

Código funcional em Java: Superando o hype

Eder Ignatowicz
Sr. Software Engineer
JBoss by Red Hat

Lambda Expressions 101



Dora



Bento

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

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

Requisito:

Filtrar todos os pugs de
cor “abricot” da lista

Filtrar todos os pugs de cor "abrilcot" da lista

```
private static List<Pug> filterAbricotPugs( List<Pug> pugs ) {  
    List<Pug> abricots = new ArrayList<>();  
    for ( Pug pug : pugs ) {  
        if ( pug.getColor().equals( "abrilcot" ) ) {  
            abricots.add( pug );  
        }  
    }  
    return abricots;  
}
```

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

```
List<Pug> abricot = filterAbricotPugs( pugs );
```

```
Pug{nome='Dora', color='abrilcot', weight=10}  
Pug{nome='Bento', color='abrilcot', weight=13}
```

Novo Requisito: :)

Filtrar todos os pugs de cor “black” da lista


```
private static List<Pug> filterAbricotPugs( List<Pug> pugs ) {  
    List<Pug> abricots = new ArrayList<>();  
    for ( Pug pug : pugs ) {  
        if ( pug.getColor().equals( "abricot" ) ) {  
            abricots.add( pug );  
        }  
    }  
    return abricots;  
}
```

```
private static List<Pug> filterBlackPugs( List<Pug> pugs ) {  
    List<Pug> abricots = new ArrayList<>();  
    for ( Pug pug : pugs ) {  
        if ( pug.getColor().equals( "black" ) ) {  
            abricots.add( pug );  
        }  
    }  
    return abricots;  
}
```

Filtrar todos os pugs de cor "black" da lista

```
private static List<Pug> filterPugByColor( List<Pug> pugs,  
                                           String color ) {  
  
    List<Pug> coloured = new ArrayList<>();  
    for ( Pug pug : pugs ) {  
        if ( pug.getColor().equals( color ) ) {  
            coloured.add( pug );  
        }  
    }  
    return coloured;  
}
```

```
List<Pug> black = filterPugByColor( pugs, "black" );
```

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

```
Pug{nome='Jesse', color='black', weight=9}
```

Novo Requisito: :)

Filtrar todos os pugs com sobrepeso (>10 kg)

Filtrar todos os pugs com sobrepeso (>10 kg)

```
private static List<Pug> filterPugBySize( List<Pug> pugs,  
                                           int size ) {  
    List<Pug> fat = new ArrayList<>();  
    for ( Pug pug : pugs ) {  
        if ( pug.getWeight() > size ) {  
            fat.add( pug );  
        }  
    }  
    return fat;  
}
```

```
List<Pug> fat = filterPugBySize( pugs, 10 );
```

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

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

Refatorador :D

```
private static List<Pug> filterPugs( List<Pug> pugs,
                                     String color,
                                     int weight,
                                     boolean flag ) {
    List<Pug> result = new ArrayList<>();
    for ( Pug pug : pugs ) {
        if ( ( flag && pug.getColor().equals( color ) ) ||
              ( !flag && pug.getWeight() > weight ) ) {
            result.add( pug );
        }
    }
    return result;
}
```

```
List<Pug> result = filterPugs(pugs, "black", 0, true);  
List<Pug> result1 = filterPugs(pugs, "", 10, false);
```

Behavior Parametrization

Behavior Parametrization

```
public interface PugPredicate {  
    boolean test(Pug pug);  
}
```

```
class BlackPugPredicate implements PugPredicate{  
  
    @Override  
    public boolean test( Pug pug ) {  
        return pug.getColor().equals( "black" );  
    }  
}
```

```
class FatPugPredicate implements PugPredicate{  
  
    @Override  
    public boolean test( Pug pug ) {  
        return pug.getWeight()>10;  
    }  
}
```


Predicate

```
List<Pug> black = filterPug( pugs,  
                             new BlackPugPredicate() );
```

```
List<Pug> fat = filterPug( pugs,  
                           new FatPugPredicate() );
```

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

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

Classes anônimas

<3

```
List<Pug> abricotsNotFat =  
    filterPug( pugs, new PugPredicate() {  
        @Override  
        public boolean test( Pug pug ) {  
            return pug.getColor().equals( "abricot" )  
                && pug.getWeight() <= 10;  
        }  
    } );  
  
abricotsNotFat.forEach( System.out::println );
```

```
Pug{nome='Dora', color='abricot', weight=10}
```

