



Linguagem de Programação III (LP35A)

Prof^a Eliana Santos

Linguagem de Programação III (LP35A)

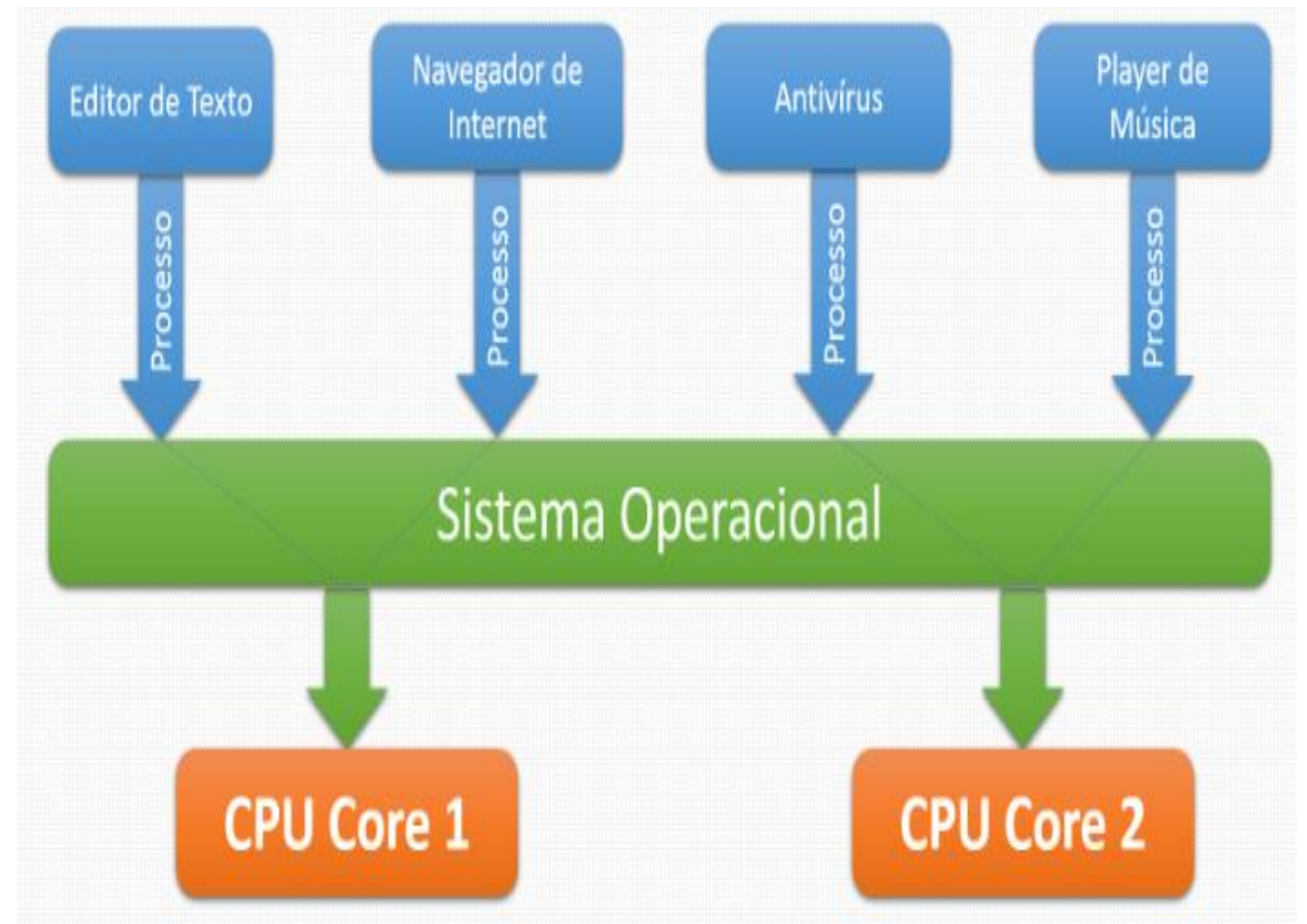
Problemas típicos e *patterns* para resolução

Linguagem de Programação III (LP35A)

