



# Linguagem de Programação III (LP35A)

Profª Eliana Santos

# Apresentação

Prof<sup>a</sup> Eliana Santos



# Olá, muito prazer!



**Analista de Sistemas** – Analise de Sistemas (USF/SP)

**MBA em Gerência de Projetos** – Especialista em Gerência de Projetos (UNIVALI)

**Mestre em Educação** - UNIVALI. “**O ensino de gerenciamento de projetos auxiliado por um jogo de computador**”

**Doutorado em Engenharia da Informação – UFABC (terminando!!!): BCI - Brain Computer Interface (Imagética Motora)**

**Certificação PMP** – Project Management Professional pelo PMI (Project Management Institute) desde 2010

**Analista de Sistemas/Programadora/Gerente de Projetos de TI - Análise e Desenvolvimento** de Sistemas, Migração de Dados, Análise de Requisitos, Análise de Processos, Desenvolvimento de Aplicações, Implantação de Sistemas, **Treinamento em Sistemas** e Gestão de Projetos, **Consultoria em TI e Gestão de Projetos**.

**Professora** - Tecnologia de Informação e Gerenciamento de Projetos

# Planejamento didático

## Metodologia

- Aula expositiva, exercícios práticos (Laboratório Informática)

## Objetivos

- Aplicar conceitos avançados de linguagens orientada a objetos na construção de aplicações. Entender modelos de concorrência e paralelismo. Analisar as vantagens e desvantagens de cada modelo. Desenvolver aplicações que manipulem elementos da linguagem. Desenvolver aplicações robustas, utilizando técnicas para controle de estado e fluxo de informações.

# Planejamento didático

## Ementa

- Este componente curricular aprofunda os conhecimentos do aluno em uma linguagem de programação orientada a objetos.
- São abordados tópicos relacionados a recursos avançados da linguagem e da orientação a objetos, como utilização do sistema de tipos para a criação de estruturas reutilizáveis, técnicas para construção de aplicações robustas, **modelos de concorrência e paralelismo e reflexão.**

# Planejamento didático

## Conteúdo Programático

- Tópicos avançados em concorrência e paralelismo
- Gerenciamento de estado e controle de fluxo de informação
- Meta-programação
- Recursos e técnicas de programação funcional aplicados à linguagem
- Técnicas de Refatoração e de Código Limpo

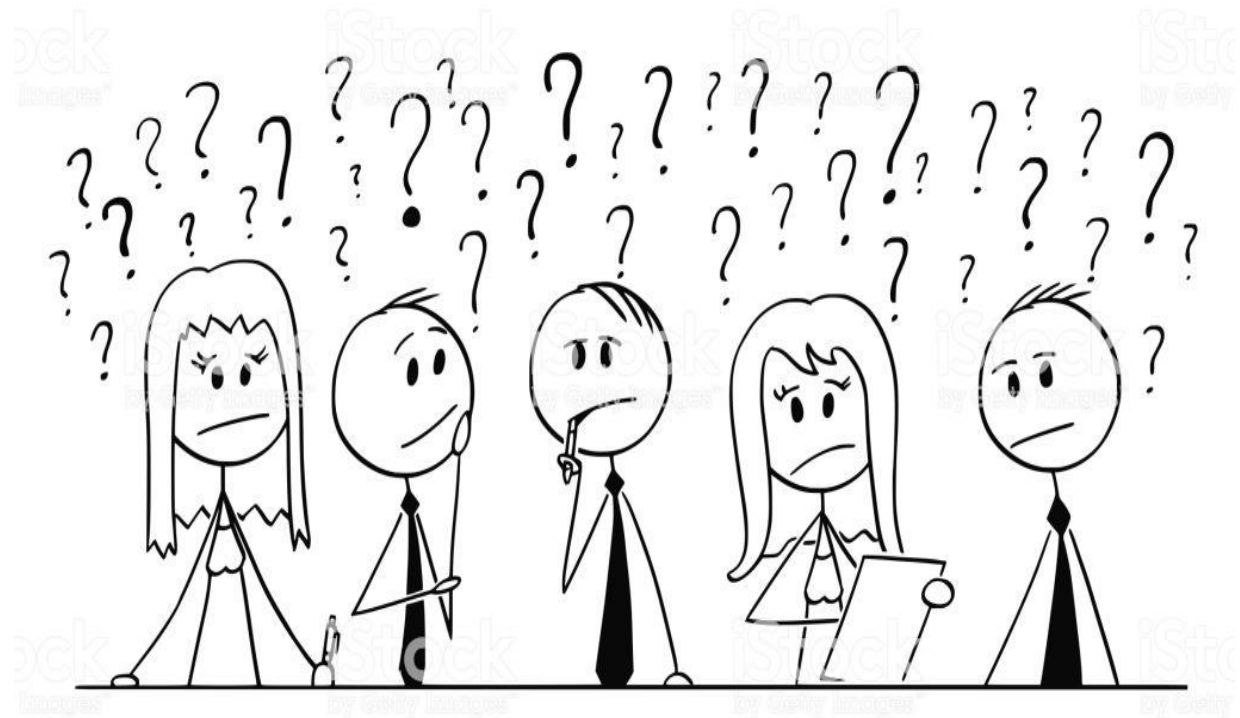


# Avaliação

- Assiduidade
- Participação nas atividades
- Prova, (teórica, prática)
- Trabalho (individual ou em grupo)
- Seminário

# Apresentações e expectativas

- Quem é você?
- De onde você vem?
- O que você faz?
- Qual a sua expectativa para esta disciplina?





# Linguagem de Programação III (LP35A)

Concorrência e Paralelismo

# Concorrência e Paralelismo

- O processamento paralelo, ou concorrente, tem como base um hardware multicore, onde dispõe-se de vários núcleos de processamento.
- **Estas arquiteturas, no início do Java, não eram tão comuns.**
- Atualmente já se encontram amplamente difundidas, tanto no contexto comercial como doméstico.
- Para que não haja desperdício desses recursos de hardware e possamos extrair mais desempenho do software desenvolvido, **é recomendado que alguma técnica de paralelismo seja utilizada.**



# Implementação de Paralelismo e Concorrência

Multicore



# Concorrência e Paralelismo

## Arquitetura Multicore

- Uma arquitetura multicore consiste em uma CPU que possui mais de um núcleo de processamento.
- Este tipo de hardware permite a execução de mais de uma tarefa simultaneamente, ao contrário das CPUs singlecore, que eram constituídas por apenas um núcleo, o que significa, na prática, que nada era executado efetivamente em paralelo.

# Concorrência e Paralelismo

## Arquitetura Multicore

- A partir do momento em que se tornou inviável desenvolver CPUs com frequências (GHz) mais altas, devido ao superaquecimento, partiu-se para outra abordagem: **criar CPUs multicore**, isto é, inserir vários núcleos no mesmo chip, com a premissa base de dividir para conquistar.

# Concorrência e Paralelismo

## Arquitetura Multicore

- Ao contrário do que muitos pensam, no entanto, os **processadores multicore não somam a capacidade de processamento**, e sim possibilitam a divisão das tarefas entre si.
- **Um processador de dois núcleos com clock de 2.0 GHz não equivale a um processador com um núcleo de 4.0 GHz.**
- A tecnologia multicore simplesmente permite a divisão de tarefas entre os núcleos de tal forma que efetivamente se tenha um processamento paralelo e, com isso, seja alcançado o tão almejado ganho de performance.



# Concorrência e Paralelismo

## Arquitetura Multicore

- **O ganho de processamento é possível apenas se o software implementar paralelismo.**
- Os Sistemas Operacionais possuem suporte a multicore, mas isso somente otimiza o desempenho do próprio SO, o que não é suficiente.
- **O ideal é cada software desenvolvido esteja apto a usufruir de todos os recursos de hardware disponíveis para ele.**

# Concorrência e Paralelismo

## Arquitetura Multicore

- Considerando o fato de que temos celulares com processadores de quatro ou oito núcleos, os softwares a eles disponibilizados devem estar preparados para lidar com esta arquitetura.
- Desde um simples projeto de robótica a um software massivamente paralelo para um supercomputador de milhões de núcleos, a opção por paralelizar ou não, pode significar a diferença entre passar dias processando uma determinada tarefa ou apenas alguns minutos.

