Pragmatic Functional Programming

Survey

What do you know about Functional Programming?

Person & People

```
public class Person {
    private String name;
    private int age;
    private Gender gender;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public Gender getGender() {
        return gender;
    }

    ...
}
```

```
public class People {
    private List<Person> people;

    ...
}
```

Requirement (I)

Find a person by age

Traditional Java

```
public Person findByAge(final int age) {
    for (Person person : people) {
        if (person.getAge() == age) {
            return person;
        }
    }

    return null;
}
```

Requirement (II)

Find a person by name

Traditional Java

```
public Person findByName(final String name) {
    for (Person person : people) {
        if (name.equals(person.getName())) {
            return person;
        }
    }

    return null;
}
```

Do you see something?

```
public Person findByAge(final int age) {
    for (Person person : people) {
        if (person.getAge() == age) {
            return person;
        }
    }

    return null;
}
```

```
public Person findByName(final String name) {
    for (Person person : people) {
        if (name.equals(person.getName())) {
            return person;
        }
    }

    return null;
}
```

Requirement (III)

Find a person by gender

Traditional Java

```
public Person findByGender(final Gender gender) {
    for (Person person : people) {
        if (gender == person.getGender()) {
            return person;
        }
    }

    return null;
}
```

Do you see something?

```
public Person findByAge(final int age) {
    for (Person person : people) {
        if (person.getAge() == age) {
            return person;
        }
    }

    return null;
}
```

```
public Person findByName(final String name) {
    for (Person person : people) {
        if (name.equals(person.getName())) {
            return person;
        }
    }

    return null;
}
```

```
public Person findByGender(final Gender gender) {
    for (Person person : people) {
        if (gender == person.getGender()) {
            return person;
        }
    }

    return null;
}
```

Do you see something?

```
public Person findByAge(final int age) {
    for (Person person : people) {
        if (person.getAge() == age) {
            return person;
        }
    }

    return null;
}
```

```
public Person findByName(final String name) {
    for (Person person : people) {
        if (name.equals(person.getName())) {
            return person;
        }
    }

    return null;
}
```

```
public Person findByGender(final Gender gender) {
    for (Person person : people) {
        if (gender == person.getGender()) {
            return person;
        }
    }

    return null;
}
```

Duplication

Gotcha

Duplicated **find** code

Cleanup the Duplication

```
public Person find(final Predicate<Person> predicate) {
    for (Person person : people) {
        if (predicate.test(person)) {
            return person;
        }
    }

    return null;
}


public interface Predicate<T> {
    boolean test(final T t);
}
```

Clean Code (VI)

```
public Person findByAge(final int age) {
    return find(new Predicate<Person>() {
        @Override
        public boolean test(final Person person) {
            return person.getAge() == age;
        }
    });
}
```

Gotcha

New query, New wrapped method

Predicate as Method

```
public static Predicate<Person> byAge(final int age) {
    return new Predicate<Person>() {
        @Override
        public boolean test(final Person person) {
            return person.getAge() == age;
        }
    };
}
```

Clean Code (VII)

```
people.find(byAge(age));
people.find(byName(name));
```

Requirement (IV)

Find a person by both age and name

By Age and Name

byAgeAndName?

Clean Code (VIII)

```
people.find(and(byAge(age), byName(name)));
```

Simple “and” and more

```
public static <T> Predicate<T> and(final Predicate<T>... predicates) {
    return new Predicate<T>() {
        @Override
        public boolean test(final T t) {
            for (Predicate<T> predicate : predicates) {
                if (!predicate.test(t)) {
                    return false;
                }
            }

            return true;
        }
    };
}
```

Requirement (V)

Find people whose age is greater than 30

Clean Code ???

```
people.find(ageGreaterThan());
```

Greater Than

```
public static <T extends Comparable<T>> Predicate<T> greaterThan(final T value) {
    return new Predicate<T>() {
        @Override
        public boolean test(T t) {
            return t.compareTo(value) > 0;
        }
    };
}
```

New byAge

```
public static Predicate<Person> byAge(final Predicate<Integer> predicate) {
    return new Predicate<Person>() {
        @Override
        public boolean test(Person person) {
            return predicate.test(person.getAge());
        }
    };
}
```

Clean Code (IV)

```
people.find(byAge(greaterThan(30)));
```

Requirement (VI)

Find all people

Find All Function

```
public People findAll(final Predicate<Person> predicate) {
    List<Person> foundPeople = new ArrayList<Person>();
    for (Person person : people) {
        if (predicate.test(person)) {
            foundPeople.add(person);
        }
    }

    return new People(foundPeople);
}
```

Clean Code (IV)

```
people.findAll(byAge(age));
```

Inspired by Functional Programming

Functional Programming

Functional Programming

“In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. ”

–Wikipedia

Functional Programming

“In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. ”

–Wikipedia

No One Knows the Answer

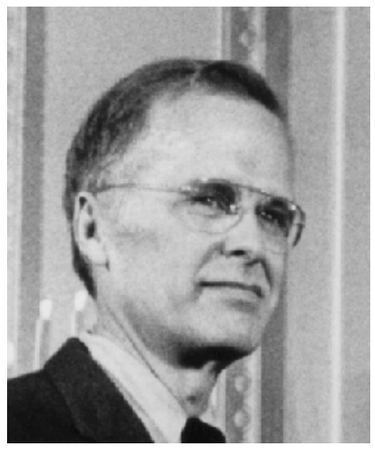
Programming Paradigm

“A programming paradigm is a fundamental style of computer programming. There are four main paradigms : object-oriented, imperative, functional and logic programming. In addition, aspect-oriented programming aims specifically to increase modularity by allowing the separation of cross-cutting concerns.”

–Wikipedia

Functional Programming Concept

Can Programming be Liberated from the von Neumann Style?



On Function

- Function is the core concept in functional programming
- Function should be first-class citizen
    - First-class means
        - It can be created on demand
        - It can be stored in a data structure
        - It can be passed as an argument to a function
        - It can be returned as the value of a function

First-class function in different languages

- functor in C++
- delegate in C#
- lambda/closure in Ruby
- lambda in C++11
- lambda in Java 8
- function in JavaScript

Thoughts with OO

- Build block
  - a class/object in OO
  - function in FP
- Programming to
  - Noun for OO
  - Verb for FP
- There is a lot of verb in OO code
  - Processor, Handler, Runner etc
  - They are functions

