Threads (Programação Concorrente)

Em várias situações, precisamos "rodar duas coisas ao mesmo tempo". Imagine um programa que gera um relatório muito grande em PDF. É um processo demorado e, para dar alguma satisfação para o usuário, queremos mostrar uma barra de progresso. Queremos então gerar o PDF e ao mesmo tempo atualizar a barrinha.

Pensando um pouco mais amplamente, quando usamos o computador também fazemos várias coisas simultaneamente: queremos navegar na internet e ao mesmo tempo ouvir música.

A necessidade de se fazer várias coisas simultaneamente, ao mesmo tempo, paralelamente, aparece frequentemente na computação. Para vários programas distintos, normalmente o próprio sistema operacional gerencia isso através de vários processos em paralelo.

Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de Threads.

Em Java, usamos a classe Thread do pacote java.lang para criarmos linhas de execução paralelas. A classe Thread recebe como argumento um objeto com o código que desejamos rodar. Por exemplo, no programa de PDF e barra de progresso:

```
public class GeraPDF {
  public void rodar () {
    // lógica para gerar o pdf...
  }
}

public class BarraDeProgresso {
  public void rodar () {
    // mostra barra de progresso e vai atualizando ela...
  }
}
```

E, no método main, criamos os objetos e passamos para a classe Thread. O método start é responsável por iniciar a execução da Thread:

```
public class MeuPrograma {
  public static void main (String[] args) {

    GeraPDF gerapdf = new GeraPDF();
    Thread threadDoPdf = new Thread(gerapdf);
    threadDoPdf.start();

    BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
    Thread threadDaBarra = new Thread(barraDeProgresso);
    threadDaBarra.start();

  }
}
```

O código acima, porém, não compilará. Como a classe Thread sabe que deve chamar o método roda? Como ela sabe que nome de método daremos e que ela deve chamar esse método especial? Falta na verdade um contrato entre as nossas classes a serem executadas e a classe Thread.

Esse contrato existe e é feito pela interface Runnable: devemos dizer que nossa classe é "executável" e que segue esse contrato. Na interface Runnable, há apenas um método chamado run. Basta implementá-lo, "assinar" o contrato e a classe Thread já saberá executar nossa classe.

```
public class GeraPDF implements Runnable {
  public void run () {
    // lógica para gerar o pdf...
  }
}

public class BarraDeProgresso implements Runnable {
  public void run () {
    // mostra barra de progresso e vai atualizando ela...
  }
}
```

A classe Thread recebe no construtor um objeto que é um Runnable, e seu método start chama o método run da nossa classe. Repare que a classe Thread não sabe qual é o tipo específico da nossa classe; para ela, basta saber que a classe segue o contrato estabelecido e possui o método run.

É o bom uso de interfaces, contratos e polimorfismo na prática!

Estendendo a classe Thread

A classe Thread implementa Runnable. Então, você pode criar uma subclasse dela e reescrever o run que, na classe Thread, não faz nada:

```
public class GeraPDF extends Thread {
  public void run () {
    // ...
  }
}
```

E, como nossa classe é uma Thread, podemos usar o start diretamente:

```
GeraPDF gera = new GeraPDF();
gera.start();
```

Apesar de ser um código mais simples, você está usando herança apenas por "preguiça" (herdamos um monte de métodos mas usamos apenas o run), e não por polimorfismo, que seria a grande vantagem. Prefira implementar Runnable a herdar de Thread.

Dormindo

Para que a thread atual durma basta chamar o método a seguir, por exemplo, para dormir 3 segundos:

```
Thread.sleep(3 * 1000);
```

Escalonador e trocas de contexto

Veja a classe a seguir:

```
public class Programa implements Runnable {

   private int id;
   // colocar getter e setter pro atributo id

   public void run () {
     for (int i = 0; i < 10000; i++) {
         System.out.println("Programa " + id + " valor: " + i);
     }
   }
 }
```

É uma classe que implementa Runnable e, no método run, apenas imprime dez mil números. Vamos usá-las duas vezes para criar duas threads e imprimir os números duas vezes simultaneamente:

```
public class Teste {
   public static void main(String[] args) {

     Programa p1 = new Programa();
     p1.setId(1);

     Thread t1 = new Thread(p1);
     t1.start();

     Programa p2 = new Programa();
     p2.setId(2);

     Thread t2 = new Thread(p2);
     t2.start();

   }
 }
```

Se rodarmos esse programa, qual será a saída? De um a mil e depois de um a mil? Provavelmente não, senão seria sequencial. Ele imprimirá 0 de t1, 0 de t2, 1 de t1, 1 de t2, 2 de t1, 2 de t2 e etc? Exatamente intercalado?