# Implementação de Paralelismo e Concorrência

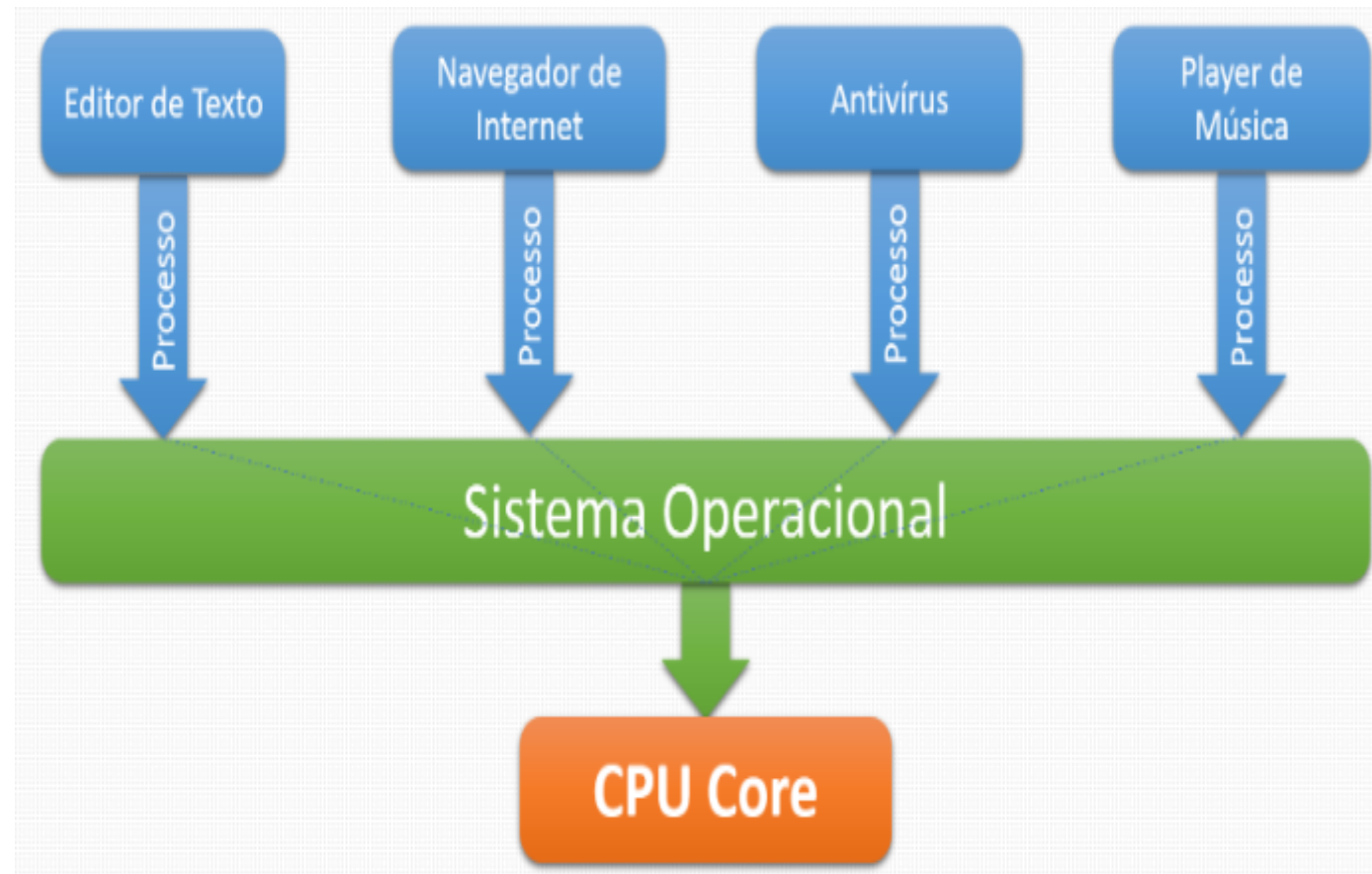
Multitask



# Concorrência e Paralelismo

## Multitasking

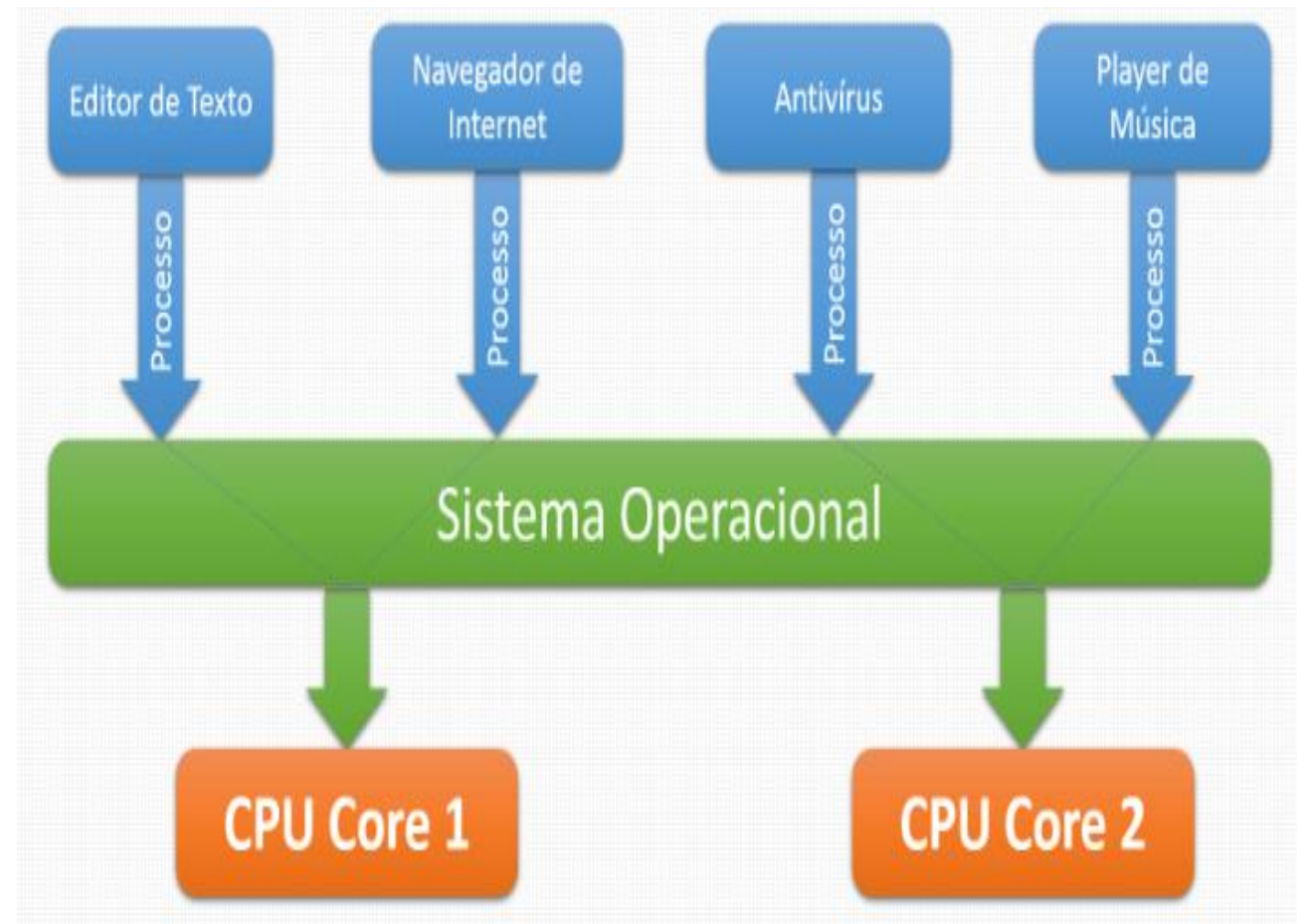
- É a capacidade que sistemas possuem de executar várias tarefas ou processos ao mesmo tempo, compartilhando recursos de processamento como a CPU.
- Esta habilidade permite ao sistema operacional intercalar rapidamente os processos ativos para ocuparem a CPU, dando a impressão de que estão sendo executados simultaneamente



# Concorrência e Paralelismo

## Multitasking

- Em arquiteturas multicore, efetivamente os processos podem ser executados simultaneamente, mas ainda depende do escalonamento no sistema operacional, pois geralmente temos mais processos ativos do que núcleos disponíveis para processar.