As Clean Code

- The code is smaller, the more reusable code will be found

- It’s easy to reuse a method instead of a class

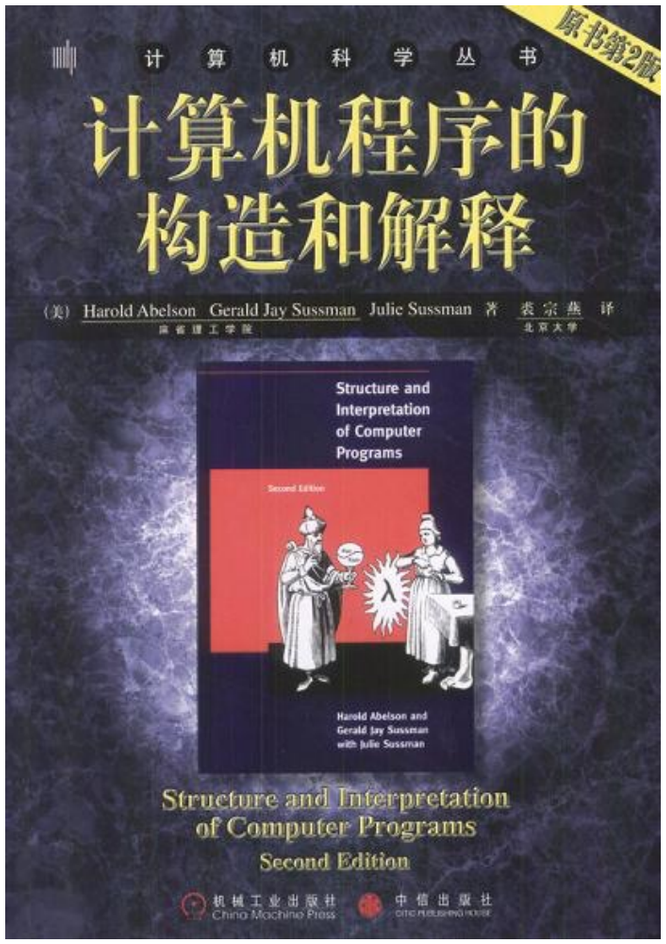
High-order Function

- A high-order function is a function that does at least one of the following:
  - take one or more functions as an input
  - output a function
- Understanding high-order function as behavior combination

Learn Functional Programming Language

- Classical Functional Programming Language
  - Lisp
  - Haskell
  - Scheme
- Pragmatic Functional Programming Language
  - Clojure
  - Scala
  - F#

SICP


计 算 机 科 学 丛 书
原书第2版
计算机程序的构造和解释
(美) Harold Abelson Gerald Jay Sussman Julie Sussman 著
麻省理工学院
裘宗燕 译
北京大学
Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman
Structure and Interpretation of Computer Programs
Second Edition
机械工业出版社 China Machine Press
中信出版社

Function in Java

Java 8 “Function”

- lambda
- function interface
- method reference

Lambda Expression

```
(int x, int y)    ->    x + y
```

Arguments **Arrow** **Body**

Lambda in Java

- Anonymous Classes Replacement
- Name does NOT matter
- Arguments is final

Lambda Syntax

```
(paramters)    ->    expression

(paramters)    ->    {statements}
```

More Lambda

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

Is Valid Lambda?

```
() -> {}

() -> “Java 8”

() -> { return “Java”; }

(int x) -> return “Java ” + x;

(int x) -> { “Java ” + x; }
```

Functional Interface

- a functional interface has exactly one abstract method
- @FunctionalInterface
    - An informative annotation type used to indicate that an interface type declaration is intended to be a functional interface

JDK Functional Interface

- Builtin Functional Interface
    - Predicate: a predicate of argument(s)
    - Function: accepts argument(s) and produces a result.
    - Consumer: accepts argument(s) and returns no result.
    - Supplier: a supplier of results
- Existing Functional Interface
    - Runnable: a Consumer without argument
    - Callable: a Function without argument
    - Comparator: a Function with two arguments and return int result

Which is Functional Interface?

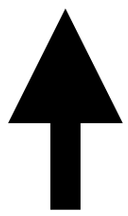
```
public interface Adder {
    int add(int x, int y);
}
```

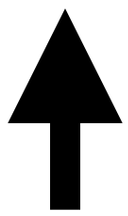
```
public interface SmartAdder extends Adder {
    int add(double x, double y);
}

public interface Nothing {
}
```

Method Reference

```
person -> person.getName();
```

```
Person::getName
```





ClassName **MethodName**

Why Method Reference

- Reuse existing method
- Shortcut for specific lambda

Build Method Reference

- **static** method (Integer::parseInt)
- **type instance** method (String::length)
- **object instance** method (System.out::println)

Static Method

```
(arg) -> ClassName.staticMethod(arg);
```

```
ClassName::staticMethod
```

Type Instance Method

```
(arg, rest) -> arg.instanceMethod(rest);
```

ClassName is the class of **arg**

```
ClassName::instanceMethod
```

Object Instance Method

```
(args) -> expr.instanceMethod(args);
```

```
expr::instanceMethod
```

Constructor Reference

```
(name, age, gender) -> new Person(name, age, gender);
```

```
Person::new
```

ClassName

Rewrite with Method Reference

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

```
BiPredicate<List<String>, String> contains = (list, element) -> list.contains(element);
```

Type Inference

- Simplify code further

- Compiler’s work

Type Inference (Cont.)

```
Comparator<Person> c = (Person p1, Person p2) -> p1.getName().compareTo(p2.getName());

Comparator<Person> c = (p1, p2) -> p1.getName().compareTo(p2.getName());
```

Type Inference (Cont.)

```
Comparator<Person> c = (Person p1, Person p2) -> p1.getName().compareTo(p2.getName());

Comparator<Person> c = (p1, p2) -> p1.getName().compareTo(p2.getName());
```

Bonus

```
Comparator<Person> c = Comparator.comparing(Person::getName);
```

Function Composition

- Comparator
- Predicate
- Function

Comparator Composition

```
Comparator<Person> c = Comparator.comparing(Person::getAge).reversed();
```

```
Comparator<Person> c = Comparator.comparing(Person::getAge).thenComparing(Person::getGender);
```

Predicate Composition

```
Predicate<Person> p = Person::isMale;

~a
Predicate<Person> negateP = p.negate();
```

```
a && b
Predicate<Person> andP = p.and(person -> person.getName().startsWith(“Jack"));

a || b
Predicate<Person> orP = p.or(person -> person.getAge() > 15);
```

Function Composition

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;

[a, b, c] -> f -> g
Function<Integer, Integer> h = f.andThen(g);

[a, b, c] -> g -> f
Function<Integer, Integer> h = f.compose(g);
```

List Transformation

Requirement (VII)

Collect all names who are qualified for China’s Youth Day.

Traditional Java

```
List<String> names = new ArrayList<String>();

for (Person person : people) {
        int age = person.getAge();
        if (age >= 14 && age <= 28) {
             names.add(person.getName());
        }
}
```

Requirement (VIII)

Collect all girl's names who are qualified for China's Youth Day.

Traditional Java

```
List<String> names = new ArrayList<String>();

for (Person person : people) {
     int age = person.getAge();
     if (person.getGender() == Gender.FEMALE) {
          if (age >= 14 && age <= 28) {
               names.add(person.getName());
          }
     }
}
```

Traditional Java

