Colocando em Prática os Conceitos

Primeiro Exemplo

- Programa da aula passada
  - Demonstra técnicas básicas de multithreading
    - Cria uma classe derivada de **`Thread`**
    - Usa o método **`sleep`**
  - Visão geral
    - O programa cria 4 threads, que dormem por uma quantidade aleatória de tempo
    - Depois que acabam de dormir, escrevem seu nome

Primeiro Exemplo

- O Programa tem duas classes
  - **`PrintThread`**
    - Derivada de **`Thread`**
    - Variável de instância **`sleepTime`**
  - **`ThreadTester`**
    - Cria 4 **`objetos PrintThread`**
- Ambas estão no mesmo arquivo **`ThreadTester.java`** (abrir, olhar, executar e ver o resultado)

```
1  // ThreadTester.java
2  // Mostra múltiplas threads escrevendo em diferentes intervalos.
3
4  public class ThreadTester {
5     public static void main( String args[] )
6     {
7        PrintThread thread1, thread2, thread3, thread4;
8
9        thread1 = new PrintThread( "thread1" );
10       thread2 = new PrintThread( "thread2" );
11       thread3 = new PrintThread( "thread3" );
12       thread4 = new PrintThread( "thread4" );
13
14       System.err.println( "\nIniciando threads" );
15
16       thread1.start();
17       thread2.start();
18       thread3.start();
19       thread4.start();
20
21       System.err.println( "Threads started\n" );
22    }
23 }
24
25 class PrintThread extends Thread {
26    private int sleepTime;
27
28    // construtor PrintThread atribui nome a thread chamando
29    // construtor Thread
```

```
30   public PrintThread( String name )
31   {
32      super( name );
33
34      // dorme entre 0 e 5 segundos
35      sleepTime = (int) ( Math.random() * 5000 );
36
37      System.err.println( "Name: " + getName() +
38                          ";  dorme: " + sleepTime );
39   }
40
41   // executa a thread
42   public void run()
43   {
44      // coloca thread para dormir por um intervalo aleatório
45      try {
46         System.err.println( getName() + " indo dormir" );
47         Thread.sleep( sleepTime );
48      }
49      catch ( InterruptedException exception ) {
50         System.err.println( exception.toString() );
51      }
52
53      // escreve nome da thread
54      System.err.println( getName() + " terminou de dormir" );
55   }
56}
```

Chamar this.getName () tem o mesmo efeito

```
Name: thread1;  dorme: 1653
Name: thread2;  dorme: 2910
Name: thread3;  dorme: 4436
Name: thread4;  dorme: 201

Iniciando threads
Threads started

thread1 indo dormir
thread2 indo dormir
thread3 indo dormir
thread4 indo dormir
thread4 terminou de dormir
thread1 terminou de dormir
thread2 terminou de dormir
thread3 terminou de dormir
```

```
Name: thread1;  dorme: 3876
Name: thread2;  dorme: 64
Name: thread3;  dorme: 1752
Name: thread4;  dorme: 3120

Iniciando threads
Threads started

thread2 indo dormir
thread4 indo dormir
thread1 indo dormir
thread3 indo dormir
thread2 terminou de dormir
thread3 terminou de dormir
thread4 terminou de dormir
thread1 terminou de dormir
```

Exemplo de Saída do Programa

Exercício passado em sala

- Modificar o programa anterior para criar 5 threads em vez de 4
  - Configuradas com diferentes prioridades
    - Método setPriority
    - Valores de 1 a 10
- Cada thread, ao ser disparada, dorme por um período aleatório e ao acordar escreve:
  - Seu nome, seu ID e o tempo durante o qual dormiu

Exercício passado em sala

- Obs.: quem fez vai observar que as prioridades não farão diferença na execução. Por que?
  - As threads executam tarefas muito simples, que não consomem tempo de processador

Exercício passado em sala

- Na JVM rodando em Windows/Linux, as threads são escalonadas de forma preemptiva seguindo a metodologia "*round-robin*"
  - Isso quer dizer que o escalonador pode pausá-las e dar tempo para outra thread ser executada
- O tempo que cada thread recebe para processar se dá conforme a **prioridade** que ela possui, ou seja, threads com prioridade mais alta **ganham mais** tempo para processar e são escalonadas **com mais frequência** do que as outras
- Para observarmos alguma diferença entre as threads com diferentes prioridades, podemos incluir uma **operação** no método run (por ex loop para incrementar um contador – ver ThreadTester2)

Segundo Exemplo

- Abra o arquivo OlaThread.java e observe seu código
- Compile e execute o programa várias vezes, e observe os resultados impressos.

Segundo Exemplo

```
class Ola extends Thread {
   String msg;
    //--construtor
   public Ola(String m) {
      msg = m;    }
   //--metodo executado pela thread ao ser iniciada
   public void run() {
      System.out.println(msg);   }
}//fim da classe thread
//--classe com o metodo main
class OlaThread {
   static final int N = 10;
   public static void main (String[] args) {
      //--cria um vetor de threads
      Thread[] threads = new Thread[N];
```

Segundo Exemplo

```
      //Cria threads usando a classe que estende Thread
   for (int i=0; i<threads.length; i++) {
      final String m = "Ola da thread " + i;
      threads[i] = new Ola(m);
   }
   //Inicia as threads
   for (int i=0; i<threads.length; i++) {
      threads[i].start();
   }
   //PASSO extra: espera pelo termino de todas as threads
   for (int i=0; i<threads.length; i++) {
      try { threads[i].join(); }
      catch (InterruptedException e) { return; }
   }
   System.out.println("Terminou");
}}
```

Observe o que acontece se você remover o passo extra

Usando a Interface `Runnabel` para criar e lançar Threads

- Programa para criar e executar três threads:
  - A primeira thread escreve a letra *a* 100 vezes.
  - A segunda thread escreve a letra *b* 100 vezes.
  - A terceira thread escreve os números inteiros de 1 até 100.

TaskThreadDemo

13

Usando a Interface `Runnabel` para criar e lançar Threads

- Segundo Exemplo - Abra o arquivo HelloThread.java, leia o programa e tente entender o que ele faz
  - Compile o programa
  - Execute o programa várias vezes e observe os resultados impressos na tela
  - Há mudanças na ordem de execução das threads? Se sim, por que isso ocorre?

Usando o método estático sleep(milliseconds)

- O método sleep(long mills) coloca a thread para dormir pelo intervalo de tempo especificado (em milisegundos)
- Por exemplo, veja o que ocorre se você modificar o código das linhas abaixo (método run da classe interna PrintNum de TaskThreadDemo.java) como segue:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    try {
      if (i >= 50) Thread.sleep(2000);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Toda vez que um número (>= 50) for escrito, a thread print100 é colocada para dormir por 2000 ms.

TaskThreadDemoSleep

15

Usando o método estático sleep(milliseconds)

- Execute a classe TaskThreadDemoSleep
- Observou alguma diferença em relação a execução da classe anterior? (TaskThreadDemo)
- Alguém saberia porque?

Aumente o valor de 2000 para 20000 por exemplo

O método estático yield()

- Você pode usar o método yield() para liberar temporariamente o processador para outras threads
- Modifique o mesmo Código do exemplo anterior do programa TaskThreadDemo.java como segue:

```
public void run() {
  for (int i = 1; i <= lastNum; i++) {
    System.out.print(" " + i);
    Thread.yield();
  }
}
```

- Toda vez que um número é escrito, a thread print100 cede a vez. Então, os números são impressos após os caracteres.

17

sleep vs. yield

Há uma grande diferença entre os dois métodos

- Chamar o método sleep coloca a thread atualmente em execução no estado **bloqueada**
- Chamar o método yield não coloca a thread chamadora no estado bloqueada
  - Ele simplesmente deixa o escalonador entrar em ação e escolher outra thread para executar.
  - Pode acontecer que a thread chamadora seja selecionada para ser executada novamente -> isso acontece quando ela tem uma prioridade maior do que todos as outras threads executáveis.

18

O métod join()

- Você pode usar o método join ( ) para forçar uma thread a esperar por outra thread para finalizar
- Por ex, modifique o código das mesmas linhas do exemplo anterior como segue:

```
public void run(){
Thread thread4 = new Thread(new PrintChar('c', 60));
thread4.start();
try {
   for (int i = 1; i < lastNum; i++){
         System.out.print(" " + i);
         if (i == 50) thread4.join();
      }
   }
   catch (InterruptedException ex) {
      }
}
```

Os números depois de 50 são escritos somente depois que a thread4 tiver terminado

TaskThreadDemoJoin

O métod join()

- Como segundo exemplo, abra novamente o arquivo HelloThread.java,
  - Descomente as linhas correspondentes ao PASSO extra e compile o programa novamente
- Execute o programa várias vezes e observe os resultados impressos na tela. Qual alteração na execução da aplicação pode ser observada e por que ela ocorre?

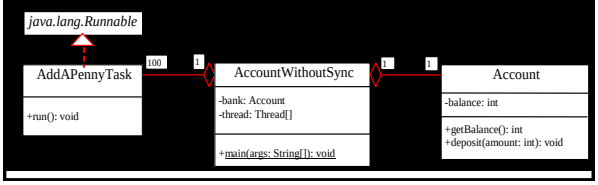
Sincronização de Threads em Java

Sincronização de Threads

- Como vimos, um recurso compartilhado pode ser corrompido se for acessado simultaneamente por múltiplas threads
  - Por exemplo, duas threads não sincronizadas acessando a mesma conta bancária podem causar conflitos

Ilustrando Conflito sobre um Recurso

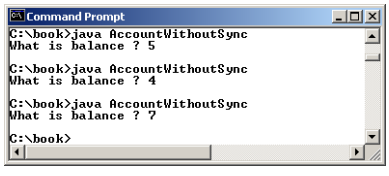
- Objetivo: escrever um programa que demonstre o problema do conflito sobre recursos
- Suponha que você crie e inicie cem threads, cada uma das quais adicionando um centavo a uma conta
- Suponha que a conta esteja inicialmente vazia



Ilustrando Conflito sobre um Recurso

AccountWithoutSync2

Execute o programa e veja o resultado!



A classe Account

```
   // classe interna representando um objeto conta (Account)
private static class Account {
    private int balance = 0;
    public int getBalance() {   return balance;   }
    public void deposit(int amount) {
        int newBalance = balance + amount;
// atraso deliberadamene adicionado para ampliar o problema de corrupção de dados
      try {
        Thread.sleep(1);
      }   catch (InterruptedException ex) {     }
      balance = newBalance;
    }  } }
```

A classe AddAPennyThread

```
// uma thread para adicionar um centavo (penny) a conta
private static class AddAPennyThread implements Runnable
{
   public void run() {
     account.deposit(1);
   }
 }
```

Atenção para o escopo do objeto account

A classe AccountWithoutSync

```
import java.util.concurrent.*;
public class AccountWithoutSync2 {

  private static Account account = new Account();
  public static void main(String[] args) {
//Cria vetor de 100 threads
      Thread[] threads = new Thread[100];
//Instancia cada thread passando o objeto Runnable do tipo AddAPennyThread
      for (int i=0; i<threads.length; i++) {
          threads[i] = new Thread(new AddAPennyThread());      }

//Inicia as threads
      for (int i=0; i<threads.length; i++) {
          threads[i].start();      }
```

A classe AccountWithoutSync

```
// espera até todas as threads terem acabado

      for (int i=0; i<threads.length; i++) {
        try { threads[i].join(); }
        catch (InterruptedException e) { return; }
      } //escreve o saldo ao final
System.out.println("What is balance ? " + account.getBalance());

  }
```

Execute e veja o resultado

Condição de Corrida

- O que causou o erro no exemplo? Aqui está um cenário possível:

Step	balance	Task 1	Task 2
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

Lembrando que balance é uma variável (recurso) compartilhada pelas threads, pois é atributo de Account