Resumo – última aula

Resumo última aula

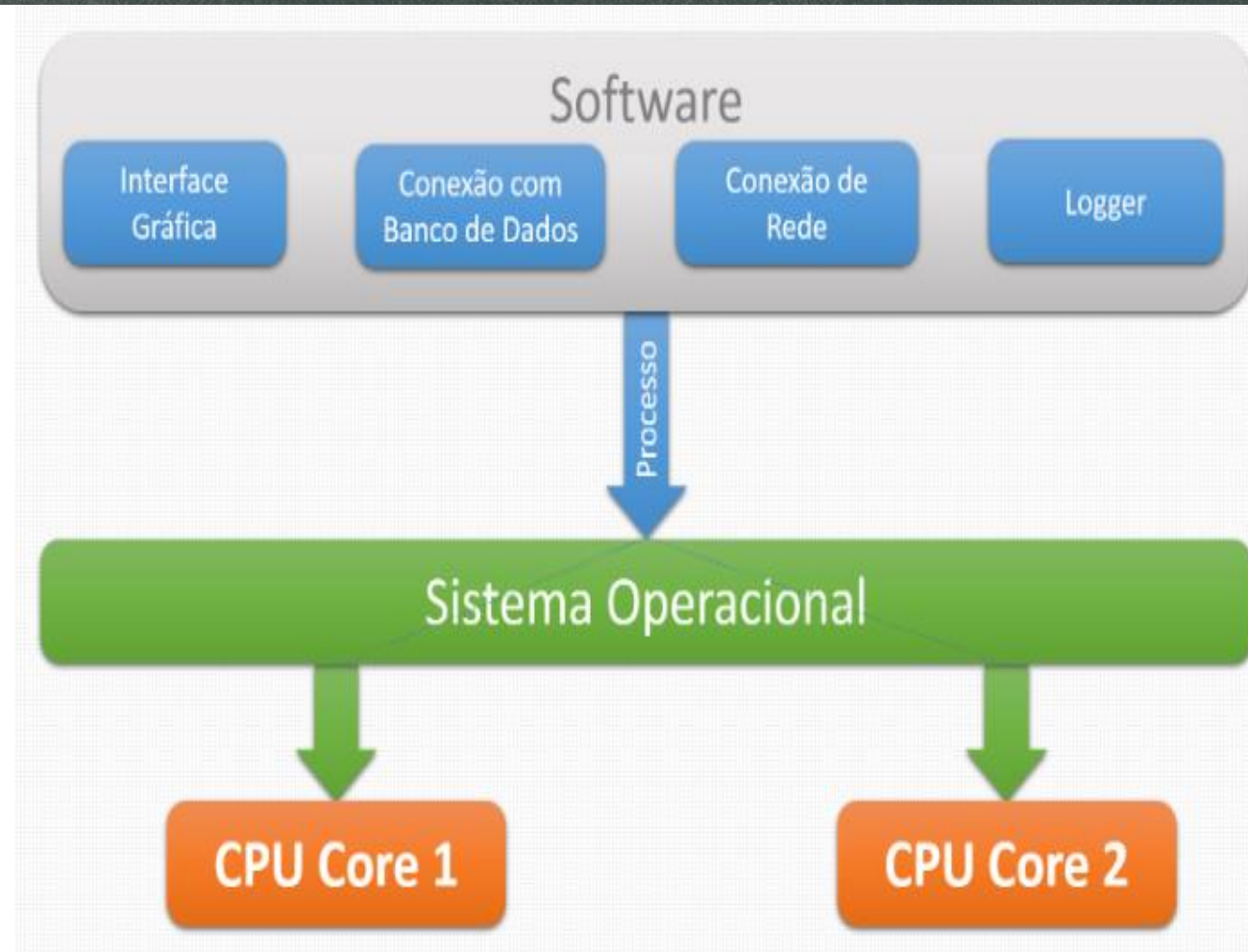
Multitasking



Resumo última aula

Multithreading

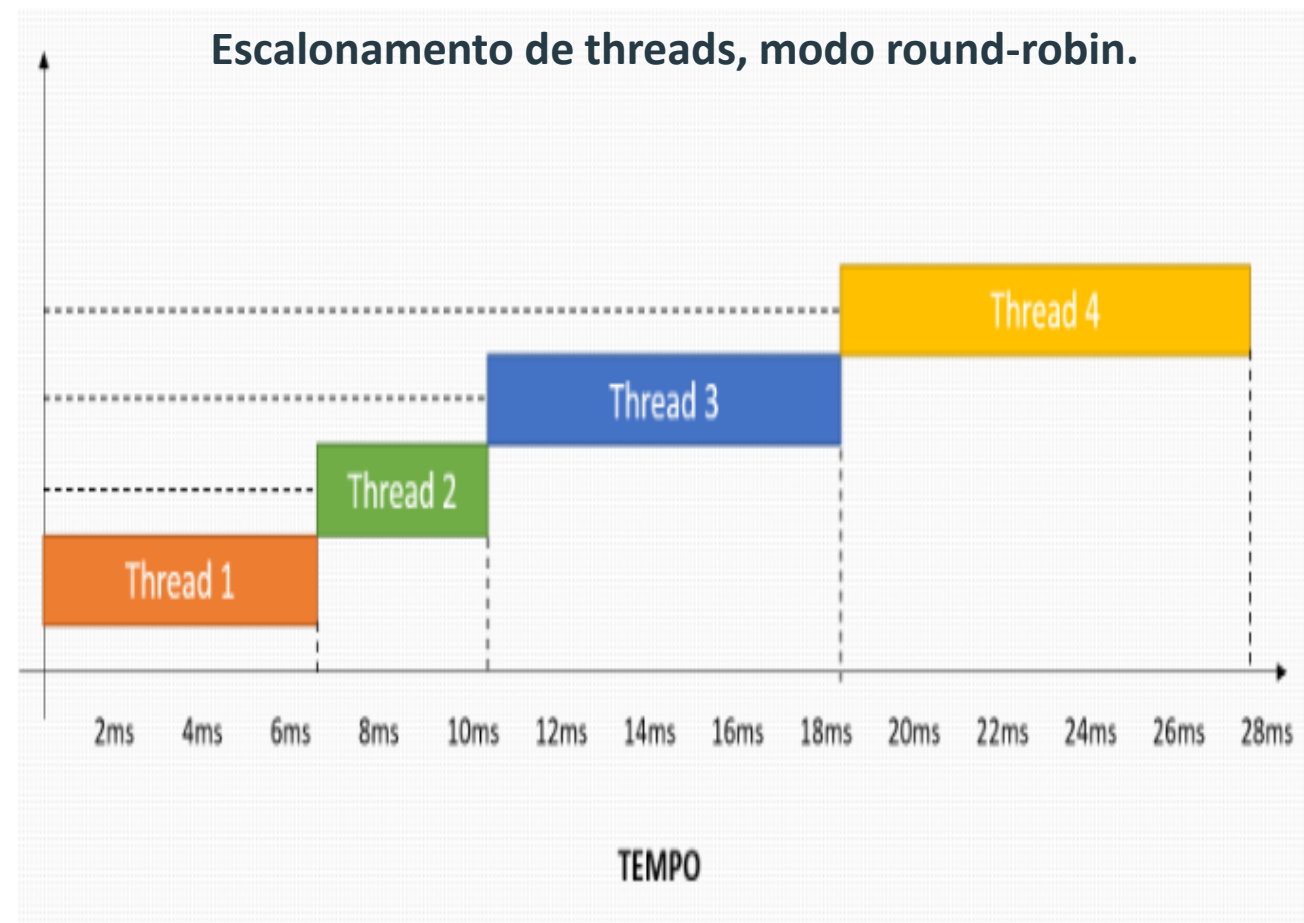
Sabendo dessa importância, nosso próximo passo foi entender **o que são as threads e como criá-las para subdividir as tarefas do software.**



Concorrência e Paralelismo

Threads

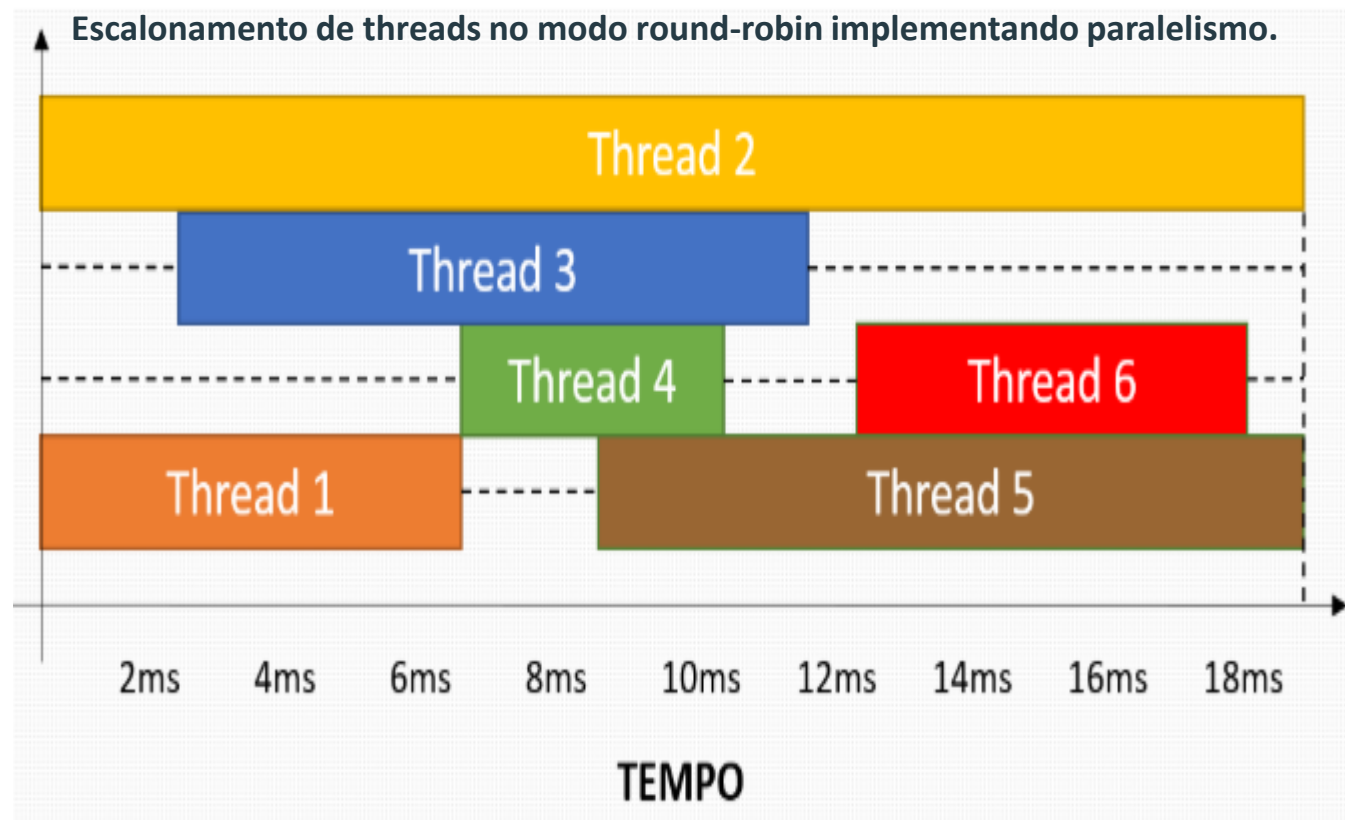
- Enquanto os processos de software não dividem um mesmo espaço de memória, as threads, sim, e isso lhes permite compartilhar dados e informações dentro do contexto do software.



