

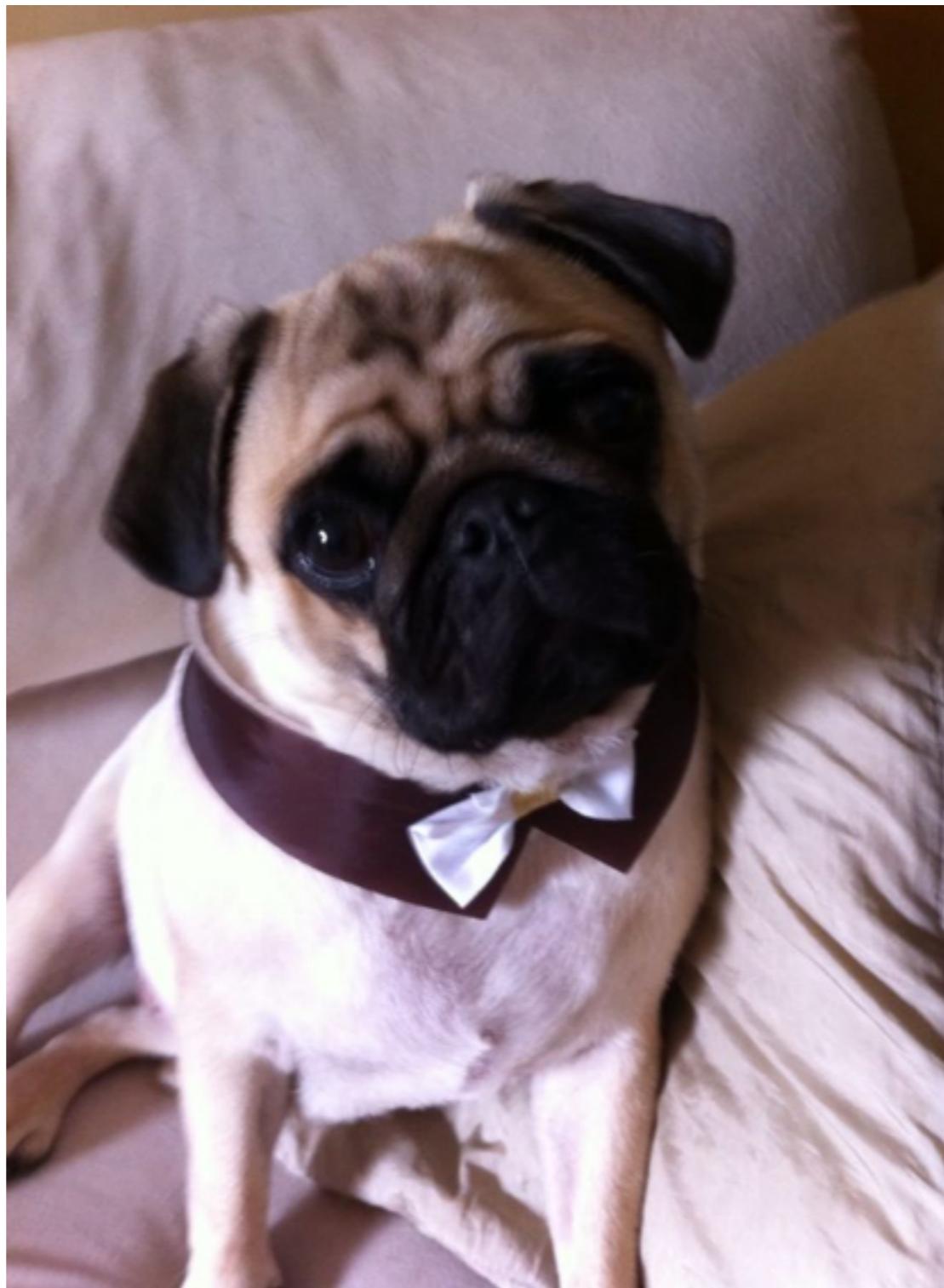
# Java Streams

## Deep Dive

Eder Ignatowicz  
Sr. Software Engineer  
JBoss by Red Hat



# Dora



# Bento

```
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 );
```

# Streams API

# Streams:

Manipula coleções de forma declarativa

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

# Streams:

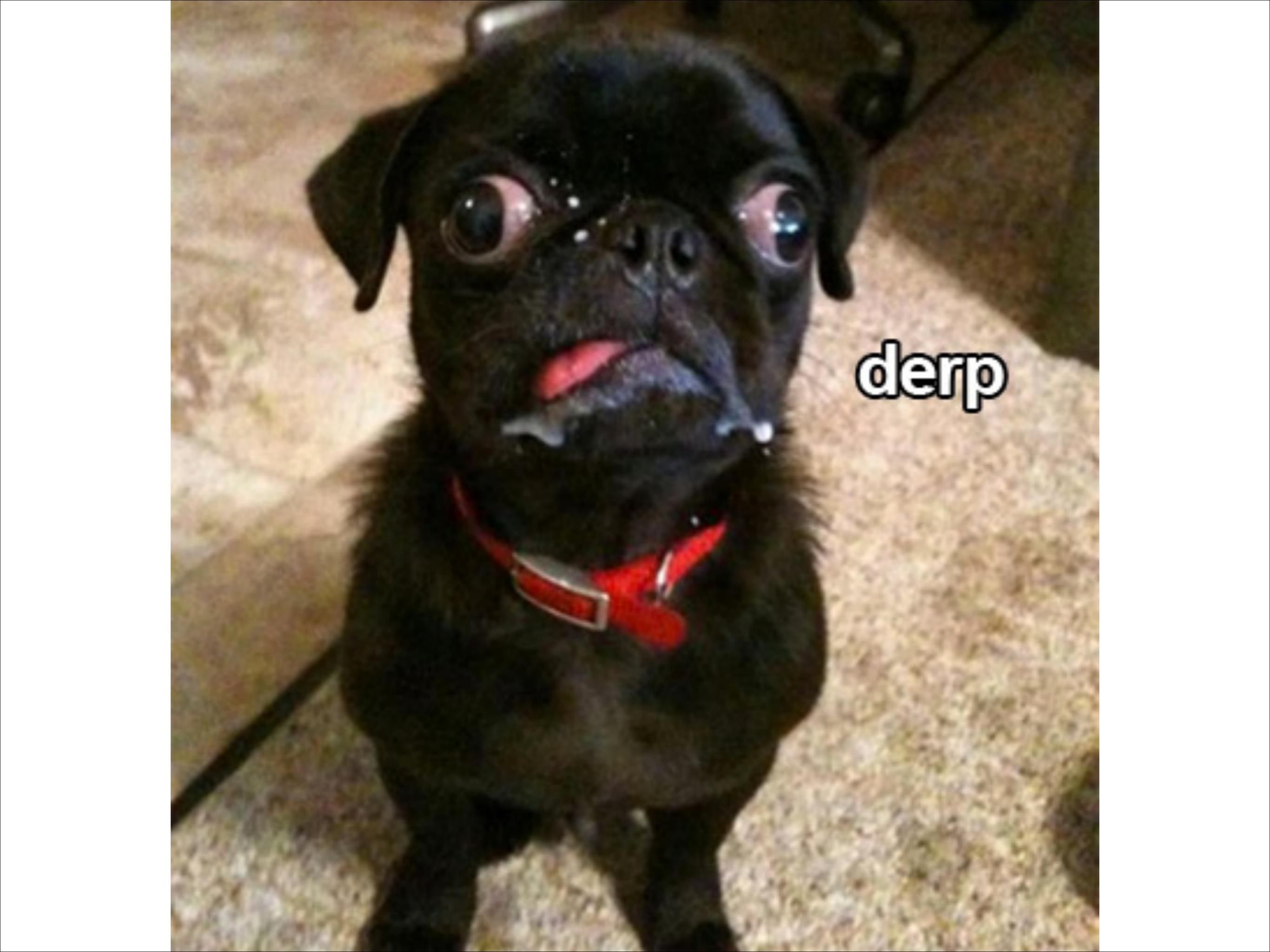
Manipula coleções de forma declarativa

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

# 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**

Java 8

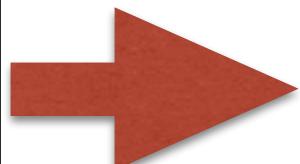
Streams API

# Em Java

```
List<String> fatName =  
  
    pugs.stream()  
  
        .filter( p -> dora.getWeight() > 9 )      Seleciona > 9 kg  
  
        .sorted( comparing( Pug::getWeight ) )  Ordena por peso  
  
        .map( Pug::getNome )                      Extrai o nome  
  
        .collect( toList() );                     Coleta em uma lista
```

# Em Java

```
List<String> fatName =
```



```
pugs.parallelStream()
```

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

Seleciona > 9 kg

```
.sorted( comparing( Pug::getWeight ) )
```

Ordena por peso

```
.map( Pug::getNome )
```

Extrai o nome

```
.collect( toList() );
```

Coleta em uma lista





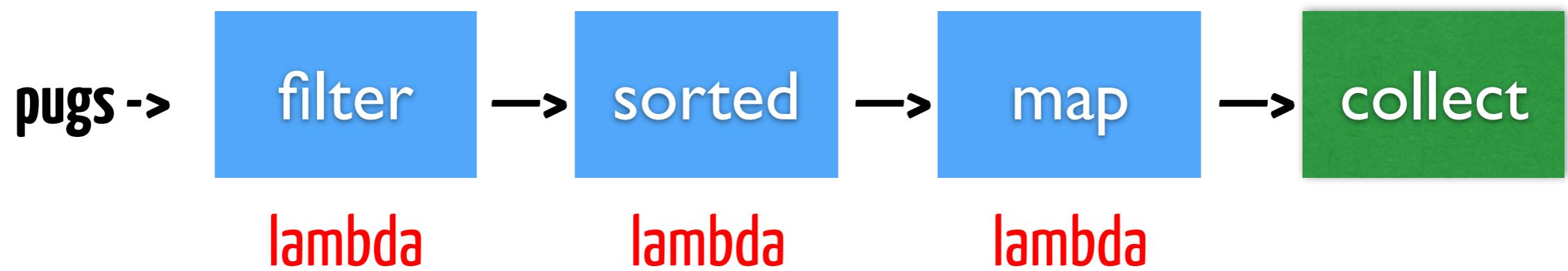
# Parallel streams

نão são

mágica!



# Stream Pipelines



# Streams API

Código:  
Declarativo  
Componentizável  
Paralelizável

# Streams

Sequência de elementos de uma fonte que suporta operações de processamento em seus dados

# Streams

Sequência de elementos de uma  
**fonte** que suporta operações de  
processamento em seus dados

# Streams

Sequência de elementos de uma  
fonte que suporta **operações de**  
**processamento em seus dados**

# Streams

Uma fonte de dados para a query

Uma cadeia de operações intermediárias  
(pipeline)

Uma operação terminal que gera o resultado

Vamos a prática

```
public Venda( Vendedor vendedor,  
               int ano,  
               int valor ) {  
    this.vendedor = vendedor;  
    this.ano = ano;  
