Booting into FP

Dhaval Dalal
@softwareartisan
https://dhavaldalal.wordpress.com

14th Nov 2019

Touted Advantages

Touted Advantages

- Less number of lines of code as compared to imperative code.
    - Hence, Functional Programmers are more “productive”

Touted Advantages

- Less number of lines of code as compared to imperative code.
    - Hence, Functional Programmers are more “productive”
- Declarative
    - Describes “what to do”, rather than imperative - “how to do”

Touted Advantages

- Less number of lines of code as compared to imperative code.
    - Hence, Functional Programmers are more “productive”
- Declarative
    - Describes “what to do”, rather than imperative - “how to do”
- Immutability
    - No assignment statements, a variable value never changes.

Touted Advantages

- Less number of lines of code as compared to imperative code.
    - Hence, Functional Programmers are more “productive”
- Declarative
    - Describes “what to do”, rather than imperative - “how to do”
- Immutability
    - No assignment statements, a variable value never changes.
- No side-effects
    - Eliminates major source of bugs, order of evaluation does not matter.

Touted Advantages

- Less number of lines of code as compared to imperative code.
    - Hence, Functional Programmers are more “productive”
- Declarative
    - Describes “what to do”, rather than imperative - “how to do”
- Immutability
    - No assignment statements, a variable value never changes.
- No side-effects
    - Eliminates major source of bugs, order of evaluation does not matter.
- Referential Transparency
    - Enables equational reasoning

Source: Why Functional Programming Matters - John Hughes

Why FP really matters?

Why FP really matters?

- Its about Modularity
  - “Our ability to decompose a problem into parts depends directly on our ability to glue solutions” - John Hughes

Why FP really matters?

- Its about Modularity
  - “Our ability to decompose a problem into parts depends directly on our ability to glue solutions” - John Hughes
- Benefits
  - Small can be coded quickly and easily.
  - Reusability of general-purpose modules.
  - Independent testing of modules.

Why FP really matters?

- Its about Modularity
  - “Our ability to decompose a problem into parts depends directly on our ability to glue solutions” - John Hughes
- Benefits
  - Small can be coded quickly and easily.
  - Reusability of general-purpose modules.
  - Independent testing of modules.
- The Glue
  - Lazy Evaluation.
  - Higher-Order Functions.

Source: Why Functional Programming Matters - John Hughes

Functions Everywhere!

Functions Everywhere!

- Function is a smallest unit of computation.

Functions Everywhere!

- Function is a smallest unit of computation.

$$f(x) = x^2 \quad \text{OR} \quad f : x \mapsto x^2$$

Functions Everywhere!

- Function is a smallest unit of computation.

$$f(x) = x^2 \quad \text{OR} \quad f : x \mapsto x^2$$

- Domain, Co-Domain and Range of a function.

  - Domain {-3, -2, -1, 0, 1, 2, 3}

  - Co-Domain {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Functions Everywhere!

- Function is a smallest unit of computation.

$$f(x) = x^2 \quad \text{OR} \quad f : x \mapsto x^2$$

- Domain, Co-Domain and Range of a function.
  - Domain {-3, -2, -1, 0, 1, 2, 3}
  - Co-Domain {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Range {0, 1, 4, 9}

Functions Everywhere!

- Function is a smallest unit of computation.

$$f(x) = x^2 \quad \text{OR} \quad f : x \mapsto x^2$$

- Domain, Co-Domain and Range of a function.
  - Domain {-3, -2, -1, 0, 1, 2, 3}
  - Co-Domain {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Range {0, 1, 4, 9}
- Function is a relation that maps the domain into co-domain such that there is exactly one input maps to a output (1:1 and not 1:n).

Functions Everywhere!

- Function is a smallest unit of computation.

$$f(x) = x^2 \quad \text{OR} \quad f : x \mapsto x^2$$

- Domain, Co-Domain and Range of a function.
  - Domain {-3, -2, -1, 0, 1, 2, 3}
  - Co-Domain {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
  - Range {0, 1, 4, 9}
- Function is a relation that maps the domain into co-domain such that there is exactly one input maps to a output (1:1 and not 1:n).
- Type of the function is $int \mapsto int$

A Pure Function

- Uses nothing other than i/p parameters (and its definition) to produce o/p $f : T \mapsto R$

- Neither modifies input arguments nor reads/modifies external state

- Call is substitutable by its body. To understand the code, you don't have to look elsewhere.

```
class Calculator {
 // (int, int) -> int
 public int add(final int x, final int y) {
    return x + y;
  }
}
```

A Side-effecting Function

- Modifies or interacts with things outside of its scope (interaction with outside world), and may also return a value.

