

THREADS EM JAVA

George Gomes Cabral

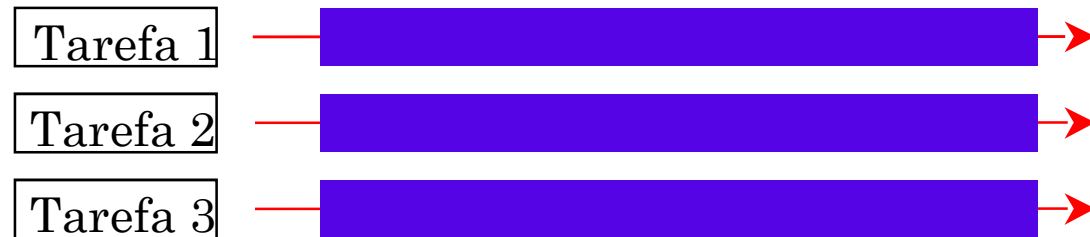
THREADS

- Fluxo seqüencial de controle dentro de um processo.
- Suporte a múltiplas linhas de execução permite que múltiplos processamentos ocorram em "paralelo" (em computadores com um processador, os threads não são executados em paralelo e, sim, concorrentemente).

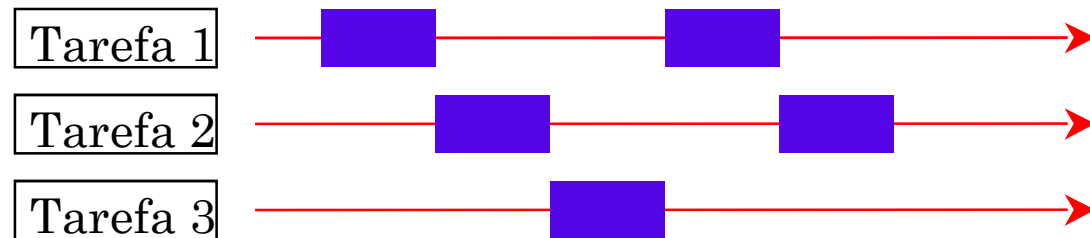


THREADS

Várias threads
em várias CPUs



Várias threads
compartilhando
uma única CPU



THREADS

- Queremos escrever programas que possam fazer várias tarefas simultaneamente.
 - baixar uma imagem pela rede.
 - requisitar um relatório atualizado do estoque.
 - rodar várias animações.
 - tudo ao mesmo tempo.
- Cada thread representa a execução de uma sequência de comandos independentes.



THREADS EM JAVA

- Suporte a multithreading faz parte da linguagem.
 - Todo programa é executado em pelo menos uma thread.

```
class UmaUnicaThread
{
    public static void main(String[] args)
    {
        // main() é executado em um única thread
        System.out.println(Thread.currentThread());
        for (int i=0; i<30; i++)
            System.out.println("i == " + i);
    }
}
```



EXEMPLO

```
class CountThread extends Thread
{ int from, to;
  public CountThread(int from, int to)
  { this.from = from; this.to = to; }
  public void run()
  { for (int i=from; i<to; i++)
    System.out.println("i == " + i);
  }
  public static void main(String[] args)
  { // Dispara 5 threads, cada uma irá contar 10 vezes
    for (int i=0; i<5; i++)
    { CountThread t = new CountThread(i*10, (i+1)*10);
      t.start();
    }
  }
}
```

O método run() é como se fosse um main(), só que não é estático

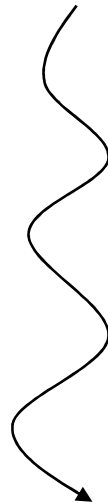


EXEMPLO

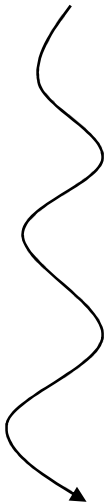
main



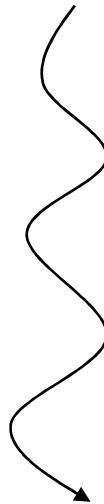
0..9



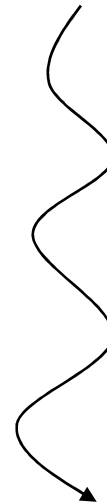
10..19



20..29



30..39



40..49



THREADS - API

○ Thread()

- Classe Thread possibilita a criação de um objeto executável.

○ void run()

- invocado pelo sistema de execução da JVM.
- deve-se sobrescrever este método para prover o código a ser executado pela thread.

○ void start()

- inicia a thread e provoca a chamada de run().

○ void stop()

- para a execução da thread.

○ void suspend()

- suspende a execução da thread.

○ void resume()

- retoma a execução interrompida por um suspend().



THREADS - API

○ **static void `sleep`(long millis)**
throws

`InterruptedException`

- põe a thread para dormir por um tempo em milissegundos.

○ **void `interrupt`()**

- interrompe a thread em execução.

○ **static boolean `interrupted`()**

- testa se a thread foi interrompida.



THREADS - API

- **boolean** `isAlive()`

- testa se a thread está rodando.

