



**IMPACT**  
**ON SOCIETY**



Jeroen Steenbeeke



Improve your code  
with functional programming concepts

# Jeroen Steenbeeke



2008 - 2016 : Topicus Education  
2016 - September 2018 : Sqills  
October 2018 - Present : Topicus Education



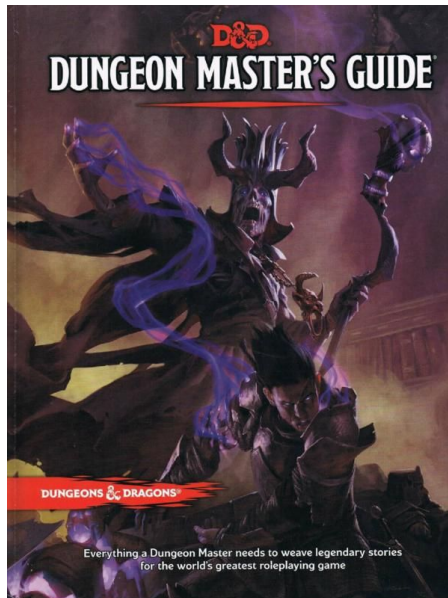
# Jeroen Steenbeeke



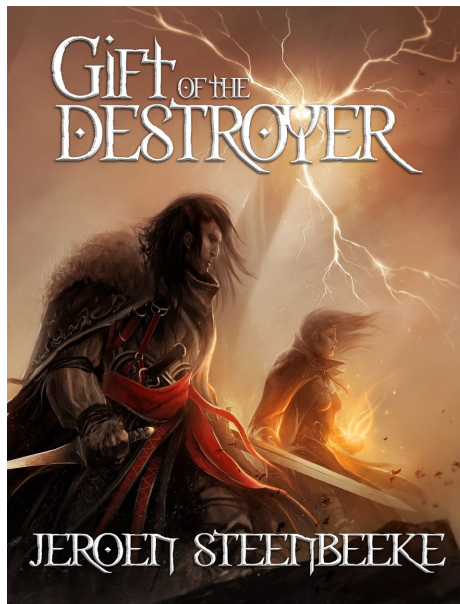
# Jeroen Steenbeeke



# Jeroen Steenbeeke



# Jeroen Steenbeeke







**IMPACT**  
**ON SOCIETY**



Jeroen Steenbeeke



Improve your code with FP concepts

# Functional Programming

In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.





# In this presentation

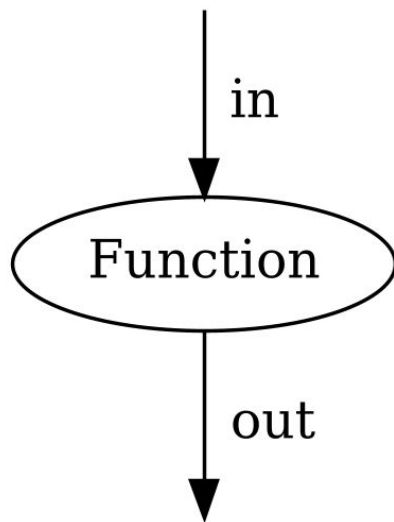
- Lots of pictures & diagrams
  - Marbles
  - Boxes
  - Conveyor belts
- Code examples
  - Java
  - Vavr (library)

# Not in this presentation

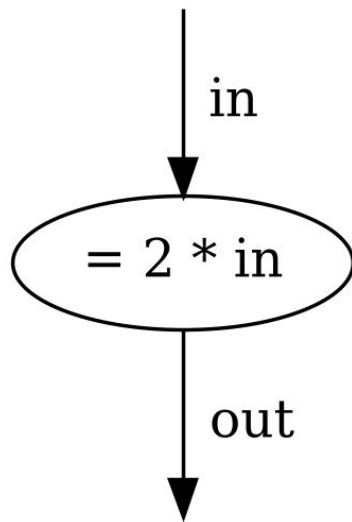
- Pure functions
- Higher order functions
- Immutability
- Mathematics

# Functions

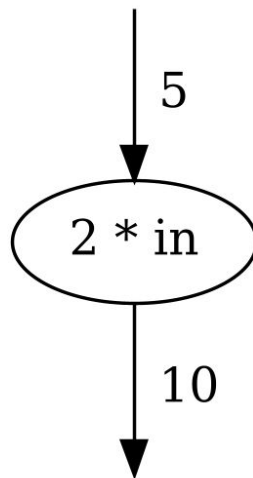
# Functions



# Functions

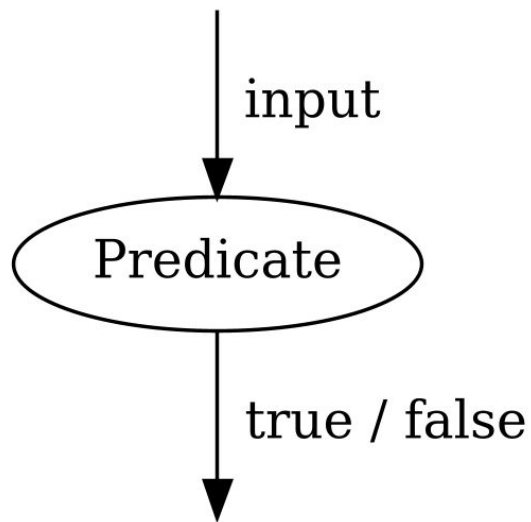


# Functions

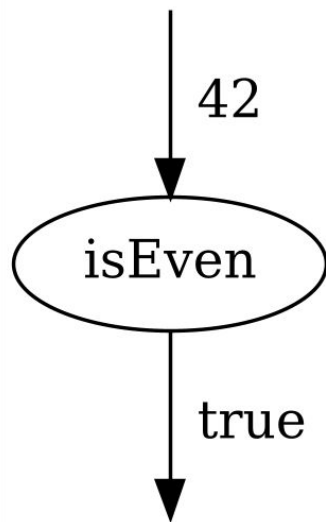




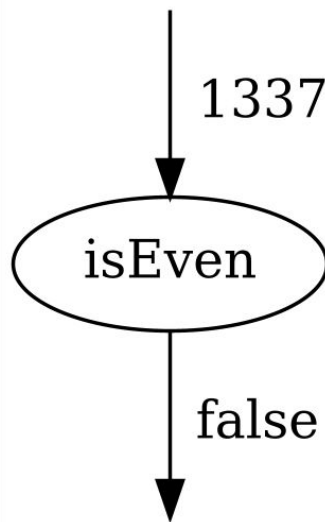
# Predicates



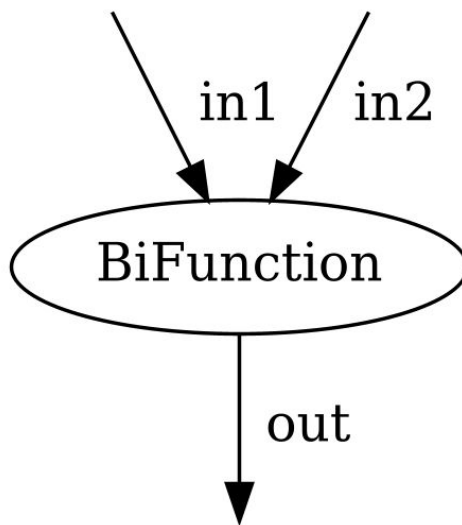
# Predicates



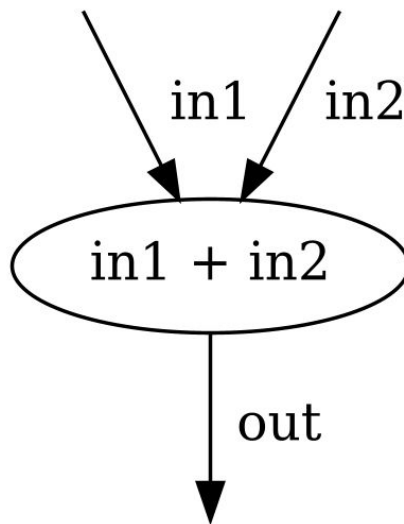
# Predicates



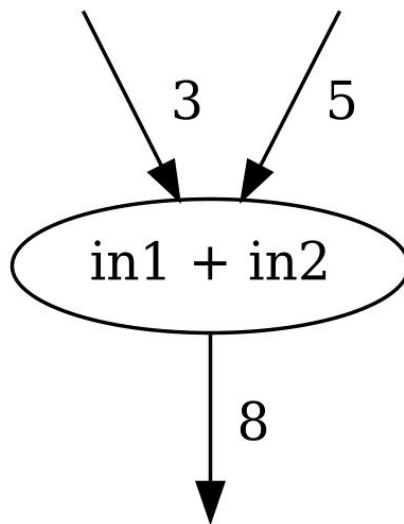
# Functions with multiple inputs



# Functions with multiple inputs

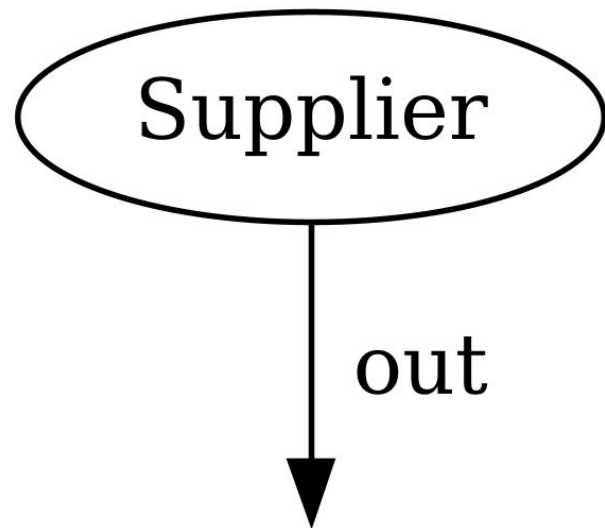


# Functions with multiple inputs

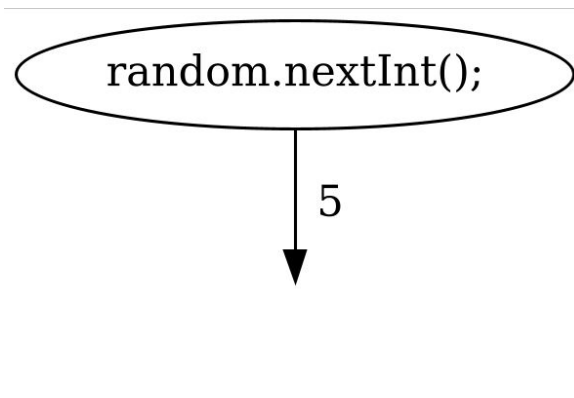




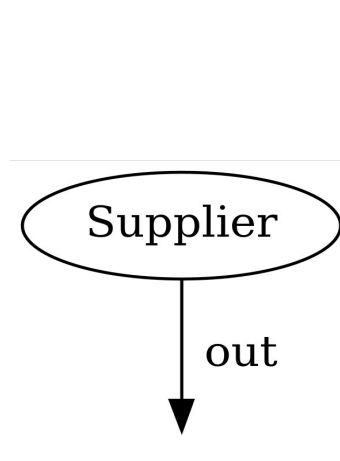
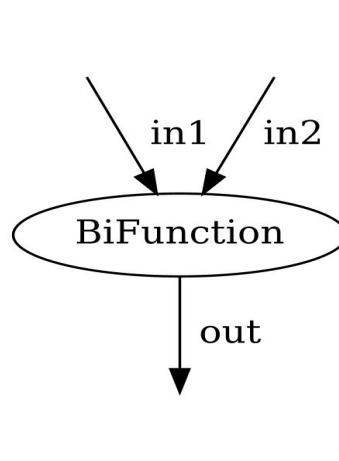
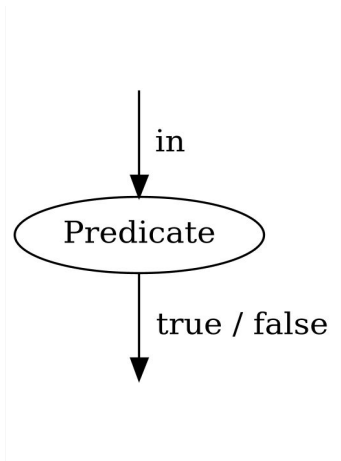
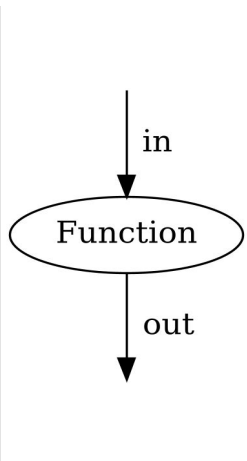
# Suppliers



# Suppliers



# Functions



# Functions

```
Function<String, Integer> integerToString =  
    s -> Integer.parseInt(s);
```

```
BiFunction<String, String, String> concatenate =  
    (a, b) -> a + b;
```

```
Predicate<String> isBlankString =  
    s -> s.isBlank();
```

```
Supplier<Integer> randomInteger =  
    () -> random.nextInt();
```

# Functions

```
Function<String, Integer> integerToString =  
    Integer::parseInt;
```

```
BiFunction<String, String, String> concatenate =  
    String::concat;
```

```
Predicate<String> isBlankString =  
    String::isBlank;
```

```
Supplier<Integer> randomInteger =  
    random::nextInt;
```

# Monads



# Monads

In functional programming, a **monad** is a software design pattern with a structure that combines program fragments (functions) and wraps their return values in a type with additional computation. In addition to defining a wrapping **monadic type**, monads define two operators: one to wrap a value in the monad type, and another to compose together functions that output values of the monad type (these are known as **monadic functions**). General-purpose languages use monads to reduce boilerplate code needed for common operations (such as dealing with undefined values or fallible functions, or encapsulating bookkeeping code). Functional languages use monads to turn complicated sequences of functions into succinct pipelines that abstract away control flow, and side-effects.