  - Reading/Writing to disk, DB, Socket etc… anything that modifies external world.

```
class Calculator {
  private int memory = 0;

  public Calculator(final int memory) {
    this.memory = memory;
  }
  public Integer add(final int x, final int y) { … }

  public Integer memoryPlus(final int n) {
    memory = add(memory, n);
     return memory;
  }
}
```

Outside of its scope

Black-Hole like Side-effecting Function

- Write Side-effect…a typical Setter! $f : T \mapsto ()$

```
class Calculator {
  private int memory = 0;

  public Calculator(final int memory) {
    this.memory = memory;
  }
  // int -> ()
  public void setMemory(final int memory) {
     this.memory = memory;
 }
}
```

Mother's Love like Side-effecting Function

- Read Side-effect…a Getter! $f : () \mapsto R$

```
class Calculator {
  private int memory = 0;

  public Calculator(final int memory) {
    this.memory = memory;
  }
  // () -> int
  public int recallMemory() {
     return memory;
 }
}
```

Politician like Side-effecting Function

- Read-Write Side-effect…a Printer! $f : () \mapsto ()$
- Neither consume anything, nor I return anything, because I do everything under the table.

```
class Calculator {
  private int memory = 0;

  public Calculator(final int memory) {
    this.memory = memory;
  }

  public void printMemory() {
    System.out.println(memory);
  }
}
```

Side-effects or Useful-effects?

Source: https://channel9.msdn.com/Blogs/Charles/Simon-Peyton-Jones-Towards-a-Programming-Language-Nirvana

Side-effects or Useful-effects?

Source: https://channel9.msdn.com/Blogs/Charles/Simon-Peyton-Jones-Towards-a-Programming-Language-Nirvana

Understanding Referential Transparency

```
class Calculator {

  private int memory;

  public Calculator(final int memory) {
    this.memory = memory;
  }

  public Integer add(final int x, final int y) {
    return x + y;
  }

  public Integer memoryPlus(final int n) {
    memory = add(memory, n);
    return memory;
  }
}
```

Understanding Referential Transparency

```
Calculator c = new Calculator();
```

Time

```
c.add(2, 3); // 5
c.add(2, 4); // 6
c.add(2, 3); // 5
c.add(2, 4); // 6
```

Referentially Transparent add :

1. Substitute any expression with its resulting value.
2. Returns same results all the time, as behaviour does not depend on history.

Time

```
c.memoryPlus(2); // 2
c.memoryPlus(3); // 5
c.memoryPlus(2); // 7
c.memoryPlus(3); // 10
```

Referentially Opaque memoryPlus :

1. Cannot replace it with resulting value.
2. Returns different results as time progresses, as behaviour depends on history.

Pure Functions, Side-Effects and Evaluation Order

```
Calculator c = new Calculator();
```

Time

```
c.add(2, 3); // 5
c.add(2, 4); // 6
c.add(2, 3); // 5
c.add(2, 4); // 6
```

Out of order execution possible

Time

```
c.memoryPlus(2); // 2
c.memoryPlus(3); // 5
c.memoryPlus(2); // 7
c.memoryPlus(3); // 10
```

Out-of-order execution not Possible

How can we make memoryPlus Referentially Transparent?

Ref. Transparent memoryPlus

Make memory Immutable

```
class Calculator {

  private final int memory;

  public Calculator(final int memory) {
    this.memory = memory;
  }

  public int add { … }

  public Calculator memoryPlus(final int n) {
    return new Calculator(add(memory, n));
  }
}
```

Return new instance from operation.

Reflections

- Referential Transparency is about replacing any expression (or function) with its resulting value.

- To be referentially transparent, function will require to work with immutable data.

