

LP Orientada a Objetos II

Wrappers e Streams

Diego Addan

DS142 - UFPR - 2023

Para hoje

Classes Utilitárias

Streams

Exercícios

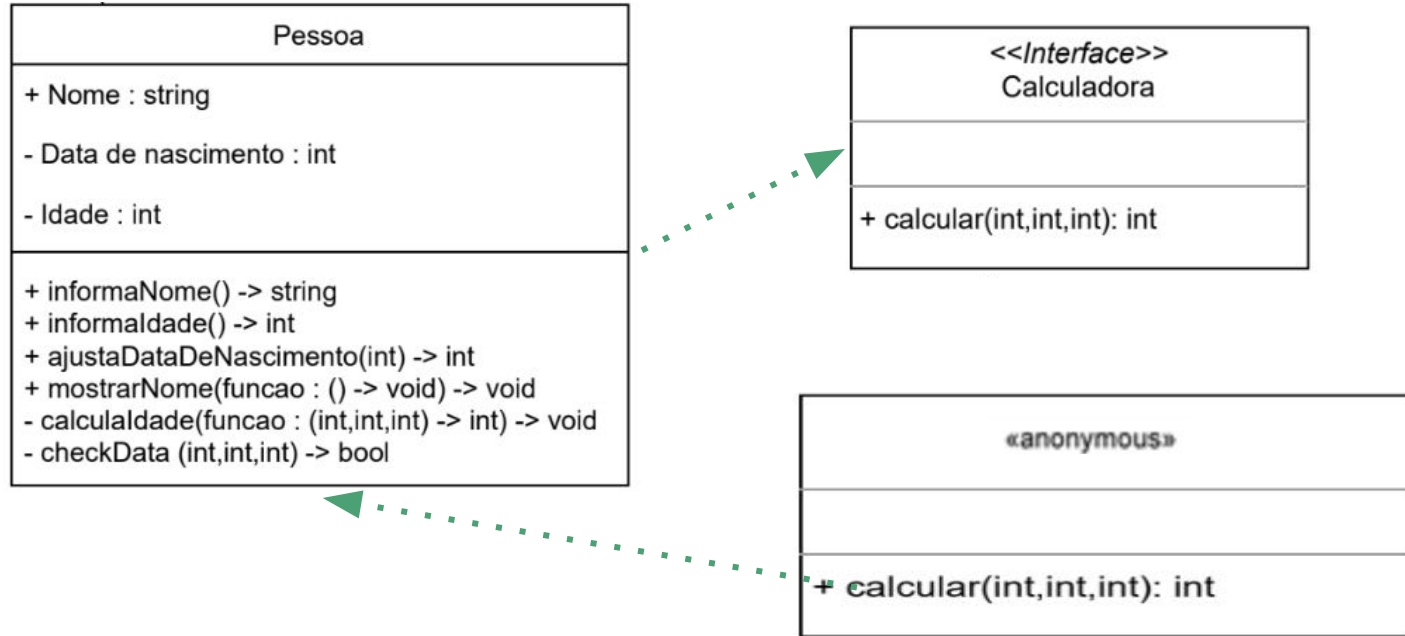
Aula passada

Exercício

1) Criar a classe *Pessoa* com as seguintes características:

- atributos: idade e dia, mês e ano de nascimento, nome da pessoa
- métodos:
 - *calculaIdade()*, que recebe a data atual em dias, mês e anos e calcula e armazena no atributo *idade* a idade atual da pessoa
 - *informaIdade()*, que retorna o valor da idade
 - *informaNome()*, que retorna o nome da pessoa
 - *ajustaDataDeNascimento()*, que recebe dia, mês e ano de nascimento como parâmetros e preenche nos atributos correspondentes do objeto.
- Criar dois objetos da classe *Pessoa*, um representando Albert Einstein (nascido em 14/3/1879) e o outro representando Isaac Newton (nascido em 4/1/1643)
- Fazer uma classe principal que instancie os objetos, inicialize e mostre quais seriam as idades de Einstein e Newton caso estivessem vivos.