Concorrência e Paralelismo

Threads

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



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();  
    }  
}
```


Concorrência e Paralelismo

Código da classe Tarefa estendendo a classe Thread.

Implementação java

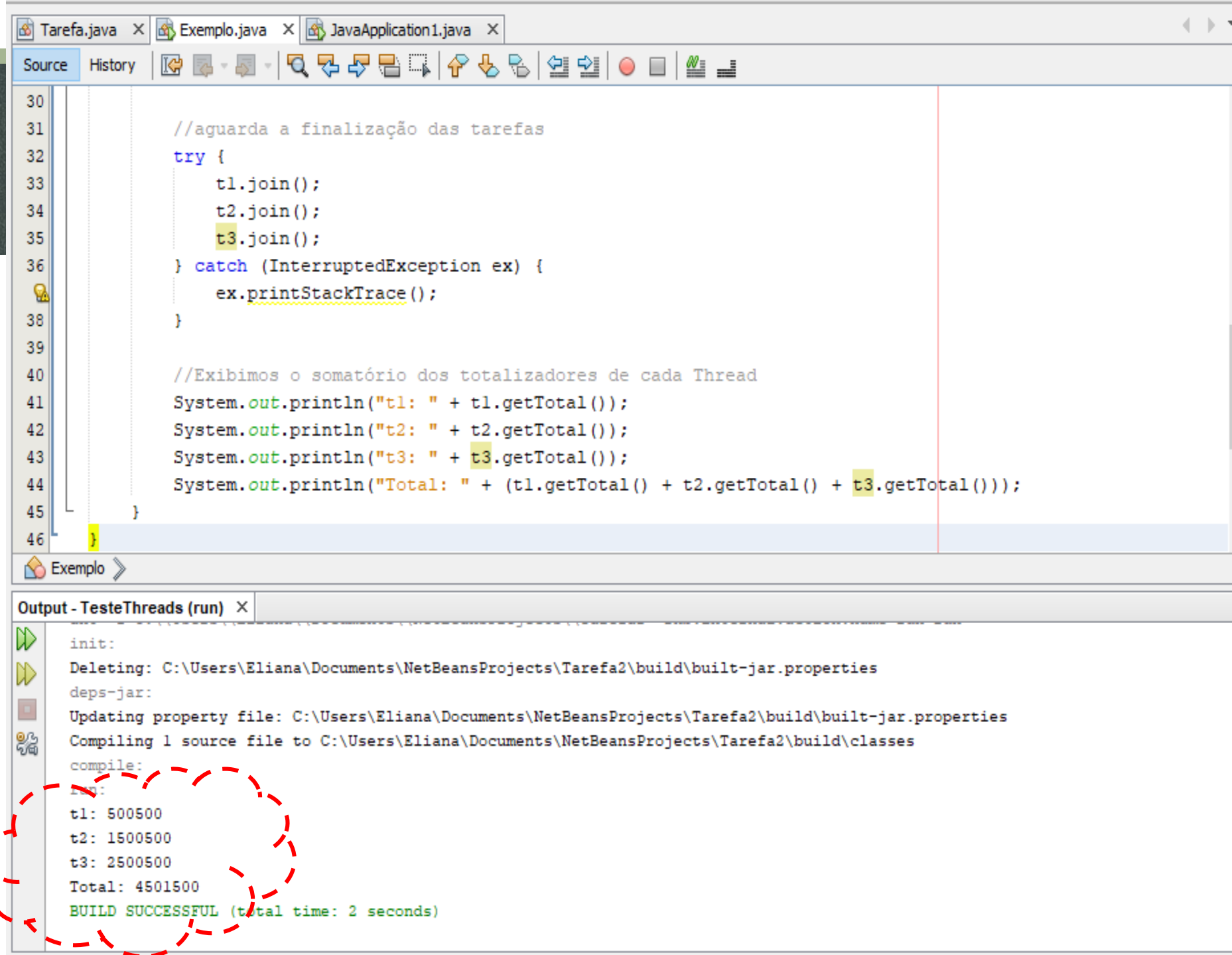
- A forma clássica de se criar uma thread é estendendo a classe **Thread**, como demonstrado na **Listagem ao lado**.
- Neste código, temos a classe **Tarefa** estendendo a **Thread**.
- A partir disso, basta sobrescrever o método **run()**, o qual fica encarregado de executar o código da thread.

```
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

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 total values of the threads and a successful build message.

```
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)
```


Linguagem de Programação III (LP35A)

Aprofundamento em threads:
compartilhamento de memória

Thread

Thread é o nome dado a um programa (uma sequência de instruções) em execução. A palavra thread, em português, significa linha (de costura). Um thread é portanto uma linha contínua de execução, uma sequência de instruções que pode, potencialmente, executar em paralelo com outros threads. Apesar de ser uma sequência contínua, um thread não precisa executar inteiro de uma vez. Ele pode ser interrompido quantas vezes for necessário, sempre continuando do ponto onde parou.

Thread

Um thread pode acontecer de forma *exclusiva* (como único programa executando em uma CPU), *concorrente* (sendo constantemente interrompido para revezar a CPU com outros programas) ou *paralela* (ocupando uma CPU em um sistema com múltiplos processadores, e executando simultaneamente com outros programas, sem interrupções).

situações:

Threads

- A execução paralela **só é possível em sistemas que possuem múltiplas unidades de processamento** (CPUs). Isto ocorre em computadores que têm mais de um processador (múltiplos chips), ou em processadores que disponibilizam mais de um núcleo (core) de processamento.
- Em 2015 a maior parte dos computadores pessoais tinham processadores com pelo menos dois cores, e alguns já rodavam com 16 cores.

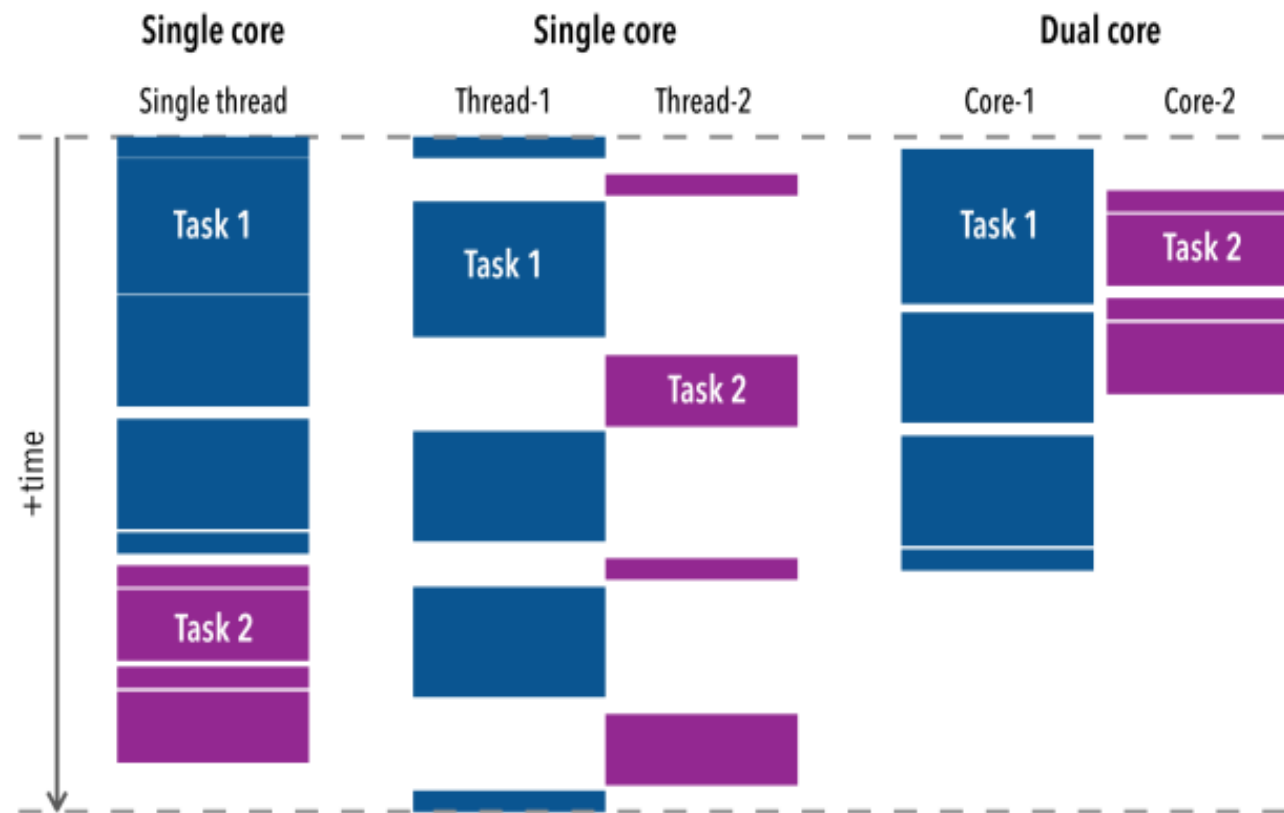


Figura 1.1 Threads executando de forma exclusiva (sequencialmente, em um único processador), concorrente (ocupando um único processador de forma alternada), e paralela (executando simultaneamente, em cores ou processadores paralelos).