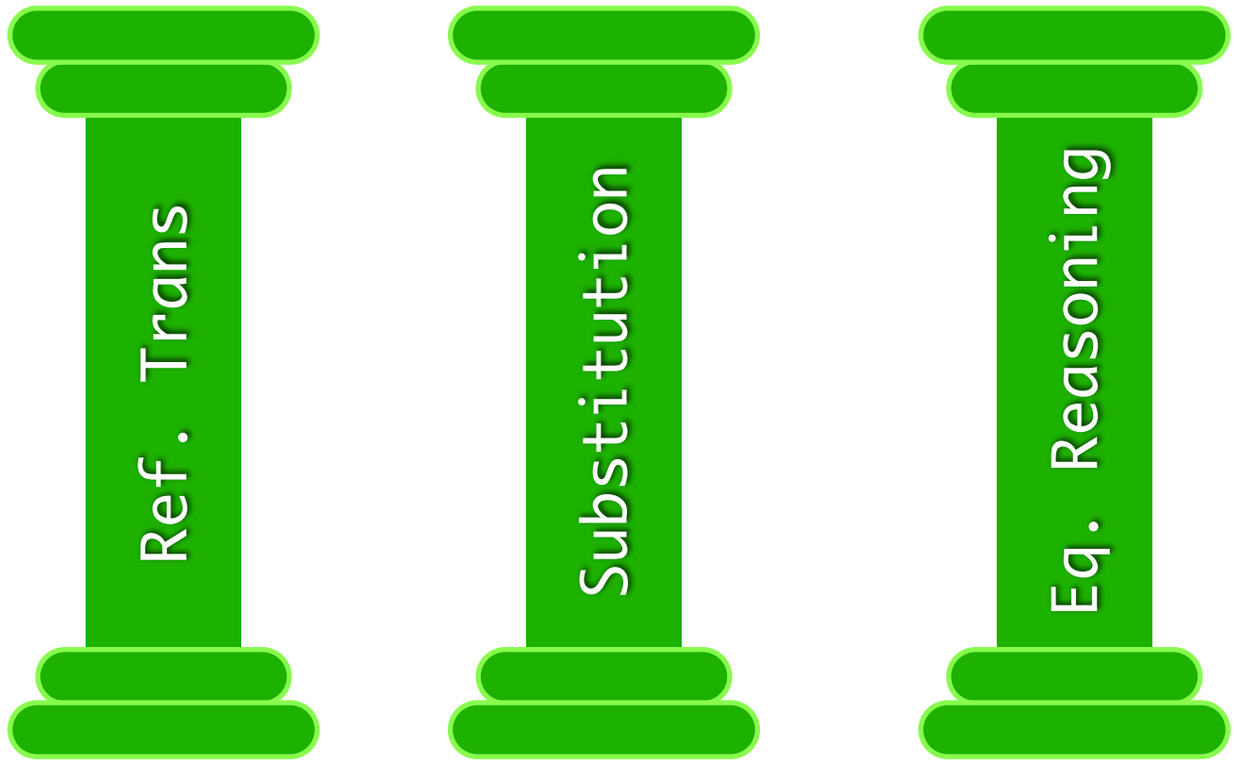
- To be referentially transparent, a function will require to be pure.

- To be referentially transparent, a function will require to be deterministic.

- Referentially transparent functions are context-free and the context is time.

https://github.com/CodeJugalbandi/FunctionalProgramming/tree/master/melodies/referentialTransparency

3 Pillars of FP



- Referential Transparency - enforces invariant expressions or functions are pure.

- Substitution Model - RT enables simple and natural mode of reasoning about program evaluation.

- Equational Reasoning - Computation proceeds like solving an algebraic equation, where we substitute and reduce to its simplest form.

Source: Functional and Reactive Domain Modelling, Debashish Ghosh

Good Deeds Happen Anonymously

- Naming is one of the hardest problem in software.
- Alleviate using an anonymous function - Lambda λ.

Drop all inessentials

```
public int add(final int x, final int y) {
  return x + y;
}
```

```
public int add(final int x, final int y) {
  return x + y;
}
```

what remains are essentials, parameters and body

```
(final int x, final int y) -> { x + y; }
```

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

SAMs Become...

