Código funcional em Java:
Superando o hype

Eder Ignatowicz
Sr. Software Engineer
JBoss by Red Hat

Dora

Bento


<3

Jesse "Pinkman"

```
        public Pug( String name,
                    String color,
                    Integer weight ) {
            this.name = nome;
            this.color = color;
            this.weight = weight;
        }
```

```
Pug dora = new Pug( "Dora", "abricot", 10 );
Pug bento = new Pug( "Bento", "abricot", 13 );
Pug jesse = new Pug( "Jesse", "black", 9 );
```

Predicate

Interface Predicate<T>

Type Parameters:

`T - the type of the input to the predicate`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

```
@FunctionalInterface
public interface Predicate<T>
```

Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since:

`1.8`

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
`default Predicate<T>`	**`and`**`(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical AND of this predicate and another.
`static <T> Predicate<T>`	**`isEqual`**`(Object targetRef)` Returns a predicate that tests if two arguments are equal according to `Objects.equals(Object, Object)`.
`default Predicate<T>`	**`negate`**`()` Returns a predicate that represents the logical negation of this predicate.
`default Predicate<T>`	**`or`**`(Predicate<? super T> other)` Returns a composed predicate that represents a short-circuiting logical OR of this predicate and another.
`boolean`	**`test`**`(T t)` Evaluates this predicate on the given argument.

```
Pug dora = new Pug( "Dora", "abricot", 10 );
Pug bento = new Pug( "Bento", "abricot", 13 );
Pug jesse = new Pug( "Jesse", "black", 9 );


List<Pug> pugs = Arrays.asList( dora, bento, jesse );

Predicate<Pug> abricot = pug ->
                    pug.getColor().equalsIgnoreCase( "abricot" );

List<Pug> abricots = pugs.stream()
        .filter( abricot )
        .collect( toList() );

print( abricots );
```

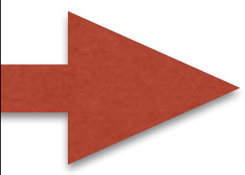
Pug{nome='Dora', color='abricot', size=10}
Pug{nome='Bento', color='abricot', size=13}

```
Pug dora = new Pug( "Dora", "abricot", 10 );
Pug bento = new Pug( "Bento", "abricot", 13 );
Pug jesse = new Pug( "Jesse", "black", 9 );


List<Pug> pugs = Arrays.asList( dora, bento, jesse );

Predicate<Pug> abricot = pug ->
                    pug.getColor().equalsIgnoreCase( "abricot" );

List<Pug> abricots = pugs.stream()
        .filter( abricot )
        .collect( toList() );

print( abricots );

Predicate<Pug> fat = pug -> pug.getWeight() > 12;

List<Pug> fatAbricots = pugs.stream()
        .filter( fat.and( abricot ) )
        .collect( toList() );

print( fatAbricots );
```

Pug{nome='Bento', color='abricot', weight=13}

Consumer

```java
public class ShoppingCartTest {

    ShoppingCart cart;

    @Before
    public void setup() {
        Item item1 = new Item( 10 );
        Item item2 = new Item( 20 );
        cart = new ShoppingCart( Arrays.asList( item1, item2 ) );
    }

    @Test
    public void totalTest() {
        cart.pay( ShoppingCart.PaymentMethod.CREDIT ) );
    }
}
```

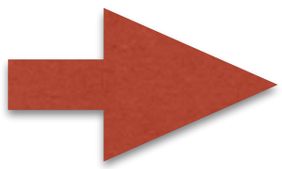


```
public class ShoppingCart {

    private List<Item> items;

     public ShoppingCart( List<Item> items ) {
         this.items = items;
     }

     public void pay( PaymentMethod method ) {
         int total = cartTotal();
         if ( method == PaymentMethod.CREDIT ) {
             System.out.println( “Pay with credit “ + total );
         } else if ( method == PaymentMethod.MONEY ) {
             System.out.println( “Pay with money “ + total );
         }
     }

      private int cartTotal() {
         return  items
                 .stream()
                 .mapToInt( Item::getValue )
                 .sum();
     }

…
}
```

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Credit: “+ amount);
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "Pay with Money: “+ amount);
}
```



```
public class ShoppingCart {
    …
    public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }

    private int cartTotal() {
     return  items
             .stream()
             .mapToInt( Item::getValue )
             .sum();
    }
}
```

Strategy

"Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam" GAMMA, Erich et al.

```java
public interface Payment {

    public void pay(int amount);

}
```

```java
public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make credit

  payment logic" );
    }
}
```

```java
public class Money implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make money

    }
}
```

```java
public class DebitCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make debit
payment logic" );
    }
}
```

```java
    public void totalTest() {
        assertEquals( 30, cart.pay( new CreditCard() ) );
        assertEquals( 30, cart.pay( new Money() ) );
        assertEquals( 30, cart.pay( new DebitCard() ) );
    }
}
```

java.util.function

Interface Consumer<T>

Type Parameters:

`T - the type of the input to the operation`

All Known Subinterfaces:

`Stream.Builder<T>`

Functional Interface:

`This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.`

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method and Description
`void`	**`accept(T t)`** Performs this operation on the given argument.
`default Consumer<T>`	**`andThen(Consumer<? super T> after)`** Returns a composed `Consumer` that performs, in sequence, this operation followed by the `after` operation.

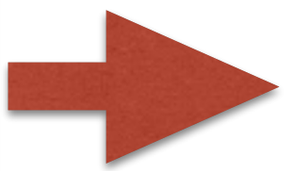
```
public void pay( Payment method ) {
        int total = cartTotal();
        method.pay( total );
    }
```

```
public class ShoppingCart {
        …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

        …
}
```

```
public class ShoppingCart {
        …

    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

        …
}
```

```
public void totalTest() {
    cart.pay( amount ->
                System.out.println( "Pay with Credit: " + amount ) );
    cart.pay( amount ->
                System.out.println( "Pay with Money: " + amount ) );
    cart.pay( amount ->
                System.out.println( "Pay with Debit: " + amount ) );
}
```

