Programação em Java

Profa. Juliana Vizzotto

juvizzotto@inf.ufsm.br

Disciplina de Paradigmas de Programação

Roteiro

- Exceções
- Threads

Tratamento de Exceções

- Permite o gerenciamento de erros em tempo de execução.
- Uma exceção em Java é um objeto que descreve uma condição de exceção que ocorreu em algum fragmento do código.
- Quando surge uma condição excepcional, um objeto *Exception* é criado e lançado no método que causa a exceção.
- Em java, uma exceção é sempre uma instância da classe *Trowable* (estudar hierarquia *Trowable*).

Tratamento de Exceções

- Todas as exceções em Java são descendentes da classe *Trowable*: se dividem em *Error* e *Exception*
- A hierarquia de *Error* descreve erros internos e exaustão de recursos dentro do Java Runtime, e.g., estouro de memória.
- Tais erros não devem ser tratados pelo programador...há muito pouco que se possa fazer quando esses erros acontecem...

Tratamento de Exceções

- Os objetos de exceção são criados automaticamente pelo runtime de Java em resposta a alguma condição de exceção. Por exemplo:

```
class ExcecaoDivisaoZero{
   public static void main(String args[]){
     int d=0;
     int a=13/d;
     System.out.println("Execução Continua!");
   }
}
```

Tratamento de Exceções

- Quando o runtime do Java tenta executar a divisão, ele percebe que o denominador é zero e constrói um objeto de exceção para fazer com que a execução do código pare e trate essa exceção.
- Neste exemplo, como não foi descrito nenhum manipulador de exceção (bloco try/catch), o programa pára a execução e é chamado o manipulador de exceção padrão do runtime.

Tratamento de Exceções

O programador deve tratar erros que descendem da classe *Exception*!: try e catch

- É sempre mais aconselhável que o programador trate da excecução para que o programa continue a execução.
- A palavra-chave try pode ser pode ser usada para especificar um bloco de código que trata a exceção.
- A palavra-chave catch para especificar qual o tipo de exceção que será capturada.

Tratamento de Exceções

O programador deve tratar erros que descendem da classe *Exception*!: try e catch

```
class ExcecaoDivisaoZero{
    public static void main(String args []){
     try {
           int d=0;
           int a=13/d;
           System.out.println("Dentro do bloco.");
      }
      catch (ArithmeticException e){
          System.out.println("Divisão por zero");
     }
     System.out.println("Continua...");
   }
}
```

Tratamento de Exceções

O programador deve tratar erros que descendem da classe *Exception*!: try e catch

- Quando acontece uma exceção e é encontrado o manipulador de exceção (o bloco try/catch) fornecido pelo programador, o interpretador sai do bloco onde aconteceu a exceção e é executado o manipulador de exceção.
- Após, a execução *continua*.

Tratamento de Exceções

Várias Cláusulas *catch*

```
class MultiCatch{
      public static void main(Strings args[]){
        try{
             int a = args.length;
             System.out.println("a = "+a);
             int b = 13/a;
             int c [] = {1};
             c[42]    = 99;
            }
        catch(ArithmeticException e){
             System.out.println("Div por 0: " +e);
        }
        catch(ArrayIndexOutOfBoundsException e){
             System.out.println("Estouro do indice"+e);
        }
```

Tratamento de Exceções

Criando a sua própria classe de exceções

- É possível criar o seu próprio tratador de exceções!
- A classe deve ser derivada da classe *Throwable* ou de uma de suas subclasses.
- Permite módulos de falhas mais compreensíveis.

```
class MinhaExcecao extends Exception{
     private int detalhe;

     public MinhaExcecao (int a){
          detalhe = a;
     }
     public String toString(){
          return "MinhaExcecao ["+detalhe+"]";
     }
}
```

Tratamento de Exceções

Usando o Throw para lançar exceções

- Uma vez que você crie a sua própria exceção, você precisa lançar essa exceção, ou seja, avisar o interpretador que aconteceu uma exceção que precisa ser tratada:

```
class DemoExcecao{
   public static void main(String args []){
        try {
              int a=11;
              if (a>10) {
                MinhaExcecao mE = new MinhaExcecao(a);
                throw mE;
                 }}
         catch (MinhaExcecao e) {
          System.out.println("Excecao capturada:"+e);}}
```

