WHIRLWIND TOUR OF SCALA

Syntax and Pure functions edition

AGENDA

- Why Scala?
- What is a pure function?
- Pure functions in Java
- Syntax demo
- Pure functions in Scala
- Go home

WHY SCALA?

- Terse syntax
- Interesting and productivity-friendly constructs
- Blend of Imperative and Functional programming ideas

WHAT ARE PURE FUNCTIONS?

- Pure functions
  - No side effects
  - Referentially transparent

PURE FUNCTIONS (JAVA)

```
CreditCard card = new CreditCard();
Cafe cafe = new Cafe();

//Buy three coffees
Coffee coffee1 = cafe.buyCoffee(card);
Coffee coffee2 = cafe.buyCoffee(card);
Coffee coffee3 = cafe.buyCoffee(card);
```

- Implementation of buyCoffee has some non-idempotent code.
- Every time you call the function, it has a different output. (CC gets charged again and more)

PURE FUNCTIONS (JAVA)

```
CreditCard card = new CreditCard();
Cafe cafe = new Cafe();

//Buy three coffees
Coffee coffee1 = cafe.buyCoffee(card);
Coffee coffee2 = cafe.buyCoffee(card);
Coffee coffee3 = cafe.buyCoffee(card);
```

```
public Coffee buyCoffee(CreditCard card) {
    Coffee coffee = new Coffee();
    card.charge(coffee.price());
    return coffee;
}
```

- Can you replace the “returned coffee” with the buyCoffee call with the same argument and expect the same result?

PURE FUNCTIONS – JAVA

```java
public int add(int first, int second){
    return first + second;
}
```

PURE FUNCTIONS (JAVA)

```
final class CoffeeChargeTuple {
    private Charge charge;
    private Coffee cup;

    public CoffeeChargeTuple(Coffee cup, Charge charge) {
        this.charge = charge;
        this.cup = cup;
    }
```

- Let's create a new Charge Tuple (since Tuples aren't available in Java)

PURE FUNCTIONS (JAVA)