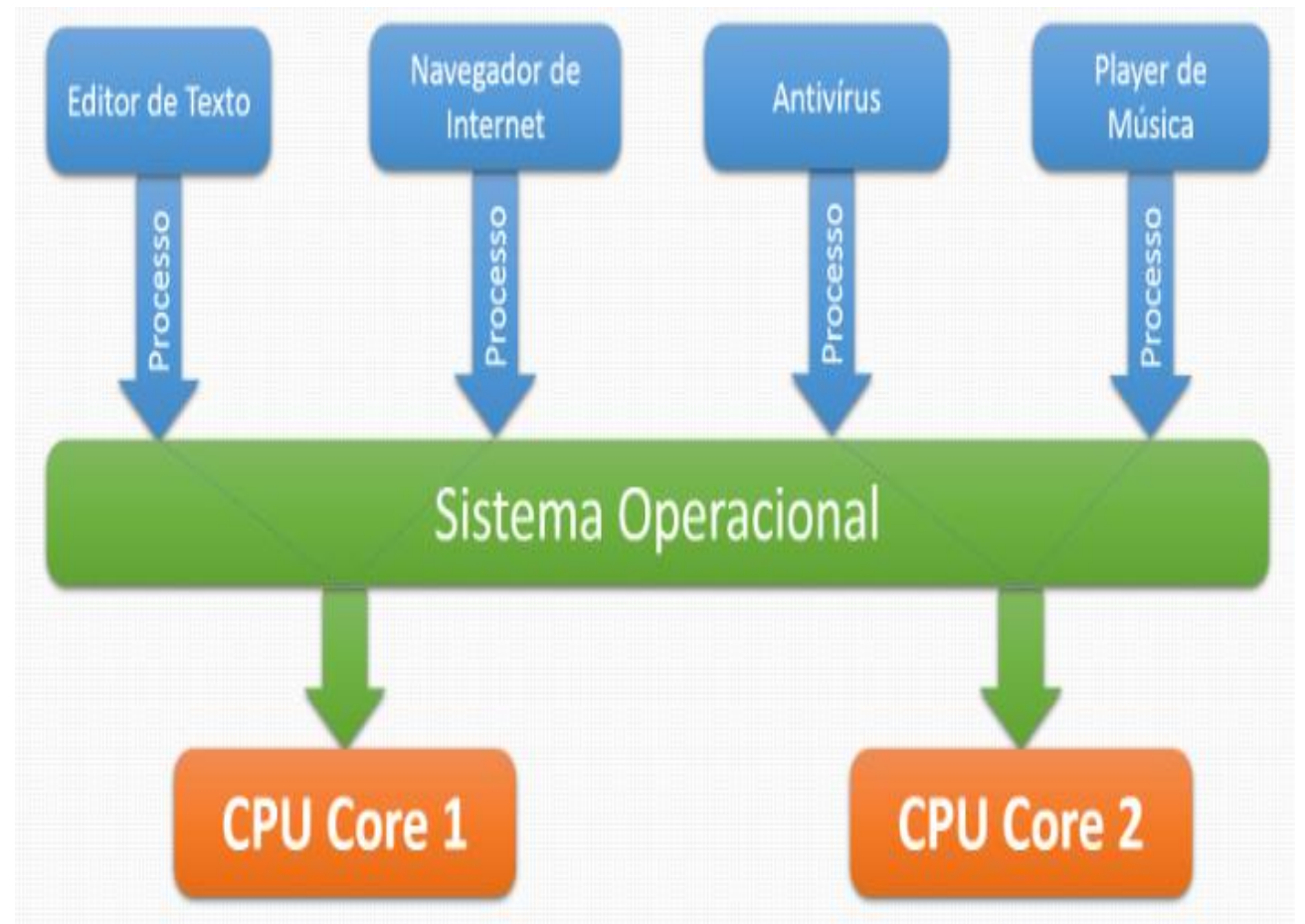




# Concorrência e Paralelismo

## Multitasking

- Desta forma, mais núcleos de processamento significam que mais tarefas simultâneas podem ser desempenhadas.
- Isto só é possível **se o software que está sendo executado sobre tal arquitetura implementa o processamento concorrente.**
- **De nada adianta um processador de oito núcleos se o software utiliza apenas um..**





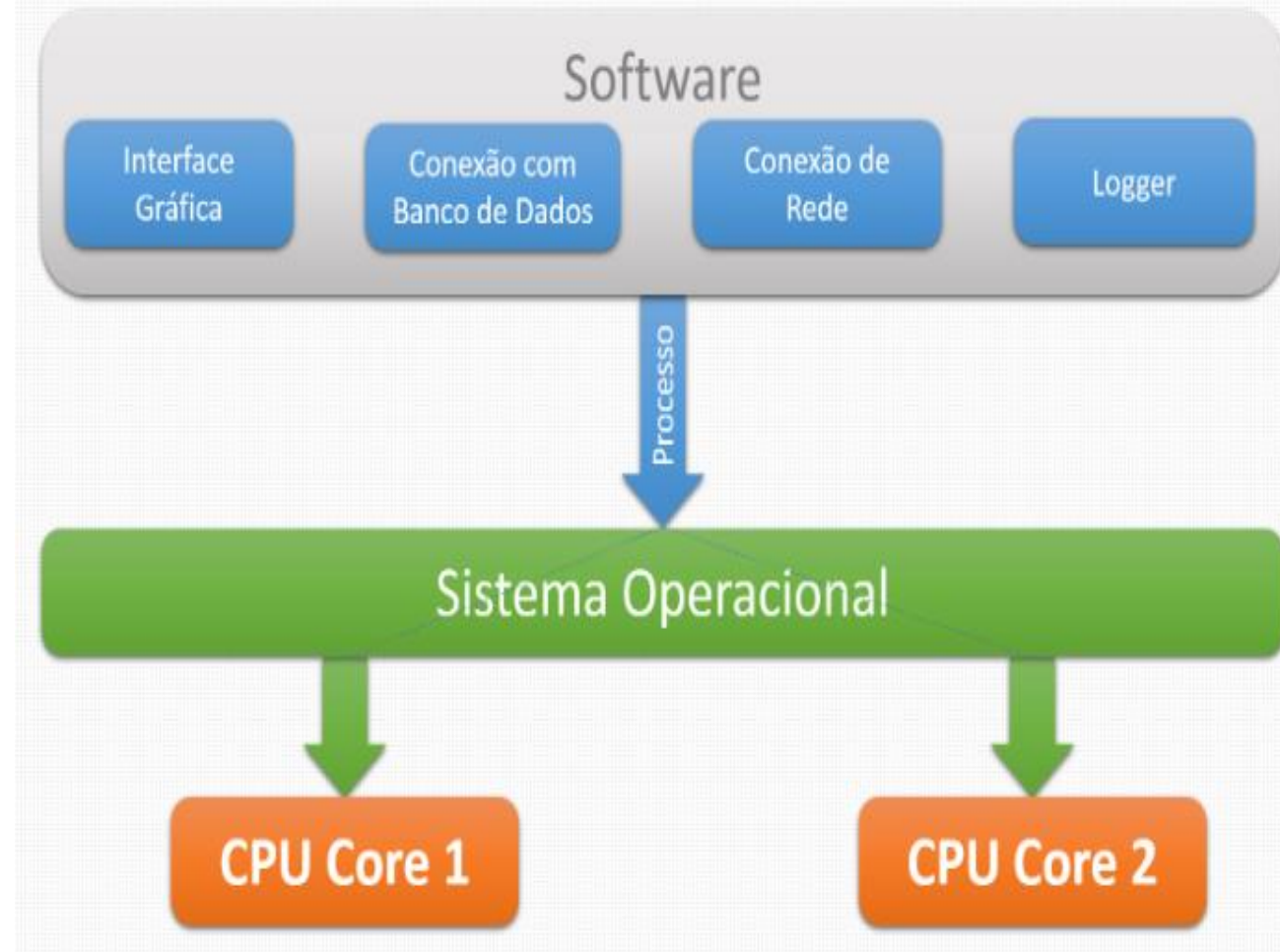
# Implementação de Paralelismo e Concorrência

Multithreading

# Concorrência e Paralelismo

## Multithreading

- Podemos compreender multithreading como uma evolução do multitasking, mas em nível de processo.
- Ele, basicamente, permite ao software subdividir suas tarefas em trechos de código independentes e capazes de executar em paralelo, chamados de threads.
- Com isto, cada uma destas tarefas pode ser executada em paralelo caso haja vários núcleos