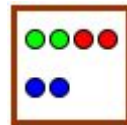
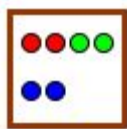
# Mowhatnow?



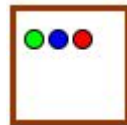
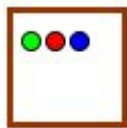
# Building blocks



# Lists



# Sets




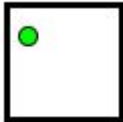
# Streams





# Options

Also known as: Optional, Maybe

Empty	Value
	
<b>Java:</b> <code>Optional.empty();</code>	<b>Java:</b> <code>Optional.of(value);</code>
<b>Vavr:</b> <code>Option.none();</code>	<b>Vavr:</b> <code>Option.some(value);</code>

# Options



Also known as: Optional, Maybe

```
Option<Integer> four =  
    Option.some( value: 4);
```

```
Option<Integer> none =  
    Option.none();
```

```
Option<Integer> notSure =  
    Option.of(methodThatMayReturnNull());
```

# Either


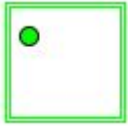
Left	Right
	
<b>Vavr:</b> <code>Either.left(value);</code>	<b>Vavr:</b> <code>Either.right(value);</code>
Left and Right can have different types!	

# Either

```
Either<String,Long> right =  
    Either.right(5L);
```

```
Either<String,Long> left =  
    Either.left("Could not determine number");
```

# Try

Failure	Success
	
<b>Vavr:</b> <code>Try.failure(new RuntimeException());</code>	<b>Vavr:</b> <code>Try.success(value);</code>

# Try

```
Try<Integer> success =  
    Try.success( value: 5);
```

```
Try<Integer> failure =  
    Try.failure(new RuntimeException());
```

```
Try<Integer> potentialFailure =  
    Try.of(() -> methodThatMayThrowAnException());
```

# Operations



# Null-checks

```
@Nullable
public String getPersonName(@Nullable Person person) {
    if (person != null) {
        return person.name();
    }

    return null;
}
```



# Replace nullable objects with Options

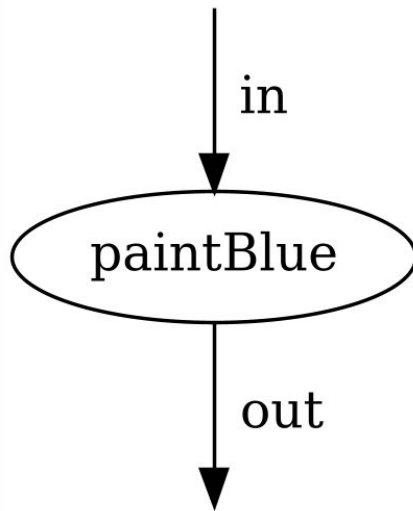
```
@NotNull
public Option<String> getPersonName(@NotNull Option<Person> person) {
    if (person.isDefined()) {
        return Option.some(person.get().name());
    }

    return Option.none();
}
```

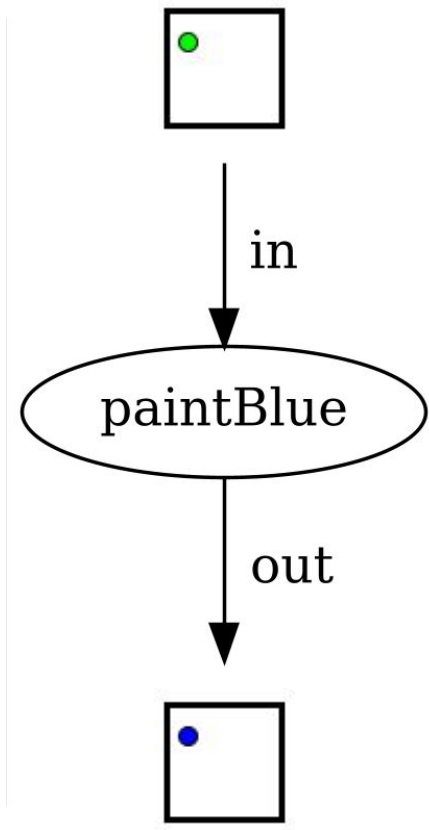
# Map

- Take the contents of your building block
- Apply function to contents
- Return a new building block with the modified contents

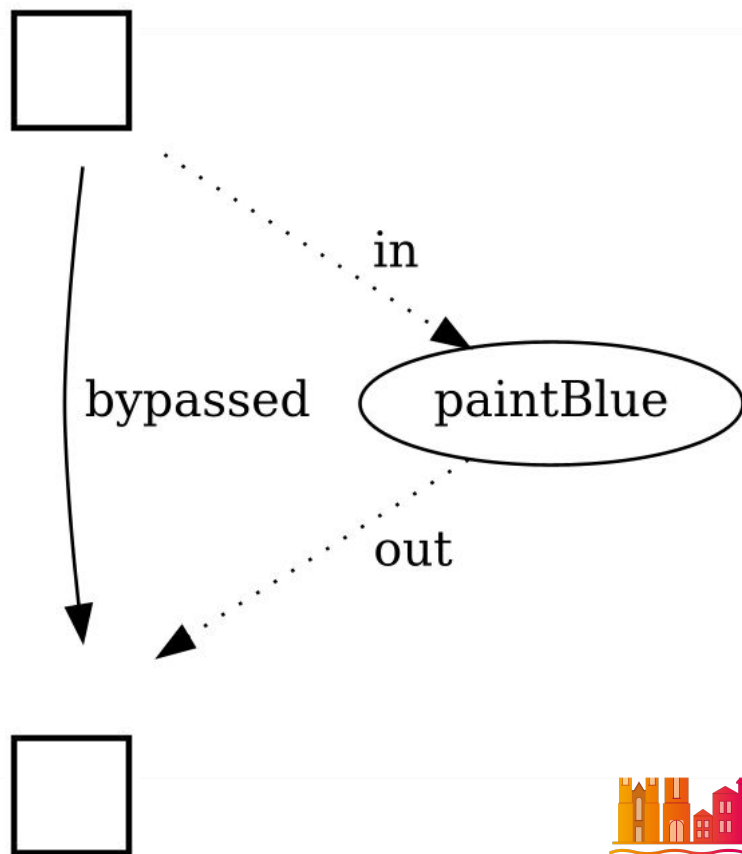
# Map



## Map - Option



## Map - Option



# Replace nullable objects with Options

```
@NotNull
public Option<String> getPersonName(@NotNull Option<Person> person) {
    if (person.isDefined()) {
        return Option.some(person.get().name());
    }

    return Option.none();
}
```

# Replace nullable objects with Options

```
@NotNull
public Option<String> getPersonNameWithMap(@NotNull Option<Person> person) {
    return person.map(Person::name);
}
```

# Replace nullable objects with Options

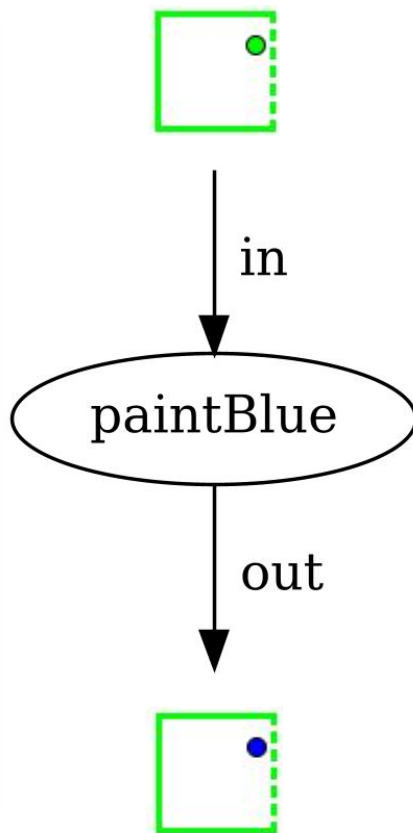
```
Person steve = new Person( name: "Steve", address: null);
```

```
System.out.println(Option.some(steve)
    .map(Person::name)); // Some(Steve)
```

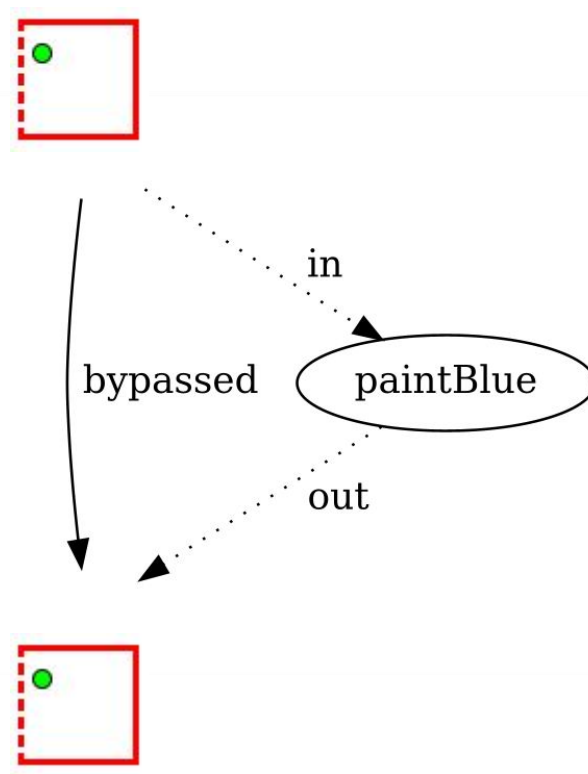
```
System.out.println(Option.<Person> none()
    .map(Person::name)); // None
```



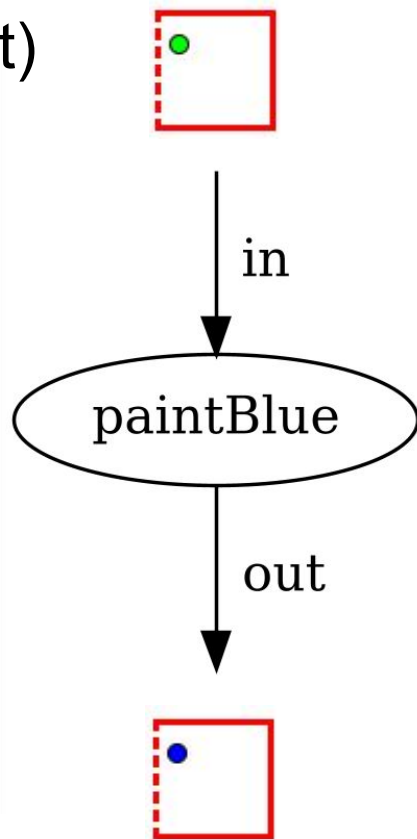
## Map - Either (right)