```
public class PaymentTypes {

    public static void money( int amount ) {
        System.out.println( "Pay with Money: " + amount );
    }

    public static void debit( int amount ) {
        System.out.println( "Pay with Debit: " + amount );
    }

    public static void credit( int amount ) {
        System.out.println( "Pay with Credit: " + amount );
    }

}
```



```
        public void totalTest() {
            cart.pay( PaymentTypes::credit );
            cart.pay( PaymentTypes::debit );
            cart.pay( PaymentTypes::money );
        }
```

```
public class ShoppingCart {
    …
    public void pay( Consumer<Integer> method ) {
        int total = cartTotal();
        method.accept( total );
    }

    …
}
```

Strategy


Happy Pug
is happy
ICANHASCHEEZBURGER.COM

Functions

Interface Function<T,R>

Type Parameters:

`T` - the type of the input to the function

`R` - the type of the result of the function

All Known Subinterfaces:

`UnaryOperator<T>`

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
@FunctionalInterface
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Since:

1.8

Method Summary

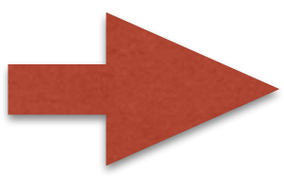
All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
`default <V> Function<T,V>`	**`andThen(Function<? super R,? extends V> after)`** Returns a composed function that first applies this function to its input, and then applies the `after` function to the result.
`R`	**`apply(T t)`** Applies this function to the given argument.
`default <V> Function<V,R>`	**`compose(Function<? super V,? extends T> before)`** Returns a composed function that first applies the `before` function to its input, and then applies this function to the result.
`static <T> Function<T,T>`	**`identity()`** Returns a function that always returns its input argument.

```
List<Pug> pugs = Arrays.asList( dora, bento, jesse );

Function<Pug, String> extractName = pug -> pug.getName();

List<String> nomes = pugs.stream()
        .map( extractName )
        .collect( Collectors.toList() );

print( nomes );
```

Dora
Bento
Jesse

```
List<Pug> pugs = Arrays.asList( dora, bento, jesse );


 Function<Pug, String> extractName = pug -> pug.getName();

 List<String> nomes = pugs.stream()
         .map( extractName )
         .collect( Collectors.toList() );

 print( nomes );
```



```
UnaryOperator<String> upper = s -> s.toUpperCase();
```

```
List<Pug> pugs = Arrays.asList( dora, bento, jesse );


 Function<Pug, String> extractName = pug -> pug.getName();

 List<String> nomes = pugs.stream()
         .map( extractName )
         .collect( Collectors.toList() );

 print( nomes );

 UnaryOperator<String> upper = s -> s.toUpperCase();

 List<String> nomesUpper = pugs.stream()
         .map( extractName.andThen( upper ) )
         .collect( Collectors.toList() );

 print( nomesUpper );
```

```java
public class Item {
    private int price;

    public Item( int price ) {
        this.price = price;
    }

    public int getPrice() {
        return price;
    }
}
```

Extras

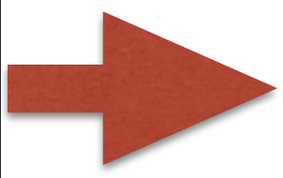
Envio

Impostos

Embalagem

```java
public interface Item {
    int getPrice();
}
```

```java
public class Book implements Item {
    private int price;

    public Book( int price ) {
        this.price = price;
    }

    @Override
    public int getPrice() {
        return price;
    }
}
```



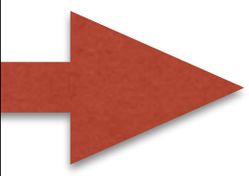
```
public abstract class ItemExtras implements Item {

    private Item item;

    public ItemExtras( Item item ) {
        this.item = item;
    }

    @Override
    public int getPrice() {
        return item.getPrice();
    }
}
```

```
public class InternationalDelivery extends ItemExtras {

    public InternationalDelivery( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 5  + super.getPrice();
    }
}
```

```
public class GiftPacking extends ItemExtras {

    public GiftPacking( Item item ) {
        super( item );
    }

    @Override
    public int getPrice() {
        return 15 + super.getPrice();
    }
}
```

```
public static void main( String[] args ) {

    Item book = new Book( 10 );
    book.getPrice(); //10

    Item international = new InternationalDelivery( book );
    international.getPrice(); //15

}
```

```
public static void main( String[] args ) {

        Item book = new Book( 10 );
        book.getPrice(); //10

        Item internationalGift = new GiftPacking(
                                     new InternationalDelivery( book ) );
        internationalGift.getPrice(); //30

}
```



```
public static void main( String[] args ) {

    Item book = new Book( 10 );
    book.getPrice(); //10

    Item internationalGiftWithTaxes = new InternacionalTaxes(
                                          new GiftPacking(
                                              new InternationalDelivery( book );
        internationalGiftWithTaxes.getPrice(); //80
    }
}
```

Decorator

"Dinamicamente, agregar responsabilidades adicionais a objetos. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades."

GAMMA, Erich et al.

```
new BufferedReader(new FileReader(new File("some.file")));
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25
}
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60

}
```

```
public static void main( String[] args ) {

        Item book = new Item( 10 );
        book.getPrice(); //10

        Function<Integer, Integer> giftPacking = value -> value + 15;
        giftPacking.apply( book.getPrice() ); //25

        Function<Integer, Integer> intTaxes = value -> value + 50;
        intTaxes.apply( book.getPrice() ); //60

        giftPacking.andThen( intTaxes ).apply( book.getPrice() ); //75

}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75

}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10 );
    Function<Integer, Integer> giftPacking = value -> value + 15;
    Function<Integer, Integer> intTaxes = value -> value + 50;

    book.setItemExtras( giftPacking, intTaxes );

    book.getPrice(); //75
}
```

```apex
public class Packing {

    public static Integer giftPacking( Integer value ) {
        return value + 15;
    }
//other packing options here
}
```

```apex
public class Taxes {

    public static Integer internacional( Integer value ) {
        return value + 50;
    }
//other taxes here
}
```

