function(user) {
    return user.age;
  }
);
console.log(ages.length);
// 4
console.log(ages);
// [25, 28, 27, 24]
var names = map(
  filter(users, function(user) {
    return user.age >= 30;
  }),
  function(user) {
    return user.name;
  }
);
console.log(names.length);
// 3
console.log(names);
// ["ID", "BJ", "JM"]
```

함수 중첩 2

```
function log_length(value) {
  console.log(value.length);
  return value;
}

console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age < 30;
      }),
      function(user) {
        return user.age;
      }
    )
  )
);
// 4
// [25, 28, 27, 24]
```

```
console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age >= 30;
      }),
      function(user) {
        return user.name;
      }
    )
  )
);
// 3
// ["ID", "BJ", "JM"]
```

변수 할당도 모두 없앨 수 있다.

filter, map

```
function filter(list, predicate) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i])) new_list.push(list[i]);
  }
  return new_list;
}

function map(list, iteratee) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    new_list.push(iteratee(list[i]));
  }
  return new_list;
}

function log_length(value) {
  console.log(value.length);
  return value;
}
```

```
console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age < 30;
      }),
      function(user) {
        return user.age;
      }
    )
  )
);
console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age >= 30;
      }),
      function(user) {
        return user.name;
      }
    )
  )
);
```

함수를 리턴하는 함수 bvalue

```
function addMaker(a) {
  return function(b) {
    return a + b;
  };
}

function bvalue(key) {
  return function(obj) {
    return obj[key];
  };
}

bvalue("a")({ a: "hi", b: "hello" }); // hi
```

bvalue를 실행할 때 넘겨준 인자 key 를 나중에 obj를 받을 익명함수가 기억한다.
bvalue의 실행 결과는 key를 기억하는 함수이고 이 함수에는 key/value 쌍으로 구성된 객체를 인자로 넘길 수 있다.

bvalue로 map의 iteratee 만들기

```
console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age < 30;
      }),
      bvalue("age")
    )
  )
);
console.log(
  log_length(
    map(
      filter(users, function(user) {
        return user.age >= 30;
      }),
      bvalue("name")
    )
  )
);
```

map이 사용할 iteratee 함수를 bvalue가 리턴한 함수로 대체했다.

화살표 함수와 함께

https://github.com/ABoutCoDing/Functional-Javascript/blob/master/1.2%20%ED%95%A8%EC%88%98%ED%98%95%20%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8%EC%9D%98%20%EC%8B%A4%EC%9A%A9%EC%84%B1.js#L272

filter로 한명 찾기

```
var users = [
  { id: 1, name: "ID", age: 32 },
  { id: 2, name: "HA", age: 25 },
  { id: 3, name: "BJ", age: 32 },
  { id: 4, name: "PJ", age: 28 },
  { id: 5, name: "JE", age: 27 },
  { id: 6, name: "JM", age: 32 },
  { id: 7, name: "HI", age: 24 }
];

console.log(
  filter(users, function(user) {
    return user.id == 3;
  })[0]
);
// { id: 3, name: "BJ", age: 32 }
```

```
var user;
for (var i = 0, len = users.length; i < len; i++)
{
  if (users[i].id == 3) {
    user = users[i];
    break;
  }
}
console.log(user);
// { id: 3, name: "BJ", age: 32 }
```

findById

```
function findById(list, id) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (list[i].id == id) return list[i];
  }
}
console.log(findById(users, 3));
// { id: 3, name: "BJ", age: 32 }
console.log(findById(users, 5));
// { id: 5, name: "JE", age: 27 }
```

findByName

```
// 1-19 findByName
function findByName(list, name) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (list[i].name == name) return list[i];
  }
}
console.log(findByName(users, "BJ"));
// { id: 3, name: "BJ", age: 32 }
console.log(findByName(users, "JE"));
// { id: 5, name: "JE", age: 27 }
```

findByAge

```
function findByAge(list, age) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (list[i].age == age) return list[i];
  }
}
console.log(findByAge(users, 28));
// { id: 4, name: "PJ", age: 28 }
console.log(findByAge(users, 25));
// { id: 2, name: "HA", age: 25 }
```

findBy

```
function findBy(key, list, val) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (list[i][key] === val) return list[i];
  }
}
console.log(findBy("name", users, "BJ"));
// { id: 3, name: "BJ", age: 32 }
console.log(findBy("id", users, 2));
// { id: 2, name: "HA", age: 25 }
console.log(findBy("age", users, 28));
// { id: 4, name: "PJ", age: 28 }
```

findBy로 안되는 경우

```
function User(id, name, age) {
  this.getId = function() {
    return id;
  };
  this.getName = function() {
    return name;
  };
  this.getAge = function() {
    return age;
  };
}
```

```
var users2 = [
  new User(1, "ID", 32),
  new User(2, "HA", 25),
  new User(3, "BJ", 32),
  new User(4, "PJ", 28),
  new User(5, "JE", 27),
  new User(6, "JM", 32),
  new User(7, "HI", 24)
];
function findBy(key, list, val) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (list[i][key] === val) return list[i];
  }
}
console.log(findBy("age", users2, 25));
// undefined
```

key가 아닌 메서드를 통해 값을 얻어야 할 때
두가지 이상의 조건이 필요할 때
=== 이 아닌 다른 조건으로 찾고자 할 때

find

```
function find(list, predicate) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i])) return list[i];
  }
}

console.log(
  find(users2, function(u) {
    return u.getAge() == 25;
  }).getName()
);
```

인자로 키와 값 대신 함수를 사용하면 모든 상황에 가능한 find 함수를 얻을 수 있다.

```
// HA
console.log(
  find(users, function(u) {
    return u.name.indexOf("P") != -1;
  })
);
// { id: 4, name: "PJ", age: 28 }
console.log(
  find(users, function(u) {
    return u.age == 32 && u.name == "JM";
  })
);
// { id: 6, name: "JM", age: 32 }
console.log(
  find(users2, function(u) {
    return u.getAge() < 30;
  }).getName()
);
// HA
```

다형성

```
console.log(
  map(
    filter(users, function(u) {
      return u.age >= 30;
    }),
    function(u) {
      return u.name;
    }
  )
);
// ["ID", "BJ", "JM"];
console.log(
  map(
    filter(users2, function(u) {
      return u.getAge() >= 30;
    }), // 메서드 실행으로 변경
    function(u) {
      return u.getName();
    }
  )
); // 메서드 실행으로 변경
// ["ID", "BJ", "JM"];
```

객체 지향 프로그래밍의 약속된 이름의 메서드를 대신 실행해주는 식으로 외부 객체에게 위임 하면 함수형 프로그래밍에서는 보조함수를 통해 완전히 위임하는 방식을 취한다. 이는 더 높은 다형성과 안정성을 보장한다.

bmatch1로 predicate 만들기

```
function bmatch1(key, val) {
  return function(obj) {
    return obj[key] === val;
  };
}

console.log(find(users, bmatch1("id", 1)));
// {id: 1, name: "ID", age: 32}
console.log(find(users, bmatch1("name", "HI")));
// {id: 7, name: "HI", age: 24}
console.log(find(users, bmatch1("age", 27)));
// {id: 5, name: "JE", age: 27}
```

bmatch1의 실행결과는 함수
key와 val을 미리 받아서 나중에 들어올 obj와 비교하는 익명 함수를 클로저로 만들어 리턴

bmatch1로 함수를 만들어 고차 함수와 협업하기

```
console.log(filter(users, bmatch1("age", 32)));
// [{ id: 1, name: "ID", age: 32},
//  { id: 3, name: "BJ", age: 32},
//  { id: 6, name: "JM", age: 32}]

console.log(map(users, bmatch1("age", 32)));
// [true, false, true, false, false, true, false]
```

bmatch

```
function object(key, val) {
  var obj = {};
  obj[key] = val;
  return obj;
}
function match(obj, obj2) {
  for (var key in obj2) {
    if (obj[key] !== obj2[key]) return false;
  }
  return true;
}
function bmatch(obj2, val) {
  if (arguments.length == 2) obj2 = object(obj2, val);
  return function(obj) {
    return match(obj, obj2);
  };
}
console.log(
  match(find(users, bmatch("id", 3)), find(users, bmatch("name", "BJ")))
);
// true
console.log(
  find(users, function(u) {
    return u.age == 32 && u.name == "JM";
  })
);
// { id: 6, name: "JM", age: 32 }
console.log(find(users, bmatch({ name: "JM", age: 32 })));
// { id: 6, name: "JM", age: 32 }
```

여러개의 key에 해당하는 value들을 비교하는 함수

findIndex

```
function findIndex(list, predicate) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i])) return i;
  }
  return -1;
}
console.log(findIndex(users, bmatch({ name: "JM", age: 32 })));
// 5
console.log(findIndex(users, bmatch({ age: 36 })));
// -1
```

Array.prototype.indexOf보다 활용도가 높은 findIndex 함수를 만들 수 있다.

고차함수

고차함수(Higher-Order Function)

함수를 인자로 받거나 함수를 리턴하는 함수
둘다하는 경우도 고차함수

`map, filter, find, findIndex ...`

인자 늘리기

```
var _ = {};
_.map = function(list, iteratee) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    new_list.push(iteratee(list[i], i, list));
  }
  return new_list;
};
_.filter = function(list, predicate) {
  var new_list = [];
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i], i, list)) new_list.push(list[i]);
  }
  return new_list;
};
_.find = function(list, predicate) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i], i, list)) return list[i];
  }
};
_.findIndex = function(list, predicate) {
  for (var i = 0, len = list.length; i < len; i++) {
    if (predicate(list[i], i, list)) return i;
  }
  return -1;
};
```

underscore의 _.map, _.filter, _.find, _.findIndex에 가깝게 수정

iteratee(list[i], i, list)처럼 두개의 인자를 추가

predicate에서 두 번째 인자 사용하기

```
console.log(
  _.filter([1, 2, 3, 4], function(val, idx) {
    return idx > 1;
  })
);
// [3, 4]

console.log(
  _.filter([1, 2, 3, 4], function(val, idx) {
    return idx % 2 == 0;
  })
);
// [1, 3]
```

function identity(v) {return v;}