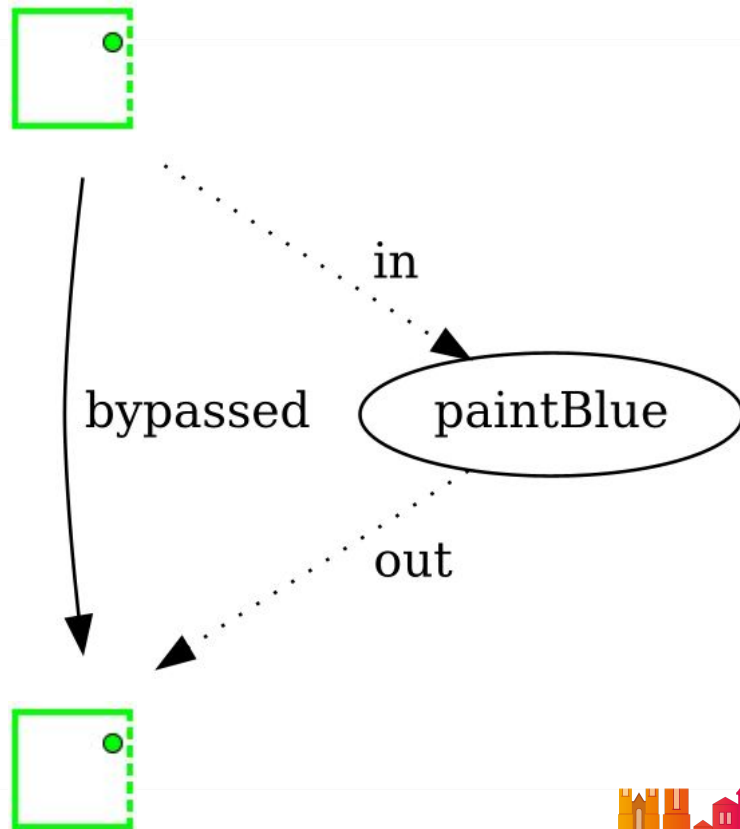
## Map - Either (left)



## MapLeft - Either (left)



## MapLeft - Either (right)



# Map - Either

```
@GET
@Path("/max-mortgage")
public Response calculateMaxMortgage()
{
    Either<String, BigDecimal> maxMortgage = businessLogic.calculateMaxMortgage();

    if (maxMortgage.isRight())
    {
        return Response.ok(maxMortgage.get()).build();
    }
    else
    {
        return Response.serverError().entity(maxMortgage.getLeft()).build();
    }
}
```

# Map - Either

```
@GET
@Path("/max-mortgage")
public Response calculateMaxMortgage()
{
    Either<String, Response> maxMortgage = businessLogic.calculateMaxMortgage()
        .map(max -> Response.ok().build());

    if (maxMortgage.isRight())
    {
        return maxMortgage.get();
    }
    else
    {
        return Response.serverError().entity(maxMortgage.getLeft()).build();
    }
}
```

# Map - Either

```
@GET
@Path("/max-mortgage")
public Response calculateMaxMortgage()
{
    Either<Response, Response> maxMortgage = businessLogic.calculateMaxMortgage() Either<String, BigDecimal>
        .map(max -> Response.ok().build()) Either<String, Response>
        .mapLeft(error -> Response.serverError().entity(error).build());

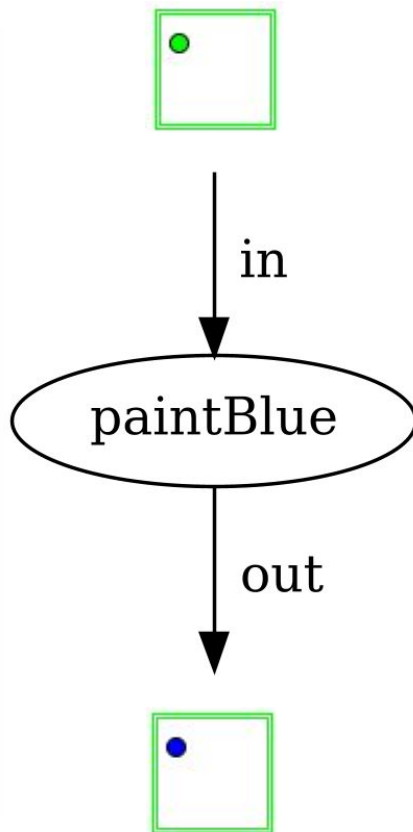
    if (maxMortgage.isRight())
    {
        return maxMortgage.get();
    }
    else
    {
        return maxMortgage.getLeft();
    }
}
```

# Map - Either (right)

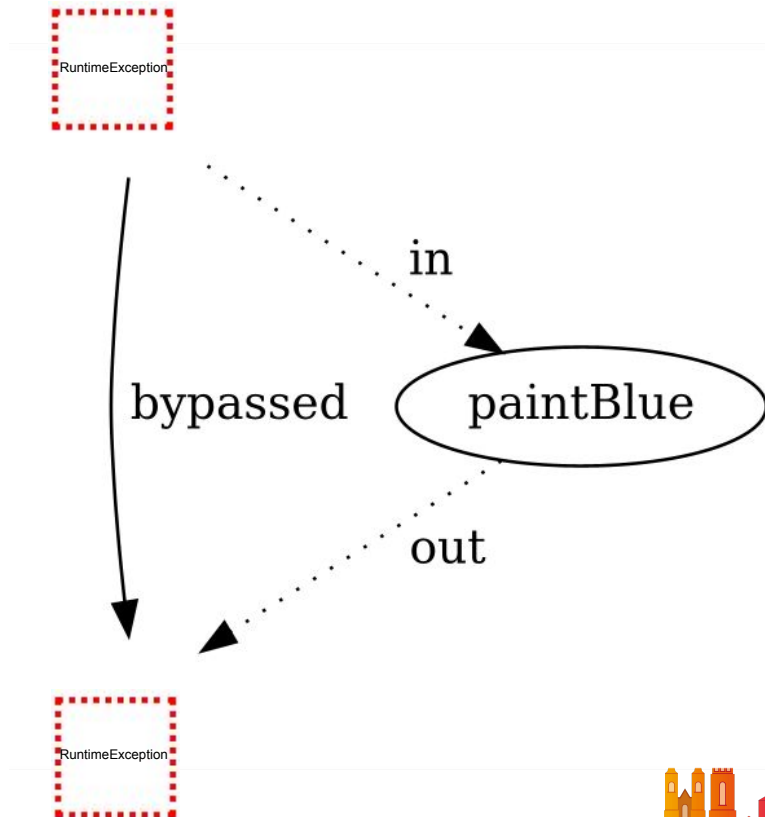
```
@GET
@Path("/max-mortgage")
public Response calculateMaxMortgage()
{
    return businessLogic.calculateMaxMortgage() Either<String, BigDecimal>
        .map(max -> Response.ok().build()) Either<String, Response>
        .getOrElseGet(error -> Response.serverError().entity(error).build());
}
```



## Map - Try (success)



# Map - Try (failure)



# Map - Try

```
@Nullable
public AccountWrapper getWrappedAccount() {
    try {
        Account account = getAccount();
        return new AccountWrapper(account);
    } catch (NoSuchElementException e) {
        return null;
    }
}
```

# Map - Try

```
@Nullable
public AccountWrapper getWrappedAccount()
{
    Try<Account> account = Try.of(() -> getAccount());
    if (account.isSuccess())
    {
        return new AccountWrapper(account.get());
    }
    else
    {
        return null;
    }
}
```

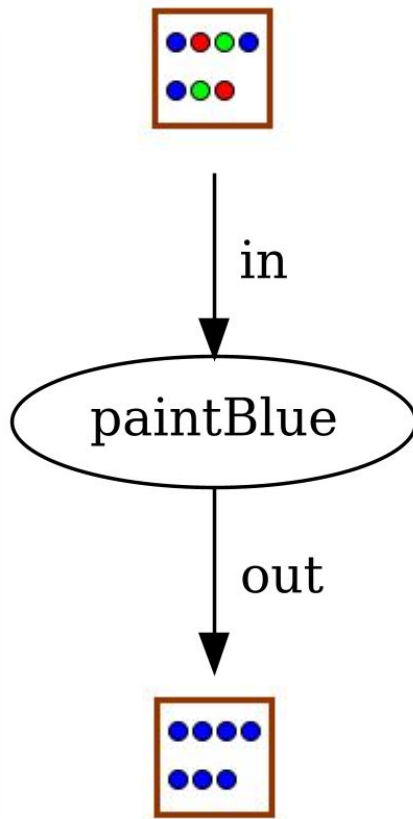
# Map - Try

```
@Nullable
public AccountWrapper getWrappedAccount()
{
    return Try.of(() -> getAccount()) Try<Account>
        .map(account -> new AccountWrapper(account)) Try<AccountWrapper>
        .getOrNull();
}
```

# Map - Try

```
@Nullable
public AccountWrapper getWrappedAccount()
{
    return Try.of(() -> getAccount()) Try<Account>
        .map(AccountWrapper::new) Try<AccountWrapper>
        .getOrNull();
}
```

# Map - List and Set



# Map - List and Set

@NotNull

```
public List<AccountWrapper> getWrappedAccounts() {
    List<AccountWrapper> result = List.empty();

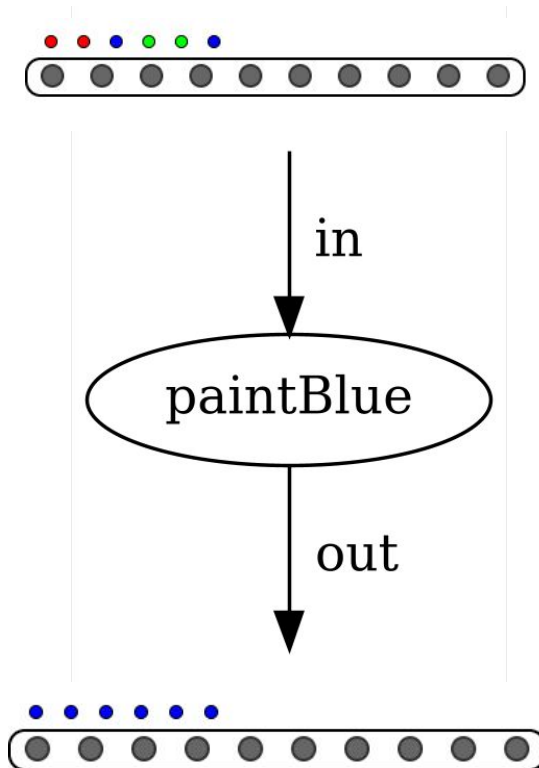
    List<Account> accounts = getAccounts();
    for (Account account: accounts) {
        result = result.append(new AccountWrapper(account));
    }
    return result;
}
```



# Map - List and Set

```
@NotNull
public List<AccountWrapper> getWrappedAccounts() {
    return getAccounts()
        .map(AccountWrapper::new);
}
```

# Map - Stream



# Map - Stream

```
@NotNull
public Stream<Integer> calculatePrimes(@NotNull Stream<Integer> indices) {
    Stream<Integer> primes = Stream.empty();

    for (Integer index : indices)
    {
        primes = primes.append(calculateNthPrime(index));
    }

    return primes;
}
```

# Map - Stream

```
@NotNull
public Stream<Integer> calculatePrimes(@NotNull Stream<Integer> indices) {
    Stream<Integer> primes = Stream.empty();

    for (Integer index : indices)
    {
        primes = primes.append(calculateNthPrime(index));
    }

    return primes;
}
```