```
public static void main( String[] args ) {
    Item book = new Item( 10, Packing::giftPacking,
                              Taxes::internacional );

    book.getPrice(); //75
}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    public Item( int price ) {
        this.price = price;
    }

    public Item( int price, Function<Integer, Integer>... itemExtras ) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    public int getPrice() {
        Function<Integer, Integer> extras =
                Stream.of( itemExtras )
                .reduce( Function.identity(), Function::andThen );
        return extras.apply( price );
    }

    public void setItemExtras( Function<Integer, Integer>... itemExtras ) {
        this.itemExtras = itemExtras;
    }
}
```

Streams API

Streams:

Manipula coleções de forma declarativa

Quais são os nomes ordenados
dos Pugs com peso maior do que 9 quilos?

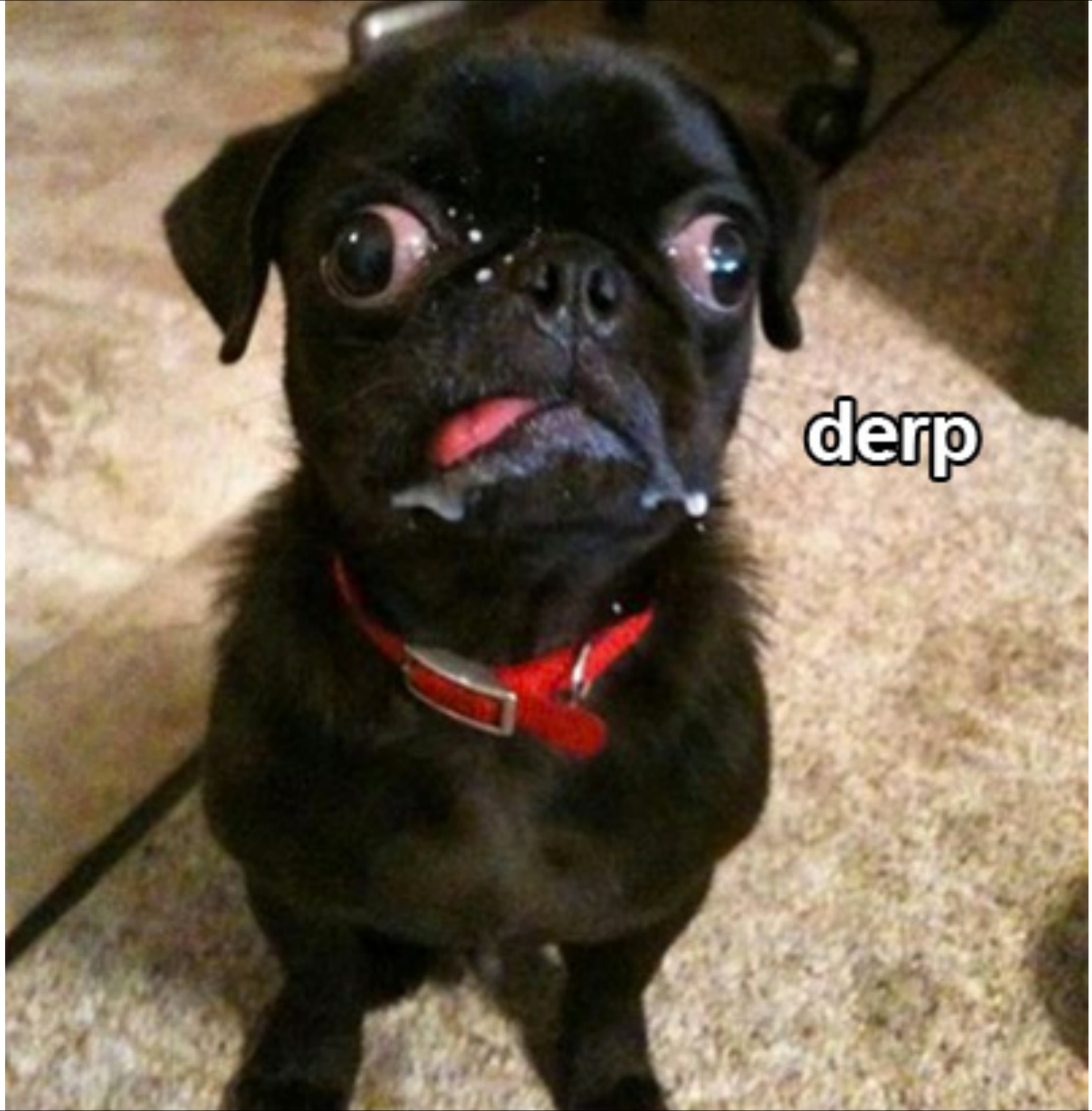
Streams:

Manipula coleções de forma declarativa

```
SELECT nome FROM pugs WHERE weight < 9 order by weight.
```

Em Java

```
List<Pug> gordinhos = new ArrayList<>();
for ( Pug pug : pugs ) {
   if ( pug.getWeight() > 9 ) {
      gordinhos.add( pug );
   }
}
Collections.sort( gordinhos, new Comparator<Pug>() {
            @Override
            public int compare( Pug p1,
                                Pug p2 ) {
                return Integer.compare( p1.getWeight(),
                                        p2.getWeight() );
            }
});

List<String> nomeGordinhos = new ArrayList<>();
for ( Pug pug : gordinhos ) {
   nomeGordinhos.add( pug.getNome() );
}
```


derp

Em Java

