함수형 프로그래밍 시작해보기

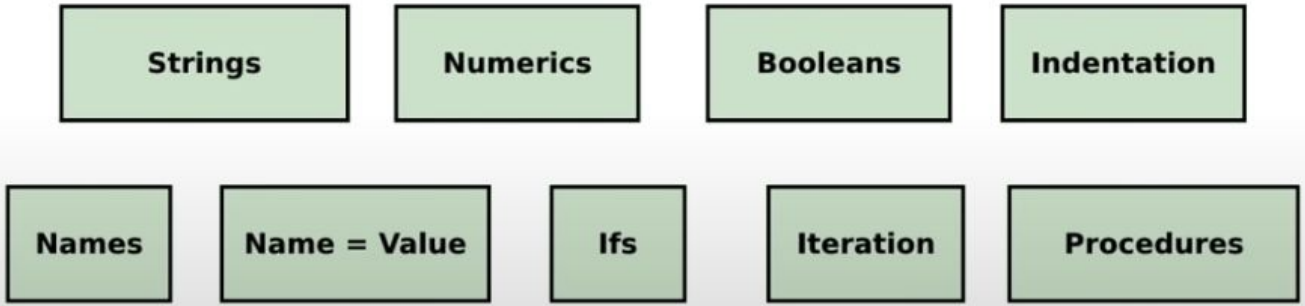
함수형 프로그래밍에 대해서 아무것도 몰랐던 유기성

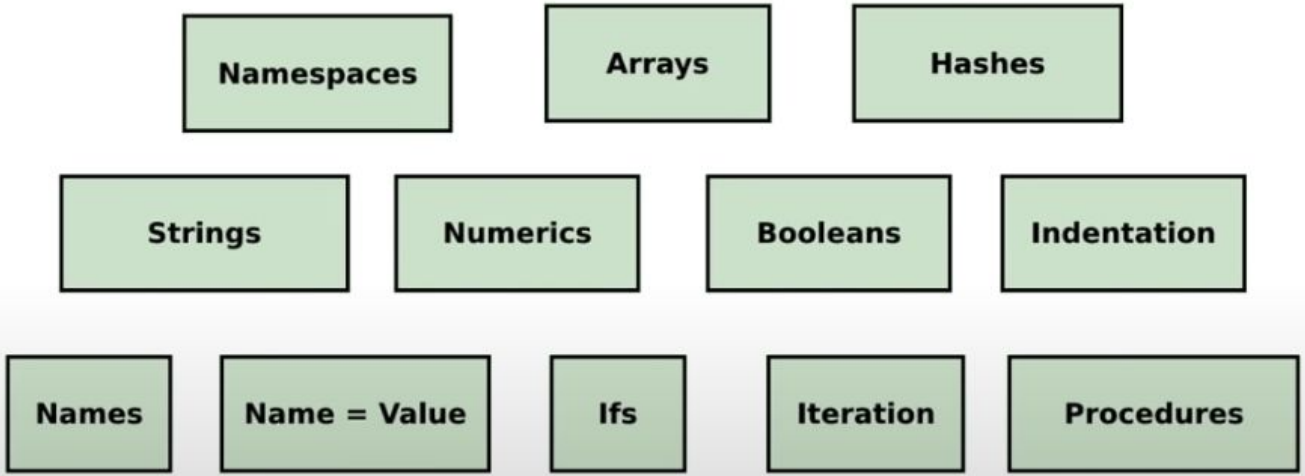
FORGET

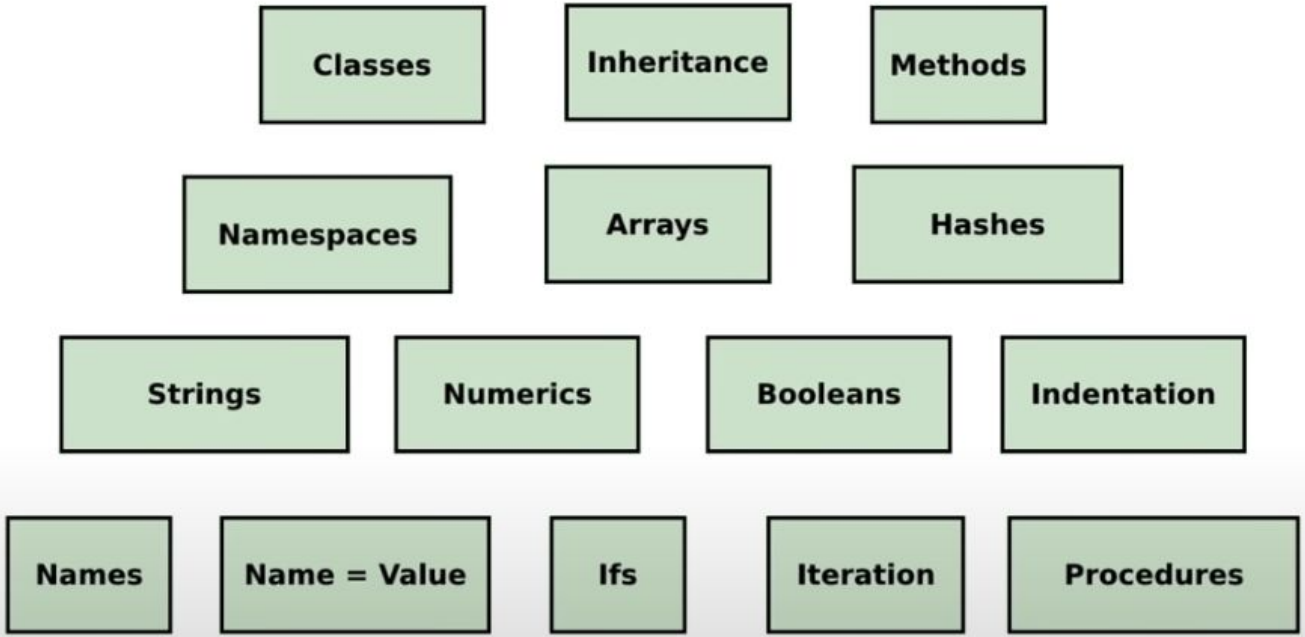
Everything You Know About Programming

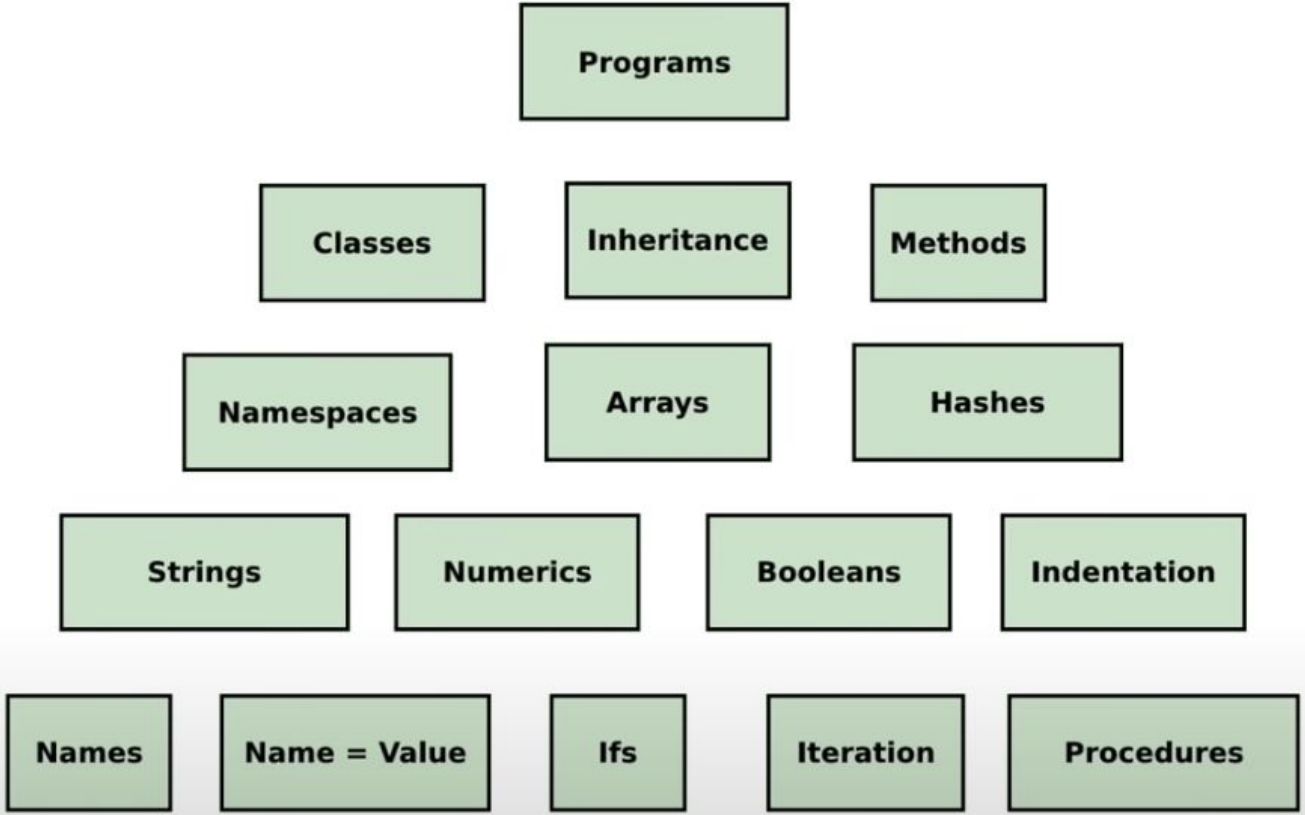
What Do You Know?


Names
Name = Value
Ifs
Iteration
Procedures


Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration
Procedures


Namespaces
Arrays
Hashes
Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration
Procedures


Classes
Inheritance
Methods
Namespaces
Arrays
Hashes
Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration
Procedures


Programs
Classes
Inheritance
Methods
Namespaces
Arrays
Hashes
Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration
Procedures

FORGET

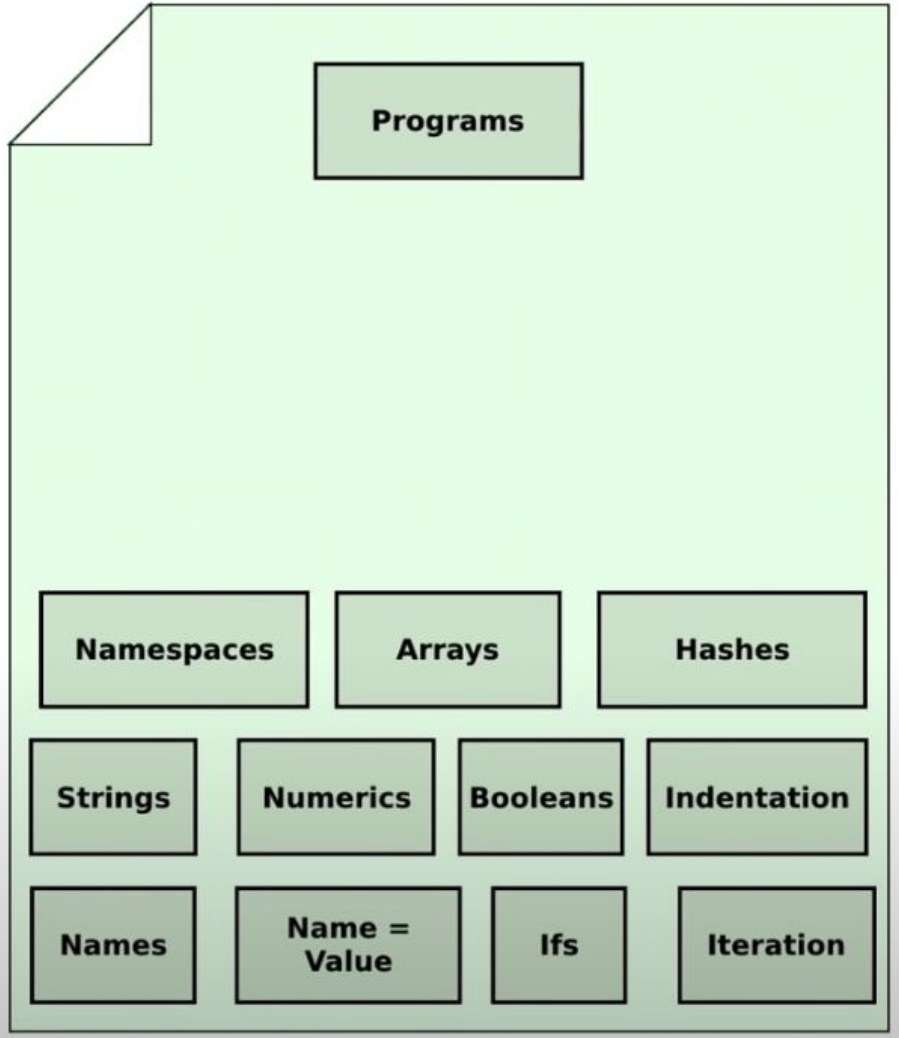
Everything You Know About Programming

REALLY?

Functional Programming

Refactor

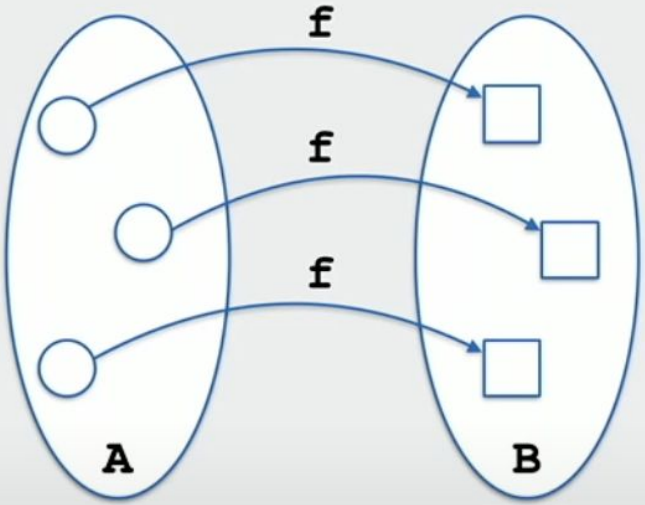
Everything You Know About Programming


Programs
Namespaces
Arrays
Hashes
Strings
Numerics
Booleans
Indentation
Names
Name =
Value
Ifs
Iteration


f(x)

Notations - Function Types

- `f :: A -> B`



Look at Input

Produce Output

No Side Effects

Pure Functional Programming

- Programs composed of “pure” functions =>
  - No side effects
  - Can be “memoized” i.e. always return the same (i.e. it’s a map)
- Side effects are[1]

Modifying a variable	변수에 새로운 값 할당
Modifying a data structure in place	자료구조를 in-place 로 변경
Setting a field on an object	오브젝트 어떤 필드의 값 세팅
Throwing an exception or halting with an error	예외를 던지거나 오류 발생해서 프로그램 중단
Printing to the console or reading user input	콘솔 입출력
Reading from or writing to a file	파일 입출력
Drawing on the screen	스크린에 그림 그리기, 렌더링

- Declarative style instead of imperative -> What not How

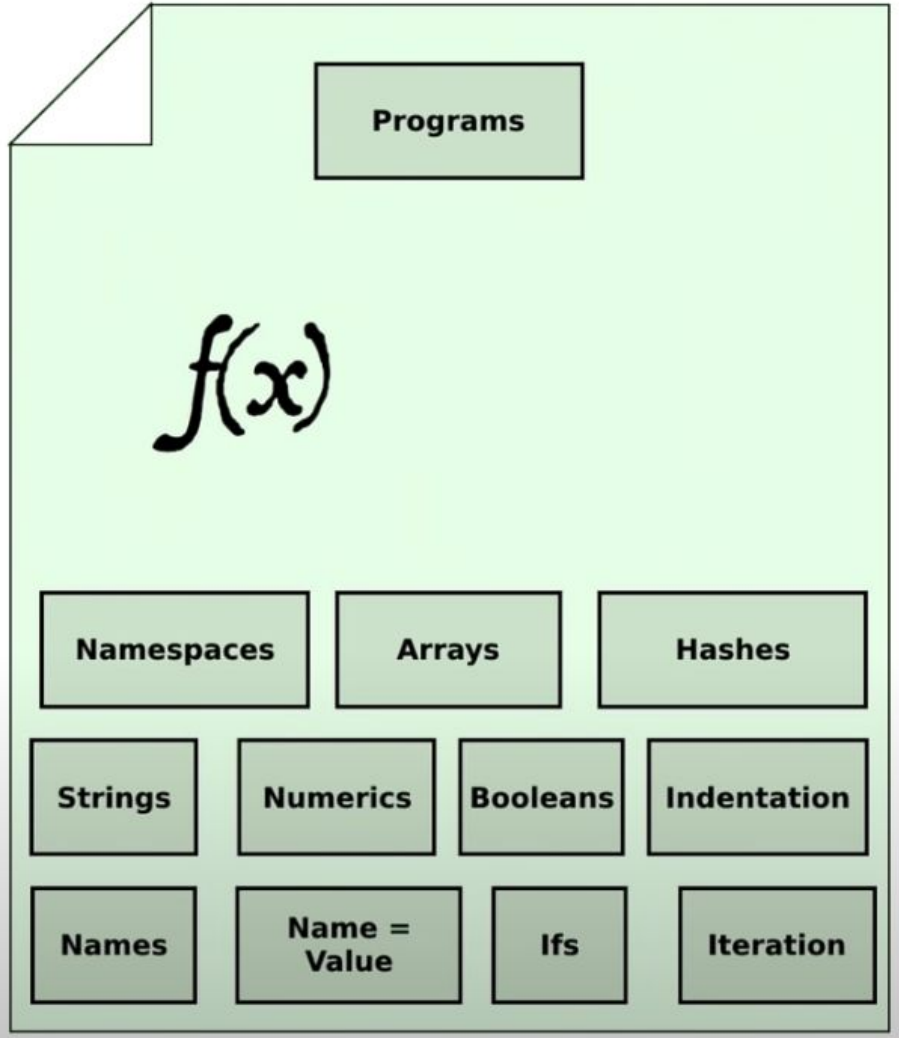
[1]Chapter 1 of “Functional Programming”, Paul Chiusano & Runar Bjarnason

```
x = ['a','b','c']

y = a(x)

x = ?
```