```
_.identity = function(v) {
  return v;
};
var a = 10;
console.log(_.identity(a));
// 10
```

_.identity 받은 인자를 그대로 뱉는 함수

```
console.log(_.filter([true, 0, 10, "a", false, null], _.identity));
// [true, 10, 'a']
```

_.filter에 _.identity와 함께 사용하면 Truth Values로 평가되는 값들 만 남는다.

falsy와 truthy

```
_.falsy = function(v) {return !v;};
_.truthy = function(v) {return !!v;};
```

some, every 만들기

```
_.some = function(list) {
  return !!_.find(list, _.identity);
};
_.every = function(list) {
  return _.filter(list, _.identity).length == list.length;
};
console.log(_.some([0, null, 2])); // true
console.log(_.some([0, null, false])); // false
console.log(_.every([0, null, 2])); // false
console.log(_.every([{}, true, 2])); // true
```

_.some : 배열에 들어 있는 값 중 하나라도 긍정적인 값이 있으면 true 하나도 없다면 false 리턴
_.every : 모두 긍정적인 값이어야 true 리턴 if나 predicate 등과 함께 사용할 때 유용하다.

연산자 대신 함수로

```
function not(v) {
  return !v;
}
function beq(a) {
  return function(b) {
    return a === b;
  };
}
```

아주 작은 함수 not, beq
!를 써도 되는데 not이 왜 필요할 까
=== 로 비교하면 되는데 beq는 왜 필요할까?

some, every 만들기2

```
_.some = function(list) {
  return !!_.find(list, _.identity);
};
_.every = function(list) {
  return beq(-1)(_.findIndex(list, not));
};
console.log(_.some([0, null, 2])); // true
console.log(_.some([0, null, false])); // false
console.log(_.every([0, null, 2])); // false
console.log(_.every([{}, true, 2])); // true
```

not 은 연산자 !가 아닌 함수이기 때문에 _.findIndex와 함께 사용할 수 있다.
list의 값 중 하나라도 부정적인 값을 만나면 predicate가 not 이므로 true를
리턴하여 해당 번째 i값을 리턴하게 된다.

함수 쪼개기

```
function positive(list) {
  return _.find(list, _.identity);
}
function negativeIndex(list) {
  return _.findIndex(list, not);
}
_.some = function(list) {
  return not(not(positive(list)));
};
_.every = function(list) {
  return beq(-1)(negativeIndex(list));
};
console.log(_.some([0, null, 2])); // true
console.log(_.some([0, null, false])); // false
console.log(_.every([0, null, 2])); // false
console.log(_.every([{}, true, 2])); // true
```

좀더 함수를 쪼개면 함수가 한가지 일만 하게 된다.

함수 합성

함수를 쪼갤 수 록 함수 합성이 쉬워진다.
다양한 함수 합성 기법 중 하나인 Underscore.js 의 _.compose

해당 함수의 결과를 자신의 왼쪽의 함수 에게 전달한다.
그리고 해당함수의 결과를 다시 자신의 왼쪽의 함수에게 전달하는 고차 함수

_.compose

```
// underscore.js 중
_.compose = function() {
  var args = arguments;
  var start = args.length - 1;
  return function() {
    var i = start;
    var result = args[start].apply(this, arguments);
    while (i--) result = args[i].call(this, result);
    return result;
  };
};
var greet = function(name) {
  return "hi: " + name;
};
var exclaim = function(statement) {
  return statement.toUpperCase() + "!";
};
var welcome = _.compose(greet, exclaim);
welcome("moe");
// 'hi: MOE!'
```

welcome 을 실행하면 먼저 exclaim을 실행하면서 "moe"를 인자로 넘겨준다. exclaim의 결과는 대문자로 변환된 "MOE!"이고 그 결과가 다시 greet의 인자로 넘어가 최종 결과로 "hi: MOE!"를 리턴한다.

_.compose로 함수 합성하기

```
// 원래 코드
_.some = function(list) {
  return not(not(positive(list)));
};
_.every = function(list) {
  return beq(-1)(negativeIndex(list));
};
```

```
_.some = _.compose(not, not, positive);
_.every = _.compose(beq(-1), negativeIndex);
```

_.compose로 _.some과 _.every를 더 간결하게 표현했다.
원래 코드와 동일하게 동작한다.
맨 오른쪽의 함수가 인자를 받아 결과를 만들고 결과는 다시 그 왼쪽의 함수에게 인자로 전달. 오른쪽에서부터 왼쪽으로 연속적으로 실행되어 최종 결과를 얻는다.

함수 합성 장점

짧고 읽기 좋은 코드도 중요한 가치이지만 좀 더 고상한 이점이 있다. 인자 선언이나 변수 선언이 적어진다는 점이다. 코드에 인자와 변수가 등장하지 않고 함수의 내부(`{statements}`)가 보이지 않는다는 것은 새로운 상황도 생기지 않는다는 말이다. 새로운 상황이 생기지 않는다는 것은 개발자가 예측하지 못할 상황이 없다는 말이다. 에러 없는 함수들이 인자와 결과에 맞게 잘 조합되어 있다면 전체의 결과 역시 에러가 날 수 없다. 상태를 공유하지 않는 작은 단위의 함수들은 테스트하기도 쉽고 테스트 케이스를 작성하기도 쉽다.

대규모 애플리케이션을 개발하다 보면 새로운 요구 사항을 반영하거나 버그를 잡거나 성능을 높이기 위해 이미 완성되었던 코드를 고쳐야 할 때가 많다. 배포된지 시간이 좀 지난 코드라면 자신이 작성한 코드일지라도 그 내용이 어림짐작만 될 뿐 아주 익숙하지 않을 것이다. 만일 지역 변수와 `if`문이 많고 중간에 `for`문도 몇 번 나오고, `j++`도 있으며 `push` 등으로 상태를 변경하는 코드라면, 당시 고려했던 모든 상황들을 다시 머릿속에 그리기 어렵다. 코드 윗부분에서 선언된 지역 변수가 특정 부분의 `if`문 안쪽에서 사용되고, 중간에 값이 변경된 다음, 그 아래 어딘가의 `if`문에서 또 사용된다면 어느 부분이 망가질지 몰라 선뜻 손대기가 어렵다.

만일 지역 변수도 없고 `if, else, for`문도 없고, 커스텀 객체의 메서드도 없고, 인자 외의 외부 상태에 의존하고 있지 않다면, 자신이 고쳐야 하는 함수의 문제에 만 집중할 수 있다. 인자와 변수 자체가 적을수록, 함수의 `{statements}`가 없거나 짧을수록, 함수들의 복잡성도 줄어들고 오류가 생길 가능성도 줄어들며 부수 효과도 줄어든다. 또한 작성한 지 오래된 코드일지라도 다시 읽고 고치기가 쉬워진다. 함수 하나하나가 무슨 일을 하는지에 대해 인자와 결과 위주로만 생각하면서 읽고 고치면 되기 때문이다.

작게 쪼개다 보면 정말 쓸모 없어 보이는 함수가 많이 나오기도 한다. 그래도 더 작은 단위로 쪼개 보라. 재사용성이 높고 재밌는 코드들이 나올 것이다. 제어문 대신 함수를, 값 대신 함수를, 연산자 대신 함수를 사용해 보자.

함수형 자바스크립트를 위한 기초

일급 함수

일급함수 (First-class Function)

- 변수에 담을 수 있다.
- 함수나 메서드의 인자로 넘길 수 있다.
- 함수나 메서드에서 리턴 할 수 있다.

자바스크립트에서 모든 값은 일급

- 아무 때나(런타임에서도) 선언이 가능하다.
- 익명으로 선언할 수 있다.
- 익명으로 선언한 함수도 함수나 메서드의 인자로 넘길 수 있다.

일급 함수

```
// (1)
function f1() {}
var a = typeof f1 == "function" ? f1 : function() {};

// (2)
function f2() {
  return function() {};
}

// (3)
(function(a, b) {
  return a + b;
})(10, 5);
// 15

// (4)
function callAndAdd(a, b) {
  return a() + b();
}
callAndAdd(
  function() {
    return 10;
  },
  function() {
    return 5;
  }
);
// 15
```

(1) `f1`은 함수를 값으로 다룰 수 있음을 보여준다. `typeof` 연산자를 사용하여 `'function'` 인지 확인하고 변수 `a`에 `f1`를 담고 있다.
(2) `f2`는 함수를 리턴한다.
(3) `a`와 `b`를 더하는 익명 함수를 선언하였으며, `a`와 `b`에 각각 `10, 5`를 전달하여 즉시 실행했다.
(4) `callAndAdd`를 실행하면서 익명 함수들을 선언했고 바로 인자로 사용되었다. `callAndAdd`는 넘겨받은 함수 둘을 실행하여 결과들을 더한다.

클로저

> 클로저는 자신이 생성될 때의 환경을 기억하는 함수다.

이 말을 보다 실용적으로 표현해 보면 '클로저는 자신의 상위 스코프의 변수를 참조할 수 있다'고 할 수 있다.
맞는 말이지만 오해의 소지가 많은 표현이다. 오해의 소지를 좀 더 줄인 정의를 만들어 보면 다음 정도이다.

> 클로저는 자신이 생성될 때의 스코프에서 알 수 있었던 변수를 기억하는 함수다.

클로저

```
function parent() {
  var a = 5;
  function myfn() {
    console.log(a);
  }
  //... 생략
}
// 혹은
function parent2() {
  var a = 5;
  function parent1() {
    function myfn() {
      console.log(a);
    }
    //... 생략
  }
  //... 생략
}
```

> 클로저로 만들 함수가 `myfn`이라면, `myfn` 내부에서 사용하고 있는 변수 중에 `myfn` 내부에서 선언되지 않은 변수가 있어야 한다. 그 변수를 `a`라고 하면 `a`라는 이름의 변수가 `myfn`을 생성하는 스코프에서 선언되었거나 알 수 있어야 한다.

본 예제의 조건을 벗어나는 함수라면 일반 함수와 다를 바 없다.

상위 스코프에서 알 수 있는 변수를 자신이 사용하고 있지 않다면 그 환경을 기억해야 할 필요가 없다.

클로저

v8과 파이어폭스는 내부 함수가 사용하는 함수중 외부 스코프의 변수가 하나도 없는 경우 클로저가 되지 않는다. 클로저가 된 경우에도 자신이 사용한 변수만 기억하며 외부 스코프의 나머지 변수는 전혀 기억하지 않는다.
그 외의 환경에서도 클로저를 외부로 리턴하여 지속적으로 참조해야만 메모리에 남는다. 그렇지 않다면 모두 가비지 컬렉션 대상이 된다.

클로저는 자신이 생성될 때의 스코프에서 알 수 있었던 변수 중 언젠가 자신이 실행될 때 사용 할 변수들만 기억하여 유지시키는 함수이다.

클로저 인가? #1

