

Aula 23

Programação *Multithreading*

Rafael Geraldelli Rossi

Introdução

- **Os computadores são capazes de realizar operações concorrentemente (paralelamente)**
- Vale ressaltar que apenas computadores que têm múltiplos processadores podem, de fato, executar múltiplas instruções concorrentemente → os sistemas operacionais em computadores de um único processador criam a ilusão da execução concorrente alternando rapidamente entre atividades

Concorrência no Java

- Em geral, as linguagens de programação fornecem instruções de controle sequenciais que permitem especificar que apenas uma ação deve ser realizada por vez, com a execução avançando para a ação seguinte depois que a anterior tiver sido concluída
- O Java disponibiliza a concorrência por meio de sua biblioteca padrão
- No Java, você especifica que um aplicativo contém **threads**, ou **linhas de execução separadas**

Concorrência no Java

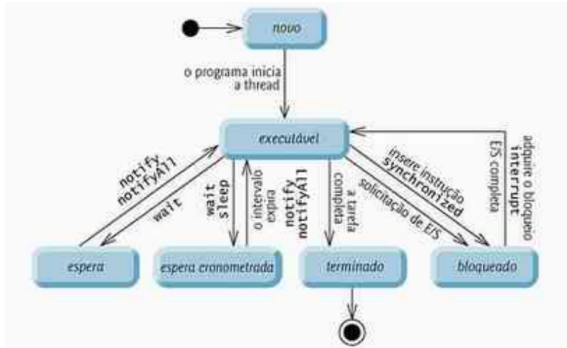
- Cada *thread* tem sua própria pilha de chamadas de método, seu próprio contador de programa, e outros recursos necessário para se ter o mesmo comportamento de um programa completo em execução
- Além disso, ao utilizar *threads*, estas podem compartilhar recursos no nível do aplicativo (ex: variáveis e objetos)
- **OBSERVAÇÃO:** a capacidade de múltiplas *threads* (ou *multithreading*) não está disponível nas linguagens C e C++ básicas (apenas por meio de bibliotecas) → chamadas não portáveis (específicas para cada sistema operacional)

Exemplos do uso de programação concorrente

- Ao fazer o *download* de um arquivo grande (ex: filme) na internet
 - O usuário pode não querer esperar até o *download* do filme inteiro terminar para iniciar a reprodução
 - Pode-se colocar múltiplas *threads* para trabalhar concorrentemente
 - *Thread 1*: *download* do vídeo
 - *Thread 2*: reprodução do vídeo
 - Para evitar instabilidades na reprodução, pode-se sincronizar as *threads* de modo que a *thread* do *player* não inicie até que haja uma quantidade suficiente de filme baixada

Estado e Ciclos de Vida de uma Thread

- Um *thread* pode se encontrar em um dos estados da figura abaixo:



Estados Novo e Executável

- Uma nova *thread* inicia seu ciclo de vida no estado **novo**
- A *thread* permanece no estado **novo** até que o programa inicia a *thread*, o que a coloca no estado **executável**
- Uma *thread* no estado **executável** está executando sua tarefa

Estado de Espera

- A *thread* executável pode transitar para o estado de **espera** enquanto espera outra *thread* realizar uma tarefa
- Uma *thread* no estado de **espera** transita de volta para o estado **executável** apenas quando outra *thread* a notifica para continuar executando

Estado de Espera Sincronizada

- Uma thread no estado **executável** pode entrar no estado de **espera sincronizada** por um intervalo específico de tempo
- A thread transita para o estado **executável** quando esse intervalo de tempo expira ou quando o evento pelo qual ela está esperando ocorre
- As *threads* em **espera sincronizada** não podem utilizar um núcleo de um processador, mesmo se houver algum disponível

Estado de Espera Sincronizada

- Uma maneira de colocar uma *thread* no estado de **espera sincronizada** é colocar a *thread* executável para “dormir”
- Uma *thread* **adormecida** permanece no estado de espera sincronizada por um período de tempo designado (chamado de **intervalo de adormecimento**) e depois ela retorna ao estado executável
- As *threads* dormem quando, por um breve período, não têm que realizar nenhuma tarefa

Estado de Espera Sincronizada

- **Exemplo de um processador de textos:**
 - Um processador de textos pode conter uma *thread* que grave periodicamente backups do documento atual no disco para fins de recuperação
 - Neste caso, se a *thread* não dormisse entre os sucessivos backups, a *thread* estaria continuamente gravando uma cópia do documento em disco
 - Esse *loop* consumiria tempo de processador, reduzindo assim o desempenho do sistema
 - É mais eficiente disparar uma *thread* especificando um intervalo de adormecimento e entrar no estado de **espera sincronizada** → voltando ao seu estado **executável** quando o intervalo de adormecimento expira (novo backup é gravado)

Estado Bloqueado

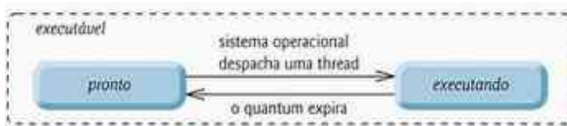
- Uma *thread* executável transita para o estado **bloqueado** quando tenta realizar uma tarefa que não pode ser completada imediatamente e deve esperar temporariamente até que alguma condição seja satisfeita
- **Ex:** quando uma *thread* emite uma solicitação de entrada/saída, o sistema operacional bloqueia a *thread* de executar até que essa solicitação de entrada/saída se complete
- Nesse ponto, a *thread* **bloqueada** transita para o estado **executável** e, desse modo, pode retomar a execução
- Uma *thread* **bloqueada** não pode utilizar um processador, mesmo se algum estiver disponível