Threads

- Um programa pode rodar sequencialmente em um computador com múltiplas CPUs. Esse é o comportamento *default*.
- Mas ele também pode **criar threads adicionais que irão executar de forma concorrente ou paralela.**
- **Dependendo da natureza da aplicação, isto poderá melhorar a sua performance, e pode ser essencial para o seu funcionamento (ex: aplicações gráficas, aplicações derede).**
- Por outro lado, o uso de threads aumenta consideravelmente a complexidade, exigindo o uso de mecanismos especiais de sincronização para garantir a ordem de execução e proteção de dados compartilhados.

Threads - exemplo

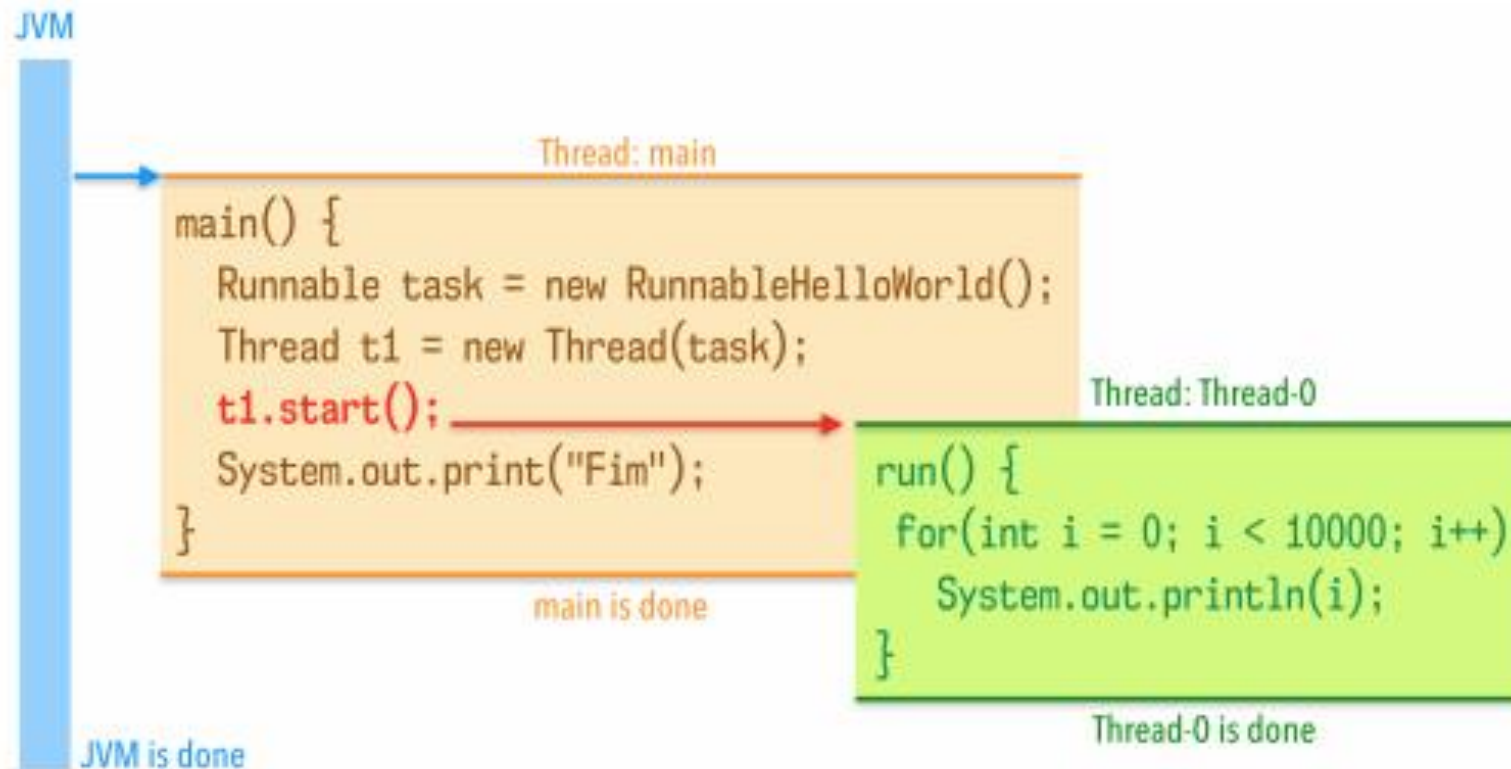


Figura 2.1 Execução de um novo thread a partir do thread "main". A palavra "Fim" será provavelmente impressa antes que o loop comece a imprimir números. O thread main provavelmente terminará antes do Thread-0.

Threads - exemplo

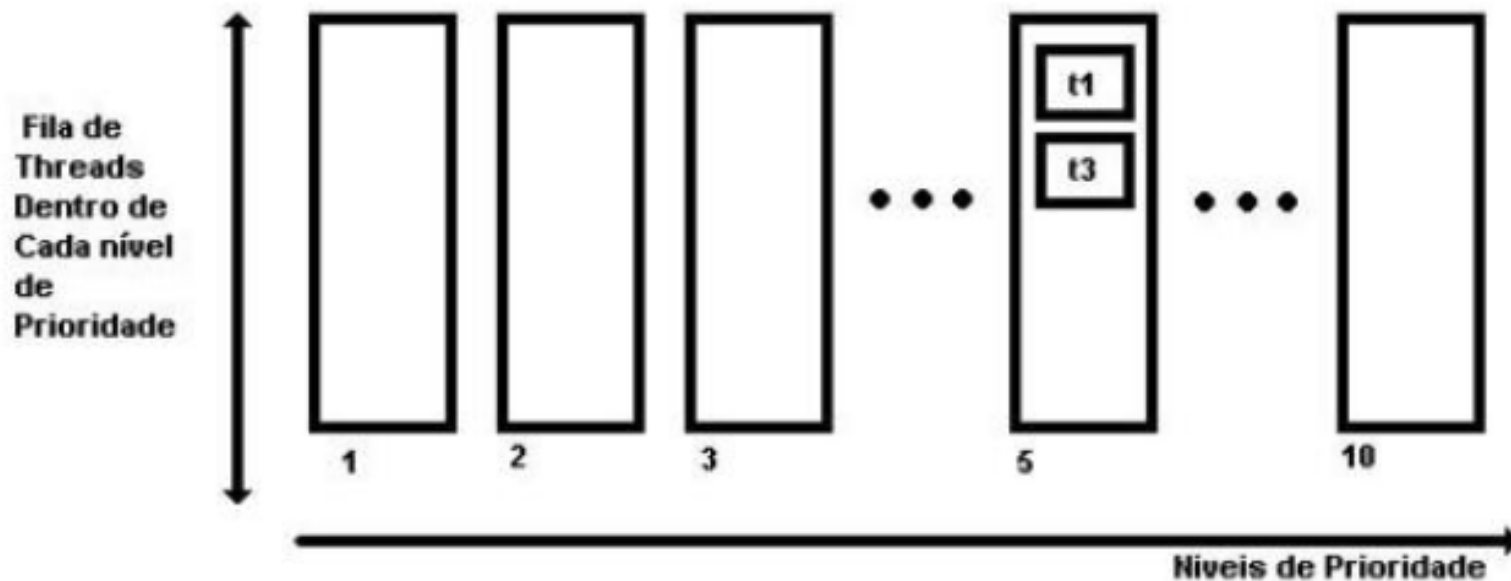
As threads sempre iniciam sua execução com a prioridade herdada da superclasse, que por default é igual a 5 (`Thread.NORM_PRIORITY`). Porém, o programador pode alterar a prioridade da thread como convier, dentro do range de prioridades de 1 (`Thread.MIN_PRIORITY`) e 10 (`Thread.MAX_PRIORITY`).