Infinite loop if  
stream is  
infinite

# Map - Stream

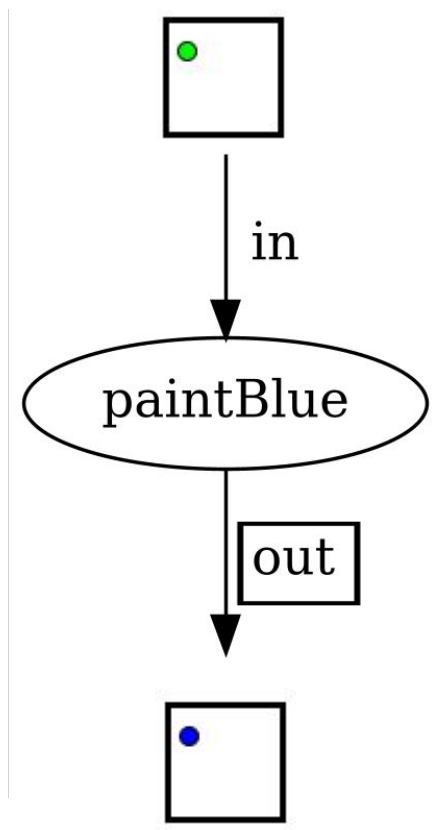
```
@NotNull
public Stream<Integer> calculatePrimes(@NotNull Stream<Integer> indices) {
    return indices.map(index -> calculateNthPrime(index));
}
```

# FlatMap

- Take the contents of your building block
- Put the contents through a function that yields a new building block of the same type
- Use function output to construct a new building block



# FlatMap - Option



# FlatMap - Option

```
@NotNull
```

```
Option<Account> getAccount(@NotNull String username);
```

```
@NotNull
```

```
Option<Permissions> getAccountPermissions(@NotNull Account account);
```



# FlatMap - Option

```
Option<Permissions> permissions = getAccount(username)  
    .map(account -> getAccountPermissions(account));
```

# FlatMap - Option

```
Option<Option<Permissions>> permissions = getAccount(username)  
    .map(account -> getAccountPermissions(account));
```

# FlatMap - Option

```
Option<Permissions> permissions = getAccount(username)
    .map(account -> getAccountPermissions(account).getOrNull());

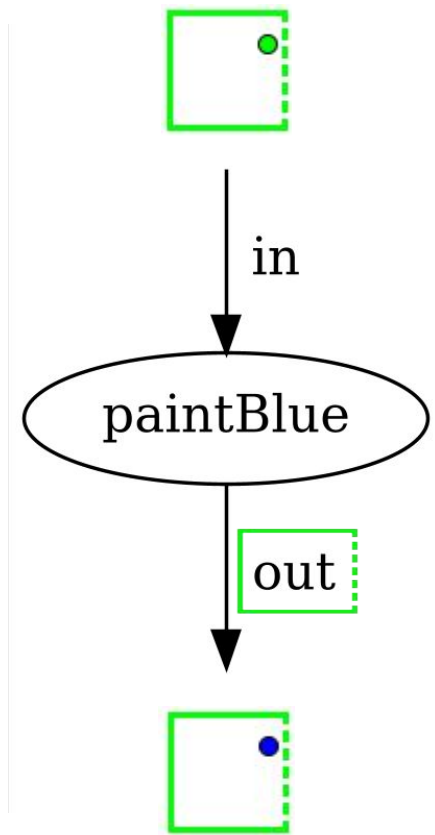
System.out.println(permissions); // Some(null)
```

# FlatMap - Option

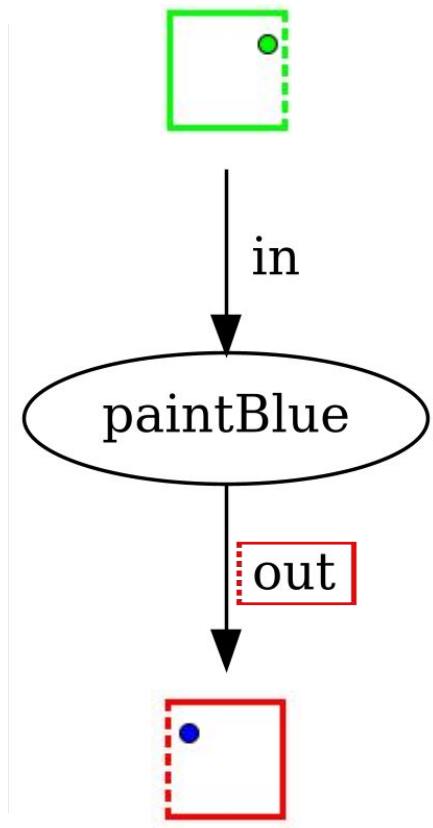
```
Option<Permissions> permissions = getAccount(username)
    .flatMap(account -> getAccountPermissions(account));

System.out.println(permissions); // None
```

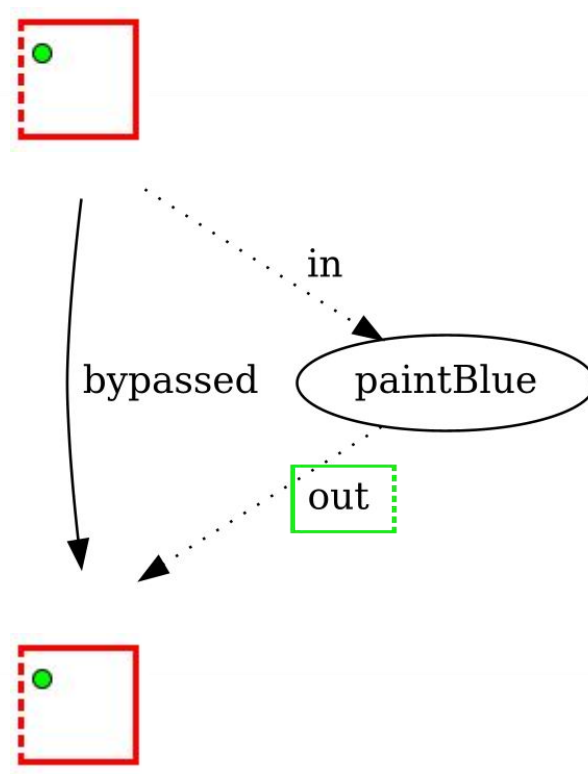
## FlatMap - Either (right)



## FlatMap - Either (right)



## FlatMap - Either (left)



## FlatMap - Either

```
@NotNull
```

```
Either<String, Address> determineAddress(  
    @NotNull String postalCode,  
    @NotNull String houseNumber);
```

```
@NotNull
```

```
Either<String, BigDecimal> determineShippingCosts(  
    @NotNull Address address);
```



# FlatMap - Either

```

Either<String, BigDecimal> shippingCosts =
    determineAddress(postalCode, houseNumber)
    .map(address -> {
        Either<String, BigDecimal> shippingCost = determineShippingCosts(address);

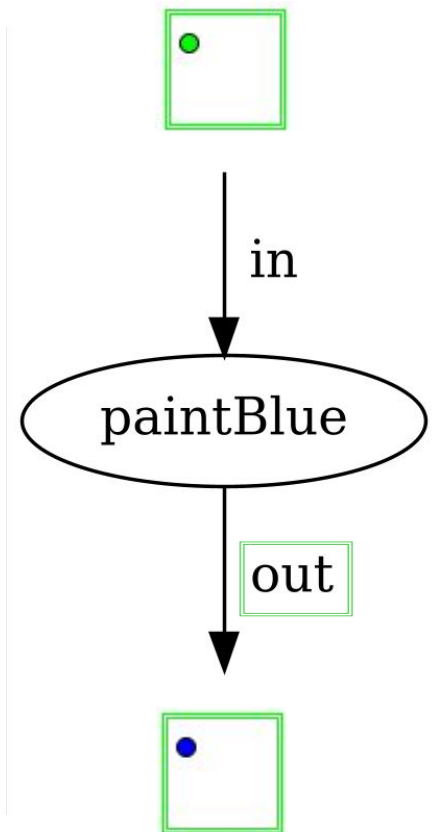
        if (shippingCost.isRight()) {
            return shippingCost.get();
        } else {
            // Help? I can't turn this into a left!
            return null;
        }
    });

```

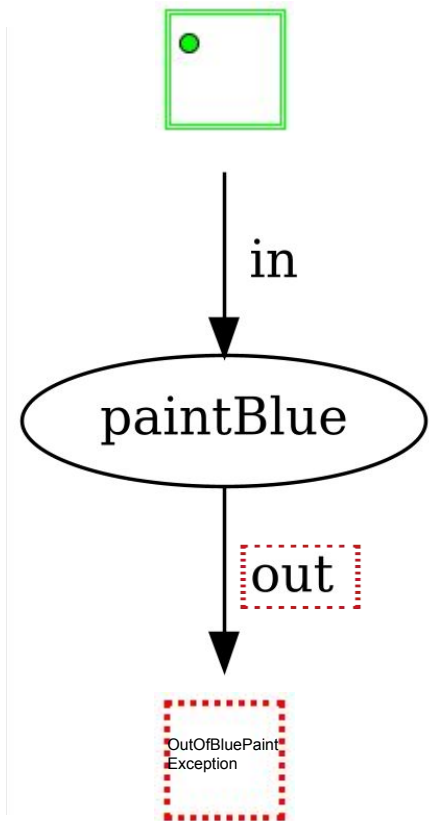
## FlatMap - Either

```
Either<String, BigDecimal> shippingCosts =  
    determineAddress(postalCode, houseNumber)  
    .flatMap(address -> determineShippingCosts(address));
```

## FlatMap - Try (success)



## FlatMap - Try (success)



## FlatMap - Try

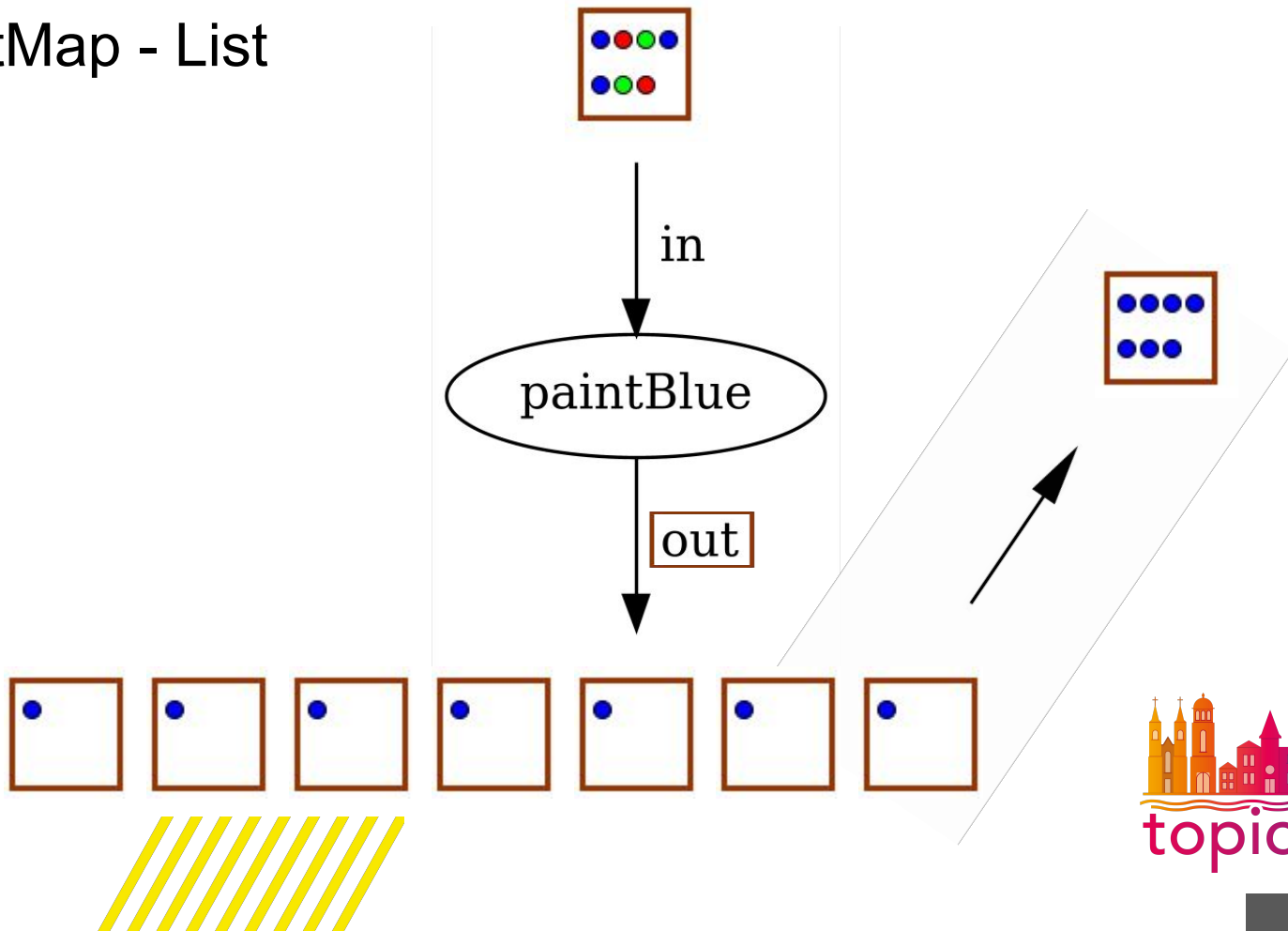
```
Try<Account> getRemoteAccount(  
    @NotNull String username);
```

```
Try<Permissions> getRemotePermissions(  
    @NotNull Account account);
```