Drop all inessentials

```
new Button().addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.print("essence");
  }
});
```

```
new Button().addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("essence");
  }
});
```

```
new Button().addActionListener(e -> System.out.print("essence"));
```

what remains are essentials, parameters and body

OR

```
new Button().addActionListener(System.out::print);
```

Lambda

- Compiled as interface implementation with synthesised method having signature of the abstract method in that interface.
  - An interface with single abstract method (SAM).
  - @FunctionalInterface - Though optional annotation, its better to have it so that it ensures that it stays as a lambda and not become something more.

```
Function<Integer, Integer> twice = x -> 2 * x;

twice.apply(3); // 6
```

Instead of `twice(3);`

It would be have been nice if some more syntactic sugar was added by Java8 to make this happen

Functional Interfaces

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
    …
}
Function<Float, Float> twice = x -> 2 * x;
twice.apply(3); // 6

// A function returns its argument unchanged
Function<Float, Float> identity = x -> x;

@FunctionalInterface
public interface BiFunction<T, U, R> {
  R apply(T t, U u);
    …
}
BiFunction<Float, Float, Float> add = (x, y) -> x + y;
add.apply(2, 3); // 5
```

A Black-Hole like Functional Interface

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
  …
}

corrupt.accept(100.34);
@FunctionalInterface
public interface BiConsumer<T, U> {
  void accept(T t, U u);
  …
}
BiConsumer<Celestial, Celestial> blackHole =
                                  (planet, asteroid) -> { }
blackHole.accept(planet, asteriod);
```

A Mother's Love like Functional Interface

```java
@FunctionalInterface
public interface Supplier<T> {
   T get();
}

Supplier<String> mother = () -> "Love";

mother.get(); // Love
```

Things Emerge without consuming anything

Mr. Spock like Functional Interface

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);
  …
}

Predicate<T> affirmative = x -> true;
affirmative.test(1); // true

Predicate<T> negative = x -> false;
negative.test(1); // false

Predicate<Integer> isEven = x -> x % 2 == 0;
isEven.test(2); // true
isEven.test(3); // false
```

Politician like Functional Interface

- Neither consume anything, nor I return anything, because I do everything under the table.

```
@FunctionalInterface
public interface Runnable {
  void run();
}
```

Every“thing” is a Lambda

- Like Object, a Function is also a thing, so
  - Pass a function to a function
  - Return a function from within a function
- A function that produces or consumes a function is called as Higher Order Function - HOF
  - Either pass existing method reference (static or instance) or write an in-place lambda where a method expects a function parameter.
- Modularization - Build abstractions with HOFs.

Pass a function to a function

- Subsumed ‘for’ loop.
  - Repetitive behaviour using Recursion.

No need for explicit looping constructs! Just a function!

```
void iterate(int times, Runnable body) {
  if (times <= 0) {
    return;
  }
  body.run();
  iterate(times - 1, body);
}

iterate(2, () -> System.out.println("Hello"));
// Hello
// Hello
```

Return a function from a function

- Subsumed Factory Method.

```
Function<Double, Double> power(double raiseTo) {
   return x -> Math.pow(x, raiseTo);
}

Function<Double, Double> square = power(2.0);
square.apply(2.0); // 4.0

Function<Double, Double> cube = power(3.0);
cube.apply(2.0);  // 8.0
```

Subsumed Aspect

- AOP - Around style

```
public<T, R> Function<T, R> time(Function<T, R> fn) {
  return t -> {
    long startTime = System.currentTimeMillis();
    R result = fn.apply(t);
    long timeTaken = System.currentTimeMillis() - startTime;
    System.out.println("Time: " + timeTaken + " ms");
    return result;
  }
}
```

Composition Pipeline

- Compose a function from other functions by aligning types.

```
// T -> U
Function<T, U> f;

// U -> R
Function<U, R> g;

// T -> U . U -> R = T -> R
Function<T, R> h = f . g;
```

- Function composition is not commutative.
  - $f \circ g \neq g \circ f$

Why Composition?

- Tackle complexity by composing behaviors.
- Compose larger functions from smaller ones.
- Smaller functions can be tested independently.
- If the parts are correct, we can then trust the correctness of the whole.
- Every part of the larger function can be reasoned about easily.

Composing Functions

- Using compose & andThen

```
Function<Integer, Integer> square = x -> x * x;

Function<Integer, Integer> twice  = x -> 2 * x;


Read Left to Right ——————→
square.compose(twice).apply(2); // 16
     ←—————————— Think Right to Left