    this.valor = valor;  
}
```

```
public Vendedor( String nome,  
                  String cidade ) {  
    this.nome = nome;  
    this.cidade = cidade;  
}
```

```
Vendedor eder = new Vendedor("Eder", "Campinas");
Vendedor pedro = new Vendedor("Pedro", "Apucarana");
Vendedor luciano = new Vendedor("Luciano", "Piracicaba");
Vendedor junior = new Vendedor("Junior", "Londrina");

List<Venda> transactions = Arrays.asList(
    new Venda( eder, 2015, 100 ),
    new Venda( eder, 2016, 200 ),
    new Venda( pedro, 2015, 300 ),
    new Venda( luciano, 2015, 500 ),
    new Venda( luciano, 2015, 400 ),
    new Venda( junior, 2016, 500 ));
```

Quais são as vendas que fizemos em  
2015? Ordenadas?

```
List<Venda> vendas2015 =  
    transactions  
        .stream()
```



```
List<Venda> vendas2015 =  
    transactions  
        .stream()  
        .filter( venda -> venda.getAno() == 2015 )
```



```
List<Venda> vendas2015 =  
    transactions  
        .stream()  
        .filter( venda -> venda.getAno() == 2015 )  
        .sorted( comparing( Venda::getValor ) )
```



```
List<Venda> vendas2015 =  
    transactions  
        .stream()  
        .filter( venda -> venda.getAno() == 2015 )  
        .sorted( comparing( Venda::getValor ) )  
        .collect( toList() );
```



```
List<Venda> vendas2015 =  
    transactions  
        .stream()  
        .filter( venda -> venda.getAno() == 2015 )  
        .sorted( comparing( Venda::getValor ) )  
        .collect( toList() );
```

→ vendas2015.forEach(System.out::println);

```
Venda{vendedor=Vendedor{nome='Eder', cidade='Campinas'}, ano=2015, valor=100}  
Venda{vendedor=Vendedor{nome='Pedro', cidade='Apucarana'}, ano=2015, valor=300}  
Venda{vendedor=Vendedor{nome='Luciano', cidade='Piracicaba'}, ano=2015, valor=400}  
Venda{vendedor=Vendedor{nome='Luciano', cidade='Piracicaba'}, ano=2015, valor=500}
```

**Em que cidades temos vendedores?**

```
List<String> cidadesAtendidas =  
    vendas.stream()  
        .map( venda ->venda.getVendedor().getCIDADE() )  
        .distinct()  
        .collect( toList() );
```

