



Java Standard Edition (JSE)

14.Threads



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com



Agenda

- Conceitos de programação concorrente
- Estendendo Thread;
- Troca de contextos;
- Garbage Collector;
- Problemas de concorrência;
- Sincronização;
- Monitores de concorrência;



Processamento paralelo

- Há situações em que precisamos executar duas coisas ao mesmo tempo;
- Podemos, por exemplo, mostrar uma barra de progresso enquanto esperamos a montagem de um relatório
- Em java, quando falamos de processamento paralelo, falamos de **Threads**;



java.lang.Thread

- Essa classe permite a criação de **linhas execução** paralelas;
- Como fica o paralelismo quando temos apenas um processador?
- A classe **Thread** recebe como argumento um objeto com o código a ser executado;
- Esse objeto deve ser um **Runnable**;



java.lang.Runnable

- Interface que define apenas o método:
 - `public abstract void run() ;`
- O método `Runnable.run()` é utilizado quando do início de uma Thread;
- O método `Thread.start()` determina o início da execução de uma determinada thread;



Exercício 01

- Crie a classe **GeradorRelatorio**, que contem o código a ser executado por uma thread:

```
//Implementação de Runnable que poderá
//ser executada por uma Thread
class GeradorRelatorio implements Runnable{
    //Método executado por Thread.start()
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println("Linha: "+i);
        }
    }
}
```



Exercício 02

- Implemente o código para Criação e execução de uma **Thread**, usando um objeto **GeradorRelatorio**:

```
public static void main(String[] args) {  
    System.out.println("Início da aplicação");  
    1 //Criação do objeto executável  
      GeradorRelatorio relatorio = new GeradorRelatorio();  
  
    2 //Criação da Thread  
      Thread executor = new Thread(relatorio);  
  
    3 //Início da execução da Thread  
      executor.start();  
}
```



Entendendo os códigos

- No exercício 01, foi criada a classe GeradorRelatorio, que implementa Runnable;
- No exercício 02, criamos uma Thread que, ao iniciar(`thread.start()`), invoca o método `GeradorRelatorio.run()`;
- Qual o resultado gerado pelo próximo exercício?



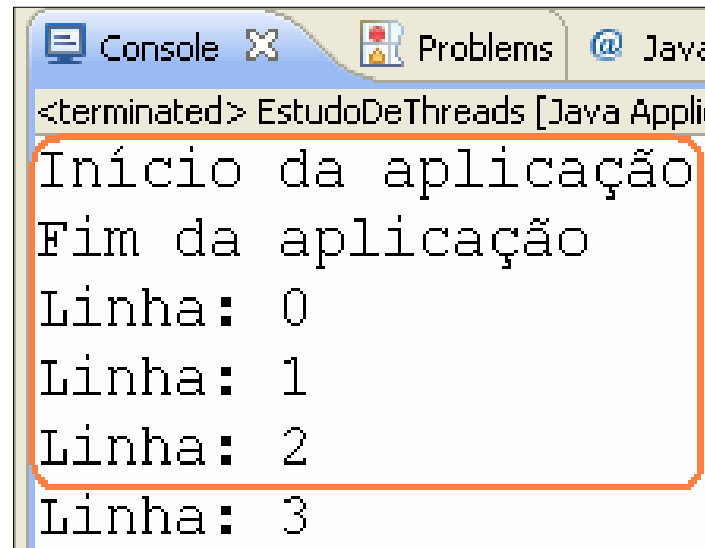
Exercício 03

- Altere o código de execução de thread, desenvolvido no exercício anterior, incluindo a mensagem final:

```
public static void main(String[] args) {  
    System.out.println("Início da aplicação");  
    //Criação do objeto executável  
    GeradorRelatorio relatorio = new GeradorRelatorio();  
    //Criação da Thread  
    Thread executor = new Thread(relatorio);  
    //Início da execução da Thread  
    executor.start();  
    System.out.println("Fim da aplicação");  
}
```

Resultado da execução

- Por que a mensagem final aparece antes da execução da Thread?
- **Thread** == novo processo independente



```
<terminated> EstudoDeThreads [Java Appli
Início da aplicação
Fim da aplicação
Linha: 0
Linha: 1
Linha: 2
Linha: 3
```



Execução de várias Threads

- Vimos que as Threads agem como processos independentes;
- Podemos solicitar a execução de vários processos ao mesmo tempo - **Multithread**;
- Vamos criar uma Thread para “exibir” uma mensagem, enquanto o relatório é processado;