Aula passada



Aula passada

```
public Pessoa(String nome, int dia, int mes, int ano) {
    this.nome = nome;
    this.ajustaDataNascimento(dia, mes, ano);
}

public static <T extends Pessoa> void calculaIdade(int diaAtual, int mesAtual, int anoAtual,
    List<T> pessoas, Consumer<T> c) {
    for (T p : pessoas) {
        int anos = anoAtual - p.anoNasc;
        if (mesAtual < p.mesNasc || (mesAtual == p.mesNasc && diaAtual < p.diaNasc)) {
            anos--;
        }
        p.idade = anos;
        c.accept(p);
    }
}
```

Aula passada

```
public class aula03 {  
    public static void main(String[] args) {  
        Pessoa pessoal = new Pessoa(nome: "Albert Ainsten", dia: 14, mes: 3, ano: 1879);  
        Pessoa pessoa2 = new Pessoa(nome: "Isaac Newton", dia: 4, mes: 1, ano: 1643);  
  
        List<Pessoa> pessoas = List.of(e1: pessoal, e2: pessoa2);  
  
        pessoal.calculaIdade(diaAtual: 11, mesAtual: 8, anoAtual: 2023, pessoas, (Pessoa p) -> {  
            System.out.println("Pessoa: " + p.nome + " | Idade calculada: " + p.idade);  
        });  
    }  
}
```

Classes Utilitárias

Parse ou Parsing de dados

Wrapper em java

Autoboxing

Unboxing



[Conference](#) [Proceedings](#) [Upcoming Events](#) [Authors](#) [Affiliations](#) [Award Winners](#)

[Jl](#) > [Proceedings](#) > [SOAP 2019](#) > [Modernizing parsing tools: parsing and analysis with object-oriented programming](#)

RESEARCH-ARTICLE



[Twitter](#) [LinkedIn](#) [Reddit](#) [Facebook](#) [Email](#)



Modernizing parsing tools: parsing and analysis with object-oriented programming

Authors:  [Steven O'Hara](#),  [Rocky Slavin](#) [Authors Info & Claims](#)

SOAP 2019: Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis • June 2019 • Pages 20–25 • <https://doi.org/10.1145/3315568.3329967>

Published: 22 June 2019 [Publication History](#) 

 0  131

   [Get Access](#)

Classes Utilitárias

Wrapper

Classes que encapsulam os tipos primitivos e transformam em objetos.

```
byte byteP = 1;  
short shortP = 1;  
int intP = 1;  
long longP = 10L;  
float floatP = 10F;  
double doubleP = 10D;  
char charP = 'W';  
boolean booleanP = false;
```


Classes Utilitárias

Wrapper

Classes que encapsulam os tipos primitivos e transformam em objetos.

```
byte byteP = 1;  
short shortP = 1;  
int intP = 1;  
long longP = 10L;  
float floatP = 10F;  
double doubleP = 10D;  
char charP = 'W';  
boolean booleanP = false;
```

```
Byte byteW = 1;  
Short shortW = 1;  
Integer intW = 1;  
Long longW = 10L;  
Float floatW = 10F;  
Double doubleW = 10D;  
Character charW = 'W';  
Boolean booleanW = false;
```

Classes Utilitárias

Wrapper

Com a letra maiúscula ele se torna objeto. Cuidados:

Polimorfismo e não escala de valor primitivo (tamanho)

Byte x = 127 ~ Byte x = (byte) 128

Float x = 10F # Float x = 10

Classes Utilitárias

Wrapper

Qual a importância: Estruturas de Dados não trabalha com tipos primitivos

ArrayList, Collections

```
List<int> lista; List<Integer> lista;
```

Dentro das coleções guardamos referência e não valores!