Read Left to Right ——————→
square.andThen(twice).apply(2); // 8
Think Left to Right ——————→
```

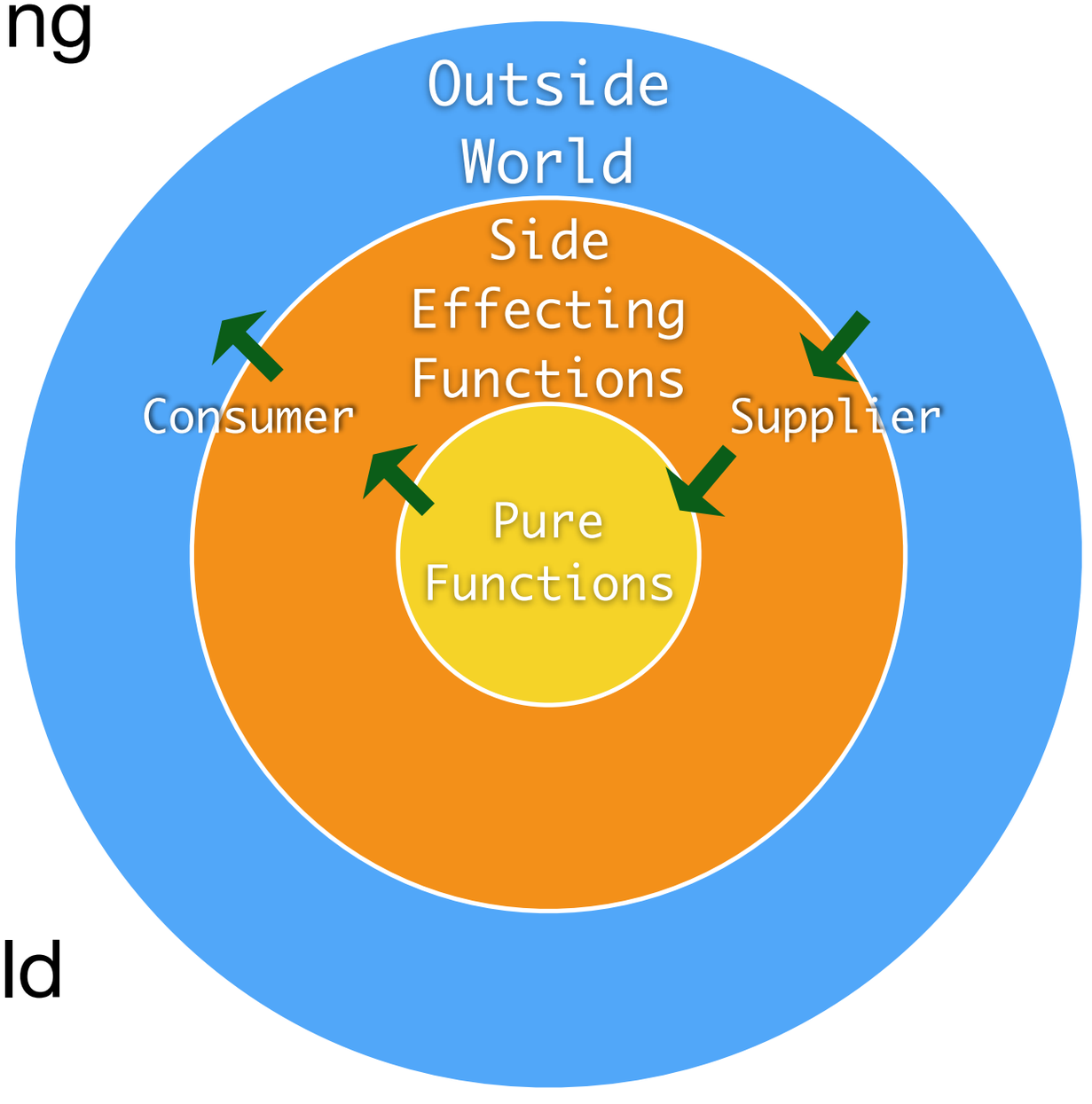
- For languages that support function composition, look for a way to go with the grain of thought. In Java, prefer using andThen

Circle of Purity

- Compose behaviours using pure Functions.

- Data flows through composed pipes.

- Interact with outside world using Side-effecting functions.



Courtesy: Venkat

Reflections

- Enforce order of evaluation.
  - In imperative programming, statements enforce order of evaluation.
  - In FP, the order of composition determines the order of evaluation.
- Function composition (and not function application) is the default way to build sub-routines in Concatenative Programming Style, a.k.a Point Free Style.
- Functions neither contain argument types nor names, they are just laid out as computation pipeline.
- Lot of our domain code is just trying to do this!
- Makes code more succinct and readable.

https://github.com/CodeJugalbandi/FunctionalProgramming/tree/master/melodies/sequencing

Eager Evaluation

- Java uses eager evaluation for method arguments.

- Args evaluated before passing to the method.