## FlatMap - Try

```
Try<Permissions> permissions = getRemoteAccount(username)  
    .flatMap(account -> getRemotePermissions(account));
```

# FlatMap - List



# FlatMap - List

@NotNull

```
List<File> getFiles(@NotNull Account account);
```



# FlatMap - List

```
@NotNull
```

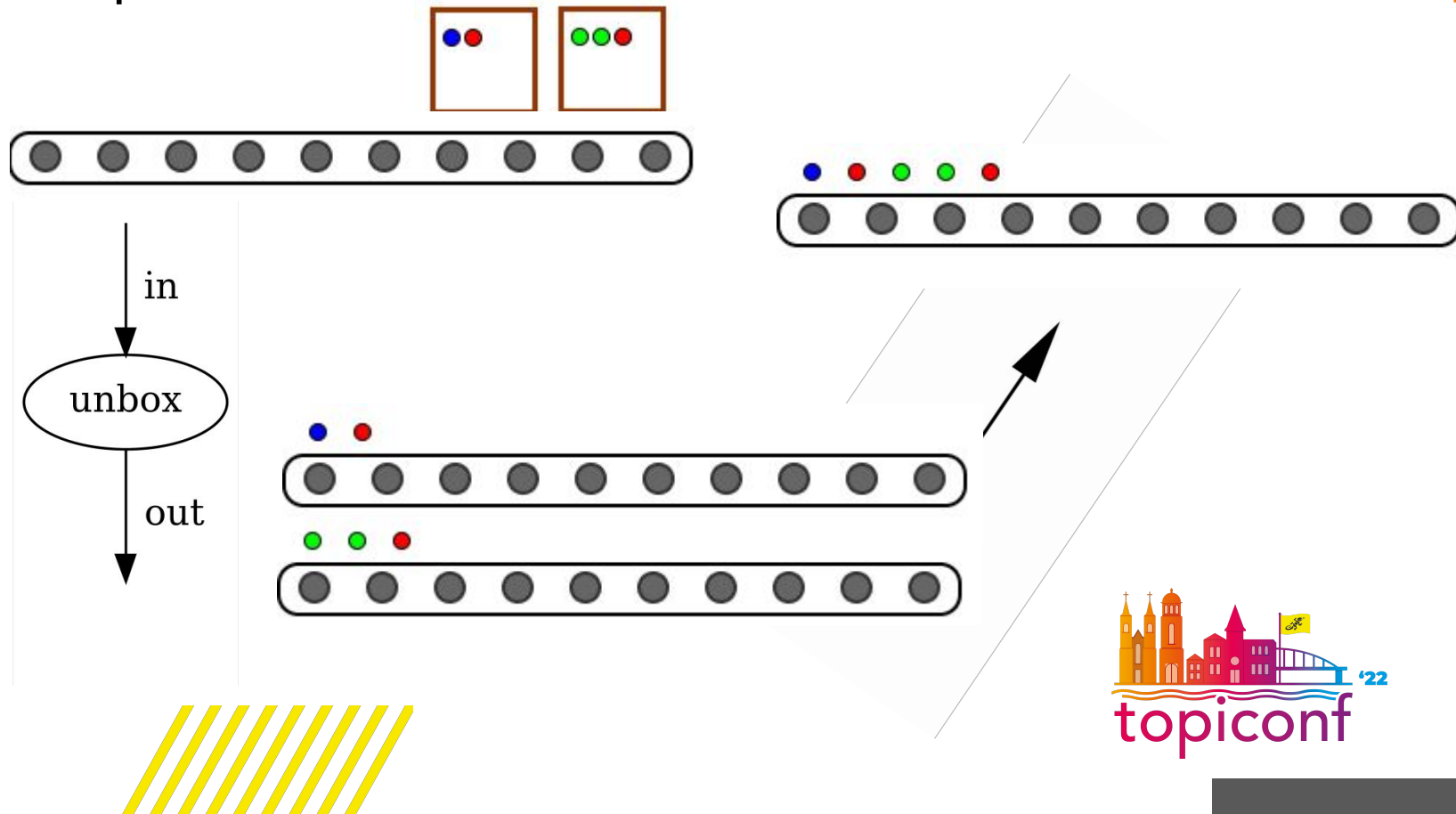
```
List<File> getFiles(@NotNull Account account);
```

## FlatMap - List

```
List<Account> accounts = getAccounts();
```

```
List<File> files = accounts  
    .flatMap(account -> getFiles(account));
```

# FlatMap - Stream



# FlatMap

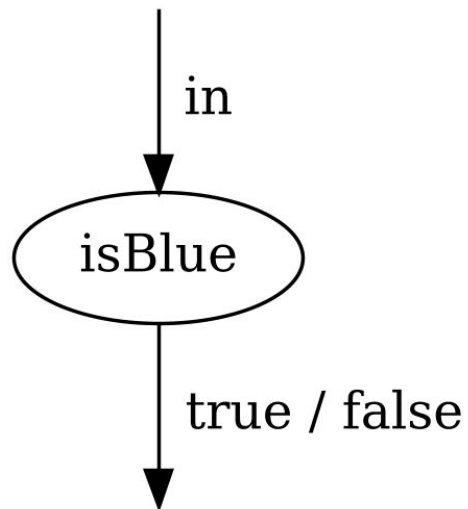
- Whenever a map operation would yield a nested building block, use flatMap instead
  - Option<Option<T>>
  - Either<A,Either<A,B>>
  - Try<Try<T>>
  - List<List<T>>
  - Set<Set<T>>
  - Stream<Stream<T>>

# Filter

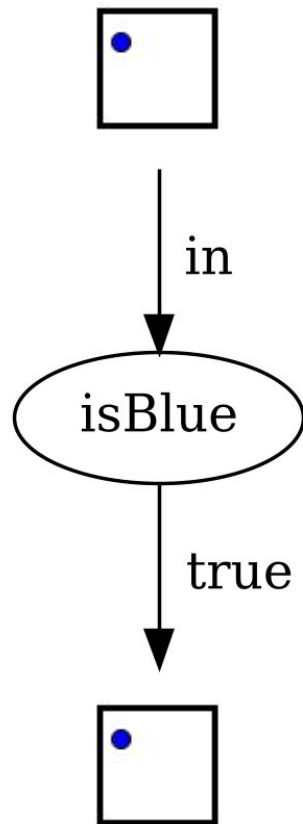
- Take the contents of your building block
- Pass it through a predicate
  - If true: keep the contents
  - If false: change building block to alternate type



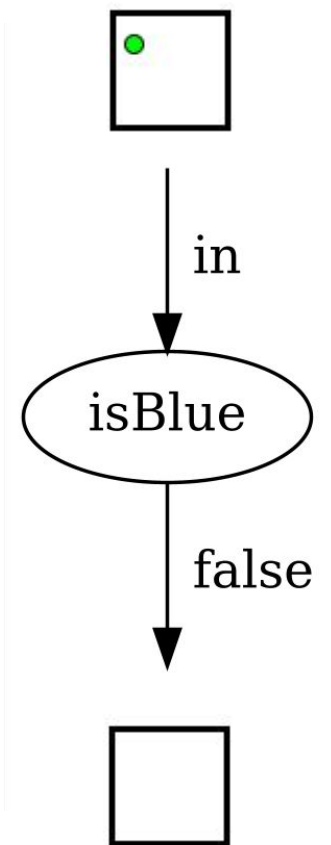
# Filter



## Filter - Option

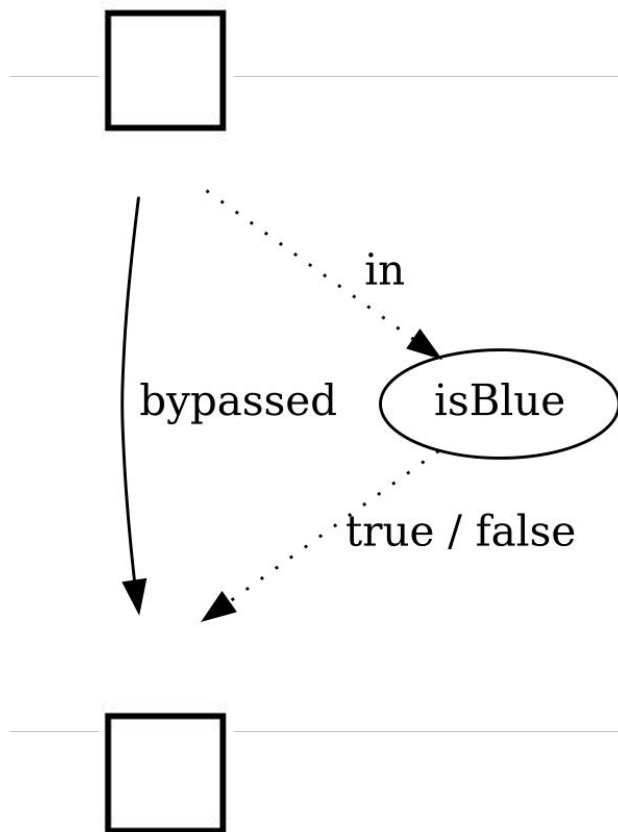


## Filter - Option





## Filter - Option



## Filter - Option

```
@NotNull
Option<Account> getAccount(
    @NotNull String username);
```

```
@NotNull
Option<Permissions> getAccountPermissions(
    @NotNull Account account);
```

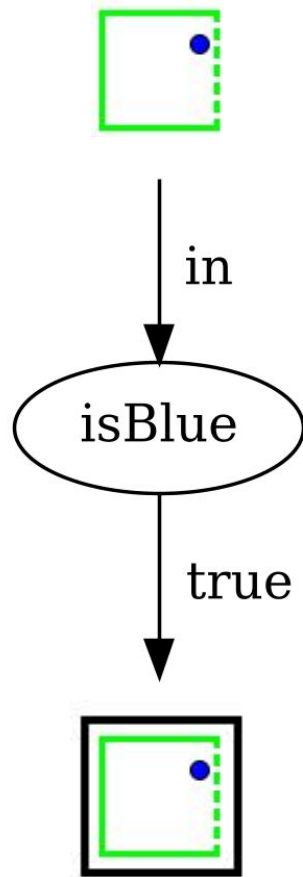
## Filter - Option

```
Option<Permissions> perm = getAccount(username)  
    .flatMap(account -> getAccountPermissions(account));
```

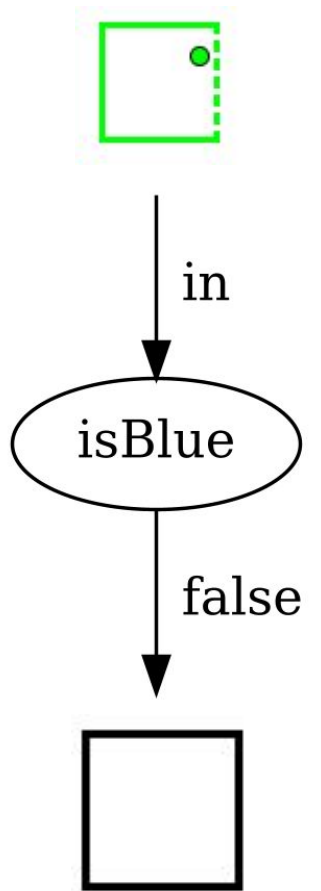
# Filter - Option

```
Option<Permissions> perm = getAccount(username) Option<Account>
    .filter(Objects::nonNull)
    .flatMap(account -> getAccountPermissions(account)) Option<Permissions>
    .filter(Objects::nonNull);
```