```
List<String> fatName =
        pugs.stream()
        .filter( p -> dora.getWeight() > 9 )
        .sorted( comparing( Pug::getWeight ) )
        .map( Pug::getNome )
        .collect( toList() );
```

Seleciona > 9 kg

Ordena por peso

Extrai o nome

Coleta em uma lista

Em Java

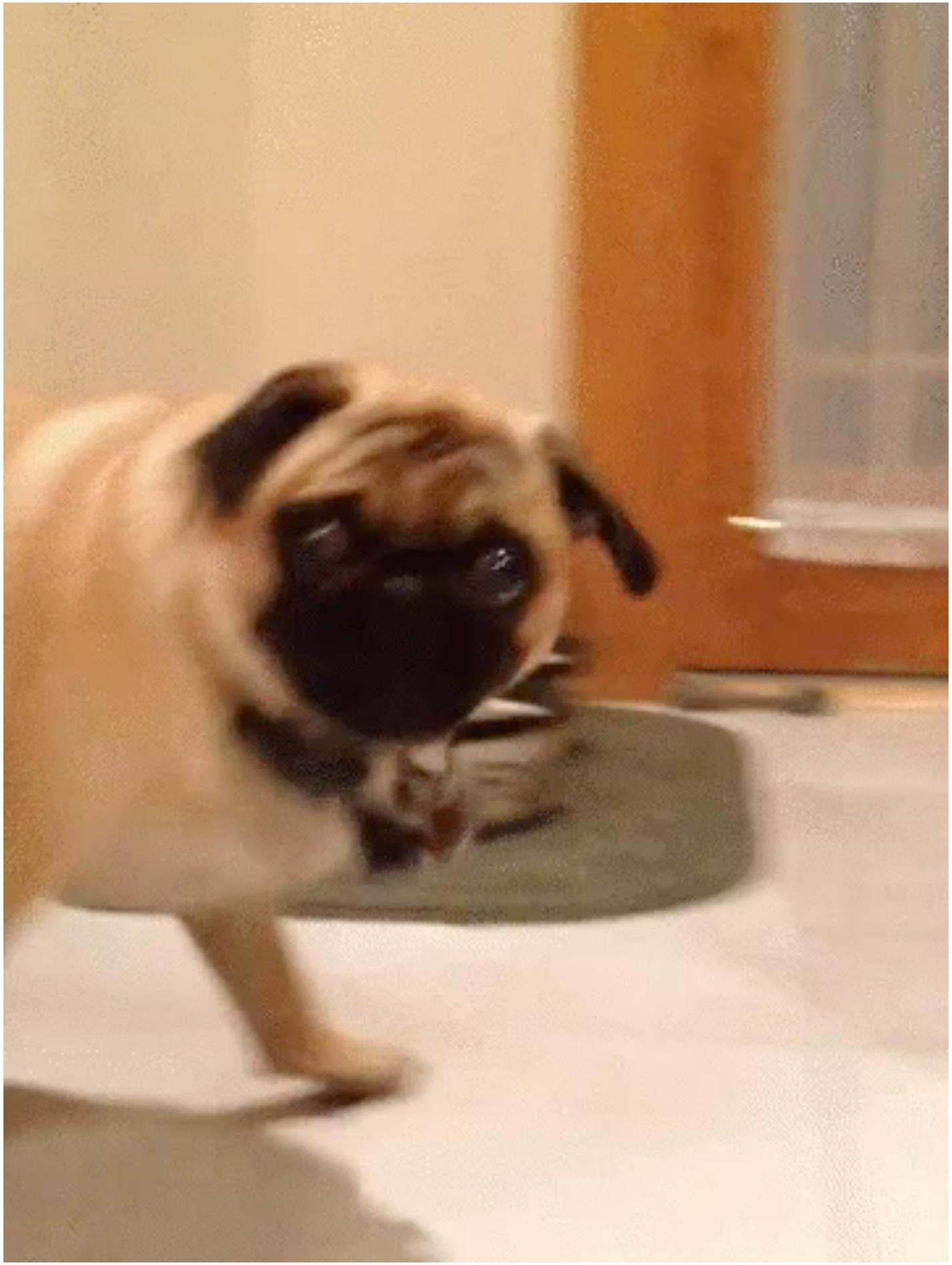
```
List<String> fatName =

        pugs.parallelStream()

        .filter( p -> dora.getWeight() > 9 )

        .sorted( comparing( Pug::getWeight ) )

        .map( Pug::getNome )