```
var a = 10;
var b = 20;
function f1() {
  return a + b;
}
f1();
// 30
```

글로벌 스코프에서 선언된 모든 변수는 그 변수를 사용하는 함수가 있는지 없는지와 관계없이 유지된다.
a와 b변수가 f1에 의해 사라지지 못하는 상황이 아니니 f1은 클로저가 아니다.

만약 node.js에서 사용할 특정 js파일에 작성되어 있었다면 f1은 클로저다.

클로저 인가? #2

```
function f2() {
  var a = 10;
  var b = 20;
  function f3(c, d) {
    return c + d;
  }
  return f3;
}
var f4 = f2();
f4(5, 7);
// 12
```

클로저가 아니다. 일단 `f1`은 클로저처럼 외부 변수를 참조하여 결과를 만든다. 게다가 상위 스코프의 변수를 사용하고 있으므로 앞서 강조했던 조건을 모두 충족시키고 있다. 그런데 왜 클로저가 아니라는 걸까?

글로벌 스코프에서 선언된 모든 변수는 그 변수를 사용하는 함수가 있는지 없는지와 관계없이 유지된다. `a`와 `b` 변수가 `f1`에 의해 사라지지 못하는 상황이 아니므로 `f1`은 클로저가 아니다.

클로저 인가? #3

```
function f4() {
  var a = 10;
  var b = 20;
  function f5() {
    return a + b;
  }
  return f5();
}
f4();
// 30
```

클로저가 있을까? 정확한 표현은 '있었다'가 되겠다. 결과적으로는 클로저는 없다고 볼 수 있다.

`f4`가 실행되고 `a`, `b`가 할당된 후 `f5`가 정의된다. 그리고 `f5`에서는 `a`, `b`가 사용되고 있으므로 `f5`는 자신이 생성된 환경을 기억하는 클로저가 된다. 그런데 `f4`의 마지막 라인을 보면 `f5`를 실행하여 리턴한다. 결국 `f5`를 참조하고 있는 곳이 어디에도 없기 때문에 `f5`는 사라지게 되고 `f5`가 사라지면 `a`, `b`도 사라질 수 있기에 클로저는 `f4`가 실행되는 사이에만 생겼다가 사라진다.

클로저 인가? #4

```
function f6() {
  var a = 10;
  function f7(b) {
    return a + b;
  }
  return f7;
}
var f8 = f6();
f8(20);
// 30
f8(10);
// 20
```

`f7`은 진짜 클로저다. 이제 `a`는 사라지지 않는다. `f7`이 `a`를 사용하기에 `a`를 기억해야하고 `f7`이 `f8`에 담겼기 때문에 클로저가 되었다.
원래대로라면 `f6`의 지역 변수는 모두 사라져야 하지만 `f6` 실행이 끝났어도 `f7`이 `a`를 기억하는 클로저가 되었기 때문에 `a`는 사라지지 않으며, `f8`을 실행할 때마다 새로운 변수인 `b` 함께 사용되어 결과를 만든다. (여기서도 만약 `f6`의 실행 결과인 `f7`을 `f8`에 담지 않았다면 `f7`은 클로저가 되지 않는다.)

혹시 위 상황에 메모리 누수가 있다고 볼 수 있을까? 그렇지 않다. 메모리가 해제 되지 않는 것과 메모리 누수는 다르다. 메모리 누수는 메모리가 해제되지 않을 때 일어나는 것은 맞지만, 위 상황을 메모리 누수라고 할 수는 없다. 위 상황은 개발자가 의도한 것이기 때문이다. `a`는 한 번 생겨날 뿐, 계속해서 생겨나거나 하지 않는다.

메모리 누수란 개발자가 '의도하지 않았는데' 메모리가 해제되지 않고 계속 남는 것을 말하며, 메모리 누수가 지속적으로 반복될 때는 치명적인 문제를 만든다. 계속해서 모르는 사이에 새어 나가야 누수라고 할 수 있다.

어쨌든 코드 1-44는 `f8`이 아무리 많이 실행되더라도 이미 할당된 `a`가 그대로 유지되기 때문에 메모리 누수는 일어나지 않는다. 이와 같은 패턴도 필요한 상황에 잘 선택하여 얼마든지 사용해도 된다.

클로저 인가? #4

```
function f9() {
  var a = 10;
  var f10 = function(c) {
    return a + b + c;
  };
  var b = 20;
  return f10;
}
var f11 = f9();
f11(30);
// 60
```

이 안에서 비동기가 일어나면 더욱 길어지기도 할 것이다. 간혹 클로저를 설명할 때 캡처라는 단어를 사용하기도 하는데 이는 오해를 일으킬만하다.

여기서 그 스코프는 함수일 수 있다. 만일 함수라면 함수가 실행될 때마다 그 스코프의 환경은 새로 만들어진다. 클로저 자신이 생성될 '때의 스코프에서 알 수 있었던'의 의미는 '클로저가 생성되고 있는 이번 실행 컨텍스트에서 알 수 있었던'이라는 의미다. '이번 실행 컨텍스트'라고 표현한 것은 그것이 계속해서 다시 발생하는 실행 컨텍스트이기 때문이고, 자신의 부모 스코프의 실행 컨텍스트도 특정 시점 안에 있는 것이기 때문에 '있었던'이라는 시점을 담은 표현으로 설명했다.

> 클로저는 자신이 생성되는 스코프의 실행 컨텍스트에서 만들어졌거나 알 수 있었던 변수 중 언젠가 자신이 실행될 때 사용할 변수들만 기억하는 함수이다. 클로저가 기억하는 변수의 값은 언제든지 남이나 자신에 의해 변경될 수 있다.

클로저 실용 사례

클로저의 강력함이나 실용성은 사실 은닉에 있지 않다.
은닉은 의미 있는 기술이자 개념이지만 은닉 자체가 달성해야 하는 과제이거나 목적은 아니다. 사실 클로저가 정말로 강력하고 실용적인 상황은 따로 있다.

1. 이전 상황을 나중에 일어날 상황과 이어 나갈 때
2. 함수로 함수를 만들거나 부분 적용을 할 때

이 중에서 '이전 상황을 나중에 일어날 상황과 이어 나갈 때'란 다음을 의미한다.

이벤트 리스너로 함수를 넘기기 이전에 알 수 있던 상황들을 변수에 담아 클로저로 만들어 기억해 두면, 이벤트가 발생되어 클로저가 실행되었을 때 기억해 두었던 변수들로 이전 상황들을 이어갈 수 있다. 콜백 패턴에서도 마찬가지로 콜백 으로 함수를 넘기기 이전 상황을 클로저로 만들어 기억해 두는 식으로 이전의 상황들을 이어 갈 수 있다.

클로저 실용 사례

```
<div class="user-list"></div>

<script>
var users = [
  { id: 1, name: "HA", age: 25 },
  { id: 2, name: "PJ", age: 28 },
  { id: 3, name: "JE", age: 27 }
];
$('.user-list').append(
  _.map(users, function(user) { // (1) 이 함수는 클로저가 아니다.
    var button = $('<button>').text(user.name); // (2)
    button.click(function() { // (3) 계속 유지되는 클로저 (내부에서 user를 사용했다.)
      if (confirm(user.name + "님을 팔로잉 하시겠습니까?")) follow(user); // (4)
    });
    return button; // (5)
  }));
function follow(user) {
  $.post('/follow', { user_id: user.id }, function() { // (6) 클로저가 되었다가 없어지는 클로저
    alert("이제 " + user.name + "님의 소식을 보실 수 있습니다.");
  });
}
```

클로저 실용 사례

1. (1)과 (5)를 보면 `users`로 `_.map`을 실행하면서 `user`마다 버튼으로 바꿔준 배열을 리턴하고 있다. 그렇게 만들어진 버튼 배열은 바로 `$('.user-list').append`에 넘겨져 화면에 그려진다.
2. (2)에서 `button`을 생성하면서 `user.name`을 새겼다. `_.map`이 보조 함수를 `user`마다 각각 실행해 주기 때문에 `user` 하나에 대한 코드만 생각하면 된다.
3. (3)에서는 클릭 이벤트를 달면서 익명 함수를 생성했고 그 함수는 클로저가 된다.
4. (3)에서 생성된 클로저는 (1)의 익명 함수의 실행 컨텍스트에서의 환경을 기억한다. 그중 기억하고 있는 외부 변수는 내부에서 사용하고 있는 `user`뿐이다. (3)에서 클로저를 만들 때의 컨텍스트는 해당 번째 `user`를 알고 있었다. 그 `user`는 외부에서 인자로 선언되었고 (3)의 내부에서 사용하기 때문에 클로저가 되어 기억하고 유지시킨다. 나중에 클릭을 통해 이 클로저가 실행되면 자신이 기억하고 있던 `user`를 이용해 (1)을 실행했을 때의 흐름을 이어간다.
5. (4)에서 `user.name`으로 `confirm`을 띄우고 기억해 둔 `user`를 `follow` 함수에게 넘긴다.
6. `follow` 함수는 `user`를 받는다. 어떤 `button`을 클릭하든지 그에 맞는 `user`가 넘어온다.
7. (6)에서는 `$.post`를 실행하면서 콜백 함수로 클로저를 만들어 넘겼다. 이 클로저는 방금 `follow`가 실행되었을 때의 환경을 기억한다. 서버가 응답을 주어 콜백 함수가 실행되고 나면 기억하고 있던 `user`를 통해 흐름을 이어 간다.

흔한 클로저 실수 사례를 잘 알고 있다면 `for + click`을 해결하기 위한 별도의 코드가 없다는 것을 눈치챘을 것이다. `for`문을 돌면서 `click` 이벤트에 리스너를 등록할 경우 `i++`(상태 변화) 때문에 지역 변수의 값이 먼저 변해서 에러가 나는데, 해당 예제에서는 별도의 일을 하지 않고도 잘 동작하고 있다. 이미 `i`값이 변할 때마다 `_.map` 함수가 `iteratee`를 실행하여 항상 새로운 실행 컨텍스트를 만들어 주기 때문이다.

`_.map`과 같은 함수는 동시성이 생길 만한 상황이더라도, 상태 변화로 인한 부수 효과로부터 자유롭고 편하게 프로그래밍할 수 있도록 해 준다. 함수형 프로그래밍은 서로 다른 실행 컨텍스트에 영향을 줄 수 있을 만한 상태 공유나 상태 변화를 만들지 않는 것이 목표에 가깝고, 이런 함수형 프로그래밍의 특성은 '흔한 클로저 실수'와 같은 문제들을 애초에 차단한다.

클로저 예제