Classes Utilitárias

Wrapper

Como são valores de referência podemos atribuir métodos ou converter primitivos.

```
Integer x = 1; //boxing
```

```
int i = x;      // é igual um parse comum intValue(x)
```

```
Integer y = Integer.parseInt("1");
```

```
Integer y = new Integer("2");
```

Classes Utilitárias

Objeto nos permite construir tipos de Parser próprios

Expressões Regulares: São controles de expressões em arquivo ou lista de valores. Podemos filtrar determinada regra de valor

Ex. Encontrar cpf em uma lista numérica, ou arquivo externo

Classes Utilitárias

Expressões Regulares (REGEX)

Baseado em **Busca** de padrões: data scraping, time-series

Validações

Substituições/Edições

Classes Utilitárias

Streams: forma de processamento de dados, que permite criar processo funcional em coleções.

Forma declarativa

Parecido com busca em BD

Classes Utilitárias

Vamos imaginar um cenário onde quero ordenar ou fazer busca em livros por valores ou atributos.

Ex: Ordenar por nome, trazer os livros com preço menor que R\$20

```
private static List<Livro> livros = new ArrayList<>() {  
    c: List.of(  
        new Livro( titulo: "O Iluminado",  preco: 19.99),  
        new Livro( titulo: "Hellraiser",  preco: 29.99),  
        new Livro( titulo: "H. P. Lovecraft",  preco: 57.00),  
        new Livro( titulo: "O Rei de Amarelo",  preco: 15.00),  
        new Livro( titulo: "Ao Cair da Noite",  preco: 34.50),  
        new Livro( titulo: "Senhor das Moscas",  preco: 14.50),  
        new Livro( titulo: "Contos Allan Poe",  preco: 10.99)  
    )  
};
```


Classes Utilitárias

Ordenar por nome, trazer os livros com preço menor que R\$20

```
public static void main(String[] args) {  
    livros.sort( c: Comparator.comparing(Livro::getTitulo));  
    List<String> nomes = new ArrayList<>();  
    System.out.println( x: livros);  
    for(Livro lv : livros){  
        if(lv.getPreco() <= 20){  
            nomes.add( e: lv.getTitulo());  
        }  
    }  
    System.out.println( x: nomes);  
}
```

Classes Utilitárias

Podemos iterar e filtrar utilizando **stream**. Coleções em java possuem esse método

Operações stream podem ser intermediárias ou finais:

- A primeira retorna um proprio stream; permite encadear operações;
- A segunda retorna um objeto processado.

Classes Utilitárias

Stream

```
List<String> nm2 = livros.stream()
    .sorted( comparator: Comparator.comparing(Livro::getTitulo))
    .filter(ln -> ln.getPreco() <= 20)
    .map(ln -> ln.getTitulo())
    .collect( collector: Collectors.toList());
System.out.println("Stream: " + nm2);
```

Classes Utilitárias

Stream

Funções lambda podem ser utilizadas para condicional em um stream

```
livros.stream( ).forEach(lv -> System.out.println(lv));
```

Dessa forma iterando coleções poderia criar regras funcionais com lambda para diversos filtros e iterações.

Classes Utilitárias

Stream

Temos uma lista com listas aninhadas de nomes:

```
List<List<String>> bd = new ArrayList<>();
List<String> diretores = List.of(e1: "Ana", e2: "Pedro", e3: "Harrison");
List<String> funcionarios = List.of(e1: "Paula", e2: "Joao", e3: "Frank", e4: "Olivia");
List<String> terceirizados = List.of(e1: "Henrique", e2: "Cintia", e3: "Andre");
bd.add(e: diretores); bd.add(e: funcionarios); bd.add(e: terceirizados);
for(List<String> pessoas: bd){
    for(String nome : pessoas){
        System.out.println(n: nome);
    }
}
```

Classes Utilitárias

Stream

Nesse caso, temos uma coleção de listas, logo precisamos achatar os dados (flattening).