Estado Terminado

- Uma *thread* executável entra no estado **terminado** (às vezes chamado estado **morto**) quando completa sua tarefa com sucesso ou, de outro modo, é finalizada (ex: erro ou exceção não capturada)

Visão do Sistema Operacional do Estado Executável

- No nível do sistema operacional, o estado executável do Java geralmente inclui dois estados separados: executável e pronto
- **OBSERVAÇÃO:** o sistema operacional oculta esses estados da JVM, que vê apenas o estado executável



Visão do Sistema Operacional do Estado Executável

- Quando uma *thread* entra pela primeira vez no estado executável, a partir do estado **novo**, ela está no estado **pronto**
- Uma *thread* **pronta** entra no estado de **execução** (isto é, começa a executar) quando o sistema operacional a atribui a um processador (despachar a *thread*)
- Na maioria dos sistemas operacionais, cada *thread* recebe uma pequena quantidade de tempo de processador (*quantum* ou fração de tempo)

Visão do Sistema Operacional do Estado Executável

- Quando seu quantum expira, a *thread* retorna ao estado pronto e o sistema operacional atribui outra *thread* ao processador
- A JVM não tem ciência das transições entre os estados **pronto** e **executável**
- A JVM apenas visualiza a *thread* como **executável** e deixa para o SO fazer as transições das *threads*

Prioridades e Agendamento de Threads

- Toda thread no Java tem uma **prioridade de thread** que ajuda a determinar a ordem em que são agendadas
- As prioridades do Java variam entre `MIN_PRIORITY` (constante = 1) e `MAX_PRIORITY` (constante = 10)
- Por padrão, toda thread recebe a prioridade `NORM_PRIORITY` (constante = 5)

Prioridades e Agendamento de Threads

- **OBSERVAÇÃO:** as constantes `MIN_PRIORITY`, `NORM_PRIORITY` e `MAX_PRIORITY` são declaradas na classe `Thread`
- Cada nova thread herda da prioridade da thread que a cria
- A maioria dos sistemas operacionais suporta o fracionamento de tempo, o que permite que *threads* de igual prioridade compartilham um processador pelo mesmo período de tempo

Prioridades e Agendamento de Threads

- Com o fracionamento de tempo, mesmo se a *thread* não tiver concluída a execução, quando seu quantum expirar, o processador é tirado da *thread* e recebe a próxima *thread* de igual prioridade, se houver alguma disponível
- O **agendador de threads** (*thread scheduler*) de um sistema operacional determina qual *thread* deve ser executada em sequência de uma *thread* em execução

Criando e Executando Threads

- Normalmente, a forma mais utilizada para criar aplicativos Java de múltiplas threads é implementado a interface `Runnable` (pacote `java.lang`) em uma classe que irá ser executada por meio de uma *thread*
- **OBSERVAÇÃO:** pode se ter o mesmo efeito estendendo a classe `Thread` já que esta implementa a interface `Runnable`
- Portanto, para se criar um objeto `Thread` deve-se ter uma objeto que é uma `Thread` (herança) ou pode-se passar como argumento do construtor um objeto que implemente a interface `Runnable`

Criando e Executando Threads

- Um objeto Runnable representa uma tarefa que pode executar concorrentemente com outras tarefas
- A interface Runnable declara o método run, que contém o código que define a tarefa que um objeto Runnable deve realizar
- Quando uma thread executando um Runnable é criada e iniciada, ela chama o método run do objeto Runnable, que executa a nova *thread*

Criando e Executando Threads

- **Ex:** fazendo um programa para disparar várias threads e mostrar sua execução em paralelo
- **OBSERVAÇÃO 1:** o tempo de “soneca” é dado em milisegundos
- **OBSERVAÇÃO 2:** o método `Thread.sleep()` gera uma exceção declarada (`InterruptedException`)

Criando e Executando Threads

```
14 public class ImprimirTarefa implements Runnable{
15
16     private int tempoSoneca;
17     private String nomeTarefa;
18     private Random random;
19
20     public ImprimirTarefa(String nomeTarefa){
21         this.nomeTarefa = nomeTarefa;
22         random = new Random();
23         tempoSoneca = random.nextInt(5000);
24     }
25
26     @Override
27     public void run() {
28         for(int i=0; i<3; i++){
29             System.out.println("Tarefa: " + nomeTarefa + " - Tempo de Soneca: " + tempoSoneca);
30             try{
31                 Thread.sleep(tempoSoneca);
32             }catch(InterruptedException exception){
33                 System.err.println("Thread encerrada prematuramente");
34             }
35         }
36     }
37
38 }
```

Criando e Executando Threads

- O método `start()` coloca a thread em estado executável e realiza uma chamada ao método `run` da interface `Runnable` para executar a tarefa

```
12 public class Principal {  
13  
14     public static void main(String[] args){  
15         System.out.println("Criando as threads...!");  
16  
17         Thread thread1 = new Thread(new ImprimirTarefa("Tarefa 1"));  
18         Thread thread2 = new Thread(new ImprimirTarefa("Tarefa 2"));  
19         Thread thread3 = new Thread(new ImprimirTarefa("Tarefa 3"));  
20  
21         System.out.println("Threads criadas... iniciando as tarefas");  
22         thread1.start();  
23         thread2.start();  
24         thread3.start();  
25  
26         System.out.println("Threads despachadas.. fim do bloco main.");  
27     }  
28 }  
29  
30
```