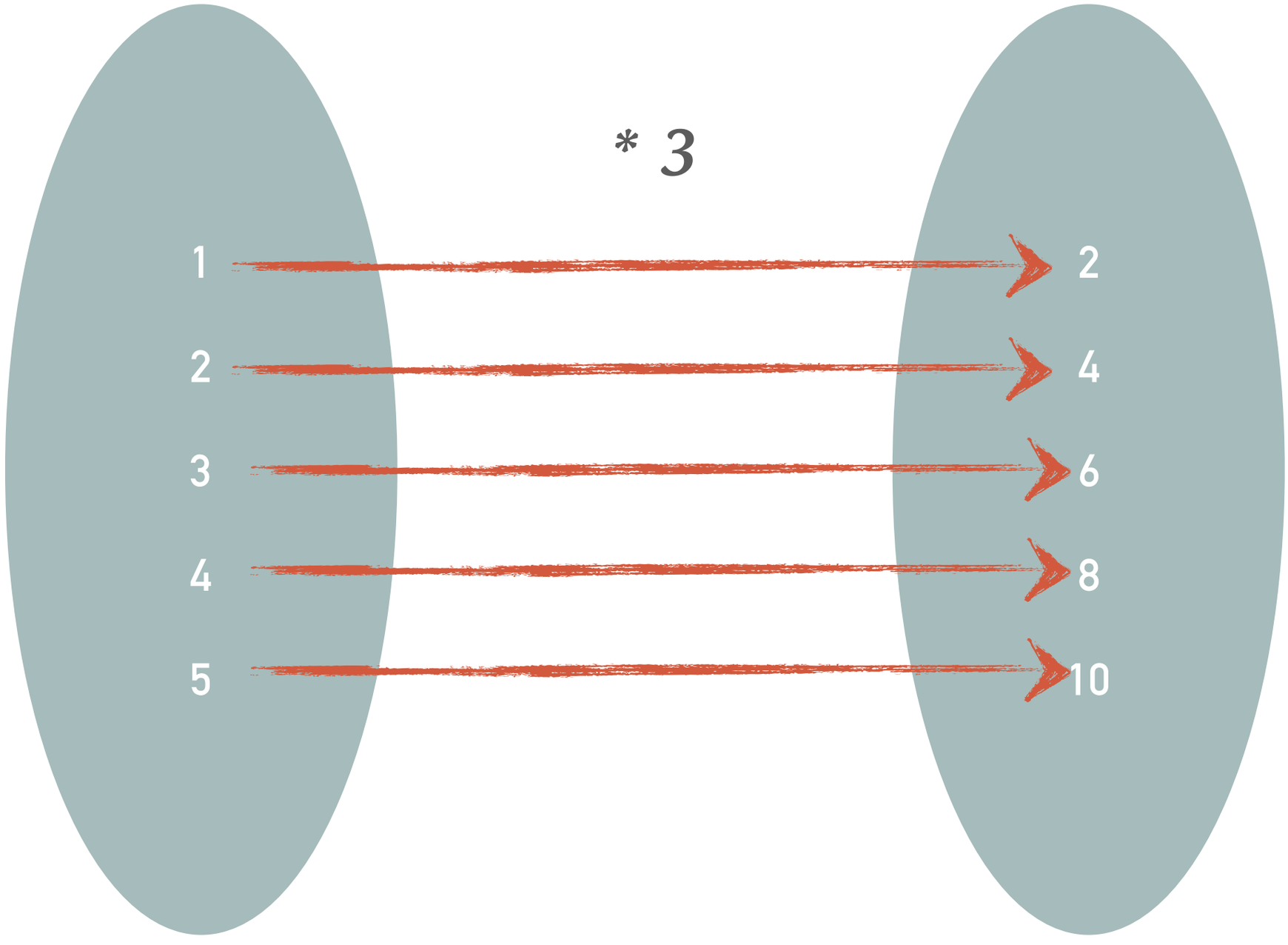
```
CoffeeChargeTuple charge1 = cafe.buyCoffeePure(card);
CoffeeChargeTuple charge2 = cafe.buyCoffeePure(card);
CoffeeChargeTuple charge3 = cafe.buyCoffeePure(card);

List<CoffeeChargeTuple> charges = Arrays.asList(charge1, charge2, charge3);

Charge mergedCharge =
        charges.stream() Stream<CoffeeChargeTuple>
                .map(CoffeeChargeTuple::charge) Stream<Charge>
                .reduce(Charge::combine) Optional<Charge>
                .get(); //Optional.get This is generally frowned upon

card.charge(mergedCharge);
```

- Each buyCoffeePure returns a Charge Tuple which has the price and the coffee.
- You combine the charge into a single one and then charge it against the card.