Expressões Lambda

<3

"Arrow"



(Pug p1,Pug p2) -> p1.getWeight().compareTo(p2.getWeight())



**Parâmetros
do
Lambda**



**Corpo
do
Lambda**

```
class BlackPugPredicate implements PugPredicate{  
    @Override  
    public boolean test( Pug pug ) {  
        return "black".equals( pug.getColor() ) ;  
    }  
}
```

```
blacks = filterPug( pugs, new BlackPugPredicate() );
```

```
blacks = filterPug( pugs,  
    (Pug pug) ->"black".equals( pug.getColor() ) );
```

Posso modificar mais um pouco?

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

```
public static <T> List<T> filter  
    (List<T> list, Predicate<T> p){  
  
    List<T> result = new ArrayList<>();  
  
    for(T e: list){  
        if(p.test(e)){  
            result.add(e);  
        }  
    }  
  
    return result;  
}
```



```
List<Pug> blacks =  
    filter( pugs,  
        (Pug pug) ->"black".equals( pug.getColor() ) );
```

```
List<Integer> pares =  
    filter(numbers,  
        i -> i % 2 == 0);
```

```
Predicate<Pug> fatPredicate = d -> d.getWeight() > 9;  
Predicate<Pug> abricotPredicate1 =  
    d -> d.getColor().equalsIgnoreCase( "abricot" );  
Predicate<Pug> abricotAndFat =  
    abricotPredicate.and( fatPredicate );
```

Predicate

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

```
public static <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T s: list){
        if(p.test(s)){
            results.add(s);
        }
    }
    return results;
}
```

```
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

Consumer

```
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 );  
        }  
    }
```

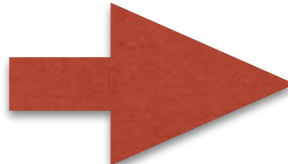
```
    private int cartTotal() {  
        return items  
            .stream()  
            .mapToInt( Item::getValue )  
            .sum();  
    }
```

```
    }
```

```
public interface Payment {  
  
    public void pay(int amount);  
  
}
```

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

```
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.


```

public interface Payment {

    public void pay(int amount);

}

```

```

public class CreditCard implements Payment {
    @Override
    public void pay( int amount ) {
        System.out.println( "make credit
        payment logic" );
    }
}

```

```

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

```

```

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

```

```

    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.		

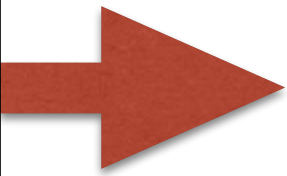
```
public void pay( Payment method ) {  
    int total = cartTotal();  
    method.pay( total );  
}
```



```
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 );  
    }
```

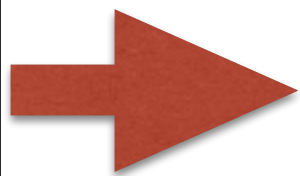
```
    ...
```

```
}
```

```
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

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

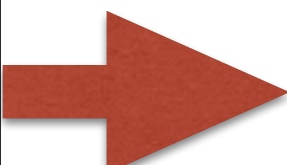
Method Summary

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 );
```

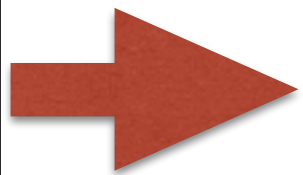
Dora
Bento
Jesse

```
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();

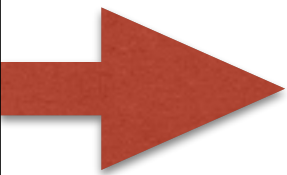
List<String> nomes = pugs.stream()
    .map( extractName )
    .collect( Collectors.toList() );

print( nomes );

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

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

print( nomesUpper );
```



```
public class Item {  
    private int price;  
  
    public Item( int price ) {  
        this.price = price;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

Extras

Envio

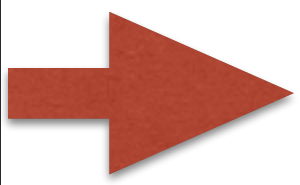
Impostos

Embalagem

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



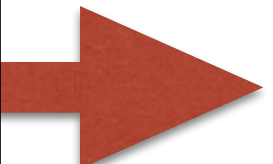
```
    public ItemExtras( Item item ) {  
        this.item = item;  
    }
```

```
    @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
```