Campinas  
Apucarana  
Piracicaba  
Londrina

**Qual foi a maior venda?**

```
OptionalInt maiorVenda =  
    vendas.stream()  
        .mapToInt(Venda::getValor)  
        .reduce( Integer::max );  
  
maiorVenda.ifPresent( i -> System.out.println(i));
```

Total de vendas?

```
OptionalInt total =  
    vendas.stream()  
        .mapToInt(Venda::getValor)  
        .reduce( Integer::sum );  
  
total.ifPresent( i -> System.out.println(i));
```

2000

Quais são as vendas de cada vendedor?  
Ordenadas?

```
Map<Vendedor, List<Venda>> vendedorPorVendas =  
vendas.stream()  
.sorted( comparing( Venda::getValor ) )  
.collect( groupingBy( Venda::getVendedor ) );
```

```
Map<Vendedor, List<Venda>> vendedorPorVendas =  
  
vendas.stream()  
.sorted( comparing( Venda::getValor ) )  
.collect( groupingBy( Venda::getVendedor ) );  
  
{Vendedor{nome='Junior', cidade='Londrina'}=[Venda{vendedor=Vendedor{nome='Junior',  
cidade='Londrina'}, ano=2016, valor=500}],  
  
Vendedor{nome='Eder', cidade='Campinas'}=[Venda{vendedor=Vendedor{nome='Eder',  
cidade='Campinas'}, ano=2015, valor=100}, Venda{vendedor=Vendedor{nome='Eder',  
cidade='Campinas'}, ano=2016, valor=200}],  
  
Vendedor{nome='Pedro', cidade='Apucarana'}=[Venda{vendedor=Vendedor{nome='Pedro',  
cidade='Apucarana'}, ano=2015, valor=300}],  
  
Vendedor{nome='Luciano', cidade='Piracicaba'}  
=[Venda{vendedor=Vendedor{nome='Luciano', cidade='Piracicaba'}, ano=2015,  
valor=400}, Venda{vendedor=Vendedor{nome='Luciano', cidade='Piracicaba'}, ano=2015,  
valor=500}]}  

```

# Refactoring Loops to Collection Pipelines

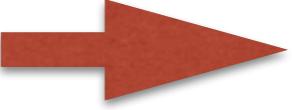
```
public class Client {  
  
    private String name;  
    private String email;  
    private Company company;  
  
    public Client( String name, String email, Company company ) {  
        this.name = name;  
        this.email = email;  
        this.company = company;  
    }  
  
    public Client( String name ) {  
        this.name = name;  
    }  
  
    public Client( String name, String email ) {  
        this.name = name;  
        this.email = email;  
    }  
  
    ...  
}
```

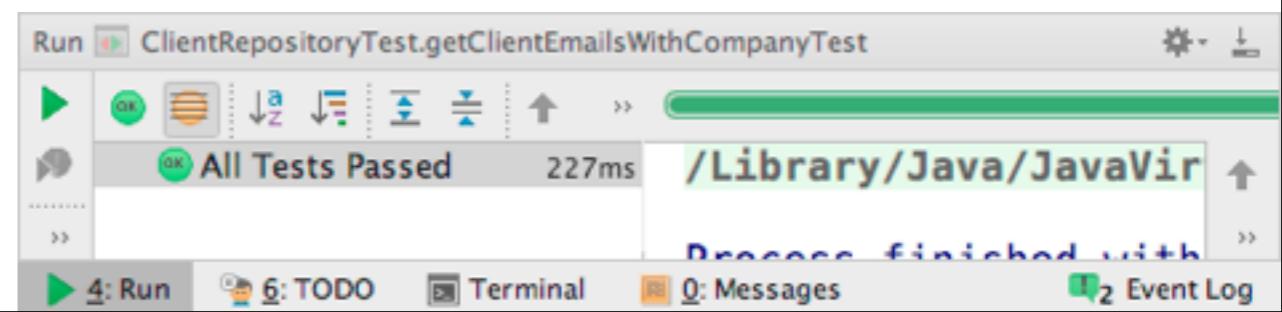
```
public class ClientRepositoryTest {

    private ClientRepository repo;

    @Before
    public void setup() {
        Company empresa = new Company( "RedHat" );
        Client completo1 = new Client( "Completo1", "completo1@redhat.com", empresa );
        Client completo2 = new Client( "Completo2", "completo2@redhat.com", empresa );
        Client semEmpresa = new Client( "SemEmpresa", "semEmpresa@ederign.me" );
        Client somenteNome = new Client( "SomenteNome" );
        repo = new ClientRepository(
                Arrays.asList( completo1, semEmpresa, completo2, somenteNome ) );
    }

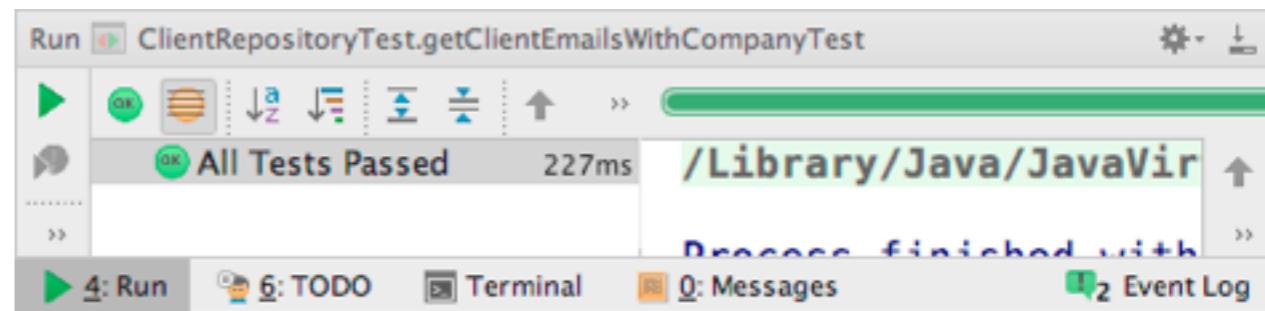
    @Test
    public void getClientEmailsWithCompanyTest() {
        List<String> clientMails = repo.getClientMails();
        assertEquals( 2, clientMails.size() );
        assertTrue( clientMails.contains( "completo1@redhat.com" ) );
        assertTrue( clientMails.contains( "completo2@redhat.com" ) );
        assertTrue( !clientMails.contains( "semEmpresa@ederign.me" ) );
    }
}
```

```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
  
     for ( Client client : clients ) {  
        if ( client.getCompany() != null ) {  
            String email = client.getEmail();  
            if ( email != null ){  
                emails.add( email );  
            }  
        }  
    }  
  
    return emails;  
}
```