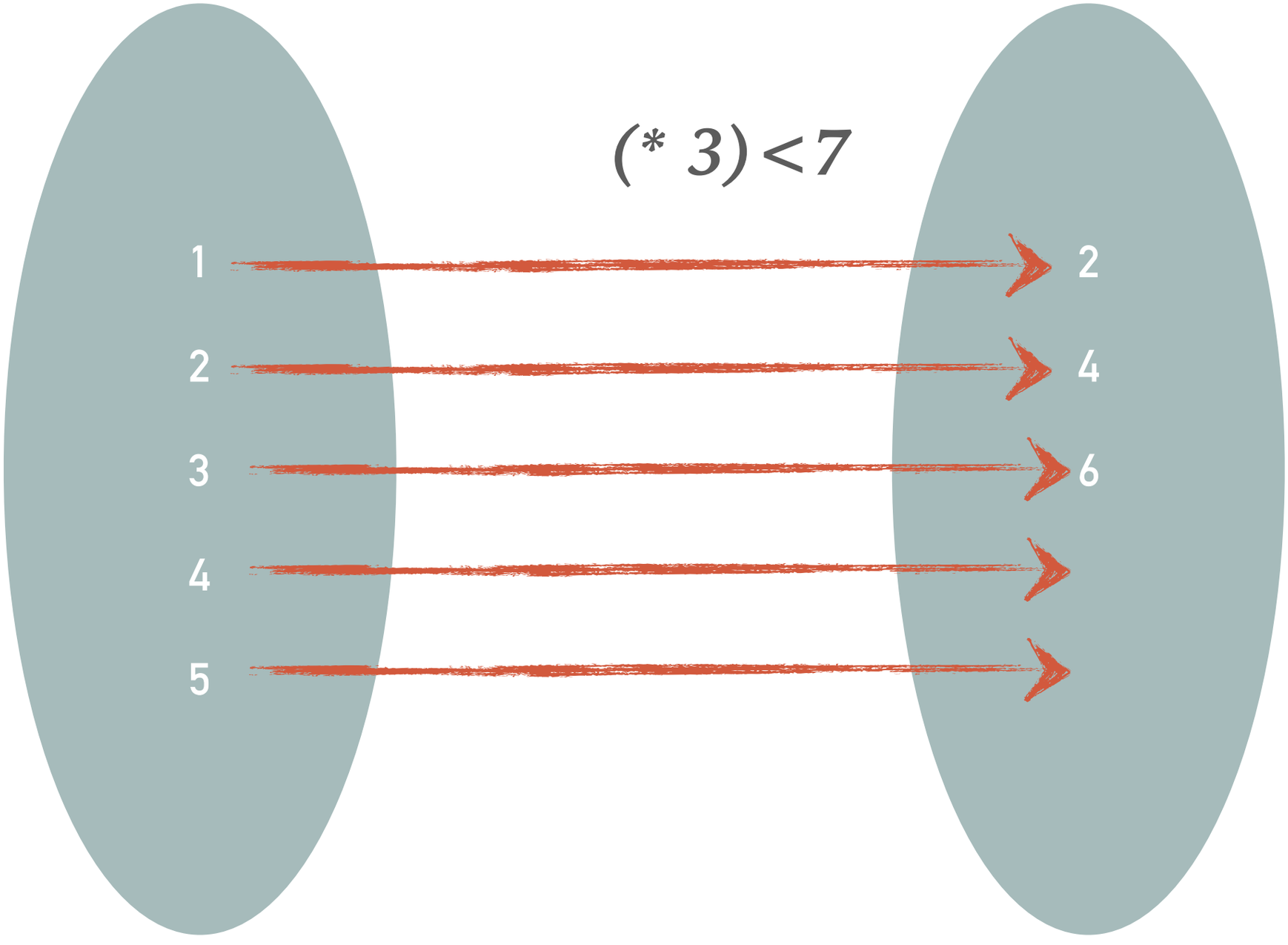
IO calls and Mutable things break referential transparency.