```
List<String> names = new ArrayList<String>();

for (Person person : people) {
    int age = person.getAge();
    if (person.getGender() == Gender.FEMALE) {
        if (age >= 14 && age <= 28) {
            names.add(person.getName());
        }
    }
}
```

New Code

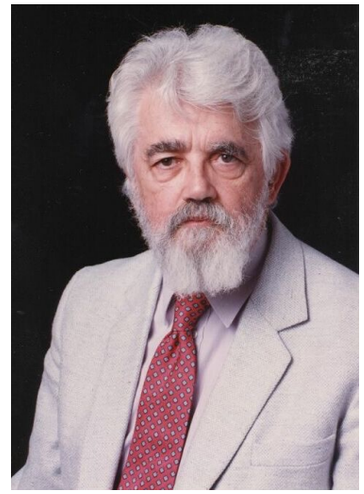
Traditional Java

```
List<String> names = new ArrayList<String>();

for (Person person : people) {
    int age = person.getAge();
    if (person.getGender() == Gender.FEMALE) {
        if (age >= 14 && age <= 28) {
            names.add(person.getName());
        }
    }
}
```

How Code Complicates

Lisp

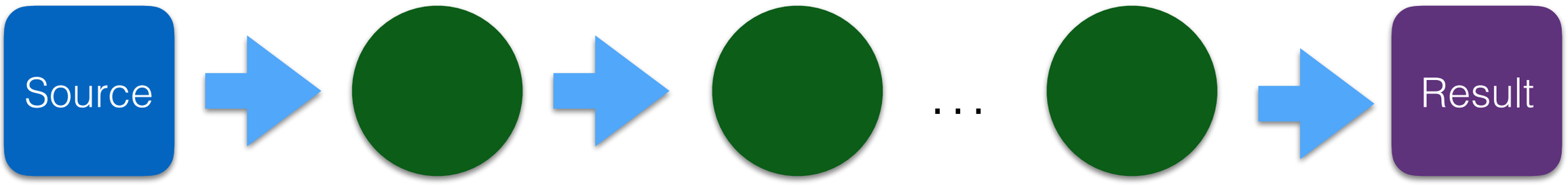
- List Processing Language
- List is the key feature of Lisp



List in different languages

- Collection/Iterable/Iterator in Java