- **void** `setPriority(int p)`

- define a prioridade de execução da thread.

- **void** `yield()`

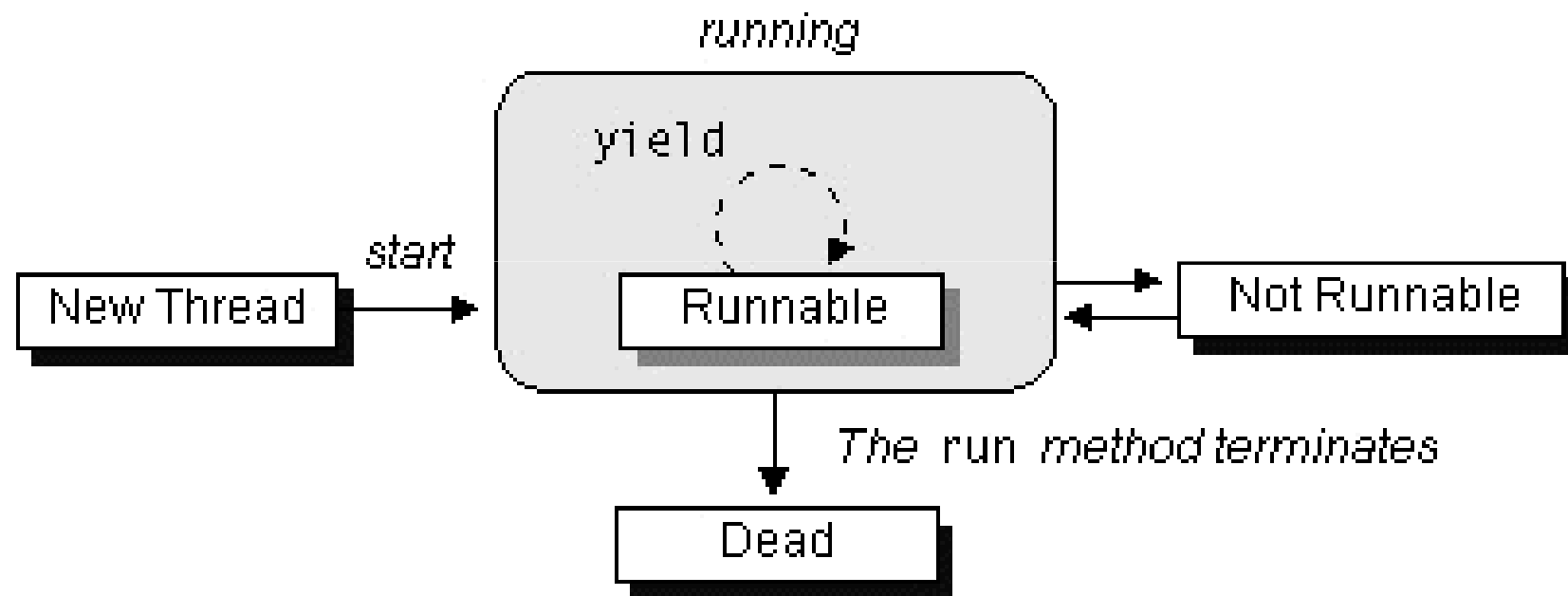
- indica ao gerenciador de threads que esta é uma boa hora para rodar outras threads.

- **void** `join()`

- espera que uma thread termine sua execução.



ESTADOS DE UMA THREAD



ESTADOS DE UMA THREAD

○ New

- Uma nova thread é um objeto thread vazio; nenhum recurso do sistema foi alocado ainda. Chamar qualquer método além de *start()* resulta em uma *IllegalThreadStateException*.

○ Runnable

- Uma thread entra no estado **Runnable** depois da invocação do método *start()*. O método *start()* aloca recursos do sistema e chama o método *run()* da thread.

○ Not Runnable

- Uma thread se encontra no estado **Not Runnable** quando... (continua)



ESTADOS DE UMA THREAD

- O método *sleep()* é invocado
- O método *wait()* é invocado
- A thread está esperando uma operação de entrada e saída.
- A thread se torna **Runnable** novamente quando...
 - O número de milissegundos do método *sleep()* se passou
 - Quando a condição por que ele está esperando mudou e ele recebe uma mensagem *notify()* ou *notifyAll()*
 - Quando a operação de I/O se completa
- **Dead**
 - O método *run* se completa
 - O método *destroy* é executado



THREADS DE USUÁRIO X DEAMONS

- Se estiver associada a outra thread, terminará juntamente com ela.
 - `minhaThread.setDaemon(true)`.
- Uma thread do tipo daemon roda sempre como um “pano de fundo” da thread que a criou.
 - Uma thread daemon termina quando a thread que a criou também terminar.
- Uma thread que não é uma daemon é chamada de thread de usuário.