        .collect( toList() );
```

Seleciona > 9 kg

Ordena por peso

Extrai o nome

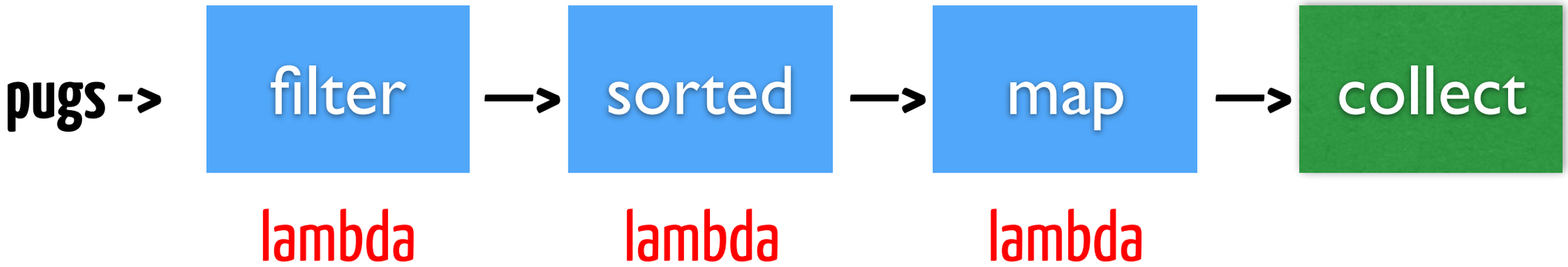
Coleta em uma lista



Parallel streams não são mágica!


Functional programming for the MEOW!

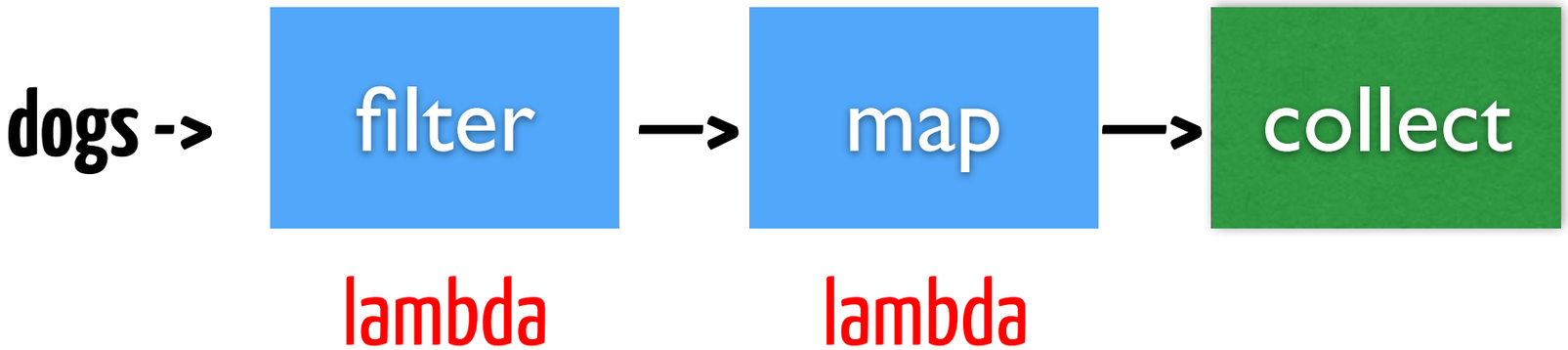
Stream Pipelines



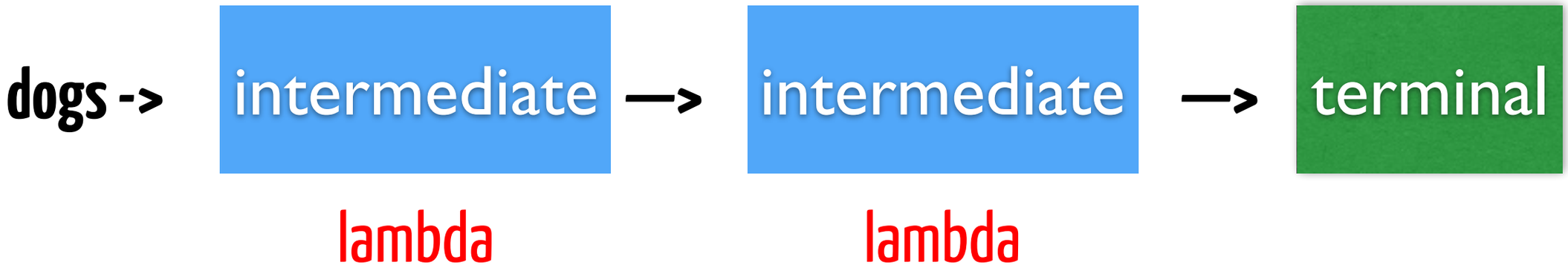
Streams

são lazy

Stream Pipelines



Stream Pipelines



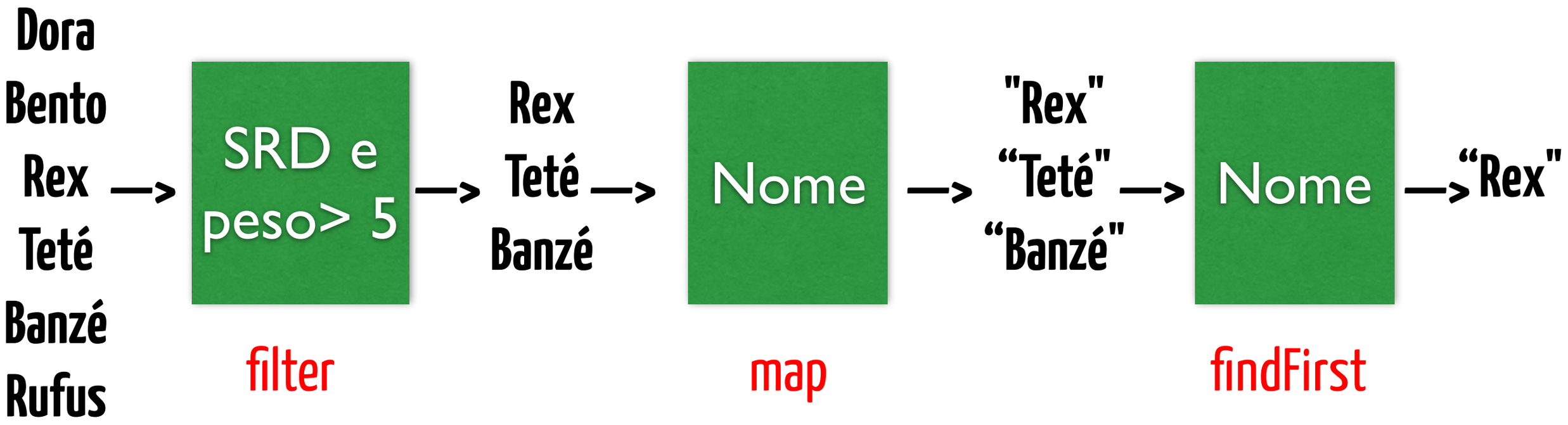
Lazy Streams