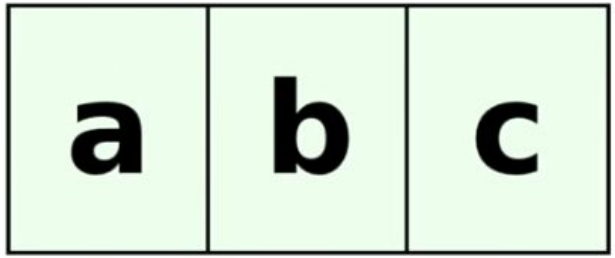
```
x = ['a','b','c']
```

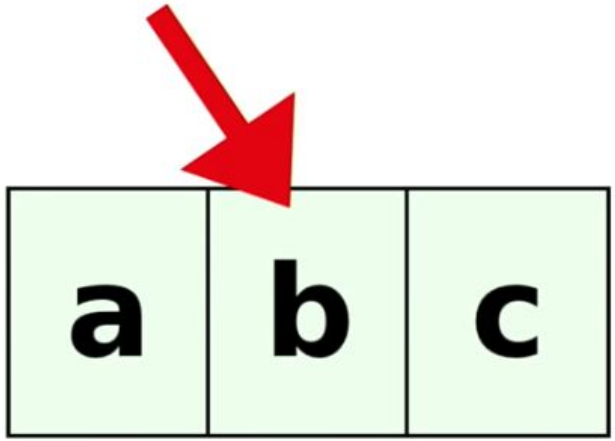
```
y = c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x) - ceil(g(x) -
c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x)
```

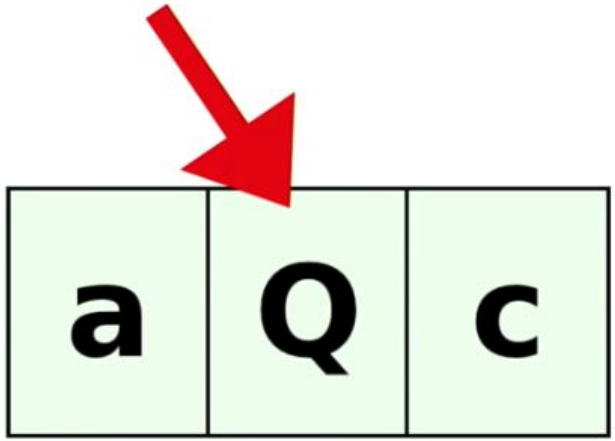
```
x = ?
```


Programs
f(x)
Namespaces
Arrays
Hashes
Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration

Problem


a
b
c


a
b
c


a
Q
c

```
x = ['a','b','c']
```

```
y = c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x) - ceil(g(x) -
c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x)
```

```
x = ?
```

```
t = c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x) - ceil(g(x)
```

x[2] = 'Q'

```
y = t + c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x)
```

```
t = c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x) - ceil(g(x)
x[2] = 'Q'
y = t + c(a(x) + b(x)) + d(x) * e(x) - 4 * (q(x) - f(x)) +
d(x) * e(x) - 4 * (q(x) - f(x)) + b(x)) + d(x) /
c(a(x) + b(x)) + d(x) * e(x)  + g(x) + h(x) *
floor(b(x)) + 16 * a(x)
```

Immutable

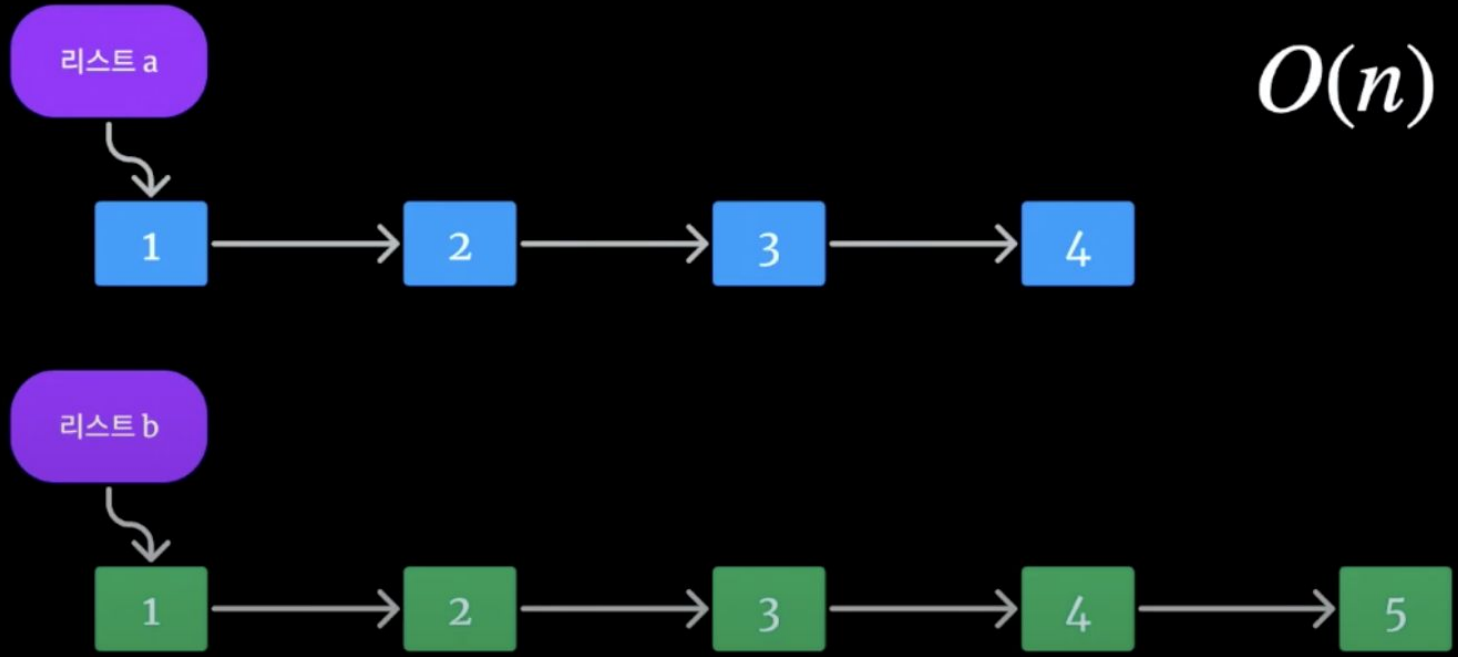
Copies

Copies

Copies

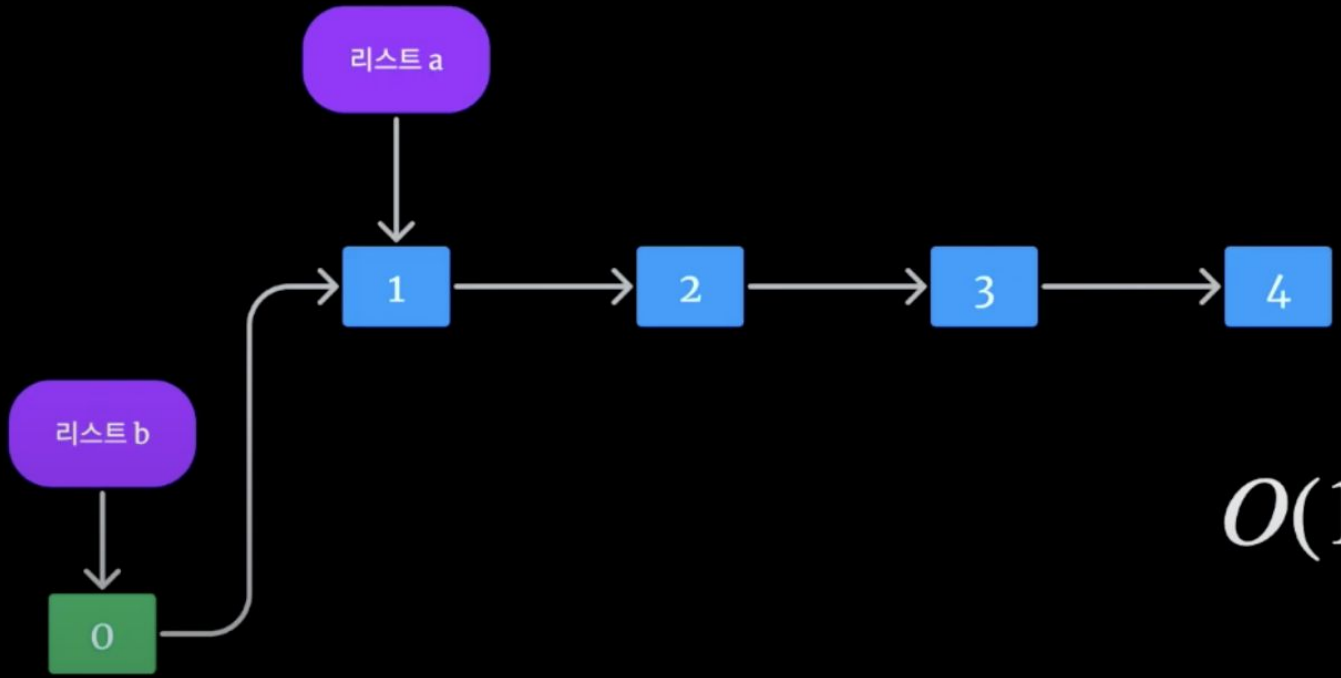
Persistent Data Structures

List(1, 2, 3, 4).append(5)



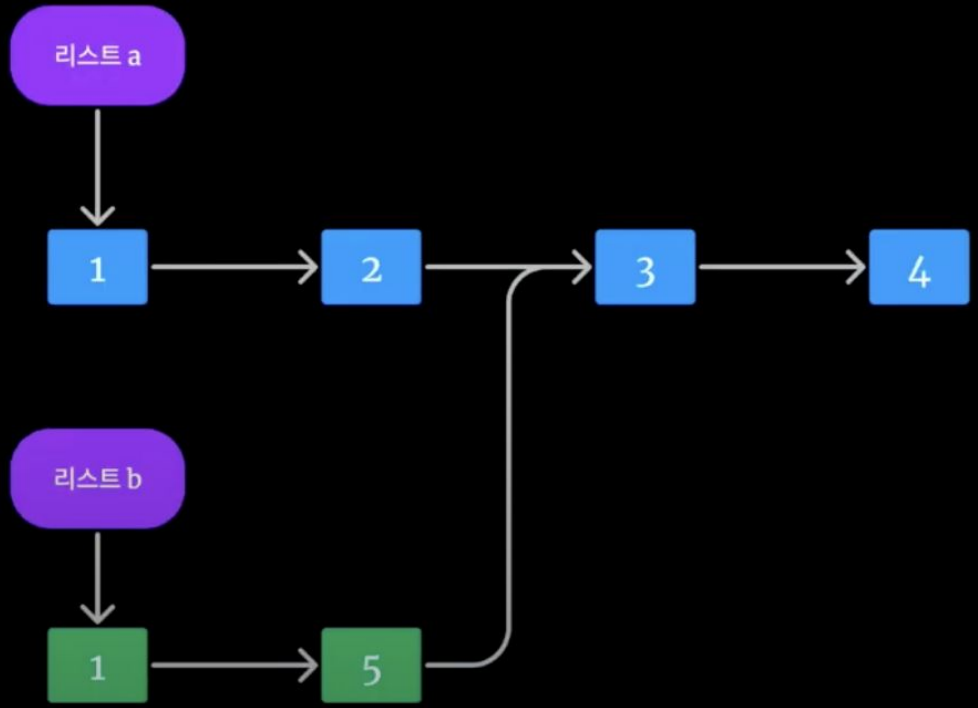
$O(n)$

List(1, 2, 3, 4).prepended(0)

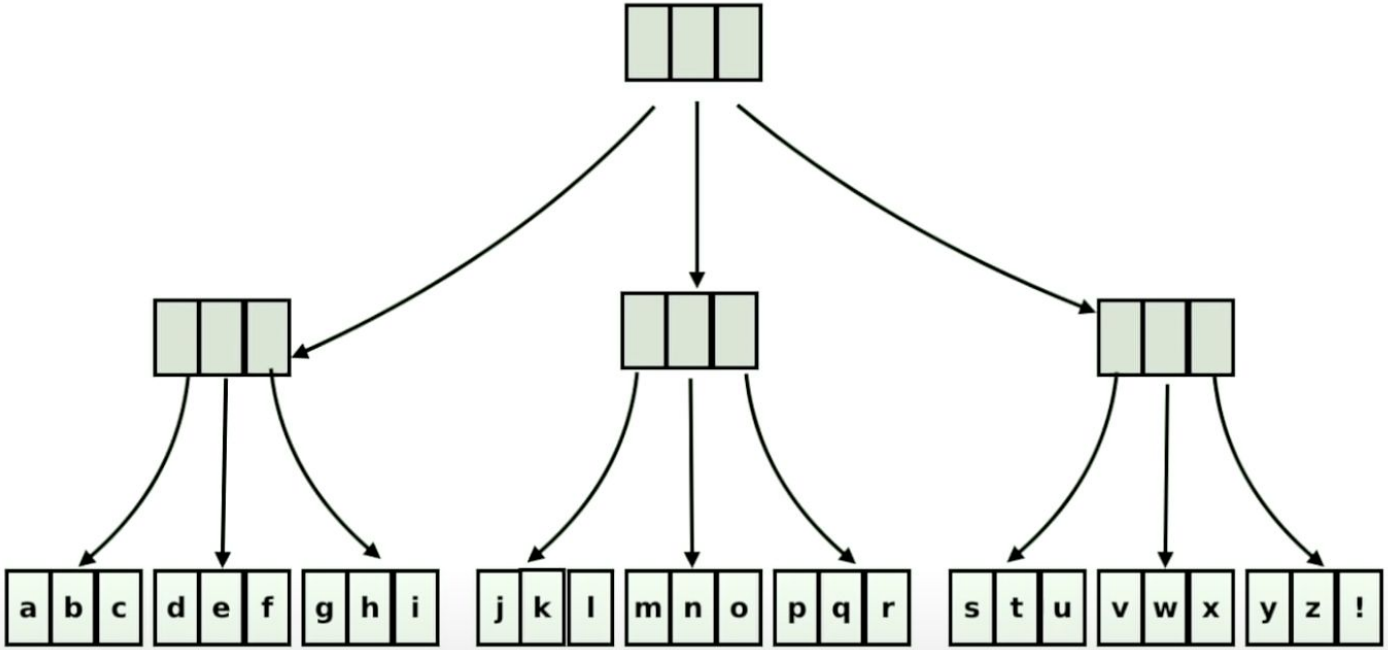


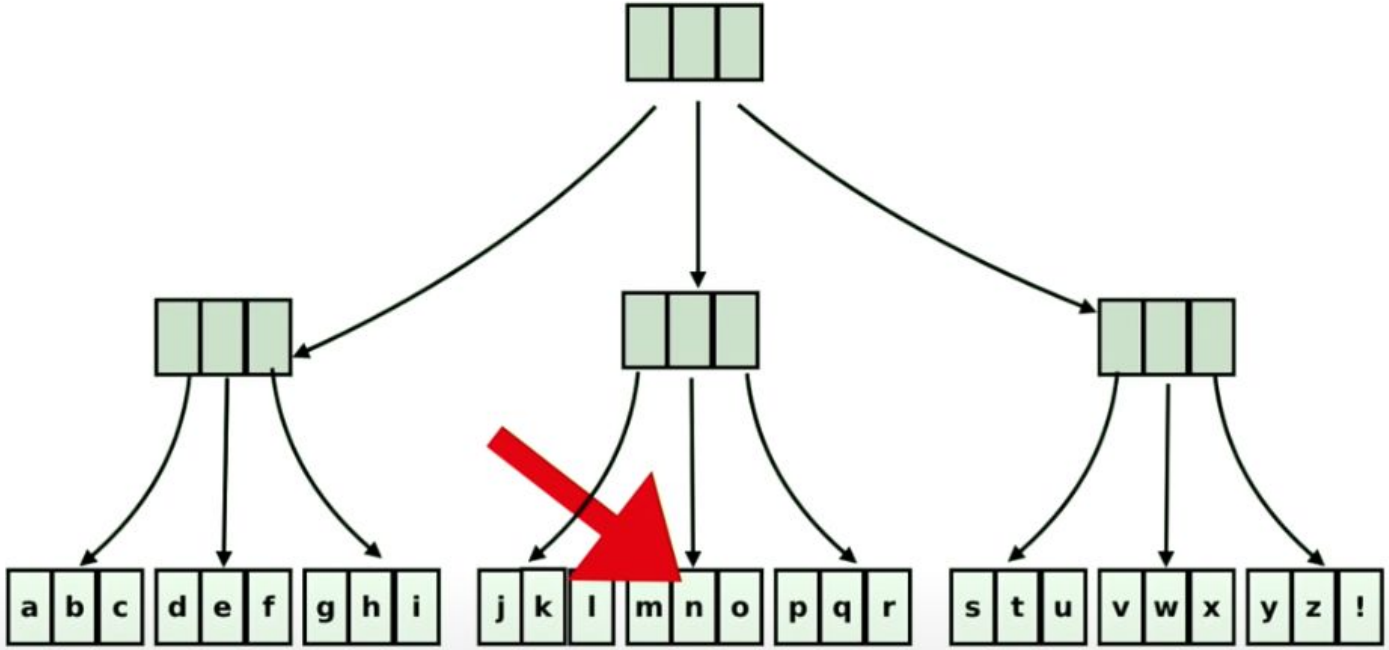
$O(1)$

List(1, 2, 3, 4).updated(1, 5)



