

# Programação Orientada a Objetos 2 (POO2)

Aula 2 - Threads  
Leandro Nogueira Couto



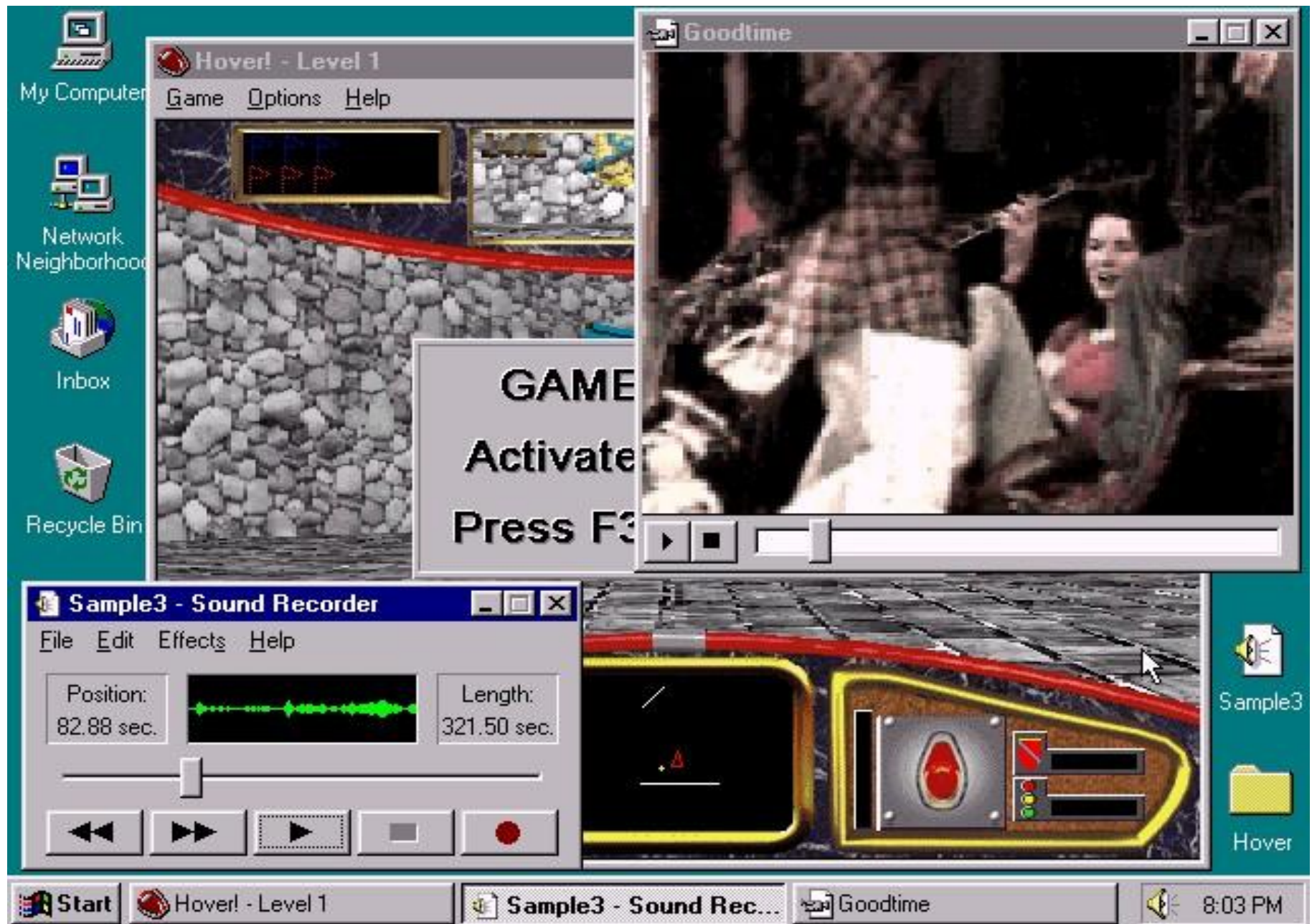
Faculdade de  
Computação



# Threads

- Hoje em dia nem percebemos que nossos computadores podem realizar mais de uma tarefa ao mesmo tempo
- Frequentemente uma mesma aplicação precisa ser capaz de fazer múltiplas tarefas simultaneamente
- Usar um processador de texto, fazer um download pelo browser, tocar música, imprimir um arquivo... já foi uma coisa impressionante um dia!