- Container in C++
- Enumerable in Ruby

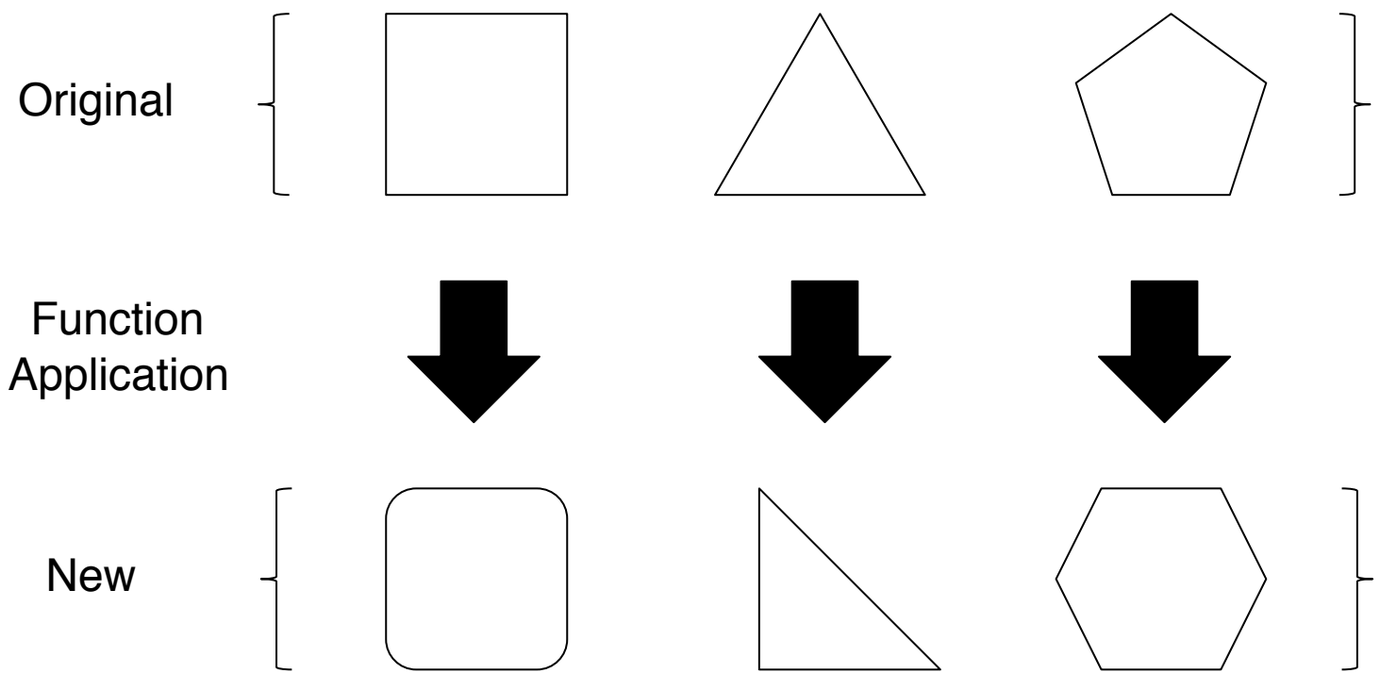
List Transformational Mindset



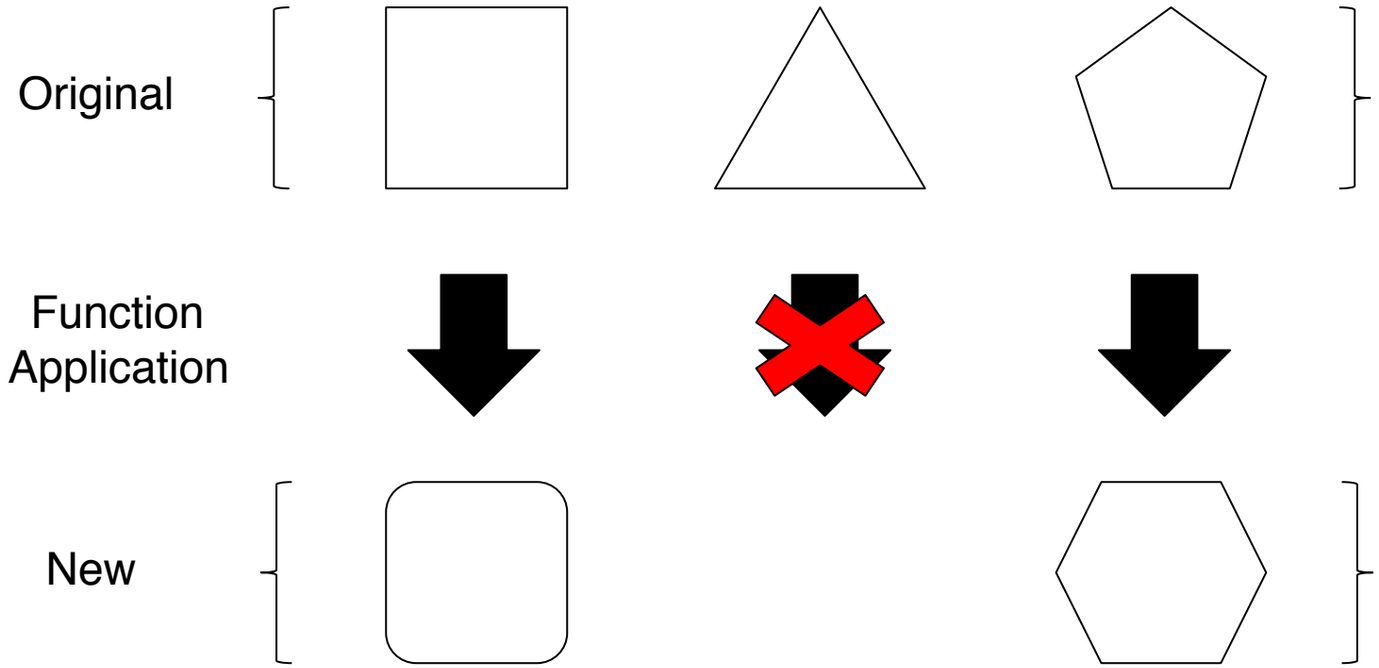
Transformational Mindset

- Three main transformation patterns:
  - Map
  - Filter
  - Reduce
- Throw for/if away
- MapReduce is inspired by map and reduce

Map

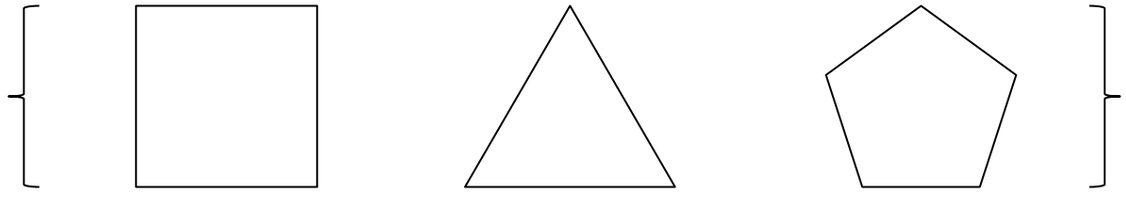


Filter

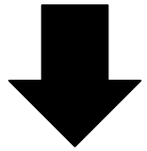


Reduce

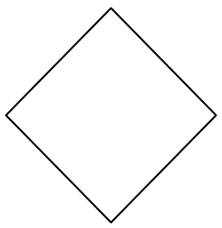
Original



Function
Application



New



Exercise

- Source: [1, 2, 3, 4]
- Map: * 2
- Filter: > 2
- Reduce: +

Java Stream

Stream

- A sequence of elements from a source that supports data processing operations
  - sequence of elements
  - source
  - data processing operations

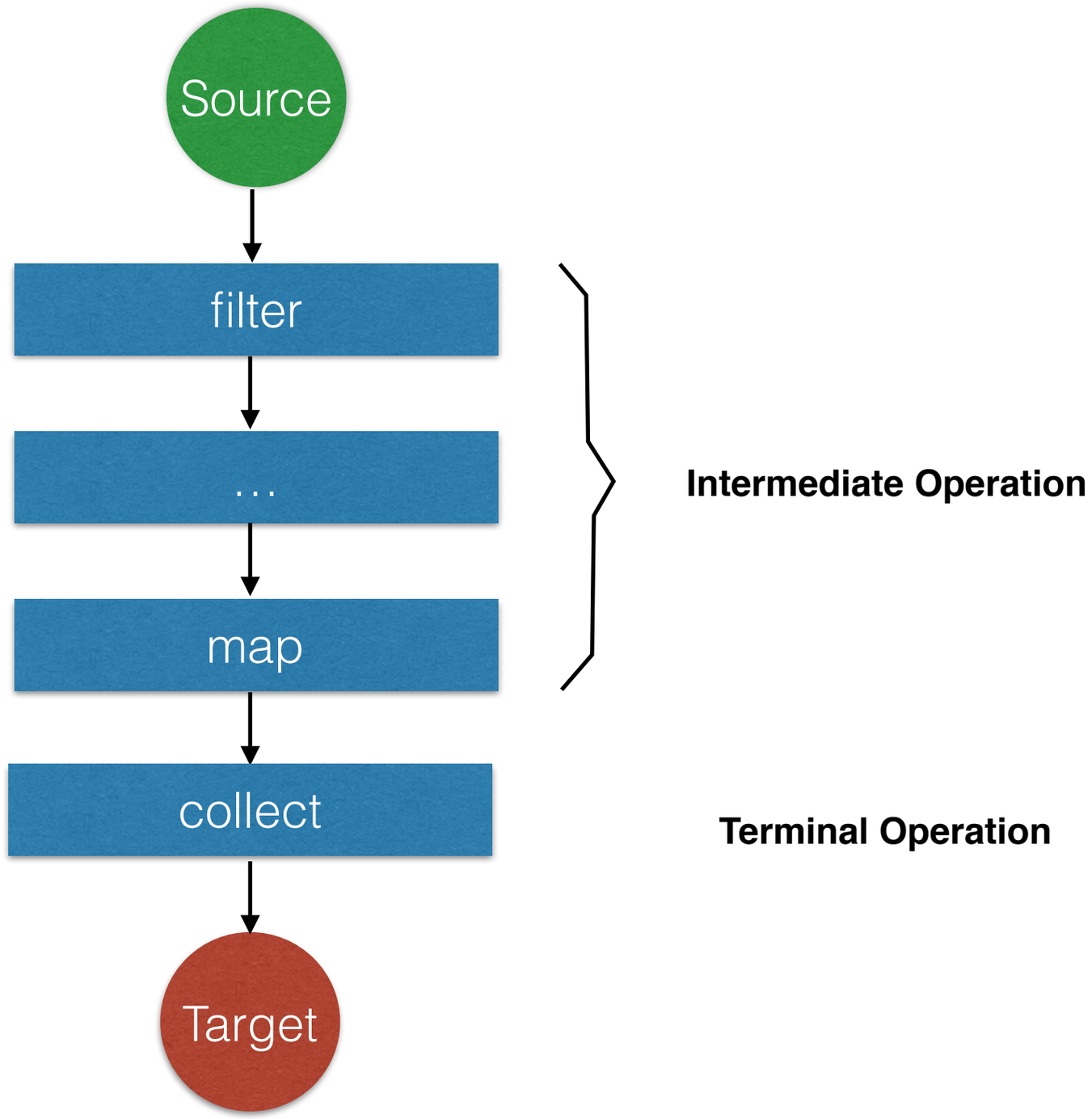
Clean Code (V)