THREADS DE USUÁRIO X DEAMONS

```
...main(...)...
```

```
thread1.setDaemon(true);
```

```
thread1.start();
```

```
thread2.setDaemon(true);
```

```
thread2.start();
```

```
thread3.start();
```

```
return;
```

Serão automaticamente finalizadas quando o método `main()` for finalizado

Pode continuar a execução mesmo após o final de `main()`



UMA CLASSE “EXECUTÁVEL”

- As vezes pode não ser conveniente criar uma subclasse de *Thread*.
 - também não será possível devido ao mecanismo de herança implementado em Java.
- A interface *Runnable* adiciona o método `run()` sem herdar nada de *Thread*.

```
public interface Runnable  
{ public void run( ); }
```



EXEMPLO

```
class CountThreadRun implements Runnable {  
    int from, to;  
  
    public CountThreadRun(int from, int to) {  
        this.from = from;  
        this.to = to;  
    }  
  
    public void run() {  
        for (int i = from; i < to; i++)  
            System.out.println("i == " + i);  
    }  
}
```



EXEMPLO

- Para executar como uma thread:

```
// Cria uma instância de Runnable  
Runnable r = new CountThreadRun(10,20);  
// Cria uma instância de Thread  
Thread t = new Thread(r);  
// inicia a execução da thread  
t.start();
```



EXEMPLO


```
/* Este programa cria uma classe chamada MultiplaImpressao*/  
public class MultiplaImpressao implements Runnable {  
    Thread threadDaClasse;  
    String string;  
    int contador;  
    int tempoDormindo;  
  
    /* método construtor da classe */  
    public MultiplaImpressao(String s, int quantasVezes, int  
        dormir) {  
        contador = quantasVezes;  
        string = s;  
        tempoDormindo = dormir;  
        threadDaClasse = new Thread(this);  
        threadDaClasse.start();  
    } /* Fim do metodo construtor */
```



EXEMPLO

```
public void run() {
    while (contador > 0) {
        System.out.println(string + " - "+ contador);
        try {
            Thread.sleep(tempoDormindo);
        } catch (Exception e) {}
        contador--;
    } /* fim-while */
} /* Fim do metodo run */

public static void main(String args[]) {
    new MultiplaImpressao("ping", 5, 300);
    new MultiplaImpressao("pong", 5, 500);
} /* Fim do metodo main */
} /* Fim da classe MultiplaImpressaoThread */
```



CONSTRUTORES

```
public Thread( );  
public Thread(Runnable target);  
public Thread(String name);  
public Thread(Runnable target, String name);  
public Thread(ThreadGroup group,  
                Runnable target);  
public Thread(ThreadGroup group, String name);  
public Thread(ThreadGroup group,  
                Runnable target, String name);
```



PRIORIDADE

- Cada *thread* apresenta uma prioridade de execução.
 - pode ser alterada com `setPriority(int p)`
 - pode ser lida com `getPriority()`
- Algumas constantes incluem:
 - `Thread.MIN_PRIORITY`
 - `Thread.MAX_PRIORITY`
 - `Thread.NORM_PRIORITY`
 - o padrão é `Thread.NORM_PRIORITY`



AGRUPANDO THREADS

- Pode-se operar as *threads* como um grupo.
 - pode-se parar ou suspender todas as threads de uma só vez.
 - `group.stop();`

```
ThreadGroup g =  
    new ThreadGroup( "um grupo de  
    threads" );
```



AGRUPANDO THREADS

- Inclua uma thread em um grupo através do construtor da thread.

```
Thread t = new Thread(g,new ThreadClass(), "Esta thread");
```



AGRUPANDO THREADS

- Para saber quantas *threads* em um grupo estão sendo executadas no momento, podemos usar o método `activeCount()`:

```
System.out.println("O número de threads "+  
    " que podem estar rodando é "+  
    g.activeCount());
```



SINCRONIZAÇÃO

- O que acontece caso tenhamos duas *threads* que irão executar o mesmo código a seguir?
 - Ambas as *threads* estão acessando o mesmo objeto.

```
int cnt = Contador.incr( ) ;
```



SINCRONIZAÇÃO - SITUAÇÃO 1

Thread 1	Thread 2	count
cnt = Contador.incr();	---	0
n = conta;	---	0
conta = n + 1;	---	1
return n;	---	1
---	cnt = Contador.incr();	1
---	n = conta;	1
---	conta = n + 1;	2
---	return n;	2



SINCRONIZAÇÃO – SITUAÇÃO 2

Thread 1	Thread 2	count
cnt = Contador.incr();	---	0
n = conta;	---	0
---	cnt = Contador.incr();	0
---	n = conta;	0
---	conta = n + 1;	1
---	return n;	1
conta = n + 1;	---	1
return n;	---	1



MONITORES

```
public class Contador2
{ private int conta = 0;
  public synchronized int incr()
  { int n = conta;
    conta = n + 1;
    return n;
  }
}
```

Não permite que este método seja executado por mais de uma thread ao mesmo tempo



MONITORES

- Também podemos usar monitores para partes de um método.

synchronized(oObjeto)

sentença; *// Sincronizada em relação a oObjeto*

Indique o recurso
a ser monitorado



MONITORES

```
void metodo(UmaClasse obj)
{ synchronized(obj)
  { obj.variavel = 5;
  }
}
```

Ninguém poderá usar este
objeto enquanto estivermos
executando estas operações