Threads

Threads

Threads

- Programação Multithread é um paradigma conceitual de programação pelo qual você divide os programas em dois ou mais processos que podem ser executados paralelamente.
- Muitas vezes, o programa implementado não exige os recursos completos do computador.
- O computador pode levar um minuto para ler os dados do usuário do teclado, mas o tempo em que a CPU está envolvida nesse processo é mínimo.
- Devido a isso a CPU fica ociosa grande parte do seu tempo.
- O problema dos ambientes tradicionais de linha de execução única é que você precisa esperar que cada uma dessas tarefas termine para após passar para a outra.
- Além disso, as threads podem se comunicar e compartilhar estruturas de dados, o que pode ser útil.

Threads

- Para determinar que certas classes do seu programa podem executar paralelamente, você tem de definí-las como uma thread.
- Podemos definir uma classe como Thread de duas maneiras:
  - Implementando a interface Runnable
  - Derivando da classe Thread

Threads

Exemplo1

```
class MinhaThread1 extends Thread{
   public MinhaThread1(String nome){
        super(nome);
   }
   public void run(){
       for(int i=0; i<4;i++){
         System.out.println(getName() + " "+i);
         try {
               sleep(400);
             }
         catch(InterruptedException e){ }
         }}}
```

Threads

```
public static void main(String args[]){
    MinhaThread1 t1 = new MinhaThread1 ("Programando");
    MinhaThread1 t2 = new MinhaThread1 ("Tomando café");
    t1.start();
    t2.start();
}
```

Threads

Resultado do Programa anterior

```
juliana@amitaba:~/Teaching/Paradigmas/Codes$ java MinhaThre
Programando 0
Tomando café 0
Programando 1
Tomando café 1
Programando 2
Tomando café 2
Programando 3
Tomando café 3
```

Threads

- Toda thread deve ter um método *run()*. É neste método que é iniciada a execução da Thread;
- Uma thread inicia sua execução pelo método *run()*, assim como uma aplicação Java inicia a sua execução pelo método main().
- O método *run()* não aceita parâmetros. Por isso, quando for necessário passar parâmetros para uma thread, isto deve ser feito através do construtor.
- A chamada do método *start()* da classe Thread, cria uma nova thread e executa o método *run()* definido nesta classe thread.
- O método *sleep()* é um método estático da classe Thread que põe a thread corrente para dormir durante um número de milisegundos.

Threads (Sincronização)

- Em Java, as threads precisam se comunicar
- No exemplo acima, utilizou-se o método *sleep()* como maneira de uma thread comunicar para a outra que está inativa.
- Tente tirar o método *sleep()* do exemplo acima. Você notará que todo o primeiro laço será executado e somente após o segundo laço executará.

Threads (Inteface Runnable)

- Da mesma maneira é possível criar a thread acima, implementando a interface Runnable.
- Como Java não aceita herança múltipla, se a classe thread já for filha de outra classe, não será possível fazer com que ela herde da classe Thread também.
- Nestes casos, ao invés de herdar da classe Thread, a classe pode implementar a interface Runnable.

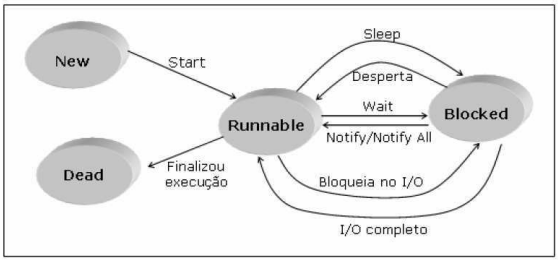
Threads (Inteface Runnable)

```
class MinhaThread2 implements Runnable{
    String nome;
    public MinhaThread2(String nome){
        this.nome = nome;
    }
    public void run(){
     for(int i=0; i<4;i++){
         System.out.println(nome+" "+i);
         try{ Thread.sleep(400);}
         catch(InterruptedException e){}
     }
}
```

Threads (Inteface Runnable)

```
public static void main(String args[]){
      MinhaThread2 p = new MinhaThread2 ("Programando");
      MinhaThread2 c = new MinhaThread2 ("Tomando café");
      Thread t1 = new Thread(p);
      Thread t2 = new Thread(c);
      t1.start();
      t2.start();
}
```

Estados das Threads



Estados das Threads

- *New*: Uma thread está no estado de nova quando ela foi criada pelo programados com o perador *new*.
- *Runnable*: Quando o método *start()* da thread é invocado, ela passa para o estado *runnable*.
- *Blocked*: Quando uma thread entra no estado de bloqueada, uma outra thread é escalonada para executar. Quando ela é reativada (e.g, pq já acabou o tempo de *sleep*, o escalonador verifica se ela tem prioridade maior que a thread que está executando. Em caso positivo, el *preempta* a thread corrente e começa a executar a thread que estava antes bloqueada.