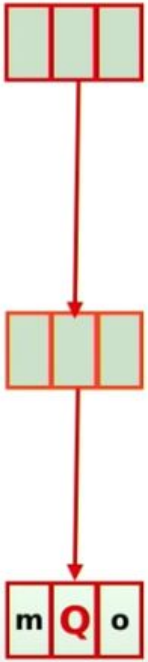
$O(n)$

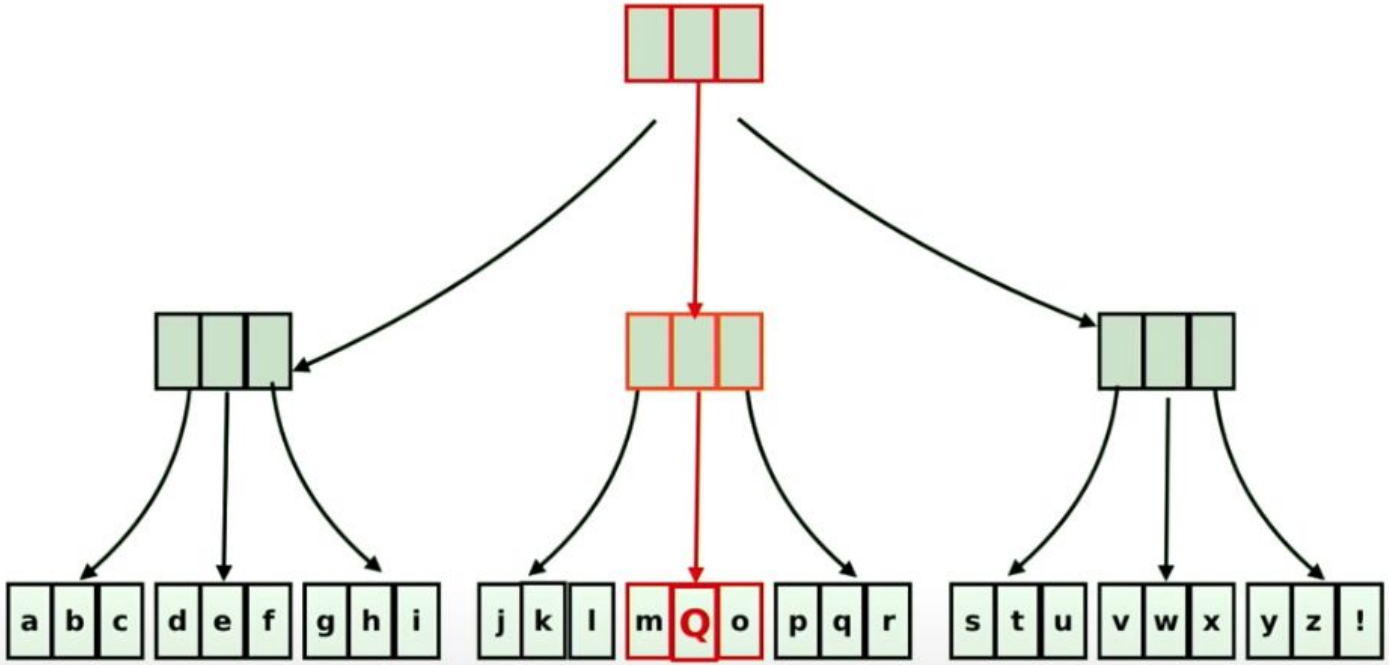

a b c
d e f
g h i
j k l
m n o
p q r
s t u
v w x
y z !


a b c
d e f
g h i
j k l
m n o
p q r
s t u
v w x
y z !


mQo

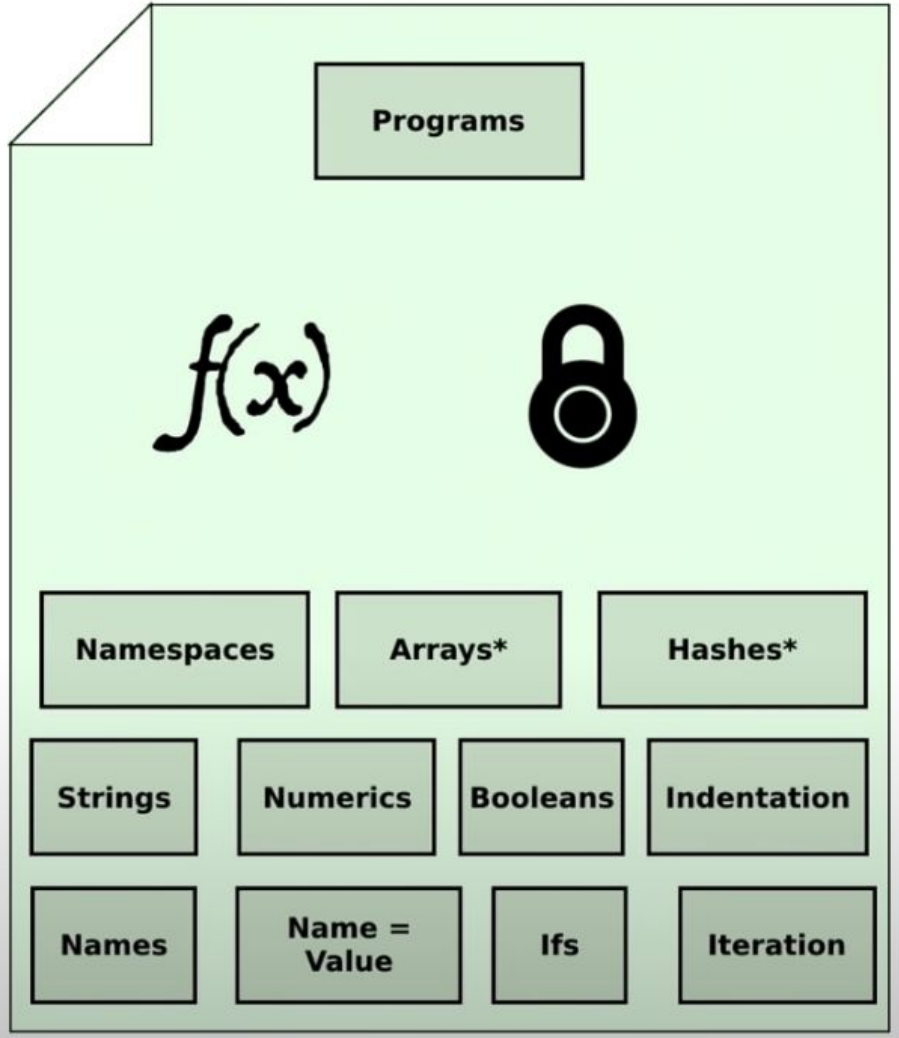

m
Q
o


m
Q
o


a b c
d e f
g h i
j k l
m Q o
p q r
s t u
v w x
y z !

3


32


Programs
f(x)
Namespaces
Arrays*
Hashes*
Strings
Numerics
Booleans
Indentation
Names
Name = Value
Ifs
Iteration

pure functions are **basically lookup tables**

`stringLength`

argument	returns
`"Hi"`	2
`"GOTO"`	4
`"Copenhagen"`	10

pure functions are **basically lookup tables**

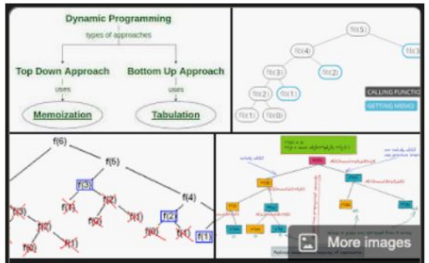
`stringLength`

key	value
`"Hi"`	2
`"GOTO"`	4
`"Copenhagen"`	10

pure functions are **basically lookup tables**

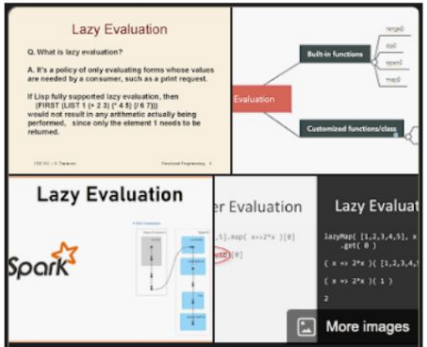
how to tell if a function is pure:

"Could this function's body be replaced with one big **lookup table access**?"



Memoization

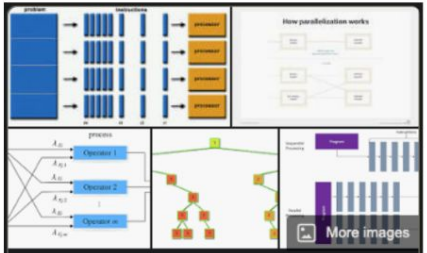
In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls to pure functions and returning the cached result when the same inputs occur again. Wikipedia



Lazy evaluation

Programming paradigm

In programming language theory, lazy evaluation, or call-by-need, is an evaluation strategy which delays the evaluation of an expression until its value is needed and which also avoids repeated evaluations. Wikipedia



Parallel computing

Programming paradigm

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. Wikipedia

People also search for

View 10+ more

Computing | Distributed computing | Grid computing | Artificial intelligence

Immutable

vs

const

함수형 프로그래밍 적용해보기

Actions

the process of doing something, typically to achieve an aim

- In FP jargon: *impure functions, effects,* or *side-effects*
- Rule of thumb: Actions depend on
  - when you run them *or*
  - how many times you run them

Calculations

computation from inputs to outputs

- In FP jargon: *pure functions, referentially transparent*
- Eternal — outside of time

Data

facts about events used as a basis for reasoning, discussion, or calculation

- Inert
- Serializable
- Requiring interpretation

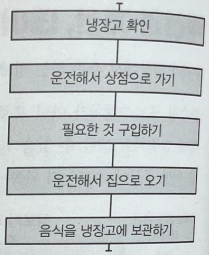
모든 것을 액션으로 분류하면 안 될 것 같습니다. 물론 아주 단순한 과정은 액션으로만 분류할 수 있지만 위에서 살펴본 장보기 과정은 그렇게 단순하지 않습니다. 놓친 것이 있는지 단계별로 다시 살펴봅시다.

냉장고 확인하기

냉장고를 확인하는 일은 확인하는 시점이 중요하기 때문에 **액션**입니다. 냉장고에 가지고 있는 제품은 **데이터**입니다. 이것을 **현재 재고**current inventory라고 합시다.

현재 재고

장보기 과정



운전해서 상점으로 가기

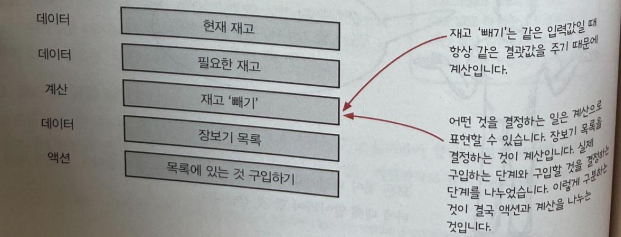
운전해서 상점으로 가는 것은 복잡한 행동이고 명확히 **액션**입니다. 사실 여기에는 **데이터**가 숨어 있습니다. 상점 위치나 가는 경로는 데이터로 볼 수 있습니다. 하지만 우리가 할 일이 자율 주행 자동차를 만드는 것이 아니기 때문에 따로 분리하지 않겠습니다.

필요한 것 구입하기

구입하는 일도 확실히 **액션**입니다. 하지만 구입 과정은 몇 단계로 나눌 수 있습니다. 필요한 것을 구입하려면 필요한 것이 무엇인지 알아야 합니다. 필요한 것은 어떻게 알 수 있을 까요? 어떻게 장을 볼지에 따라 다르겠지만 필요하지만 없는 제품의 목록을 만드는 것이 가장 쉽습니다.

필요한 재고 - 현재 재고 = 장보기 목록

앞에 '냉장고 확인하기' 단계에서 만든 **현재 재고 데이터**를 사용했습니다. 이제 '필요한 것 구입하기' 단계를 몇 단계로 더 나눌 수 있습니다.



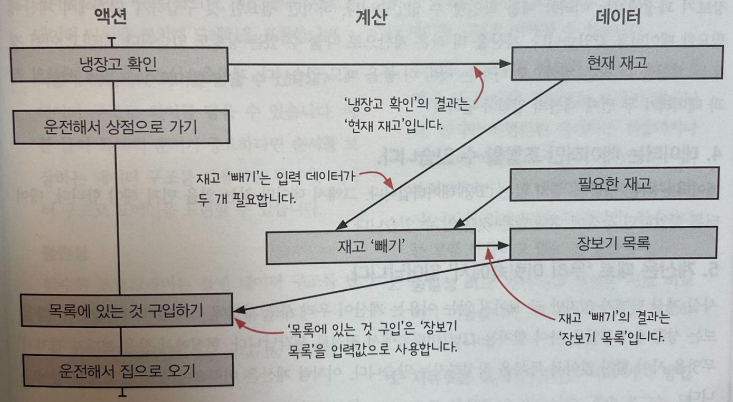
운전해서 집으로 오기

운전해서 집으로 오는 단계도 더 나눌 수 있지만, 우리가 다루려고 하는 범위가 아니기 때문에 더 나누지 않겠습니다.

그럼 앞에 이야기한 내용을 반영해서 장보기 과정을 다시 정리해 봅시다.

액션과 계산, 데이터를 더 명확하게 하기 위해 액션과 계산, 데이터에 해당하는 단계를 각각 다른 열에 그려봅시다. 그리고 데이터는 액션과 계산의 입력과 출력으로 사용되기 때문에 선으로 연결해 봅시다. 이제 장보기 과정이 완성된 것 같습니다.

장보기 과정



이렇게 반복하면 액션과 계산, 데이터를 더 많이 찾을 수 있고 풍부한 모델을 만들 수 있습니다.

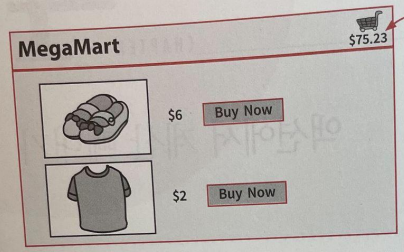
예를 들어 '냉장고 확인하기' 단계는 '냉장실 확인하기' 단계와 '냉동실 확인하기' 단계로 나눠 액션으로 만들 수 있습니다. 각 액션은 각각의 데이터를 만들 수 있고 연결할 수 있습니다. '목록에 있는 것 구입하기' 단계도 '장바구니에 담기'와 '계산하기' 단계로 나눌 수 있습니다.

계속 나누다 보면 점점 더 복잡해진다고 생각할 수 있습니다. 하지만 액션에 숨어 있는 다른 액션이나 계산 또는 데이터를 발견하기 위해 나눌 수 있는 만큼 나누는 것이 좋습니다.

MegaMart.com에 오신 것을 환영합니다

여러분의 장바구니는 항상 가득 차 있습니다.

MegaMart는 온라인 쇼핑몰입니다. 경쟁력을 유지하고 있는 중요한 기능 중 하나는 쇼핑 중에 장바구니에 담겨 있는 제품의 금액 합계를 볼 수 있는 기능입니다.



항상 현재 장바구니 금액 합계를 보여줍니다.

MegaMart 코드를 공개합니다.

보안 서약은 하지 않아도 됩니다.

장바구니 제품과 금액 합계를 담고 있는 전역변수

```
var shopping_cart = [];
var shopping_cart_total = 0;

function add_item_to_cart(name, price) {
  shopping_cart.push({
    name: name,
    price: price
  });
  calc_cart_total();
}

function calc_cart_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
  set_cart_total_dom();
}
```

장바구니에 제품을 담기 위해 cart 배열에 레코드를 추가

장바구니 제품이 바뀌었기 때문에 금액 합계를 업데이트

모든 제품값 더하기

금액 합계를 반영하기 위해 DOM 업데이트

마지막 줄은 웹페이지를 변경하기 위해 **DOM**document object model을 업데이트하는 코드입니다.

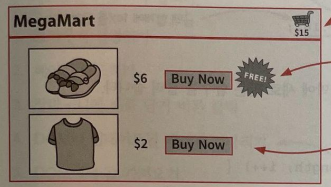
용어 설명

DOM(document object model)은 웹브라우저 안에 있는 HTML 페이지를 메모리상에 표현한 것입니다.

무료 배송비 계산하기

새로운 요구사항

MegaMart는 구매 합계가 20 달러 이상이면 무료 배송을 해주려고 합니다. 그래서 장바구니에 넣으면 합계가 20달러가 넘는 제품의 구매 버튼 옆에 무료 배송 아이콘을 표시해 주려고 합니다.



현재 장바구니에 들어있는 제품 금액의 합계를 보여줍니다.

이 제품을 장바구니에 담으면 총 구입 금액이 21달러이기 때문에 무료 배송 아이콘을 표시해 줍니다.

이 제품은 장바구니에 담아도 총 구입 금액이 17달러이기 때문에 무료 배송 아이콘을 표시하지 않습니다.

절차적인 방법으로 구현하기

절차적인 방법이 더 이해하기 쉬울 수 있습니다.

구매 버튼에 무료 배송 아이콘을 표시하기 위한 함수를 만듭니다. 지금은 이해하기 쉬운 절차적 스타일로 작성하지만, 뒤에서 함수형 스타일로 리팩터링을 하겠습니다.

```
function update_shipping_icons() {
  var buy_buttons = get_buy_buttons_dom();
  for(var i = 0; i < buy_buttons.length; i++) {
    var button = buy_buttons[i];
    var item = button.item;
    if(item.price + shopping_cart_total >= 20)
      button.show_free_shipping_icon();
    else
      button.hide_free_shipping_icon();
  }
}
```

페이지에 있는 모든 구매 버튼을 가져와 반복문을 적용합니다.

무료 배송이 가능한지 확인합니다.

결정에 따라 무료 배송 아이콘을 보여주거나 보여주지 않습니다.

합계 금액이 바뀔 때마다 모든 아이콘을 업데이트하기 위해 `calc_cart_total()` 함수 마지막에 `update_shipping_icons()` 함수를 불러 줍니다.

```
function calc_cart_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
  set_cart_total_dom();
  update_shipping_icons();
}
```

앞에서 만든 함수입니다.

여기에 아이콘을 업데이트하는 코드를 추가하겠습니다.

MegaMart 개발팀의 모토

동작하면 배포한다!

세금 계산하기

다음 요구사항

장바구니의 금액 합계가 바뀔 때마다 세금을 다시 계산해야 합니다. 함수형 프로그래밍을 적용하지 않고 이 기능을 추가하는 일은 어렵지 않습니다.

```
function update_tax_dom() {
  set_tax_dom(shopping_cart_total * 0.10);
}
```

함수를 새로 만듭니다.

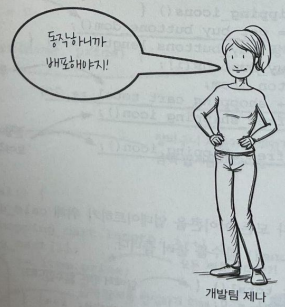
금액 합계에 10%를 곱합니다.

DOM을 업데이트합니다.

앞에서 한 것처럼 calc_cart_total() 함수 마지막에 새로 만든 함수를 불러 줍니다.

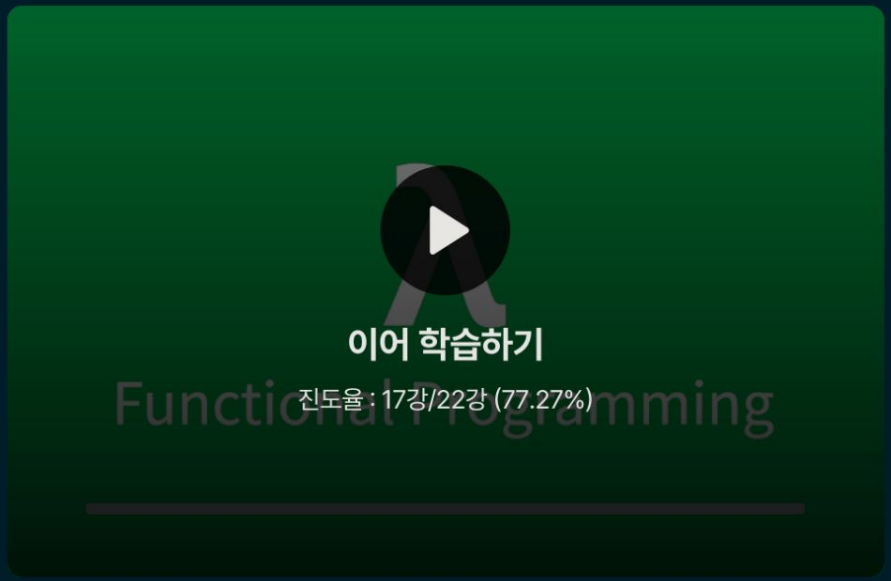
```
function calc_cart_total() {
  shopping_cart_total = 0;
  for(var i = 0; i < shopping_cart.length; i++) {
    var item = shopping_cart[i];
    shopping_cart_total += item.price;
  }
  set_cart_total_dom();
  update_shipping_icons();
  update_tax_dom();
}
```

페이지에 세금을 업데이트하기 위해 코드를 추가합니다.



