Linguagem Java

Adenilso da Silva Simão

17/05/05

Introdução

- Uma *thread* é uma linha de execução dentro de um programa
  - Conceito de SO
- Java implementa suporte nativo a *thread*s
  - Duas formas de se implementar
    - Estendendo a classe `Thread`
    - Implementando a interface `Runnable`

Estendendo a classe `Thread`

- Deve-se sobrepor o método `run()`
  - Ele deve conter toda a atividade da *thread*

```
1  public class SimpleThread extends Thread {
2    public SimpleThread (String str) {
3      super (str);
4    }
5    public void run () {
6      for (int i = 0; i < 10; i++) {
7              System.out.println (i + " " + getName ());
8              try {
9                sleep ((long) (Math.random () * 1000));
10             }
11       catch (InterruptedException e) { }
12       }
13     System.out.println ("DONE! " + getName ());
```

Estendendo a classe `Thread` (II)

- Pode-se criar agora várias `SimpleThreads`

```
1 public class ThreadsTest {
2   public static void main (String[]args) {
3     new SimpleThread("POO").start();
4     new SimpleThread("BD").start();
5     new SimpleThread("SO").start();
6     new SimpleThread("RC").start();
7   }
8 }
```

Estendendo a classe `Thread` (III)

- E quando a classe já estende outra
  - Implemente a interface `Runnable`
    - Defina o método `run()`

```
1  public class OtherThread implements Runnable {
2    private String name = null;
3    public OtherThread (String str) {
4      name = str;
5    }
6    public String getName () {
7      return name;
8    }
9    public void run () {
10     // ...
11   }
12 }
```

Estendendo a classe `Thread` (IV)

- Crie um objeto `Thread` com um objeto da classe que implementa o `Runnable`

```
1 public class ThreadsTest {
2   public static void main (String[]args) {
3     new Thread(new OtherThread("POO")).start();
4     new Thread(new OtherThread("BD")).start();
5     new Thread(new OtherThread("SO")).start();
6     new Thread(new OtherThread("RC")).start();
7   }
8 }
```

Estendendo a classe `Thread` (V)

- Uma *thread* congela quando
  - está dormindo (com o método `sleep`)
  - está esperando alguma condição (com o método `wait`)
  - está esperando uma operação de I/O
- Uma *thread* só para quando o método `run()` para de executar
  - O método `stop()` existe, mas não deve ser utilizado

Race Condition

- Para evitar que duas *threads* entrem simultaneamente em uma região crítica
  - Usa-se a palavra reservada `synchronized`
    - Pode ser usar para métodos
- Sempre que uma *thread* chama um método `synchronized` de um objeto
  - Outras *threads* não podem chamar nenhum método `synchronized` do mesmo objeto
    - Pode chamar os outros métodos
  - Quando um método termina, outro pode começar

Race Condition (II)

- Exemplo

```
1 public class CubbyHole {
2   private int contents;
3   private boolean available = false;
4
5   public synchronized int get (int who) {...}
6
7   public synchronized void put (int who, int value) {
8 }
```

Sincronizando

- Pode-se utilizar os métodos `wait` e `notify` para coordenar as *threads*
  - Quando um método `synchronized` percebe que não tem o que fazer, ele pode esperar até que alguém o avise
  - `wait()`
    - Espera indefinidamente pela notificação
  - `wait(long timeout)`
    - Espera no máximo `timeout` segundos pela notificação
  - `wait(long timeout, int nanos)`
    - Espera no máximo `timeout` segundos e `nanos` nanosegundos pela notificação
- Enquanto uma *thread* espera, outra pode entrar na região crítica, fazer alguma coisa e notificar quem está esperando
  - `notify`
  - `notifyAll`

Sincronizando (II)

- Pode-se utilizar o `synchronized` em um bloco de código
  - Informa-se um objeto (qualquer) que será usado como `lock`
    - Somente uma *thread* pode executar o bloco de código de cada vez com o mesmo objeto

```
1 class UmaClasse {
2   int k = 0;
3   int getK() {
4     return k;
5   }
6   int setK(int k) {
7     this.k = k;
8   }
9 }
```

Sincronizando (III)

```
1 ...
2   UmaClasse a;
3 ...
4   synchronized (a) {
5     int x = a.getK();
6     x = x + 1;
7     a.setK(x);
8   }
```

volatile

- Cada *thread* tem uma cópia local das variáveis
  - A JVM decide quando sincronizar as cópias locais com a "memória principal"

```
1   int i1;
2   int geti1() {return i1;}
```

  - *Threads* diferentes podem ter valores diferentes para `i1`
  - Quando isso não é desejável
    - Usa-se a palavra reservada `volatile`

```
1 volatile int i2;
2 int geti2() {return i2;}
```

  - A JVM mantém os atributos `volatite`s sempre sincronizados
- Antes de entrar em um método ou bloco `synchronized`
  - A JVM sincroniza tudo

Deadlock e Starvation

- *Deadlock* e *Starvation*
  - Fica, naturalmente, sobre a responsabilidade do programador evitar que aconteçam

Exercícios

- Implemente um simulador para uma fila de banco
  - Existem vários caixas que atendem (demoram um tempo aleatório para atender um cliente)
- Implemente duas situações
  - Uma fila para todos os caixas
  - Uma fila para cada caixa
  - Compare os tempos médios de espera em cada situação