```
// 1. 흔한 클로저 실수 - 어떤 버튼을 클릭해도 JE
var buttons = [];
for (var i = 0; i < users.length; i++) {
  var user = users[i];
  buttons.push(
    $("<button>")
      .text(user.name)
      .click(function() {
        console.log(user.name);
      })
  );
}
$(".user-list").append(buttons);

// 2. 절차지향적 해결 - 어차피 함수의 도움을 받아야 함, 각각 다른 이름이 잘 나옴
var buttons = [];
for (var i = 0; i < users.length; i++) {
  (function(user) {
    buttons.push(
      $("<button>")
        .text(user.name)
        .click(function() {
          console.log(user.name);
        })
    );
  })(users[i]);
}
$(".user-list").append(buttons);
```

```
// 3. 함수적 해결 - 깔끔한 코드는 덤
$(".user-list").append(
  _.map(users, function(user) {
    return $("<button>")
      .text(user.name)
      .click(function() {
        console.log(user.name);
      });
  })
);
```

클로저를 많이 사용하라!

고차함수

고차함수 (Higher-Order Function)

1. 함수를 인자로 받아 대신 실행하는 함수
2. 함수를 리턴하는 함수
3. 함수를 인자로 받아서 또 다른 함수를 리턴하는 함수

'고차 함수를 적극적으로 활용하는 프로그래밍'

함수를 인자로 받아 대신 실행하는 함수

```
function callWith10(val, func) {
  return func(10, val);
}

function add(a, b) {
  return a + b;
}

function sub(a, b) {
  return a - b;
}

callWith10(20, add);
// 30
callWith10(5, sub);
// 5
```

`callWith10`은 고차 함수다. 함수를 받아 내부에서 대신 실행하기 때문이다. `func`라는 이름의 인자로 `add`나 `sub` 함수를 받아, 역시 인자로 받았던 `val`과 함께 `10`을 인자로 넘기면서 대신 실행한다.

함수를 리턴하는 함수

```
function constant(val) {
  return function() {
    return val;
  }
}
var always10 = constant(10);
always10();
// 10
always10();
// 10
always10();
// 10
```

`constant` 함수는 실행 당시 받았던 `10`이라는 값을 받아 내부에서 익명 함수를 클로저로 만들어 `val`을 기억하게 만든 후 리턴한다. 리턴된 함수에는 `always10`이라는 이름을 지어주었다. `always10`을 실행하면 항상 `10`을 리턴한다. `constant`처럼 함수를 리턴하는 함수도 고차 함수다.

함수를 대신 실행하는 함수를 리턴하는 함수

```
function callWith(val1) {
  return function(val2, func) {
    return func(val1, val2);
  }
}
var callWith10 = callWith(10);
callWith10(20, add);
// 30
var callWith5 = callWith(5);
callWith5(5, sub);
// 0
```

`callWith`는 함수를 리턴하는 함수다. `val1`을 받아서 `val1`을 기억하는 함수를 리턴했다. 리턴된 그 함수는 이후에 `val2`와 `func`를 받아 대신 `func`를 실행해준다. `callWith`에 10을 넣어 앞서 만들었던 `callWith10`과 동일하게 동작하는 함수를 만들었다. 이제는 `callWith`를 이용해 `callWith5`든 `callWith30`든 만들 수 있다. 함수를 리턴하는 함수를 사용할 경우 다음처럼 변수에 담지 않고 바로 실행해도 된다.

```
callWith(30)(20, add);
// 50
callWith(20)(20, sub);
// 0
```

```
callWith([1, 2, 3])(function(v) { return v * 10; }, _.map);
// [10, 20, 30]
_.get = function(list, idx) {
  return list[idx];
};
var callWithUsers = callWith([
  { id: 2, name: "HA", age: 25 },
  { id: 4, name: "PJ", age: 28 },
  { id: 5, name: "JE", age: 27 }
]);
callWithUsers(2, _.get);
// { id: 5, name: "JE", age: 27 }
callWithUsers(function(user) {
  return user.age > 25;
}, _.find);
// { id: 4, name: "PJ", age: 28 }
callWithUsers(function(user) {
  return user.age > 25;
}, _.filter);
// [{ id: 4, name: "PJ", age: 28 },
//  { id: 5, name: "JE", age: 27 }];
callWithUsers(function(user) {
  return user.age > 25;
}, _.some);
// true
callWithUsers(function(user) {
  return user.age > 25;
}, _.every);
// false
```

콜백 함수라 잘못 불리는 보조 함수

'콜백 함수를 받아 자신이 해야 할 일을 모두 끝낸 후 결과를 되돌려 주는 함수도 고차 함수다.'

보통은 비동기가 일어나는 상황에서 사용되며 콜백 함수를 통해 다시 원래 위치로 돌아오기 위해 사용되는 패턴이다.
콜백 패턴은 클로저 등과 함께 사용할 수 있는 매우 강력한 표현이자 비동기 프로그래밍에 있어 없어서는 안 될 매우 중요한 패턴이다. 콜백 패턴은 끝이 나면 컨텍스트를 다시 돌려주는 단순한 협업 로직을 가진다.

함수를 리턴하는 함수와 부분 적용

`addMaker, bvalue, bmatch, callWith`와 같은 함수들은 약속된 개수의 인자를 미리 받아 둔다.
클로저로 만들어진 함수가 추가적으로 인자를 받아 로직을 완성해 나가는 패턴을 갖는다.
이와 유사한 기법들로는 `bind, curry, partial` 등이 있다.
이런 기법들을 통틀어 칭하는 특별한 용어는 없지만 다음과 같은 공통점을 갖는다.

'기억하는 인자 혹은 변수가 있는 클로저'를 리턴한다.

함수형 자바스크립트에서 함수를 리턴하는 함수의 실용성은 꽤 높다.

bind

```
function add(a, b) {
  return a + b;
}
var add10 = add.bind(null, 10);
add10(20);
// 30
```

`bind`는 첫 번째 인자로 `bind`가 리턴할 함수에서 사용될 `this`를 받는다.
두 번째 인자부터 함수에 미리 적용될 인자들이다.
인자를 미리 적용해 두기 위해 `this`로 사용될 첫 번째 인자에 `null`을 넣은 후 `10`을 넣었다.
`add10`과 같이 `this`를 사용하지 않는 함수이면서 왼쪽에서부터 순서대로만 인자를 적용하면 되는 상황에서는 원하는 결과를 얻을 수 있다. `bind`의 아쉬운 점은 두 가지다. 인자를 왼쪽에서부터 순서대로만 적용할 수 있다는 점과 `bind`를 한 번 실행한 함수의 `this`는 무엇을 적용해두었든 앞으로 바꿀 수 없다는 점이다.

https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

bind

많은 자바스크립트 개발자들이 `bind`에서 `this`가 제외된 버전의 `curry`를 만들어 좀 더 간결한 코드를 제안했다. 잘 구현된 사례로 Lodash의 `_.curry`가 있다. `_.curry`는 함수가 필요로 하는 인자의 개수가 모두 채워질 때까지는 실행이 되지 않다가 인자의 수가 모두 채워지는 시점에 실행된다.
`_.curry`는 `bind`와 달리 `this`를 제외하고 인자만 적용해 둘 수 있어 좀 더 간결하게 코딩할 수 있고, 이후에 `this`를 적용할 수 있다는 점에서 `bind`보다 낫다.

함수형 자바스크립트의 저자인 마이클 포거스도 '자바스크립트에서는 커링보다는 부분 적용이 더 실용적'이라고 말한다.

`bind`는 왼쪽에서부터 원하는 만큼의 인자를 지정해 둘 수 있지만 원하는 지점을 비워 두고 적용할 수는 없다. 예를 들어 어떤 함수가 필요로 하는 인자가 3개가 있는데 그중 두 번째 인자만을 적용해 두고 싶다면 `bind`로는 이것을 할 수 없다.

존 레식의 partial

```
Function.prototype.partial = function() {
  var fn = this, args = Array.prototype.slice.call(arguments); // (1)
  return function() { // (2)
    var arg = 0;
    for (var i = 0; i < args.length && arg < arguments.length; i++) // (5)
      if (args[i] === undefined) args[i] = arguments[arg++]; // (6)
    return fn.apply(this, args);
  };
};

function abc(a, b, c) {
  console.log(a, b, c);
}

var ac = abc.partial(undefined, 'b', undefined); // (3)
ac('a', 'c'); // (4)
// a b c
```

존 레식의 partial

(1)우선 `partial`이 실행되면 `fn`에 자기 자신인 `this`를 담는다. 여기서 자기 자신은 `abc` 같은 함수다. `args`에는 `partial`이 실행될 때 넘어온 인자들을 배열로 변경하여 `args`에 담아둔다. (2)`fn`과 `args`는 리턴된 익명 함수가 기억하게 되므로 지워지지 않는다. (3)`abc.partial`을 실행할 때 첫 번째 인자와 세 번째 인자로 넘긴 `undefined` 자리는 나중에 `ac`가 실행될 때 채워질 것이다. (4)`ac`를 실행하면서 넘긴 ``'a'``와 ``'c'``는 (5)리턴된 익명 함수의 `arguments`에 담겨 있다. (6)`for`를 돌면서 미리 받아두었던 `args`에 `undefined`가 들어 있던 자리를 `arguments`에서 앞에서부터 꺼내면서 모두 채운다. 다 채우고 나면 미리 받아두었던 `fn`을 `apply`로 실행하면서 인자들을 배열로 넘긴다.

`partial`은 구현이 잘 된 것은 아니다. 함수의 인자로 `undefined`를 사용하고 싶을 수도 있는데 `undefined`가 인자를 비워 두기 위한 구분자로 사용되고 있기 때문에, `undefined`를 미리 적용하고 싶다면 방법이 없다. 또한 초기에 `partial`을 실행할 때 나중에 실제로 실행될 함수에서 사용할 인자의 개수만큼 꼭 미리 채워 놓아야만 한다. 만일 개수를 채워 놓지 않으면 아래와 같이 동작한다.