개발팀 제나

예제코드로 파악해보기



개발 · 프로그래밍 > 프로그래밍 언어

자바스크립트로 알아보는 함수형 프로그래밍 (ES5)

★★★★★ (4.9) 297개의 수강평 · **12437명**의 수강생

유인동

JavaScript 함수형 프로그래밍

폴더에 추가 공유하기

README.rst

pydash

pypi v7.0.6 | build passing | coverage 100% | license MIT License

The kitchen sink of Python utility libraries for doing "stuff" in a functional way. Based on the Lo-Dash Javascript library.

Note

Looking for a library that is more memory efficient and bette
around generators and iteration and has iteratee-first functic

Links

- Project: https://github.com/dgilland/pydash
- Documentation: http://pydash.readthedocs.org
- PyPi: https://pypi.python.org/pypi/pydash/
- Github Actions: https://github.com/dgilland/pydash/actic

README.md

Lodust - API Documentation

Lodust is the Rust version of **Lodash**, which is a modern Javascript utilty library delivering modularity, performance & extras.

README.md

lodash

Site | Docs | Contributing | Wiki | Code of Conduct

The Lodash library exported as a UMD module.

```
$ bun run build
$ lodash -o ./dist/lodash.js
$ lodash core -o ./dist/lodash.core.js
```

Download

- Core build (~4 kB gzipped)
- Full build (~24 kB gzipped)
- CDN copies jsDelivr 271M hits/month

Lodash is released under the MIT license & supports modern e
Review the build differences & pick one that's right for you.

README.md

Tilde

Tilde is a functional tool-belt for Android. It not just only provides functional methods for Kotlin but also some wrapper for Java so you can play around with the library in Java too. Tilde is inspired from Dollar in Swift which is similar to Lodash and Underscore.js in Javascript. It also provides Java 6 and 7 users with almost all the useful methods in Java 8.

> NOTE: This library is under development and documentation is incomplete

README.md

Dollar

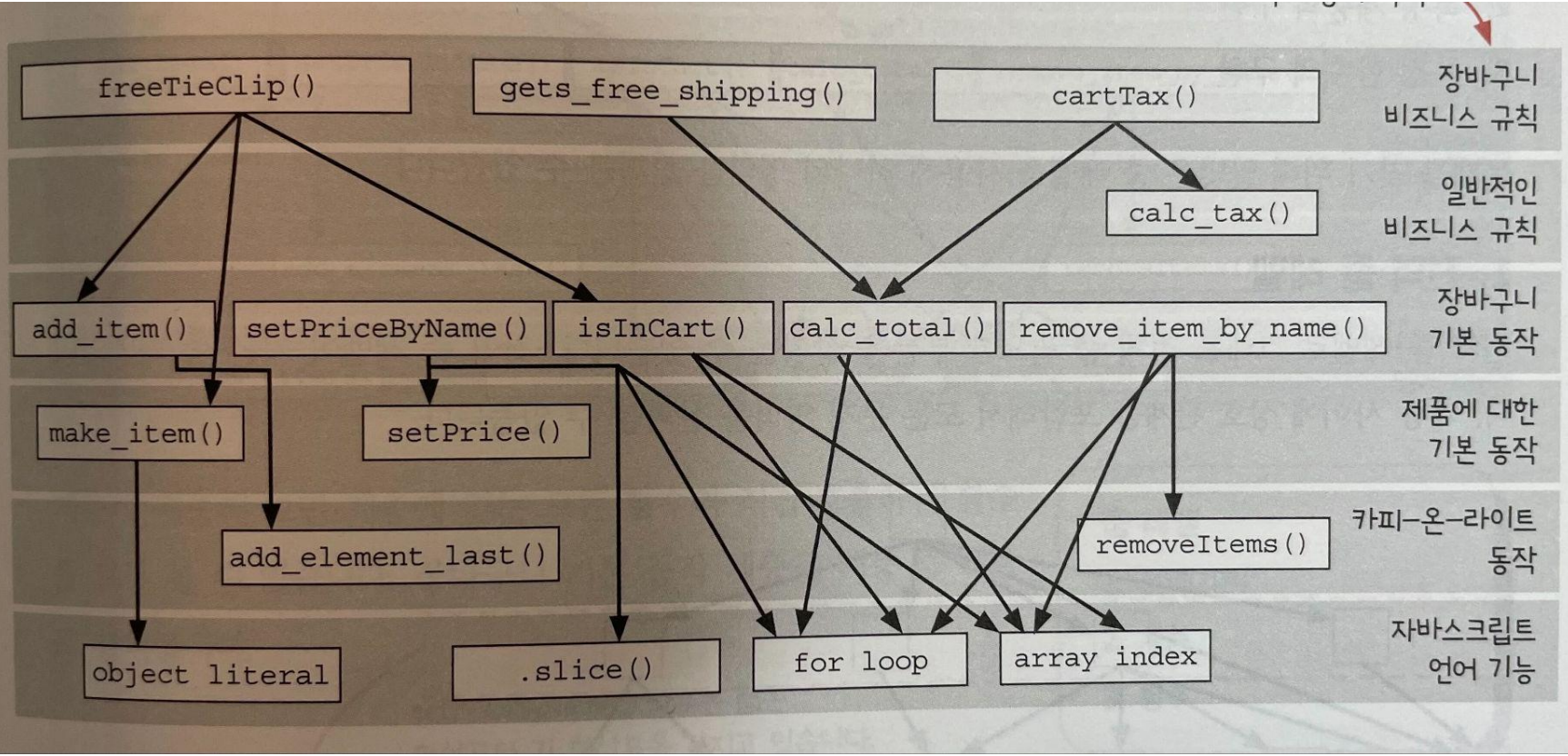
build unknown | pod v9.0.0 | Reviewed by Hound

gitter join chat

Dollar is a Swift library that provides useful functional programming helper methods without extending any built in objects. It is similar to Lo-Dash or Underscore.js in Javascript.

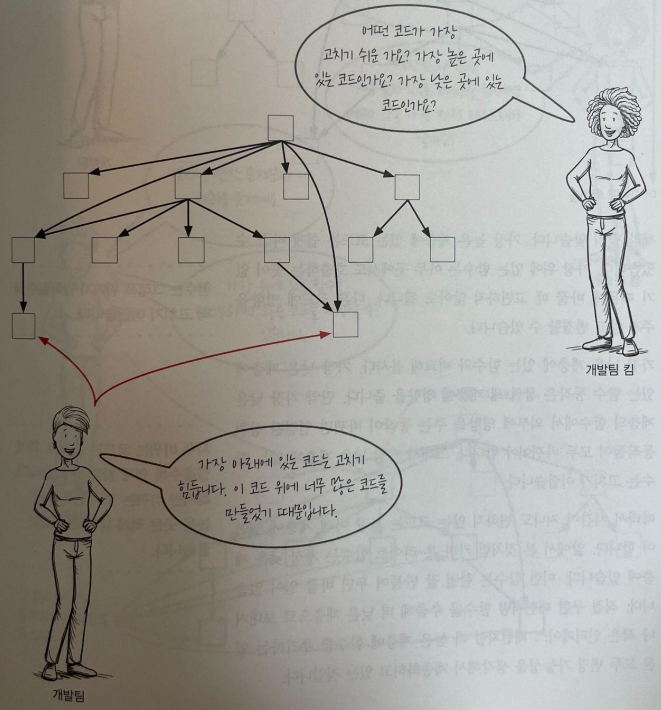
Cent is a library that extends certain Swift object types using the extension feature and gives its two cents to Swift language. It is now moved into a separate repo to support `Swift Package Manager`

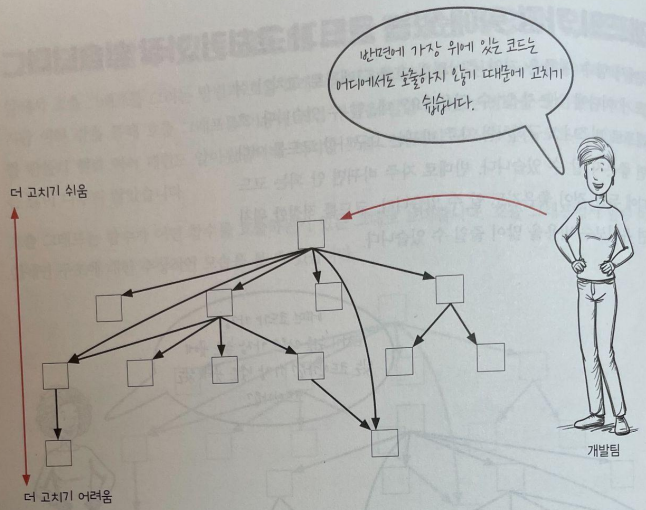
NOTE: Starting Swift 4 `$` is no longer a valid identifier. So you get the following `error: '$' is not an identifier; use backticks to escape it` . Instead use `Dollar` .


freeTieClip()
gets_free_shipping()
cartTax()
장바구니
비즈니스 규칙
calc_tax()
일반적인
비즈니스 규칙
add_item()
setPriceByName()
isInCart()
calc_total()
remove_item_by_name()
장바구니
기본 동작
make_item()
setPrice()
제품에 대한
기본 동작
add_element_last()
removeItems()
카피-온-라이트
동작
object literal
.slice()
for loop
array index
자바스크립트
언어 기능

그래프의 가장 위에 있는 코드가 고치기 가장 쉽습니다

그래프에서 함수 이름을 지워 추상화한 호출 그래프로 고치기 쉬운 코드가 어디에 있는지 알 수 있을까요? 네, 알 수 있습니다. 호출 그래프로 비즈니스 규칙처럼 자주 바뀌는 요구사항 코드를 어디에 두면 좋은지 알 수 있습니다. 반대로 자주 바뀌면 안 되는 코드는 어디에 두는 것이 좋은지도 알 수 있습니다. 코드를 적절한 위치에 두면 유지보수 비용을 많이 줄일 수 있습니다.





세라 말이 맞습니다. 가장 높은 계층에 있는 코드는 쉽게 바꿀 수 있습니다. 가장 위에 있는 함수는 아무 곳에서도 호출하는 곳이 없기 때문에 바꿀 때 고민하지 않아도 됩니다. 다른 코드에 영향을 주지 않고 변경할 수 있습니다.

함수는 그래프 위에서 멀어질수록 더 고치기 어렵습니다.

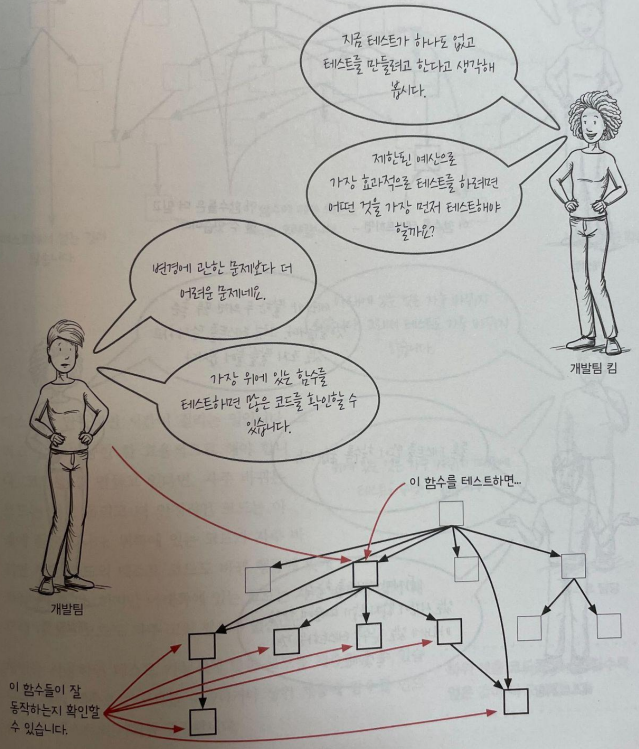
가장 낮은 계층에 있는 함수와 비교해 봅시다. 가장 낮은 계층에 있는 함수 동작은 상위 세 계층에 영향을 줍니다. 만약 가장 낮은 계층의 함수에서 외부에 영향을 주는 동작이 바뀌면 연결된 상위 동작들이 모두 바뀌어야 합니다. 그래서 가장 낮은 계층에 있는 함수는 고치기 어렵습니다.

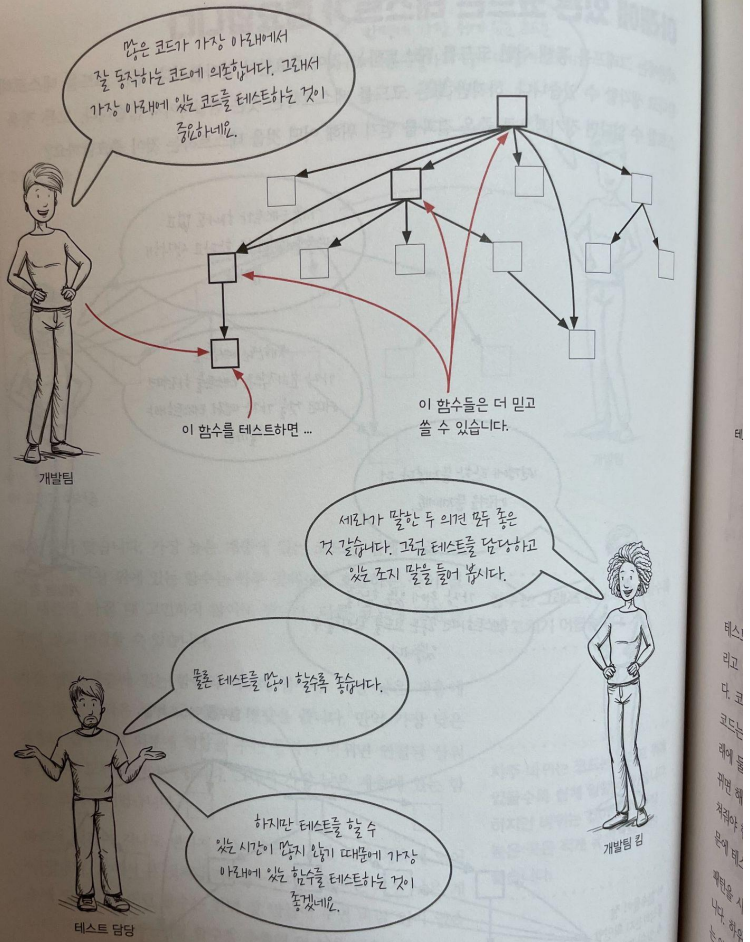
따라서 시간이 지나도 변하지 않는 코드는 가장 아래 계층에 있어야 합니다. 앞에서 본 것처럼 카피-온-라이트 함수는 가장 낮은 계층에 있습니다. 이런 함수는 한번 잘 만들어 두면 바꿀 일이 없습니다. 직접 구현 패턴처럼 함수를 추출해 더 낮은 계층으로 보내거나 작은 인터페이스 패턴처럼 더 높은 계층에 함수를 추가하는 일은 모두 변경 가능성을 생각해서 계층화하고 있는 것입니다.

자주 바뀌는 코드는 그래프 위에 있을수록 쉽게 일할 수 있습니다. 하지만 바뀌는 것이 많은 가장 높은 곳은 적게 유지하는 것이 좋습니다.

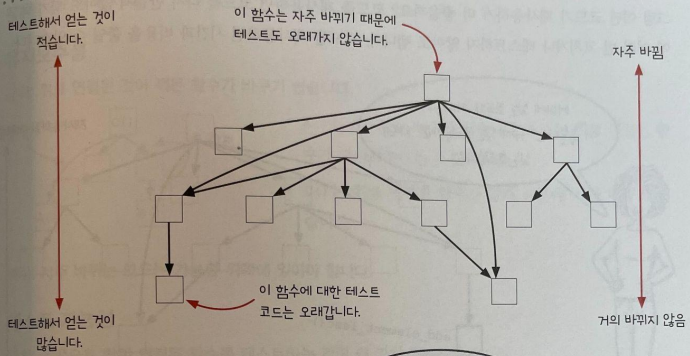
아래에 있는 코드는 테스트가 중요합니다

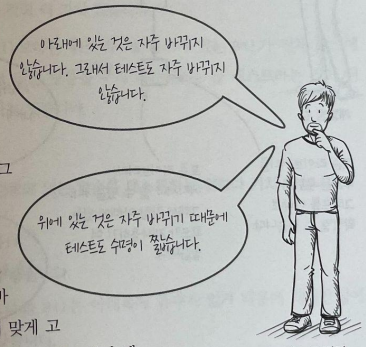
이번에는 그래프를 통해 어떤 코드를 테스트하는 것이 중요한지 알아봅시다. 모든 코드를 테스트해야 한다고 생각할 수 있습니다. 하지만 모든 코드를 테스트하는 것은 현실적이지 않습니다. 모든 것을 테스트할 수 없다면 장기적으로 좋은 결과를 얻기 위해 어떤 것을 테스트하는 것이 중요할까요?




짠은 코드가 가장 아래에서
잘 동작하는 코드에 의존합니다. 그래서
가장 아래에 있는 코드를 테스트하는 것이
중요하네요.
개발팀
이 함수를 테스트하면 ...
이 함수들은 더 믿고
쓸 수 있습니다.
세라가 말한 두 의견 모두 좋은
것 같습니다. 그점 테스트를 담당하고
있는 조지 말을 들어 봅시다.
개발팀 킴
물론 테스트를 많이 할수록 좋습니다.
하지만 테스트를 할 수
있는 시간이 많지 않기 때문에 가장
아래에 있는 함수를 테스트하는 것이
좋겠네요.
테스트 담당

제대로 만들었다면 가장 아래에 있는 코드보다
가장 위에 있는 코드가 더 자주 바뀔 것입니다.