A JVM tem um dispositivo que escala as threads. Este dispositivo sempre tenta escalonar a thread de maior prioridade que esteja no estado executável. A thread escalonada passa a promover outras threads nos seguintes momentos:

- 25.Quando é chamado o método `yield()`;
- 26.Quando passa para o estado de encerrada ou bloqueada;
- 27.Quando passa a estar executável outra thread de prioridade mais alta.

Threads - exemplo

- Dentro de cada nível de prioridade existe um conceito de fila, para que seja possível escalonar todas as threads de mesma prioridade, fornecendo os mesmos privilégios de escalonamento.



Threads

- A programação com threads possui regras próprias, padrões e recomenda até mesmo paradigmas diferentes.
- **Mesmo que seu programa nunca crie um thread, você pode estar usando uma biblioteca ou framework que cria threads, e precisa estar atento às consequências.**
- Escrever código imune a threads (thread-safe) requer o controle do acesso ao seu estado, especialmente estado mutável e compartilhado.
- Java possui suporte nativo à programação com threads e recursos para lidar com alguns dos problemas da programação concorrente

Implementação de Paralelismo e Concorrência

O método INTERRUPT

Controle de Threads

Interrupção de um thread: interrupt()

- Vimos que é muito fácil criar e iniciar um thread. E para interrompê-lo?
- Uma vez iniciado, um thread só termina (normalmente) quando seu método run() terminar.

- Interrupção é gerada invocando o método `interrupt` do objeto thread a ser interrompido

```
public void interrupt()
```

```
...  
t.interrupt();  
...
```

Controle de Threads

Interrupção de um thread: interrupt()

- Esta técnica de *interrupção*, envolve **ligar ou desligar um flag** (que chamaremos de INTERRUPT) no thread indicando se ele deve ou não ser interrompido.
- **Ligar o flag INTERRUPT é apenas uma sinalização e não causa a interrupção do thread**, mas *pode* (normalmente *deve*) ser usado pelo programa para tomar a decisão de finalizar ou continuar a sua execução.
- A finalização deve ocorrer *normalmente*, ou seja, o código deve levar a execução para que o método run() termine.

```
public void interrupt()
```

```
...  
t.interrupt();  
...
```

Controle de Threads

```
public class InterruptRunnable implements Runnable {  
    @Override public void run() {  
        boolean interrupt = false;  
        while(!interrupt) {  
            interrupt = Thread.interrupted();  
            System.out.println(">INTERRUPT flag: " + interrupt);  
        }  
        System.out.println("INTERRUPTED flag: " + interrupt);  
        System.out.println("Thread " + Thread.currentThread().getName() + " is DONE!");  
    }  
}
```

Exemplo

- O exemplo abaixo demonstra como `interrupted()` pode ser usado para finalizar um thread que executa indefinidamente dentro de um bloco `while()`, imprimindo o valor de `INTERRUPT`.
- O bloco `while()` testa, a cada repetição, a variável que contém o valor booleano retornado pelo método `interrupted()`, que será sempre `false` enquanto `INTERRUPT` estiver desligado.
- Quando outro thread chamar o método `interrupt()` deste thread, `INTERRUPT` será ligado, a variável mudará para `true` forçando a saída do loop, terminando o método `run()` e causando a finalização normal do thread:

Controle de Threads

Exemplo

- Esta classe cria e inicia um thread, e depois roda um loop demorado para permitir que o esse thread execute algumas vezes.
- Em seguida, chama o método `interrupt()`, que irá inverter o estado de `INTERRUPT`.

```
public class InterruptFlagExample {  
    public static void main(String[] args) {  
        Runnable runnable = new InterruptRunnable();  
        Thread t1 = new Thread(runnable);  
        t1.start();  
  
        for(int i = 0; i < 1_000_000; i++) {  
            // atrasa thread principal para que o thread t1 possa executar algumas vezes  
        }  
  
        t1.interrupt(); // esta chamada irá ligar o flag de interrupção em t1  
        System.out.println("Thread " + Thread.currentThread().getName() + " is DONE!");  
    }  
}
```

Controle de Threads

Executando o código acima teremos o seguinte resultado:

```
>INTERRUPTED flag: false  
... (várias execuções)  
>INTERRUPTED flag: false  
>INTERRUPTED flag: false  
>INTERRUPTED flag: true  
INTERRUPTED flag: true  
Thread main is DONE!  
Thread Thread-0 is DONE!
```

Há outras maneiras de escrever o código acima, mas é importante lembrar que `interrupted()` faz *duas* coisas: inverte o estado da flag `INTERRUPT` e retorna o estado atual. Para apenas obter o estado do thread sem causar efeitos colaterais use `isInterrupted()`.

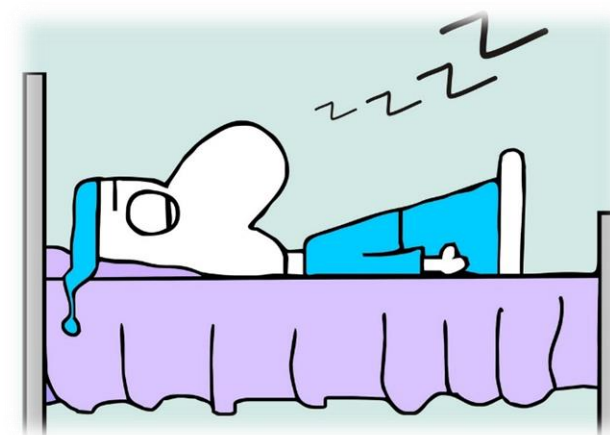
Implementação de Paralelismo e Concorrência

O método sleep ()

Controle de Threads

- Pondo um thread para dormir: `Thread.sleep()`

No exemplo anterior fizemos um thread atrasar um pouco rodando um loop de muitas repetições. Esta, obviamente, não é uma forma muito eficiente de gerar atrasos, já que ocupa a CPU e não garante um atraso consistente em diferentes plataformas (o tempo será menor em máquinas mais rápidas). Uma maneira mais eficiente de fazer isto é *colocar o thread para dormir*.

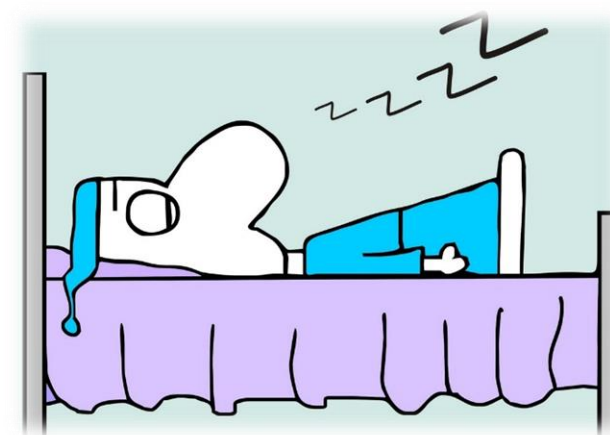


Controle de Threads

- Pondo um thread para dormir: `Thread.sleep()`
- Um thread que está ocupando a CPU pode interromper sua execução temporariamente chamando o método estático `Thread.sleep(milissegundos)`. Por exemplo, a instrução:

`Thread.sleep(1000);`

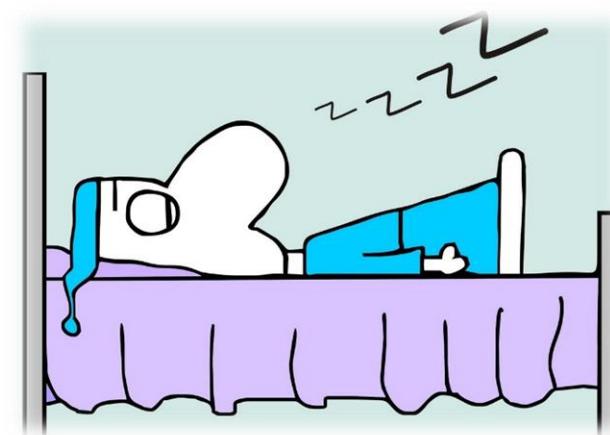
- Faz com que o thread executando no momento durma por pelo menos 1 segundo (1000 milissegundos).
- **Nesse intervalo, outros threads que estejam esperando acesso à CPU terão oportunidade de executar, independente de sua prioridade.**