# Threads



# Threads

- Java suporta programação concorrente desde a versão 5.0 (estamos em qual versão?)
- Usando **threads**
- Outros nomes: contexto de execução, *lightweight process*
- Preciso de múltiplos cores? O que é paralelismo de verdade?

# Threads

Threads executam dentro do mesmo programa (o browser, o jogo, o editor de texto)

Exemplo: abas do Browser

Cada operação de ordenação executa “ao mesmo tempo” que as outras e de forma independente

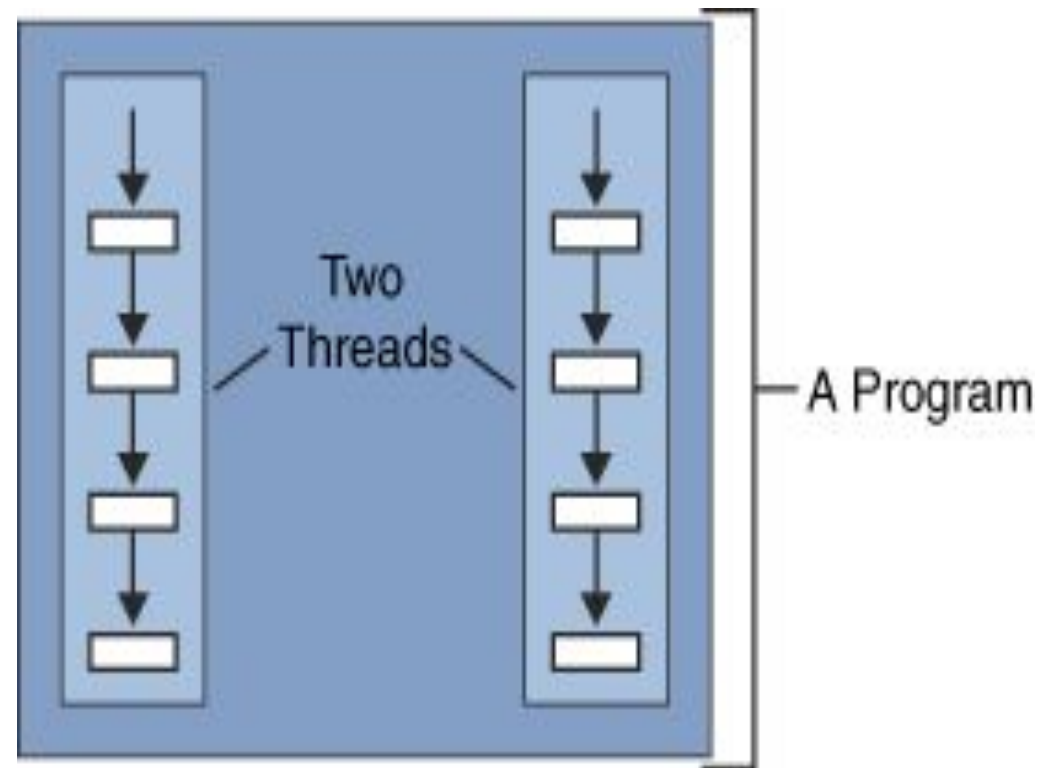
Como o SO lida com tudo isso? Escalona através de *time slicing*



# Threads

Definição: um thread é um fluxo único de controle sequencial dentro de um programa

A coisa fica mais interessante quando temos mais de um thread no mesmo programa (ver figura)



# Threads

Coisas que ocorrem ao mesmo tempo em um browser:

- scroll
- download de um arquivo
- tocar uma animação
- tocar um som
- imprimir uma página em background
- download de uma nova página
- execução de um applet
- alteração das configurações