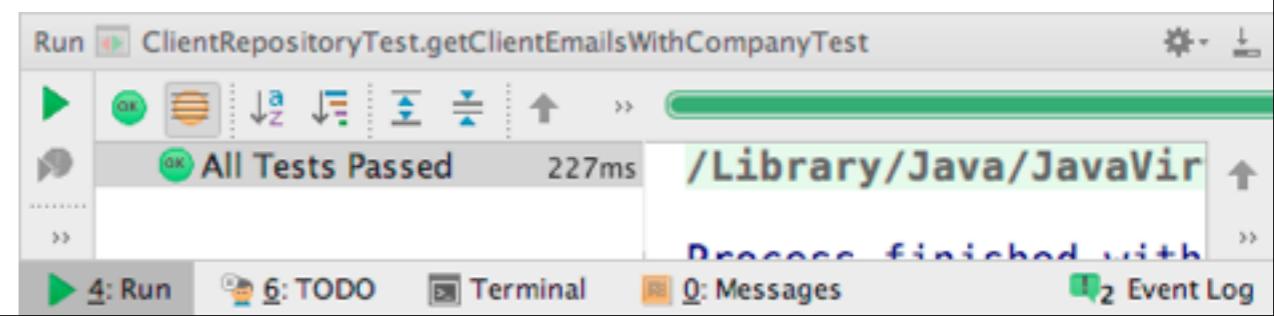
```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
    List<Client> pipeline = clients;  
    for ( Client client : pipeline ) {  
        if ( client.getCompany() != null ) {  
            String email = client.getEmail();  
            if ( email != null ){  
                emails.add( email );  
            }  
        }  
    }  
    return emails;  
}  
}
```

Extract  
Variable



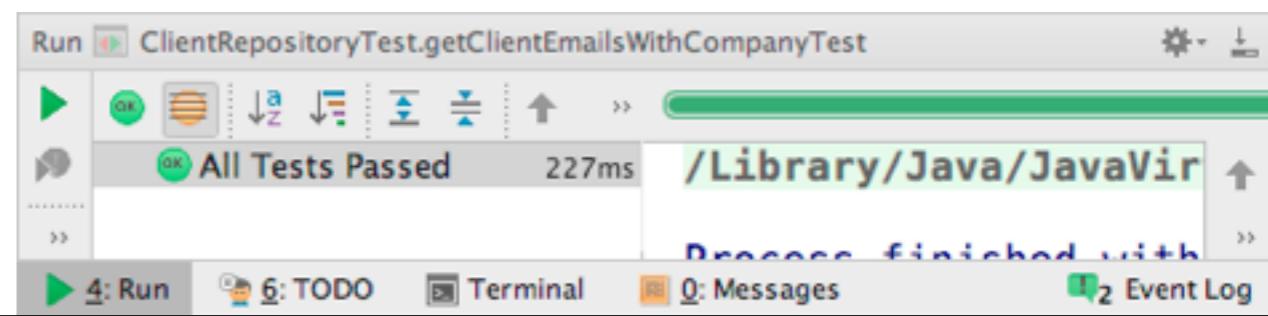
```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
    List<Client> pipeline = clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .collect( Collectors.toList() );  
    for ( Client client : pipeline ) {  
        if ( client.getCompany() != null ) {  
            String email = client.getEmail();  
            if ( email != null ) {  
                emails.add( email );  
            }  
        }  
    }  
    return emails;  
}
```

Filter  
Operation



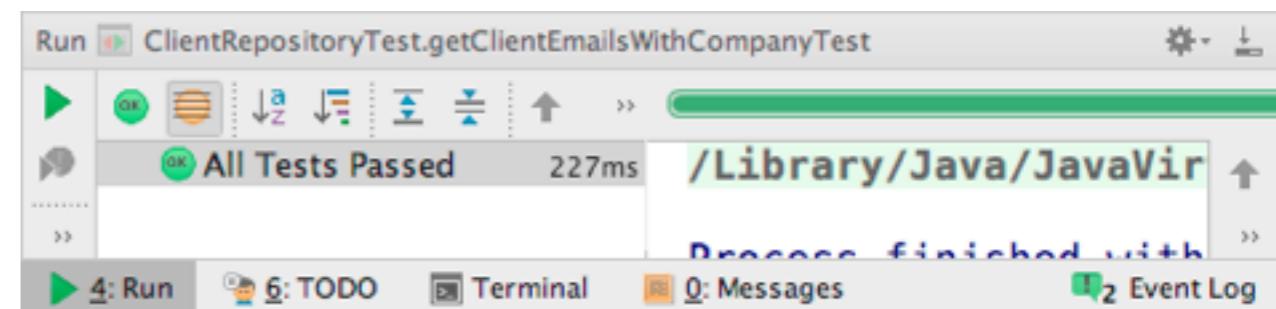
```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
    List<String> pipeline = clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .map( c -> c.getEmail() )  
        .collect( Collectors.toList() );  
    for ( String mail : pipeline ) {  
        String email = client.getEmail();  
        if ( mail != null ) {  
            emails.add( mail );  
        }  
    }  
  
    return emails;  
}
```

Map Operation



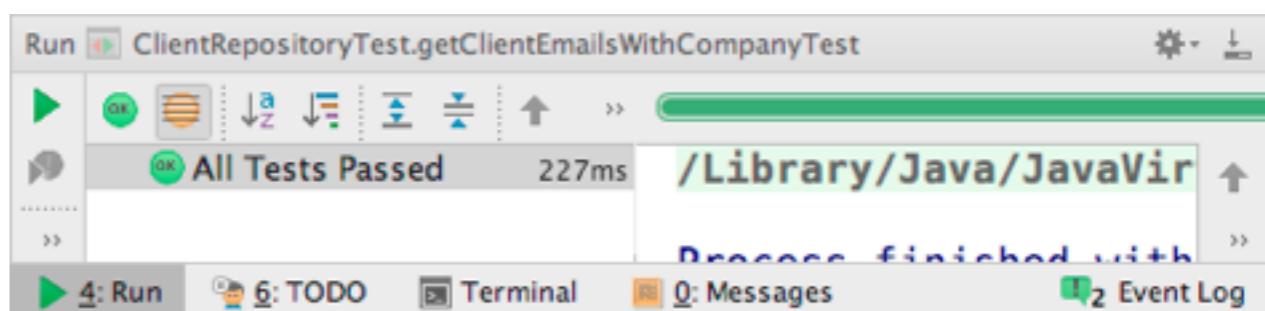
```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
    List<String> pipeline = clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .map( c -> c.getEmail() )  
        .filter( m -> m != null )  
        .collect( Collectors.toList() );  
  
    for ( String mail : pipeline ) {  
        if ( mail != null ) {  
            emails.add( mail );  
        }  
    }  
  
    return emails;  
}
```