```
public int first(int x, int y) {
  System.out.println("Inside First...");
  return x;
}

public int loop() {
  System.out.println("Inside Loop...");
  return loop();
}

System.out.println(first(10, 20));
// Inside First…
// 10
System.out.println(first(10, loop())); // Stackoverflow
System.out.println(first(loop(), 20)); // Stackoverflow
```

Lazy Evaluation

- Lazy means don’t compute until there exists a demand.
- Haskell OTOH is lazy by default.
- Args not evaluated before passing to the method, instead evaluated upon use.

```haskell
loop = loop

first x y = x

main :: IO ()
main = do
  print $ first 10 20   -- 10
  print $ first 10 loop -- 10
  print $ first loop 10 -- Infinite Loop
```

Lazy Evaluation

- Scala is eager by default like Java, and provides a “lazy” switch.

```
def loop: Int = loop

def first(x: Int, y: Int) = x

first(10, 20)   // 10
first(loop, 20) // Non-termination
first(10, loop) // Non-termination

scala> lazy val x = loop
x: Int = <lazy>

scala> x  // Non-termination
```

Simulating Lazy Evaluation

- To delay the evaluation of args (lazy), wrap them in lambda.
  - Call the lambda when we need to evaluate.
  - Immutability and Purity makes lazy evaluation possible.

```
public static int first(Supplier<Integer> x, Supplier<Integer> y) {
  System.out.println("Inside First...");
  return x.get();
}
public static int loop() {
  System.out.println("Inside Loop...");
  return loop();
}

System.out.println(first(10, 20));                    // 10
System.out.println(first(() -> 10, () -> loop())); // 10
System.out.println(first(() -> loop(), () -> 20)); // Stackoverflow
```

Representation of computation and not the computation itself.

Imperative Filtering and Collecting

```
String sentence = "all mimsy were the borogoves and the momeraths";
String [] words = sentence.split(" ");
StringBuilder caps = new StringBuilder();

for (var word : words) {
  if (word.length() < 4) {
    caps.append(word.toUpperCase());
    caps.append(" ");
  }
}

String capitalized = caps.toString().trim();
System.out.println(capitalized); // ALL THE AND THE
```

Collector

Enumeration and Filtering interleaved

- One of the motivations of Streams is to make parallelism more accessible to developers.

Streams (Lazy List)

```
String sentence = "all mimsy were the borogoves and the momeraths";
String capitalized = Stream.of(sentence.split(" "))
    .filter(word -> word.length() < 4)
    .map(String::toUpperCase)
    .collect(Collectors.joining(" "));

System.out.println(capitalized);
```

- One Element at a time passes through the pipeline

```
String capitalized = Stream.of(sentence.split(" "))
    .peek(word -> System.out.println("Filter = " + word))
    .filter(word -> word.length() < 4)
    .peek(word -> System.out.println("Map = " + word))
    .map(String::toUpperCase)
    .collect(Collectors.joining(" "));
System.out.println(capitalized);
```

How can we improve this?

```
public static String capitalize(String s) {
  try {
    Thread.sleep(2000);
  } catch (Exception e) {
  }
  return s.toUpperCase();
}

String sentence = "all mimsy were the borogoves and the momeraths";
long startTime = System.currentTimeMillis();
String capitalized = Stream.of(sentence.split(" "))
  .filter(word -> word.length() < 4)
  .map(word -> capitalize(word))
  .collect(Collectors.joining(" "));
long timeTaken = System.currentTimeMillis() - startTime;

System.out.println(String.format("%s, Took %d(ms)", capitalized, timeTaken));
// ALL THE AND THE, Took 8043(ms)
```

Parallelizing

- One of the motivations of Streams is to make parallelism more accessible to developers.