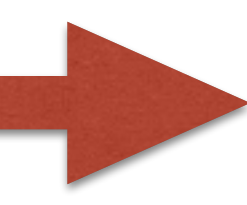
```
    Function<Integer, Integer> giftPacking = value -> value + 15;  
    giftPacking.apply( book.getPrice() ); //25
```

```
    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  
  
}
```



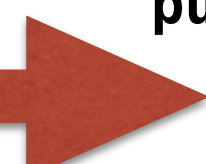
```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

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

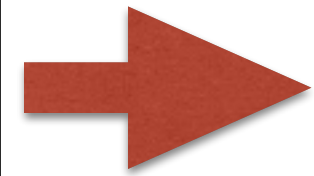
    public Item( int price, Function<Integer, Integer>... itemExtras ) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

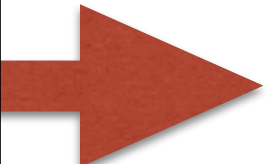
    public int getPrice() {
        int priceWithExtras = price;
        for ( Function<Integer, Integer> itemExtra : itemExtras ) {
            priceWithExtras = itemExtra.apply( priceWithExtras );
        }
        return priceWithExtras;
    }

    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  
}
```

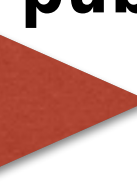


```
public class Packing {
```

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

```
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  
}
```

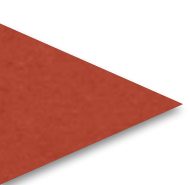
```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

    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;
    }
}
```



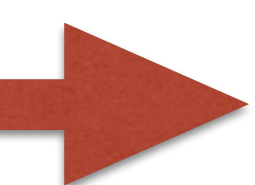
```
public class Item {
    private int price;
    private Function<Integer, Integer>[] itemExtras = new Function[]{};

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

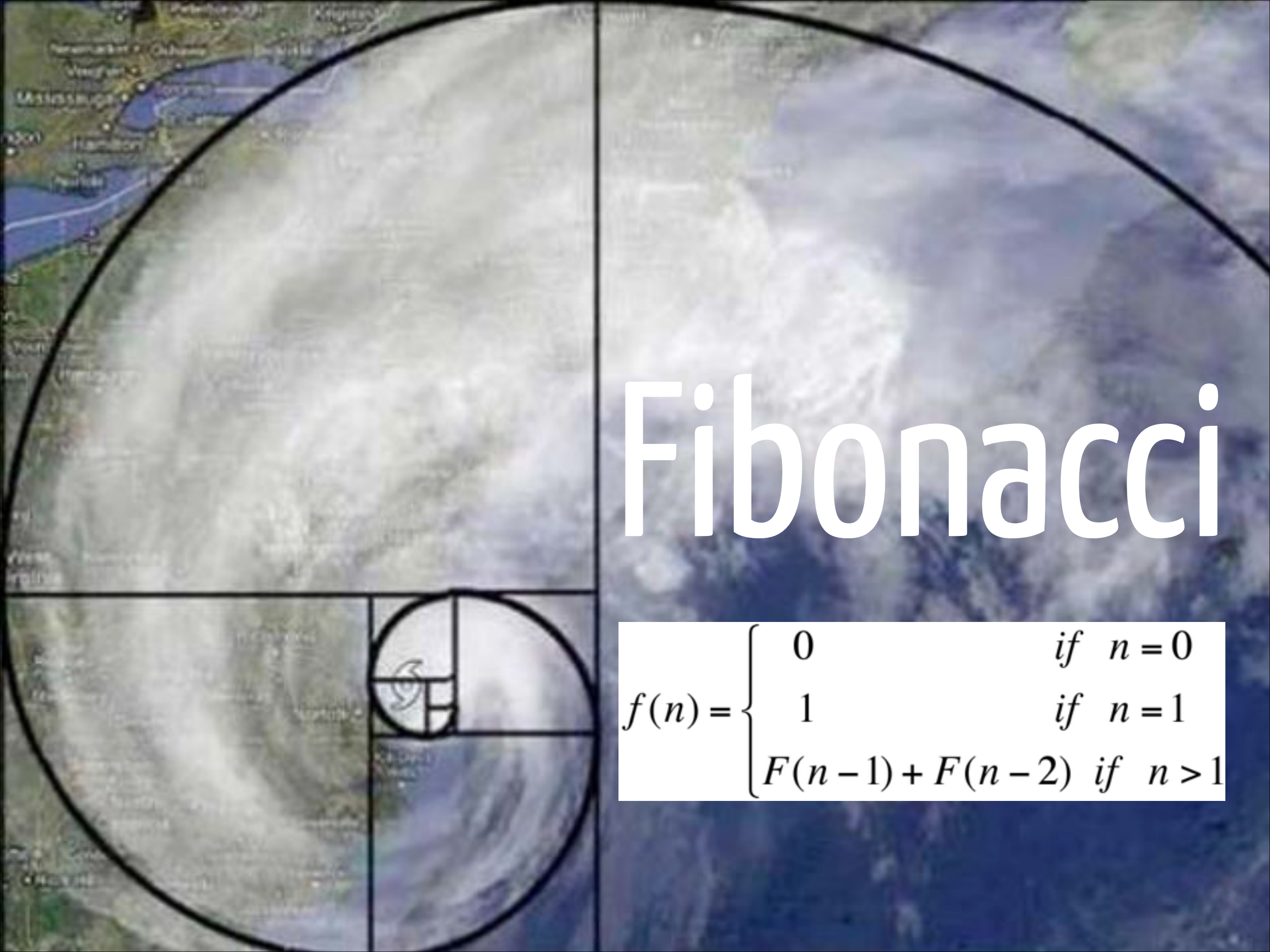
    public Item( int price, Function<Integer, Integer>... itemExtras ) {
        this.price = price;
        this.itemExtras = itemExtras;
    }

    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;
    }
}
```