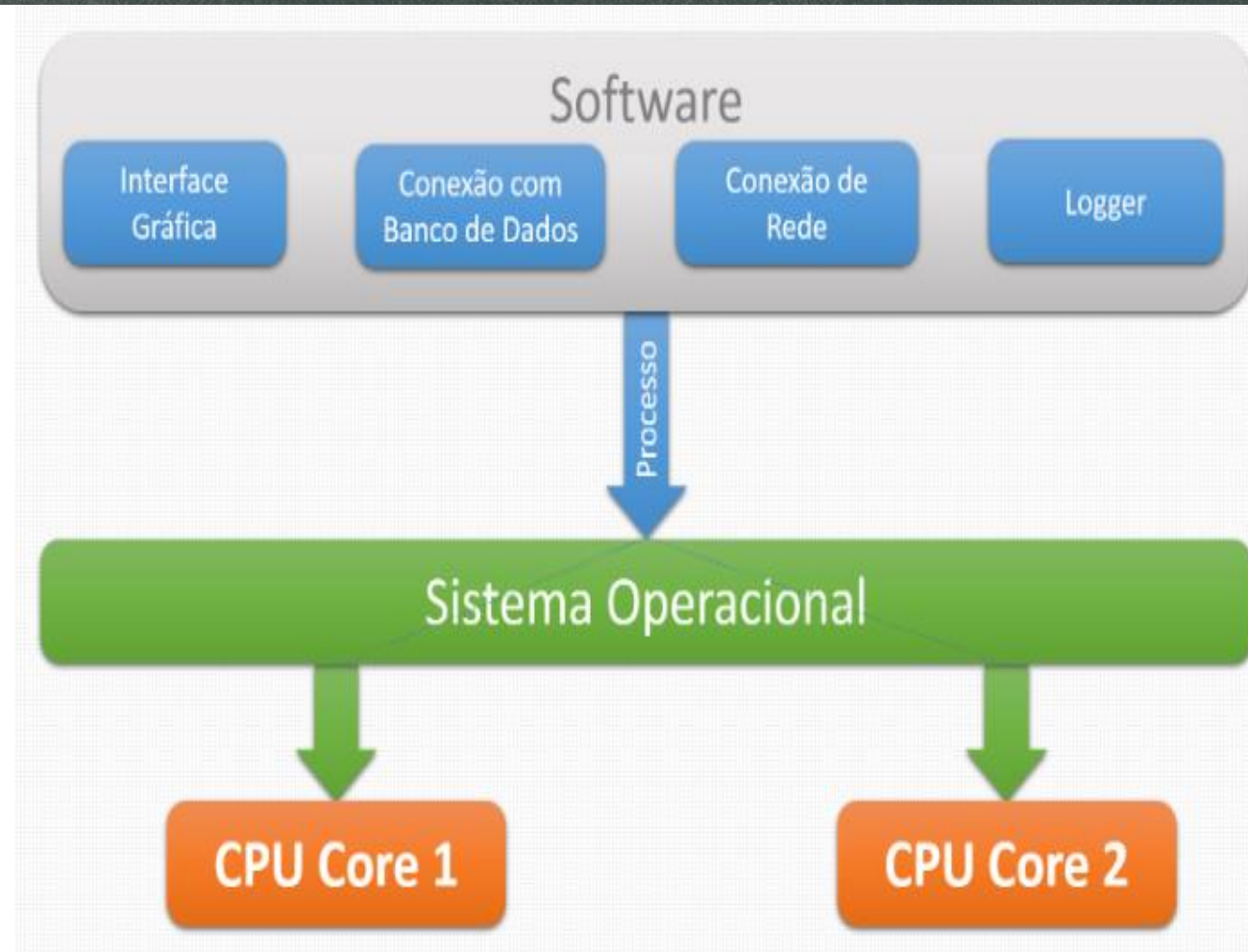




# Concorrência e Paralelismo

## Multithreading

- Diversos benefícios são adquiridos com este recurso, mas, sem dúvida, **o mais procurado é o ganho de performance**.
- Também é válido destacar o **uso mais eficiente da CPU**.
- Sabendo dessa importância, nosso próximo passo é entender **o que são as threads e como criá-las para subdividir as tarefas do software**.





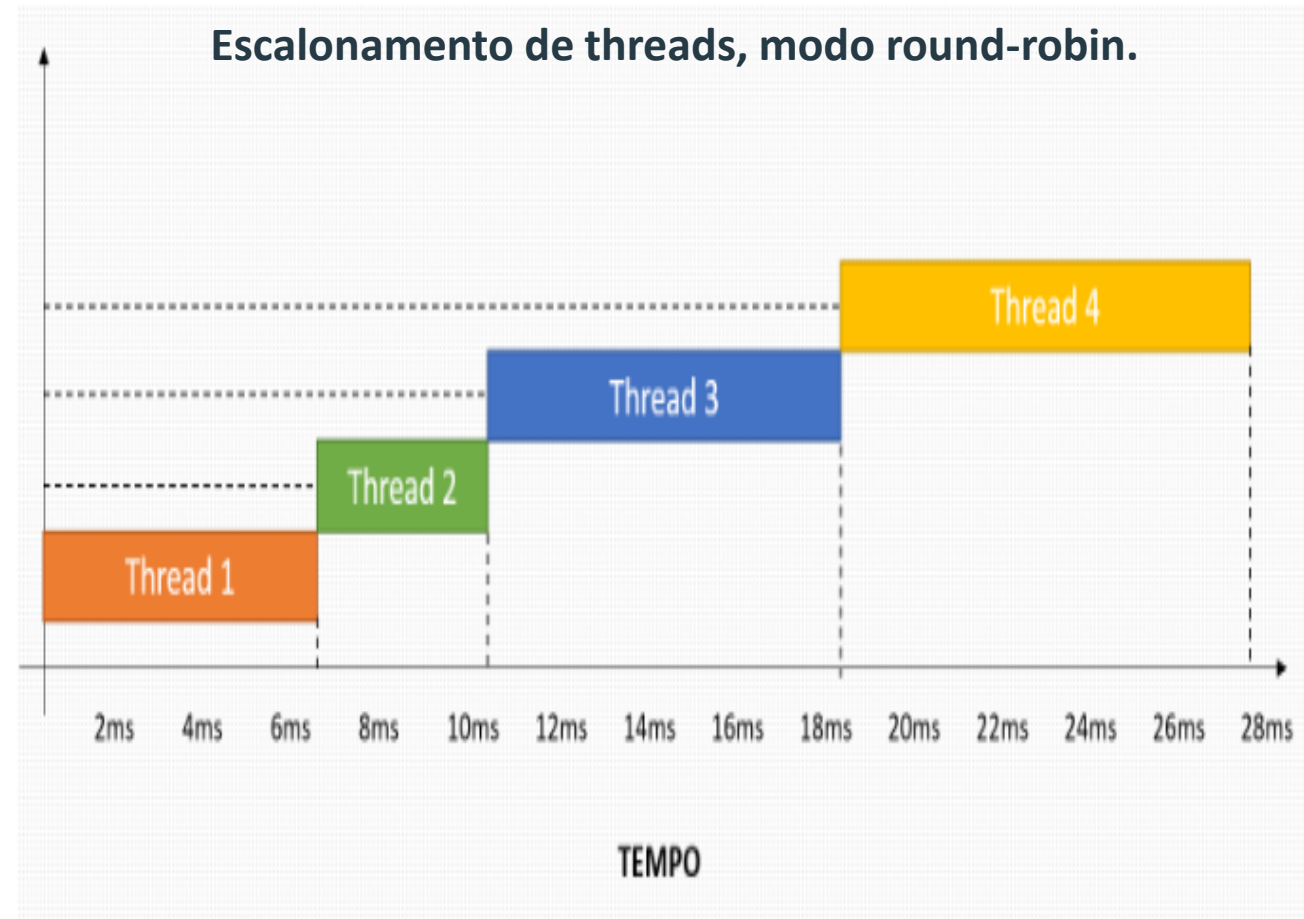
# Implementação de Paralelismo e Concorrência

Threads

# Concorrência e Paralelismo

## Threads

- **Thread** é um sistema que divide tarefas a fim de executá-las de forma simultânea ou sequencialmente, porém de maneira muito rápida.
- **Round-Robin** é um dos mais antigos e simples algoritmos de escalonamento.
- É largamente usado, e foi projetado especialmente para sistemas time-sharing