# Threads

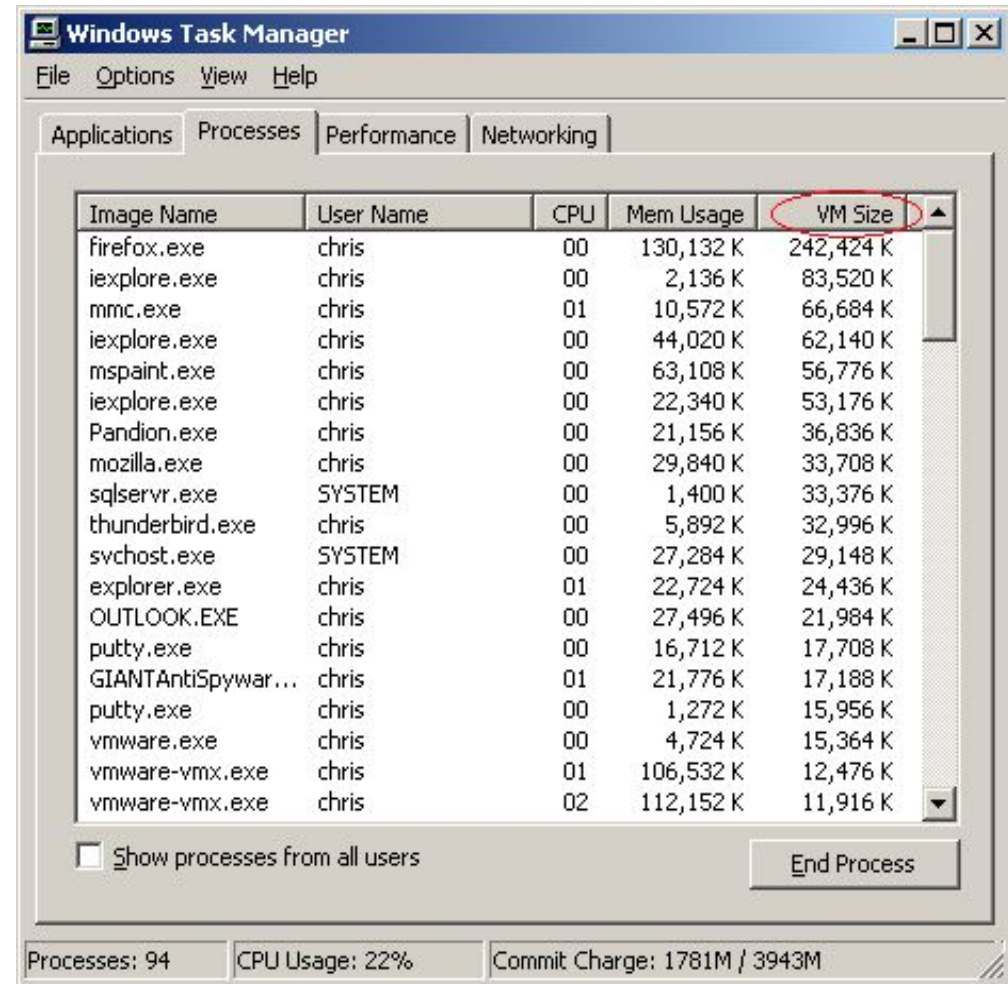
Uma thread parece um processo mas:

- Compartilha o mesmo "espaço de endereçamento"
  - É muito rápido chavear a execução entre threads mas não entre processos
- .Uma thread recebe alguns recursos próprios durante a execução
- Uma pilha de execução para poder chamar métodos, passar parâmetros, alocar variáveis locais
  - Um "Program Counter"
  - Chamamos isso o "contexto de execução do thread"
  - Alguns autores chamam thread de "contexto de execução"

# Threads

## .Processos vs Threads:

- Um processo tem um ambiente de execução auto-contido. Um processo geralmente possui um conjunto de recursos para executar.
- Em particular, cada processo tem seu próprio espaço de memória