```
people.stream()
      .filter(Person::isAgeQualified)
      .map(Person::getName)
      .collect(Collectors.toList());
```

Understanding Stream

- Declarative Programming
- External vs. Internal Iteration
- Iterate only once
- Stream operation
- Parallel

Stream Operation



Working with Stream

- A data source (such as a collection) to perform a query on
- A chain of intermediate operations that form a stream pipeline
- A terminal operation that executes the stream pipeline and produces a result

More about Transformational Mindset

- Convert computation into transformations.
- **What to do** over **How to do**
- DSL

More about Map

- Flattening streams: **flatMap**

More about Filter

- Filtering unique elements: **distinct**
- Truncating a stream: **limit**
- Skipping elements: **skip**

More about Filter: Find

- Find an element: **findAny**

- Find the first element: **findFirst**

More about Filter: Find

- Find an element: **findAny**
- Find the first element: **findFirst**

What's the difference?

More about Filter: Match

- Checking if one element matches: **matchAny**
- Checking if all elements match: **matchAll**
- Checking if no element matches: **noneMatch**

More about Reduce

- reduce/fold
- sum
- max/min

Return a list of all the unique characters for a list of words

```
["Hello", "World"]
```

```
["H", "e", "l", "o", "W", "r", "d"]
```

```
words.stream()
     .map(word -> word.split(""))
     .flatMap(Arrays::stream)
     .distinct()
     .collect(toList());
```

Parallel

Parallel Stream

- **parallelStream()**, create a possibly parallel stream with collection as its source
- **sequential()**, mark this stream as sequential
- **parallel()**, mark this stream as parallel

Parallel Stream Tips

- Profile your code
- Ensure your code work in parallel mode
- Use primitive stream for primitive

Spliterator

- Splittable iterator
- Try to split stream

Spliterator (Cont.)

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```

Numeric Stream

Numeric Stream

- No insidious boxing cost
- Better performance

Numeric Stream (Cont.)

- IntStream
- DoubleStream
- LongStream

Creating Numeric Stream

- of
- range
- rangeClosed

Numeric Stream (Cont.)

- sum
- average
- max/min
- boxed

Numeric Stream (Cont.)

- mapToInt
- mapToDouble
- mapToLong
- mapToObj

Collector

Collector Review

```
people.stream()
      .filter(Person::isAgeQualified)
      .map(Person::getName)
      .collect(Collectors.toList());
```

Collector

- As an advanced reducing
- **Collector** interface
- Predefined collector

Predefined Collector

- Defined in **Collectors**
- Major functionalities
  - Reducing and summarizing elements into a single value
  - Grouping elements
  - Partitioning elements

Reducing

- counting
- maxBy/minBy
- averagingInt/Double/Long
- summingInt/Double/Long

Summarizing

```
IntSummaryStatistics result = people.stream()
                                    .collect(summarizingInt(Person::getAge));
```

Output:

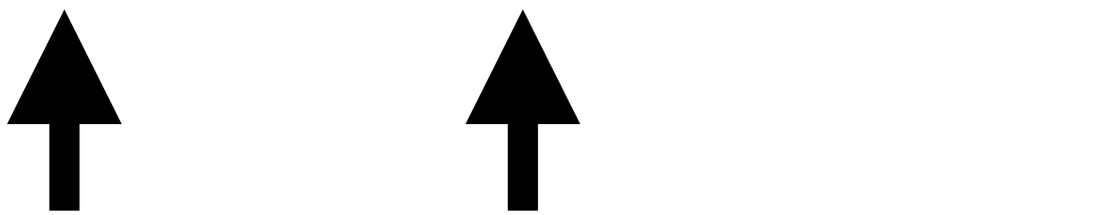
```
IntSummaryStatistics{count=2, sum=40, min=20, average=20.000000, max=20}
```

String joining

```
Person jack = new Person("Jack", 20, Gender.MALE);
Person rose = new Person("Rose", 20, Gender.FEMALE);
List<Person> people = asList(jack, rose);
String reult = people.stream()
                .map(Person::getName)
                .collect(joining(" "));
```

Real Reducing

```
reducing(0, Person::getAge, (m, n) -> m + n)
```



Initial Value **Mapper** **Operator**

Real Reducing

```
Integer result = people.stream()
        .collect(reducing(0, Person::getAge, (m, n) -> m + n));
```

Grouping

```
Map<Gender, List<Person>> result = people.stream()
                .collect(groupingBy(Person::getGender));
```

Multilevel Grouping

```
Map<Gender, Map<Integer, List<Person>>> result = people.stream()
    .collect(groupingBy(Person::getGender, groupingBy(Person::getAge)));
```

Collecting Data in Subgroups

```
Map<Gender, Long> result = people.stream()
    .collect(groupingBy(Person::getGender, counting()));
```

Partitioning

- A special case for grouping
- **partitioningBy**
- A partitioning function is a predicate

Partitioning Case

```
Map<Boolean, List<Person>> result = people.stream()
                                    .collect(partitioningBy(Person::isMale));
```

Collect in Partition

```
Map<Boolean, Long> result = people.stream()
    .collect(partitioningBy(Person::isMale, counting()));
```

Collector

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```

Understanding Collector

- **supplier()** - making a new result container
- **accumulator()** - adding an element to result container
- **finisher()** - applying the final transformation to result container
- **combiner()** - merging two result container
- **characteristics()** - defining collector behaviour

Write your own collector only for performance

Immutability

Bad Case: SimpleDateFormat

```
class Sample {
    private static final DateFormat format = new SimpleDateFormat("yyyy.MM.dd");