Resultados da Pesquisa Salida x

Console do Depurador x Teste (run) x

run:
Criando as threads...!
Threads criadas... iniciando as tarefas
Threads despachadas.. fim do bloco main.
Tarefa: Tarefa 3 - Tempo de Soneca: 1202
Tarefa: Tarefa 1 - Tempo de Soneca: 1470
Tarefa: Tarefa 3 - Tempo de Soneca: 1202
Tarefa: Tarefa 1 - Tempo de Soneca: 1470
Tarefa: Tarefa 3 - Tempo de Soneca: 1202
Tarefa: Tarefa 2 - Tempo de Soneca: 2529
Tarefa: Tarefa 1 - Tempo de Soneca: 1470
Tarefa: Tarefa 2 - Tempo de Soneca: 2529

Criando e Executando Threads

- No exemplo anterior:
 - O código no método `main` é executado na **thread principal**, que é criada pela própria JVM.
 - Quando o método `main` termina, o programa continua executando porque ainda haverá *threads* que estão ativas
 - O programa não terminará até que a última *thread* complete a execução
 - O programa principal termina antes das outras *threads*

Criando e Executando Threads

- Outra forma de criar threads é criando um Runnable() diretamente dentro do construtor da classe Thread

```
13 public class TesteThread {  
14  
15     public static void main(String[] args) {  
16  
17         Thread thread1 = new Thread(new Runnable() {  
18             @Override  
19             public void run() {  
20                 for(int i=0; i<100; i++){  
21                     System.out.println("Tread1");  
22                 }  
23             }  
24         });  
25         thread1.start();  
26  
27         Thread thread2 = new Thread(new Runnable() {  
28             @Override  
29             public void run() {  
30                 for(int i=0; i<100; i++){  
31                     System.out.println("Tread2");  
32                 }  
33             }  
34         });  
35         thread2.start();  
36  
37     }  
38  
39 }
```

Alguns Métodos Úteis para se Gerenciar Threads

- Métodos úteis para o gerenciamento de *threads*
 - `start()`: inicia um *thread*
 - `stop()`: para uma *thread*
 - `isAlive()`: verifica se uma *thread* ainda está “viva”
 - `setPriority(int prioridade)`: define a prioridade da *thread*

Alguns Métodos Úteis para se Gerenciar Threads

- Métodos úteis para o gerenciamento de *threads*
 - `suspend()`: suspende a *thread* fazendo com que ela possa continuar sua execução do ponto onde parou posteriormente
 - `resume()`: continua a execução de uma *thread* que foi suspensa
 - `sleep(int tempo)`: faz a *thread* “dormir” pelo tempo definido no parâmetro
 - `interrupt()`: faz a *thread* sair do estado de *wait* (espera) ou *sleep* (espera sincronizada)

Gerenciamento de *threads* com o *framework* Executor

- Embora seja possível criar threads explicitamente, recomenda-se utilizar a interface Executor para gerenciar a execução dos objetos Runnable
- Em geral, um objeto Executor cria e gerencia um grupo de threads chamado **pool de threads** para executar objetos Runnable
- Com o Executor: reutiliza-se threads existente para eliminar o *overhead* de criar uma nova *thread* para cada tarefa e podem aprimorar o desempenho otimizando do número de threads a fim de assegurar que o processador permaneça ocupado, sem criar tantas threads que esgotam os recursos do aplicativo

Gerenciamento de *threads* com o *framework* Executor

- A interface `Executor` declara um método único chamado `execute`, que aceita um `Runnable` como um argumento
- O `Executor` atribui cada `Runnable` passado para seu método `execute` uma das threads disponível no *pool* de threads
- Se não houver *thread* disponível, o `Executor` cria uma nova *thread* ou espera uma thread torna-se disponível e atribui a essa *thread* o `Runnable` que foi passado para o método `execute`

Gerenciamento de *threads* com o *framework* Executor

- A interface `ExecutorService` estende o `Executor` e declara muitos outros métodos para gerenciar o ciclo de vida de um `Executor`
- Um `ExecutorService` pode ser criado utilizando métodos `static` declarados na classe `Executor` (pacote `java.util.concurrent`)
- Um `Executors` fornece o método `newCachedThreadPool` para obter um `ExecutorService` que cria novas *threads* conforme requerido pelo aplicativo
- O método `shutdown()` deve ser invocado para encerrar as *threads* quando suas tarefas terminarem

Gerenciamento de *threads* com o *framework* Executor

```
16 public class Principal2 {
17
18     public static void main(String[] args){
19         System.out.println("Criando as threads...!");
20
21         Thread thread1 = new Thread(new ImprimirTarefa("Tarefa 1"));
22         Thread thread2 = new Thread(new ImprimirTarefa("Tarefa 2"));
23         Thread thread3 = new Thread(new ImprimirTarefa("Tarefa 3"));
24
25         System.out.println("Threads criadas... iniciando as tarefas");
26
27         ExecutorService threadExecutor = Executors.newCachedThreadPool();
28         threadExecutor.execute(thread1);
29         threadExecutor.execute(thread2);
30         threadExecutor.execute(thread3);
31
32         threadExecutor.shutdown();
33
34         System.out.println("Threads despachadas.. fim do bloco main.");
35     }
36 }
37
```