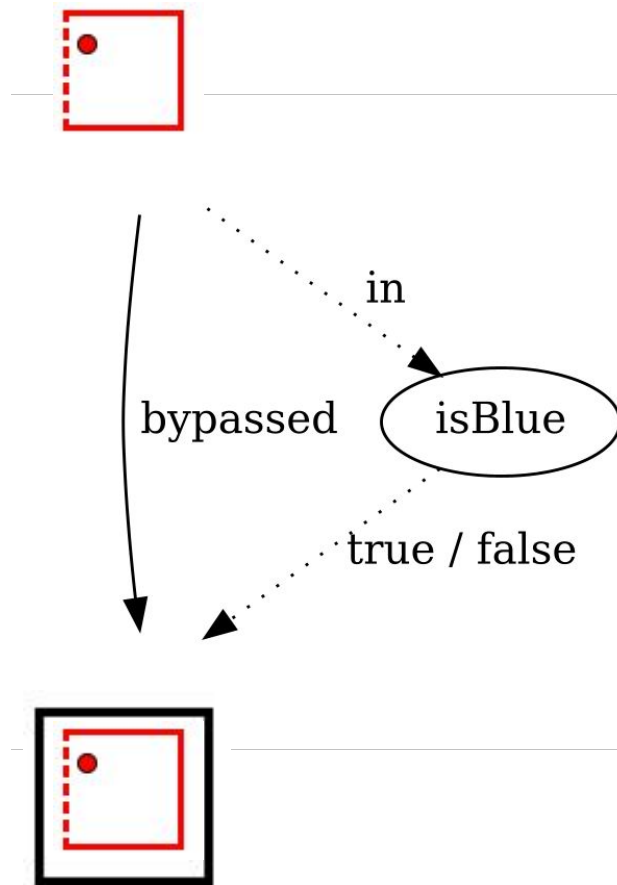
## Filter - Either (right)



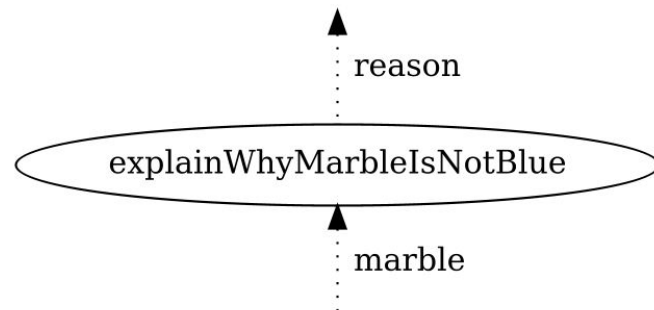
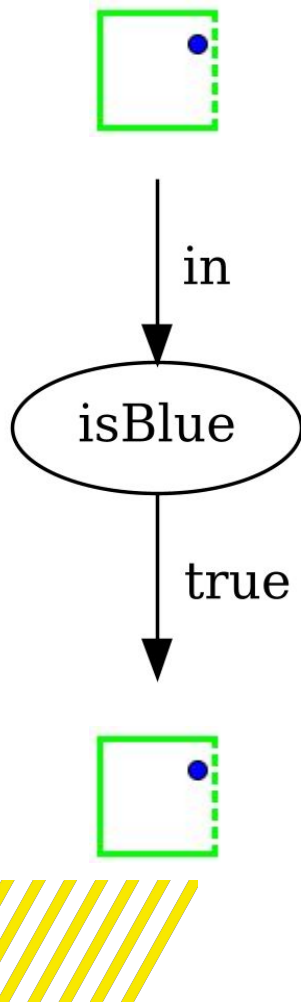
## Filter - Either (right)



## Filter - Either (left)

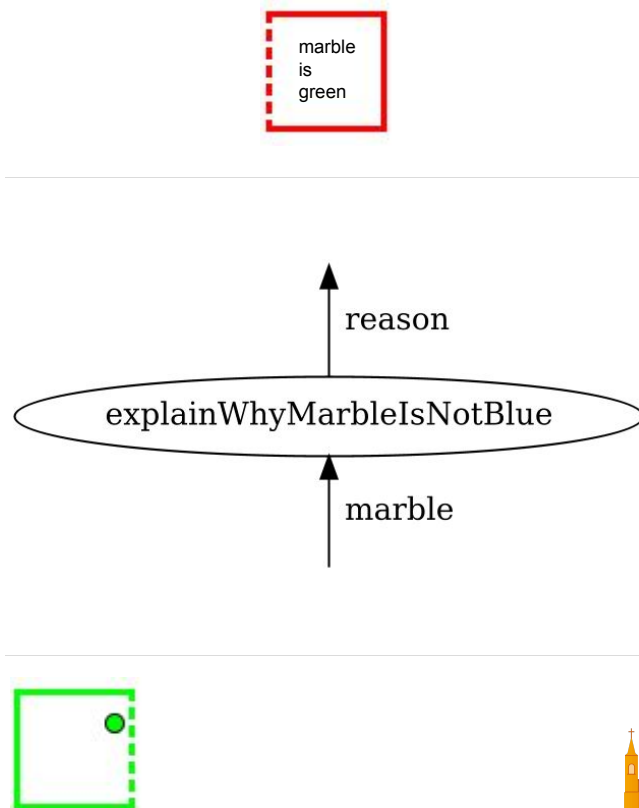
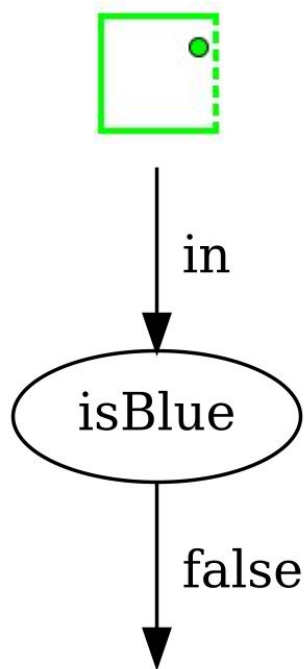


FilterOrElse -  
Either (right)



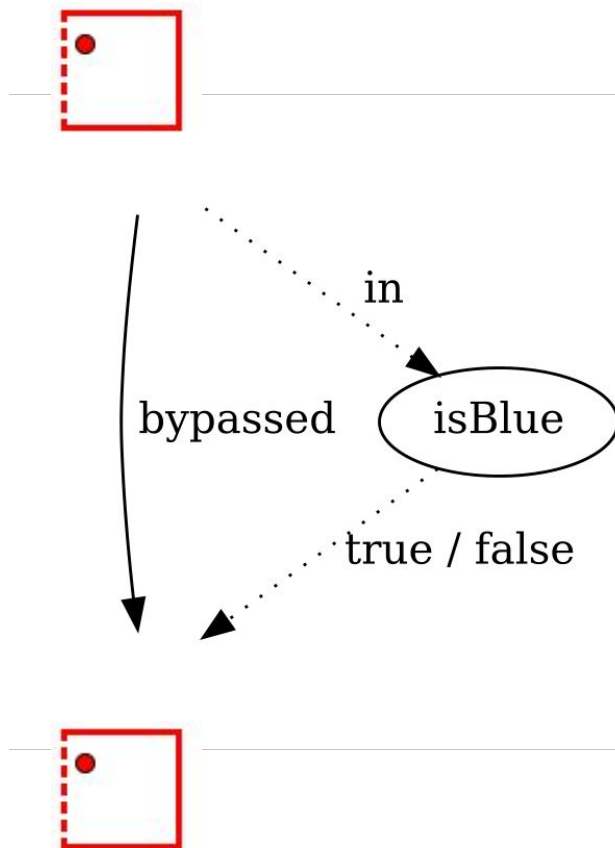
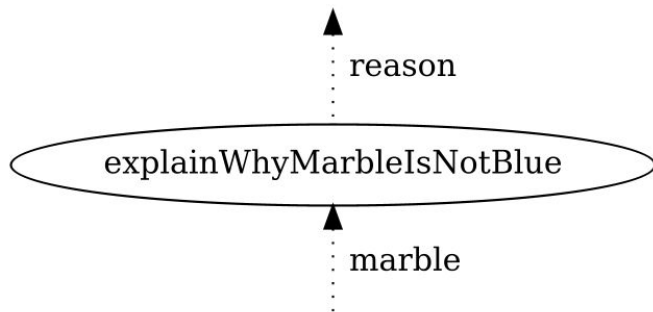


## FilterOrElse - Either (right)



# FilterOrElse

- Either (left)



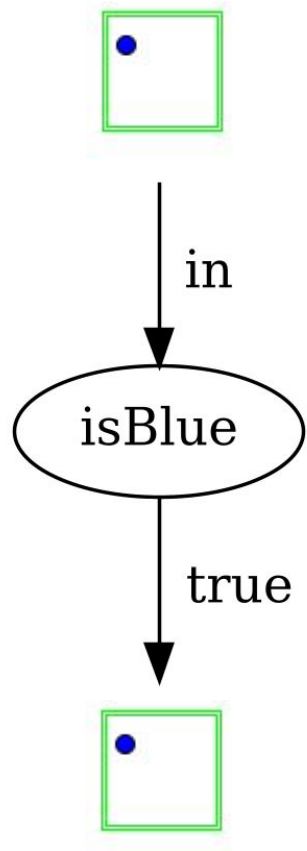
# Filter

```
Option<Either<String,
    BigDecimal>> shippingCosts = determineAddress(postalCode, houseNumber) Either<String, Address>
    .flatMap(address -> determineShippingCosts(address)) Either<String, BigDecimal>
    .filter(costs -> costs.compareTo(BigDecimal.ZERO) > 0);
```

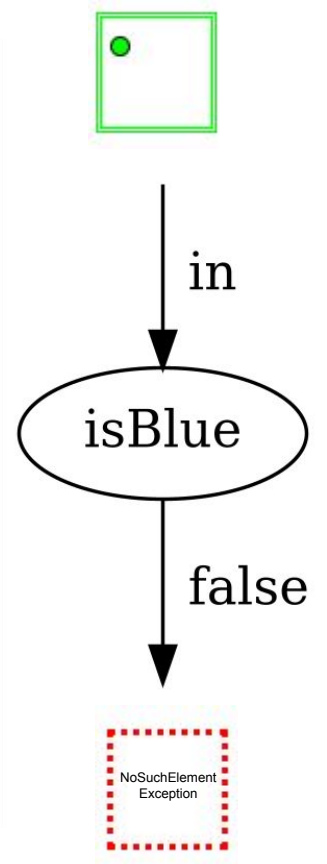
# Filter

```
Either<String,
    BigDecimal> shippingCosts = determineAddress(postalCode, houseNumber) Either<String, Address>
    .flatMap(address -> determineShippingCosts(address)) Either<String, BigDecimal>
    .filterOrElse(costs -> costs.compareTo(BigDecimal.ZERO) > 0,
        costs -> "Shipping costs must be positive, but actual costs are %s"
            .formatted(costs));
```