Windows Task Manager

File Options View Help

Applications Processes Performance Networking

Image Name	User Name	CPU	Mem Usage	VM Size
firefox.exe	chris	00	130,132 K	242,424 K
iexplore.exe	chris	00	2,136 K	83,520 K
mmc.exe	chris	01	10,572 K	66,684 K
iexplore.exe	chris	00	44,020 K	62,140 K
mspaint.exe	chris	00	63,108 K	56,776 K
iexplore.exe	chris	00	22,340 K	53,176 K
Pandion.exe	chris	00	21,156 K	36,836 K
mozilla.exe	chris	00	29,840 K	33,708 K
sqlservr.exe	SYSTEM	00	1,400 K	33,376 K
thunderbird.exe	chris	00	5,892 K	32,996 K
svchost.exe	SYSTEM	00	27,284 K	29,148 K
explorer.exe	chris	01	22,724 K	24,436 K
OUTLOOK.EXE	chris	00	27,496 K	21,984 K
putty.exe	chris	00	16,712 K	17,708 K
GIANTAntiSpywar...	chris	01	21,776 K	17,188 K
putty.exe	chris	00	1,272 K	15,956 K
vmware.exe	chris	00	4,724 K	15,364 K
vmware-vmx.exe	chris	01	106,532 K	12,476 K
vmware-vmx.exe	chris	02	112,152 K	11,916 K

☐ Show processes from all users

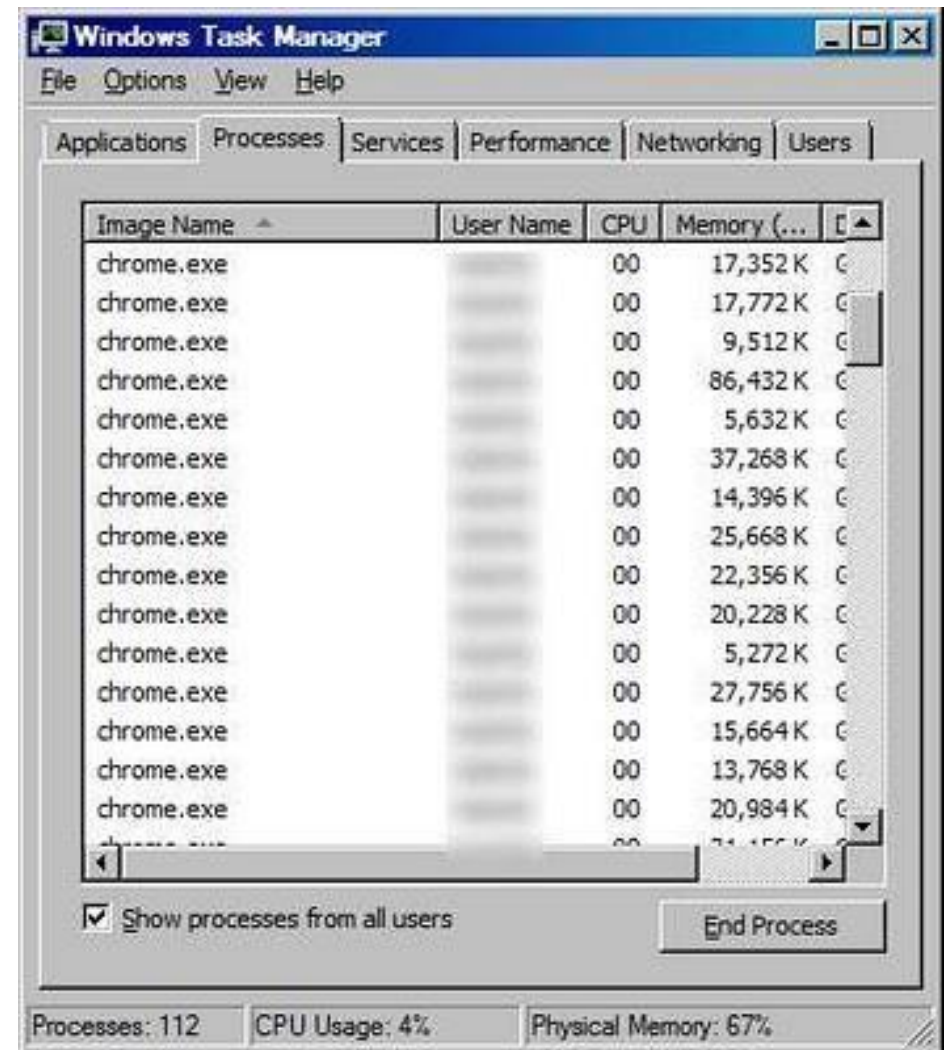
End Process

Processes: 94 CPU Usage: 22% Commit Charge: 1781M / 3943M

# Threads

## .Processos vs Threads:

- Muitas vezes vemos processos como programas ou aplicativos, mas pode ser que um programa tenha múltiplos processos.