Filter  
Operation

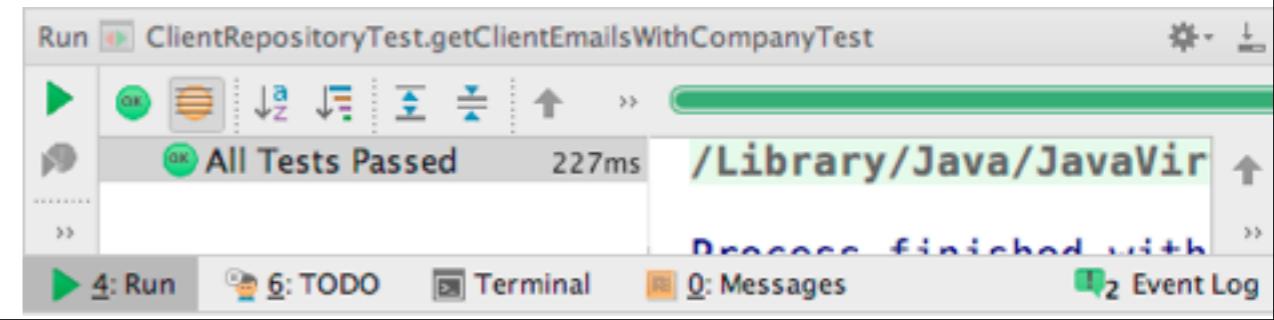


```
public List<String> getClientMails() {  
    ArrayList<String> emails = new ArrayList<>();  
    return clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .map( c -> c.getEmail() )  
        .filter( m -> m != null )  
        .collect( Collectors.toList() );  
    for ( String mail : pipeline ) {  
        if ( mail != null ) {  
            emails.add( mail );  
        }  
    }  
    return emails;  
}
```

Pipeline



```
public List<String> getClientMails() {  
    return clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .map( c -> c.getEmail() )  
        .filter( m -> m != null )  
        .collect( Collectors.toList() );  
}
```