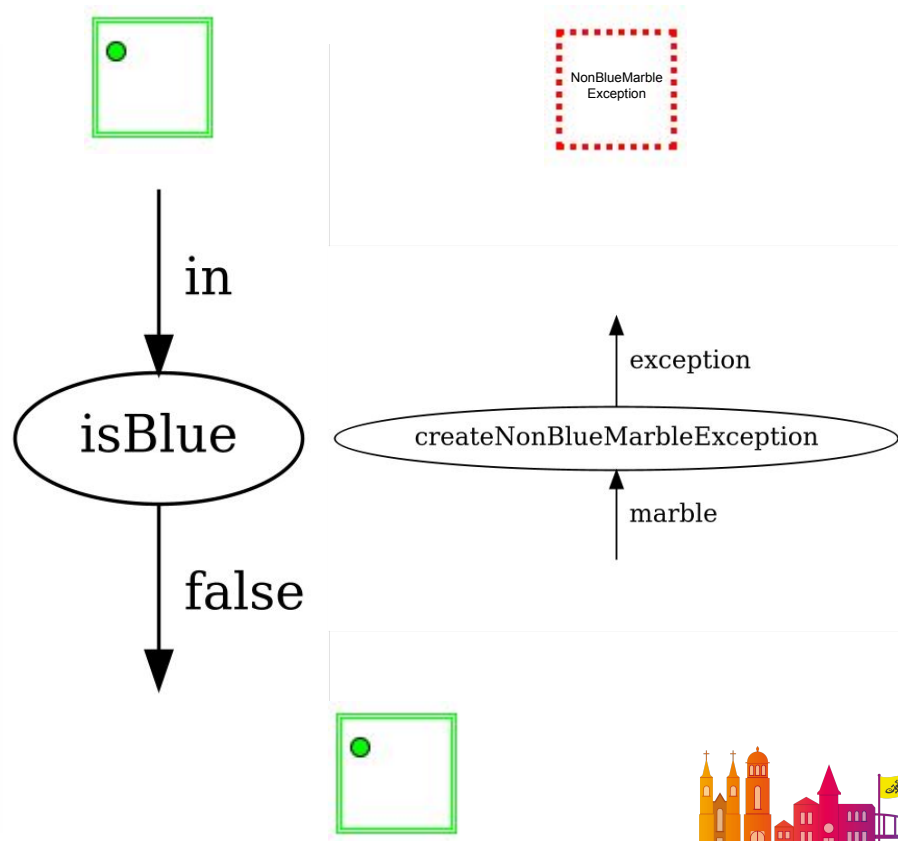
## Filter - Try (success)



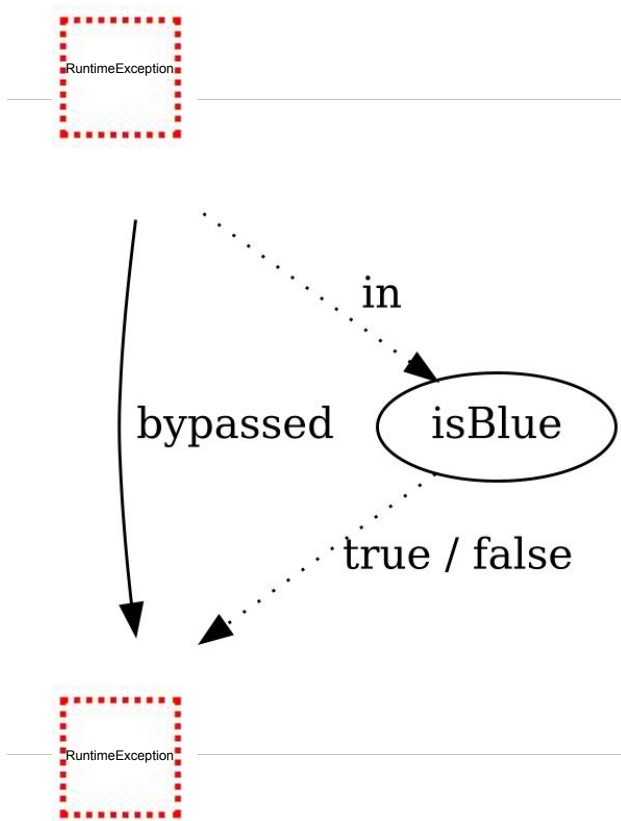
## Filter - Try (success)



## Filter - Try (success)



# Filter - Try (failure)

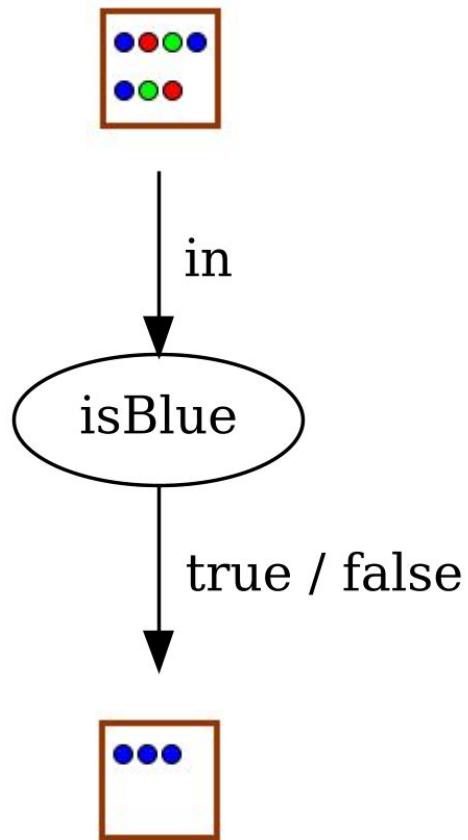




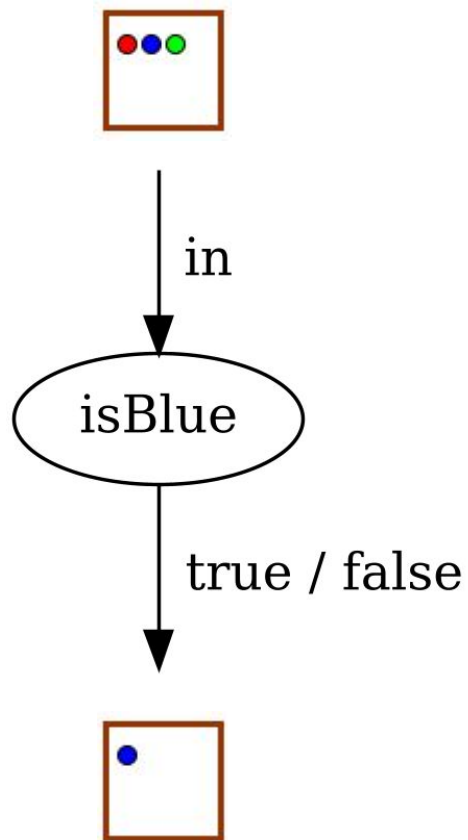
# Filter - Try (failure)

```
Try<Permissions> permissions =
    getRemoteAccount(username) Try<Account>
        .filter(Objects::nonNull)
        .flatMap(account -> getRemotePermissions(account)) Try<Permissions>
        .filter(Objects::nonNull, p -> new NullPointerException());
```

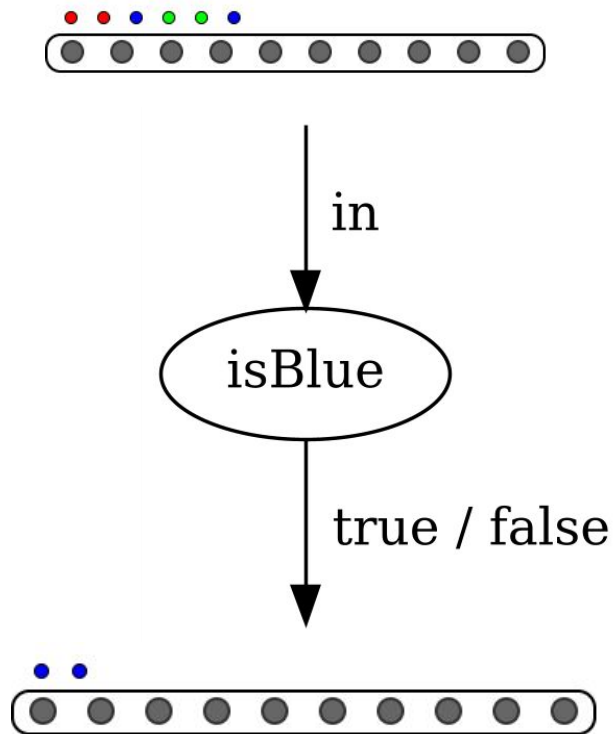
## Filter - List



## Filter - Set



# Filter - Stream



# Filter - List / Set / Stream

```
List<Integer> evenNumbers = List.of( ...elements: 1, 2, 3, 4, 5)
    .filter(number -> isEven(number));

System.out.println(evenNumbers); // List(2, 4)
```

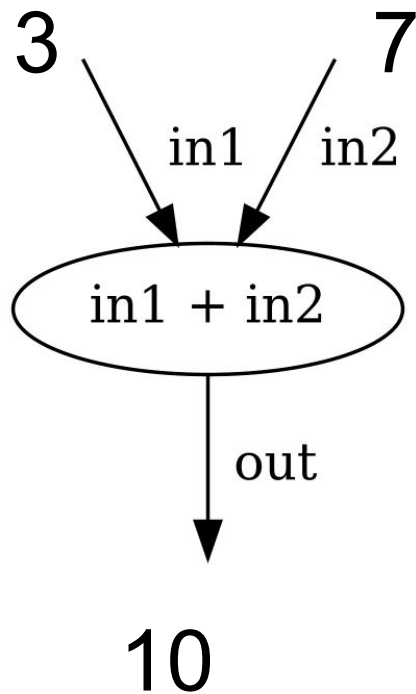
# Reduce

- Take two elements from your List / Set / Stream
- Apply a BiFunction to turn it into a single element
- Repeat until only 1 element remains
- Depending on implementation an empty List / Set / Stream either yields an Option or throws an exception



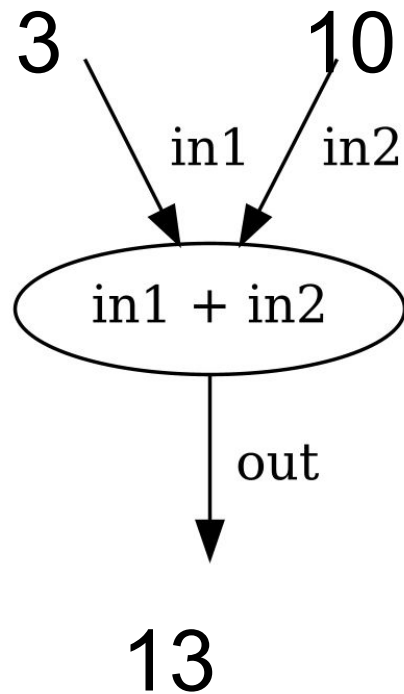
## Reduce

3 1 3 3 7



# Reduce

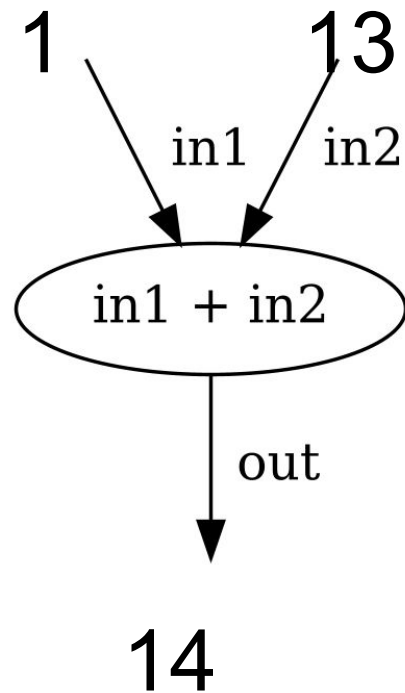
3 1 3 10





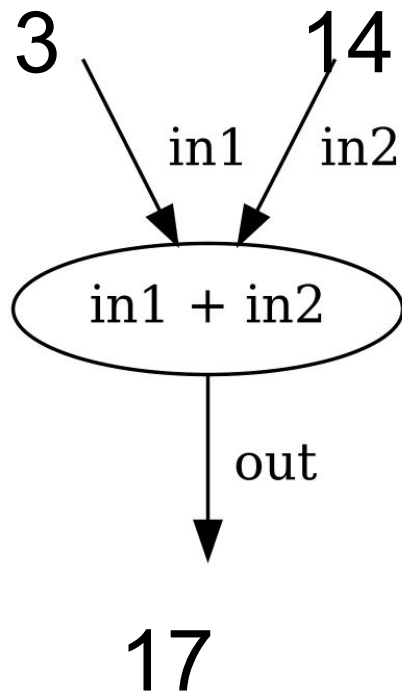
# Reduce

3 1 13



# Reduce

3 14



# Reduce

```
Integer sum = List.of( ...elements: 3, 1, 3, 3, 7)
    .reduce((a, b) -> a + b);
```

```
System.out.println(sum); // 17
```