A classe Stream utiliza o método **Flatmap**, que identifica e processa por atributos dentro de listas aninhadas.

```
bd.stream().flatMap(Collection::stream).forEach(System.out::println);
```

A diferença do map e flatmap é que o segundo retira os dados aninhados em um segundo nível tratando todos os valores no processamento da coleção

Classes Utilitárias

Stream: **Finding** e **Matching** (Busque e Combine)

Aceita condições específicas. Ex: Existe algum livro com preço acima de 50? Ou se todos os livros tem valor cadastrado?

```
System.out.println(livros.stream().anyMatch(l -> l.getPreco() > 50));
```

```
System.out.println(livros.stream().allMatch(l -> l.getPreco() > 0));
```

```
System.out.println(livros.stream().noneMatch(l -> l.getPreco() < 0));
```

Processa um predicado, e retorna um booleano.

Classes Utilitárias

Stream: **Finding** e **Matching** (Busque e Combine)

Aceita condições específicas. Ex: Existe algum livro com preço acima de 50?

```
livros.stream()
```

```
.filter(l -> l.getPreco( ) > 10)
```

```
.sorted(Comparator.comparing(Livro::getPreco))
```

```
.findFirst()      //ou findAny()
```

```
.ifPresent(System.out::println);
```


Classes Utilitárias

Stream: **Reduce**, permite **executar operações terminais entre parâmetros**

```
List<Integer> num = List.of( 22, 41, 36, 71, 80, 34);
```

```
num.stream().reduce((x,y) -> x+y).ifPresent(System.out::println);
```

Ou `system.out.println(num.stream().reduce(0, (x,y) -> x+y));`

O interessante é que a operação lógica (f)Lambda permite simplificar o processamento de um cálculos ou operações lógicas em coleções.

```
num.stream().reduce((x,y) -> x > y ? x : y ).ifPresent(System.out::println);
```

Classes Utilitárias

Ex. Some todos os preços dos livros maiores que R\$20

```
livros.stream()  
  .map(Livro::getPreco)  
  .filter(preco -> preco > 20)  
  .reduce((x,y) -> x+y)  
  .ifPresent(System.out::println);
```

Classes Utilitárias

Gerando Streams: Podemos utilizar métodos de tipos primitivos como o `IntStream` para gerar cadeias de dados que podem ser iterados por condições e predicados:

Ex: Gerar uma lista numérica de 1-100 e exibir os números pares

```
IntStream.range(0, 101)  
    .filter(n -> n % 2 == 0)  
    .forEach(n->System.out.println(n));
```

Classes Utilitárias

Gerando Streams:

Podemos utilizar este tipo de gerador para listas de objetos ou valores complexos.

Isso nos permite testar processos lógicos sequenciais:

- Sensores
- Criptografia
- Grafo

Classes Utilitárias

Gerando Streams: o Comando `Stream.of(T...values)` aceita uma lista de valores de qualquer tipo.

Ex

```
Stream.of("valor1", "valor2", "valor3", "valorN")  
    .map(String::hashCode)  
    .forEach(s -> System.out.print(s + "! "));
```

Classes Utilitárias

Gerando Streams: Com vetores.

Podemos utilizar funções prontas do Stream com vetores de qualquer tipo

Ex

```
Int num[ ] = {1, 2, 4, 3, 6, 4, 7, 9, 1}
```

```
Arrays.stream(num).sum().ifPresent(System.out::println);
```

```
Arrays.stream(num).min().ifPresent(System.out::println);
```

```
Arrays.stream(num).max().ifPresent(System.out::println);
```

```
Arrays.stream(num).average().ifPresent(System.out::println);
```

Classes Utilitárias

Streams iterativos

Temos dois métodos iterativos de Streams: **iterate** e **generate**

O `iterate` serve para gerar valores, baseado em uma operação descritiva

```
Stream.iterate(1, n -> n+2).forEach(System.out::println);
```

Útil para simulações de entradas on-the-fly (ex. time-series).