Exercício 04

- Crie a classe **BarraDeProgresso**, que será executada por uma Thread, durante a impressão do relatório:

```
//Runnable para barra de progresso
class BarraDeProgresso implements Runnable{
    //Método de "exibição" da barra
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println("== Barra: "+i);
        }
    }
}
```



Exercício 05

- Agora, nossa aplicação deve imprimir o relatório e exibir a barra, ao mesmo tempo:

```
public static void main(String[] args) {  
    System.out.println("Início da aplicação");  
    //Criação dos objetos executáveis  
    GeradorRelatorio relatorio = new GeradorRelatorio();  
    BarraDeProgresso barra = new BarraDeProgresso();  
    //Processo de impressão do relatório  
    Thread threadRelatorio = new Thread(relatorio);  
    threadRelatorio.start();  
    //Processo de exibição da barra de progresso  
    Thread threadBarra = new Thread(barra);  
    threadBarra.start();  
    System.out.println("Fim da aplicação");  
}
```



Resultado gerado

- Os processo concorrem por tempo de execução no processador;
- As regras de escalonamento definem o tempo de execução de cada processo;

```
Início da aplicação  
Linha: 0  
Fim da aplicação  
Linha: 1  
Linha: 2  
== Barra: 0  
Linha: 3  
== Barra: 1
```



Entendendo os conceitos

- É serviço do escalonador de threads(**scheduler**) alternar o tempo de execução de cada Thread iniciada(**start**)
- A **troca de contexto** ocorre quando o escalonador salva o estado da thread atual, para recuperar depois, e pára sua execução;
- Neste momento, é iniciada **uma nova** thread ;
- ou é recuperado o estado de outra que estava parada, voltando a executar;



Interrupção de execução

- Imagine que temos a necessidade de imprimir os nomes das frutas de um cesto;
- Queremos “dar um tempo” para a leitura dos usuário;
- Com isso, precisamos que, após a impressão de um nome, seja realizada uma pausa na execução;
- Para isso, usamos `Thread.sleep(tempo)`

Exercício 06 – Crie a cesta de frutas

```
class CestaFrutas implements Runnable{
    public void run() {
        //Criação da lista de frutas
        String [] ingredientes=
            {"Banana", "Mamão", "Maca", "Abacate"};
        System.out.println("Inicio do run()");
        //Impressão da lista de frutas
        for (String fruta : ingredientes) {
            System.out.println(fruta);
            //Dormindo por 3 segundos
            try {
                Thread.sleep(3 * 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Fim do run()");
    }
}
```



Exercício 07

- Implemente e execute o código a seguir;
- Qual o resultado?

```
public static void main(String[] args) {  
    //Criação do objeto executável  
    CestaFrutas salada = new CestaFrutas();  
    //Criação da thread  
    Thread executar = new Thread(salada);  
    //Execução da thread  
    executar.start();  
}
```



Herança com Thread

- Consultando a documentação java, verificamos que a classe **Thread** implementa **Runnable**;
- Isso nos permite criar uma classe filha de Thread e sobrescrever o método **run()**;
- Isso nos permite colocar na mesma classe o executor e o executado;



Exercício 08

- Implemente a herança de Threads

```
//Classe filha de Thread
public class ThreadFilha extends Thread {
    //Método executado
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println("Linha: " + i);
        }
    }
    public static void main(String[] args) {
        //Chamada ao metodo executor
        new ThreadFilha().start();
    }
}
```



Comentando o código

- É mais fácil criar uma classe filha de Thread do que usar um objeto Runnable;
- Mas não é uma boa prática estender uma Thread;
- Com extends Thread, nossa classe ficaria muito limitada, não podendo herdar os componentes de outra;



Garbage Collector

- O Garbage Collector (coletor de lixo) é uma **Thread** responsável por jogar fora todos os objetos que não estão sendo referenciados;
- Imaginemos o código a seguir:

```
public static void main(String[] args) {  
    //Criando e referenciando dois objetos Conta  
    1 ContaBancaria conta1 = new ContaBancaria(500);  
      ContaBancaria conta2 = new ContaBancaria(300);  
  
    2 //Mudando a referência  
      conta1 = conta2;  
}
```



De olho no código

- Com a execução do item 1, são criados dois objetos ContaBancaria e atribuídos às referências `conta1` e `conta2`;
- No item 2, as duas referências passam a apontar para o mesmo objeto;
- Neste momento, quantos objetos existem na memória? Um ou dois?
- Perdemos a referência a um dos objetos;



Entendendo os conceitos

- O objeto sem referência **não** pode mais ser acessado;
- Podemos afirmar que ele saiu da memória?
- Como Garbage Collector é uma Thread, por ser executado a qualquer momento;
- Com isso, dizemos que o objeto sem referência está **disponível** para coleta;