# Reduce

```
Integer sum = List.of( ...elements: 3, 1, 3, 3, 7)  
    .reduce(Integer::sum);
```

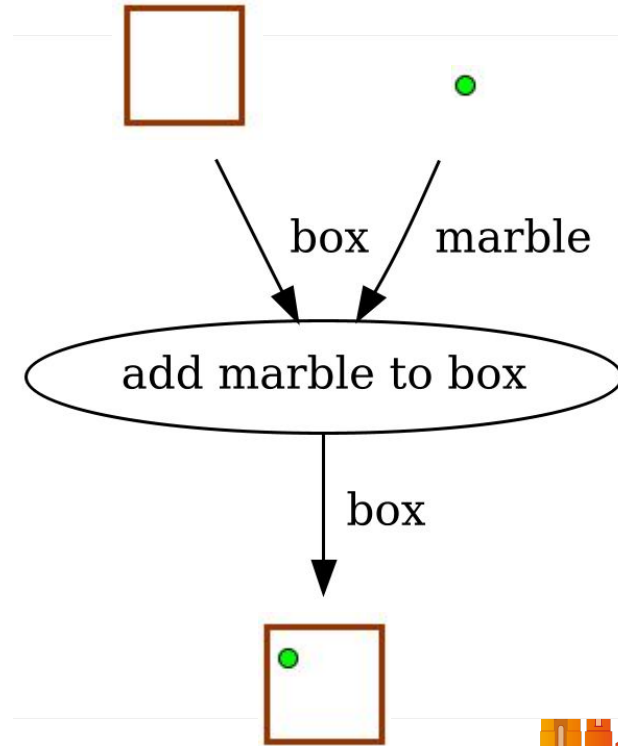
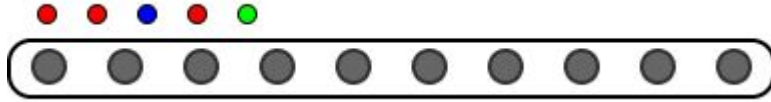
```
System.out.println(sum); // 17
```

# Fold

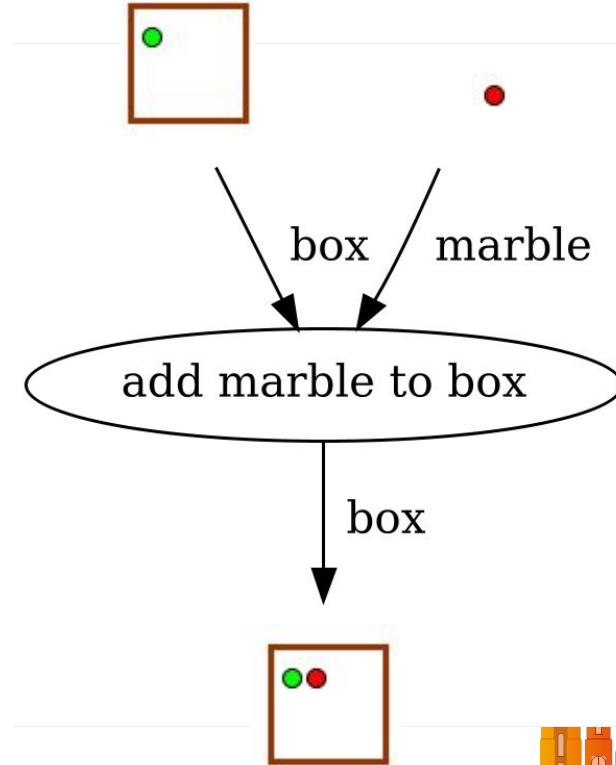
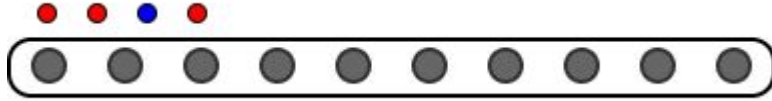
- Similar to reduce, but combines values with a seed object
- BiFunction with intermediate object and next value as inputs
- Can be done in left (front to end) or right (end to front) order, assuming underlying collection is ordered



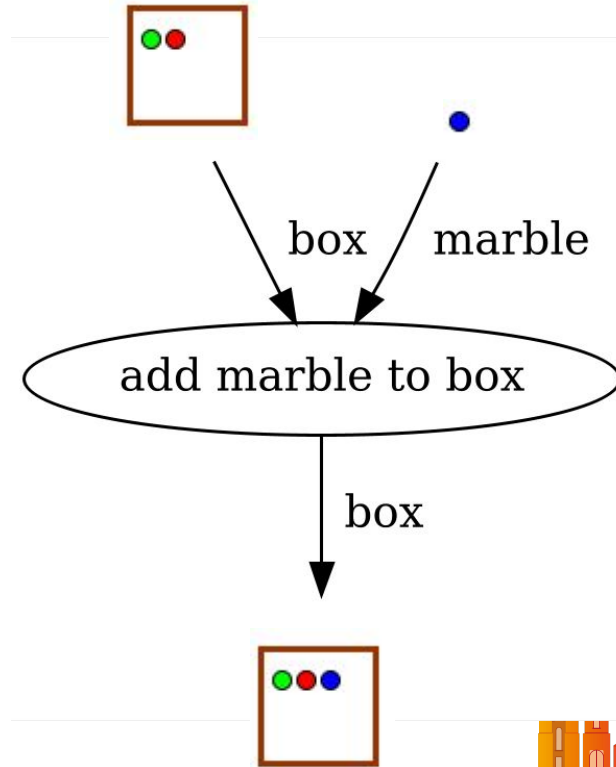
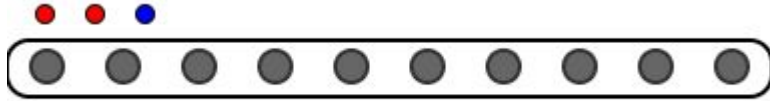
# Fold



# Fold

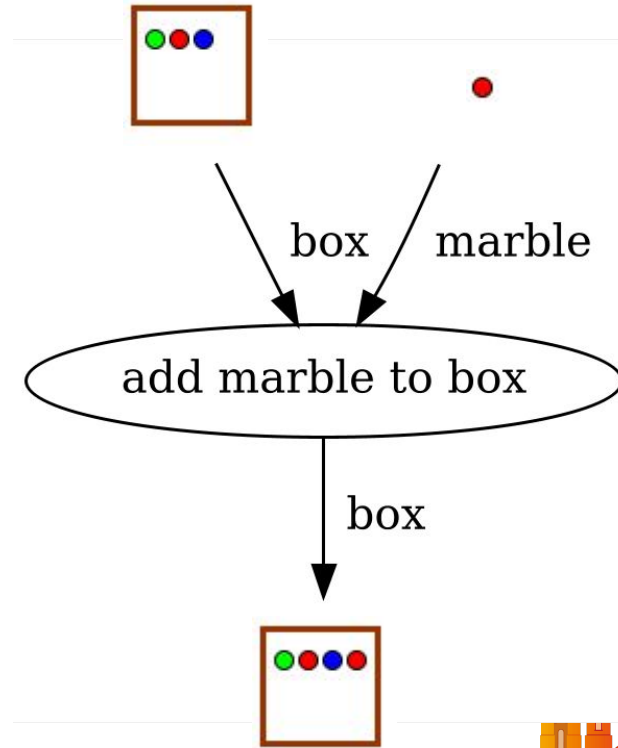
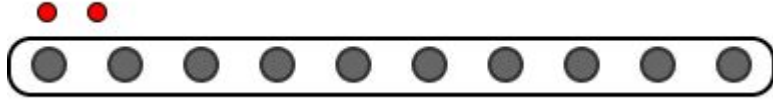


# Fold

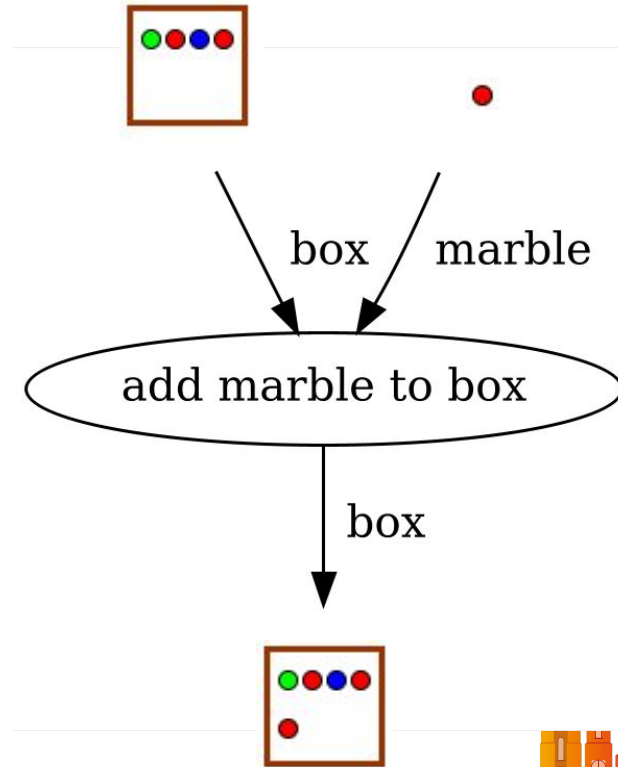
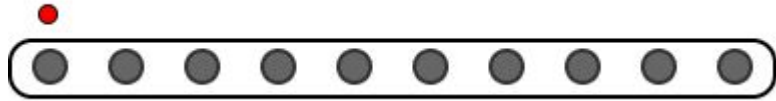




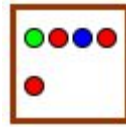
# Fold



# Fold



# Fold - End result



# FoldLeft

```
String result = List.of( ...elements: 1, 2, 3, 4, 5)
    .foldLeft( zero: "-",
        (string, number) ->
            string + number + "-");
```

```
System.out.println(result); // -1-2-3-4-5-
```

# FoldRight

```
String result = List.of( ...elements: 1, 2, 3, 4, 5)
    .foldRight( zero: "-",
        (number, string) ->
            string + number + "-");
```

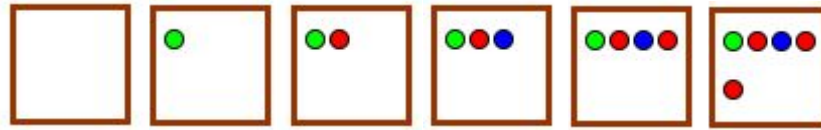
```
System.out.println(result); // -5-4-3-2-1-
```

# Scan

- Similar to fold
- All intermediate values returned rather than just end result
- Caveat: requires seed object to be immutable



# Scan - End result



# ScanLeft

```
List<String> result =  
    List.of( ...elements: 1, 2, 3, 4, 5)  
        .scanLeft( zero: "-",  
                   (string, number) -> string + number + "-");
```

```
result.forEach(System.out::println);
```

```
/* -  
 * -1-  
 * -1-2-  
 * -1-2-3-  
 * -1-2-3-4-  
 * -1-2-3-4-5-  
 */
```



# ScanRight

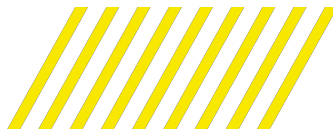
```
List<String> result =  
    List.of( ...elements: 1, 2, 3, 4, 5)  
        .scanRight( zero: "-",  
                    (number, string) -> string + number + "-");
```

```
result.forEach(System.out::println);
```

```
/* -  
* -5-  
* -5-4-  
* -5-4-3-  
* -5-4-3-2-  
* -5-4-3-2-1-  
*/
```

# What did we cover?

- Functions
- Monads / building blocks
  - Option / Optional
  - Either
  - Try
- Operations
  - Map
  - Flatmap
  - Filter
  - Reduce
  - Fold
  - Scan



# What didn't we cover?

- Immutability
  - Immutable objects
  - Lenses
- More about functions
  - Pure functions
  - Higher order functions
  - Currying
- More building blocks
  - Futures
  - Lazy
  - Validation



## Further reading

- Java
  - [vavr.io](https://vavr.io)
  - [github.com/jsteenbeeke/vavr-workshop](https://github.com/jsteenbeeke/vavr-workshop)
- Kotlin
  - [arrow-kt.io](https://arrow-kt.io)
- Clojure
  - [clojure.org](https://clojure.org)





**IMPACT**  
**ON SOCIETY**



Jeroen Steenbeeke



Improve your code with FP concepts