Controle de Threads

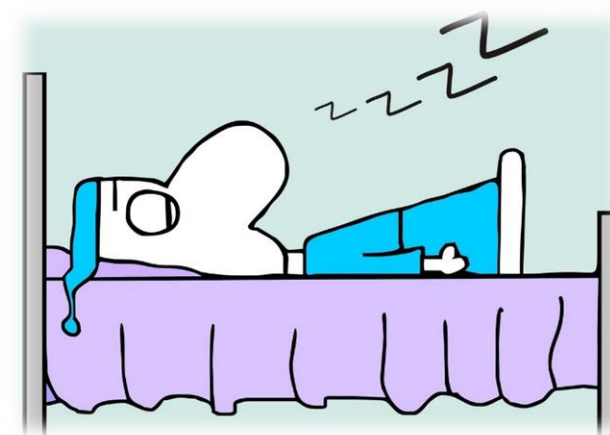
Se a flag `INTERRUPT` for ativada em um thread durante a execução de `sleep()`, será lançada uma `InterruptedException`. Essa é uma exceção checada e por isto é obrigatório que ela seja tratada (normalmente em um bloco try-catch). Se `Thread.sleep()` está sendo usado apenas para gerar um atraso em um ambiente de testes, é comum usar um bloco catch vazio para ignorar a exceção. A interrupção, se ocorrer, poderá encurtar a duração do `sleep()` mas o bloco catch vazio irá impedir a finalização do thread.

```
try {  
    Thread.sleep(1000);  
} catch (InterruptedException ignorada) { /* ignorando a interrupção */ }
```



Controle de Threads

- Mas normalmente você irá querer finalizar o thread quando o `sleep()` for interrompido.
- Outra maneira de interrompê-lo é esperar que uma interrupção ocorra quando o thread estiver dormindo, já que isto lançará uma exceção.
- Para isto é necessário que a rotina inclua chamadas periódicas a `sleep()`.



Controle de Threads

Exemplo 2 – sleep ()

```
public class RandomLetters implements Runnable {  
    @Override public void run() {  
        try {  
            while(true) {  
                System.out.print(" " + (char)('A' + new Random().nextInt(26)));  
                Thread.sleep(200);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("\n" + Thread.currentThread().getName() + " interrupted.");  
            System.out.println("INTERRUPTED flag: " + Thread.currentThread().isInterrupted());  
        }  
        System.out.println("Thread " + Thread.currentThread().getName() + " is DONE!");  
    }  
}
```

- O exemplo ilustra essa situação. O método run() da classe RandomLetters imprime uma sequência infinita de letras usando um bloco while(true), mas espera por 200 milissegundos entre a impressão de cada letra.
- Se INTERRUPT for ligado durante uma repetição, a próxima chamada a sleep() irá provocar uma InterruptedException que finaliza o thread:

Controle de Threads

```
public class InterruptSleepExample {  
    public static void main(String[] args) {  
        Runnable runnable = new RandomLetters();  
        Thread t1 = new Thread(runnable);  
        t1.start();  
        try { Thread.sleep(2000); } catch (InterruptedException ignored) {}  
  
        t1.interrupt(); // sets interrupt flag in t1  
  
        System.out.println("\nThread " + Thread.currentThread().getName() + " is DONE!");  
    }  
}
```

Exemplo 2 – sleep ()

- A classe acima cria e inicia um novo thread (implementado pela classe RandomLetters), e espera dois segundos antes de chamar interrupt().

Controle de Threads

Abaixo está um possível resultado da execução do programa acima:

```
V Z S G F C U Z P I  
Thread main is DONE!
```

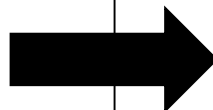
```
Thread-0 was interrupted.  
INTERRUPTED flag is now: false  
Thread Thread-0 is DONE!
```

O flag `INTERRUPT` era `true` quando causou a exceção, mas no bloco `catch` já é `false` porque ele *é desligado automaticamente* pelo método `sleep()`, pouco depois que acontece a exceção.

Controle de Threads

- Quando a `InterruptedException` for capturada e *não houver intenção de finalizar o thread*, o flag `INTERRUPT` *deve ser redefinido*, para permitir que outro código lide com a interrupção se desejar:

```
public class RandomLetters implements Runnable {
    @Override public void run() {
        try {
            while(true) {
                System.out.print(" " + (char)('A' + new Random().nextInt(26)));
                Thread.sleep(200);
            }
        } catch (InterruptedException e) {
            System.out.println("\n" + Thread.currentThread().getName() + " interrupted.");
            System.out.println("INTERRUPTED flag: " + Thread.currentThread().isInterrupted());
        }
        System.out.println("Thread " + Thread.currentThread().getName() + " is DONE!");
    }
}
```



```
public void run() {
    while(true) { // loop continua mesmo com interrupção
        try {
            // chamadas externas que poderão lidar com o INTERRUPT
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.out.println("Thread interrompido que não será finalizado.");
            Thread.currentThread().interrupt(); // IMPORTANTE!
        }
    }
    System.out.println("Thread finalizado.");
}
```

Controle de Threads

- Havendo um bloco try-catch que captura `InterruptedException` para finalizar um thread (não é o caso do exemplo anterior), ele deve estar sempre *fora de um loop*, não apenas devido à performance, mas também porque o código fica mais simples e legível. Esta é uma estrutura padrão para métodos `run()`:

```
public void run() {  
    while(true) { // loop continua mesmo com interrupção  
        try {  
            // chamadas externas que poderão lidar com o INTERRUPT  
            Thread.sleep(100);  
        } catch(InterruptedException e) {  
            System.out.println("Thread interrompido que não será finalizado.");  
            Thread.currentThread().interrupt(); // IMPORTANTE!  
        }  
    }  
    System.out.println("Thread finalizado.");  
}
```



```
public void run() {  
    try {  
        while(true) {  
            Thread.sleep(100);  
        }  
    } catch(InterruptedException e) {  
        System.out.println("Thread interrompido.");  
    }  
    System.out.println("Thread finalizado.");  
}
```


Controle de Threads

Algumas observações sobre o método sleep()

- O método sleep() também **pode ser chamado com dois argumentos**, sendo o segundo um valor em nanossegundos (mas o sistema não garante essa precisão).
- É útil para gerar delays como nos exemplos mostrados quando a precisão não for uma questão importante.

Controle de Threads

Algumas observações sobre o método sleep()

- **CUIDADO!** Thread.sleep() não é um mecanismo de agendamento de threads!
- Embora seja comum fazer chamadas periódicas ao método sleep() no thread que está executando para garantir que outros threads tenham oportunidade de executar, **essa não é a melhor solução e poderá trazer problemas em aplicações mais complexas**, por exemplo, em aplicações onde muitos threads disputam o acesso a recursos escassos e precisam esperar que outros threads terminem.
- É possível realizar sincronização com base em *notificações* usando wait() e notify(), que é uma solução mais eficiente e mais segura para esses casos.

Controle de Threads

Algumas observações sobre o método sleep()

- **CUIDADO!!** Thread.sleep() também não é a melhor solução para construir um temporizador!
- Existem classes específicas na API Java para tarefas que precisam ser executadas depois de um intervalo, em determinada data, ou que precisam ser repetidos periodicamente.
- Para agendar ou repetir um único thread pode-se usar a classe java.util.Timer e TimerTask (introduzidas no Java 1.3).
- Para agendar a execução e repetição de vários threads é mais eficiente usar um ScheduledThreadPoolExecutor (disponível a partir do Java 5).