Threads.Principal2 > main >

Resultados da Pesquisa Salva x

Console do Depurador x Teste (run) x

run:
Criando as threads...!
Threads criadas... iniciando as tarefas
Threads despachadas.. fim do bloco main.
Tarefa: Tarefa 2 - Tempo de Soneca: 3083
Tarefa: Tarefa 3 - Tempo de Soneca: 1268
Tarefa: Tarefa 1 - Tempo de Soneca: 2726
Tarefa: Tarefa 3 - Tempo de Soneca: 1268
Tarefa: Tarefa 3 - Tempo de Soneca: 1268
Tarefa: Tarefa 1 - Tempo de Soneca: 2726
Tarefa: Tarefa 2 - Tempo de Soneca: 3083
Tarefa: Tarefa 1 - Tempo de Soneca: 2726
Tarefa: Tarefa 2 - Tempo de Soneca: 3083

Gerenciamento de *threads* com o *framework* Executor

- Verificando se as *threads* terminaram dentro de um período de tempo

```
18 public class Principal2 {  
19  
20     public static void main(String[] args){  
21         System.out.println("Criando as threads...!");  
22  
23         Thread thread1 = new Thread(new Tarefa2("Tarefa 1"));  
24         Thread thread2 = new Thread(new Tarefa2("Tarefa 2"));  
25         Thread thread3 = new Thread(new Tarefa2("Tarefa 3"));  
26  
27         System.out.println("Threads criadas... iniciando as tarefas");  
28  
29         ExecutorService threadExecutor = Executors.newFixedThreadPool(10);  
30         threadExecutor.execute(thread1);  
31         threadExecutor.execute(thread2);  
32         threadExecutor.execute(thread3);  
33         threadExecutor.shutdown();  
34  
35         try{  
36             boolean tempoLimite = threadExecutor.awaitTermination(1, TimeUnit.SECONDS);  
37             if(tempoLimite){  
38                 System.out.println("Threads terminaram a tempo");  
39             }else{  
40                 System.out.println("Threads não terminaram a tempo");  
41             }  
42         }catch(Exception e){  
43             e.printStackTrace();  
44         }  
45         System.out.println("Threads despachadas.. fim do bloco main.");  
46     }  
47  
48 }  
49  
50 }
```

Localizar: exception Anterior Próximo

Resultados da Pesquisa Salva x

Console do Depurador x Teste (run) x

run:
Criando as threads...!
Threads criadas... iniciando as tarefas
Threads não terminaram a tempo
Threads despachadas.. fim do bloco main.
Tarefa 1
Tarefa 2
Tarefa 3

Gerenciamento de *threads* com o *framework* Executor

- Verificando se as *threads* terminaram (NÃO FAÇAM ISSO!!!)

```
18 public class Principal2 {
19
20     public static void main(String[] args){
21         System.out.println("Criando as threads...!");
22
23         Thread thread1 = new Thread(new Tarefa2("Tarefa 1"));
24         Thread thread2 = new Thread(new Tarefa2("Tarefa 2"));
25         Thread thread3 = new Thread(new Tarefa2("Tarefa 3"));
26
27         System.out.println("Threads criadas... iniciando as tarefas");
28
29         ExecutorService threadExecutor = Executors.newFixedThreadPool(10);
30         threadExecutor.execute(thread1);
31         threadExecutor.execute(thread2);
32         threadExecutor.execute(thread3);
33         threadExecutor.shutdown();
34
35         while(!threadExecutor.isTerminated()){ }
36
37         System.out.println("Threads despachadas.. fim do bloco main.");
38     }
39 }
40
41
```

Localizar: exception | Anterior | Próximo

Resultados da Pesquisa Saída x

Console do Depurador x | Teste (run) x

run:
Criando as threads...!
Threads criadas... iniciando as tarefas
Tarefa 2
Tarefa 3
Tarefa 1
Threads despachadas.. fim do bloco main.

Gerenciamento de *threads* com o *framework* Executor

- Vale ressaltar que o método `sleep` da classe `Thread` pode ser utilizado para conceitos gerais

```
16 public static void main(String[] args){  
17  
18     try{  
19         for(int i=0;i<1000;i++){  
20             Thread.sleep(1000);  
21             System.out.println(".");  
22         }  
23     }catch(Exception e){  
24         e.printStackTrace();  
25     }  
26  
27 }  
28  
29 }
```

Sincronização

- Quando múltiplas *threads* compartilham um objeto e ele é modificado por uma ou várias delas, podem ocorrer resultados indeterminados, a menos que o acesso ao objeto compartilhado seja gerenciado adequadamente
- Se um *thread* estiver no processo de atualização de um objeto compartilhado e outra *thread* também tentar atualizá-lo, não é claro a atualização de qual *thread* entra em vigor
- Quando isso acontecer, não se pode confiar no comportamento do programa → não há como garantir que o objeto conterá os valores apropriados

Sincronização de Threads

- O problema pode ser resolvido fornecendo a somente uma thread por vez o código de acesso exclusivo que manipula o objeto compartilhado
- Durante esse tempo, outras *threads* que desejarem manipular o objeto são mantidas em espera
- Quando a *thread* com acesso exclusivo ao objeto terminar de manipulá-lo, outras *threads* que desejarem manipular o objeto são mantidas na espera