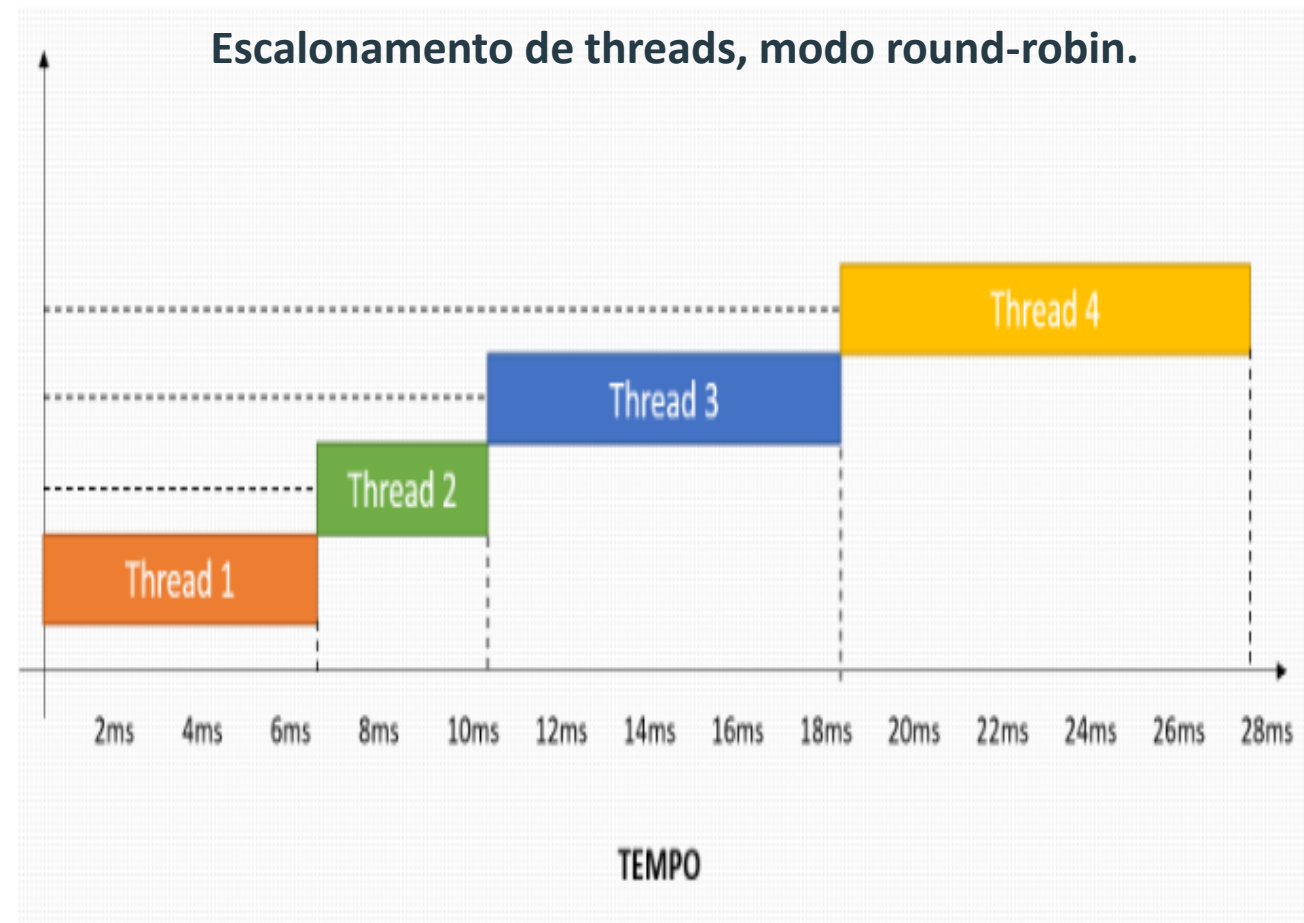




# Concorrência e Paralelismo

## Threads

- Na plataforma Java, as threads são o único mecanismo de concorrência suportado.
- Podemos entender esse recurso como trechos de código que operam independentemente da sequência de execução principal.