# Threads

## .Processos vs Threads:

- Apesar de terem espaços de memória exclusivos, processos podem se comunicar por Inter Process Communications (IPC), uma feature do SO. A comunicação pode ocorrer até entre processos de diferentes sistemas (Sistemas Distribuídos)
- O Java permite a criação de processos com o Objeto `ProcessBuilder`, mas aqui vamos nos concentrar em threads

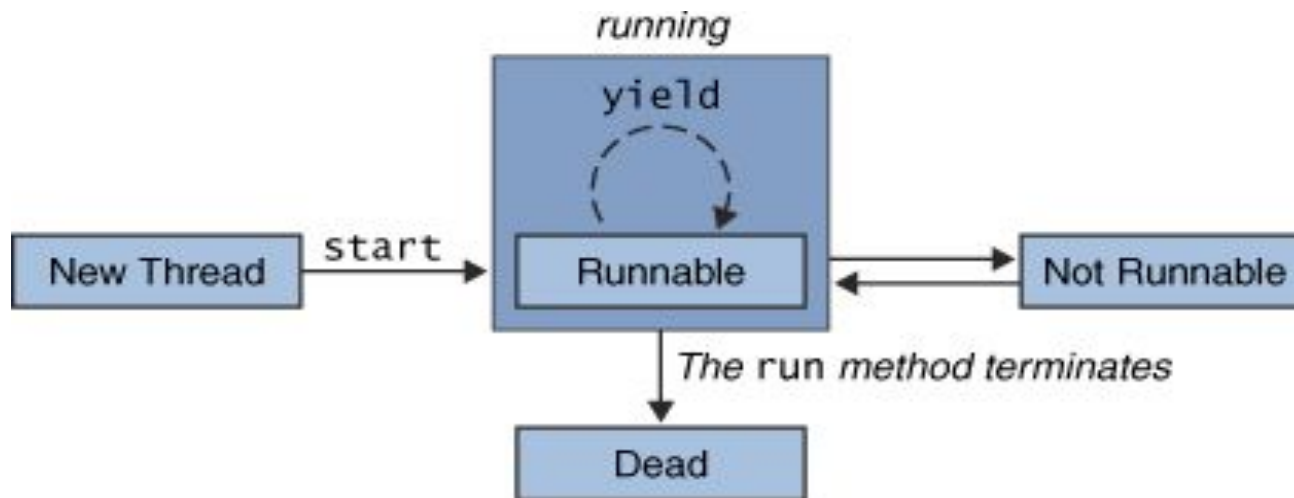
# Threads

## .Processos vs Threads:

- Todo processo possui ao menos uma thread
- Criar uma thread é menos custoso que criar um processo novo
- Threads compartilham entre si os recursos de um processo, como memória ou arquivos abertos. Isso torna a comunicação mais eficiente, mas potencialmente problemática
- A máquina virtual do Java tem mais threads executando além da **main thread** do programador. Há threads de sistema que fazem coisas como tratamento de entradas e gerenciamento de memória (e.g. Garbage Collector)

# Threads

- .Mais detalhes sobre o funcionamento de uma thread:
- .Diagrama de Estados Finitos:



# Threads

## Em Java:

Cada thread está associada a uma instância da class **Thread**

Uma vez que **Thread** é instanciada, é preciso prover o código que rodará naquela thread. Há duas formas de fazer isso:

- Prover um objeto **Runnable** com o método **run()**

- Criar uma subclasse de **Thread** que implementa o método **run()**

# Threads

• Objeto **Runnable**, A interface **Runnable** define um único método, **run()**, que deve conter o código executado na thread. Esse objeto é passado ao construtor da thread:

```
• public class HelloRunnable implements Runnable {  
  
    • public void run() {  
        • System.out.println("Hello World!");  
    }  
  
    • public static void main(String args[]) {  
        • (new Thread(new HelloRunnable())).start();  
    }  
• }
```



# Threads

Criar uma subclasse de **Thread**. A classe **Thread** implementa **Runnable**, mas seu método **run()** não faz nada. Temos que criar uma subclasse e introduzir (*override*) sua própria implementação de **run()**:

```
public class HelloThread extends Thread {  
  
    public void run() {  
        System.out.println("Hello from a thread!");  
    }  
  
    public static void main(String args[]) {  
        (new HelloThread()).start();  
    }  
}
```

# Threads

- .Note que para disparar a thread chamamos o método **Thread.start()**.
- .Qual das abordagens usar? Você consegue compará-las?

# Threads

.Note que para disparar a thread chamamos o método **Thread.start()**.

.Qual das abordagens usar?

.A primeira é boa pois permite que sua classe (que implementa **Runnable**) seja subclasse de outra classe que não **Thread**. Assim, é o método mais geral. Criar uma subclasse de **Thread** é mais simples, mas é limitante pois sua classe precisa necessariamente ser subclasse de **Thread** e você não pode usar herança pra outra coisa.

# Threads

## .Métodos úteis:

`Thread.sleep()`

- A thread dorme por alguns segundos, livrando tempo de processador para outras threads
- O tempo pode variar, pois não se garante a execução tão precisa (depende do SO) e o sleep pode ser interrompido por uma exception

# Threads

## .Métodos úteis:

`Thread.join()`

- Espera essa thread morrer
- A thread que chama vai esperar o término da thread chamada. Muito útil para evitar conflitos!

# Threads

## .Métodos úteis:

`Thread.interrupt()`

- Esse método, chamado para o objeto Thread da thread que se quer interromper, pára a execução da thread.
- Sucedeu métodos problemáticos que foram deprecados, como **stop()**, **suspend()** e **resume()**.
- Lembre-se, a interrupção está sendo chamada por outra thread!
- O que fazer para que a thread suporte sua própria interrupção?

# Threads

Por exemplo, se o código anterior estivesse no método **run()** do objeto **Runnable**, ele poderia esperar por uma `InterruptedException`:

# Threads

## .Métodos úteis:

`Thread.interrupted()`

- Checa se a thread for interrompida, se sim, a **flag de interrupt** é limpa
- `Thread.isInterrupted()`
- Checa se alguma outra thread foi interrompida. Não altera a **flag de interrupt** daquela thread
- `isAlive()`, `currentThread()`, `setPriority()`, `getPriority()`, `setName()`, `getName()` ...
- [Site da Oracle](#) tem as especificações e detalhes



# Threads

## .Métodos úteis:

`Thread.join()`

- Faz a thread atual (a que chama o método) **esperar** o fim da execução da outra (a que teve o método chamado), para "unir" novamente as duas
- Assim como sleep, é sensível a interrupções (InterruptedException)