```
List<Dog> dogs = Arrays.asList(
            new Dog( "Dora", 10, Dog.BREED.PUG ),
            new Dog( "Bento", 13, Dog.BREED.PUG ),
            new Dog( "Rex", 8, Dog.BREED.SRD ),
            new Dog( "Tetezinha", 6, Dog.BREED.SRD ),
            new Dog( "Banze", 7, Dog.BREED.SRD ),
            new Dog( "Rufus", 15, Dog.BREED.BULLDOG ) );
```

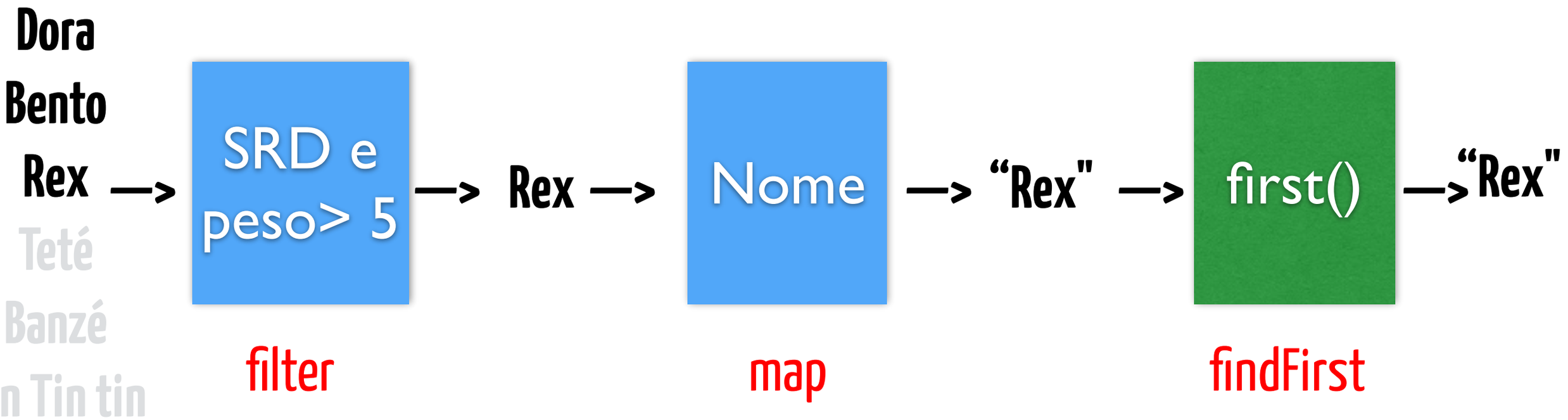
Qual o nome do primeiro SRD que pesa mais do que 5kg?

```
String nomePrimeiroSRDMaiorDoQue5Kg =
        dogs.stream()

                .filter( dog -> {
                    return dog.getBreed().equals( Dog.BREED.SRD )
                            && dog.getWeight() > 5;
                } )

                .map( dog -> {
                    return dog.getName();
                } )

                .findFirst()

                .get();
```

Eager Streams



Lazy Stream


Dora
Bento
Rex
Teté
Banzé
Rin Tin tin
SRD e peso> 5
filter
Rex
Nome
map
"Rex"
first()
findFirst
"Rex"

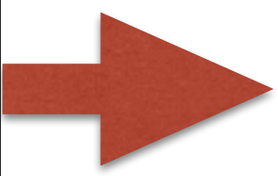
```
List<Dog> dogs = Arrays.asList(
        new Dog( "Dora", 10, Dog.BREED.PUG ),
        new Dog( "Bento", 13, Dog.BREED.PUG ),
        new Dog( "Rex", 8, Dog.BREED.SRD ),
        new Dog( "Tetezinha", 6, Dog.BREED.SRD ),
        new Dog( "Banze", 7, Dog.BREED.SRD ),
        new Dog( "Rufus", 15, Dog.BREED.BULLDOG ) );


String nomePrimeiroSRDMaiorDoQue5Kg =
        dogs.stream()

            .filter( dog -> {
                return dog.getBreed().equals( Dog.BREED.SRD )
                        && dog.getWeight() > 5;
            } )

            .map( dog -> {
                return dog.getName();
            } )

            .findFirst()

            .get();
```

filter - Dora
filter - Bento
filter - Rex
map - Rex
Re

Recursões

Fibonacci

$$f(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ F(n-1) + F(n-2) & if \quad n > 1 \end{cases}$$