Estados das Threads

Uma thread está no estado bloqueado quando uma das seguintes ações ocorre:

- É chamado o métod *sleep()* da thread.
- A thread chama o método *wait*.
- A thread está bloqueada em uma operação de I/O.

Para saber o estado de uma thread podemos usar o método *isAlive()*. Esse método retorna *true*, caso a thread esteja runnable ou bloqueada.

Prioridades das Threads

- As prioridades de threads são usadas para definir quando cada thread deve ter permissão para ser executada.
- Quando uma thread de prioridade mais alta torna-se ativa ou sai de uma espera de I/O, ela deve poder executar imediatamente, com preempção da thread de prioridade mais baixa.
- As threads de mesma prioridade devem fazer preempção mútua através de um algoritmo de escalonamento "round-robin" para compartilhar o tempo da CPU.
- O pacote de threads em Java precisa trabalhar com o SO.

Prioridades das Threads

- Cada thread em Java tem uma prioridade. É possível determinar a prioridade de uma thread como o método *SetPriority(int prioridade)*. Os valores para a prioridade podem variar de 1 a 10. A prioridade default é 5.
- O escalonador da JVM tenta escalonar a thread de maior prioridade que esteja no estado runnable. Esta thread permanecerá executando até que:
  - seja chamado o método *yield()*;
  - A thread finalize a sua execução;
  - Uma thread de maior prioridade torna-se runnable.

Prioridades das Threads

- Para garantir que uma thread fique inativa, para que outras threads possam ocupar a CPU, pode ser usado o método *yield()*.
- Uma thread pode chamar o método *yield()* ou *sleep()* sempre que estiver dentro de uma grande laço para não monopolizar a sistema.
- Threads que não seguem essa filosofia são chamadas egoístas.
- Note que a chamada do método *sleep()* permite que outras threads de prioridade menor possam executar. Porém, o método *yield()* apenas dá chance de executar para threads de mesma ou maior prioridade.

Prioridades das Threads

```
class ThreadsPrioridade extends Thread{
   public ThreadsPrioridade(String nome){
     super(nome); }
    public ThreadsPrioridade(String nome, int prior){
       super(nome);
       setPriority(prioridade);}
    public void run(){
       for(int i=0; i<4;i++){
  System.out.println(getName()+ " "+i);
          yield();}
    }
```

Prioridades das Threads

```
 public static void main(String args[]){
   ThreadsPrioridade t1 = new ThreadsPrioridade ("Menor", 3
   ThreadsPrioridade t2 = new ThreadsPrioridade ("Maior", 7
   ThreadsPrioridade t3 = new ThreadsPrioridade ("Default1"
   ThreadsPrioridade t4 = new ThreadsPrioridade ("Default2"
   t1.start();
   t2.start();
   t3.start();
   t4.start();
 }
}
```

Sincronização

- Uma thread executando pode acessar qualquer objeto a qual ela tem referência.
- É possível que duas threads tentem acessar o mesmo objeto, interferindo umas nas outras.
- Deve-se tomar cuidado que as threads acessem o objeto uma por vez, i.e., elas devem SINCRONIZAR!
- Para tratar dessa questão, Java expõe o conceito de MONITOR.
- Todo o objeto possui um monitor implícito.

Sincronização

- Quando uma thread tenta acessar um método marcado como *synchronized* de um determinado objeto, ele entra no *monitor* desse objeto.
- Todas as outras threads que desejarem acessar um método *synchronized* deste objeto devem esperar.
- Para a thread liberar o monitor para que outra thread possa acessar o objeto, ela precisa apenas sair do método *synchronized*.