SYNTAX

MAP



FILTER



FOLD/REDUCE

Addition

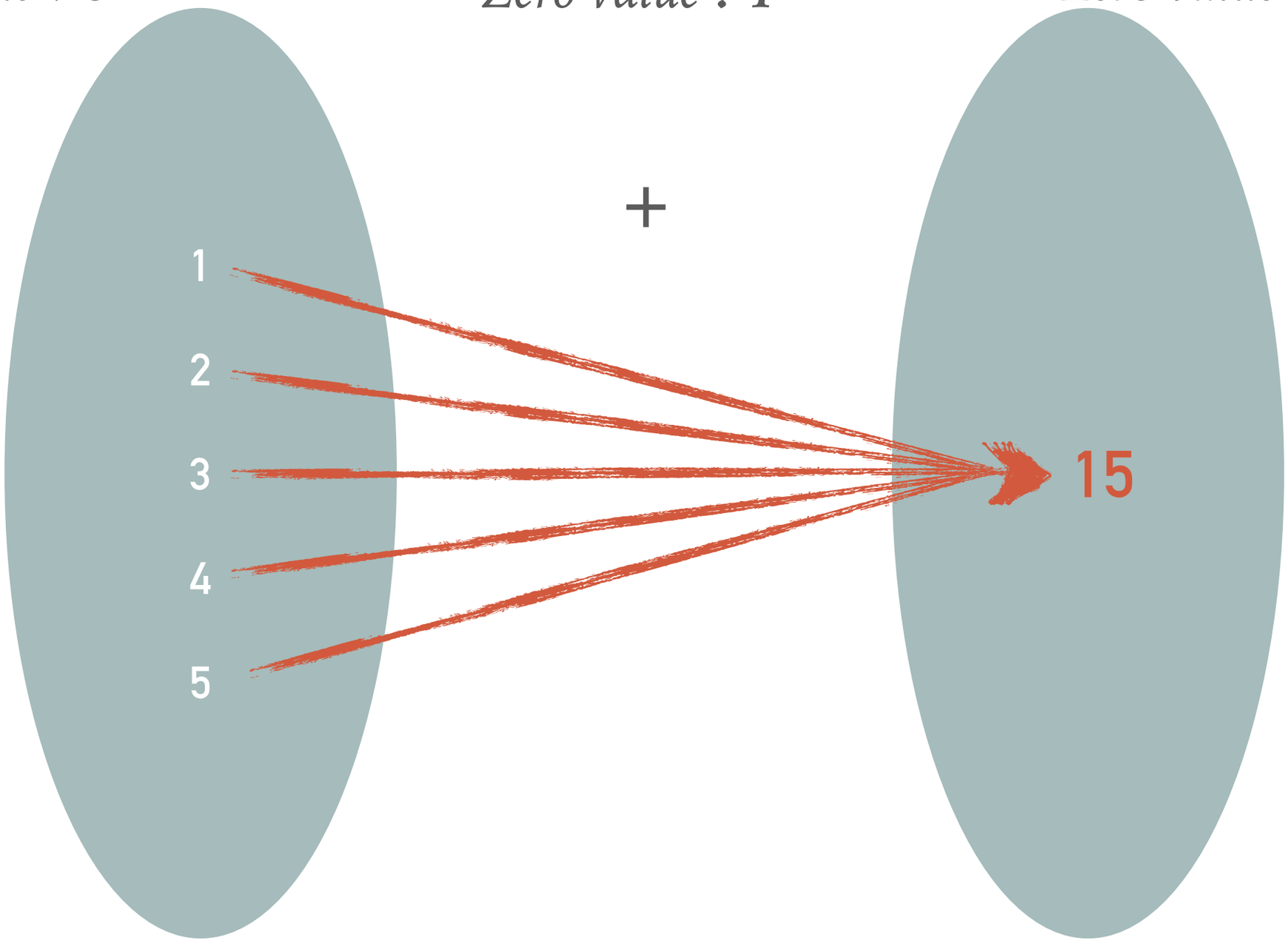
Binary operation : +
*Zero value : **0***