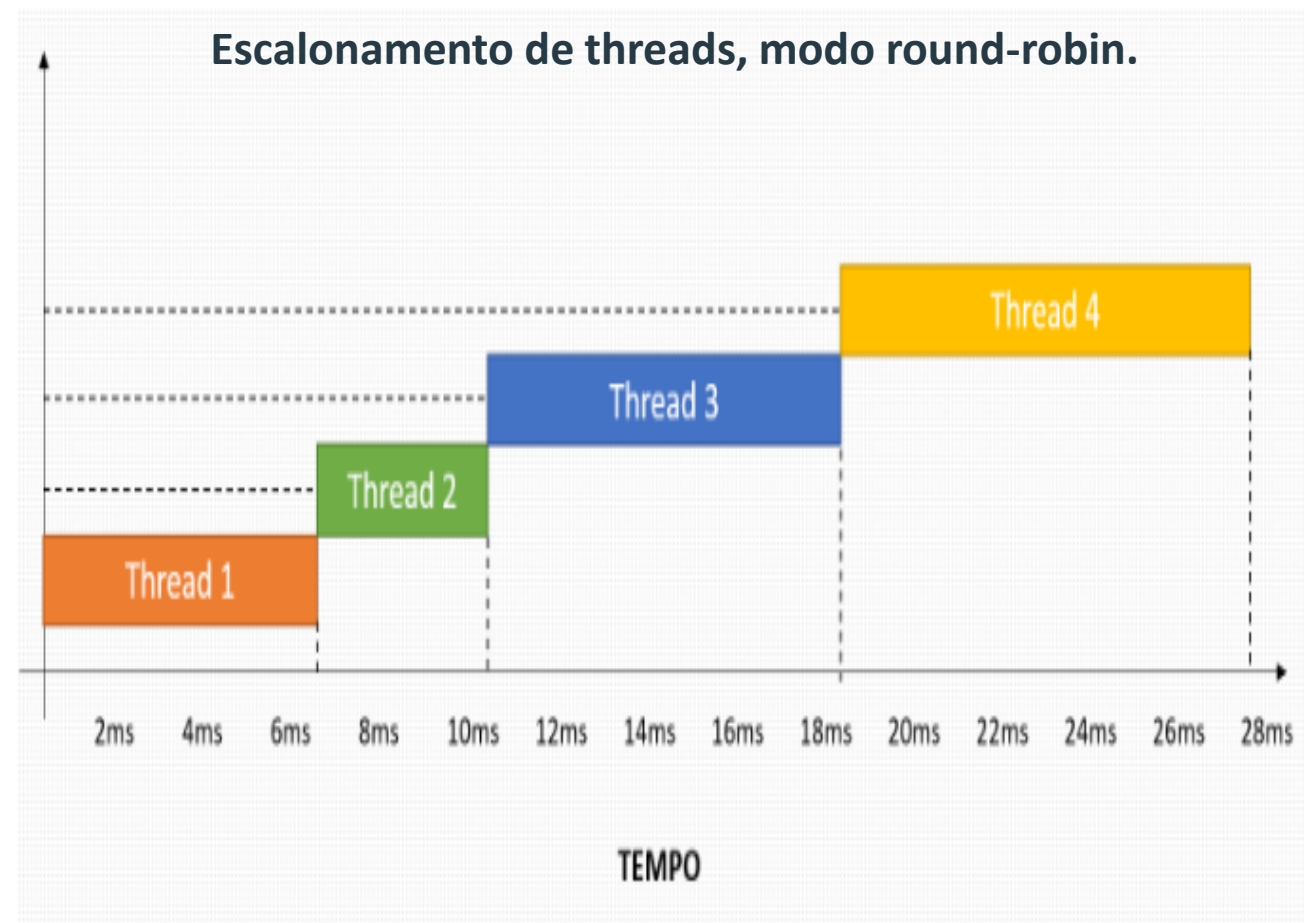




# Concorrência e Paralelismo

## Threads

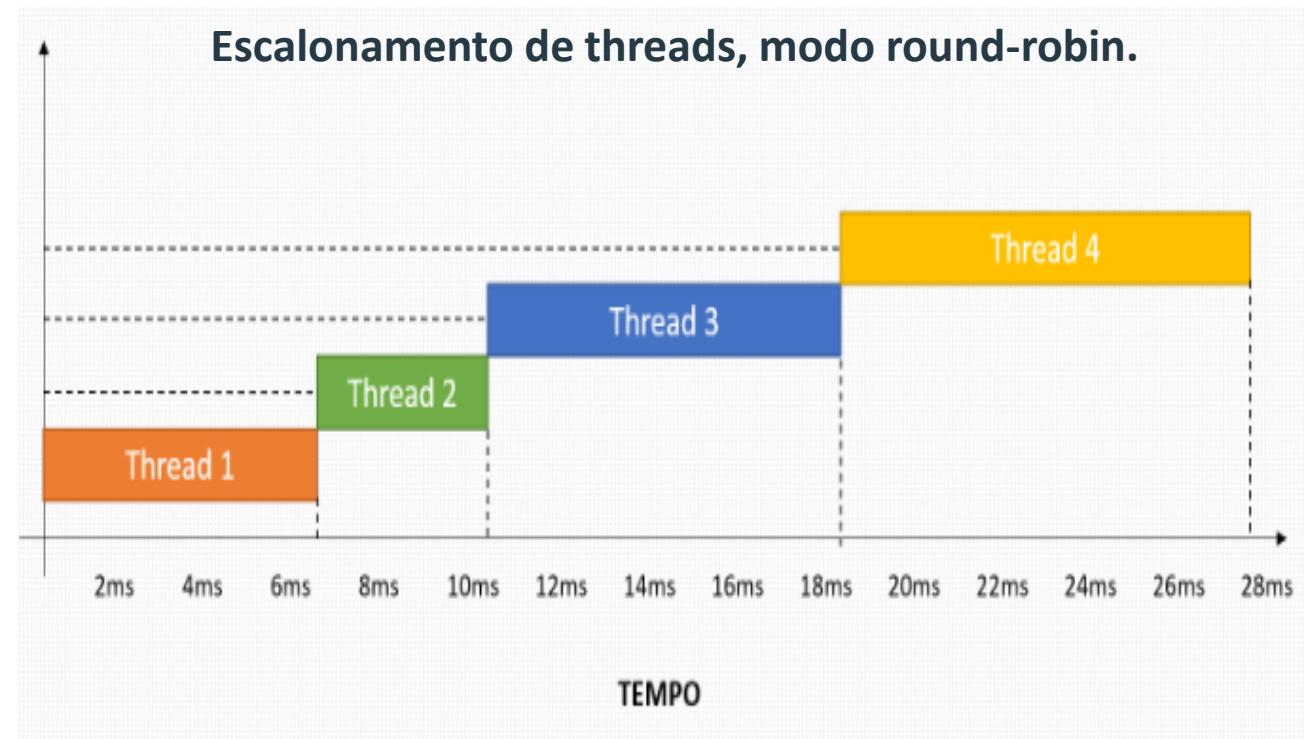
- Enquanto os processos de software não dividem um mesmo espaço de memória, as threads, sim, e isso lhes permite compartilhar dados e informações dentro do contexto do software.



# Concorrência e Paralelismo

## Threads

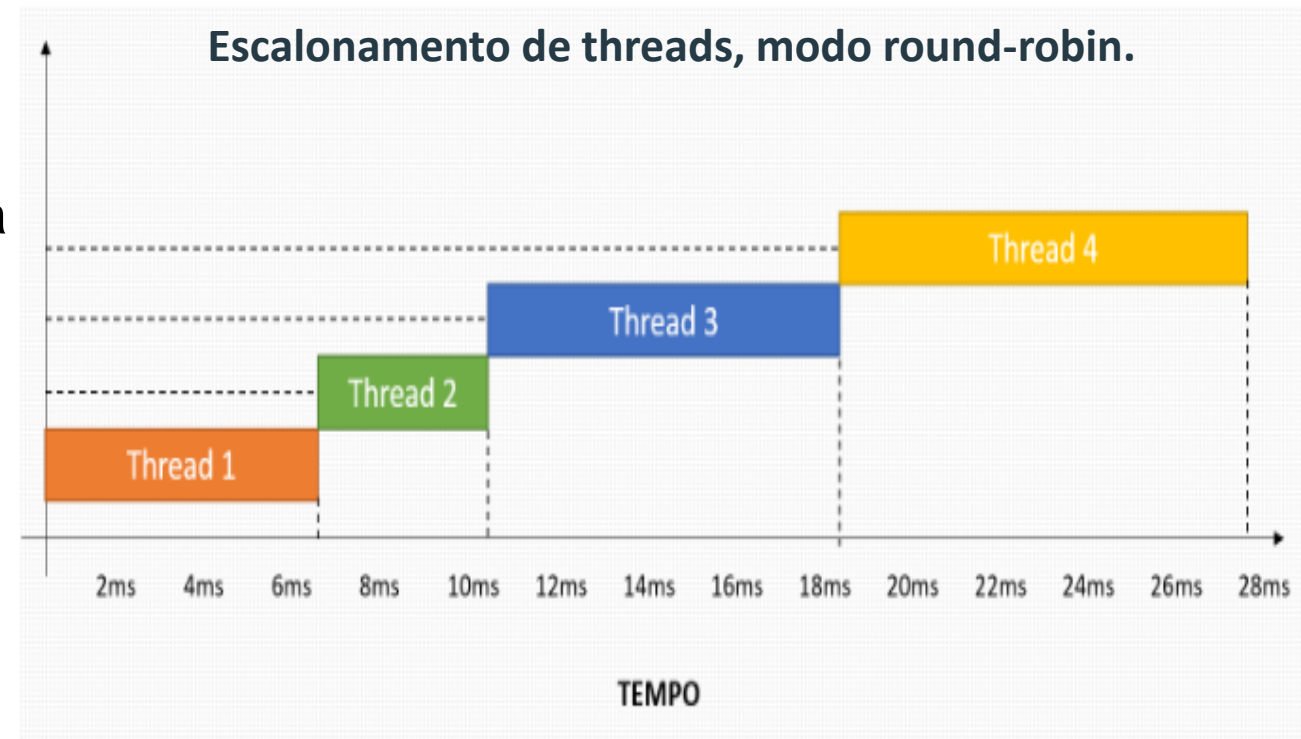
- Cada objeto de thread possui:
  - Identificador único e inalterável,
  - Nome,
  - Prioridade,
  - Estado,
  - Gerenciador de exceções,
  - Espaço para armazenamento local e
  - uma série de estruturas utilizadas pela JVM e pelo sistema operacional, **salvando seu contexto enquanto ela permanece pausada pelo escalonador.**



# Concorrência e Paralelismo

## Threads

- Também é possível observar que apenas uma thread é executada por vez.
- Isto normalmente acontece em casos onde só há um núcleo de processamento, o software implementa um sincronismo de threads que não as permite executar em paralelo ou quando o sistema não faz uso de threads.