```
var ac2 = abc.partial(undefined, 'b');
ac2('a', 'c');
// a c undefined
```

존 레식의 partial

`partial`이 가진 제약은 '인자 개수 동적으로 사용하기'나 '`arguments` 객체 활용'과 같은 자바스크립트의 유연함을 반영하지 못한다는 점에서 특히 아쉽다.

이렇다면 커링에 비해 특별히 좋을 것이 없다.
자바스크립트에서는 함수 상단부에서 정의해 둔 인자 개수보다 적게 인자를 넘기거나 `arguments` 객체를 이용해 더 많은 인자를 넘기는 기법을 많이 사용하기 때문이다.
특히 함수형 자바스크립트에서는 이런 기법이 더욱 많이 사용된다.
만일 `add`라는 함수가 다음과 같이 구현되어 있었다면 `partial`과는 합이 더욱 맞지 않는다.

존 레식의 partial 문제점

```
function add() {
  var result = 0;
  for (var i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }
  return result;
}

add(1, 2, 3, 4, 5);
// 15
var add2 = add.partial(undefined, 2);
add2(1, 3, 4, 5);
// 3
var add3 = add.partial(undefined, undefined, 3, undefined, undefined);
add3(1, 2, 4, 5);
// 15
add3(50, 50, 50, 50);
// 15 <--- 버그
add3(100, 100, 100, 100);
// 15 <--- 버그
```

존 레식의 partial 문제점

위 상황에서 `add2`는 `3, 4, 5` 인자를 무시하게 된다. `add3`처럼 하면 `1, 2, 4, 5`를 모두 사용할 수 있게 되지만 `undefined`로라도 인자 개수를 채워놔야 해서 코드가 깔끔하지 못하고, `partial` 이후에는 역시 4개 이상의 인자를 사용할 수 없다는 단점이 생긴다. 인자를 적게 넣을 수도 없다.

그런데 이런 것을 떠나 위 코드에는 훨씬 치명적인 문제가 있다. 존 레식이 의도한 것인지는 모르겠지만 그가 만든 `partial` 함수로 만든 함수는 재사용이 사실상 불가능하다. 한 번 `partial`을 통해 만들어진 함수를 실행하고 나면 클로저로 생성된 `args`의 상태를 직접 변경하기 때문에, 다음번에 다시 실행해도 같은 `args`를 바라보고 이전에 적용된 인자가 남는다. 결과적으로 `partial`로 만들어진 함수는 단 한 번만 정상적으로 동작한다. 아마도 실수일 것이다. 다음 (1), (2)와 같이 두 줄 만 변경하면 두 번 이상 실행해도 정상적으로 동작한다.

실수 고치기

```
Function.prototype.partial = function() {
  var fn = this, _args = arguments; // (1) 클로저가 기억할 변수에는 원본을 남기기
  return function() {
    var args = Array.prototype.slice.call(_args); // (2) 리턴된 함수가 실행될 때마다 복사하여 원본
지키기
    var arg = 0;
    for (var i = 0; i < args.length && arg < arguments.length; i++)
      if (args[i] === undefined) args[i] = arguments[arg++]; // 실행때마다 새로 들어온 인자 채우기
    return fn.apply(this, args);
  };
};
var add3 = add.partial(undefined, undefined, 3, undefined, undefined);
add3(1, 2, 4, 5);
// 15
add3(50, 50, 50, 50);
// 203
add3(10, 20, 30, 40);
// 103
```

실수 고치기

자바스크립트 개발자는 이 같은 실수를 많이 할 수 있다.
클로저가 기억하는 변수도 변수이며 값은 변할 수 있다. 이처럼 상태를 변경하는 코드는 위험하다.
더 함수적인 프로그래밍을 하면 위와 같은 실수를 최소화할 수 있다.

이제 좀 더 나은 버전의 `partial` 함수를 확인해 보자. Underscore.js의 `_.partial`은 앞서 소개된 `partial`의 아쉬운 점들이 해결된 부분 적용 함수다. Underscore.js나 Lodash의 `_.partial`은 내부의 많은 다른 함수들과 본체를 공유하고 있어 꽤나 복잡하기 때문에 여기서 내부를 함께 소개하기는 좀 어렵다. 여기서는 앞서 보여준 `partial`보다 발전된 사용법과 동작만 확인하자.

Underscore.js 의 _.partial

https://github.com/ABoutCoDing/Functional-Javascript/blob/master/1.4%20%ED%95%A8%EC%88%98%ED%98%95%20%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8%EB%A5%BC%20%EC%9C%84%ED%95%9C%20%EA%B8%B0%EC%B4%88.js#L341

Underscore.js 의 _.partial

`_.partial`은 적용해 둘 인자와 비워둘 인자를 구분하는 구분자로 `undefined` 대신 `_`를 사용한다. `_`는 자바스크립트에서 사용하는 일반 값이 아니므로 구분자로 사용하기 더 적합하며 표현력도 좋다. 다른 언어의 Partial application에서도 `_`를 사용하니 더욱 어울린다. 한 가지 더 좋아진 점은 모든 인자 자리를 미리 확보해 두지 않아도 된다는 점이다. 실행 타이밍에서 인자를 많이 사용하든 적게 사용하든 모두 잘 동작한다. 인자 개수가 동적인 자바스크립트와 잘 어울린다. 또한 `bind`와 달리 `this`를 적용해 두지 않았으므로 메서드로도 사용이 가능하다.

함수형 자바스크립트를 위한 문법 다시보기

함수형 자바스크립트를 위한 문법 다시보기

객체와 대괄호 다시보기

난해해 보이는 문법들을 확인하는 목적

더 짧은 코드를 위해
추상화의 다양한 기법
if를 없애기 위해
특별한 로직을 위해
게시를 위해
은닉을 위해
함수를 선언하고 참조하기 위해
컨텍스트를 이어주기 위해

아무 곳에서나 함수 열기, 함수 실행을 원하는
시점으로 미뤄서 실행하기

다양한 key/value 정의 방법

```
var obj = { a: 1, b: 2 }; // (1)
obj.c = 3;
obj["d"] = 4; // (2)
var e = "e";
obj[e] = 5;
function f() {
  return "f";
}
obj[f()] = 6;
console.log(obj);
// { a: 1, b: 2, c: 3, d: 4, e: 5, f: 6 }
```

띄어쓰기, 특수문자, 숫자

```
// 띄어쓰기를 써도 key로 만들 수 있다.
var obj2 = { " a a a ": 1 };
obj2[" b b b "] = 2;
console.log(obj2);
// { " a a a ": 1, " b b b ": 2 }
// 특수문자를 써도 key로 만들 수 있다.
var obj3 = { "margin-top": 5 };
obj3["padding-bottom"] = 20;
console.log(obj3);
// { margin-top: 5, padding-bottom: 20 }
// 숫자도 key로 쓸 수 있다.
var obj4 = { 1: 10 };
obj4[2] = 20;
console.log(obj4);
// { 1: 10, 2: 20 }
```

객체와 key

코드가 실행되지 않는 key 영역

```
var obj5 = { (true ? "a" : "b"): 1 };
// Uncaught SyntaxError: Unexpected token (
```

{} 안쪽의 key 영역에서는 코드를 실행할 수 없다.

코드가 실행되는 key 영역

```
var obj6 = {};
obj6[true ? "a" : "b"] = 1;
console.log(obj6);
// { a: 1 }
```

[] 사이에는 문자열이 담긴 변수도 넣을 수 있고 연산자도 사용할 수 있어서 함수도 실행할 수 있다.
[]에서는 코드를 실행할 수 있다.

ES6에서 동작하는 {} 안쪽에서 대괄호 사용하기

```
var obj5 = { [true ? "a" : "b"]: 1 };
// { a: 1 }
```

함수나 배열에 달기

함수를 객체로 사용

```
function obj8() {}
obj8.a = 1;
obj8.b = 2;
console.log(obj8.a);
// 1
console.log(obj8.b);
// 2
```

호이스팅

```
obj9.a = 1;
obj9.b = 2;
console.log(obj9.a + obj9.b);
// 3
function obj9() {}
```

배열에 숫자가 아닌 key 사용하기

```
var obj10 = [];
obj10.a = 1;
console.log(obj10);
// [a: 1]
console.log(obj10.length);
// 0
```

배열에 숫자로 key 사용하기

```
var obj11 = [];
obj11[0] = 1;
obj11[1] = 2;
console.log(obj11);
// [1, 2]
console.log(obj11.length);
// 2
```

배열에 숫자로 key를 직접 할당해도 push와 동일하게 동작한다. 자동으로 length도 올라간다.

기본 객체의 메서드 지우기

```
var obj = { a: 1, b: 2, c: 3 };
delete obj.a;
delete obj["b"];
delete obj["C".toLowerCase()];
console.log(obj);
// {};
delete Array.prototype.push;
var arr1 = [1, 2, 3];
arr1.push(4);
// Uncaught TypeError: arr1.push is not a function
```

함수정의 다시보기

```
function add1(a, b) {
  return a + b;
}
var add2 = function(a, b) {
  return a + b;
};
var m = {
  add3: function(a, b) {
    return a + b;
  }
};
```

일반적인 함수 정의

```
function add1(a, b) {
  return a + b;
}
var add2 = function(a, b) {
  return a + b;
};
var m = {
  add3: function(a, b) {
    return a + b;
  }
};
```

호이스팅

에러가 나는 상황이지만
호이스팅이다.