```
public static Long fib( int n ) {
      if ( n < 2 ) {
            return new Long( n );
      } else {
            return fib( n - 1 ) + fib( n - 2 );
      }
}
```

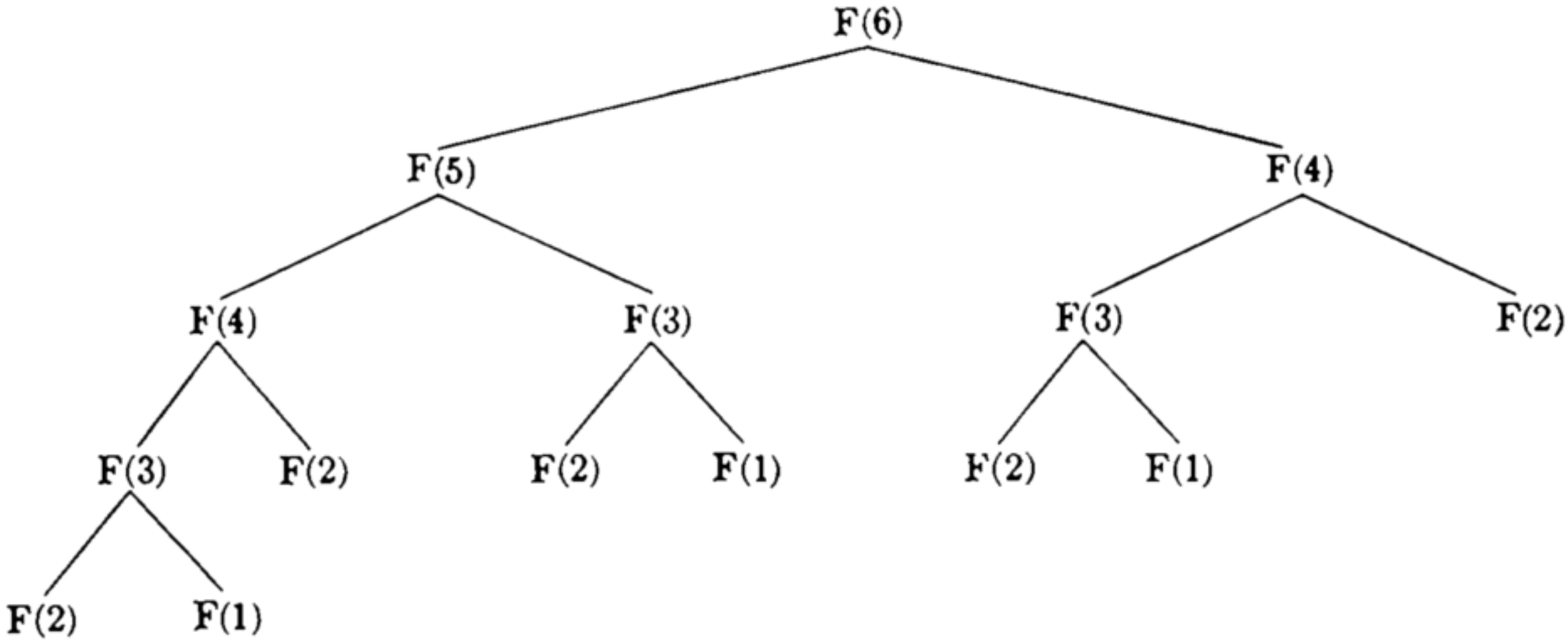


Fig. 1

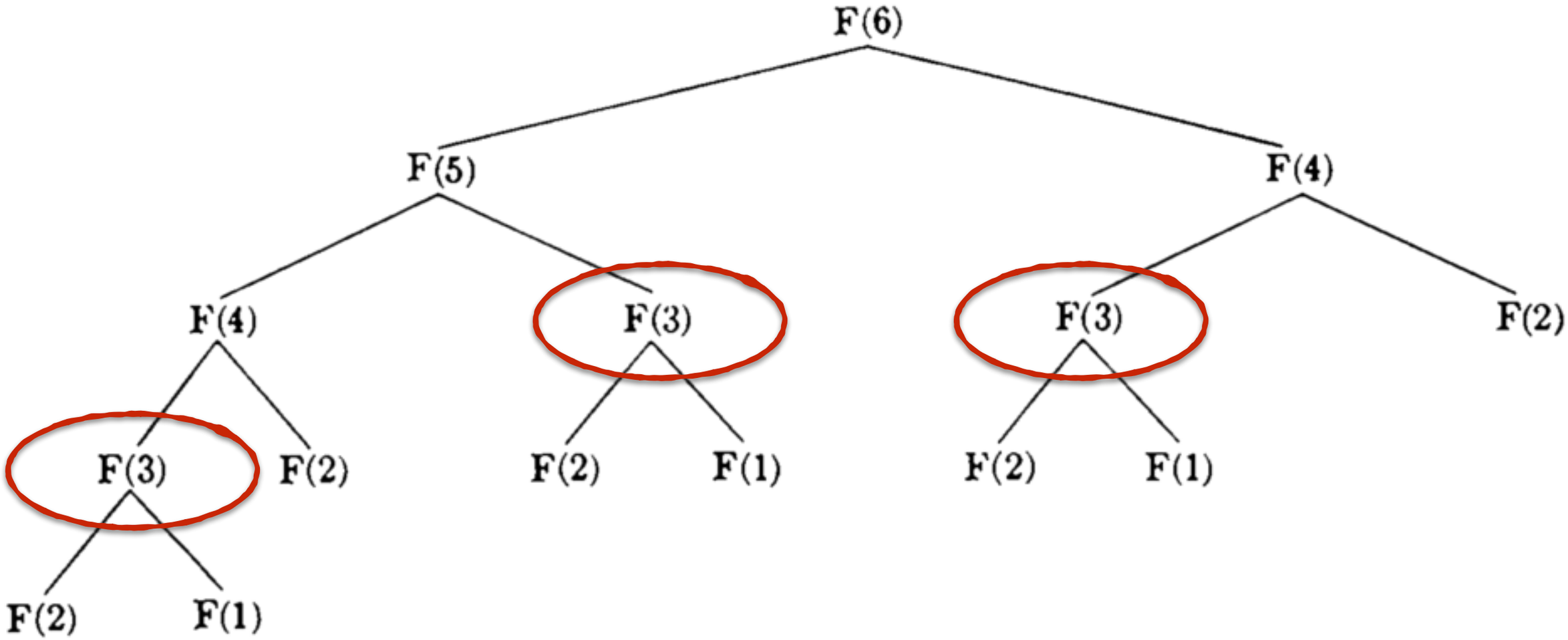


FIG. 1

Memoization

Pure Functions

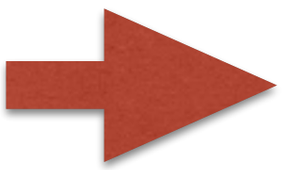
In computer programming, a function may be described as a pure function if both these statements about the function hold:

1-) 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 (usually—see below.

2-) **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 (usually—see below)

Manual

```
Integer doubleValue(Integer x) {
      return x * 2;
  }

Integer doubleValue(Integer x) {

      if (cache.containsKey(x)) {
          return cache.get(x);
      } else {
          Integer result = x * 2;
          cache.put(x, result) ;
          return result;
      }