# Happy Pug

is happy

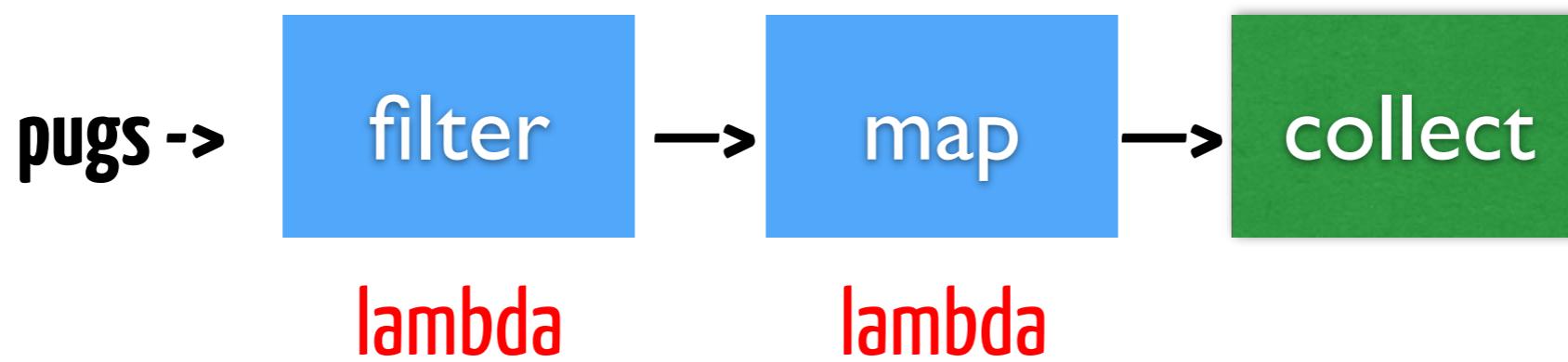


Saturday, October 5, 13

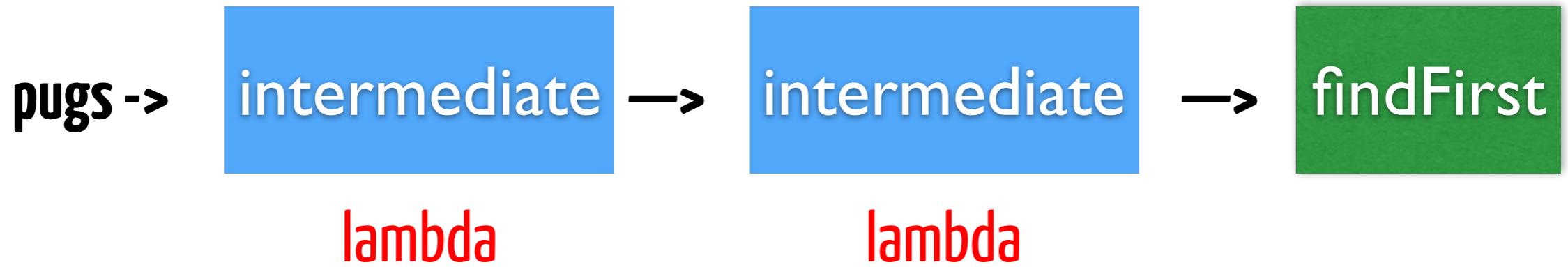


Streams  
são lazy

# Stream Pipelines



# Stream Pipelines

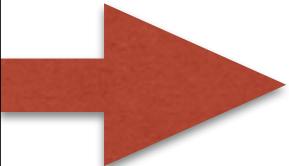


# Lazy Streams

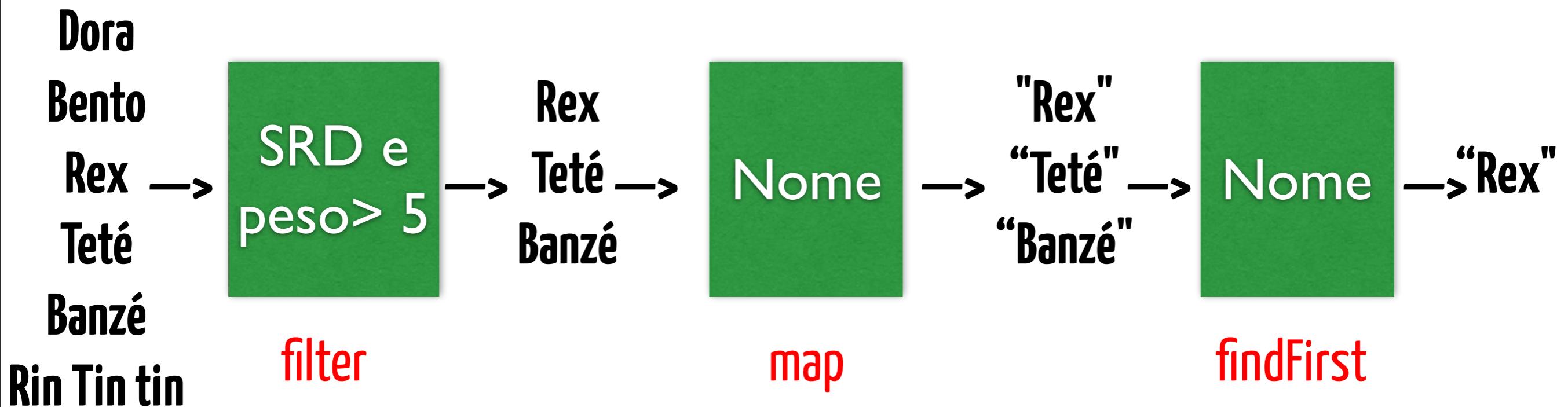
```
List<Dog> dogs = Arrays.asList(  
    new Dog( "Dora", 10, Dog.BREED.PUG ),  
    new Dog( "Bento", 13, Dog.BREED.PUG ),  
    new Dog( "Rex", 8, Dog.BREED.SRD ),  
    new Dog( "Tetezinha", 6, Dog.BREED.SRD ),  
    new Dog( "Banze", 7, Dog.BREED.SRD ),  
    new Dog( "Rufus", 15, Dog.BREED.BULLDOG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

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

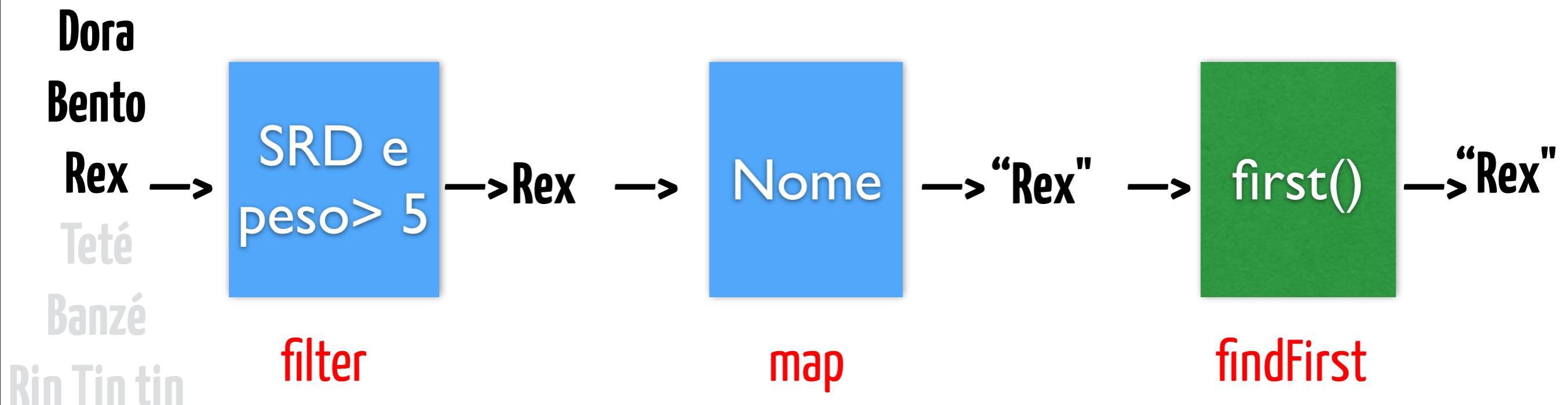
```
String nomePrimeiroSRDMaiorDoQue5Kg =  
    dogs.stream()  
  
        .filter( dog -> {  
            return dog.getBreed().equals( Dog.BREED.SRD )  
                && dog.getWeight() > 5;  
        } )  
  
        .map( dog -> {  
            return dog.getName();  
        } )  
  
        .findFirst()  
  
        .get();
```



# Eager Streams



# Lazy Stream

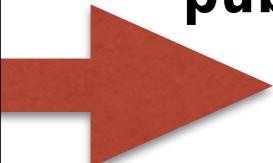


```
List<Dog> dogs = Arrays.asList(  
    new Dog( "Dora", 10, Dog.BREED.PUG ),  
    new Dog( "Bento", 13, Dog.BREED.PUG ),  
    new Dog( "Rex", 8, Dog.BREED.SRD ),  
    new Dog( "Tetezinha", 6, Dog.BREED.SRD ),  
    new Dog( "Banze", 7, Dog.BREED.SRD ),  
    new Dog( "Rufus", 15, Dog.BREED.BULLDOG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

**Streams**  
**"infinitos"**

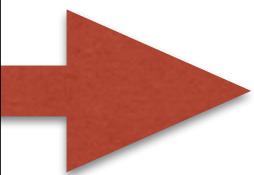
Criar uma lista de números  
primos infinita



```
public static boolean isPrime( final int number ) {  
    return number > 1 &&  
        IntStream.rangeClosed( 2, ( int ) Math.sqrt( number ) )  
            .noneMatch( divisor -> number % divisor == 0 );  
}
```

```
public static int primeAfter( final int number ) {  
    if ( isPrime( number + 1 ) ) {  
        return number + 1;  
    } else {  
        return primeAfter( number + 1 );  
    }  
}
```

```
public static IntStream primesInfinityStream( int fromNumber) {  
    return IntStream.iterate( primeAfter( fromNumber - 1 ),  
                             Primes::primeAfter );  
}
```



```
primesInfinityStream( 1 )  
    .limit( 10 )  
    .forEach( i -> System.out.print( i+ ", " ) );  
  
primesInfinityStream( 1000 )  
    .limit( 5 )  
    .forEach( i -> System.out.print( i+ ", " ) );
```

2, 3, 5, 7, 11, 13, 17, 19, 23, 29,

1009, 1013, 1019, 1021, 1031

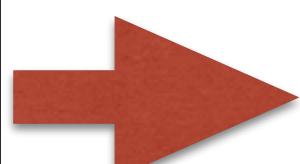
# FlatMap

```
Developer eder = new Developer();
eder.bestBooks( "Java 8 in Action", "SICP", "The Little Schemer" );
```

```
Developer dora = new Developer();
dora.bestBooks( "Effective Java", "Pragmatic Programmer", "SICP" );
```

```
List<Developer> devs = Arrays.asList( eder, dora );
```

```
List<String> booksNames = devs
    .stream()
    .map( dev -> dev.getBooks() ) //Stream<Set<String>>
    .flatMap( books -> books.stream() )//Stream<String>
    .distinct()
    .collect( Collectors.toList() );
booksNames.forEach( System.out::println );
```



# IntStream

DoubleStream, LongStream...

```
IntStream.of( 1, 2, 3);
           // > 1, 2, 3
IntStream.range(1, 3);
           // > 1, 2
```

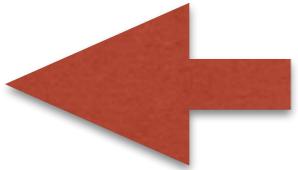
```
dogs.stream() //Stream<Dog>  
    .filter( dog -> dog.getBreed().equals( Dog.BREED.PUG ) ) //Stream<Dog>  
    .map( Dog::getWeight ) //Stream<Integer> boxing :(  
    .filter( weight -> weight > 10 ) //Stream<Integer> boxing reboxing  
    .mapToInt( weight -> weight ) //IntStream  
    .sum();
```