```
add1(10, 5);
// 15;
add2(10, 5);
// Uncaught TypeError: add2 is not a function(…)(anonymous function)
function add1(a, b) {
  return a + b;
}
var add2 = function(a, b) {
  return a + b;
};
```

선언한적 없는 함수 실행

```
// 2-15 선언한적 없는 함수 실행
hi();
// Uncaught ReferenceError: hi is not defined

// 2-16 선언한적 없는 변수 참조하기
var a = hi;
// Uncaught ReferenceError: hi is not defined
```

실행하지 않고 참조만 해보기

```
console.log(add1);
// function add1(a, b) { return a + b; }
console.log(add2); // 에러가 나지 않는다.
// undefined
function add1(a, b) {
  return a + b;
}
var add2 = function(a, b) {
  return a + b;
};
```

호이스팅을 이용하여 리턴문 아래에 함수 선언하기

```
function add(a, b) {
  return valid() ? a + b : new Error();
  function valid() {
    return Number.isInteger(a) && Number.isInteger(b);
  }
}
console.log(add(10, 5));
// 15;
// console.log(add(10, "a"));
// Error(...)
```

호이스팅을 이용해 코드의 순서를 이해하기 편하게 배치

```
// (1) end가 먼저 정의되어 코드가 다소 복잡하게 읽힌다.
app.post('/login', function(req, res) {
  db.select("users", { where: { email: req.body.email } }, function(err, user) {
    function end(user) {
      req.session.user = user;
      res.redirect('/');
    }
    if (user && user.password === req.body.password) return end(user);
    db.insert("users", {
      email: req.body.email,
      password: req.body.password
    }, function(err, user) {
      end(user);
    });
  });
});
```

호이스팅을 이용해 코드의 순서를 이해하기 편하게 배치

```
// (2) 호이스팅 덕분에 end를 나중에 정의해도 잘 동작한다. 읽기 더 편하다.
app.post('/login', function(req, res) { // (3)
  db.select("users", { where: { email: req.body.email } }, function(err, user) {
    if (user && user.password === req.body.password) return end(user);
    db.insert("users", {
      email: req.body.email,
      password: req.body.password
    }, function(err, user) {
      end(user);
    });
    function end(user) {
      req.session.user = user;
      res.redirect('/');
    }
  });
});
```

일반적인 즉시 실행 방식

```
(function(a) {
  console.log(a);
  // 100
})(100);
```

괄호 없이 정의가 가능한(즉시 실행도 가능한) 다양한 상황

```
!(function(a) {
  console.log(a);
  // 1
})(1);
true &&
  (function(a) {
    console.log(a);
    // 1
  })(1);
1
  ? (function(a) {
      console.log(a);
      // 1
    })(1)
  : 5;
0,
  (function(a) {
    console.log(a);
    // 1
  })(1);
```

```
var b = (function(a) {
  console.log(a);
  // 1
})(1);
function f2() {}
f2(
  (function(a) {
    console.log(a);
    // 1
  })(1)
);
var f3 = (function c(a) {
  console.log(a);
  // 1
})(1);
new (function() {
  console.log(1);
  // 1
})();
```

객체 생성 방법

```
var pj = new (function() {
  this.name = "PJ";
  this.age = 28;
  this.constructor.prototype.hi = function() {
    console.log("hi");
  };
})();
console.log(pj);
// { name: "PJ", age: 28 }
pj.hi();
// hi
```

즉시 실행하며 this 할당하기

```
var a = function(a) {
  console.log(this, a);
  // [1], 1
}.call([1], 1);
```

함수의 메서드인 call을 바로 .으로 접근할 수 있으며 익명함수를
즉시 실행하면서 this를 할당 할 수도 있다.

new Function 이나 eval을 써도 될까요?

서버에서 클라이언트가 보낸 값을 이용해 new Function 이나 eval을 사용하는 것이 아니라면 보안 문제라는 것은 있을 수 없다.
서버에서도 마찬가지로 서버에서 생성한 값만으로 new Function이나 eval을 한다면 보안문제가 생기지 않는다.

new Function 사용법

```
var a = eval("10 + 5");
console.log(a);
// 15
var add = new Function("a, b", "return a + b;");
add(10, 5);
// 15
```

간단 버전 문자열 화살표 함수

```
function L(str) {
  var splitted = str.split("=>");
  return new Function(splitted[0], "return (" + splitted[1] + ");");
}

L("n => n * 10")(10);
// 100
L("n => n * 10")(20);
// 200
L("n => n * 10")(30);
// 300
L("a, b => a + b")(10, 20);
// 30
L("a, b => a + b")(10, 5);
// 15
```

10,000번 선언해보기

```
console.time("익명 함수");
for (var i = 0; i < 10000; i++) {
  (function(v) {
    return v;
  })(i);
}
console.timeEnd("익명 함수");
// 익명 함수: 0.9ms ~ 1.7ms
console.time("new Function");
for (var i = 0; i < 10000; i++) {
  L("v => v")(i); // new Function
}
console.timeEnd("new Function");
// new Function: 337ms ~ 420ms
```

익명 함수와 문자열 화살표 함수

```
console.time("1");
var arr = Array(10000);
_.map(arr, function(v, i) {
  return i * 2;
});
console.timeEnd("1");
// 1: 0.5ms ~ 0.7ms
console.time("2");
var arr = Array(10000);
_.map(arr, L("v, i => i * 2")); // new Function
console.timeEnd("2");
// 2: 0.5ms ~ 0.8ms
```

eval로 한 번 더 감싼 경우

```
console.time("3");
var arr = Array(10000);
_.map(arr, eval("L('v, i => i * 2')")); // eval + new Function
console.timeEnd("3");
// 3: 0.6ms ~ 0.9ms
```

1,000배의 성능 차이

```
// (1)
console.time("4");
var arr = Array(10000);
_.map(arr, function(v, i) {
  return (function(v, i) {
    // 안에서 익명 함수를 한번 더 만들어 즉시 실행
    return i * 2;
  })(v, i);
});
console.timeEnd("4");
// 4: 0.8ms ~ 1.8ms
console.time("5");
var arr = Array(10000);
_.map(arr, function(v, i) {
  return L("v, i => i * 2")(v, i); // 안에서 문자열 화살표 함수로 함수를 만들어 즉시 실행
});
console.timeEnd("5");
// 5: 362ms ~ 480ms
```

메모이제이션 기법

```
// 원래의 L
function L(str) {
  var splitted = str.split('=>');
  return new Function(splitted[0], 'return (' + splitted[1] + ');');
}

// 메모이제이션 기법
function L2(str) {
  if (L2[str]) return L2[str]; // (1) 혹시 이미 같은 `str`로 만든 함수가 있다면 즉시 리턴
  var splitted = str.split('=>');
  return L2[str] = new Function(splitted[0], 'return (' + splitted[1] + ');');
  // 함수를 만든 후 L2[str]에 캐시하면서 리턴
}
```

코드 구조는 그대로이지만 성능이 좋아짐

```
console.time('6');
var arr = Array(10000);
_.map(arr, function(v, i) {
  return L2('v, i => i * 2')(v, i);
});
console.timeEnd('6');
// 6: 0.5ms ~ 1.2ms
```

Partial.js의 문자열 화살표 함수

```
try { var has_lambda = true; eval('a=>a'); } catch (err) { var has_lambda = false; }
_.l = _.lambda = function f(str) {
  if (typeof str !== 'string') return str;
  if (f[str]) return f[str]; // (1)
  if (!str.match(/=>/)) return f[str] = new Function('$', 'return (' + str + ')'); // (2)
  if (has_lambda) return f[str] = eval(str); // (3) ES6
  var ex_par = str.split(/\s*=>\s*/);
  return f[str] = new Function( // (4)

ex_par[0].replace(/(?:\b[A-Z]|\.[a-zA-Z_$])[a-zA-Z_$\d]*|[a-zA-Z_$][a-zA-Z_$\d]*\s*:|this|arguments|'(?:[^'\\]|\
\.)*'|"(?:[^"\\]|\\.)*"/g, '').match(/([a-z_$][a-z_$\d]*)/gi) || [],
    'return (' + ex_par[1] + ')');
};
console.log( _.l('(a, b) => a + b')(10, 10) );
// 20
console.log( _.l('a => a * 5')(10) );
// 50
console.log( _.l('$ => $ * 10')(10) );
// 100
// 사용하는 인자가 하나일 때 인자 선언부를 생략한 문자열 화살표 함수
console.log( _.l('$ * 10')(10) );
// 100
console.log( _.l('++$')(1) );
// 2
```

유명(named) 함수

```
// 유명 함수 표현식

var f1 = function f() {
  console.log(f);
};
```

```
// 익명 함수에서 함수가 자신을 참조하는 법

var f1 = function() {
  console.log(f1);
};
f1();
// function() {
//   console.log(f1);
// }
// 위험 상황
var f2 = f1;
f1 = 'hi~~';
f2();
// hi~~;
```

익명 함수에서 함수가 자신을 참조하는 법2

```
var f1 = function() {
  console.log(arguments.callee);
};
f1();
// function() {
//   console.log(arguments.callee);
// }
var f2 = f1;
f1 = null;
f2();
// function() {
//   console.log(arguments.callee);
// }
```

유명 함수의 자기 참조

```
var f1 = function f() {
  console.log(f);
};
f1();
// function f() {
//   console.log(f);
// }
var f2 = f1;
f1 = null;
f2();
// function f() {
//   console.log(f);
// }
```

아주 안전하고 편한 자기 참조

```
var hi = 1;
var hello = function hi() {
  console.log(hi);
};
hello();
// function hi() {
//   console.log(hi);
// }
console.log(hi);
// 1
console.log(++hi);
// 2
hello();
// function hi() {
//   console.log(hi);
// }
console.log(hello.name == 'hi');
// true
```