```
public static String capitalize(String s) {
  try {
    Thread.sleep(2000);
  } catch (Exception e) {
  }
  return s.toUpperCase();
}

String sentence = "all mimsy were the borogoves and the momeraths";
long startTime = System.currentTimeMillis();
String capitalized = Stream.of(sentence.split(" “))
  .parallel()
  .filter(word -> word.length() < 4)
  .map(word -> capitalize(word))
  .collect(Collectors.joining(" "));
long timeTaken = System.currentTimeMillis() - startTime;

System.out.println(String.format("%s, Took %d(ms)", capitalized, timeTaken));
// ALL THE AND THE, Took 2027(ms)
```

Don’t Parallelize everything

```
long startTime = System.currentTimeMillis();
long sum = Stream.iterate(0, x -> x + 1)
    .limit(10000)
    .filter(n -> n % 2 == 0)
    .reduce(0, (acc, elem) -> acc + elem);
long diff = System.currentTimeMillis() - startTime;
System.out.println(sum + " Took " + diff + “(ms)");
// 24995000 Took 36(ms)
```

- It takes time to create threads, scatter input and gather results.

```
long startTime = System.currentTimeMillis();
long sum = Stream.iterate(0, x -> x + 1)
    .limit(10000)
    .parallel()
    .filter(n -> n % 2 == 0)
    .reduce(0, (acc, elem) -> acc + elem);
long diff = System.currentTimeMillis() - startTime;
System.out.println(sum + " Took " + diff + "(ms)");
// 24995000 Took 56(ms)
```

Streams - Infinitely Lazy

- Streams are pull-based, consumer decides the pace of pull.

- With Streams, only essential space is allocated upon materialization, the rest is in ether :)
  - This reduces memory footprint (as you don't bring every item in memory).

```
Stream.iterate(22, x -> x + 1);
```

- Because of laziness findFirst() only runs till it find the first matching element.

```
Stream.iterate(22, x -> x + 1)
  .filter(n -> n % 5 == 0)
  .findFirst()
  .ifPresent(System.out::println); // 25
```

De-structuring

- It is about extracting data from the innards of the Data-Structure.

- Gives us a short-hand way of naming parts of the data-structure.

```
val list = 1 :: 2 :: 3 :: 4 :: Nil
val first :: second :: rest = list
println(first)   // 1
println(second)  // 2
println(rest)    // List(3, 4)

case class Name(first: String, last: String, salutation: String)

val dd = Name("Dhaval", "Dalal", "Mr.")
val Name(fName, lName, s) = dd
println(fName)  // Dhaval
println(lName)  // Dalal
```

Pattern Matching

- In Pattern Matching, we are not only matching on relative positions of elements in the data-structure, but also trying to dispatch to different definitions based on the value inside the data structure.

- This dispatch is a kind of conditional construct. So, it's about associating a definition with a particular set of i/ps.

```
val dd = Name("Dhaval", "Dalal", "Mr.")
val nancy = Name("Nancy", "Drew", "Ms.")
def greet(name: Name) = name match {
  case Name("Nancy", "Drew", _) => s"Hello Ms. Drew"
  case Name(f@"Dhaval", l, s) => s"Hey $f"
  case name => s"Hi ${name.first.toUpperCase}"
}

println(greet(dd))                                  // Hey Dhaval
println(greet(nancy))                               // Hello Ms. Drew
println(greet(Name("Peter", "Shmo", "Mr.")))  // Hi PETER
```

Implementing Recursion Using Pattern Matching In Erlang

```erlang
sum(Acc, []) -> Acc;
sum(Acc, [X|XS]) -> sum(Acc + X, XS).

sum(List) -> sum(0, List).

main(_) ->
  io:format("~p~n", [sum([])]),
  io:format("~p~n", [sum([1, 2, 3, 4])]).
```

Currying

- Refers to the phenomena of rewriting a N-arg function to a nest of functions, each taking only 1-arg at a time

```
// (String, Integer) -> String
String merge(String x, Integer y) {
  return x + y.toString();
}
// String -> (Integer -> String)
Function<Integer, String> merge(String x) {
  return y -> x + y.toString();
}
// () -> (String -> (Integer -> String))
Function<String, Function<Integer, String>> merge() {
  return x -> y -> x + y.toString();
}