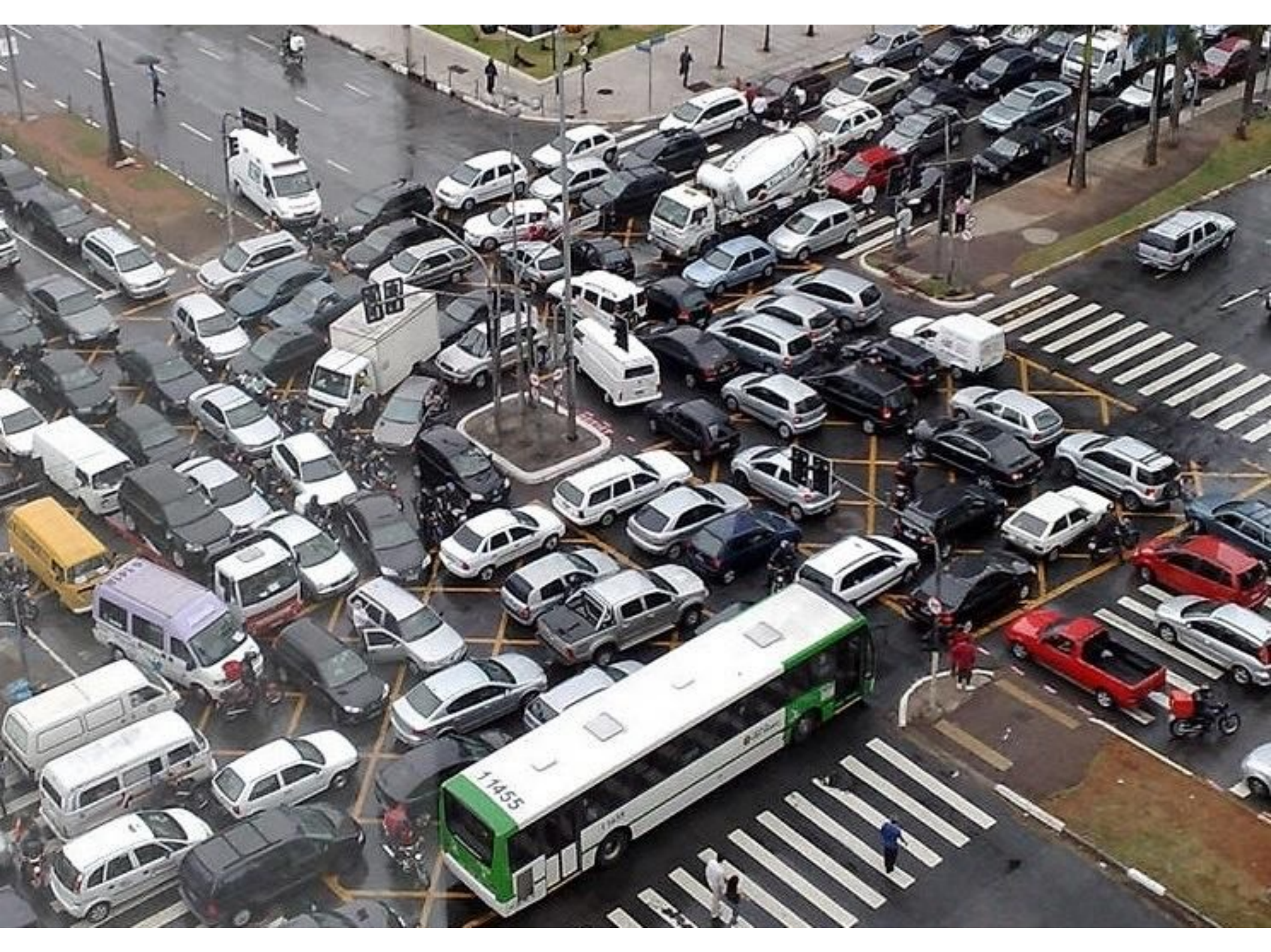
# Threads

Exemplo de código...

# Threads

- .Problemas com threads: **Interferência**
- .Pode ocorrer quando duas operações, executando sobre os mesmo dados, se sobrepõe, resultando em resultados inesperados. Vide exemplo abaixo
- .(É preciso atomicidade para evitar esse tipo de conflito)







# Threads

.Até operações que parecem atômicas podem se sobrepor. Elas podem ter subdivisões dentro da Java Virtual Machine.

Exemplo: c++;

Como funciona:

- Recupera o valor de c
- Incrementa o valor em 1
- Armazena o novo valor de volta em c

# Threads

- .Suponha agora que uma thread A invoque **c++** ao mesmo tempo que outra thread B invoca **c--**, e que elas se sobreponham da seguinte forma:
- .Thread A: Obtém c.
- .Thread B: Obtém c.
- .Thread A: Incrementa valor. Resultado é 1.
- .Thread B: Decrementa valor. Resultado é -1.
- .Thread A: Armazena valor 1.
- .Thread B: Armazena valor -1.
- .Se a variável c valia zero, quanto ela deveria valer? E quanto ela vale agora? Não é determinístico!