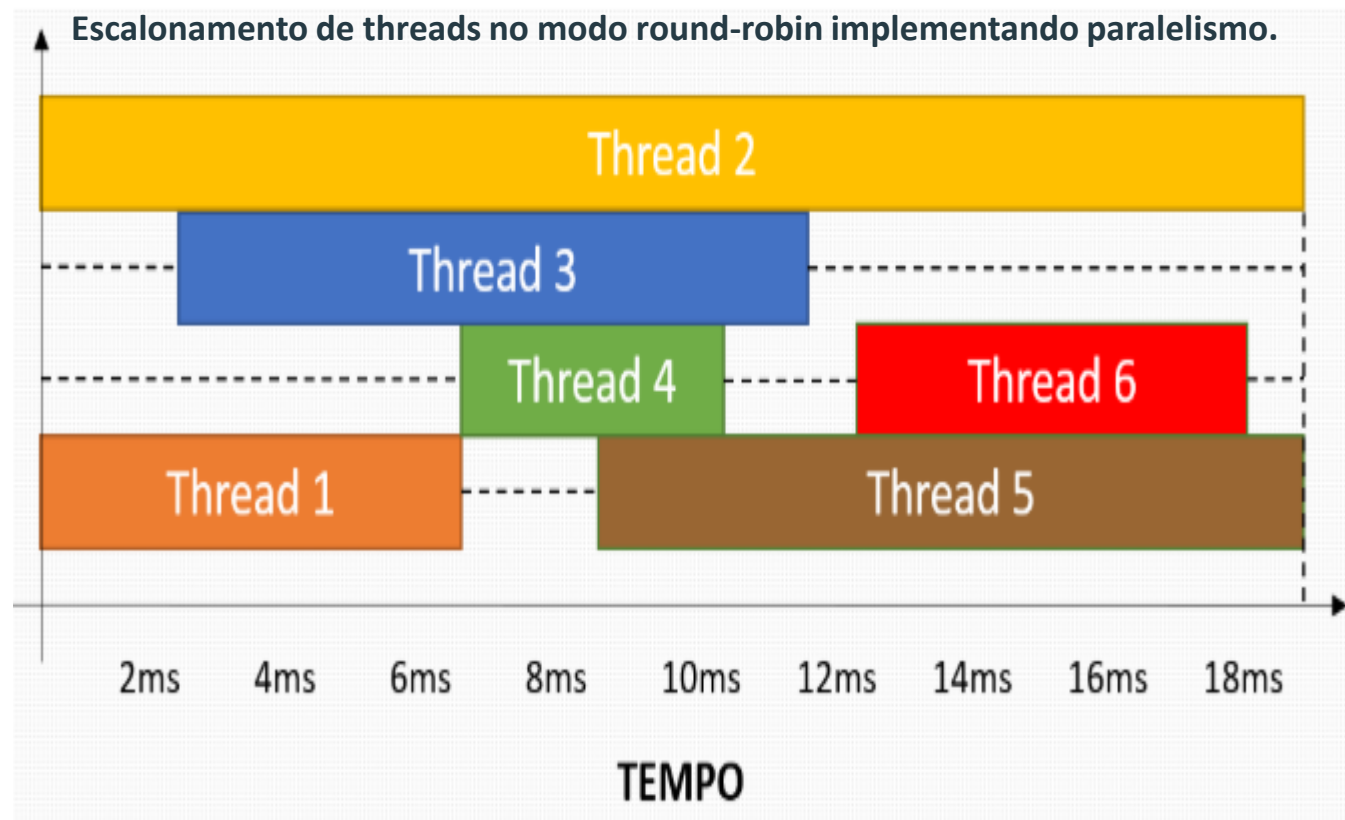




# Concorrência e Paralelismo

## Threads

- Aqui, por outro lado, temos um cenário bem diferente, com várias threads executando paralelamente e otimizando o uso da CPU.



# Implementação de Paralelismo e Concorrência

Implementação em java

# interface Runnable (java.lang.Runnable).

## Implementação em java

- Desde seu início a plataforma Java foi projetada para suportar programação concorrente.
- De lá para cá, principalmente a partir da versão 5, foram incluídas APIs de alto nível que nos fornecem cada vez mais recursos para a implementação de tarefas paralelas, como as APIs presentes nos pacotes **java.util.concurrent.\***.



# Concorrência e Paralelismo

## Implementação em java

- Toda aplicação Java possui, no mínimo, uma thread.
- Ela é criada e iniciada pela JVM quando iniciamos a aplicação e sua tarefa é executar o método **main()** da classe principal.
- Assim, executará sequencialmente os códigos contidos neste método até que **termine**, quando a thread encerrará seu processamento e a aplicação poderá ser finalizada.

# Concorrência e Paralelismo

## Implementação java

- Em Java, existem basicamente duas maneiras de criar threads:
  - Estender a classe **Thread** (**java.lang.Thread**); e
  - Implementar a interface **Runnable** (**java.lang.Runnable**).

# Concorrência e Paralelismo

- **Linhas de execução**

Para controlar o fluxo de execução de um programa, utiliza-se uma (ou várias) *linha de execução* que, em Java, é representada por uma instância da classe **Thread**, encontrada no pacote **java.lang**.



# Concorrência e Paralelismo

## ▪ Alguns métodos da classe Threads:

- `public void run()`  
Especifica as operações a serem realizadas pela linha de execução. Este método deve ser sobre-escrito pelo programador, codificando as operações em questão.
- `start()`  
Deslança a execução da linha de execução; não pode ser sobre-escrito pelo programador.
- `suspend()`  
Suspende a execução da linha de execução; não pode ser sobre-escrito pelo programador.
- `resume()`  
Retoma a execução de uma linha de execução que foi suspensa; não pode ser sobre-escrito pelo programador.
- `stop()`  
Encerra a execução da linha de execução; não pode ser sobre-escrito pelo programador.
- `sleep( long tempo )`  
Este é um método *estático*, ou seja cuja chamada afeta todas as instâncias da classe. O resultado da chamada é que todas as linhas de execução "dormem" durante o intervalo especificado pelo argumento, em milissegundos. Este método pode "lançar" uma exceção do tipo `InterruptedException` que *deve* ser capturada ou relançada pelo objeto que efetua a chamada. O lançamento e a captura de exceções é assunto para uma próxima aula, mas o código necessário no caso discutido aqui pode ser encontrado no exemplo abaixo.

# Implementação de Paralelismo e Concorrência

Interface Runnable (`java.lang.Runnable`).

# Interface **Runnable**

Exemplo de thread implementando a interface **Runnable**.

## Implementação java

- Ao lado, de forma simples e objetiva, é apresentado um exemplo de como implementar uma **Thread** para executar uma **subtarefa em paralelo**.

```
public class ExemploThread {  
  
    public static void main(String[ ] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //código para executar em paralelo  
                System.out.println("ID: " + Thread.currentThread().getId());  
                System.out.println("Nome: " + Thread.currentThread().getName());  
                System.out.println("Prioridade: " + Thread.currentThread().getPriority());  
                System.out.println("Estado: " + Thread.currentThread().getState());  
            }  
        }).start();  
    }  
}
```



# Interface Runnable

Exemplo de thread implementando a interface **Runnable**.

## Implementação java

- Neste exemplo pode-se observar também o código utilizado para buscar alguns dados da thread atual, tais como **ID, nome, prioridade, estado**.

```
public class ExemploThread {  
  
    public static void main(String[ ] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //código para executar em paralelo  
                System.out.println("ID: " + Thread.currentThread().getId());  
                System.out.println("Nome: " + Thread.currentThread().getName());  
                System.out.println("Prioridade: " + Thread.currentThread().getPriority());  
                System.out.println("Estado: " + Thread.currentThread().getState());  
            }  
        }).start();  
    }  
}
```

# Interface **Runnable**

Exemplo de thread implementando a interface **Runnable**.

## Implementação java

- Para isso, primeiramente é necessário codificar um **Runnable**, o que pode ser feito diretamente na criação da Thread, como demonstrado em amarelo ao lado, ou implementar uma classe própria que estenda **Runnable**

```
public class ExemploThread {  
  
    public static void main(String[ ] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //código para executar em paralelo  
                System.out.println("ID: " + Thread.currentThread().getId());  
                System.out.println("Nome: " + Thread.currentThread().getName());  
                System.out.println("Prioridade: " + Thread.currentThread().getPriority());  
                System.out.println("Estado: " + Thread.currentThread().getState());  
            }  
        }).start();  
    }  
}
```

# Interface Runnable

## Implementação java

- Em outras palavras, pode-se criar uma instância da classe **Thread**, passando como argumento do construtor uma referência ao objeto **Runnable** que implementa a interface.
- Este objeto passa então a atuar como "alvo" da linha de execução, o que significa que o método **run()** associado à linha de execução é na verdade aquele implementado no objeto **Runnable**

Exemplo de thread implementando a interface **Runnable**.

```
public class ExemploThread {  
  
    public static void main(String[ ] args) {  
        new Thread(new Runnable() {  
            @Override  
            public void run() {  
                //código para executar em paralelo  
                System.out.println("ID: " + Thread.currentThread().getId());  
                System.out.println("Nome: " + Thread.currentThread().getName());  
                System.out.println("Prioridade: " + Thread.currentThread().getPriority());  
                System.out.println("Estado: " + Thread.currentThread().getState());  
            }  
        }).start();  
    }  
}
```



# Implementação de Paralelismo e Concorrência

classe Thread (java.lang.Thread)

# Concorrência e Paralelismo

## Implementação java

- Além de tais informações que podem ser capturadas, **é possível manipular as threads** utilizando alguns dos seguintes métodos:
  - O método estático **Thread.sleep()**, por exemplo, faz com que a thread em execução espere por um período de tempo sem consumir muito (ou possivelmente nenhum) tempo de CPU;
  - O método **join()** congela a execução da thread corrente e aguarda a conclusão da thread na qual esse método foi invocado;
  - Já o método **wait()** faz a thread aguardar até que outra invoque o método **notify()** ou **notifyAll()**; e
  - O método **interrupt()** acorda uma thread que está dormindo devido a uma operação de **sleep()** ou **wait()**, ou foi bloqueada por causa de um processamento longo de I/O.

# Concorrência e Paralelismo

Código da classe Tarefa estendendo a classe Thread.

## Implementação java

- A forma clássica de se criar uma thread é estendendo a classe **Thread**, como demonstrado na **Listagem ao lado**.
- Neste código, temos a classe **Tarefa** estendendo a **Thread**.
- A partir disso, basta sobrescrever o método **run()**, o qual fica encarregado de executar o código da thread.

```
public class Tarefa extends Thread {  
  
    private final long valorInicial;  
    private final long valorFinal;  
    private long total = 0;  
  
    //método construtor que receberá os parâmetros da tarefa  
    public Tarefa(int valorInicial, int valorFinal) {  
        this.valorInicial = valorInicial;  
        this.valorFinal = valorFinal;  
    }  
  
    //método que retorna o total calculado  
    public long getTotal() {  
        return total;  
    }  
  
    /*  
    Este método se faz necessário para que possamos dar start() na Thread  
    e iniciar a tarefa em paralelo  
    */  
    @Override  
    public void run() {  
        for (long i = valorInicial; i <= valorFinal; i++) {  
            total += i;  
        }  
    }  
}
```



# Concorrência e Paralelismo

Código da classe Exemplo, utiliza a classe Tarefa.