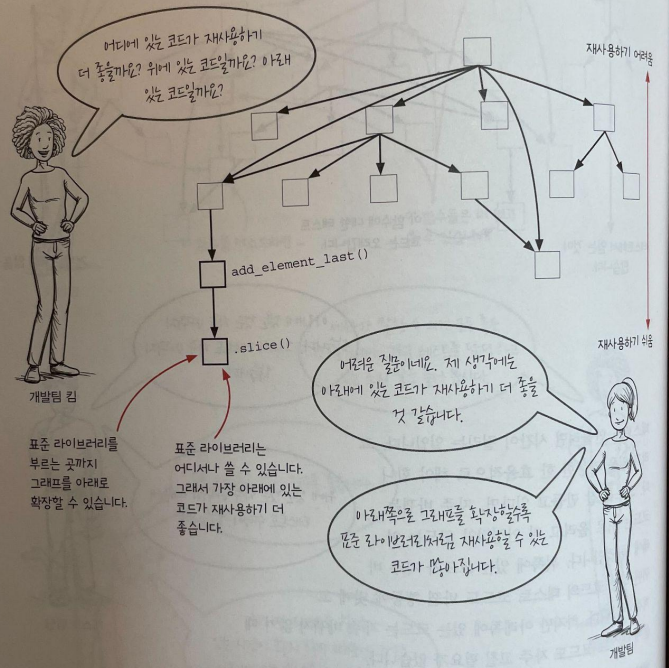
테스트도 만들려면 시간이 걸리는 일입니다. 그리고 일을 가능한 한 효율적으로 해야 합니다. 코드를 잘 만들고 있다면, 자주 바뀌는 코드는 위로 올리고 더 안정적인 코드는 아래에 둘 것입니다. 위쪽에 있는 코드가 자주 바뀌면 해당 코드의 테스트 코드도 바뀐 행동에 맞게 고쳐줘야 합니다. 하지만 아래쪽에 있는 코드는 자주 바뀌지 않기 때문에 테스트 코드도 자주 고칠 필요가 없습니다.

패턴을 사용하면 테스트 가능성에 맞춰 코드를 계층화할 수 있습니다. 하위 계층으로 코드를 추출하거나 상위 계층에 함수를 만드는 일은 테스트의 가치를 결정합니다.

하위 계층 코드를 테스트할수록
얻은 것이 더 오래갑니다.

아래에 있는 코드가 재사용하기 더 좋습니다

위에 있는 코드가 바꾸기 쉽고, 아래에 있는 코드를 테스트하는 것이 더 중요하다는 것을 알았습니다. 그럼 어떤 코드가 재사용하기 더 좋을까요? 코드를 재사용하면 코드를 다시 만들지 않아도 되기 때문에 여러 번 고치거나 테스트하지 않아도 됩니다. 코드를 재사용하면 시간과 비용을 줄일 수 있습니다.

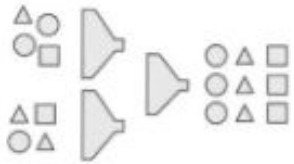


앞에서 계층형 구조를 만들면 자연스럽게 재사용성이 좋아지는 것을 봤습니다. 낮은 계층으로 함수를 추출하면 재사용할 가능성이 많아지는 것을 봤습니다. 낮은 계층은 재사용하기 더 좋습니다. 계층형 설계 패턴을 적용하면 재사용 가능한 계층으로 코드를 만들 수 있습니다.

아래쪽으로 가리키는 화살표가 많은 함수는 재사용하기 어렵습니다.

a program is...

pipeline of input->output



사실상

끝!

Understanding "effects"

Normal World

Wrapper World

What is an effect?

Could be anything really. It's vague!

- A generic type

  `List<_>`

- A type enhanced with extra data

  `Option<_>, Result<_>`

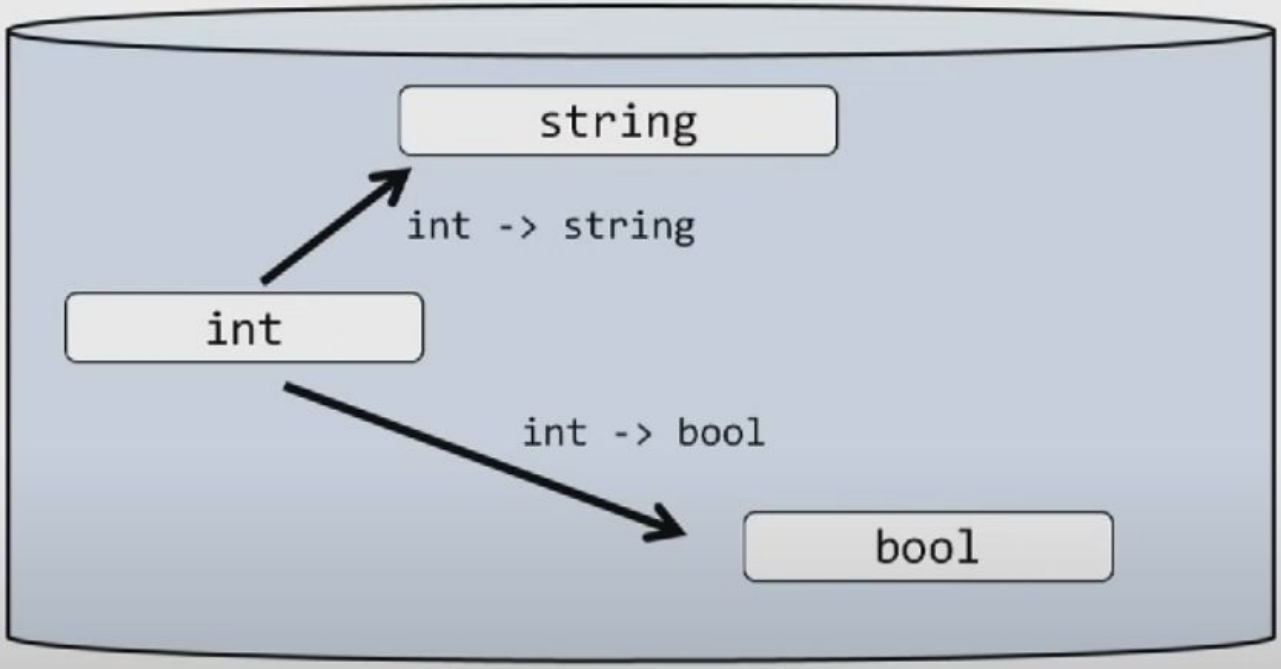
- A type that can change the outside world

  `Async<_>, Task<_>, Random<_>`

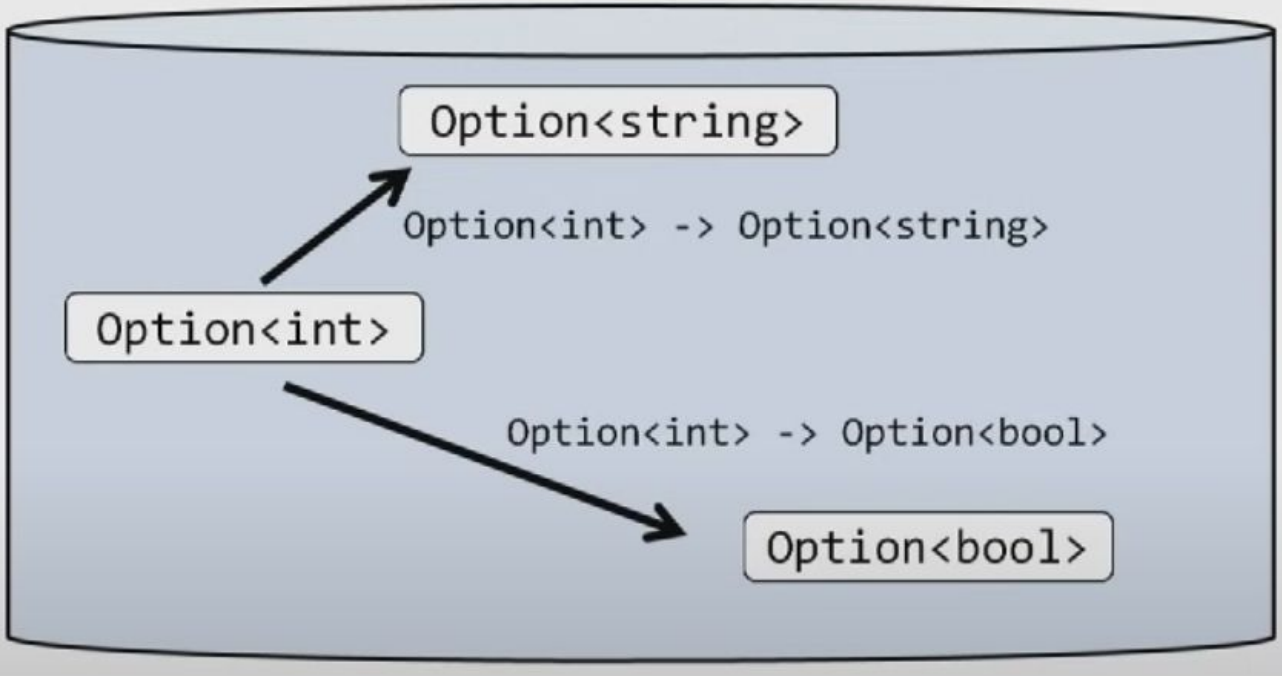
- A type that carries state

  `State<_>, Parser<_>`

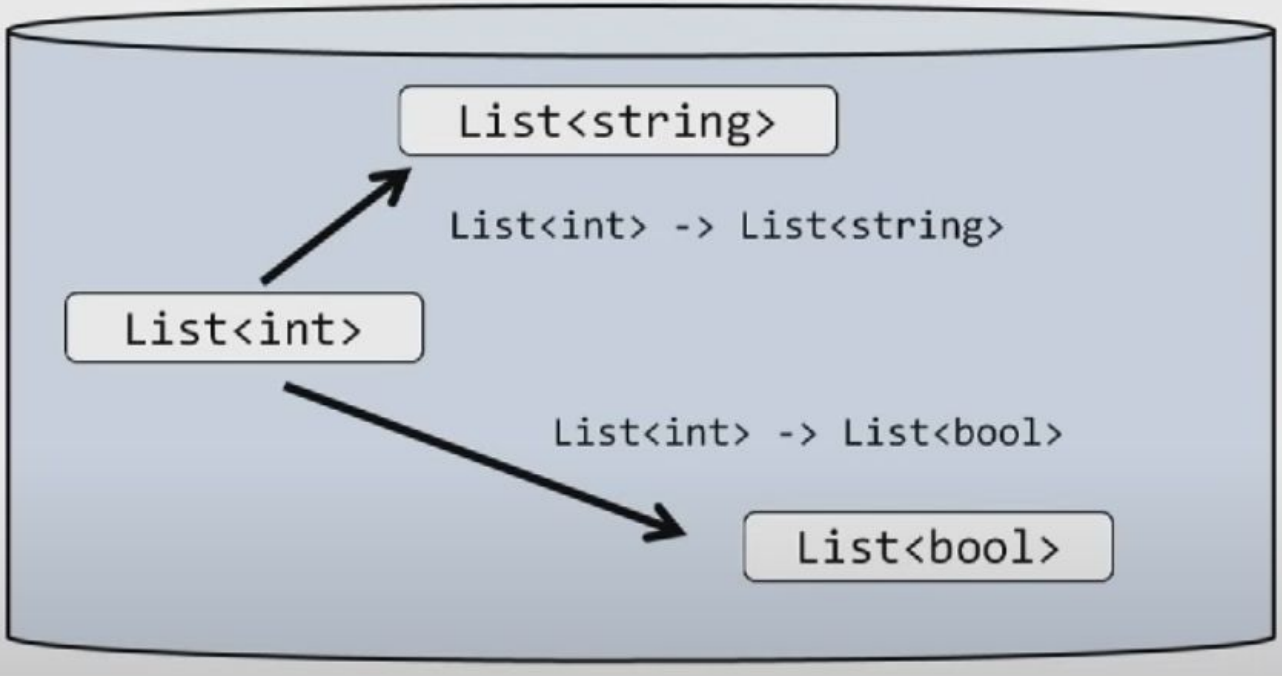
"Normal" world



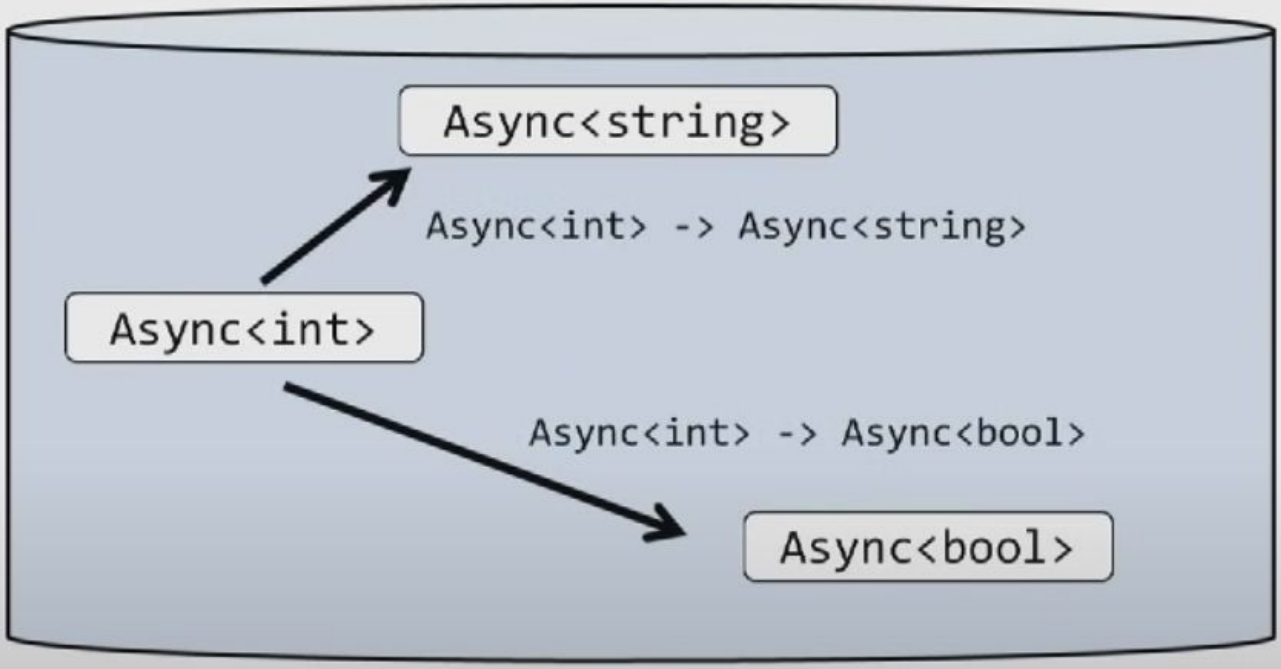
"Option" world



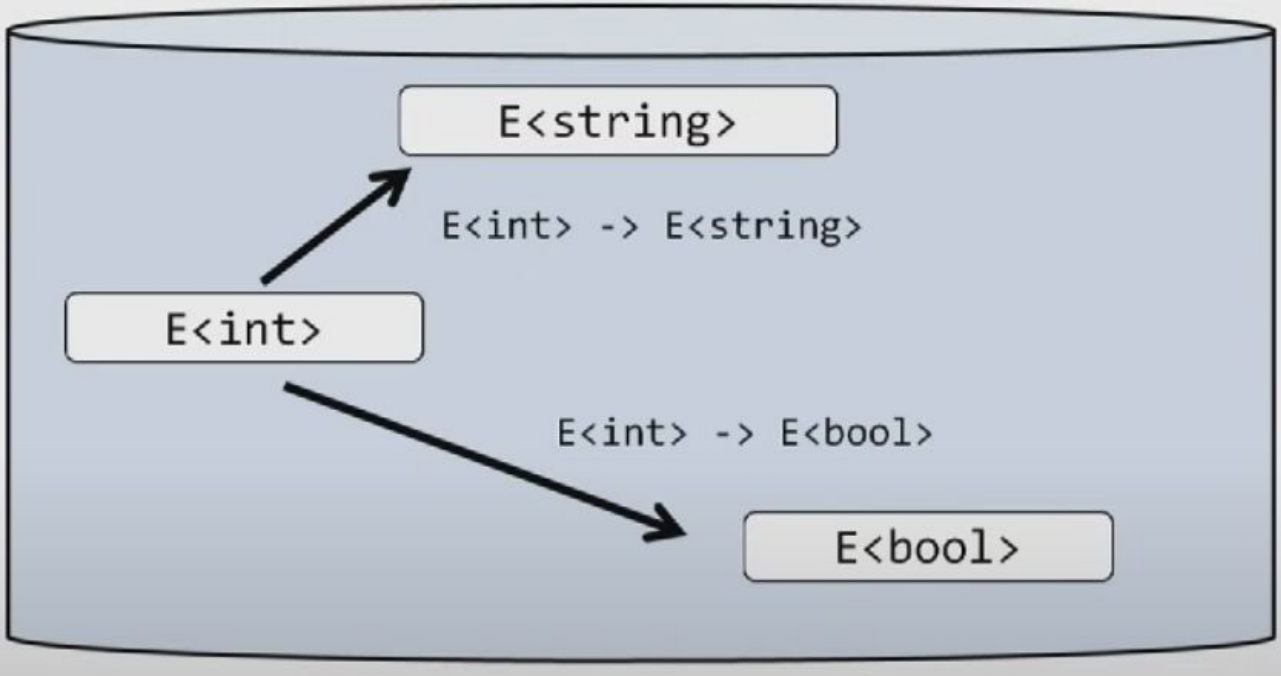
"List" world


List<string>
List<int> -> List<string>
List<int>
List<int> -> List<bool>
List<bool>