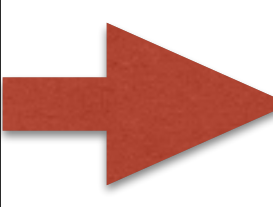


Recursões



Fibonacci

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



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

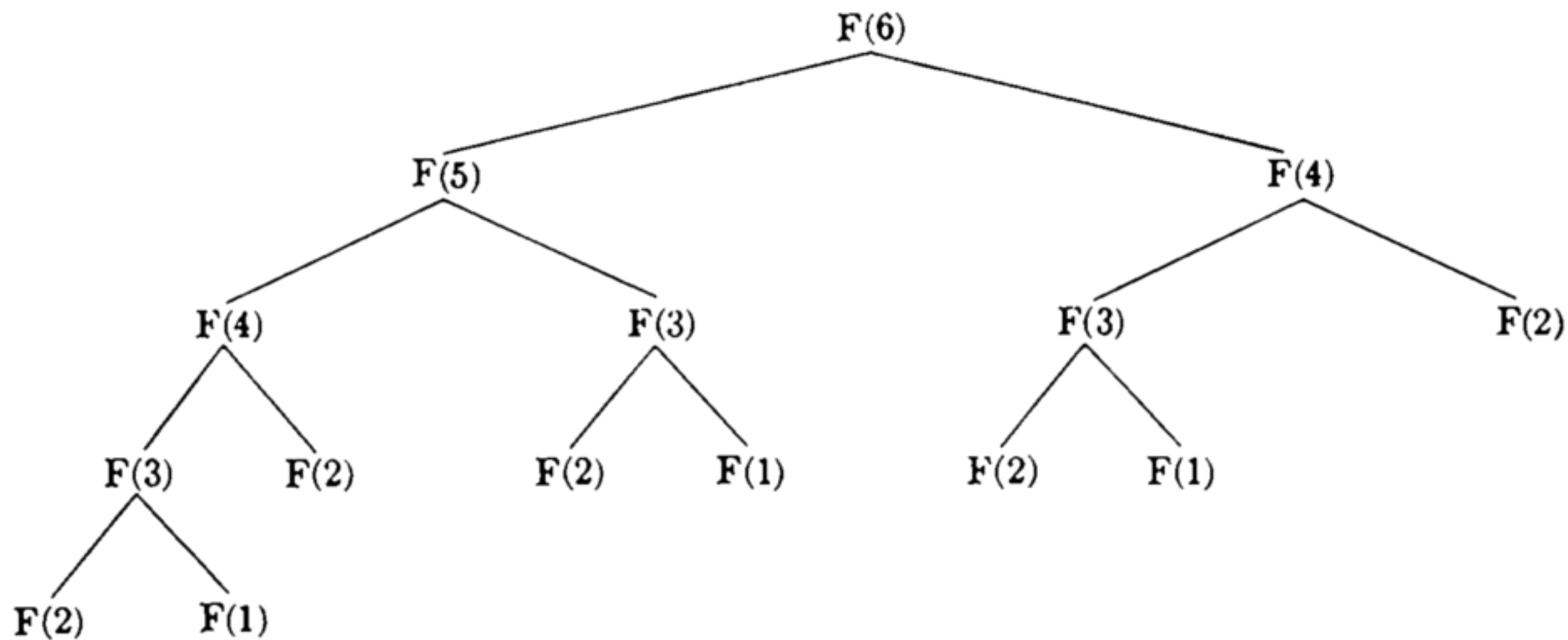


FIG. 1

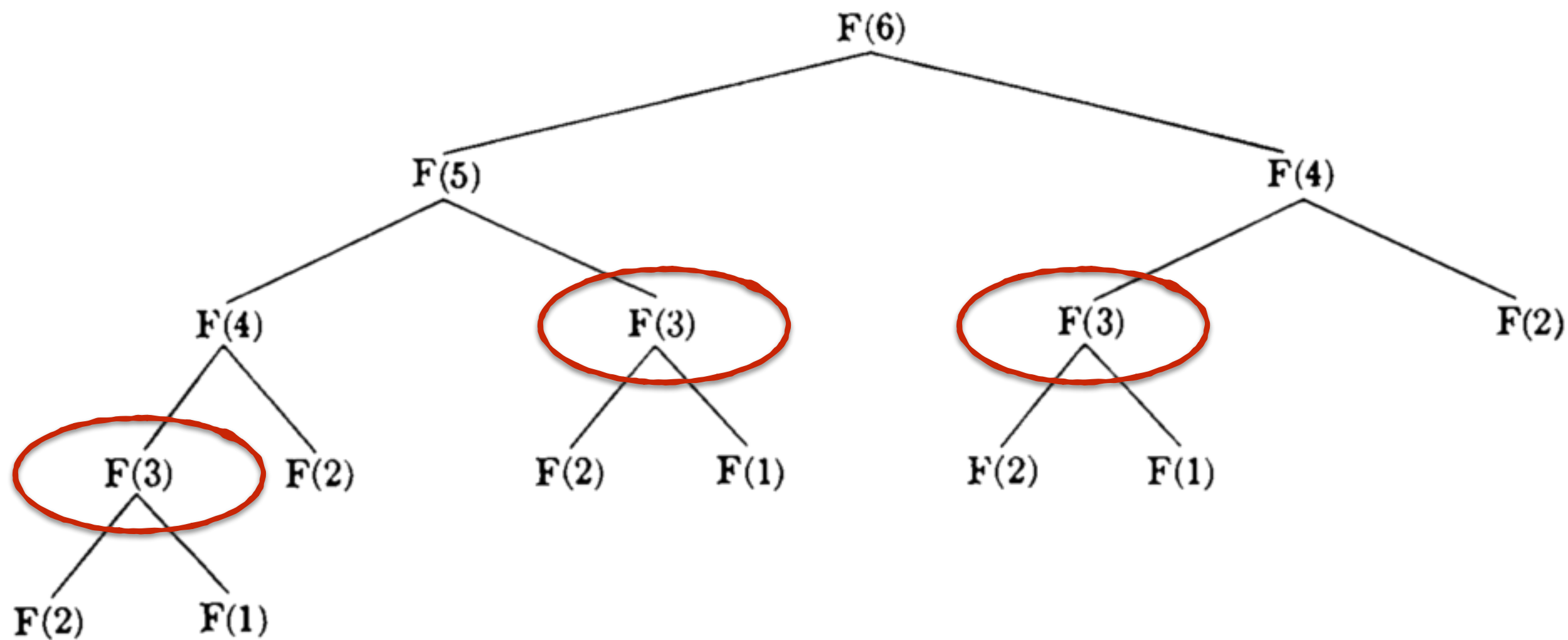


FIG. 1

Memoization

Pure Functions

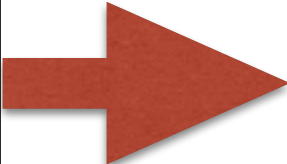
In computer programming, a function may be described as a pure function if both these statements about the function hold:

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;  
}
```

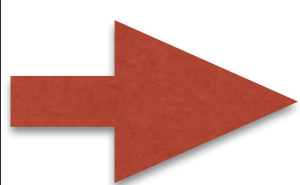