Sincronização de Threads

- Quando a *thread* com acesso exclusivo ao objeto terminar de manipulá-lo, uma das *threads* que foi mantida na espera tem a permissão de prosseguir
- Esse processo, chamado de sincronização de threads, coordena o acesso a dados compartilhados por múltiplas threads concorrentes
- Sincronizando *threads* dessa maneira, você pode assegurar que cada *thread* que acessa um objeto compartilhado exclui todas as outras *threads* de fazerem isso simultaneamente → **EXCLUSÃO MÚTUA**

Monitores

- Uma maneira comum de realizar a sincronização é utilizar os **monitores** predefinidos do Java
- Todo objeto tem um monitor e um **bloqueio de monitor** (ou **bloqueio intrínseco**)
- O monitor assegura que o bloqueio de monitor do seu objeto é mantido por no máximo uma única *thread* em qualquer dado momento
- Desse modo, monitores e bloqueios de monitores podem ser utilizados para forçar a exclusão mútua

Monitores

- Se uma operação exigir que a *thread* em execução mantenha um bloqueio enquanto a operação for realizada, uma *thread* deve adquirir o bloqueio antes de prosseguir com a operação
- Outras *threads* tentando realizar uma operação que requer o mesmo bloqueio serão bloqueadas até que a primeira *thread* libere o bloqueio → a partir daí outras *threads* tentarão adquirir o bloqueio
- Para especificar que uma *thread* deve manter um bloqueio de monitor para executar um bloco de código, o código deve ser colocado em uma instrução `synchronized`
- O monitor permite que apenas uma *thread* por vez execute instruções dentro de blocos `synchronized`

Monitores

Exemplo de um bloco synchronized

```
synchronized (objeto) { instruções }
```

- No bloco synchronized, *objeto* é o objeto cujo bloqueio de monitor será adquirido
- Objeto é normalmente `this` se for o objeto no qual a instrução synchronized aparece
- O Java também permite **métodos** synchronized → funcionamento semelhante ao bloco synchronized

Exemplo sem sincronização de Threads

```
16 public class ArraySimples {
17
18     int[] array;
19     int indice;
20     Random random;
21
22     public ArraySimples(int tamanho){
23         array = new int[tamanho];
24         random = new Random();
25     }
26
27     public void add(int valor){
28         try{
29             Thread.sleep(random.nextInt(500));
30         }catch(InterruptedException e){
31             e.printStackTrace();
32         }
33
34         array[indice] = valor;
35         System.out.println(Thread.currentThread().getName() + " escreveu o valor " + valor + " na posição " + indice);
36         indice++;
37     }
38
39     public String toString(){
40         return Arrays.toString(array);
41     }
42
43 }
```

Exemplo sem sincronização de Threads

```
13 public class EscritorArray implements Runnable {
14
15     ArraySimples arrayCompartilhado;
16     int valorInicial;
17
18     public EscritorArray(int valorInicial, ArraySimples arrayCompartilhado){
19         this.valorInicial = valorInicial;
20         this.arrayCompartilhado = arrayCompartilhado;
21     }
22
23     @Override
24     public void run() {
25         for(int i=valorInicial;i<valorInicial + 3;i++){
26             arrayCompartilhado.add(i);
27         }
28     }
29 }
30
```

Exemplo sem sincronização de Threads

```
16 public class Principal {  
17  
18     public static void main(String[] args){  
19  
20         ArraySimples arraySimples = new ArraySimples(11);  
21  
22         EscritorArray esc1 = new EscritorArray(0, arraySimples);  
23         EscritorArray esc2 = new EscritorArray(6, arraySimples);  
24  
25         ExecutorService executor = Executors.newCachedThreadPool();  
26         executor.execute(esc1);  
27         executor.execute(esc2);  
28         executor.shutdown();  
29     }  
30 }  
31  
32 }  
33
```

Resultados da Pesquisa Saída x

Console do Depurador x Teste (run) x

```
run:  
pool-1-thread-2 escreveu o valor 6 na posição 0  
pool-1-thread-1 escreveu o valor 0 na posição 0  
pool-1-thread-2 escreveu o valor 7 na posição 1  
pool-1-thread-1 escreveu o valor 1 na posição 2  
pool-1-thread-2 escreveu o valor 8 na posição 3  
pool-1-thread-1 escreveu o valor 2 na posição 4
```

Exemplo de sincronização de Threads

- A classe `ArraySimples`, apresentada anteriormente, é suscetível a erros se acessada concorrentemente por múltiplas *threads* → método `add`
- Qualquer número de *threads* pode ler e modificar os dados da classe `ArraySimples` concorrentemente
- Deve-se, portanto, assegurar que nenhuma outra *thread* pode ler ou alterar o valor do índice ou modificar o conteúdo do array em nenhum momento durante essas três operações → tornar essas três **operações atômicas**
- A atomicidade pode ser alcançada utilizando uma instrução `synchronized` ou um método `synchronized`

Exemplo de sincronização de Threads

```
16 public class ArraySimples {
17
18     int[] array;
19     int indice;
20     Random random;
21
22     public ArraySimples(int tamanho){
23         array = new int[tamanho];
24         random = new Random();
25     }
26
27     public synchronized void add(int valor){
28         int posicao = indice;
29
30         try{
31             Thread.sleep(random.nextInt(500));
32         }catch(InterruptedException e){
33             e.printStackTrace();
34         }
35
36         array[posicao] = valor;
37         System.out.println(Thread.currentThread().getName() + " escreveu o valor " + valor + " na posição " + posicao);
38         ++indice;
39     }
40
41     public String toString(){
42         return Arrays.toString(array);
43     }
44
45 }
```

Exemplo de sincronização de Threads

```
16 public class Principal {
17
18     public static void main(String[] args){
19
20         ArraySimples arraySimples = new ArraySimples(11);
21
22         EscritorArray esc1 = new EscritorArray(0, arraySimples);
23         EscritorArray esc2 = new EscritorArray(6, arraySimples);
24
25         ExecutorService executor = Executors.newCachedThreadPool();
26         executor.execute(esc1);
27         executor.execute(esc2);
28         executor.shutdown();
29     }
30 }
31
32
33
```

Resultados da Pesquisa Salda x

Console do Depurador x Teste (run) x

run:
pool-1-thread-1-escreveu o valor 0 na posição 0
pool-1-thread-1-escreveu o valor 1 na posição 1
pool-1-thread-1-escreveu o valor 2 na posição 2
pool-1-thread-2-escreveu o valor 6 na posição 3
pool-1-thread-2-escreveu o valor 7 na posição 4
pool-1-thread-2-escreveu o valor 8 na posição 5

Exemplo Produtor/Consumidor

- A programação envolvendo produtor e consumidor é um exemplo clássico envolvendo *threads*
- Neste tipo de aplicação, um consumidor consome aquilo que é produzido por um produtor
- Exemplos:
 - Uma *thread* faz um *download*, a outra *thread* reproduz o conteúdo ou grava o conteúdo em disco
 - Uma *thread* faz cálculos matemáticos e a outra *thread* vai exibindo os resultados parciais dos cálculos
 - ...

Exemplo Produtor/Consumidor

- Para um correto funcionamento do exemplo produtor/consumidor é necessário que um consumidor só consuma se tiver sido produzido algo
- Não adianta o consumidor consumir nada
- Pensando em *threads*, é necessário que uma *thread* representando o consumidor aguarde até que a *thread* representando o produtor produza algo
- Para isso é necessário que as *threads* estejam sincronizadas

Exemplo Produtor/Consumidor Desconsiderando a Sincronização

```
15 public interface Buffer {  
16     //Coloca um valor inteiro no buffer  
17     public void set(int valor) throws InterruptedException;  
18  
19     //Retorna um valor inteiro do buffer  
20     public int get() throws InterruptedException;  
21  
22 }
```

Exemplo Produtor/Consumidor Desconsiderando a Sincronização

```
14 public class BufferNaoSincronizado implements Buffer{
15
16     private int buffer = -1;
17
18     @Override
19     public synchronized void set(int valor) throws InterruptedException {
20         System.out.println("Produtor escreveu: " + valor);
21         buffer = valor;
22     }
23
24     @Override
25     public synchronized int get() throws InterruptedException {
26         System.out.println("Consumidor leu: " + buffer);
27         return buffer;
28     }
29
30 }
```

Exemplo Produtor/Consumidor Desconsiderando a Sincronização

```
16 public class Produtor implements Runnable{
17
18     Random random;
19     Buffer bufferCompartilhado;
20
21     public Produtor(Buffer bufferCompartilhado){
22         this.bufferCompartilhado = bufferCompartilhado;
23         random = new Random();
24     }
25
26     @Override
27     public void run(){
28         //Armazena valores de 1 a 10 no buffer
29         for(int cont=1;cont<=10;cont++){
30             try{
31                 Thread.sleep(random.nextInt(3000));
32                 bufferCompartilhado.set(cont);
33             }catch(InterruptedException e){
34                 e.printStackTrace();
35             }
36         }
37     }
38 }
39 }
```

Exemplo Produtor/Consumidor Desconsiderando a Sincronização

```
16 public class Consumidor implements Runnable {
17
18     private Random random;
19     private Buffer bufferCompartilhado;
20
21     public Consumidor(Buffer bufferCompartilhado){
22         this.bufferCompartilhado = bufferCompartilhado;
23         random = new Random();
24     }
25
26     @Override
27     public void run() {
28         // Lê 10 vezes o valor do buffer e soma os valores
29         for(int cont=1;cont<=10;cont++){
30             try{
31                 Thread.sleep(random.nextInt(3000));
32                 bufferCompartilhado.get();
33             }catch(InterruptedException exception){
34             }
35         }
36     }
37 }
38
39 }
```

Exemplo Produtor/Consumidor Desconsiderando a Sincronização

```
16 public class Principal4 {
17
18     public static void main(String[] args){
19
20         ExecutorService threadExecutor = Executors.newCachedThreadPool();
21
22         Buffer bufferCompartilhado = new BufferNaoSincronizado();
23
24         threadExecutor.execute(new Produtor(bufferCompartilhado));
25         threadExecutor.execute(new Consumidor(bufferCompartilhado));
26
27         threadExecutor.shutdown();
28     }
29
30 }
```

Localizar: exception | Anterior | Próximo

Threads.Principal4 > main

resultados da Pesquisa Salda x

Console do Depurador x Teste (run) x

run:
Consumidor leu: -1
Produtor escreveu: 1
Produtor escreveu: 2
Consumidor leu: 2
Consumidor leu: 2
Produtor escreveu: 3
Produtor escreveu: 4
Consumidor leu: 4
Produtor escreveu: 5
Consumidor leu: 5
Consumidor leu: 5
Produtor escreveu: 6
Consumidor leu: 6
Consumidor leu: 6
Produtor escreveu: 7
Produtor escreveu: 8
Consumidor leu: 8
Consumidor leu: 8
Produtor escreveu: 9
Produtor escreveu: 10

