CAPÍTULO 17

Programação Concorrente e Threads

"O único lugar onde o sucesso vem antes do trabalho é no dicionário."
— Albert Einstein

Ao término desse capítulo, você será capaz de:

- executar tarefas simultaneamente;
- colocar tarefas para aguardar até que um determinado evento ocorra;
- entender o funcionamento do Garbage Collector.

17.1 - THREADS

"Duas tarefas ao mesmo tempo"

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**.

Threads em Java

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);
```

17.2 - Escalonador e trocas de contexto

Veja a classe a seguir:

```
1  public class Programa implements Runnable {
2
3    private int id;
4    // colocar getter e setter pro atributo id
5
6    public void run () {
7      for (int i = 0; i < 10000; i++) {
8          System.out.println("Programa " + id + " valor: " + i);
9      }
10   }
11 }
```

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:

```
1  public class Teste {
2    public static void main(String[] args) {
3
4      Programa p1 = new Programa();
5      p1.setId(1);
6
7      Thread t1 = new Thread(p1);
8      t1.start();
9
10     Programa p2 = new Programa();
11     p2.setId(2);
12
13     Thread t2 = new Thread(p2);
14     t2.start();
15
16   }
17 }
```

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.

17.3 - Garbage Collector

O **Garbage Collector** (coletor de lixo, lixeiro) funciona como uma `Thread` responsável por jogar fora todos os objetos que não estão sendo referenciados por nenhum outro objeto - seja de maneira direta ou indireta.

Considere o código:

```
Conta conta1 = new ContaCorrente();
Conta conta2 = new ContaCorrente();
```

Até este momento, sabemos que temos 2 objetos em memória. Aqui, o *Garbage Collector* não pode eliminar nenhum dos objetos, pois ainda tem alguém se referindo a eles de alguma forma.

Podemos, então, executar uma linha que nos faça perder a referência para um dos dois objetos criados, como, por exemplo, o seguinte código:

```
conta2 = conta1;
```

Quantos objetos temos em memória?

Perdemos a referência para um dos objetos que foram criados. Esse objeto já não é mais acessível. Temos, então, apenas um objeto em memória? Não podemos afirmar isso! Como o *Garbage Collector* é uma Thread, você não tem garantia de

quando ele vai rodar. Você só sabe que, em algum momento no futuro, aquela memória vai ser liberada.

Algumas pessoas costumam atribuir `null` a uma variável, com o intuito de acelerar a passagem do *Garbage Collector* por aquele objeto:

```
for (int i = 0; i < 100; i++) {
  List x = new ArrayList();
  // faz algumas coisas com a arraylist
  x = null;
}
```

Isso rarissimamente é necessário. O *Garbage Collector* age apenas sobre objetos, nunca sobre variáveis. Nesse caso, a variável `x` não existirá mais a cada iteração, deixando a `ArrayList` criada sem nenhuma referência para ela.

System.gc()

Você nunca consegue forçar o Garbage Collector, mas chamando o método estático `gc` da classe `System`, você está sugerindo que a Virtual Machine rode o Garbage Collector naquele momento. Se sua sugestão vai ser aceita ou não, isto depende de JVM para JVM, e você não tem garantias. Evite o uso deste método. Você não deve basear sua aplicação em quando o Garbage Collector vai rodar ou não.

Finalizer

A classe `Object` define também um método `finalize`, que você pode reescrever. Esse método será chamado no instante antes do Garbage Collector coletar este objeto. Não é um destrutor, você não sabe em que momento ele será chamado. Algumas pessoas o utilizam para liberar recursos "caros" como conexões, threads e recursos nativos. Isso deve ser utilizado apenas por segurança: o ideal é liberar esses recursos o mais rápido possível, sem depender da passagem do Garbage Collector.

17.4 - Exercícios

1. Teste o exemplo deste capítulo para imprimir números em paralelo.

Escreva a classe Programa:

```
1  public class Programa implements Runnable {
2
3    private int id;
4    // colocar getter e setter pro atributo id
5
6    public void run () {
7      for (int i = 0; i < 10000; i++) {
8        System.out.println("Programa " + id + " valor: " + i);
9      }
10   }
11 }
```

Escreva a classe de Teste:

```
1  public class Teste {
2    public static void main(String[] args) {
3
4      Programa p1 = new Programa();
5      p1.setId(1);
6
7      Thread t1 = new Thread(p1);
8      t1.start();
9
10     Programa p2 = new Programa();
11     p2.setId(2);
12
13     Thread t2 = new Thread(p2);
14     t2.start();
15
16   }
17 }
```

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

17.5 - 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:

```
1  public class Programa1 implements Runnable {
2    public void run () {
3      for (int i = 0; i < 10000; i++) {
4        System.out.println("Programa 1 valor: " + i);
5      }
6    }
```

```
7 }
```

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`.

E com lambda do Java 8?

Dá para ir mais longe com o Java 8, utilizando o lambda. Como `Runnable` é uma interface funcional (contém apenas um método abstrato), ela pode ser facilmente escrita dessa forma:

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

A sintaxe pode ser um pouco estranha. Como não há parâmetros a serem recebidos pelo método `run`, usamos o `()` para indicar isso. Vale lembrar, mais uma

vez, que no lambda não precisamos escrever o nome do método que estamos implementando, no nosso caso o `run`. Isso é possível pois existe apenas um método abstrato na interface.

Quer deixar o código mais enxuto ainda? Podemos passar o lambda diretamente para o construtor de `Thread`, sem criar uma variável temporária! E logo em seguida chamar o `start`:

```
new Thread(() -> {
  for(int i = 0; i < 10000; i++)
    System.out.println("programa 1 valor " + i);
}).start();
```

Obviamente o uso excessivo de lambdas e classes anônimas pode causar uma certa falta de legibilidade. Você deve lembrar que usamos esses recursos para escrever códigos mais legíveis, e não apenas para poupar algumas linhas de código. Caso nossa implementação do lambda venha a ser de várias linhas, é um forte sinal de que deveríamos ter uma classe a parte somente para ela.

CAPÍTULO ANTERIOR:

Collections framework

PRÓXIMO CAPÍTULO:

E agora?

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter