UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
Prof. Sérgio Teixeira de Carvalho

Threads em Java
Atividade 1[1]

Introdução

Em programação concorrente há duas unidades básicas de execução: processos e *threads*. Na linguagem Java a programação concorrente é mais comum com *threads.* Entretanto, processos são importantes.

Um sistema de computação normalmente tem muitos processos e *threads.* Um processo tem um ambiente de execução próprio com um conjunto completo de recursos para a sua execução; em particular, um processo tem seu próprio espaço de memória.

Threads existem dentro de um processo; todo processo tem pelo menos uma *thread*. *Threads* compartilham os recursos do processo, incluindo memória e arquivos abertos. Isto torna eficiente, mas potencialmente problemática, a comunicação.

A execução *multithread* é uma característica da plataforma Java. Toda aplicação tem pelo menos uma *thread* – ou várias. Do ponto de vista do programador, você inicia com uma *thread,* chamada *main thread* - pense a *main thread* como o aplicativo Java com o método main(). Esta *thread* tem a capacidade de criar *threads* adicionais.

Definindo e iniciando uma Thread

Cada *thread* está associada a uma instância da classe Thread. Uma aplicação que cria uma instância de Thread deve fornecer o código que executará naquela *thread*. Há duas formas de fazer isso:

- Fornecer um objeto Runnable. A interface Runnable define um único método, run(); neste método você define o código que será executado na *thread.* O objeto Runnable é passado para o construtor da classe Thread, como no exemplo a seguir:

1 Baseado no tutorial Essencial Classes, Sun Microsystems.

Exemplo 1

```
public class ThreadSimples implements Runnable {

    public void run() {
    System.out.println("Olá de uma thread!");
    }

    public static void main(String args[]) {
         ThreadSimples simples = new ThreadSimples();
         Thread thread = new Thread(simples);
         thread.start();
    }
    }
```

- Fornecer uma subclasse de Thread. A própria classe Thread implementa a interface Runnable, embora o método run() não faça nada (esteja vazio na classe Thread). Uma aplicação pode derivar da classe Thread (extends Thread) e fornecer sua própria implementação para run(), como no exemplo:

Exemplo 2

```
public class ThreadSimples extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
      ThreadSimples simples = new ThreadSimples();
        simples.start();
    }
}
```

Observe que, nos dois exemplos, deve-se invocar o método start() para iniciar uma nova *thread*. A invocação do método start() provoca o início da nova *thread* executando o método run().

Qual deles você usaria? O primeiro exemplo (empregando a interface Runnable) é mais geral, pois você pode, além de implementar Runnable, herdar de outra classe, se assim for necessário. O segundo exemplo (empregando a classe Thread) é mais fácil de usar, mas é limitado, pois você necessariamente precisar herdar da classe Thread, impedindo a herança de outra classe, se fosse necessário. Lembre-se que em Java não é possível herdar de duas ou mais classes (herança múltipla).

A classe Thread define métodos úteis para o gerenciamento de *threads.* Há métodos estáticos que fornecem informações sobre as *threads* ou mesmo que afetam o status das mesmas.

1o. Exercício:
Altere o Exemplo 1 acima, codificando uma nova classe denominada ExecutaThread. Esta classe deve ter apenas o método main() e a classe ThreadSimples deve ter apenas o método run().

2o. Exercício:
Altere o Exemplo 2 acima, codificando uma nova classe denominada ExecutaThread. Esta classe deve ter apenas o método main() e a classe ThreadSimples deve ter apenas o método run().

Pausa na execução com o método sleep()

O método sleep() da classe Thread faz com que a *thread* corrente suspenda sua execução por um período especificado de tempo. Esta é uma forma eficiente de liberar o processador para outras *threads* de uma aplicação também em execução no sistema de computação.

Duas versões sobrecarregadas de sleep() são fornecidas: uma que especifica o tempo em milisegundos e outra que especifica o tempo em nanosegundos. No entanto, esses métodos não têm a precisão garantida, pois dependem dos procedimentos do sistema operacional. Além disso, durante o período do *sleep,* a thread pode ser terminada por interrupções (*interrupts*).

Exemplo 3

```
public class ThreadSimples {
    public static void main(String args[]) throws InterruptedException {
        String info[] = {
            "Java",
            "é uma boa linguagem.",
            "Com threads",
            "é melhor ainda."
        };

        for (int i = 0; i < info.length; i++) {
            Thread.sleep(4000);
            System.out.println(info[i]);
        }
    }
}
```

Observe que quando o método estático Thread.sleep() é invocado a thread corrente, no caso a *main* é suspensa por 4000 milisegundos, correspondente a 4 segundos.

Repare a utilização do *throws InterruptedException.* Isso é necessário pois se, durante o período de sleep de uma thread (dormindo), esta for interrompida, ocorrerá uma exceção que deve ser tratada.

3o. Exercício:
Encontre na classe Thread um método capaz de mostrar o nome da *thread* em execução.
Altere o exemplo 3, inserindo este método.

Interrupção (*interrupts*)

Uma interrupção (interrupt) é uma indicação para uma *thread* de que ela deve parar o que está fazendo. A interrupção de uma thread é feita invocando o método interrupt().

Mas, como uma *thread* sabe que recebeu uma interrupção? Depende do que ela está fazendo. Se a thread, na sua execução, está freqüentemente invocando métodos que tratam a exceção InterruptedException, ela simplesmente retorna do método run() depois do tratamento da exceção (*catch*). Por exemplo, suponha que no Exemplo 3 acima, queremos tratar a situação da thread receber uma interrupção durante seu período de sleep:

Exemplo 4

```
for (int i = 0; i < info.length; i++) {
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        return;
    }
}
```

Muitos métodos que tratam InterruptedException, como o sleep(), são projetos para cancelar sua operação corrente e retornar imediatamente que uma interrupção é recebida.

E sua uma thread ficar muito tempo executando uma operação sem tratar InterruptedException? Ela deve periodicamente invocar o método interrupted(), o qual retorna *true* se uma interrupção tiver sido recebida. Por exemplo:

Exemplo 5

```
for (int i = 0; i < entrada.length; i++) {
    criptografa(entrada[i]);
    if (Thread.interrupted()) {
        return;
    }
}
```

Neste exemplo, o código testa se houve algum pedido de interrupção – interrupted(). Se sim a *thread* termina retornando do run().

Flag de interrupção

O mecanismo de interrupção é implementado usando um *flag* conhecido como *interrupt status.* A invocação a interrupt() liga este *flag*. Quando um *thread* verifica se houve uma interrupção invocando interrupted(), o *interrupt status* é desligado.

Método join()

O método join() permite a uma *thread* esperar que outra complete sua execução. Se *t* é um objeto Thread que está em execução,

```
t.join()
```

faz com que a thread corrente suspenda sua execução (faz uma pausa) até que a thread *t* termine.

4o. Exercício:

O código abaixo consiste de duas *threads*, main e Loop. Faça o seguinte:

(i) Execute a classe.

(ii) Coloque comentários em <u>todas</u> as linhas do código, com um bom nível de detalhamento.

(iii) Descreva o funcionamento da classe.

```
public class ThreadSimples {

   static void mensagem(String messagem) {
       String nomeThread = Thread.currentThread().getName();
       System.out.println(nomeThread + " " + messagem);
   }

   private static class Loop implements Runnable {
       public void run() {
           String info[] = {
               "Java",
               "é uma boa linguagem.",
               "Com threads,",
               "é melhor ainda."
           };
           try {
               for (int i = 0; i < info.length; i++) {
                   Thread.sleep(4000);
                   mensagem(info[i]);
               }
           } catch (InterruptedException e) {
               mensagem("Nada feito!");
           }
       }
   }

   public static void main(String args[]) throws InterruptedException {

        long paciencia = 1000 * 60 * 60;

        if (args.length > 0) {
           try {
                paciencia = Long.parseLong(args[0]) * 1000;
           } catch (NumberFormatException e) {
               System.err.println("Argumento deve ser um inteiro.");
               System.exit(1);
           }

       }

       mensagem("Iniciando a thread Loop");
       long inicio = System.currentTimeMillis();
       Thread t = new Thread(new Loop());
       t.start();

       mensagem("Esperando que a thread Loop termine");
       while (t.isAlive()) {
          mensagem("Ainda esperando...");
           t.join(1000);
           if (((System.currentTimeMillis() - inicio) > paciencia) &&
                   t.isAlive()) {
```

```
                mensagem("Cansado de esperar!");
                t.interrupt();
                t.join();
            }
        }
        mensagem("Finalmente!");
    }
}
```