Exemplo Produtor/Consumidor Considerando a Sincronização

- O primeiro passo na sincronização de acesso ao buffer é implementar os métodos `get` e `set` como métodos `synchronized`
- Isso garante que uma *thread* obtenha o bloqueio de monitor no objeto `Buffer` antes de tentar acessar os dados do buffer, mas não assegura automaticamente que a *thread* realizará uma operação somente se o buffer estiver no estado apropriado
- **Precisa-se, portanto, que as threads esperem até que certas condições sejam verdadeiras**

Exemplo Produtor/Consumidor Considerando a Sincronização

- No caso de colocar um novo item no buffer, a condição que permite à operação prosseguir é que o buffer não esteja cheio
- No caso de consumir um item do buffer, a condição que permite que a operação prossiga é que o buffer não esteja vazio
- Se a condição em questão for verdadeira, a operação pode prosseguir; se falsa, a *thread* deve esperar até que ela se torne verdadeira

Exemplo Produtor/Consumidor Considerando a Sincronização

- Quando um thread estiver esperando uma condição, ela é removida da disputa do processador é colocada no estado de espera e o bloqueio que ela mantém é liberado
- Para uma thread prosseguir ou esperar de acordo com determinadas condições, pode-se utilizar os métodos `wait()`, `notify()` e `notifyAll()`
- Os métodos `wait()`, `notify()` e `notifyAll()`, que são da classe `Object` e que, portanto, são herdados por todas as outras classes

Exemplo Produtor/Consumidor Considerando a Sincronização

- Uma thread pode chamar o método `wait()` no objeto `synchronized` → a *thread* permanece no estado de espera enquanto outras *threads* tentam realizar suas tarefas envolvendo os objetos `synchronized`
- Quando uma *thread* que executa uma instrução em um ambiente `synchronized` completa ou satisfaz a condição que outra thread pode estar esperando, ela pode chamar o método `notify()` (no objeto `synchronized`) para permitir que uma *thread* em espera transite para o estado executável novamente

Exemplo Produtor/Consumidor Considerando a Sincronização

- Nesse ponto, a *thread* que transitou do estado de espera para o estado executável pode tentar readquirir o bloqueio de monitor no objeto
- Se uma *thread* chamar `notifyAll()` em um objeto `synchronized`, então todas as *threads* que esperam o bloqueio de monitor se tornarão elegíveis para readquirir o bloqueio → todas elas transitarão para o estado executável

Exemplo Produtor/Consumidor Considerando a Sincronização

```
12 public class BufferSincronizado implements Buffer{
13
14     private int buffer;
15     private boolean ocupado;
16
17     public BufferSincronizado(){
18         buffer = -1;
19         ocupado = false;
20     }
21
22     @Override
23     public synchronized void set(int valor) throws InterruptedException {
24         while(ocupado){
25             System.out.println("Produtor tentou escrever");
26             wait();
27         }
28         buffer = valor;
29         ocupado = true;
30
31         System.out.println("Produtor escreveu " + buffer);
32         notifyAll();
33     }
34
35     @Override
36     public synchronized int get() throws InterruptedException {
37         while(!ocupado){
38             System.out.println("Consumidor tentou ler");
39             wait();
40         }
41         ocupado = false;
42         System.out.println("Consumidor leu " + buffer);
43
44         notifyAll();
45
46         return buffer;
47     }
48 }
49 }
```

Exemplo Produtor/Consumidor Considerando a Sincronização

```
15 public class Principals {  
16  
17     public static void main(String[] args){  
18  
19         ExecutorService threadExecutor = Executors.newCachedThreadPool();  
20  
21         Buffer bufferCompartilhado = new BufferSincronizado();  
22  
23         threadExecutor.execute(new Produtor(bufferCompartilhado));  
24         threadExecutor.execute(new Consumidor(bufferCompartilhado));  
25  
26         threadExecutor.shutdown();  
27     }  
28 }  
29  
30
```

Resultados da Pesquisa Salda X

Console do Depurador X Teste (run) X

run:
Produtor escreveu 1
Produtor tentou escrever
Consumidor leu 1
Produtor escreveu 2
Produtor tentou escrever
Consumidor leu 2
Produtor escreveu 3
Produtor tentou escrever
Consumidor leu 3
Produtor escreveu 4
Produtor tentou escrever
Consumidor leu 4
Produtor escreveu 5
Consumidor leu 5
Produtor escreveu 6
Consumidor leu 6
Consumidor tentou ler
Produtor escreveu 7
Consumidor leu 7
Consumidor tentou ler
Produtor escreveu 8
Consumidor leu 8
Produtor escreveu 9
Produtor tentou escrever
Consumidor leu 9
Produtor escreveu 10
Consumidor leu 10

Multithreading com GUI

- Os aplicativos Swing apresentam uma série excepcional de desafios para a programação de múltiplas *threads*
- Todos os aplicativos Swing têm uma única *thread*, chamada **thread de despacho de eventos** para tratar interações com os componentes GUI do aplicativo
- As interações típicas são atualizar componentes GUI ou processar ações de usuário, como cliques de mouse
- Todas as tarefas que exigem interações com a GUI de um aplicativo são colocadas em uma fila de eventos e executadas em sequência pela *thread* de despacho de eventos

Multithreading com GUI

- Normalmente, é suficiente realizar cálculos simples na *thread* de despacho de eventos na sequência com manipulações de componentes GUI
- Se um aplicativo deve realizar um cálculo longo em resposta a uma interação de interface com o usuário, a *thread* de despacho de eventos não pode ocupar-se de outras tarefas na fila de evento enquanto a *thread* estiver presa nesse cálculo