## Implementação java

- Para testarmos o paralelismo com a classe da classe Tarefa, criamos a classe Exemplo com o método **main()**, responsável por executar o programa (ao lado).

```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```

# Concorrência e Paralelismo

Código da classe Exemplo, utiliza a classe Tarefa.

## Implementação java

- Neste exemplo, após criar as threads (em amarelo), chama-se o método **start()** (em vermelho) de cada uma delas, para que iniciem suas tarefas.

```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```

# Concorrência e Paralelismo

Código da classe Exemplo, utiliza a classe Tarefa.

## Implementação java

- Logo após, em um bloco *try-catch*, temos a invocação dos métodos **join()** (em vermelho).
- Ele faz com que o programa aguarde a finalização de cada thread para que depois possa ler o valor totalizado por cada tarefa.

```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```



# Concorrência e Paralelismo

Código da classe Exemplo, utiliza a classe Tarefa.

## Implementação java

- Observe, que cada tarefa recebe seu intervalo de valores a calcular, sendo somado, ao todo, de 0 a 3000.

```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```

# Concorrência e Paralelismo

## Implementação java

- Vamos acrescentar o print dos resultados de t1, t2 e t3, antes do total geral.

Código da classe Exemplo, utiliza a classe Tarefa.

```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("t1: " + t1.getTotal());  
        System.out.println("t2: " + t2.getTotal());  
        System.out.println("t3: " + t3.getTotal());  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```

# Concorrência e Paralelismo

Código da classe Exemplo, utiliza a classe Tarefa.

## Implementação java

- Observe, na **ao lado**, que cada tarefa recebe seu intervalo de valores a calcular, sendo somado, ao todo, de 0 a 3000, mas e se tivéssemos uma única lista de valores que gostaríamos de somar para obter o valor total?
- Neste caso, as threads precisariam concorrer pela lista.
- Isso é o que chamamos de concorrência de dados e geralmente traz consigo diversos problemas.

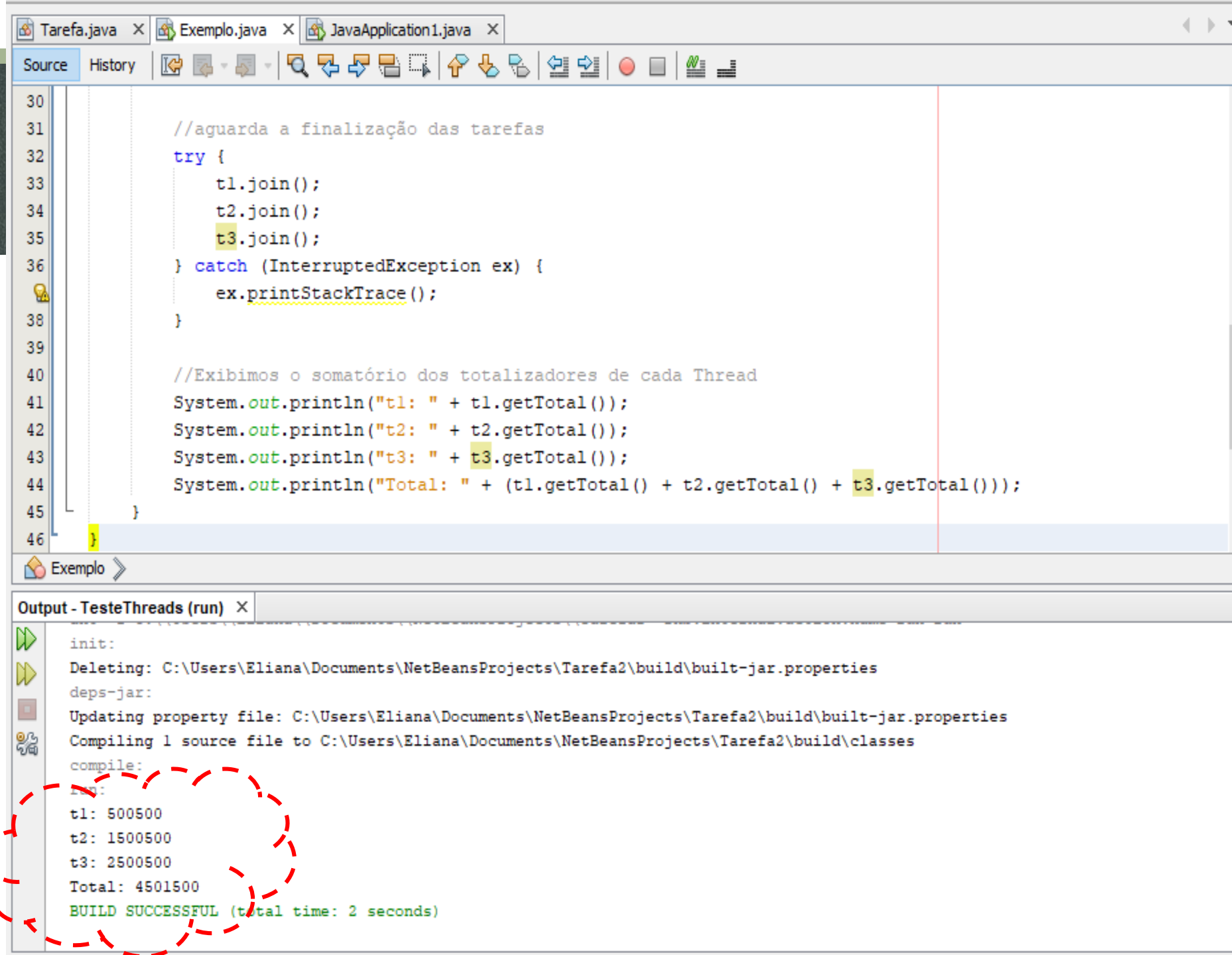
```
public class Exemplo {  
  
    public static void main(String[] args) {  
        //cria três tarefas  
        Tarefa t1 = new Tarefa(0, 1000);  
        t1.setName("Tarefa1");  
        Tarefa t2 = new Tarefa(1001, 2000);  
        t2.setName("Tarefa2");  
        Tarefa t3 = new Tarefa(2001, 3000);  
        t3.setName("Tarefa3");  
  
        //inicia a execução paralela das três tarefas, iniciando três novas threads no programa  
        t1.start();  
        t2.start();  
        t3.start();  
  
        //aguarda a finalização das tarefas  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException ex) {  
            ex.printStackTrace();  
        }  
  
        //Exibimos o somatório dos totalizadores de cada Thread  
        System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));  
    }  
}
```



# Concorrência e Paralelismo

## Implementação java

- Resultado da execução, implementado no NetBeans...



The screenshot displays the NetBeans IDE interface. The top pane shows the source code of a Java application with three threads (t1, t2, t3) and their total values. The bottom pane shows the output of the program, which includes the same total values and a successful build message. A red dashed circle highlights the output values, and red parentheses are placed next to the text 'Resultado da execução, implementado no NetBeans...'.

```
30
31 //aguarda a finalização das tarefas
32 try {
33     t1.join();
34     t2.join();
35     t3.join();
36 } catch (InterruptedException ex) {
37     ex.printStackTrace();
38 }
39
40 //Exibimos o somatório dos totalizadores de cada Thread
41 System.out.println("t1: " + t1.getTotal());
42 System.out.println("t2: " + t2.getTotal());
43 System.out.println("t3: " + t3.getTotal());
44 System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));
45 }
46 }
```

Output - TesteThreads (run) X

```
init:
Deleting: C:\Users\Eliana\Documents\NetBeansProjects\Tarefa2\build\build-jar.properties
deps-jar:
Updating property file: C:\Users\Eliana\Documents\NetBeansProjects\Tarefa2\build\build-jar.properties
Compiling 1 source file to C:\Users\Eliana\Documents\NetBeansProjects\Tarefa2\build\classes
compile:
run:
t1: 500500
t2: 1500500
t3: 2500500
Total: 4501500
BUILD SUCCESSFUL (total time: 2 seconds)
```

# Obrigada!