Prioridades das Threads

```
class SemSincronismo{
  public static void main(String args[]){
    Chamame alvo = new Chamame();
    Chamador t1  = new Chamador(alvo,"Oi");
    Chamador t2  = new Chamador(alvo,"Mundo");
    Chamador t3  = new Chamador(alvo,"Sincronizado");
    t1.start();
    t2.start();
    t3.start();
}
}
```

Prioridades das Threads

```
class Chamador extends Thread{
   Chamame alvo;
   String msg;
   public Chamador(Chamame alvo, String s){
     this.alvo = alvo;
     msg = s;}
   public void run(){
    alvo.chamar(msg);
  }
}
```

Prioridades das Threads

```
class Chamame{
   public void chamar(String msg){
     System.out.print("["+msg);
     try{
         Thread.sleep(1000);}
     catch(InterruptedException e){};
     System.out.println("]");
  }
}
```

Prioridades das Threads

```
juliana@amitaba:~/Paradigmas/Codes$ java SemSincronismo
[Oi[Mundo[Sincronizado]
]
]
```

Este resultado obtido se deve pq não existe nada determinando que se quer que aquele pedaço de código seja ATÔMICO (indivisível).

Prioridades das Threads

Podemos usar a palavra-chave *synchronized* para garantir que as threads não executem aquele método ao mesmo tempo.

```
class Chamame{
   public synchronized void chamar(String msg){
     System.out.print("["+msg);
     try{
         Thread.sleep(1000);}
     catch(InterruptedException e){};
     System.out.println("]");
  }
}
```

Aspectos da Programação Concorrente

- Concorrência: um conjunto de fluxos de execução independentes que disputam o uso dos recursos de arquitetura para terem suas instruções executadas.
- Disputa reflete o *compartilhamento* do tempo de uso do processador e de espaço de armazenamento na memória.
- Programação concorrente: busca-se a implementação de uma única aplicação, utilizando-se do recurso e decompondo-a em diversos fluxos de execução.
- O que implica que os fluxos não sejam completamente independentes.
- Sendo cada fluxo de execução responsável por uma parte da solução do problema, de alguma forma é necessário que resultados obtidos por em destes fluxos seja comunicado a outro de forma a compor a solução final.

Aspectos da Programação Concorrente

- Desta forma, programar de forma concorrente implica detectar na aplicação as atividades que não possuem restrições temporais e os pontos onde restrições temporais existem.
- Para cada conjunto de instruções, devemos *definir quais produzem resultados necessários ao início da execução de outro conjunto e quais conjuntos são independentes*.
- Esta forma de programação permite que diferentes fluxos de instrução colaborem entre si, empregando mecanismos de sincronização.
- Um programa concorrente é, desta forma, composto por um conjunto de fluxos de execução que, de alguma forma ordenada, trocam informações.
- Um programa concorrente será corretamente executado se todas as tarefas definidas forem executadas respeitando as sincronizações definidas entre elas.

Aspectos da Programação Concorrente

- Em um ambiente *Multithread* a concorrência é definida por um conjunto de atividades concorrentes sendo executadas em uma arquitetura de $n \geq 1$ processadores em um espaço de endereçamento (memória) único.
- A forma de interação mais natural, e menos onerosa, para compartilhamento de informações em um programa multithread é através do espaço de armazenamento provido pelo próprio computador.
- O mecanismo de comunicação explora o fato de que os dados manipulados pela instruções contidas em diferentes fluxos de execução são variáveis armazenadas na memória.
- Sendo a memória um recurso compartilhado, um dado escrito por uma instrução em um fluxo de execução pode ser lido por uma instrução em outro fluxo.

O método *join()*

- Sincronização com o *término* de uma thread.

Trabalho Final

- Implementar um programa paralelo em Java usando Threads para gerar um Fractal no conjunto de Mandelbrot.
- Um fractal é um objeto geométrico que pode ser divido em partes, cada uma das quais semelhante ao objeto original.
- Um fractal pode ser gerado por um padrão repetido, tipicamente um processo recorrente ou iterativo.
- Uma propriedade interessante do Fractal no conjunto de Mandelbrot é que os valores associados a cada ponto do plano complexo pode ser computado de forma *independente*.
- Assim, várias threads podem, de forma concorrente, computar regiões deste plano.
- A *combinação* dos resultados prove a imagem final.