```
var z1 = function z() {
  console.log(z, 1);
};
var z2 = function z() {
  console.log(z, 2);
};
z1();
// function z() {
//   console.log(z, 1);
// }
z2();
// function z() {
//   console.log(z, 2);
// }
console.log(z1.name == z2.name);
// true
z;
// Uncaught ReferenceError: z is not defined
```

재귀를 이용한 flatten

```
function flatten(arr) {
  return function f(arr, new_arr) { // (1)
    arr.forEach(function(v) {
      Array.isArray(v) ? f(v, new_arr) : new_arr.push(v); // (3)
    });
    return new_arr;
  }(arr, []); // (2)
}
flatten([1, [2], [3, 4]]);
// [1, 2, 3, 4]
flatten([1, [2], [[3], 4]]);
// [1, 2, 3, 4]
flatten([1, [[2], [[3], [[4], 5]]]]);
// [1, 2, 3, 4, 5]
```

즉시 실행 + 유명 함수 기법이 아닌 경우

```
function flatten2(arr, new_arr) {
  arr.forEach(function(v) {
    Array.isArray(v) ? flatten2(v, new_arr) : new_arr.push(v); // (3)
  });
  return new_arr;
}
flatten2([1, [2], [3, 4]], []); // 항상 빈 Array를 추가로 넘겨야하는 복잡도 증가
function flatten3(arr, new_arr) {
  if (!new_arr) return flatten3(arr, []); // if 문이 생김
  arr.forEach(function(v) {
    Array.isArray(v) ? flatten3(v, new_arr) : new_arr.push(v); // (3)
  });
  return new_arr;
}
flatten3([1, [2], [3, 4]]);
```

인자, this, arguments 출력

```
function test(a, b, c) {
  console.log("a b c: ", a, b, c);
  console.log('this:', this);
  console.log('arguments:', arguments);
}

// 2-45 실행하면서 넘긴 인자와 출력 된 정보들
test(10); // (1)
// a b c: 10 undefined undefined
// this: Window {...}
// arguments: [10]
test(10, undefined); // (2)
// a b c: 10 undefined undefined
// this: Window {...}
// arguments: [10, undefined]
test(10, 20, 30); // (3)
// a b c: 10 20 30
// this: Window {...}
// arguments: [10, 20, 30]
```

이게 맞아?

```
function test2(a, b) {
  b = 10;
  console.log(arguments);
}
test2(1); // (1)
// [1]
test2(1, 2); // (2)
// [1, 10]
```

객체의 값과 변수의 값

```
var obj1 = {
  0: 1,
  1: 2
};
console.log(obj1);
// { 0: 1, 1: 2 }
var a = obj1[0];
var b = obj1[1];
b = 10;
console.log(obj1);
// { 0: 1, 1: 2 } <------- 바뀌지 않음
console.log(obj1[1]);
// 2
console.log(b);
// 10             <------- b 만 바뀜
```

반대로 해보기

```
function test3(a, b) {
  arguments[1] = 10;
  console.log(b);
}
test3(1, 2);
// 10
```

```
// 2-49
test(10); // (1)
// a b c: 10 undefined undefined
// this: Window {...}
// arguments: [10]
test(10, undefined); // (2)
// a b c: 10 undefined undefined
// this: Window {...}
// arguments: [10, undefined]
test(10, 20, 30); // (3)
// a b c: 10 20 30
// this: Window {...}
// arguments: [10, 20, 30]
```

메서드로 만들기

```
var o1 = { name: "obj1" };
o1.test = test;          // test 함수를 o1의 메서드로 할당
o1.test(3, 2, 1);
// a b c: 3 2 1
// this: Object {name: "obj1"}
// arguments: [3, 2, 1]
var a1 = [1, 2, 3];
a1.test = test;          // test 함수를 a1의 메서드로 할당
a1.test(3, 3, 3);
// a b c: 3 3 3
// this: Array [1, 2, 3]
// arguments: [3, 3, 3]
```

```
// 2-51
var o1_test = o1.test;
o1_test(5, 6, 7);
// a b c: 5 6 7
// this: Window {...}
// arguments: [5, 6, 7]
```

```
// 2-52
(a1.test)(8, 9, 10);
// a b c: 8 9 10
// this: Array [1, 2, 3]
// arguments: [8, 9, 10]
a1['test'](8, 9, 10);
// a b c: 8 9 10
// this: Array [1, 2, 3]
// arguments: [8, 9, 10]
```

```
// 2-53
console.log(test == o1.test && o1.test == a1.test);
// true
```

```
// 2-54
test.call(undefined, 1, 2, 3);
test.call(null, 1, 2, 3);
test.call(void 0, 1, 2, 3);
// a b c: 1 2 3
// this: Window {...}
// arguments: [1, 2, 3]
```

```
// 2-55
test.call(o1, 3, 2, 1);
// a b c: 3 2 1
// this: Object {name: "obj1"}
// arguments: [3, 2, 1]
test.call(1000, 3, 2, 1);
// a b c: 3 2 1
// this: Number 1000
// arguments: [3, 2, 1]
```

```
// 2-56
o1.test.call(undefined, 3, 2, 1);
// a b c: 3 2 1
// this: Window {...}
// arguments: [3, 2, 1]
o1.test.call([50], 3, 2, 1);
// a b c: 3 2 1
// this: Array [50]
// arguments: [3, 2, 1]
```

```
// 2-57
test.apply(o1, [3, 2, 1]);
// a b c: 3 2 1
// this: Object {name: "obj1"}
// arguments: [3, 2, 1]
test.apply(1000, [3, 2, 1]);
// a b c: 3 2 1
// this: Number 1000
// arguments: [3, 2, 1]
o1.test.apply(undefined, [3, 2, 1]);
// a b c: 3 2 1
// this: Window {...}
// arguments: [3, 2, 1]
o1.test.apply([50], [3, 2, 1]);
// a b c: 3 2 1
// this: Array [50]
// arguments: [3, 2, 1]
```

```
// 2-58
test.apply(o1, { 0: 3, 1: 2, 2: 1, length: 3 }); // Array가 아님
// a b c: 3 2 1
// this: Object {name: "obj1"}
// arguments: [3, 2, 1]
(function() {
  test.apply(1000, arguments);  // arguments 객체 역시 Array가 아님
})(3, 2, 1);
// a b c: 3 2 1
// this: Number 1000
// arguments: [3, 2, 1]
```

```
// 2-59
(function() {
  arguments.length--;
  test.apply(1000, arguments);
})(3, 2, 1);
// a b c: 3 2 undefined
// this: Number 1000
// arguments: [3, 2]
test.apply(1000, [1].concat([2, 3]));
// a b c: 1 2 3
// this: Number 1000
// arguments: [1, 2, 3]
```

네이티브 코드 활용하기

```
var slice = Array.prototype.slice;
function toArray(data) {
  return slice.call(data);
}
function rest(data, n) {
  return slice.call(data, n || 1);
}
var arr1 = toArray({ 0: 1, 1: 2, length: 2 });
// [1, 2]
arr1.push(3);
console.log(arr1);
// [1, 2, 3];
rest([1, 2, 3]);
// [2, 3];
rest([1, 2, 3], 2);
// [3]
```

```
// 2-61
if (var a = 0) console.log(a);
// Uncaught SyntaxError: Unexpected token var

// 2-62
if (function f1() {}) console.log('hi');
// hi
// f1();
// Uncaught ReferenceError: f1 is not defined
```

이미 선언되어 있는 변수의 값 재할당

```
var a;
if (a = 5) console.log(a); // (1)
// 5
if (a = 0) console.log(1); // (2)
else console.log(a);
// 0
if (!(a = false)) console.log(a); // (3)
// false
if (a = 5 - 5); // (4)
else console.log(a);
// 0
```

```
// 2-64
var obj = {};
if (obj.a = 5) console.log(obj.a);
// 5
if (obj.b = false) console.log(obj.b); // (2)
else console.log('hi');
// hi
var c;
if (c = obj.c = true) console.log(c); // (3)
// true
```

```
// 2-65
function add(a, b) {
  return a + b;
}
if (add(1, 2)) console.log('hi1');
var a;
if (a = add(1, 2)) console.log(a);
// 3
if (function() { return true; }()) console.log('hi');
// hi
```

```
// 2-66
var a = true;
var b = false;
var v1 = a || b;
console.log(v1);
// true
var v2 = b || a;
console.log(v2);
// true
var v3 = a && b;
console.log(v3);
// false
var v4 = b && a;
console.log(v4);
// false
```

```
// 2-67
var a = "hi";
var b = "";
var v1 = a || b; // (1) `a`가 긍정적인 값이면 `||` 이후를 확인하지 않아 `a` 값이 `v1`에
담긴다.
console.log(v1);
// "hi"
var v2 = b || a; // (2) `b`가 부정적이여서 `a`를 확인 했고 `a`의 값이 담겼다.
console.log(v2);
// "hi"
var v3 = a && b; // (3) `a`가 긍적적인 값이어서 `&&` 이후를 확인하게 되고 `b` 값이 담긴다.
console.log(v3);
// ""
var v4 = b && a; // (4) `b`가 부정적인 값이어서 `&&` 이후를 확인할 필요 없이 `b` 값이 담긴다.
console.log(v4);
// ""
```

```
// 2-68
console.log(0 && 1);
// 0
console.log(1 && 0);
// 0
console.log([] || {});
// []
console.log([] && {});
// {}
console.log([] && {} || 0);
// {}
console.log(0 || 0 || 0 || 1 || null);
// 1
console.log(add(10, -10) || add(10, -10));
// 0
console.log(add(10, -10) || add(10, 10));
// 20
var v;
console.log((v = add(10, -10)) || v++ && 20);
// 0
var v;
console.log((v = add(10, -10)) || ++v && 20);
// 20
```

if else 대체 하기