# Threads

- .Existem diversas formas de garantir sincronismo entre threads.
- .Métodos como `start()` e `join()` oferecem pontos de sincronismo.
- .Outra solução muito apropriada são os métodos **sincronizados**, ou **synchronized** methods, criados usando a keyword **synchronized**

# Threads

- .Chamadas para métodos sincronizados não podem se entrelaçar em threads diferentes. Se uma thread está executando um método sincronizado, qualquer outra chamada para métodos sincronizados do mesmo objeto é bloqueada.
- .Depois do término do método, garante-se que as mudanças feitas serão vistas por todas as threads ( o mesmo ocorre com start() e join()). Chamamos essa garantia de relação **happens-before**.



# Threads

- .Outro exemplo de problema possível quando se usa threads são **deadlocks e similares (starvation e livelock)**
- .Eles se referem a problemas envolvendo a distribuição de recursos e deve-se estar atento para evitá-los
- .Um **deadlock** ocorre quando uma thread espera pelo resultado da outra, e depende da outra para continuar. Elas esperam indefinidamente por um resultado que não virá jamais, e o programa congela

# Threads

.Objetos Friends fazem um handshake (método bow()) e esperam pelo bow() do outro objeto antes de continuar (bowback()). E se ambos invocarem bow() ao mesmo tempo?

# Threads

## **.Starvation**

- Uma thread pode ser negligenciada com pouco ou nenhum acesso ao processador. Imagine um método sincronizado longo ("egoísta") que ocupa muito tempo do processador; se isso é frequente, outras threads sincronizadas ficarão bloqueadas com frequência

## **.Livelock**

- Pode haver situações em que threads não estão bloqueadas, mas ainda assim não conseguem continuar. Thread A reage à thread B, que reage à thread A, que reage à thread B, num loop infinito. Semelhante a duas pessoas tentando passar uma porta ao mesmo tempo

# Threads

## .Executores

- Em projetos maiores, faz sentido separar a criação e gerenciamento de threads de sua execução
- Para isso podemos usar executors
- Ver documentação da Oracle para mais detalhes sobre o funcionamento; aqui daremos uma visão geral e conceitual

.Se **r** é um objeto **Runnable** e **e** é um objeto **executor**, podemos substituir

```
. (new Thread(r)).start();
```

```
.por
```

```
.e.execute(r);
```

# Threads

- .Como um executor lida com a criação da thread é decidido pela máquina virtual em tempo de execução.
- .É possível que nem mesmo uma nova classe seja criada, mas que a tarefa seja enfileirada num espaço vago de uma thread já existente!
- .Outras interfaces semelhantes são **ExecutorService** (admite objetos **Runnable** ou **Callable**, podendo retornar um valor) e **ScheduledExecutorService** (adiciona a possibilidade de passar um intervalo de delay, efetivamente "agendando" a execução)

# Threads

- .Como um executor lida com a criação da thread é decidido pela máquina virtual em tempo de execução.
- .É possível que nem mesmo uma nova classe seja criada, mas que a tarefa seja enfileirada num espaço vago de uma thread já existente!
- .Outras interfaces semelhantes são **ExecutorService** (admite objetos **Runnable** ou **Callable**, podendo retornar um valor) e **ScheduledExecutorService** (adiciona a possibilidade de passar um intervalo de delay, efetivamente "agendando" a execução, até mesmo periodicamente)
- .Outra opção é usar **Thread Pools**: criam um banco de threads que são alocadas automaticamente, livrando um pouco o *overhead* de criação de threads

# Threads

- .Com **Thread Pools** não é preciso se preocupar com o ciclo de vida das threads (criação, destruição, etc)
- .**ThreadPoolExecutor** ou **ScheduledThreadPoolExecutor**, implementam interface **ExecutorService**.
- .O programador pode especificar:
  - . O número básico e máximo do pool (número de threads)
  - . O tipo de estrutura de dados para armazenar as tarefas
  - . Como tratar tarefas rejeitadas
  - . Como criar e terminar threads