0.185s

```
dogs.stream()//Stream<Dog>  
    .filter( dog -> dog.getBreed().equals( Dog.BREED.PUG ) )//Stream<Dog>  
    .mapToInt( Dog::getWeight ) //IntStream  
    .filter( weight -> weight > 10 ) //IntStream  
    .sum();
```

0.004s

```
dogs.stream()//Stream<Dog>  
    .filter( dog -> dog.getBreed().equals( Dog.BREED.PUG ) )//Stream<Dog>  
    .mapToInt( Dog::getWeight ) //IntStream  
    .filter( weight -> weight > 10 ) //IntStream  
    .max();
```



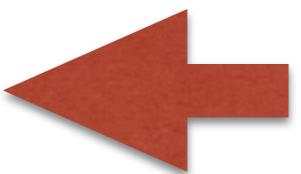
0.004s

```
dogs.stream()//Stream<Dog>  
    .filter( dog -> dog.getBreed().equals( Dog.BREED.PUG ) )//Stream<Dog>  
    .mapToInt( Dog::getWeight ) //IntStream  
    .filter( weight -> weight > 10 ) //IntStream  
    .summaryStatistics();
```

```
IntSummaryStatistics{count=224, sum=4490, min=11,  
average=20.044643, max=30}
```

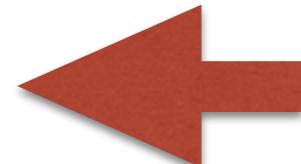
# ParallelStreams

```
dogs.stream()//Stream<Dog>  
    .mapToInt( Dog::getWeight ) //IntStream  
    .filter( weight -> weight > 10 ) //IntStream  
    .sum();
```



0.129s

```
dogs.parallelStream() // Stream<Dog>  
    .mapToInt( Dog::getWeight ) // IntStream  
    .filter( weight -> weight > 10 ) // IntStream  
    .sum();
```

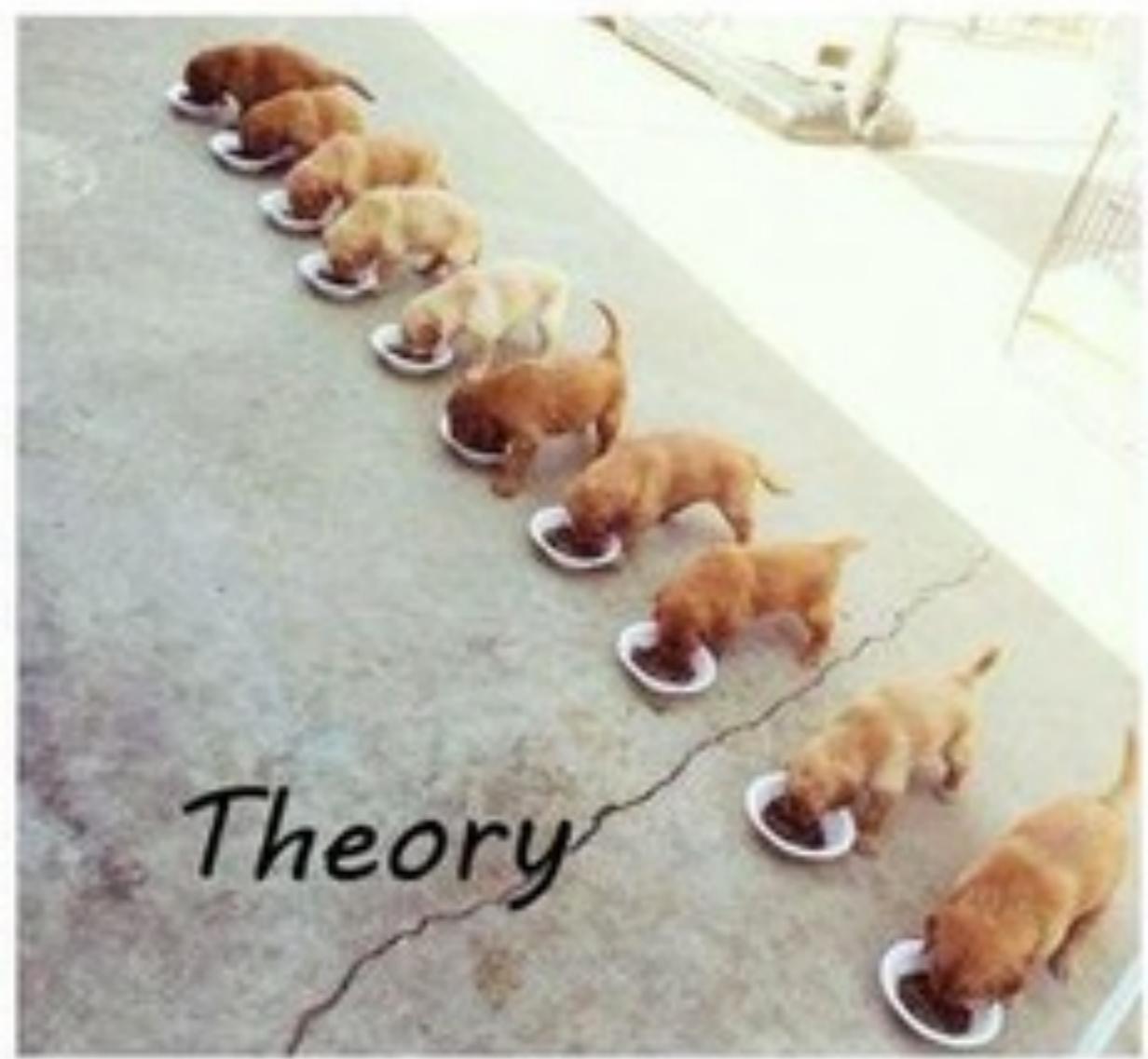


0.017s

# Happy Pug

is happy

# Multithreaded programming



```
dogs.parallelStream()
    • mapToInt( Dog::getWeight )
    • reduce( Math::max )
    • ifPresent( p -> System.out.println( "Maior peso: " + p ) );
```

