Threads

Fundamentos de várias threads

- Há dois tipos distintos de multitarefas: baseada em processos e baseada em Threads.
- É importante entender a diferença entre os dois. Um processo é, em essência, um programa que está sendo executado.
- Portanto a multitarefa *baseada em processos* é o recurso que permite que o computador execute dois ou mais programas ao mesmo tempo.
- Por exemplo, é a multitarefa baseada em processos que nos permite executar o compilador Java ao mesmo tempo em que estamos usando um editor de texto ou navegador de Internet.

Fundamentos de várias threads

- Na multitarefa baseada em processo, um programa é a menor unidade de código que pode ser despachada pelo agendador.
- Em um ambiente multitarefa *baseado em threads*, a thread é a menor unidade de código que pode ser despachada.
- Ou seja, o mesmo programa pode executar duas ou mais tarefas ao mesmo tempo. Por exemplo, um editor de texto pode formatar texto ao mesmo tempo em que está imprimindo, contanto que essas duas ações estejam sendo executadas por duas threads separadas.

Fundamentos de várias threads

- Uma vantagem importante do uso de várias threads é que ele permite a criação de programas muito eficientes, porque podemos utilizar o tempo ocioso que está presente em quase todos os programas.
- Usando várias threads, o programa pode executar outra tarefa durante seu tempo ocioso.
- É importante entender que os recursos multithread Java funcionam tanto no sistema de multiprocessadores e no sistema de multicore.

Fundamentos de várias threads

- Em um sistema single-core, duas ou mais threads não são executadas realmente ao mesmo tempo, o tempo ocioso da CPU é que é utilizado.
- No entanto, em sistemas multiprocessadores/multicore, é possível duas ou mais threads serem executadas simultaneamente.
- Em muitos casos, isso pode melhorar ainda mais a eficiência do programa e aumentar a velocidade de certas operações.

A classe Thread e a interface Runnable

- O sistema de várias threads de Java tem como base a classe **Thread** e a interface que a acompanha, **Runnable.**
- As duas estão empacotadas em **java.lang. Thread** encapsula uma thread de execução. Para criar uma nova thread, o programa deve estender **Thread** ou implementar a interface **Runnable**.
- A classe **Thread** define vários métodos que ajudam a gerenciar as threads.
- Todos os processos têm pelo menos uma thread de execução, que geralmente é chamada de *thread principal.*

Criando uma Thread

- Você pode criar uma thread instanciando um objeto de tipo **Thread**.
- Na maioria dos exemplos que veremos, usaremos a abordagem que implementa **Runnable**. As duas abordagens usam a classe **Thread** para instanciar, acessar e controlar a thread. A única diferença, é como uma classe para threads é criada.
- A interface **Runnable** concebe uma unidade de código executado. Você pode construir uma thread em qualquer objeto que implementar a interface **Runnable**.

Criando uma Thread

- **Runnable** só define um método, chamado **run()**. Dentro de **run()**, você definirá o código que constitui a nova thread. É importante saber que **run()** pode chamar outros métodos, usar outras classes e declarar variáveis da mesma forma que a thread principal.
- A única diferença é que **run()** estabelece o ponto de entrada de uma thread de execução concorrente dentro do programa.
- Essa thread terminará quando **run()** retornar.
- Exemplo da pg. 370, 374 e 377.
- Exercício da pg. 375.

Determinando quando uma thread termina

- Costuma ser útil saber quando uma thread terminou. Nos exemplos anteriores, a título de ilustração foi útil manter a thread principal ativa até as outras threads terminarem.
- Felizmente, Thread fornece dois meios pelos quais você pode determinar se uma thread terminou.
- O primeiro é chamar o método é isAlive() na thread.
- exemplo da pg. 380.
- Veja um programa que usa **join()** para assegurar que a thread principal seja a última a terminar.
- exemplo da pg. 381.

Sincronização

- Quando várias threads são usadas, às vezes é necessário coordenar as atividades de duas ou mais.
- O processo que faz tudo isso se chama sincronização.
- Essencial para a sincronização em Java é o conceito de monitor, que controla o acesso a um objeto.
- Um monitor funciona implementado o conceito de bloqueio.
- Quando um objeto é bloqueado por uma thread, nenhuma outra thread pode ganhar acesso a ele.

Sincronização

- Quando a thread termina, o objeto é desbloqueado e fica disponível para ser usado em outra thread.
- Todos os objetos em Java têm um monitor. Esse recurso existe dentro da própria linguagem Java.
- Logo, todos os objetos podem ser sincronizados a sincronização é suportada pela palavra-chave **synchronized** e alguns métodos bem definidos que todos os objetos tem.

Usando métodos sincronizados

- Você pode sincronizar o acesso a um método modificando-o com a palavra **synchronized**.
- Quando esse método for chamado a thread chamadora entrará no monitor do objeto, que então será bloqueado.
- Enquanto ele estiver bloqueado, nenhuma outra thread poderá entrar no método ou em qualquer outro método sincronizado definido pela classe do objeto.
- Quando a thread retornar do método, o monitor desbloqueará o objeto, permitindo que ele seja usado pela próxima thread. Logo, a sincronização é obtida sem que você faça praticamente nenhum esforço de programação.
- Exemplo da pg. 386.

A instrução Synchronized

- Embora a criação de métodos **synchronized** dentro das classes que criamos seja um meio fácil e eficaz de obter sincronização, ele não funciona em todos os casos.
- Por exemplo, podemos querer sincronizar o acesso a algum método que não seja modificado por **synchronized**.
- Isso pode ocorrer por querermos usar uma classe que não foi criada por nós, e sim por terceiros, e não termos acesso ao código-fonte.
- Eemplo da pg. 390.

Suspendendo, retornando e encerrando threads

- Às vezes é útil suspender a execução de uma thread.
- Por exemplo, uma thread separada pode ser usada para exibir a hora do dia. Se o usuário não quiser um relógio, sua thread pode ser suspensa.
- O mecanismo de suspensão, encerramento e retomada de threads difere entre as versões antigas de Java e as versões mais modernas, a partir de Java 2.
- Antes de Java 2, os programas usavam suspend(), resume() e stop(), que são métodos definidos por Thread, por pausar, reiniciar e encerrar a execução de uma thread.

Suspendendo, retornando e encerrando threads

- Embora esses métodos pareçam uma abordagem perfeitamente sensata e conveniente para o gerenciamento da execução das threads, eles não devem mais ser usados.
- O método suspend() da classe Thread foi substituido em Java 2.
- Isso foi feito porque as vezes suspend() pode causar problemas sérios que envolvem deadlock.
- O método resume() também foi substituido; ele não causa problemas, mas não pode ser usado sem o método suspend() como complemento.

Suspendendo, retornando e encerrando threads

- O método stop() da classe Thread também foi substituido em Java 2.
- A razão é que esse método às vezes também pode causar problemas sérios.
- A thread deve ser projetada de modo que o método run() verifique periodicamente se ela deve suspender, retornar ou encerrar sua própria execução.
- Exemplo da pg. 399.