Multiplication

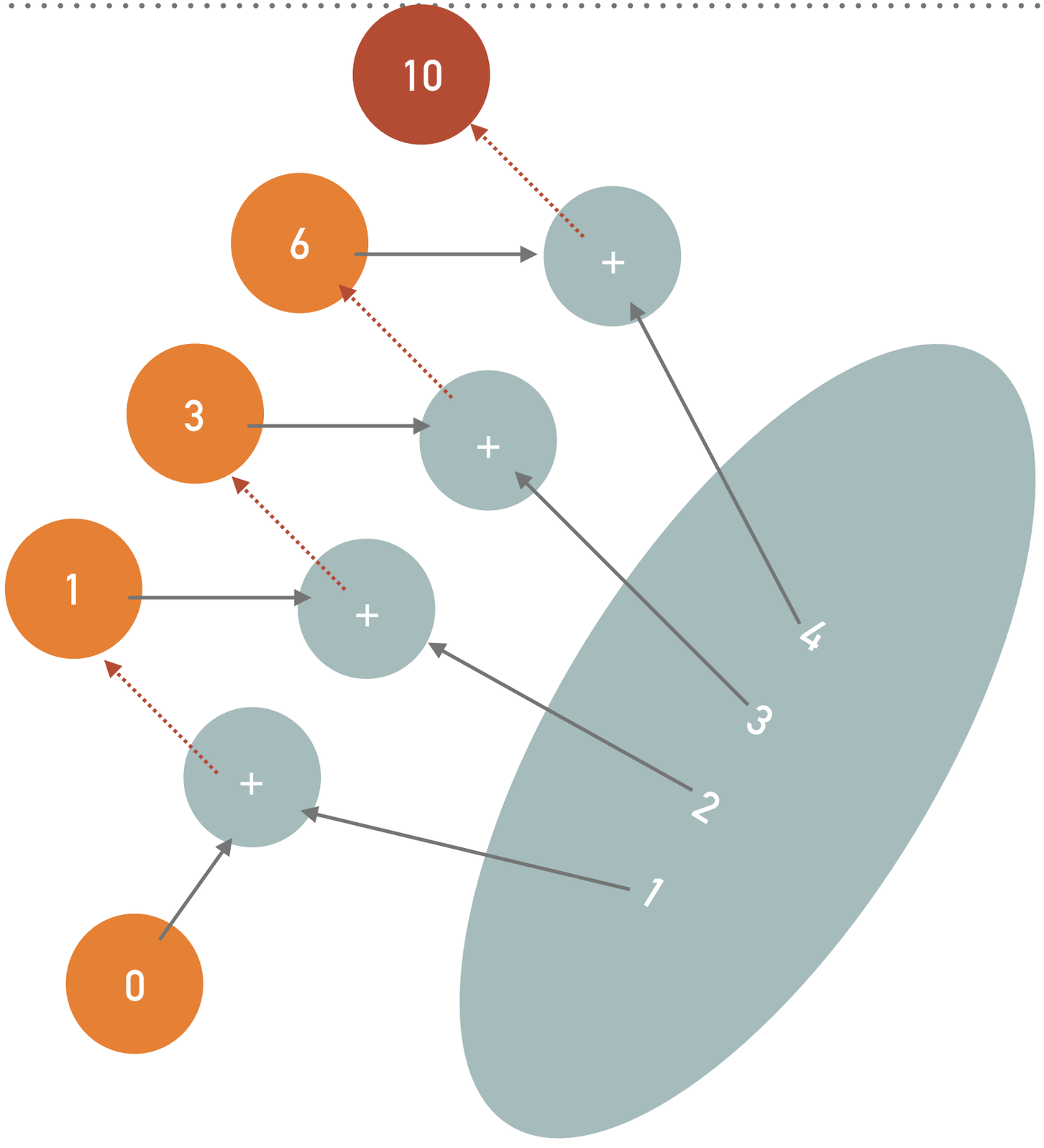
*Binary operation : **
*Zero value : **1***

Concatenation

*Binary operation : **append***
*Zero value : **“”***



FOLD/REDUCE



BUYING COFFEE – JAVA VS SCALA

```java
public class Cafe {

    public CoffeeChargeTuple buyCoffeePure(CreditCard card) {
        Coffee coffee = new Coffee();
        return new CoffeeChargeTuple(coffee, new Charge(card, coffee.price()));
    }
```

```scala
object CafeS {
  def buyCoffeePure(card: CreditCardS): (CoffeeS, ChargeS) = {
    val coffee = new CoffeeS
    (coffee, ChargeS(card, coffee.price()))
  }
}
```

~~case class CafeS(name: String) {~~
~~def buyCoffeePure(card: CreditCardS): (CoffeeS, ChargeS) = {~~
~~val coffee = new CoffeeS~~
~~(coffee, ChargeS(card, coffee.price()))~~
~~}~~
~~}~~

BUYING COFFEE – JAVA VS SCALA

```java
CoffeeChargeTuple charge1 = cafe.buyCoffeePure(card);
CoffeeChargeTuple charge2 = cafe.buyCoffeePure(card);
CoffeeChargeTuple charge3 = cafe.buyCoffeePure(card);

List<CoffeeChargeTuple> charges = Arrays.asList(charge1, charge2, charge3);

Charge mergedCharge =
        charges.stream() Stream<CoffeeChargeTuple>
                .map(CoffeeChargeTuple::charge) Stream<Charge>
                .reduce(Charge::combine) Optional<Charge>
                .get(); //Optional.get This is generally frowned upon

card.charge(mergedCharge);
```

```scala
val charge1 = CafeS.buyCoffeePure(card)
val charge2 = CafeS.buyCoffeePure(card)
val charge3 = CafeS.buyCoffeePure(card)

//val (coffees, charges) = List(charge1, charge2, charge3).unzip
val (_, charges) = List(charge1, charge2, charge3).unzip

val mergedCharge =
  charges
    .map(_.charge)
    .reduce(ChargeS.combine)

TransactionGateway.charge(mergedCharge)
```