Na verdade, não sabemos exatamente qual é a saída. Rode o programa várias vezes e observe: em cada execução a saída é um pouco diferente.

O problema é que no computador existe apenas um processador capaz de executar coisas. E quando queremos executar várias coisas ao mesmo tempo, e o processador só consegue fazer uma coisa de cada vez? Entra em cena o escalonador de threads.

O escalonador (scheduler), sabendo que apenas uma coisa pode ser executada de cada vez, pega todas as threads que precisam ser executadas e faz o processador ficar alternando a execução de

cada uma delas. A ideia é executar um pouco de cada thread e fazer essa troca tão rapidamente que a impressão que fica é que as coisas estão sendo feitas ao mesmo tempo.

O escalonador é responsável por escolher qual a próxima thread a ser executada e fazer a troca de contexto (context switch). Ele primeiro salva o estado da execução da thread atual para depois poder retomar a execução da mesma. Aí ele restaura o estado da thread que vai ser executada e faz o processador continuar a execução desta. Depois de um certo tempo, esta thread é tirada do processador, seu estado (o contexto) é salvo e outra thread é colocada em execução. A troca de contexto é justamente as operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.

Quando fazer a troca de contexto, por quanto tempo a thread vai rodar e qual vai ser a próxima thread a ser executada, são escolhas do escalonador. Nós não controlamos essas escolhas (embora possamos dar "dicas" ao escalonador). Por isso que nunca sabemos ao certo a ordem em que programas paralelos são executados.

Você pode pensar que é ruim não saber a ordem. Mas perceba que se a ordem importa para você, se é importante que determinada coisa seja feita antes de outra, então não estamos falando de execuções paralelas, mas sim de um programa sequencial normal (onde uma coisa é feita depois da outra, em uma sequência).

Todo esse processo é feito automaticamente pelo escalonador do Java (e, mais amplamente, pelo escalonador do sistema operacional). Para nós, programadores das threads, é como se as coisas estivessem sendo executadas ao mesmo tempo.

E em mais de um processador?

A VM do Java e a maioria dos SOs modernos consegue fazer proveito de sistemas com vários processadores ou multi-core. A diferença é que agora temos mais de um processador executando coisas e teremos, sim, execuções verdadeiramente paralelas.

Mas o número de processos no SO e o número de Threads paralelas costumam ser tão grandes que, mesmo com vários processadores, temos as trocas de contexto. A diferença é que o escalonador tem dois ou mais processadores para executar suas threads. Mas dificilmente terá uma máquina com mais processadores que threads paralelas executando.

Exercícios

Teste o exemplo deste capítulo para imprimir números em paralelo.
Escreva a classe Programa:

```
public class Programa implements Runnable {

   private int id;
   // colocar getter e setter pro atributo id

   public void run () {
     for (int i = 0; i < 10000; i++) {
       System.out.println("Programa " + id + " valor: " + i);
     }
```

```
  }
}
```

Escreva a classe de Teste:

```
public class Teste {
  public static void main(String[] args) {

    Programa p1 = new Programa();
    p1.setId(1);

    Thread t1 = new Thread(p1);
    t1.start();

    Programa p2 = new Programa();
    p2.setId(2);

    Thread t2 = new Thread(p2);
    t2.start();

  }
}
```

Rode várias vezes a classe Teste e observe os diferentes resultados em cada execução. O que muda?

E as classes anônimas?

É comum aparecer uma classe anônima junto com uma thread. Vimos como usá-la com o Comparator. Vamos ver como usar em um Runnable.

Considere um Runnable simples, que apenas manda imprimir algo na saída padrão:

```
public class Programa1 implements Runnable {
   public void run () {
     for (int i = 0; i < 10000; i++) {
       System.out.println("Programa 1 valor: " + i);
     }
   }
}
```

No seu main, você faz:

```
Runnable r = new Programa1();
Thread t = new Thread(r);
t.start();
```