Podemos limitar com o operador **limit()**

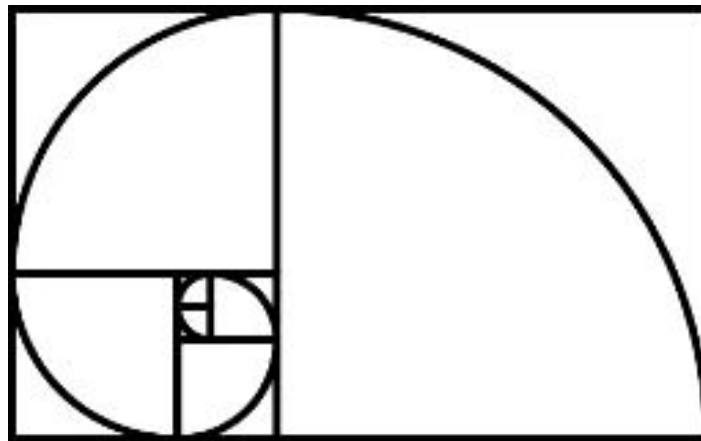
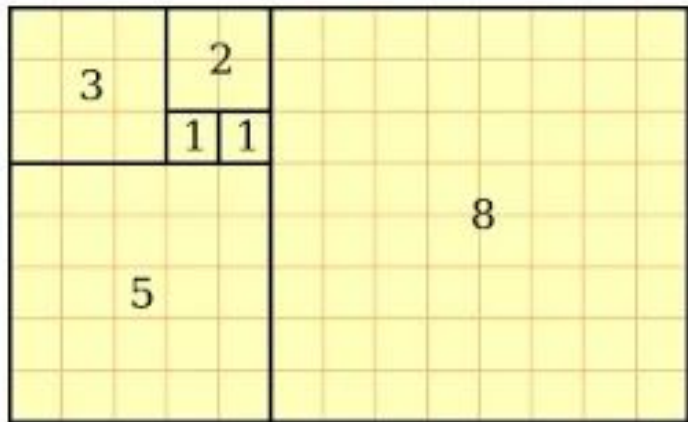
```
Stream.iterate(1, n -> n+2).limit(20).forEach(System.out::println);
```

Classes Utilitárias

Streams iterativos

Sequência de **Fibonacci** é a sequência numérica proposta pelo matemático Leonardo Pisa, mais conhecido como Fibonacci: $F_n = F_{n-1} + F_{n-2}$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...



Classes Utilitárias

Streams iterativos

Sequência de **Fibonacci** com stream

(0,1) (1,1) (1,2) (2,3) (3,5) (5,8)

```
Stream.iterate(new int[]{0, 1}, n -> new int[]{n[1], n[0]+n[1]})  
  
    .limit(20)  
  
    .forEach(a -> System.out.println(Arrays.toString(a)));
```

Classes Utilitárias

Streams iterativos

O comando **generate** não aceita comando iterativo e sim um supplier (ex. Número aleatório)

```
Random gerador = new Random();
```

```
Stream.generate(()->gerador.nextInt(1, 1000))
```

```
.limit(10)
```