```
Integer doubleValue(Integer x) {  
    if (cache.containsKey(x)) {  
        return cache.get(x);  
    } else {  
        Integer result = x * 2;  
        cache.put(x, result);  
        return result;  
    }  
}
```

```
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;
```

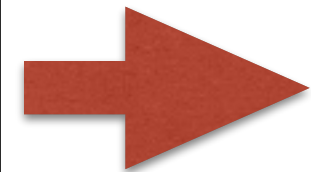
```
}
```

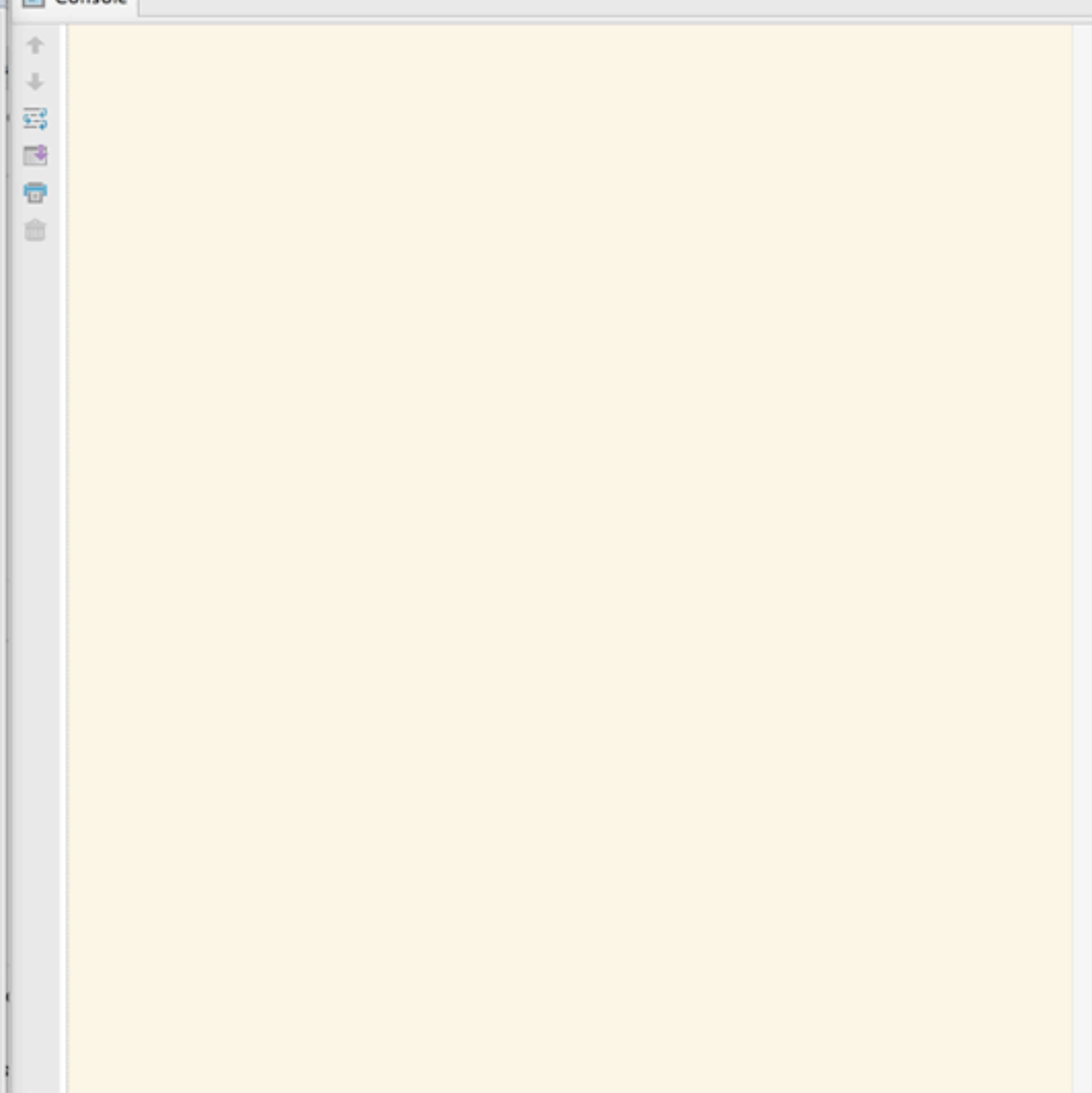
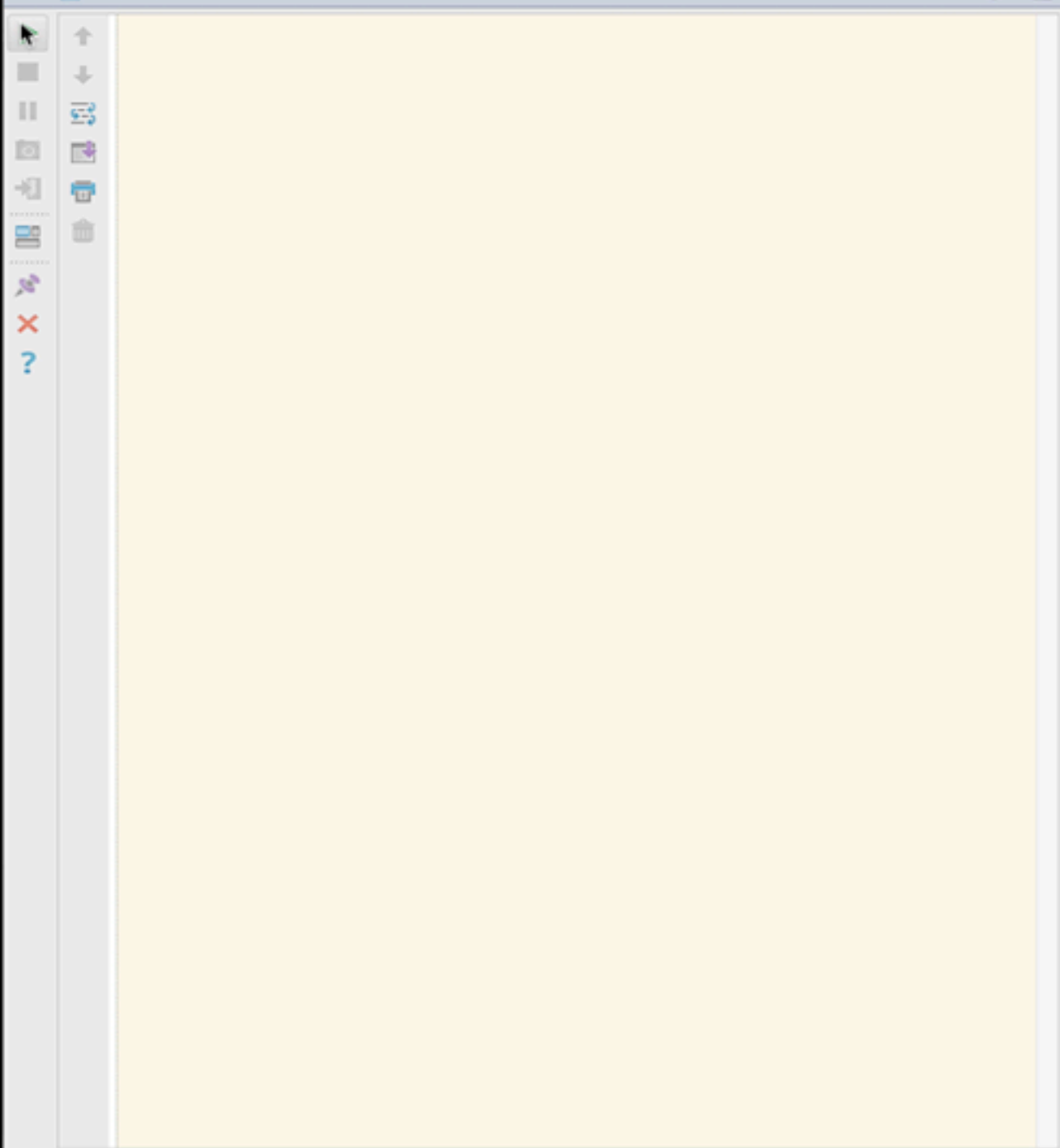
Java + FP
to the rescue

```
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 ) );  
}
```





Currying

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

$$f(2,3)$$

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

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

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

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

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

$$\text{CtoF}(x) = x * 9/5 + 32$$

```
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 ) {
```



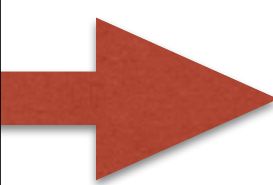
```
    DoubleUnaryOperator convertCtoF = curriedConverter( 9.0 / 5, 32 );
```

```
    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.