"Async" world

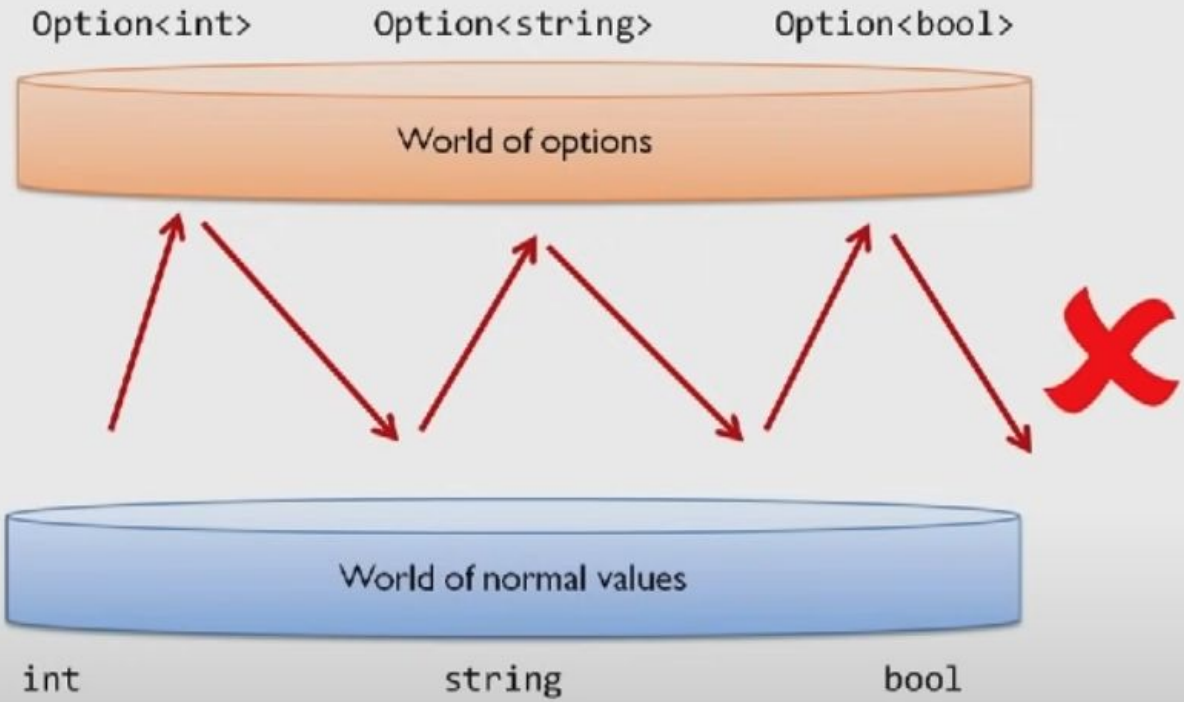

Async<string>
Async<int> -> Async<string>
Async<int>
Async<int> -> Async<bool>
Async<bool>

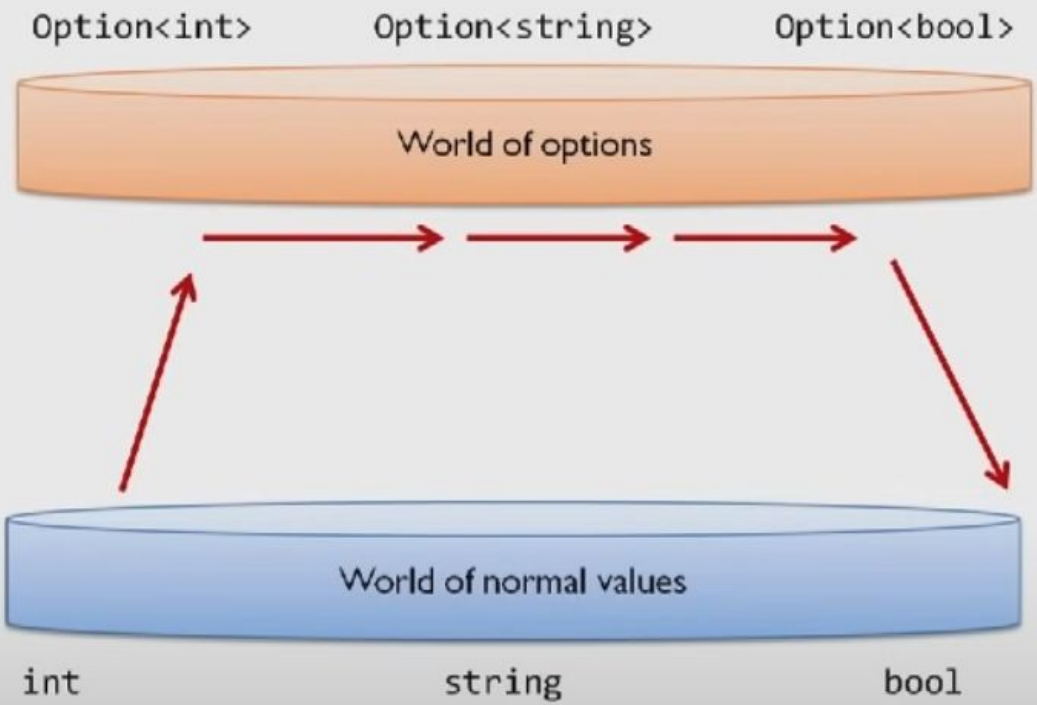
"Effects" world



Problem:

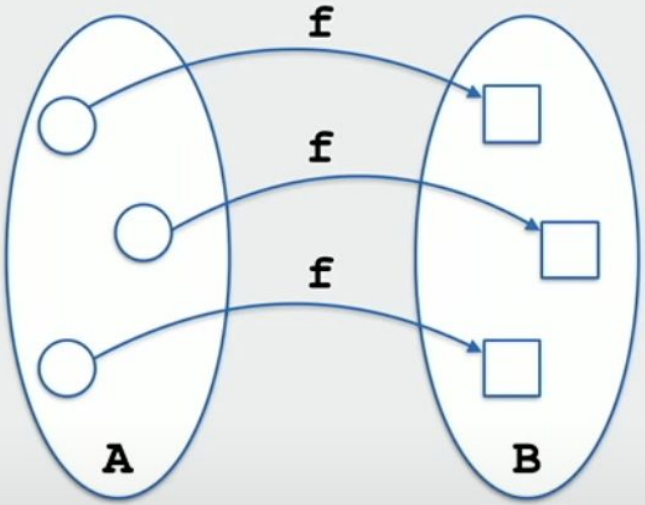
How to do stuff in an effects world?


Option<int>
Option<string>
Option<bool>
World of options
World of normal values
int
string
bool

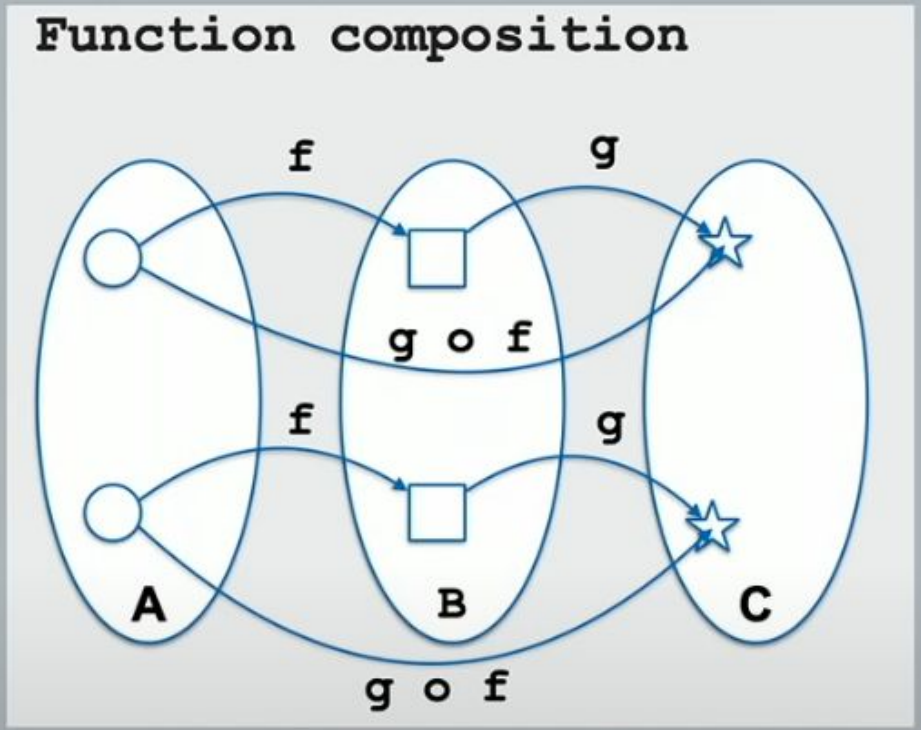

Option<int>
Option<string>
Option<bool>
World of options
World of normal values
int
string
bool

Notations - Function Types

- `f :: A -> B`

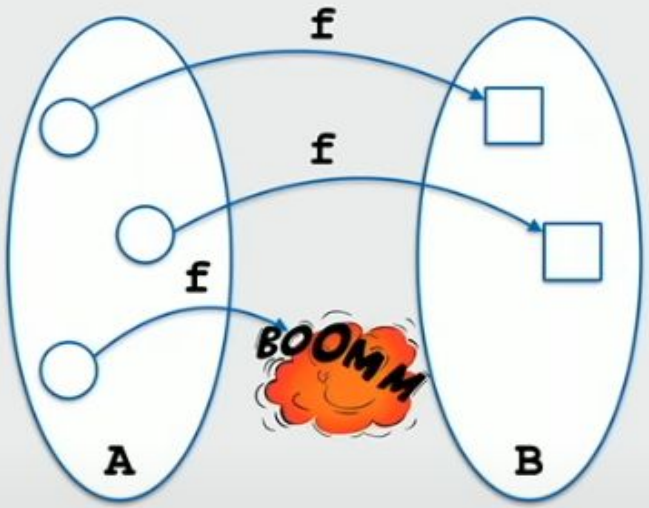


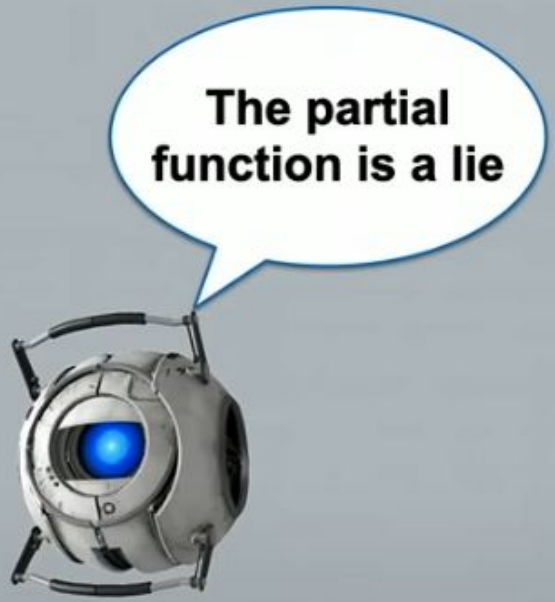
Notations - Function Composition



Pure Functional Programming

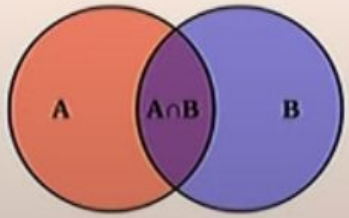
- exceptions = partial function





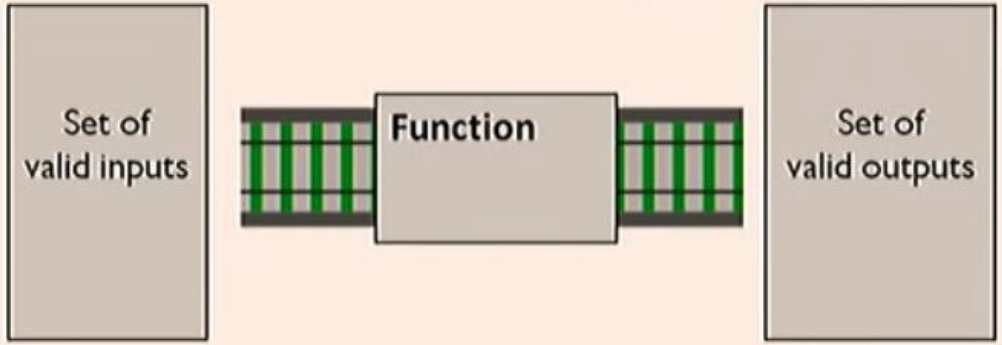
Core principle:

Types are not classes

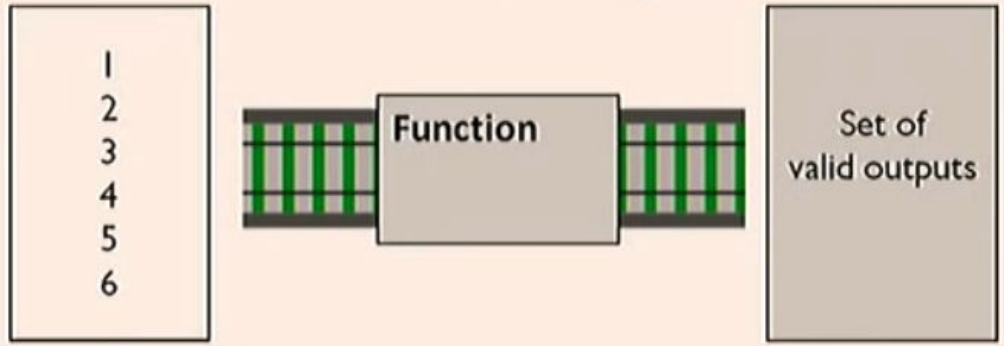


So, what is a type then?

A type is a just a name
for a set of things

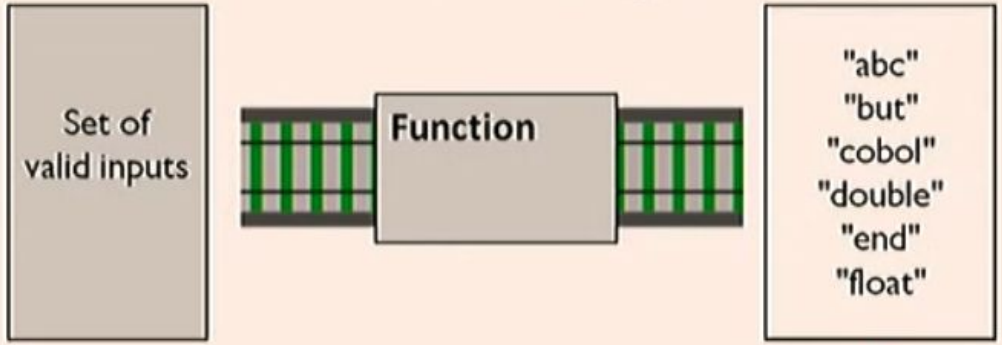


A type is a just a name
for a set of things



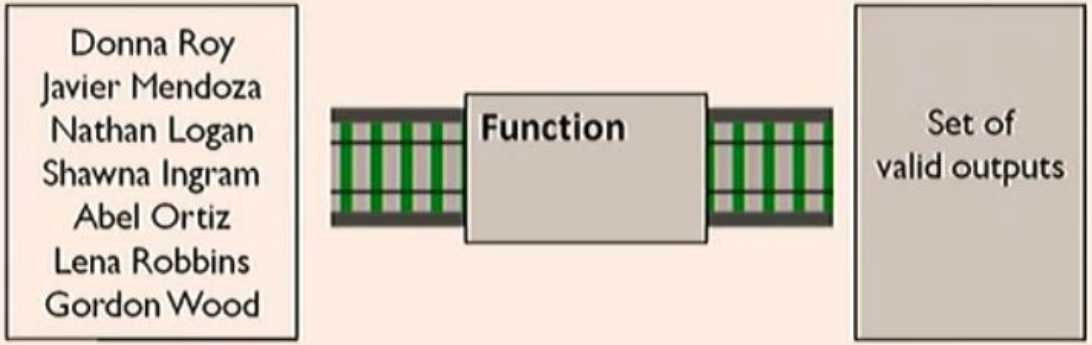
This is type
"integer"

A type is a just a name
for a set of things



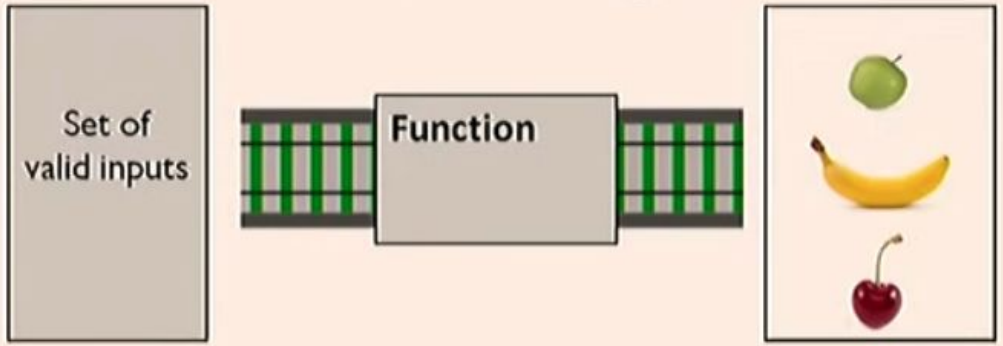
This is type
"string"

A type is a just a name
for a set of things



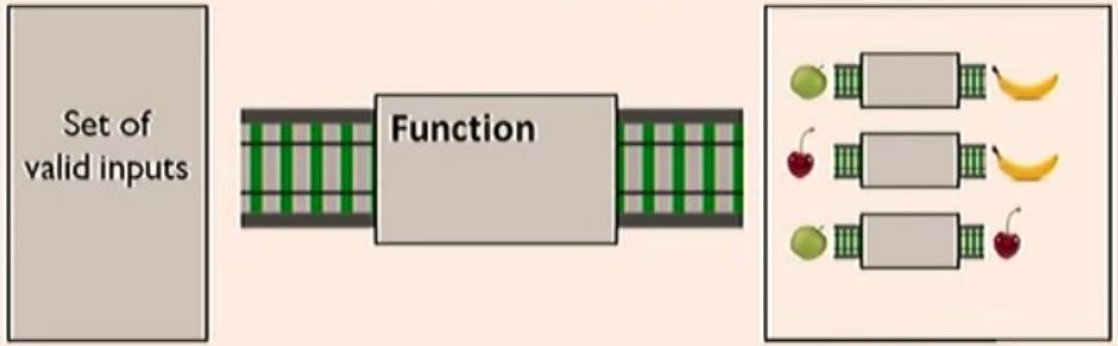
This is type
"Person"