Implementação de Paralelismo e Concorrência

O método join ()

Controle de Threads:

- Muitas vezes uma tarefa executada em paralelo produz resultados que precisam ser usados por outro thread.
- Para isto é necessário saber se os resultados estão prontos, ou seja, se a tarefa terminou normalmente.
- Em programação sequencial isto é fácil: basta ordenar as instruções.
- A instrução seguinte só irá executar quando a anterior terminar.
- **Mas em uma aplicação concorrente isto não funciona, já que as tarefas rodam em paralelo.**

Controle de Threads:

- Suponha que temos três threads realizando tarefas parciais cujos resultados depois precisam ser combinados.
- **Como garantir que uma rotina de combinação só execute quando todas as outras terminarem?**
- Uma maneira simples de fazer isto seria usando Thread.sleep().
- Pode-se fazer o thread principal dormir **enquanto o outros trabalham**, determinando um intervalo de tempo suficiente para que os outros threads tenham tempo de terminar suas tarefas.

Controle de Threads:

- O exemplo a seguir espera 10 segundos:

```
public class SleepExample {  
    public static void main(String[] args) {  
        Runnable r1 = new SimpleDelay(2000);  
        Runnable r2 = new SimpleDelay(5000);  
        new Thread(r1).start();  
        new Thread(r2).start();  
        try { Thread.sleep(10000); } catch (InterruptedException ignored) {}  
        System.out.println("Thread main is DONE!");  
    }  
}
```

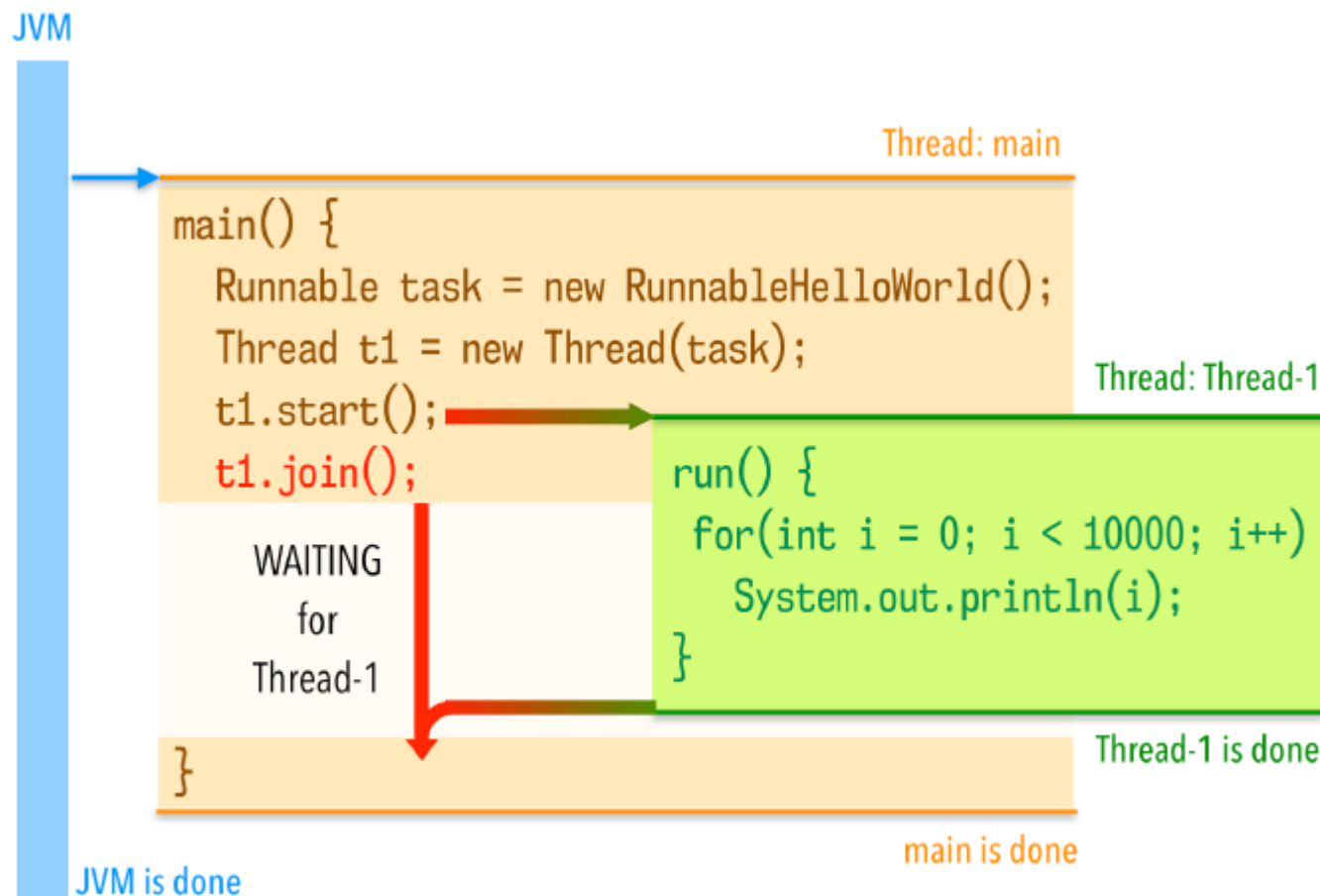
```
public class SimpleDelay implements Runnable {  
    int delay;  
    public SimpleDelay(int delay) {  
        this.delay = delay;  
    }  
    @Override public void run() {  
        System.out.println(Thread.currentThread().getName() + " started.");  
        try { Thread.sleep(delay); } catch (InterruptedException ignored) {}  
        System.out.println(Thread.currentThread().getName() + " finished.");  
    }  
}
```

Controle de Threads

- Embora a solução apresentada funcione como solução de sincronização, ela é ineficiente, pois haverá desperdício de tempo se os outros threads terminarem muito antes.
- Pior é o risco de alguma tarefa durar mais que o esperado e seu resultado não estar pronto na hora que o thread principal acordar.