System.gc()

- Você consegue executar o Garbage Collector;
- Mas chamando o método estático `System.gc()` você está sugerindo que a JVM rode o Garbage Collector naquele momento;
- Sua sugestão pode ser aceita ou não;
- Você não deve basear sua aplicação na execução do Garbage Collector;



Problemas de concorrência

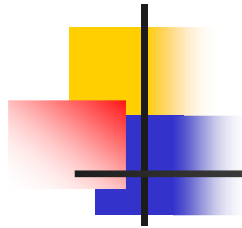
- Imagine que temos um Banco deve possuir milhões de contas do tipo poupança;
- Todo dia 10 de cada mês, precisamos atualizar o saldo da poupança em 1%;
- Nesse dia, é executada uma thread que carrega a coleção de contas do banco e executa o método atualizar();



Métodos de ContaBancaria

- Vamos pensar na seguinte implementação para os métodos depositar() e atualizar():

```
public void depositar(double valor){  
    double novoSaldo = this.saldo + valor;  
    this.saldo = novoSaldo;  
}  
  
public void atualizar(double taxa){  
    double saldoAtualizado = this.saldo * (1 + taxa);  
    this.saldo = saldoAtualizado;  
}
```



Cenário do caos:

- Imagine uma Conta com saldo de 100 reais. Um cliente entra na agência e faz um depósito de 1000 reais.
- Isso dispara uma Thread no banco que chama o método `deposita()`; ele começa calculando o novoSaldo que
- passa a ser 1100 (`novoSaldo = this.saldo + valor`). Só que por algum motivo que desconhecemos, o escalonador pára essa thread.



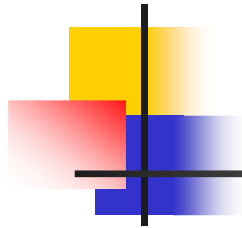
Cenário do caos – cont...

- Neste exato instante, ele começa a executar uma outra Thread que chama o método atualiza da mesma Conta, por exemplo, com taxa de 1%. Isso quer dizer que o novoSaldo passa a valer 101 reais (`saldoAtualizado = this.saldo * (1 + taxa)`).
- Neste instante o escalonador troca de Threads novamente. Agora ele executa (`this.saldo = novoSaldo`) na Thread que fazia o depósito; o saldo passa a valer **R\$ 1100,00**.
- Acabando o deposita(), o escalonador volta pra Thread do atualiza() e executa(`this.saldo = saldoAtualizado`), fazendo o saldo valer **R\$ 101,00**.



Resultado do caos

- O depósito de mil reais foi ignorado e seu Cliente ficará pouco feliz com isso;
- O problema, foi o **acesso simultâneo** de duas Threads ao mesmo objeto.
- Dizemos que essa classe não é **thread safe**, isso é, não está pronta para ter uma instância utilizada entre várias threads concorrentes;



Região crítica

- Esses métodos não deveriam ser acessados por duas threads, ao mesmo tempo;
- **Região crítica** é o trecho de código que apenas uma thread pode acessar por vez;
- Como bloquear o acesso concorrente a um método?



O modificador de acesso synchronized

- Métodos synchronized só podem ser acessados por uma thread por vez;

```
public synchronized void depositar(double valor){  
    double novoSaldo = this.saldo + valor;  
    this.saldo = novoSaldo;  
}  
  
public synchronized void atualizar(double taxa){  
    double saldoAtualizado = this.saldo * (1 + taxa);  
    this.saldo = saldoAtualizado;  
}
```

- Uma thread espera o fim da execução de outra que iniciou primeiro;



Vector e Hashtable

- Duas collections muito famosas são Vector e Hashtable;
- Essas coleções são **Thread safe**, ao contrário de suas irmãs ArrayList e HashMap;
- Por que não usamos sempre classes thread safe?
- Devido ao custo da operação;



Bibliografia

- Java - Como programar, de Harvey M. Deitel
- Use a cabeça! - Java, de Bert Bates e Kathy Sierra
- (Avançado) Effective Java Programming Language Guide, de Josh Bloch



Referências WEB

- SUN: www.java.sun.com
- Threads:
<http://download.oracle.com/javase/tutorial/essential/concurrency/>

Fóruns e listas:

- Javaranch: www.javaranch.com
- G.U.J: www.guj.com.br

Apostilas:

- Argonavis: www.argonavis.com.br
- Caelum: www.caelum.com.br



Java Standard Edition (JSE)

14. Threads



Esp. Márcio Palheta

Gtalk: marcio.palheta@gmail.com