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

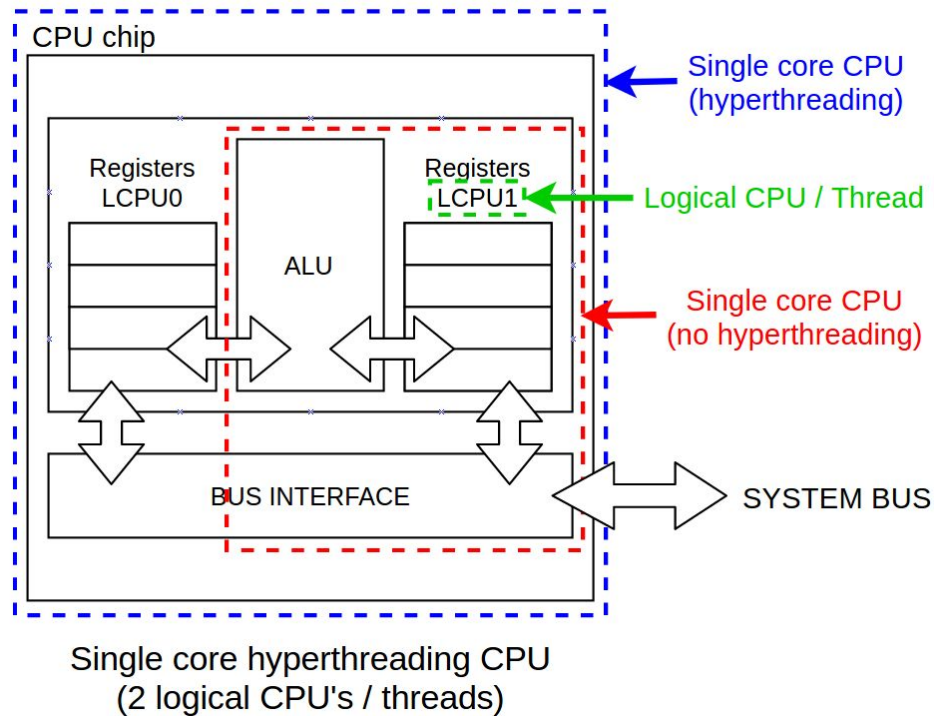
Classes Utilitárias

Streams paralelos

Você pode paralelizar os stream por thr

Processadores ~ Núcleos

```
System.out.println(Runtime  
.getRuntime().availableProcessors());
```



Classes Utilitárias

Streams paralelos

Exemplo, queremos somar um valor alto (Ex 10 milhões):

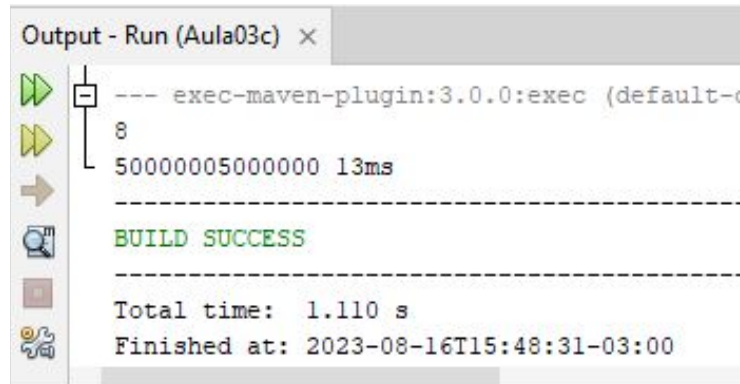
```
public class Aula03c {  
    static long num = 10_000_000;  
    public static void main(String[] args) {  
        long res = 0;  
        long init = System.currentTimeMillis();  
        for(long i = 1; i <= num; i++){  
            res = res + i;  
        }  
        long end = System.currentTimeMillis();  
        System.out.println(res + " " + (end-init)+"ms");  
    }  
}
```

Classes Utilitárias

Streams paralelos

Exemplo, queremos somar um valor alto:

```
public class Aula03c {  
    static long num = 10_000_000;  
    public static void main(String[] args) {  
        long res = 0;  
        long init = System.currentTimeMillis();  
        for(long i = 1; i <= num; i++){  
            res = res + i;  
        }  
        long end = System.currentTimeMillis();  
        System.out.println(res + " " + (end-init)+"ms");  
    }  
}
```



Classes Utilitárias

Streams paralelos

Podemos fazer o mesmo processo com Stream:

```
Stream.iterate(1L, i -> i+1).limit(num).reduce(0L, Long::sum);
```

A performance cai consideravelmente

No entanto, podemos utilizar o comando `.parallel()`

```
Stream.iterate(1L, i -> i+1).limit(num).parallel().reduce(0L, Long::sum);
```