}
```

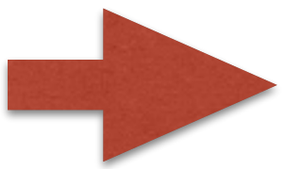


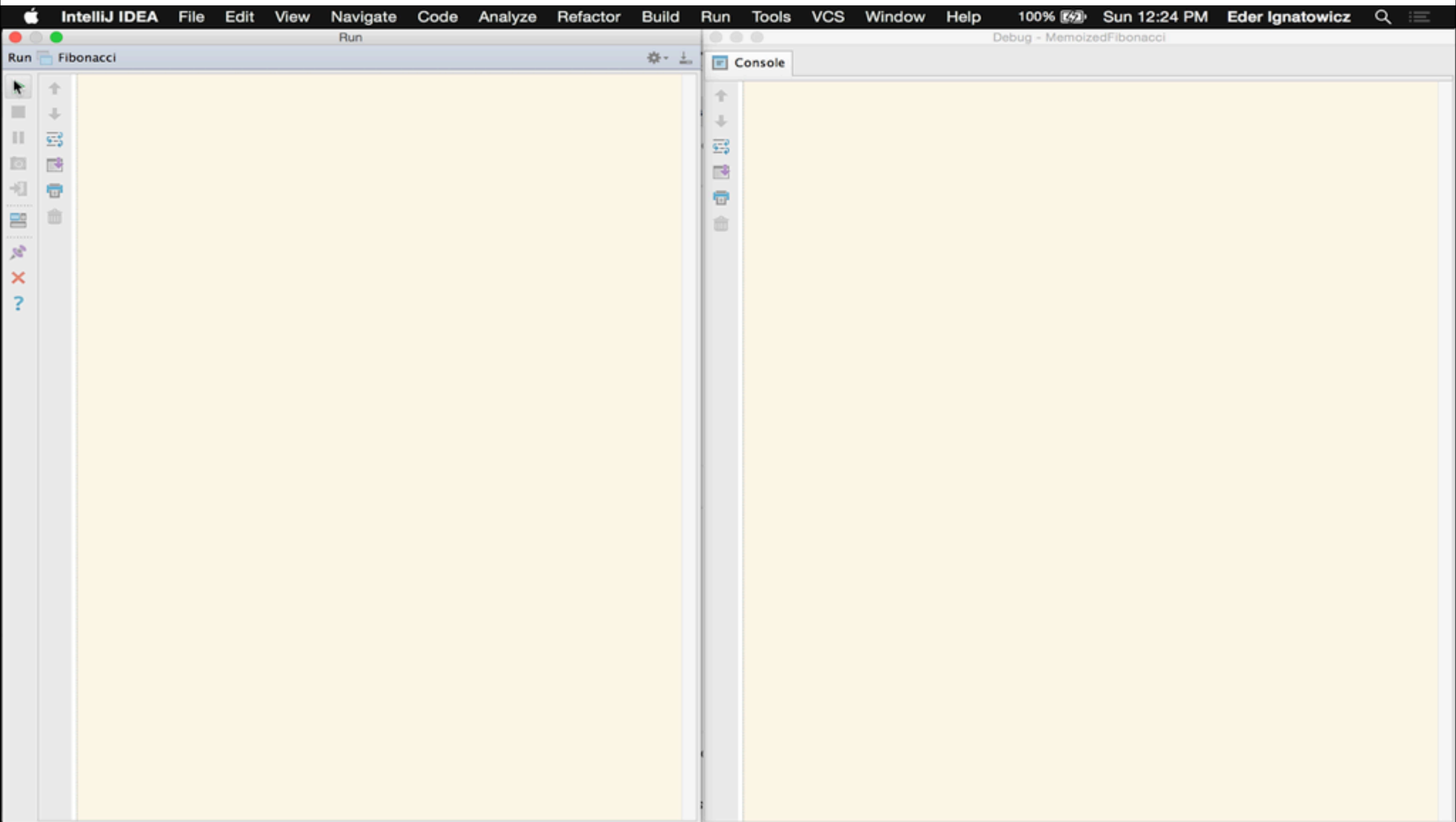
```
private Map<Integer, Integer> cache = new ConcurrentHashMap<>();

public Integer fib(int n) {
    if (n == 0 || n == 1) return n;

    Integer result = cache.get( n );

    if (result == null) {
        synchronized (cache) {
            result = cache.get(n);

            if (result == null) {
                result = fib(n - 2) + fib(n - 1);
                cache.put(n, result);
            }
        }
    }

    return result;
}
```

Java + FP

to the rescue

```
private static Map<Integer, Long> memo = new HashMap<>();

static {
    memo.put( 0, 0L ); //fibonacci(0)
    memo.put( 1, 1L ); //fibonacci(1)
}

public static long fibonacci( int x ) {
    return memo.
        computeIfAbsent(
            x, n -> fibonacci( n - 1 ) + fibonacci( n - 2 ) );
}
```




IntelliJ IDEA File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help 100% Sun 12:24 PM Eder Ignatowicz
Run
Run Fibonacci
Debug - MemoizedFibonacci
Console

Currying

$$f(x,y) = y/x$$

f(2,3)

f(x,y) = y/x

$$f(2, y) = y / 2$$

$$g(y) = f(2,y) = y/2$$

$$g(y) = f(2,y) = y/2$$

$$g(3) = f(2,3) = 3/2$$

```
CtoF(x) = x * 9/5 + 32
```

```
static double converter( double x, double f, double b ) {
    return x * f + b;
}

public static void main( String[] args ) {

    Double celsius = 15.0;
    Double fahrenheit = converter( celsius, 9.0 / 5, 32 ); //59 F
}
```

```
static double converter( double x, double f, double b ) {
    return x * f + b;
}
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
    return x -> x * f + b;
}
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
     return x -> x * f + b;
 }

 public static void main( String[] args ) {

     DoubleUnaryOperator convertCtoF = curriedConverter( 9.0 / 5, 32 );

     convertCtoF.applyAsDouble( 35 ); //95 F
     convertCtoF.applyAsDouble( 15 ); //59 F

 }
```

```
static DoubleUnaryOperator curriedConverter( double f, double b ) {
    return x -> x * f + b;
}

public static void main( String[] args ) {

    DoubleUnaryOperator convertCtoF = curriedConverter( 9.0 / 5, 32 );

    convertCtoF.applyAsDouble( 35 ); //95 F

    DoubleUnaryOperator convertKmToMi = curriedConverter( 0.6214, 0 );

    convertKmToMi.applyAsDouble( 804.672 ); //500milhas
}
```

```
DoubleUnaryOperator convertBRLtoUSD = curriedConverter( 0.27, 0 );

double usd = convertBRLtoUSD.applyAsDouble( 100 );//27 USD

DoubleUnaryOperator convertUSDtoEUR = curriedConverter( 0.89, 0 );
convertUSDtoEUR.applyAsDouble( usd ); //24.03 EUR

convertBRLtoUSD.andThen( convertUSDtoEUR ).applyAsDouble( 100 );

                    //24.03 EUR
```

E agora?

Programar funcional em Java é uma mudança de paradigma

Java é multi-paradigma

Imperativo, OO e Funcional

Escolha o melhor deles para o seu problema

Programação Funcional trouxe uma nova vida para o Java

DIVIRTA-SE!

Obrigado!!!

@ederign

redhat.