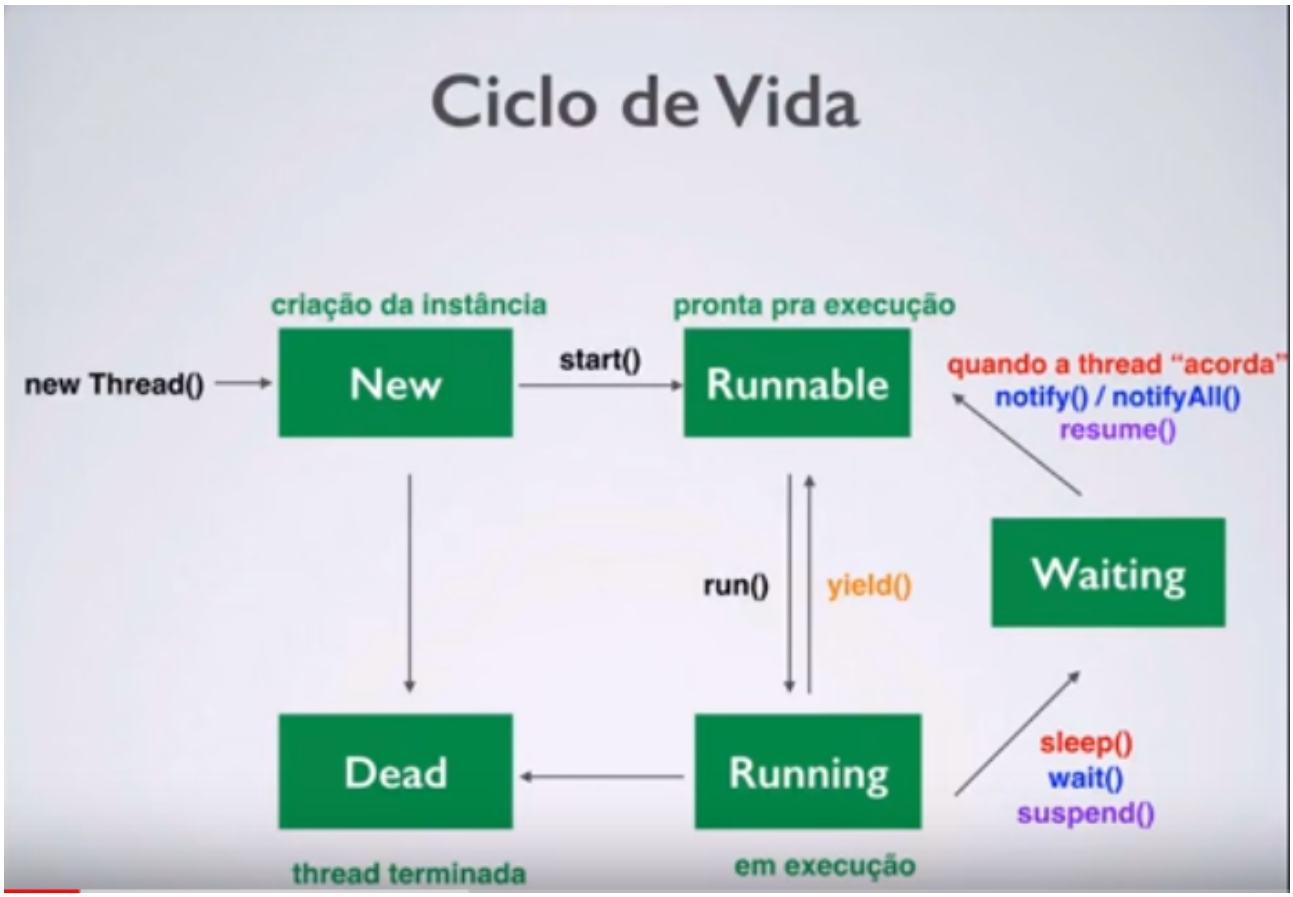
Em vez de criar essa classe Programa1, podemos utilizar o recurso de classe anônima. Ela nos permite dar new numa interface, desde que implementemos seus métodos. Com isso, podemos colocar diretamente no main:

```
Runnable r = new Runnable() {
  public void run() {
    for(int i = 0; i < 10000; i++)
      System.out.println("programa 1 valor " + i);
  }
};
Thread t = new Thread(r);
t.start();
```

Limitações das classes anônimas

O uso de classes anônimas tem limitações. Não podemos declarar um construtor. Como estamos instanciando uma interface, então não conseguimos passar um parâmetro para ela. Como então passar o id como argumento? Você pode, de dentro de uma classe anônima, acessar atributos da classe dentro da qual foi declarada! Também pode acessar as variáveis locais do método, desde que eles sejam final.