A type is a just a name
for a set of things

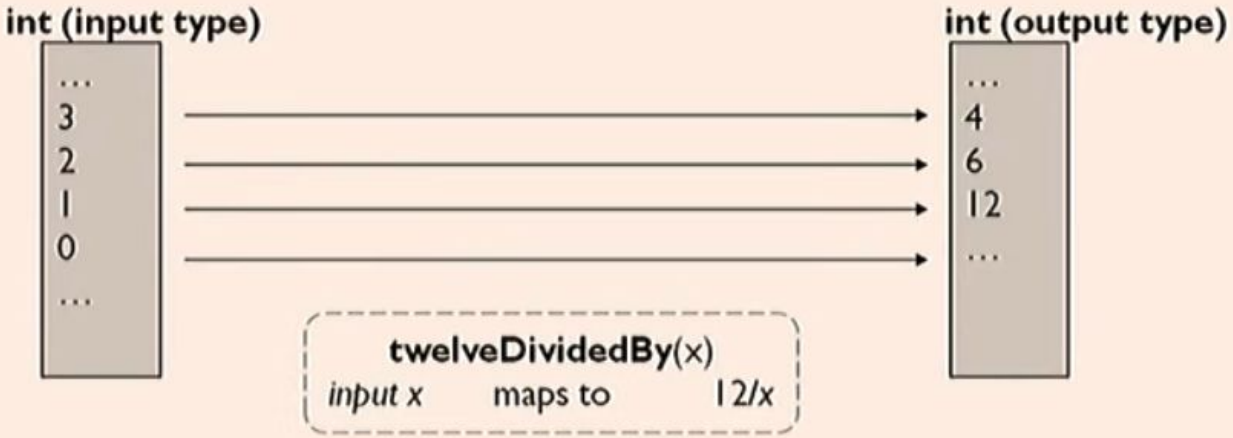


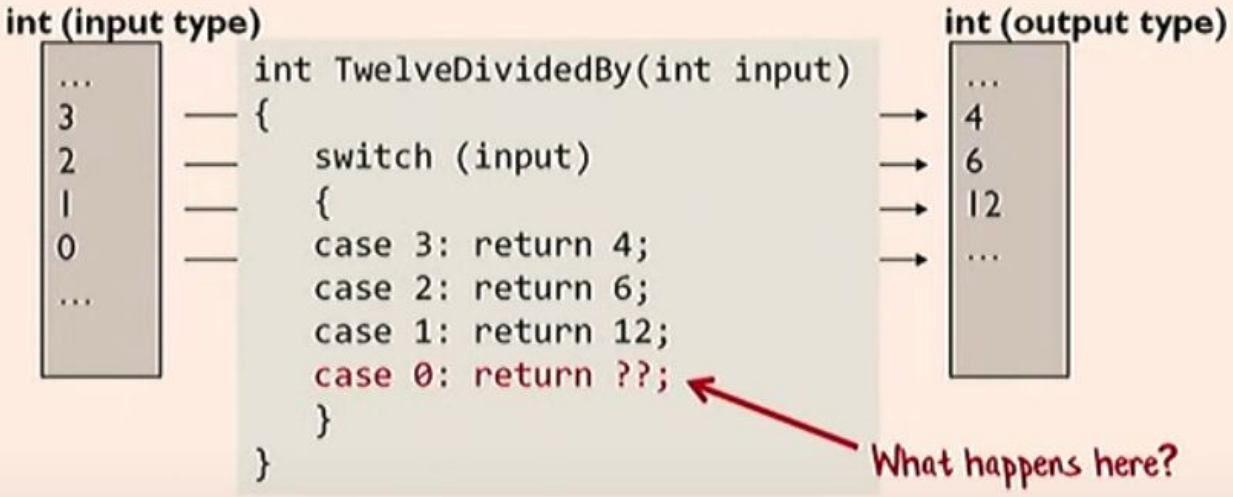
This is type
"Fruit"

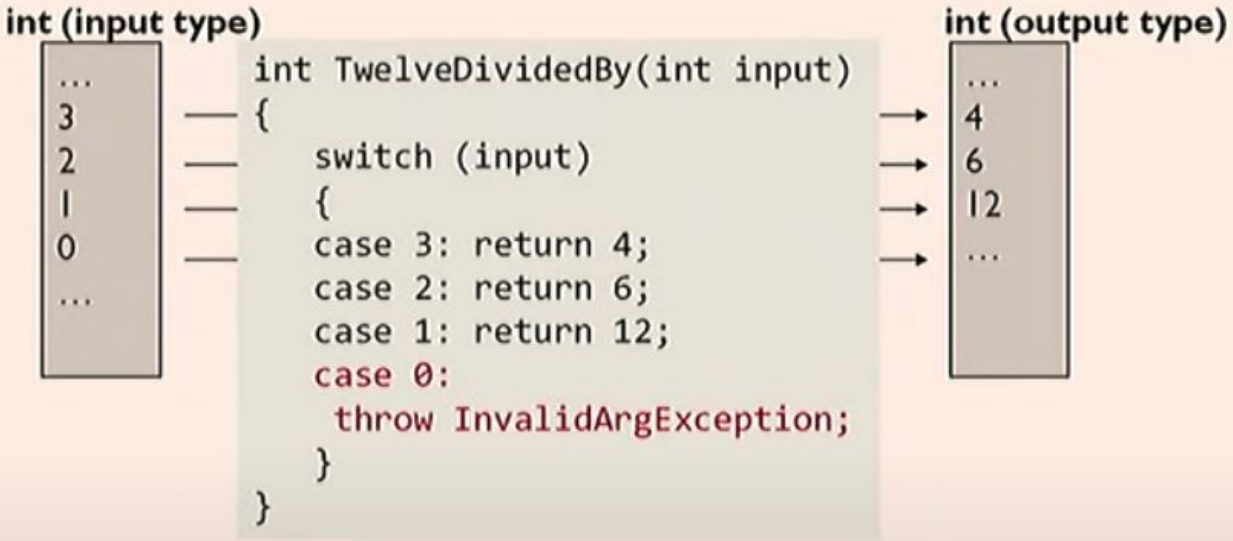
A type is a just a name
for a set of things



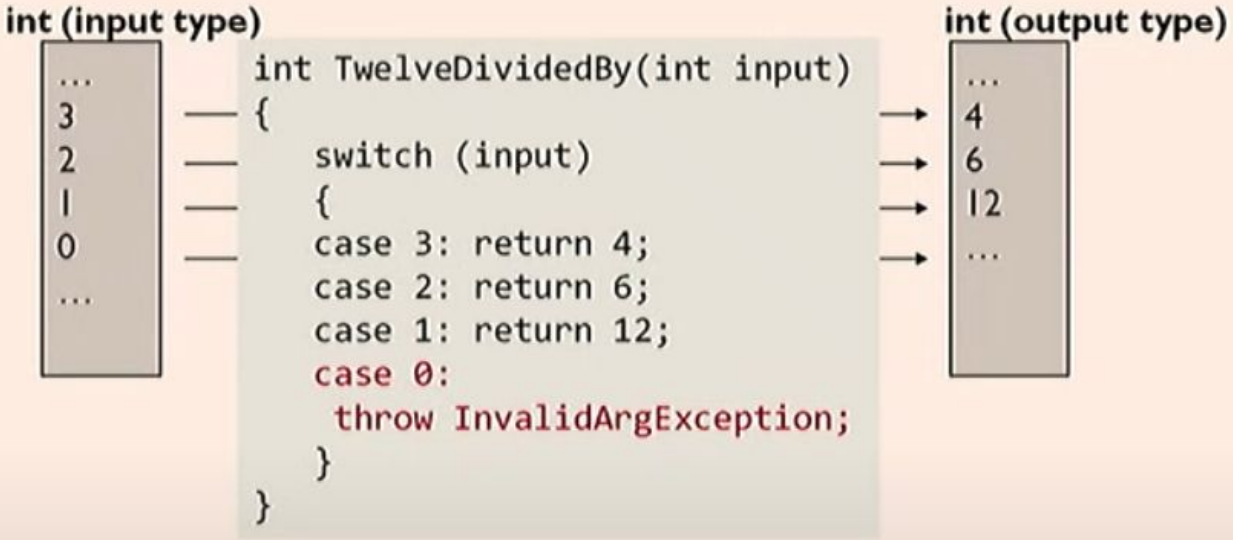
This is a type of
Fruit->Fruit functions


int (input type)
...
3
2
1
0
...
int (output type)
...
4
6
12
...
twelveDividedBy(x)
input x maps to 12/x


int (input type)
...
3
2
1
0
...
int TwelveDividedBy(int input)
{
switch (input)
{
case 3: return 4;
case 2: return 6;
case 1: return 12;
case 0: return ??;
}
}
int (output type)
...
4
6
12
...
What happens here?

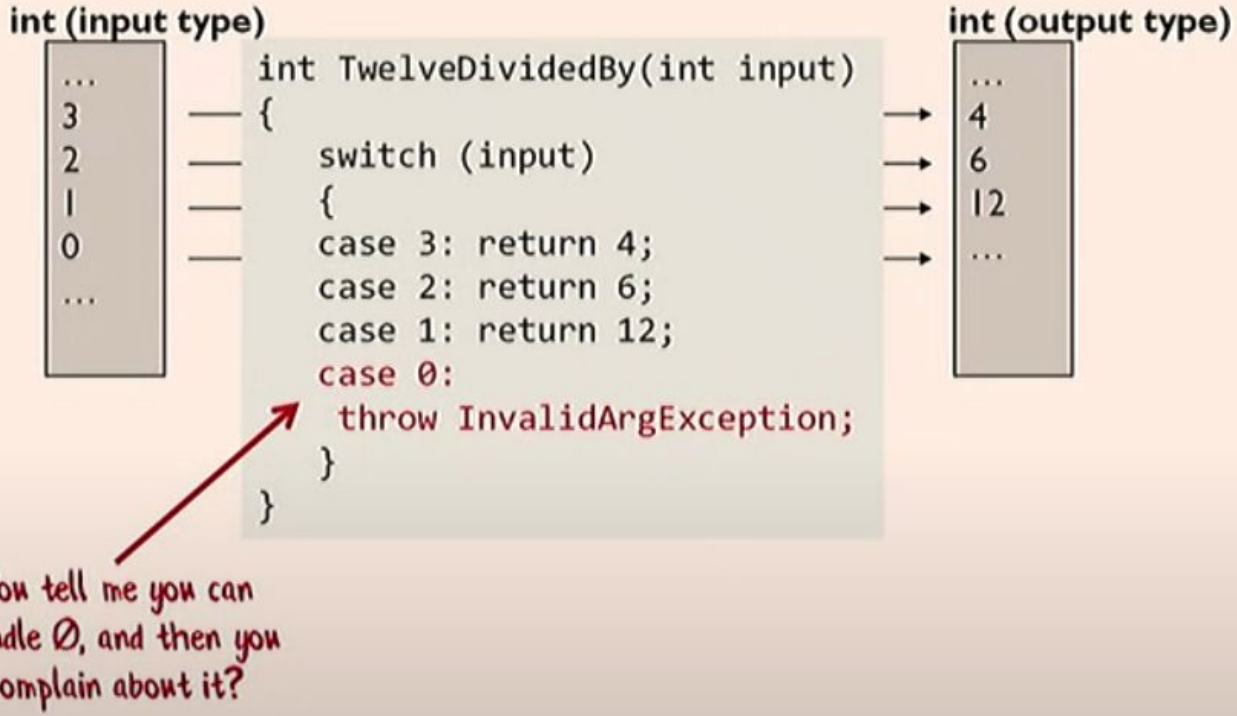

int (input type)
...
3
2
1
0
...
int TwelveDividedBy(int input)
{
switch (input)
{
case 3: return 4;
case 2: return 6;
case 1: return 12;
case 0:
throw InvalidArgException;
}
}
int (output type)
...
4
6
12
...

int -> int


int (input type)
...
3
2
1
0
...
int TwelveDividedBy(int input)
{
switch (input)
{
case 3: return 4;
case 2: return 6;
case 1: return 12;
case 0:
throw InvalidArgException;
}
}
int (output type)
...
4
6
12
...

This type signature is a lie!

```
int -> int
```

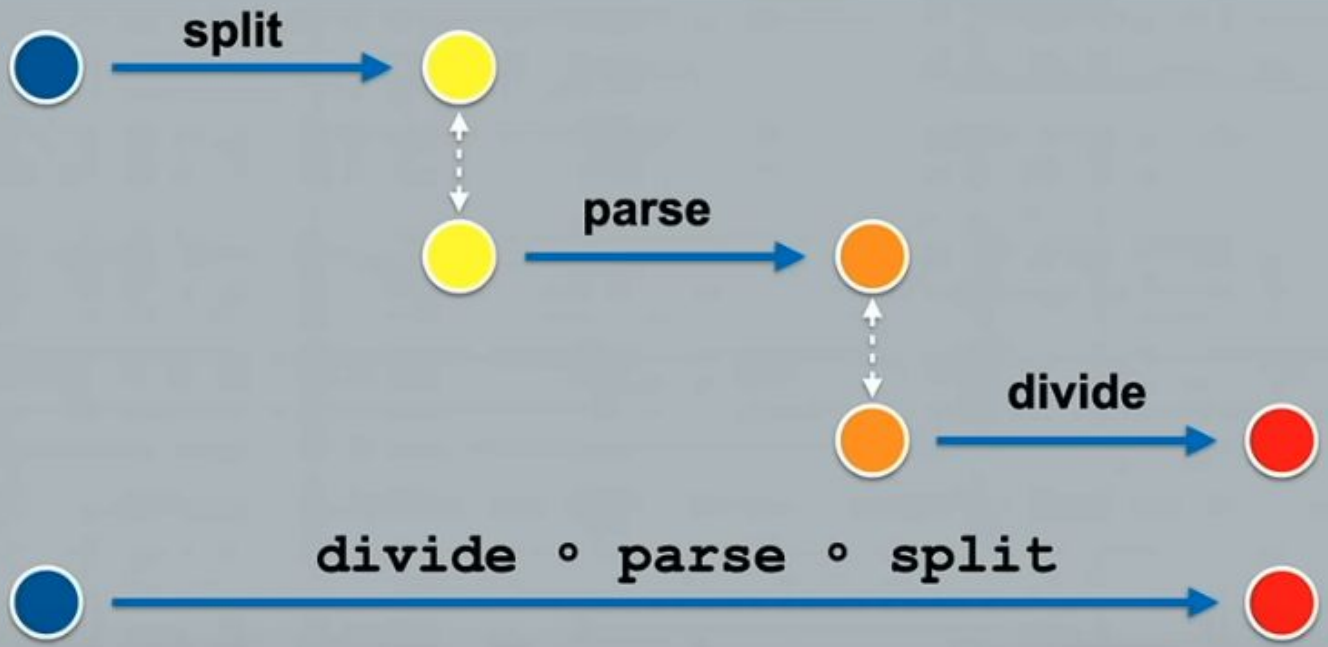


Use case

input="126,3", output:126/3 = 42

```
public (String, String) split(String s){
  return s.split(",")
}

public (double, double) parse((String, String) p){
  return (parseDouble(p._1), parseDouble(p._2))
}

public double divide ((double, double) numbers){
  return numbers._1 / numbers._2
}
```

Use case



Use case

Exception handling

Optional<T>	*Either<L, R>*
Optional.of("OK") *Optional.empty()*	*Either.left(new Exception("…"))* *Either.right("OK")*

Use case

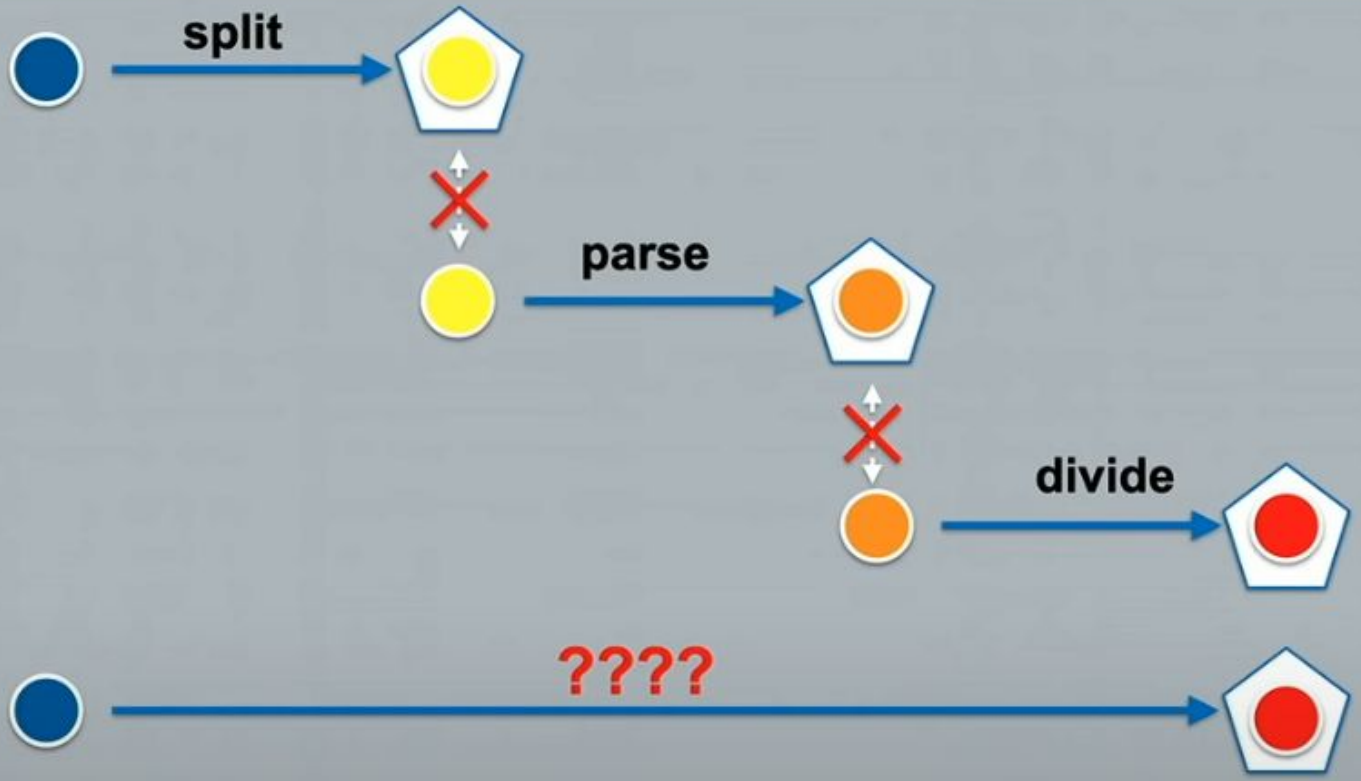
Let's try with Optional

```
public Optional<(String,String)> split(String s)

public Optional<(double, double)> parse((String, String) p)

public Optional<double> divide ((double, double) numbers)
```

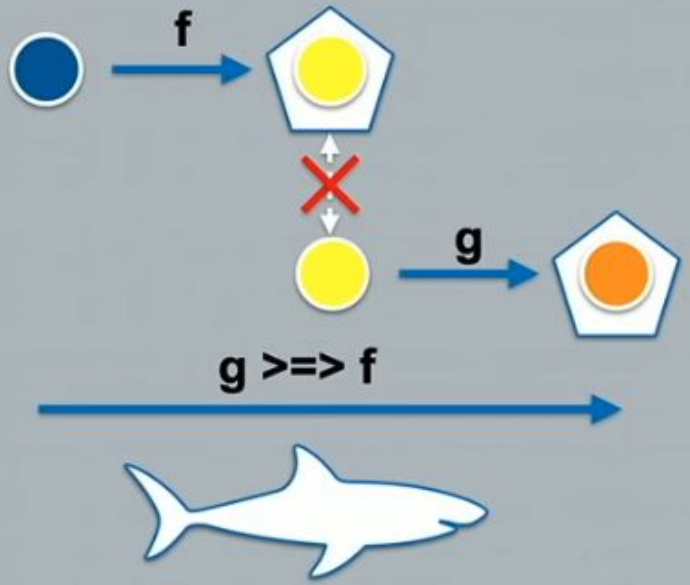
Use case

```
public Optional<Double> myProgram(String s){
  final Optional<(String,String)> split = split(s);
  if(split.isPresent()){
    final Optional<(double, double)> parse = parse(split.get());
    if(parse.isPresent()){
      final Optional<Double> result = divide(parse.get());
      if(result.isPresent()){
        return result.get();
      }
    }
  }
  return Optional.empty();
}
```