```
function addFriend(u1, u2) {
  if (u1.friends.indexOf(u2) == -1) {
    if (confirm("친구로 추가할까요?")) {
      u1.friends.push(u2);
      alert('추가 되었습니다.');
    }
  } else {
    alert('이미 친구입니다.')
  }
}
var pj = { name: "PJ", friends: [] };
var ha = { name: "HA", friends: [] };
console.log(addFriend(pj, ha));
// 친구로 추가할까요? -> 확인 -> 추가 되었습니다.
console.log(addFriend(pj, ha));
// 이미 친구입니다.
function addFriend2(u1, u2) {
  (u1.friends.indexOf(u2) == -1 || alert('이미 친구입니다.')) &&
  confirm("친구로 추가할까요?") && u1.friends.push(u2) && alert('추가 되었습니다.');
}
var pj = { name: "PJ", friends: [] };
var ha = { name: "HA", friends: [] };
console.log(addFriend2(pj, ha));
// 친구로 추가할까요? -> 확인 -> 추가 되었습니다.
console.log(addFriend2(pj, ha));
// 이미 친구입니다.
```

삼항연산자

```
// 2-70
var a = false;
var b = a ? 10 : 30;
console.log(b);
// 30
```

```
// 2-71
var a = false;
var c = a ? 10 : function f(arr, v) {
  if (!arr.length) return v;
  v += arr.shift();
  return f(arr, v);
} ([1, 2, 3], 0); // <--- 즉시 실행
console.log(c);
// 6
```

```
// 2-72
var c = a ? 10 : function f(arr, v) {
  return arr.length ? f(arr, v + arr.shift()) : v;
} ([1, 2, 3], 0);
console.log(c);
// 6
```

일반 괄호

```
// 2-73 일반 괄호
(5);
(function() { return 10; });
```

함수를 실행하는 괄호

```
// 2-74 함수를 실행하는 괄호
var add5 = function(a) { // 새로운 공간
  return a + 5;
};
var call = function(f) { // 새로운 공간
  return f();
};
/* 함수를 실행하는 괄호 */
add5(5);
// 10
call(function() { return 10; });
// 10
```

실행 타이밍

```
// 2-75 실행 타이밍
console.log(1);
setTimeout(function() {
  console.log(3)
}, 1000);
console.log(2);
// 1
// 2
// 3 (1초 뒤)
```

콜백 함수로 결과 받기

```
var add = function(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 1000);
};
add(10, 5, function(r) {
  console.log(r);
  // 15
});
```

함수 실행 괄호의 마법과 비동기

```
// 2-77
var add = function(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 1000);
};
var sub = function(a, b, callback) {
  setTimeout(function() {
    callback(a - b);
  }, 1000);
};
var div = function(a, b, callback) {
  setTimeout(function() {
    callback(a / b);
  }, 1000);
};
add(10, 15, function(a) {
  sub(a, 5, function(a) {
    div(a, 10, function(r) {
      console.log(r);
      // 약 3초 후에 2가 찍힘
    });
  });
});
```

```
// 2-78
console.log(div(sub(add(10, 15), 5), 10));
// undefined가 찍히고 callback이 없다는 에러가 남
// Uncaught TypeError: callback is not a function
// Uncaught TypeError: callback is not a function
// Uncaught TypeError: callback is not a function
```

함수를 감싸서 없던 공간 만들기

```
function wrap(func) { // (1) 함수 받기
  return function() { // (2) 함수 리턴하기, 이것이 실행됨
    /* 여기에 새로운 공간이 생김, 나중에 함수를 실행했을 때 이 부분을 거쳐감 */
    return func.apply(null, arguments); // (3)
  }
}
var add = wrap(function(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 1000);
});
add(5, 10, function(r) {
  console.log(r);
  // 15
});
```

실행 이전의 공간에서 비동기 제어와 관련된 일 추가하기

```
function _async(func) {
  return function() {
    arguments[arguments.length++] = function(result) { // (1)
      _callback(result); // (6)
    };
    func.apply(null, arguments);  // (2)
    var _callback; // (3)
    function _async_cb_receiver(callback) { // (4)
      _callback = callback; // (5)
    }
    return _async_cb_receiver;
  };
}
var add = _async(function(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 1000);
});
add(20, 30)(function(r) { // (7)
  console.log(r);
  // 50
});
```

인자를 넘기면서 실행하는 부분과 결과를 받는 부분 분리

```
var add = _async(function(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 1000);
});
var sub = _async(function(a, b, callback) {
  setTimeout(function() {
    callback(a - b);
  }, 1000);
});
var div = _async(function(a, b, callback) {
  setTimeout(function() {
    callback(a / b);
  }, 1000);
});
add(10, 15)(function(a) {
  sub(a, 5)(function(a) {
    div(a, 10)(function(r) {
      console.log(r);
      // 약 3초 후에 2가 찍힘
    });
  });
});
```

```
// 2-82
function _async(func) {
  return function() {
    arguments[arguments.length++] = function(result) {
      _callback(result);
    };
    // 변경된 부분
    (function wait(args) {
      /* 새로운 공간 추가 */
      for (var i = 0; i < args.length; i++)
        if (args[i] && args[i].name == '_async_cb_receiver')
          return args[i](function(arg) { args[i] = arg; wait(args); });
      func.apply(null, args);
    })(arguments);
    var _callback;
    function _async_cb_receiver(callback) {
      _callback = callback;
    }
    return _async_cb_receiver;
  };
}
var add = _async(function(a, b, callback) {
  setTimeout(function() {
    console.log('add', a, b);
    callback(a + b);
  }, 1000);
});
```

```
var sub = _async(function(a, b, callback) {
  setTimeout(function() {
    console.log('sub', a, b);
    callback(a - b);
  }, 1000);
});
var div = _async(function(a, b, callback) {
  setTimeout(function() {
    console.log('div', a, b);
    callback(a / b);
  }, 1000);
});
var log = _async(function(val) {
  setTimeout(function() {
    console.log(val);
  }, 1000);
});
log(div(sub(add(10, 15), 5), 10));
// 약 4초 뒤 2
log(add(add(10, 10), sub(10, 5)));
// 약 3초 뒤 25


// 2-83 추가된 부분 자세히 보기
//  변경 전
func.apply(null, arguments);
// 변경 후
(function wait(args) {
  for (var i = 0; i < args.length; i++)
    if (args[i] && args[i].name == '_async_cb_receiver')
      return args[i](function(arg) { args[i] = arg; wait(args); });
// 재귀
  func.apply(null, args);
})(arguments);
```

익명 함수와 화살표 함수 비교

```
// 한 줄 짜리 함수
var add = function(a, b) { return a + b; };
var add = (a, b) => a + b;
// 두 줄 이상의 함수
var add2 = function(a, b) {
  var result = a + b;
  return result;
};
var add2 = (a, b) => {
  var result = a + b;
  return result;
}
```

익명함수와의 문법비교

```
// 2-85
// 인자가 없는 함수
var hi = function() {
  console.log('hi');
};
var hi = () => console.log('hi');
hi();
// 인자가 하나인 함수
var square = function(a) {
  return a * a;
};
var square = a => a * a;
```

identity와 constant

```
var identity = function(v) {
  return v;
};
var identity = v => v;
var constant = function(v) {
  return function() {
    return v;
  }
};
var constant = v => () => v;
```

익명함수와의 문법비교

```
// 2-87
var gte = (a, b) => a <= b;
var lte = (a, b) => a >= b;
gte(1, 1);
// true
gte(1, 2);
// true
lte(1, 1);
// true
lte(2, 1);
// true
```

익명함수와의 기능비교

```
// 2-88
(function() {
    console.log(this, arguments);
    // {hi: 1} [1, 2, 3]
    (()=> {
        console.log(this, arguments);
        // {hi: 1} [1, 2, 3]
        (()=> {
            console.log(this, arguments);
            // {hi: 1} [1, 2, 3]
        }) ();
    }) ();
}).call({ hi: 1 }, 1, 2, 3)
```

화살표 함수의 실용 사례

```
// 2-89
[1, 2, 3].map(function(v) {
    return v * 2;
});
// [2, 4, 6]
[1, 2, 3].map(v => v * 2);
// [2, 4, 6]
[1, 2, 3, 4, 5, 6].filter(function(v) {
    return v > 3;
});
// [4, 5, 6]
[1, 2, 3, 4, 5, 6].filter(v => v > 3);
// [4, 5, 6]
[1, 2, 3].reduce(function(a, b) {
    return a + b;
});
// 6
[1, 2, 3].reduce((a, b) => a + b);
// 6
```

화살표 함수 재귀

```
function log(arg) {
  console.log(arg);
}
((a, b) => (f => f(f)) (f => log(a) || a++ == b || f(f)))(1, 5);
// 1 2 3 4 5

(function(a, b) {
  (function(f) {
    f(f);
  }) (function(f) {
    log(a) || a++ == b || f(f);
  });
})(6, 10);
// 6 7 8 9 10
```

```
// 2-92
((a, b) => (f => f(f)) (f => log(a) || a++ == b || f(f)))(1, 5);
/* 기억      재귀 시작                    ( 조건부 )    재귀   실행 */
// 1 2 3 4 5
```

화살표 함수 재귀

```
var start = f => f(f);
var logger = (a, b) => start(f => log(a) || a++ == b || f(f));
logger(6, 10);
// 6 7 8 9 10
// 위와 동일한 코드를 function 키워드를 사용하여 확인
var start = function(f) {
  f(f);
};
var logger = function(a, b) {
  start(function(f) {
    log(a) || a++ == b || f(f);
  })
};
logger(1, 5);
// 1 2 3 4 5
```

```
((a) => start(f => log(a) || --a && f(f)))(5);
// 5 4 3 2 1
```

화살표 함수 재귀

```
// 2-95
var each = (arr, iter, i=0) => start(f => iter(arr[i]) || ++i < arr.length && f(f));
each([5, 2, 4, 1], function(v) {
  console.log(v);
});
// 5 2 4 1
each(['a', 'b', 'c'], function(v) {
  console.log(v);
});
// a b c
```

Contact

ABCD Facebook Community
https://www.facebook.com/groups/aboutCoding/
ABCD 홈페이지
http://abcds.kr/
ABCD Slack
https://abcds.slack.com/messages/C0GS0ENFK

dev.h
http://devh.kr
https://www.facebook.com/devhkr