O escalonador cuida da distribuição



Classes Utilitárias

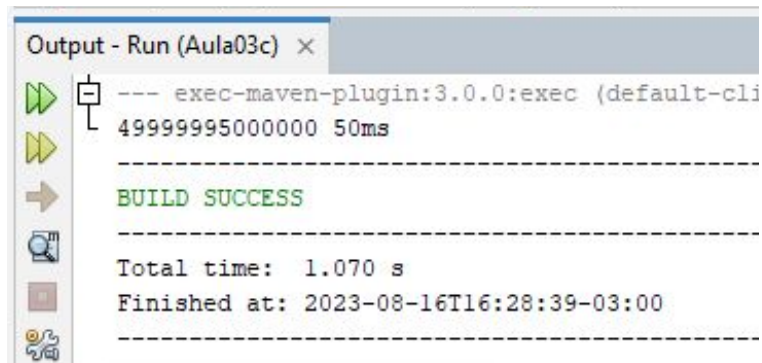
Streams paralelos

Não existem muito ganho paralelizando sequências lineares

Distribuição x gargalo de memória

Existem soluções paliativas mas ainda não é paralelo

```
res = LongStream.range(1L, num)
    .parallel()
    .reduce(0L, Long::sum);
```



Classes Utilitárias

Streams paralelos

Qualquer função paralela deve ser analisada com Benchmark

Ferramentas corretas: OpenMP, CUDA, Compilado Interpretado

GPUs, multi GPU e multi Nós. OpenAcc, NCCL

Multiprocessamento: Múltiplos processadores executando simultaneamente.

Multitarefa: Múltiplos processos rodando simultaneamente

Multithreading: Múltiplas partes do mesmo programa rodando simultaneamente.

Classes Utilitárias

Streams paralelos

Tracing of Multi-Threaded Java Applications in Score-P Using Bytecode Instrumentation

Publisher: VDE

[Cite This](#)

[PDF](#)

Jan Frenzel ; Kim Feldhoff ; Rene Jaekel ; Ralph Mueller-Pfefferkorn [All Authors](#)

1

Cites in
Paper

70

Full
Text Views



Abstract

[Authors](#)

[Citations](#)

[Metrics](#)

Abstract:

Thread-parallel Java applications received a substantial boost in the research field of High Performance Computing over the past years. In order to efficiently execute multithreaded Java applications, an analysis of their performance is indispensable. This requires a scalable runtime performance measurement infrastructure due to the high number of used threads. The established, open-source tracing framework Score-P provides such an infrastructure. However, it only provides basic support for multi-threaded Java applications so far. In this paper, we present a more sophisticated instrumentation approach based on Java bytecode transformations and implement the approach in Score-P. The approach allows to trace an application without requiring users to modify their source code. In contrast to instrumentation approaches based on the Java Virtual Machine Tool Interface JVMTI, it does not need filter checking and thus, has a comparable low run-time overhead. However, if needed, function and thread related events of the application can be filtered at runtime. Additionally, class files can be selected such that only those classes of an application are recorded, which users are interested in. We apply the proposed instrumentation approach to the LU kernel of the established Java benchmark suite SPECjvm2008 at a modern HPC shared-memory machine

Concluindo...

Classes utilitárias

Wrappers

Stream

Exercícios

Exercício

1 - Utilizando Streams, crie um programa em Java que implemente a função **uniao**. A função recebe como parâmetros dois arrays de inteiros e retorna um novo array contendo a união de v1 e v2.

Por exemplo, se $v1 = \{11, 13, 45, 7\}$ e $v2 = \{24, 4, 16, 81, 10, 12\}$, v3 irá conter $\{11, 13, 45, 7, 24, 4, 16, 81, 10, 12\}$.

2 - Usando Streams, faça um sistema que encontre:

a) totalPares: a função recebe como parâmetro um array de inteiros e retorna um número inteiro indicando o total de números pares existentes no array.

b) maiorValor: a função recebe como parâmetro um array de inteiros e retorna o maior número existente no array.

Referências

1. DEITEL. JAVA Como Programar. 8a. ed. São Paulo: Pearson Prentice Hall, 2010.
2. JANDL JUNIOR, Peter. Java Guia do Programador. São Paulo: Novatec, 2014.
3. FREEMAN, Eric. Use a cabeça: padrões e projetos. 2. ed. rev. Rio de Janeiro: Alta Books, 2009