`java.util.Spliterator`  
`java.util.concurrent.ForkJoinPool.commonPool()`

# java.util.Spliterator

# Spliterator = splitter + iterator

## Interface Spliterator<T>

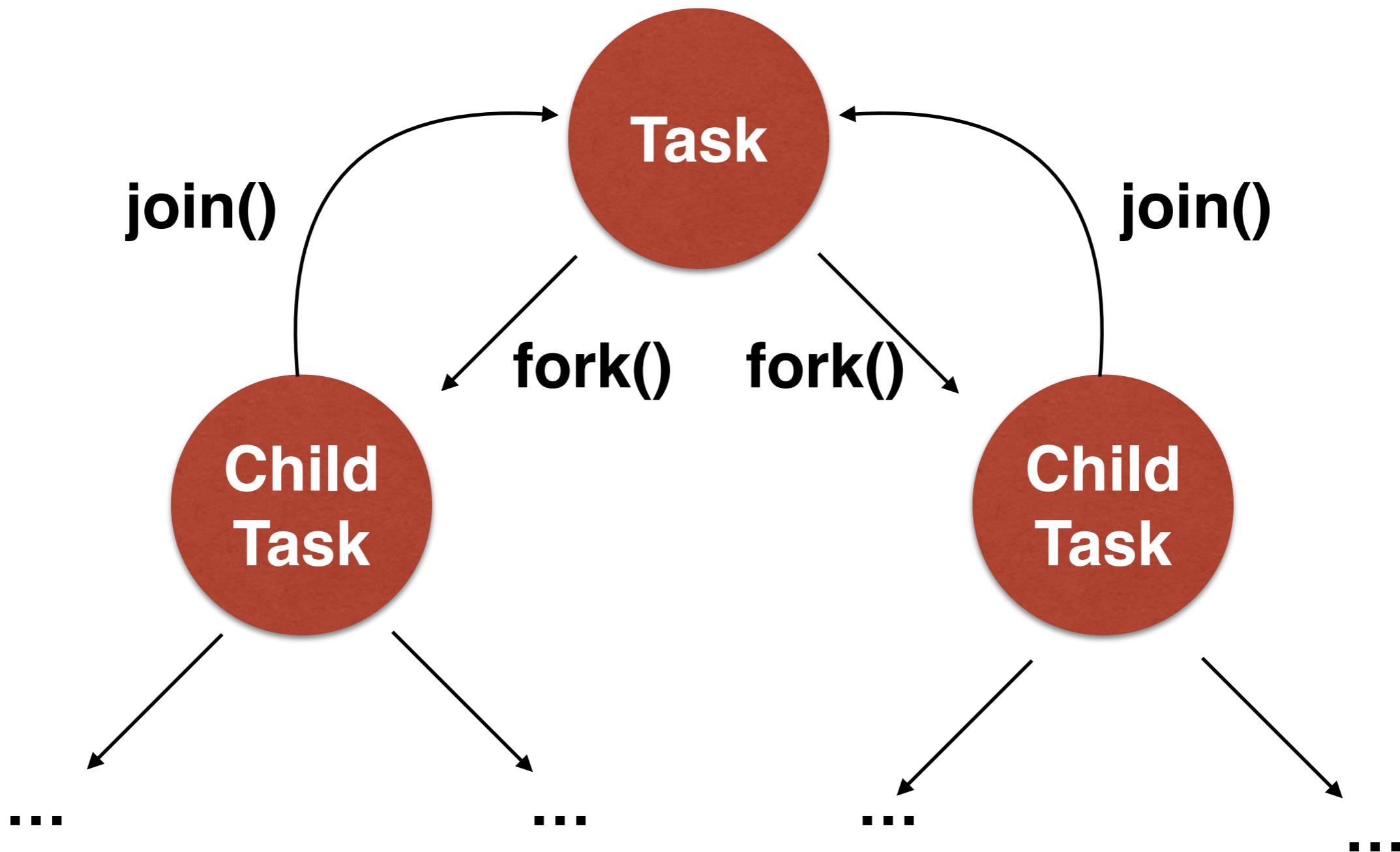
### Method Summary

All Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type		Method and Description	
int		<b>characteristics()</b> Returns a set of characteristics of this Spliterator and its elements.	
long		<b>estimateSize()</b> Returns an estimate of the number of elements that would be encountered by a <code>forEachRemaining(java.util.function.Consumer&lt;? super T&gt;)</code> traversal, or returns <code>Long.MAX_VALUE</code> if infinite, unknown, or too expensive to compute.	
default void		<b>forEachRemaining(Consumer&lt;? super T&gt; action)</b> Performs the given action for each remaining element, sequentially in the current thread, until all elements have been processed or the action throws an exception.	
default Comparator<? super T>		<b>getComparator()</b> If this Spliterator's source is <b>SORTED</b> by a <b>Comparator</b> , returns that Comparator.	
default long		<b>getExactSizeIfKnown()</b> Convenience method that returns <code>estimateSize()</code> if this Spliterator is <b>SIZED</b> , else -1.	
default boolean		<b>hasCharacteristics(int characteristics)</b> Returns <code>true</code> if this Spliterator's <code>characteristics()</code> contain all of the given characteristics.	
boolean		<b>tryAdvance(Consumer&lt;? super T&gt; action)</b> If a remaining element exists, performs the given action on it, returning <code>true</code> ; else returns <code>false</code> .	
<b>Spliterator&lt;T&gt;</b>		<b>trySplit()</b> If this spliterator can be partitioned, returns a Spliterator covering elements, that will, upon return from this method, not be covered by this Spliterator.	

	seq	parallel
ArrayList	8.33 ms	6.33 ms
LinkedList	12.74 ms	19.57 ms
HashSet	20.76 ms	16.01 ms
TreeSet	19.79 ms	15.49 ms

**java.util.concurrent.ForkJoinPool.commonPool()**

# Fork Join Framework

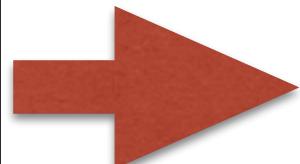


# CommonPool

## Singleton fork-join pool instance

```
public static String query( String question ) {
    List<String> engines = new ArrayList<>();
    engines.add( "https://www.google.com/?q=" );
    engines.add( "https://duckduckgo.com/?q=" );
    engines.add( "https://www.bing.com/search?q=" );

    // get element as soon as it is available
    Optional<String> result = engines.stream()
        .parallel().map( ( base ) -> {
        String url = base + question;
        // open connection and fetch the result
        return Util.read( url );
    } ).findAny();
    return result.get();
}
```



```
public static String query( String question ) {
    List<String> engines = new ArrayList<>();
    engines.add( "https://www.google.com/?q=" );
    engines.add( "https://duckduckgo.com/?q=" );
    engines.add( "https://www.bing.com/search?q=" );

    // get element as soon as it is available
    Optional<String> result = engines.stream()
        .parallel().map( ( base ) -> {
        String url = base + question;
        // open connection and fetch the result
        return Util.read( url );
    } ).findAny();
    return result.get();
}
```

```
dogs.parallelStream()
    .mapToInt( Dog::getWeight )
    .reduce( Math::max )
    .ifPresent( p -> System.out.println( "Maior peso: " + p ) );
```

I HAD A  
BAD  
DAY





**WITH GREAT POWER**

**COMES GREAT RESPONSIBILITY**

```
public List<String> getClientMails() {  
    return clients  
        .stream()  
        .filter( c -> c.getCompany() != null )  
        .map( c -> c.getEmail() )  
        .filter( m -> m != null )  
        .collect( Collectors.toList() );  
}
```

# Happy Pug

is happy

# Obrigado!!!



@ederign



redhat.