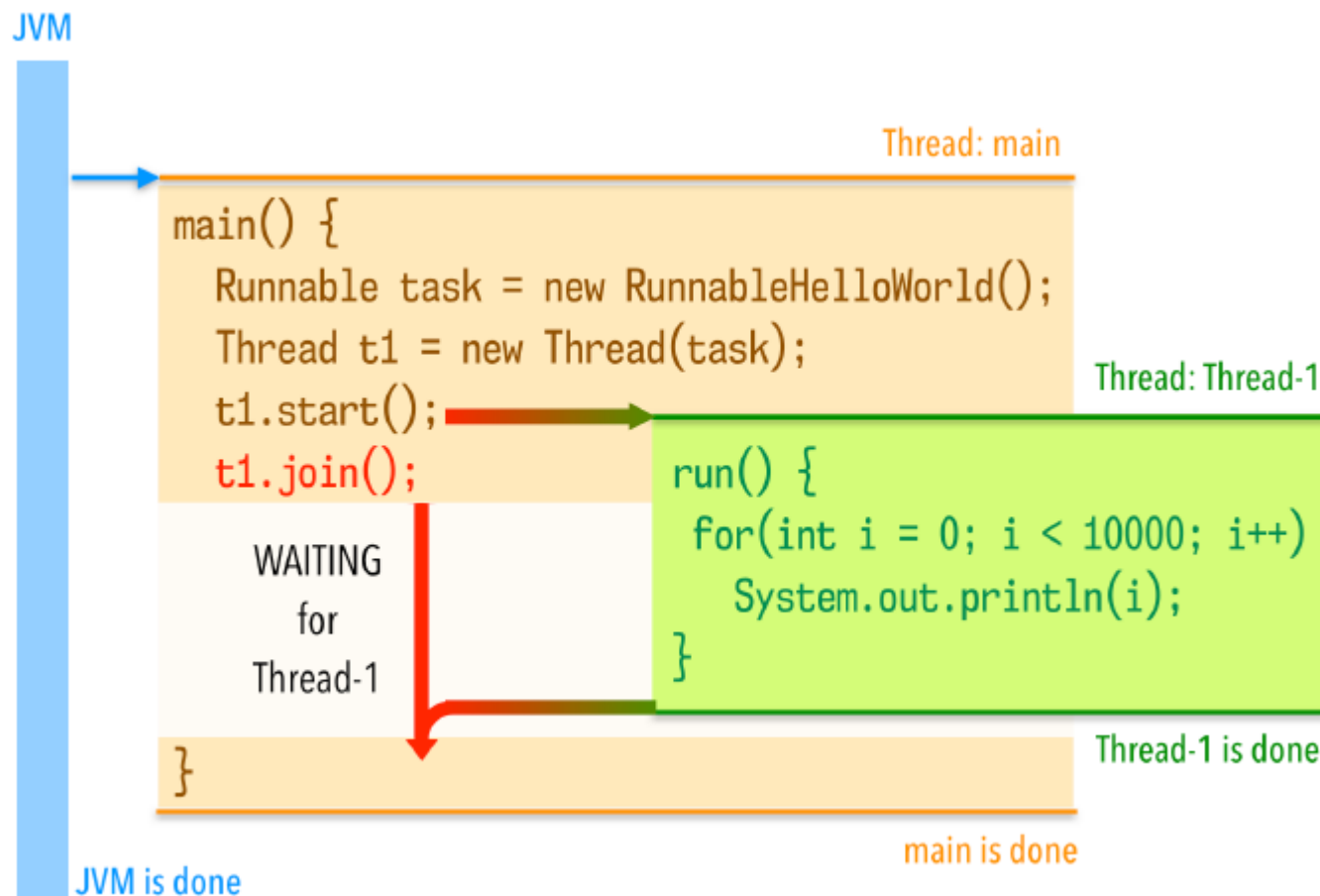
Controle de Threads

- Uma solução mais eficiente é usar o método de instância `join()`.
- Ao ser chamado, o método `join()` faz *o thread ao qual pertence esperar pelo fim do thread que chamou o método*



Controle de Threads

- O método `join()`, como todos os métodos que suspendem threads, lança `InterruptedException`, que precisa ser capturada ou declarada.



Esperando um thread terminar: join()

Exemplo implementado com join():

- O thread principal (main) chama o join() do primeiro thread secundário e suspende o fluxo de execução neste ponto.
- Quando o primeiro thread terminar, main prossegue.
- Se o segundo thread já tiver terminado, ele finaliza.
- Caso contrário suspende novamente a execução de main no ponto onde é chamado o segundo join().
- Quando ambos os threads secundários terminarem, main imprime a última linha e termina.

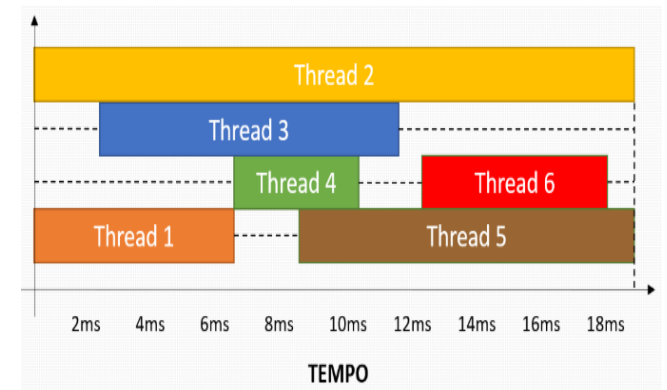
```
public class JoinExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(new SimpleDelay(2000));  
        t1.start();  
        Thread t2 = new Thread(new SimpleDelay(5000));  
        t2.start();  
  
        System.out.println("Waiting for " + t1.getName());  
        try {  
            t1.join();  
        } catch (InterruptedException e) { ... }  
        System.out.println("Waiting for " + t2.getName());  
        try {  
            t2.join();  
        } catch (InterruptedException e) { ... }  
        System.out.println("Thread main is DONE!");    }  
    }  
}
```

Esperando um thread terminar: join()

Exemplo implementado com join():

- Neste exemplo, o thread principal (main) chama o join() do primeiro thread secundário e suspende o fluxo de execução neste ponto.
- Quando o primeiro thread terminar, main prossegue.
- Se o segundo thread já tiver terminado, ele finaliza.
- Caso contrário suspende novamente a execução de main no ponto onde é chamado o segundo join().
- Quando ambos os threads secundários terminarem, main imprime a última linha e termina.

```
public class JoinExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(new SimpleDelay(2000));  
        t1.start();  
        Thread t2 = new Thread(new SimpleDelay(5000));  
        t2.start();  
  
        System.out.println("Waiting for " + t1.getName());  
        try {  
            t1.join();  
        } catch (InterruptedException e) { ... }  
        System.out.println("Waiting for " + t2.getName());  
        try {  
            t2.join();  
        } catch (InterruptedException e) { ... }  
        System.out.println("Thread main is DONE!");  
    }  
}
```



Esperando um thread terminar: join()

Exemplo implementado com join():

- O método join() é um mecanismo de baixo nível, pois lida diretamente com os threads envolvidos e a única informação que mantém sobre uma tarefa é se ela terminou ou não.
- Java oferece várias outras alternativas para controlar a execução de threads.
- Algumas também operam no nível dos threads, como *notificações* usando wait() e notify() que veremos a seguir.
- Mas há também abstrações de nível mais alto, como objetos Future, que guardam o resultado da tarefa e permitem construir mecanismos de callback, sincronizadores e mecanismos assíncronos não-bloqueantes que permitem concatenar tarefas, ativadas pela finalização das tarefas anteriores.

Implementação de Paralelismo e Concorrência

Atividades

Exercício 1

- . Adapte o exercício anterior reaproveitando o Runnable que recebe dois argumentos (um nome e um delay em milissegundos). Deve ser possível criar uma instância desta forma:

```
Runnable objeto = new ImplementacaoDeRunnable("Texto", 200);
```

para imprimir a palavra “Texto” com um delay de 200 milissegundos.

- a. Implemente run() de forma que imprima uma linha do texto recebido, seguido por um número sequencial, a cada 200 milissegundos, da forma:

```
Texto 1
```

```
Texto 2
```

```
Texto 3
```

- b. Escreva um método main para rodar o thread.
- c. Implemente um mecanismo de interrupção no método run(), que permita que o interrupt() chamado por outro thread finalize a execução. Use o main ou outro thread para interromper o thread depois de um determinado tempo.

Exercício 2 e 3

Crie uma classe executável e no seu método `main()` crie e inicie três threads distintos, usando instâncias diferentes da mesma classe criada no item anterior. Escolha textos diferentes e atrasos diferentes (variando entre 200 e 1000 milissegundos). Faça o thread principal dormir por três intervalos seguidos de 5 segundos, interrompendo, após cada intervalo, cada um dos threads criados.

Escreva um método que calcule a raiz quadrada (`Math.sqrt`) de todos os números ímpares de 1 a 99 e guarde os resultados em um `ArrayList` (`Collections.synchronizedList`). Imprima os resultados na tela esperando 50 milissegundos entre cada operação. Escreva outro método que calcule a raiz cúbica (`Math.cbrt`) de todos os números pares de 2 até 1000 e guarde os

Orientações Gerais

Atividades aula 02: `interrupt()`, `sleep()`, `join()`

- 1) Monte todos os códigos-exemplos explorados em aula e execute para fixar os conceitos abordados;
- 2) Faça os exercícios propostos e poste sua solução no GITHUB. Coloque seu nome e matricula nos comentários dos códigos desenvolvidos;
- 3) Os códigos devem ser postados em formato doc ou txt para facilitar o teste em qualquer IDE;
- 4) Você poderá desenvolver os exercícios propostos na IDE que julgar mais confortável;
- 5) Você precisa ter testado anteriormente os exemplos para poder aproveitar o código para o exercício 1.

▪ **Bom trabalho!!!**

↔ Code

🔔 Issues 0

🔗 Pull requests 0

▶ Actions

📁 Projects 0

📖 Wiki

🛡 Security

📊 Insights

⚙ Settings

Exemplos de codigos java

Edit

[Manage topics](#)

🕒 5 commits

🌿 1 branch

📦 0 packages

🏷 0 releases

👤 1 contributor

Branch: master ▾

New pull request

Create new file

Upload files

Find file

Clone or download ▾



elianasantos Add files via upload ...

Latest commit a193f94 now

📄 Exemplo Interrupt.docx	Add files via upload	12 hours ago
📄 Exemplo Join.docx	Add files via upload	40 minutes ago
📄 Exemplo Sleep.docx	Add files via upload	2 hours ago
📄 Exercicios aula 02.pdf	Add files via upload	now

Obrigada!