Ciclo de Vida de uma Thread



- o método start() deixa a Thread pronta para a execução

-o método run() executa a Thread
- quando está executando a Thread pode terminar indo para o estado Dead (morta)
- ou podemos colocar a Thread para dormir, esperar ou suspender. Quando fizermos uma dessas três ações colocamos a Thread em estado de espera (Waiting)
- quando a Thread está em estado de espera podemos acordá-la (notificar – notify()) para que elas saiam do estado de espera ou podemos resumir (resume()) a sua execução e ela voltar para o estado "pronta para a execução" e depois podemos executá-la novamente
- ainda quando a Thread está em execução podemos fazer ela voltar para o estado "pronta para execução" através do método yield()

Exercício 1 (resolvido): Criar uma clase chamada minha MinhaThread que é uma classe que pode ser executada por uma Thread, que tem dois atributos um nome e um tempo, tem um método construtor que recebem uma string e um inteiro e atribui para atributos nome e tempo, e um método que conta até 6 e escreve na tela "nome da Thread + contador + nro contado"e a cada contagem espera um certo tempo de acordo com o atributo tempo da classe e ainda escreva também na tela "(nome da thread) terminou a execução" após terminar a contagem.
Crie uma aplicação que crie três objetos da classe MinhaThread e os execute cada um em uma thread diferente.

Código da Classe Minha Thread

```
public class MinhaThread implements Runnable {
   private String nome;
   private int tempo;

   public MinhaThread(String nom, int tem){
      this.nome = nom;
      this.tempo = tem;
      Thread t = new Thread(this);
      t.start();
   }

   public void run() {
     try{
      for (int i=0;i<6;i++){
          System.out.println(nome+"contador" +i);
          Thread.sleep(tempo);
      }
     }catch (InterruptedException e){
        e.printStackTrace();
     }
     System.out.println(nome + "terminou a execução");
   }
}
```

Código do método main()

```
      MinhaThread mt1 = new MinhaThread("#1", 500);
      MinhaThread mt2 = new MinhaThread("#2", 1000);
      MinhaThread mt3 = new MinhaThread("#3", 700);
```

Exercicio 2: Escreva após a execução de todas as Threads “Programa Finalizado” no método main() - após t3.start(); escreva o seguinte código:

```
while (t1.isAlive() || t2.isAlive() || t3.isAlive(){
    try {
            Thread.sleep(200);
    }  catch (InterruptedException e) {
             e.printStackTrace();
    }
}
System.out.println(“Programa Finalizado”);
}
```

Desta forma o programa vai esperar terminar a execução das três threads para após seguir a execução do programa escrevendo “Programa Finalizado”.

Existe ainda uma forma mais prática de fazer isso que fizemos acima, que é usando o método join() da classe Thread, para isso acontecer substituimos o código anterior pelo código seguinte:

Se quisermos que ele execute toda a thread t1 primeiro e somente depois inicie a thread t2 colocamos entre t1.start() e t2.start() o seguinte código:

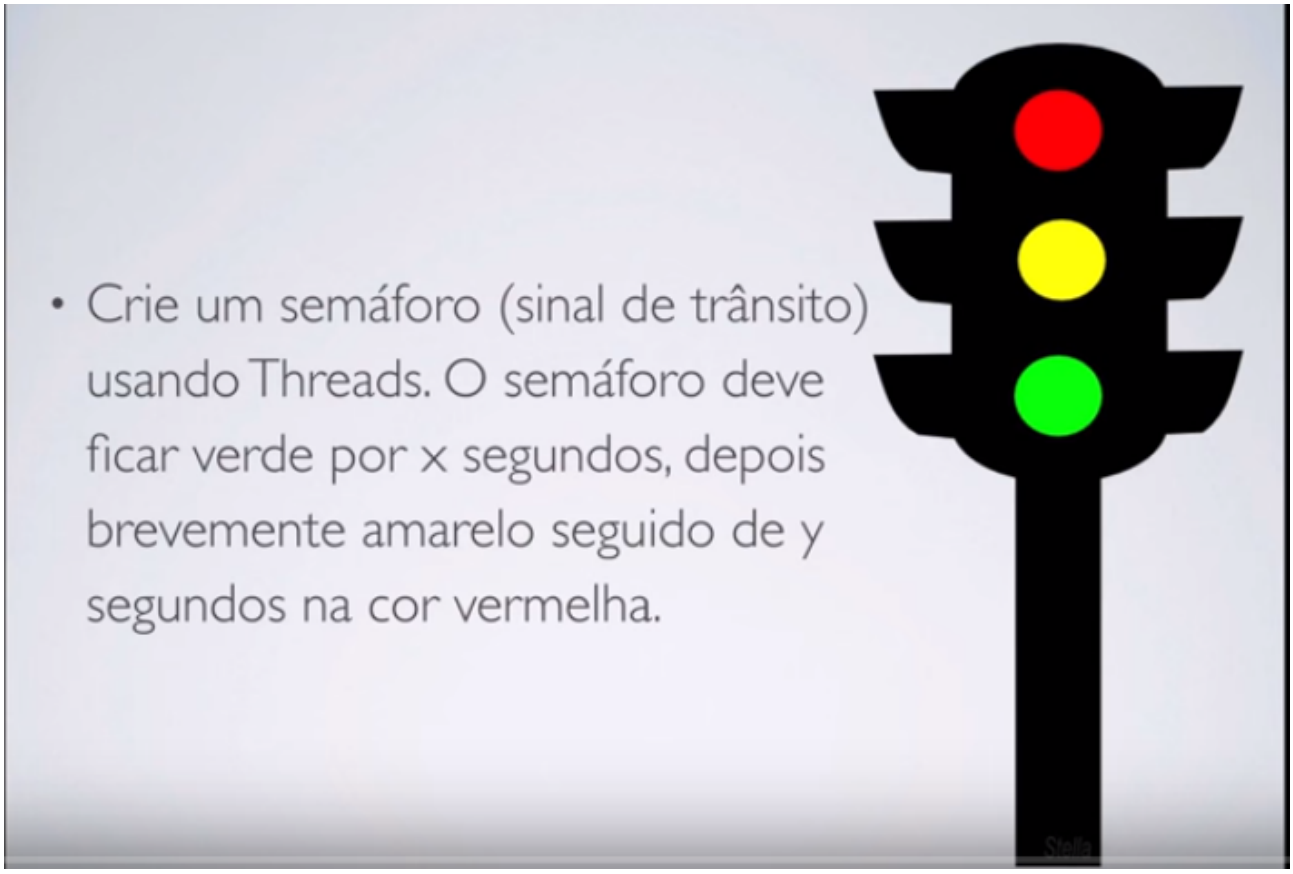
```
try{
    t1.join(200);     //a thread 2 só vai executar após a t1 terminar ou passar o tempo 200
                      //milisegundos. O que acontecer primeiro.
} catch (InterruptedException e){
    e.printStackTrace();
}
```

Ou para fazermos imprimir na tela “programa finalizado” somente após a execução das três threads usamos o seguinte código:

```
try{
    t1.join();
    t2.join();
    t3.join();
} catch (InterruptedException e){
    e.printStackTrace();
}
```

E se mudarmos o tempo para 500, 700, 800 mesmo assim ele executará todas as threads e após mostrará a mensagem.

Exercício 3:




```
public class ThreadSemaforo implements Runnable {
     private CorSemaforo cor;
     private boolean parar;
     private boolean corMudou;


public ThreadSemaforo() {
    this.cor = CorSemaforo.VERMELHO;
    this.parar = false;
    this.corMudou = false;
    new Thread(this).start();
}

    @Override
    public void run() {
        while (!parar){
            try {
                Thread.sleep(this.cor.getTempoEspera());
            this.mudarCor();
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }
```

```
    private synchronized void mudarCor(){
        switch (this.getCor()){
            case VERMELHO:
                this.cor = CorSemaforo.VERDE;
                break;
            case AMARELO:
                this.cor = CorSemaforo.VERMELHO;
                break;
            case VERDE:
                this.cor = CorSemaforo.AMARELO;
                break;
        }
        this.corMudou = true;
        notify();
    }

    public synchronized void esperaCorMudar(){
         while (!this.corMudou){
             try {
                 wait();
             } catch (InterruptedException e) {
                  e.printStackTrace();
             }
         }
         this.corMudou = false;
    }

    public synchronized void desligarSemaforo(){
        this.parar = true;
    }

    public CorSemaforo getCor() {
        return cor;
    }
}
```

```
public enum CorSemaforo {
     VERDE(1000), VERMELHO(300), AMARELO(1000);
     private int tempoEspera;

     CorSemaforo(int tempoEspera) {
         this.tempoEspera = tempoEspera;
     }

    public int getTempoEspera() {
        return tempoEspera;
    }
}
```

```
public static void main(String[] args) {
        ThreadSemaforo semaforo = new ThreadSemaforo();
        for (int i=0; i< 10;i++){
            System.out.println(semaforo.getCor());
            semaforo.esperaCorMudar();
        }
        semaforo.desligarSemaforo();
    }
```