Multithreading com GUI

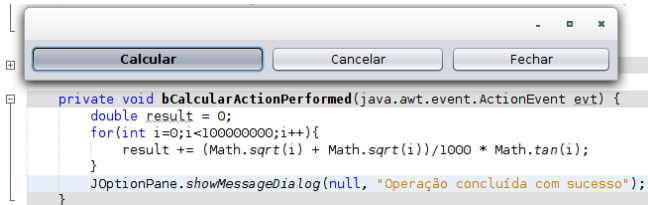
- Isso faz os componentes GUI tornarem-se indiferentes
- É preferível tratar um cálculo demorado em uma *thread* separada, liberando a *thread* de despacho de eventos para continuar o gerenciamento de outras interações de GUI
- Às vezes também é interessante usar as *threads* para ter um efeito visual mais bonito

Cálculo Demorado

- Um exemplo do benefício do uso de *Threads* seria realizar um processamento “demorado” ao clicar em um determinado botão (no nosso exemplo é o botão Calcular)
- Ao realizar o cálculo sem considerar uma *thread*, todos os botões da interface ficam parados, além do fato de não dar opção ao usuário de cancelar uma operação
- Ao considerar uma *thread*, pode-se tanto permitir que o usuário continue interagindo com a interface quanto permitir que este interrompa a execução quando quiser

Cálculo Demorado

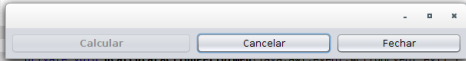
- Sem o uso de threads, o código fica da seguinte forma (demais botões travados):



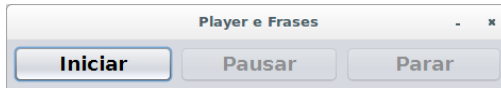
Cálculo Demorado

- Com threads, o código fica da seguinte forma:

```
19 Thread calcular;  
20  
21 public TravenetoBotao1() {  
22     initComponents();  
23 }  
24  
25  
26  
83  
84  
85  
86 calcular = new Thread(new Runnable(){  
87     @Override  
88     public void run() {  
89         bCalcular.setEnabled(false);  
90         double result = 0;  
91         for(int i=0;i<100000000;i++){  
92             result += (Math.sqrt(i) + Math.sqrt(i))/1000 * Math.tan(i);  
93         }  
94         JOptionPane.showMessageDialog(null, "Operação concluída com sucesso");  
95         bCalcular.setEnabled(true);  
96     }  
97 });  
98 calcular.start();  
99  
100  
101  
102 private void bCancelarActionPerformed(java.awt.event.ActionEvent evt) {  
103     calcular.stop();  
104     bCalcular.setEnabled(true);  
105 }  
106  
107 private void bFecharActionPerformed(java.awt.event.ActionEvent evt) {  
108     System.exit(0);  
109 }
```



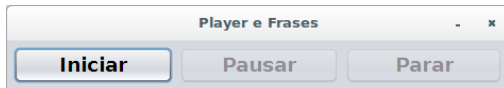
Simulando um Player



Campos e Construtor

```
public class TestePlayer extends javax.swing.JFrame {  
    Thread threadPlayer;  
  
    public TestePlayer() {  
        initComponents();  
        bParar.setEnabled(false);  
        bPausarContinuar.setEnabled(false);  
        bPausarContinuar.setText("Pausar");  
    }  
}
```

Simulando um Player



```
private void bIniciarActionPerformed(java.awt.event.ActionEvent evt) {  
    threadPlayer = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            for(int i=0;i<10000000;i++){  
                System.out.println("Frase " + (i+1));  
            }  
        }  
    });  
    threadPlayer.start();  
  
    bPausarContinuar.setText("Pausar");  
    bPausarContinuar.setEnabled(true);  
    bIniciar.setEnabled(false);  
    bParar.setEnabled(true);  
}
```

Simulando um Player



```
private void bPausarContinuarActionPerformed(java.awt.event.ActionEvent evt) {  
    if(bPausarContinuar.getText().equals("Pausar")){  
        threadPlayer.suspend();  
        bPausarContinuar.setText("Continuar");  
    }else{  
        threadPlayer.resume();  
        bPausarContinuar.setText("Pausar");  
    }  
}
```

Simulando um Player



```
private void bPararActionPerformed(java.awt.event.ActionEvent evt) {  
    threadPlayer.stop();  
    bPausarContinuar.setEnabled(false);  
    bParar.setEnabled(false);  
    bIniciar.setEnabled(true);  
}
```

Material Complementar

- Java - Multithreading

http://www.tutorialspoint.com/java/java_multithreading.htm

- Lesson: Concurrency

<https://docs.oracle.com/javase/tutorial/essential/concurrency/>

- Programação Concorrente e Threads

[https://www.caelum.com.br/apostila-java-orientacao-objetos/
programacao-concorrente-e-threads/](https://www.caelum.com.br/apostila-java-orientacao-objetos/programacao-concorrente-e-threads/)

Imagem do Dia



Programação Orientada a Objetos

<http://lives.ufms.br/moodle/>

Rafael Geraldeli Rossi
rafael.g.rossi@ufms.br

Slides baseados em [Deitel and Deitel, 2010]

Referências Bibliográficas I



Deitel, P. and Deitel, H. (2010).

Java: How to Program.

How to program series. Pearson Prentice Hall, 8th edition.