COMBINE – JAVA

```
public Charge combine(Charge other) throws RuntimeException {
    if (this.card.equals(other.card)) {
        return new Charge(card,  price: this.price + other.price);
    } else {
        throw new RuntimeException ("Cannot combine charges made against two different cards");
    }
}
```

- The throwing Exception is not idiomatic FP and there are other ways to wrap your “error throwing” code

```
public Charge combine(Charge thisC, Charge other) throws RuntimeException {
    if (thisC.card.equals(other.card)) {
        return new Charge(card,  price: thisC.price + other.price);
    } else {
        throw new RuntimeException ("Cannot combine charges made against two different cards");
    }
}
```

COMBINE – JAVA

```java
public Charge combine(Charge other) throws RuntimeException {
    if (this.card.equals(other.card)) {
        return new Charge(card,  price: this.price + other.price);
    } else {
        throw new RuntimeException ("Cannot combine charges made against two different cards");
    }
}
```

- The throwing Exception is not idiomatic FP and there are other ways to wrap your “error throwing” code

```java
public Charge combine(Charge thisC, Charge other) throws RuntimeException {
    if (thisC.card.equals(other.card)) {
        return new Charge(card,  price: thisC.price + other.price);
    } else {
        throw new RuntimeException ("Cannot combine charges made against two different cards");
    }
}
```

SEMIGROUP !!

MAP

```
list.map(each => each * 2)
```

- Let's look at the function signature of map

```
def map[A, B](fa: List[A])(f: A => B): List[B]
```

- **map** function accepts two parameters - a list and a function that accepts a type and returns a different/same type. The map function returns a List of the new type.

MAP

```
list.map(each => each * 2)
```

- Let's look at the function signature of map

```
def map[A, B](fa: List[A])(f: A => B): List[B]
```

FUNCTOR !!

- **map** function accepts two parameters - a list and a function that accepts a type and returns a different/same type. The map function returns a List of the new type.

FOLD

```scala
val list = List(1, 2, 3, 4, 5)

list.foldLeft(0)((acc, curr) => acc + curr)
```

```
list.foldLeft(zero)(combine)
```

Addition

Binary operation : +
*Zero value : **0***

Multiplication

*Binary operation : **
*Zero value : **1***

Concatenation

*Binary operation : **append***
Zero value : “”

FOLD

```scala
val list = List(1, 2, 3, 4, 5)

list.foldLeft(0)((acc, curr) => acc + curr)
```

MONOID !!

```
list.foldLeft(zero)(combine)
```

Addition

Binary operation : +
*Zero value : **0***

Multiplication

*Binary operation : **
Zero value : 1

Concatenation

*Binary operation : **append***
Zero value : “”

MONOID

```scala
trait Semigroup[A] {
  def combine(a1: A, a2: A): A
}
```

```scala
trait Monoid[A] extends Semigroup[A] {
  def zero: A
}
```

- A semigroup with a “zero” value is a Monoid.


That's all Folks!

SUPPLEMENTARY

WHAT IS FUNCTIONAL PROGRAMMING?

- Pure functions (**aka** No side effects **aka** Referentially transparent)
- Immutable data
- Algebraic datatypes (disjoint unions) and pattern matching
- Higher order functions
- Higher-kinded types
- Parametric Polymorphism (Type classes)
- Mapping with Categories

CAN'T YOU DO THIS IN JAVA/PYTHON

- Of course you can, kinda.
- Pattern matching
- Typeclasses (Adhoc polymorphism)
- Higher kinded types
- Converting everything to a stream before you apply a map/filter and converting back to a stream
- There's Scala and then there's idiomatic Scala