System.out.println(merge("Test#", 1));                  // Test#1
System.out.println(merge(“Test#").apply(2));            // Test#2
System.out.println(merge().apply(“Test#").apply(3));    // Test#3
```

Currying

$$f : (T, U, V) \mapsto R$$

For each arg, there is another nested function, that takes a arg and returns a function taking the subsequent arg, until all the args are exhausted.

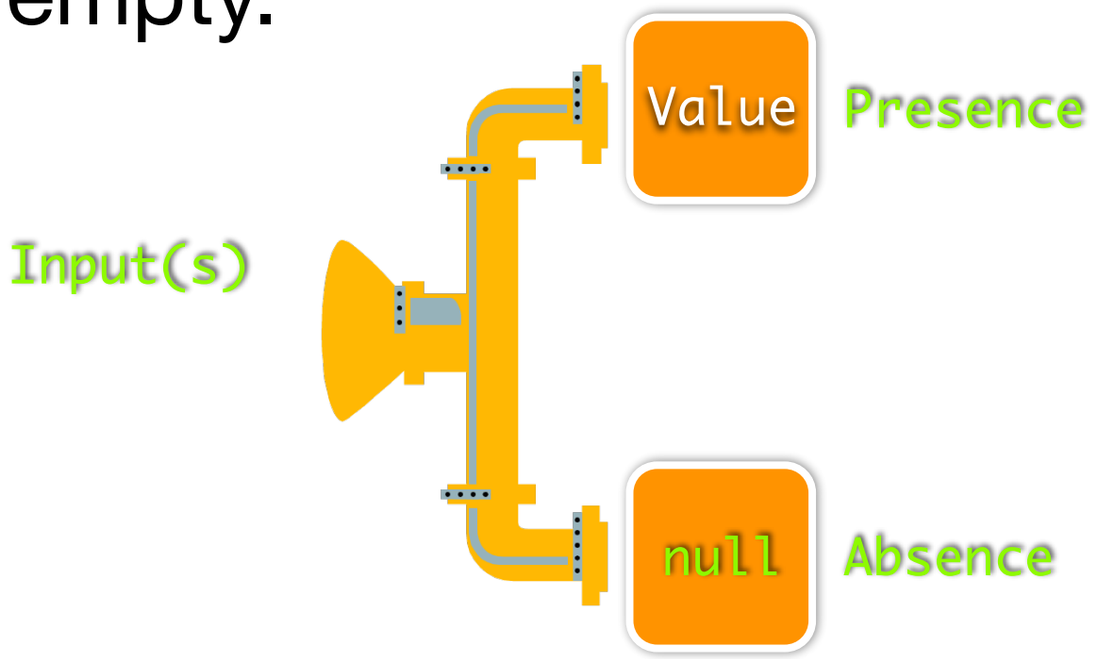
$$f' : T \mapsto U \mapsto V \mapsto R$$

$$f = f'$$

Make Absence Explicit Optional<T>

- Stop null abuse!
  - Don’t return null, use Optional<T> so that it explicitly tells that in the type.

- A container or view it as a collection containing single value or is empty.



Before

```
String frequentFlyerId = "123456789";
UserRepository userRepository = new UserRepository();
User user = userRepository.findBy(frequentFlyerId);
String targetPage = null;
int miles = 7000;
if (user != null) {
  int newPoints = user.addPointsUsing(miles);
  targetPage = "http://tierPage";
} else {
  targetPage = "http://loginPage";
}
System.out.println(targetPage);
```

After

```
String frequentFlyerId = "123456789";
int miles = 7000;
UserRepository userRepository = new UserRepository();
String targetPage = userRepository.findBy(frequentFlyerId)
  .map(user -> {
    int newPoints = user.addPointsUsing(miles);
    return "http://tierPage";
  })
  .orElse("http://loginPage");

System.out.println(targetPage);
```

Upcoming Sessions

Naresha K . - Eclipse Collections, Java Streams & Vavr - What's in them for Functional Programming

Ball Room 1 | Jvm Bootcamp | 45_mins | talk | beginner

Tamizhvendan S - Deep Dive Into Pattern Matching And Destructuring

Ball Room 1 | Jvm Bootcamp | 45_mins | demonstration | beginner

Dhaval Dalal - Make your program spicy - Currying and Partial Function Application (PFA)

Ball Room 1 | Jvm Bootcamp | 45_mins | demonstration | beginner

Tamizhvendan S / Ravindra Jaju - JVM Language Interoperability

Ball Room 1 | Jvm Bootcamp | 45_mins | demonstration | beginner

Anirban Bhattacharjee - KPI driven development : Gradual journey from imperative to functional development in JAVA

Ball Room 1 | Applying Fp | 45_mins | talk | beginner

Ravindra Jaju - Asynchronous Functional Programming on the JVM

Ball Room 1 | Jvm Bootcamp | 45_mins | demonstration | beginner

Thank - You!