    public String getCurrentDateText() {
        return format.format(new Date());
    }
}
```

Bad Case: SimpleDateFormat (Cont.)

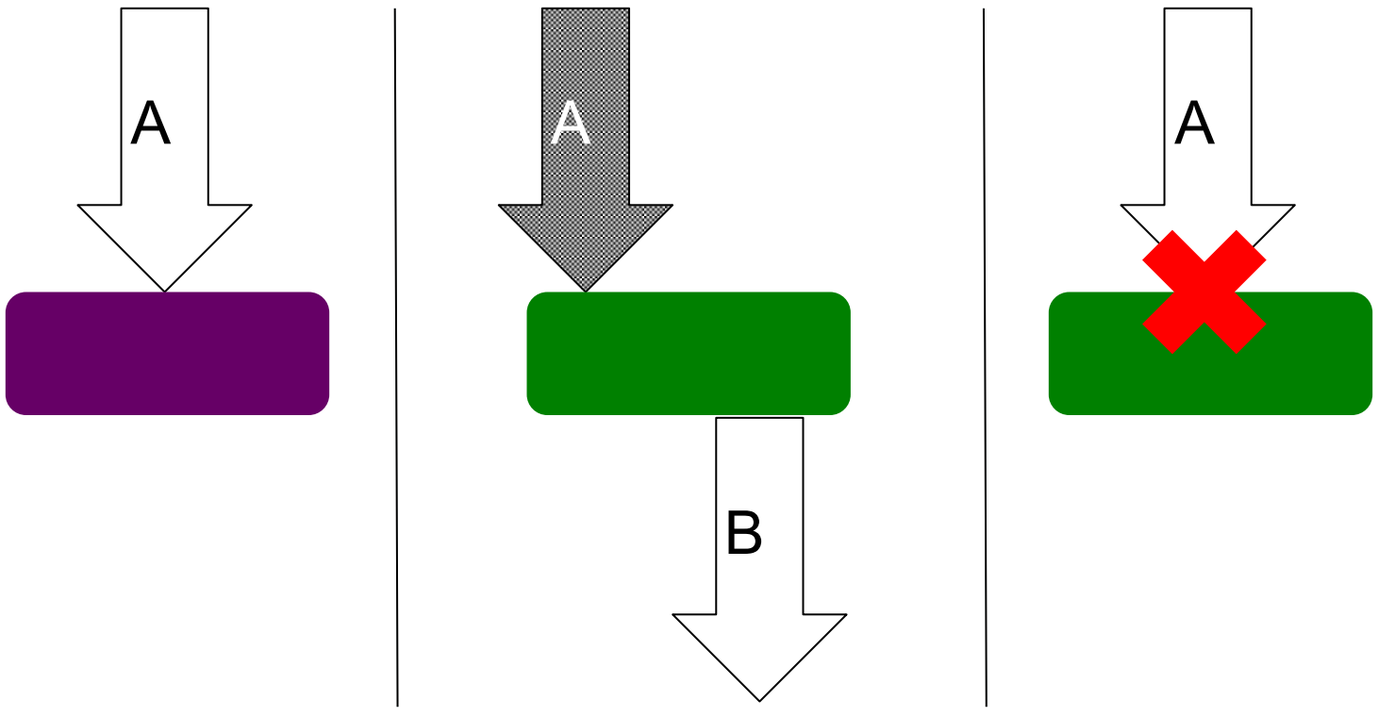
```
public class Sample2 {
    public String getCurrentDateText() {
        DateFormat format = new SimpleDateFormat("yyyy.MM.dd");
        return format.format(new Date());
    }
}
```

Bad Case: SimpleDateFormat (Cont. II)

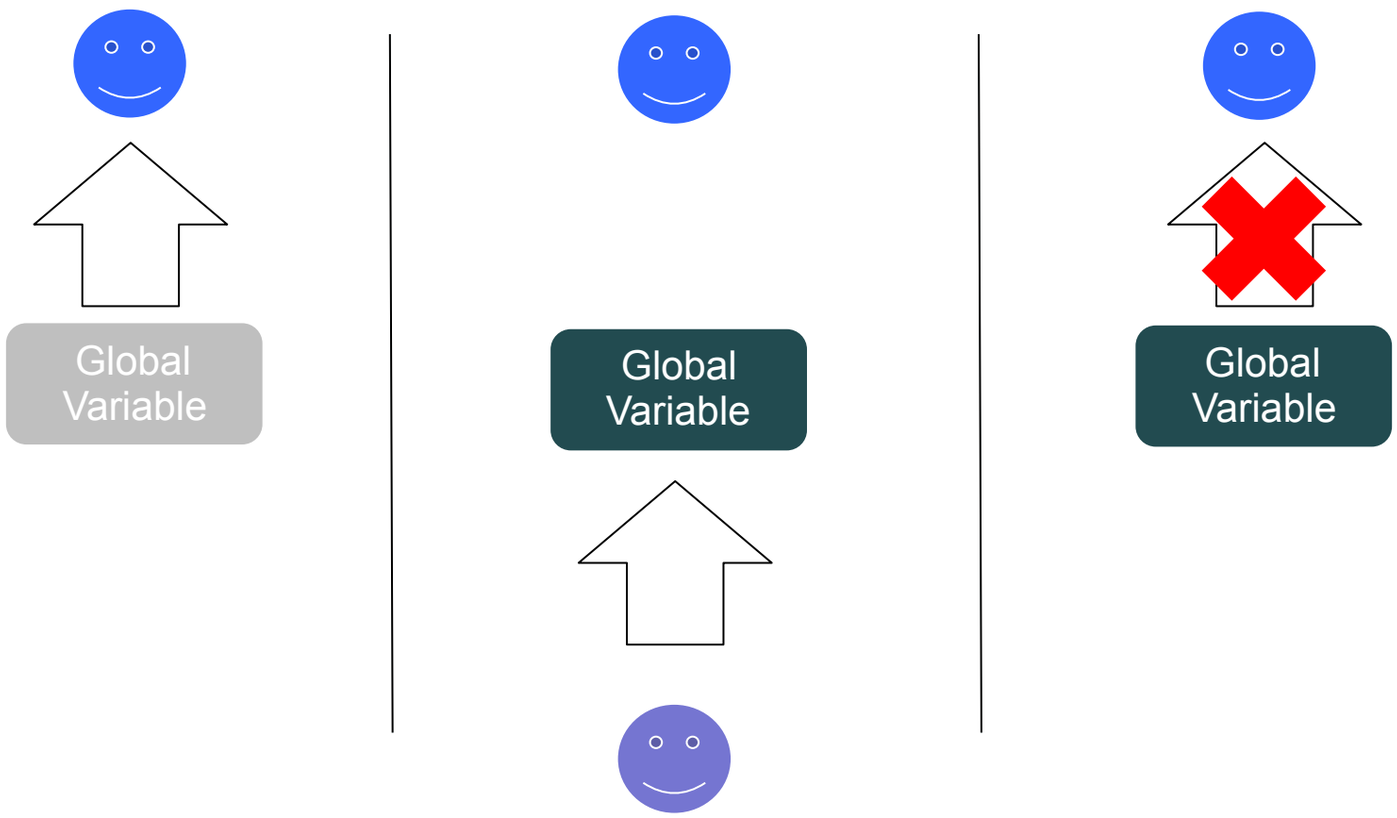
- Reason in SimpleDateFormat implementation

```
calendar.setTime(date);
```

What's Wrong?



The Same Situation with Global Variables

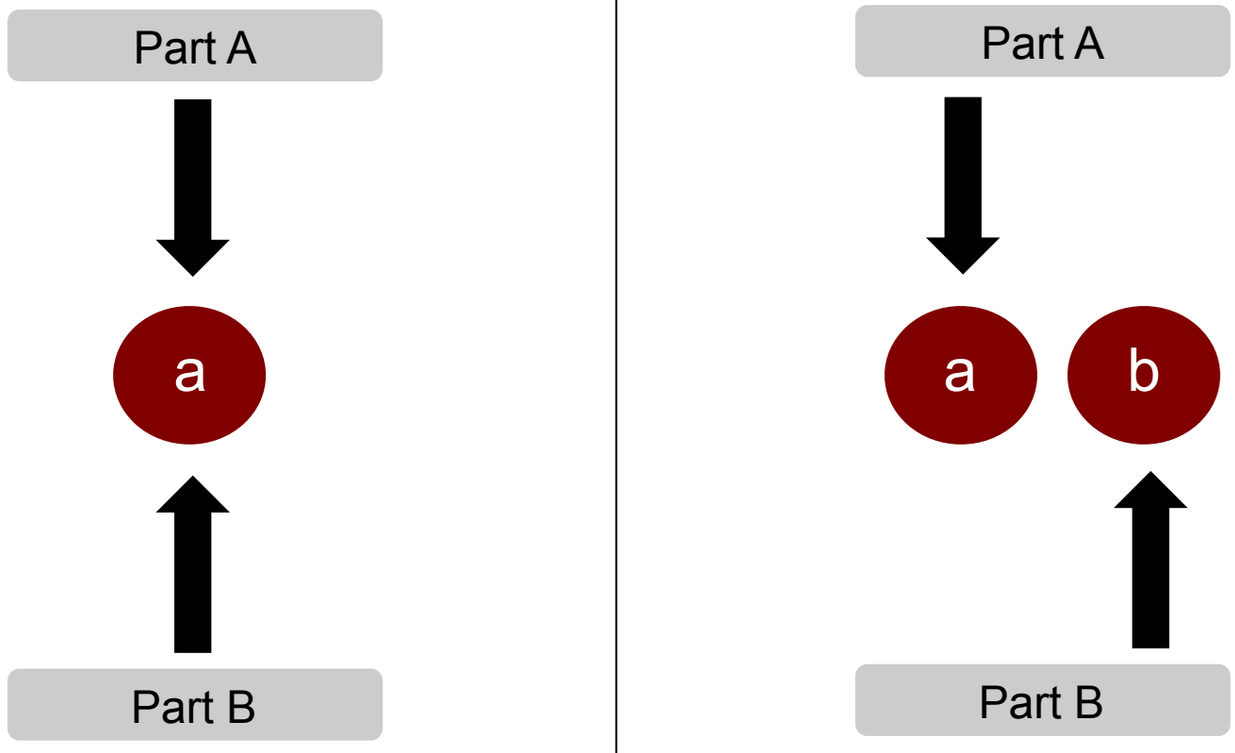


All About Mutability

When Everything Is Mutable

- Lock
- Synchronization
- Difficult-to-find bug
- …

Real Concern: Value, Not Variable



When We Talk About Immutable…

- Referential Transparency
- Pure
- Stateless
- No Side-Effect

Immutable Java

- Prefer **final**
  - **final** class
  - **final** method
  - **final** field
  - **final** argument
  - **final** local “variable”
- Write pure function

Immutable Class

- Pure function
    - which does not changing any field
- Immutable Class
    - initialize fields only in constructor
    - all methods should be pure functions
    - if needed, method should return new object rather than changing **this**
- Example: String

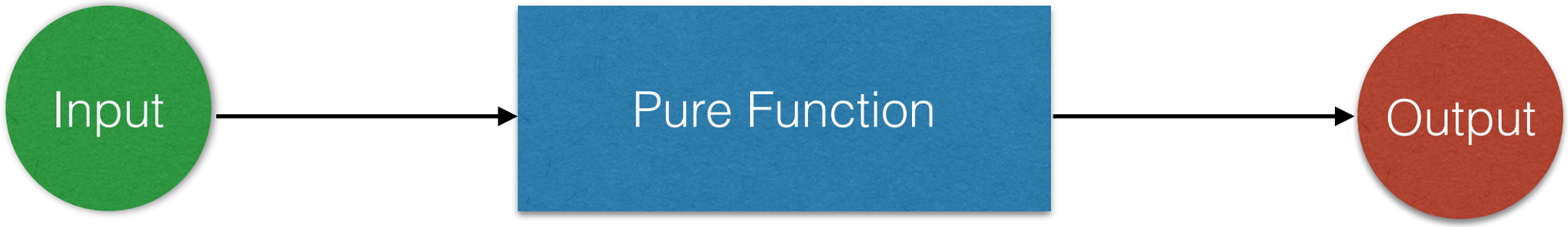
Pragmatic: Use Local Variable in a Method

```
public String replace(char oldChar, char newChar) {
    if (oldChar != newChar) {
        int len = value.length;
        int i = -1;
        char[] val = value; /* avoid getfield opcode */

        while (++i < len) {
            if (val[i] == oldChar) {
                break;
            }
        }
        if (i < len) {
            char buf[] = new char[len];
            for (int j = 0; j < i; j++) {
                buf[j] = val[j];
            }
            while (i < len) {
                char c = val[i];
                buf[i] = (c == oldChar) ? newChar : c;
                i++;
            }
            return new String(buf, true);
        }
    }
    return this;
}
```

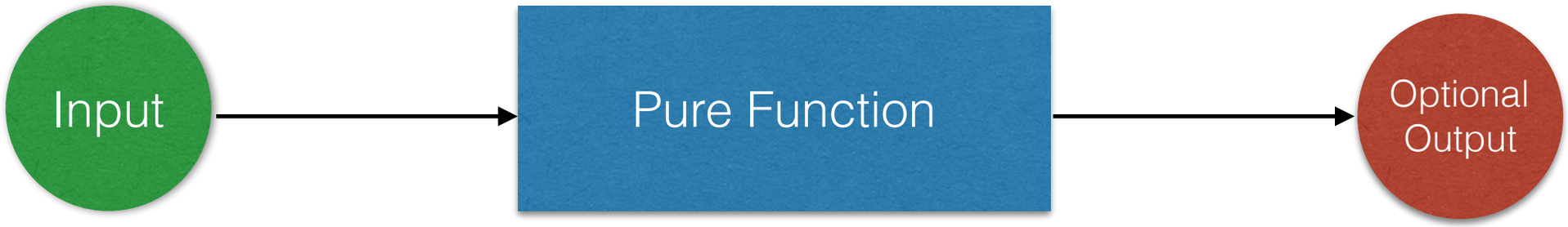
Pure Function

- In computer programming, a function may be described as pure if both these statements about the function hold:
    - The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change as program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices.
    - Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.

Pure Function (Cont.)



Exceptional Pure Function



John Carmak on Functional Programming

- http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/
- On pure function
  - This is an abstraction of course; every function has side effects at the CPU level, and most at the heap level, but the abstraction is still valuable.
- Pure function nice properties
  - Thread Safety
  - Reusability
  - Testability
  - Understandability and maintainability

“OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts..”

–Michael Feathers

What's the Result?

```
Person jack = new Person("Jack", 20, Gender.MALE);
Person rose = new Person("Rose", 20, Gender.FEMALE);
List<Person> people = asList(jack, rose);
```

```
Function<Person, String> function = mock(Function.class);
when(function.apply(jack)).thenReturn("Jack");

people.stream().map(function);

verify(function).apply(jack);
```

The Answer Is …

FAILURE

But The Question Is …

WHY

The Real Answer Is …

LAZY

Lazy Evaluation

“In programming language theory, lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing). The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.”

–Wikipedia

Java Lazy Evaluation

- Proxy Pattern
- Singleton Pattern

```
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }
}
```

“All problems in computer science can be solved by another level of indirection.”

–David Wheeler

The Benefits of Lazy Evaluation

- Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions

- The ability to construct potentially infinite data structures

- The ability to define control flow (structures) as abstractions instead of primitives

Memorization

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

–Wikipedia

Memoization in Java

- Guava Suppliers
  - memoize
  - memoizeWithExpiration
- Map.computeIfAbsent

Infinite Stream

```
Stream.iterate(0, number -> number + 1)
        .skip(2)
        .limit(3)
        .forEach(System.out::println);
```

Infinite Stream (Cont.)

```
Stream.generate(Math::random)
        .skip(2)
        .limit(3)
        .forEach(System.out::println);
```

Implement an infinite Fabonacci stream

An Infinite Fabonacci Stream

```
Stream.iterate(new int[]{0, 1},
        t -> new int[]{t[1], t[0] + t[1]})
    .map(t -> t[0]);
```

Optional

Requirement

Get a person’s country of birth.

Traditional Java

```
public Country getBirthCountry() {
    return person.getBirthPlace()
                 .getCity()
                 .getProvince()
                 .getCountry();
}
```

Traditional Java

```
public Country getBirthCountry() {
    return person.getBirthPlace()
                 .getCity()
                 .getProvince()
                 .getCountry();
}
```

No, It's Wrong

Traditional Java

```
public Country getBirthCountry() {
    Place place = person.getBirthPlace();
    if (place != null) {
        City city = place.getCity();
        if (city != null) {
            Province province = city.getProvince();
            if (province != null) {
                return province.getCountry();
            }
        }
    }

    return null;
}
```

Why Null check is always missing?

“Null sucks.”

–Doug Lea

“I call it my billion-dollar mistake..”

–Sir C. A. R. Hoare, on his invention of the null reference

Optional

- A container object which may or may not contain a non-null value.

- Null replacement

Creating a Optional

- **of()** - with non-null value
- **empty()** - empty Optional instance
- **ofNullable()** - value may be null

Dereference Optional

isPresent is required before get

```
if (country.isPresent()) {
    return country.get();
}
```

Dereference Optional (Cont.)

Default Value

```
country.orElse(china);
```

Dereference Optional (Cont.)

Default Supplier

```
country.orElseGet(Country::new);
```

Dereference Optional (Cont.)

Default to Throw Exception

```
country.orElseThrow(IllegalArgumentException::new);
```

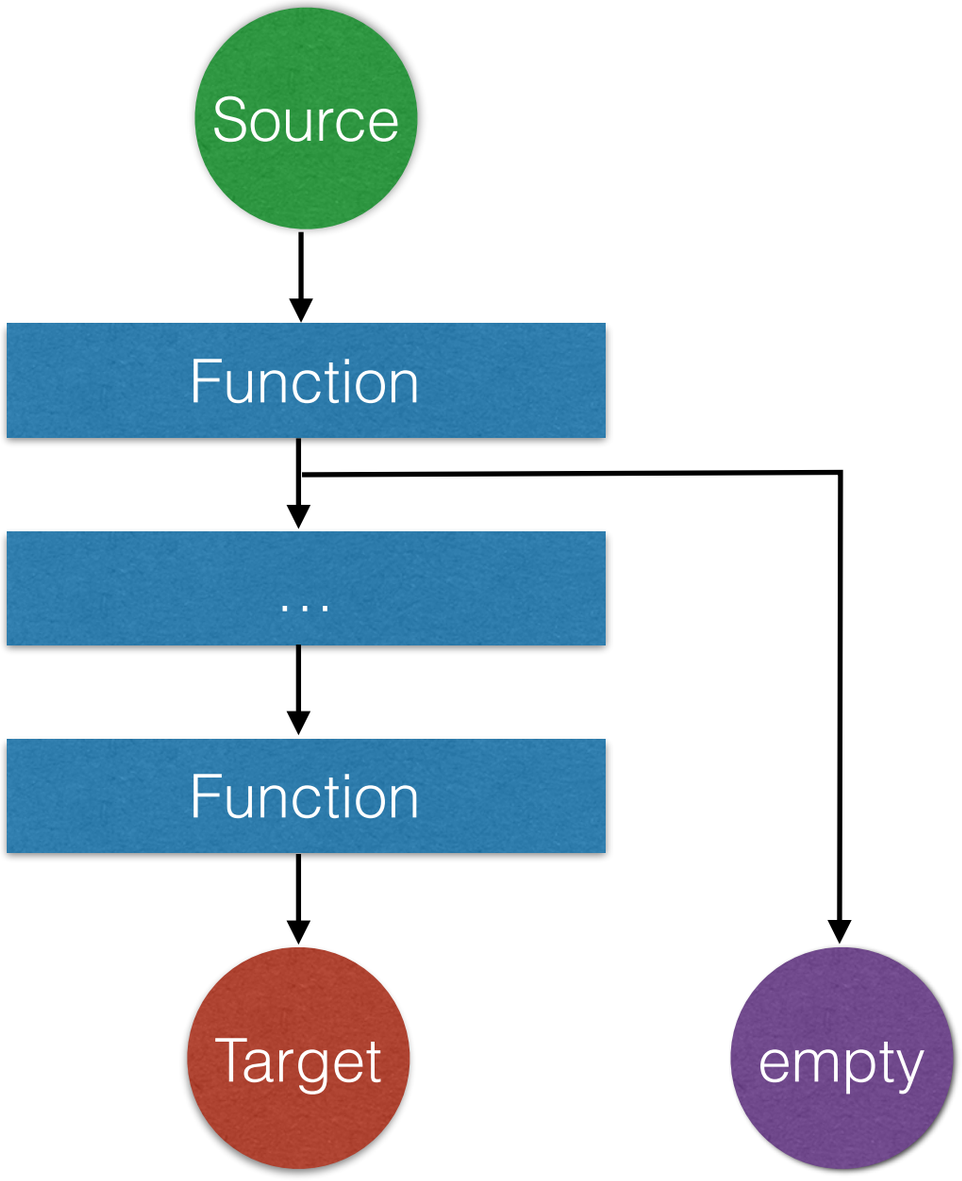
Extract Value from Optional

- **map()** - extract plain value
- **flatMap()** - extract Optional value

Reject with filter

- **filter()**
  - empty, if it is empty
  - empty, if predicate returns false
  - this, if predicate returns true

Chain Map/Filter



Clean Code

```
public Optional<Country> getBirthCountry() {
    return Optional.ofNullable(this.birthPlace)
                   .flatMap(Place::getCity)
                   .flatMap(City::getProvince)
                   .flatMap(Province::getCountry);
}
```

Optional is Maybe Monad

Functional Thinking