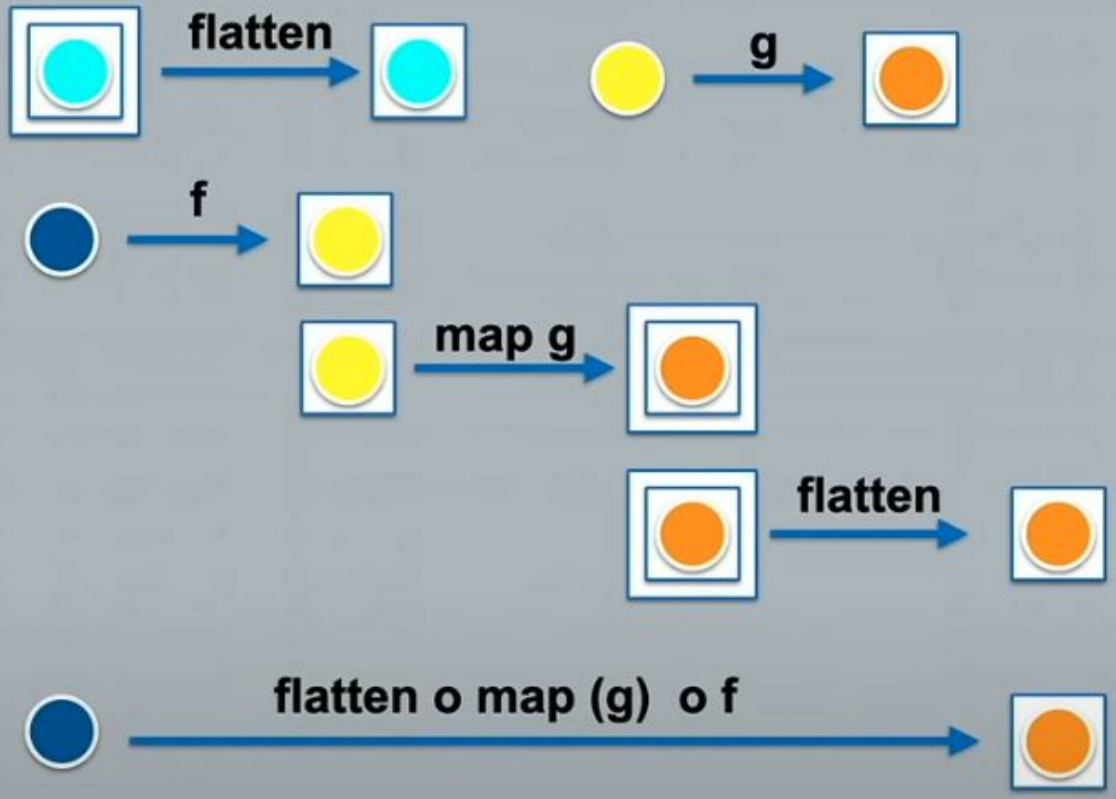
Use case



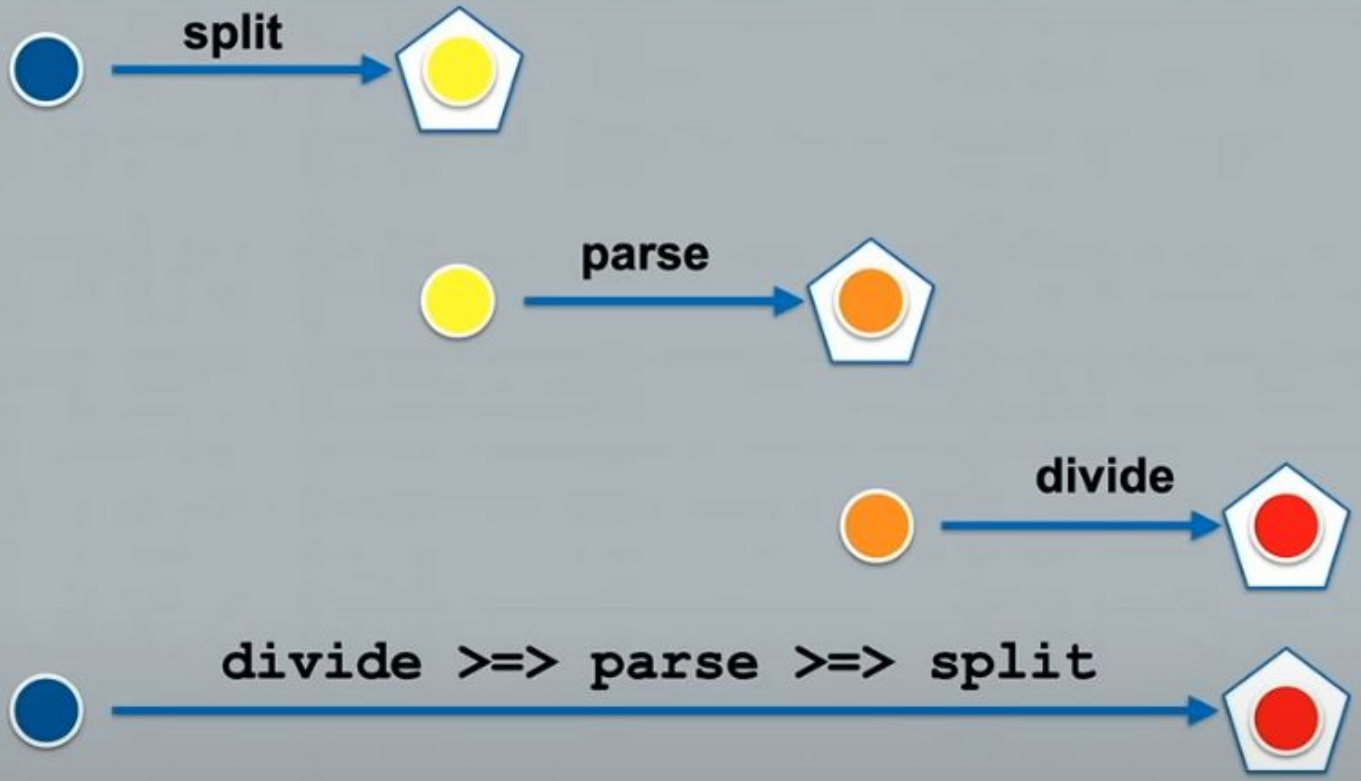
Use case



Monad



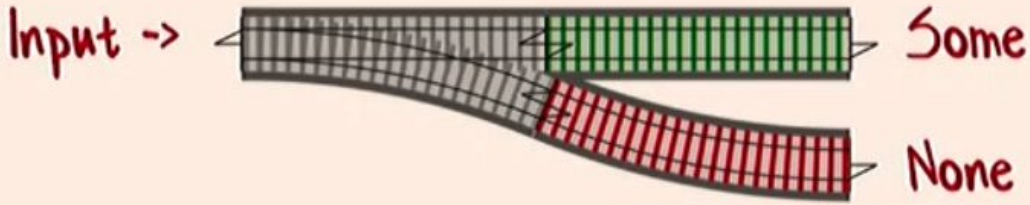
Monad



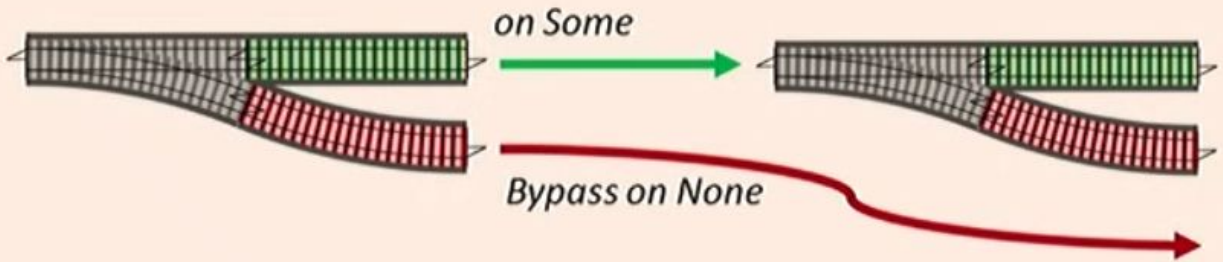
Monad

```
public Optional<Double> myProgram(String s) {
   return split(s)
      .flatMap(split -> parse(split))
      .flatMap(parse -> divide(parse));
}
```

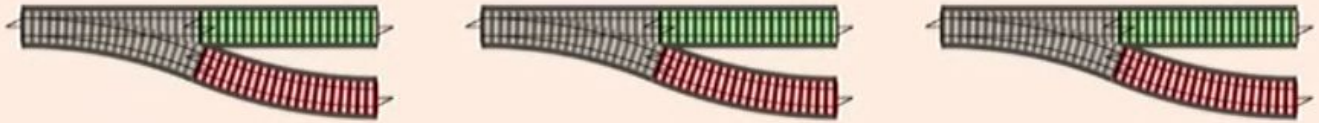
A switch analogy



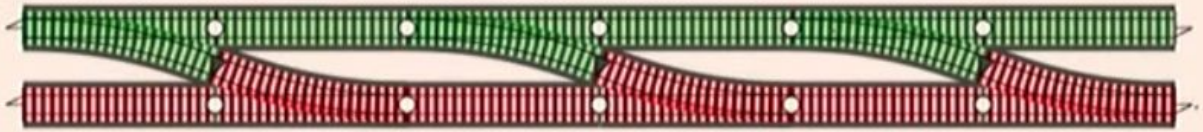
Connecting switches



Connecting switches



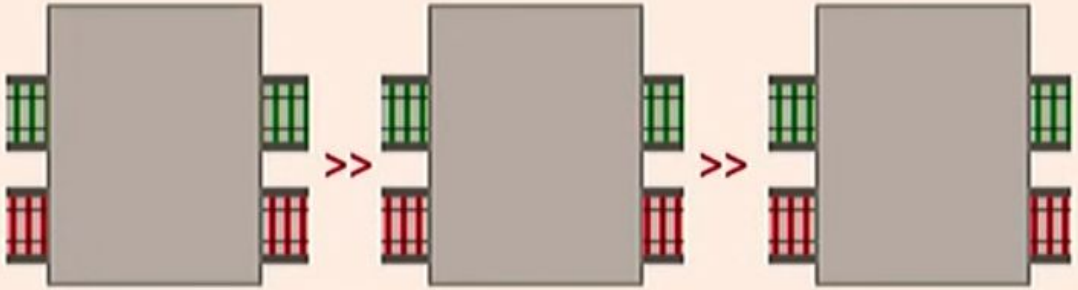
Connecting switches



Composing switches

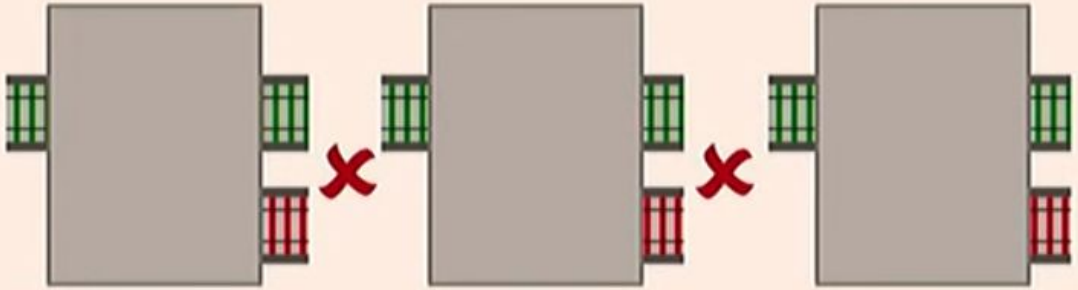
Composing one-track functions is fine...

Composing switches



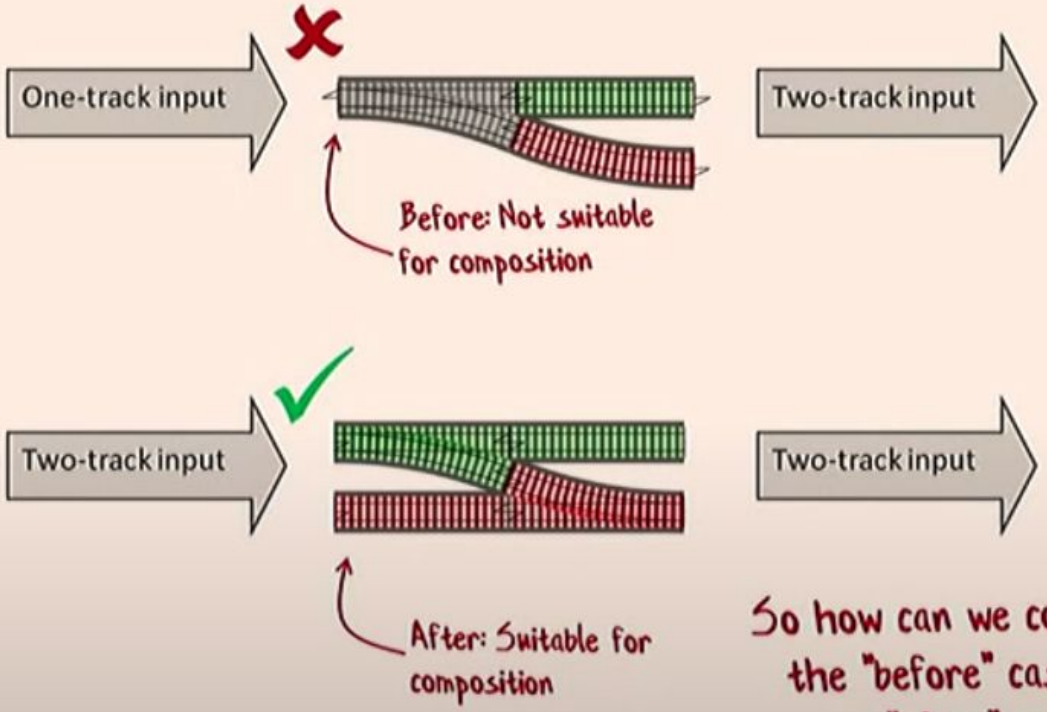
... and composing two-track functions is fine...

Composing switches



... but composing switches is not allowed!

Composing switches



"Bind" is the answer!
Bind all the things!

FP'ers get excited by bind

Pyramid of doom: using bind

```
let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

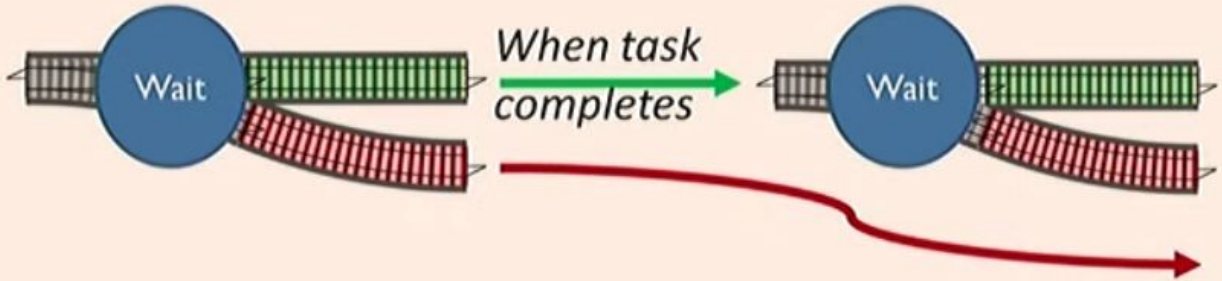
```
let example input =
    let x = doSomething input
    if x.IsSome then
        let y = doSomethingElse (x.Value)
        if y.IsSome then
            let z = doAThirdThing (y.Value)
            if z.IsSome then
                let result = z.Value
                Some result
            else
                None
        else
            None
    else
        None
```

Pyramid of doom: using bind

```
Let bind f opt =
    match opt with
    | Some v -> f v
    | None -> None
```

```
let example input =
    doSomething input
    |> bind doSomethingElse
    |> bind doAThirdThing
    |> bind ...
```

Connecting tasks



Pyramid of doom: using bind for tasks

```
let taskBind f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)
```

a.k.a "promise" "future"

```
let taskExample input =
    let taskX = startTask input
    taskX.WhenFinished (fun x ->
        let taskY = startAnotherTask x
        taskY.WhenFinished (fun y ->
            let taskZ = startThirdTask y
            taskZ.WhenFinished (fun z ->
                z  // final result
```

Pyramid of doom: using bind for tasks

```
let taskBind f task =
    task.WhenFinished (fun taskResult ->
        f taskResult)
```

```
let taskExample input =
    startTask input
    |> taskBind startAnotherTask
    |> taskBind startThirdTask
    |> taskBind ...
```

Use case without error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  validateRequest(request);
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Use case with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  db.updateDbFromRequest(request);
  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Use case with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  var result = db.updateDbFromRequest(request);
  if (!result) {
    return "Customer record not found"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Use case with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  smtpServer.sendEmail(request.Email)

  return "OK";
}
```

Use case with error handling

```
string UpdateCustomerWithErrorHandling()
{
  var request = receiveRequest();
  var isValidated = validateRequest(request);
  if (!isValidated) {
     return "Request is not valid"
  }
  canonicalizeEmail(request);
  try {
    var result = db.updateDbFromRequest(request);
    if (!result) {
      return "Customer record not found"
    }
  } catch {
    return "DB error: Customer record not updated"
  }

  if (!smtpServer.sendEmail(request.Email)) {
    log.Error "Customer email not sent"
  }

  return "OK";
}
```

A structure for managing errors


```
let validateInput input =
   if input.name = "" then
      Failure "Name must not be blank"
   else if input.email = "" then
      Failure "Email must not be blank"
   else
      Success input  // happy path
```

Connecting switches

Connecting switches

This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

예제코드로 파악해보기

함수의 순수성

액션
계산
데이터

불변(persistent)
자료구조

변수의 불변성

다음 발표 주제

- 객체지향 프로그래밍 다시 생각해보기
- lodash 라이브러리 사용해보기 (유용한 유틸리티 함수 소개하기)
- 공변성(covariance)과 반공변성(contravariance)
- 애자일 시작해보기 (개인적인 궁금증: 스크럼과 일일회의는 어떻게 다른걸까?)