Condição de Corrida

- O efeito desse cenário é que a Tarefa 1 não fez nada, porque na Etapa 4 a Tarefa 2 substitui o resultado da Tarefa 1
- Obviamente, o problema é que a Tarefa 1 e a Tarefa 2 estão acessando um recurso comum, dessa forma gerando conflitos
  - Problema comum conhecido como condição de corrida
- Diz-se que uma classe é **thread-safe** se um objeto da classe não causar uma condição de corrida na presença de múltiplas threads
- Conforme demonstrado no exemplo, a classe Account **não é thread-safe**.

A palavra-chave `synchronized`

- Para evitar conflitos sobre recursos, é necessário evitar que mais de uma thread entrem simultaneamente em uma determinada parte do programa -> conhecida como **região crítica**
- Qual a região crítica no exemplo?

o método deposit

A palavra-chave `synchronized`

- Há várias maneiras de corrigir o problema no exemplo
- Uma possível abordagem é tornar a classe Account thread-safe dicionando a palavra-chave `synchronized` no método `deposit`
  - sincroniza o método de modo que **apenas uma thread** possa acessa-lo por vez:

```
public synchronized void deposit(double
amount)
```

Sincronizando métodos de instância e métodos estáticos

- Um método sincronizado **adquire um bloqueio** (lock) antes de ser executado.
- No caso de um método de instância, o bloqueio está no **objeto** para o qual o método foi invocado.
- No caso de um método estático, o bloqueio está na **classe**.
- Se uma thread invoca um método de instância sincronizado em um objeto, o bloqueio desse objeto é adquirido primeiro, então o método é executado e, finalmente, o bloqueio é liberado
  - Outra thread invocando o mesmo método desse objeto é bloqueada até o bloqueio ser liberado.

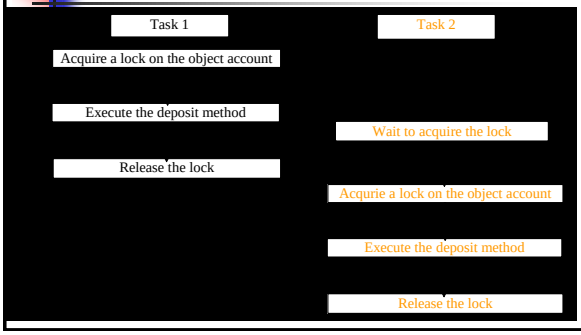
Sincronizando métodos de instância e métodos estáticos

- Se uma thread invoca um método estático sincronizado em um objeto, o bloqueio da respectiva classe é primeiro adquirido, então o método é executado e, finalmente, o bloqueio é liberado
  - Outra thread invocando o mesmo método dessa classe é bloqueada até o bloqueio ser liberado

Sincronizando métodos de instância e métodos estáticos

- Com o método `deposit` sendo sincronizado, o cenário anterior não ocorre.
- Se a Tarefa 2 começar a entrar no método e a Tarefa 1 já estiver executando o método, a Tarefa 2 é bloqueada até que a Tarefa 1 termine a execução do método.
- Faça a alteração sugerida no código e observe o resultado!

Sincronizando métodos de instância e métodos estáticos



Blocos sincronizados

- Invocar um método de instância sincronizada de um objeto adquire um bloqueio **sobre o objeto** e invocar um método estático sincronizado de uma classe adquire um bloqueio **sobre a classe**
- Um comando `synchronized` pode ser usado para adquirir um bloqueio em qualquer objeto, não apenas **neste** objeto, ao executar um bloco de código em um método
  - Este bloco é referido como um bloco sincronizado (`synchronized`)

Blocos sincronizados

- A forma geral de um bloco sincronizado é a seguinte:

```
synchronized (expr) {
  statements;
}
```

- A expressão `expr` referencia um dado objeto a ser sincronizado
- Se o objeto já estiver sendo usado por outra thread (que detém o lock), a thread em questão fica bloqueada até o lock ser liberado
- Quando um lock for obtido sobre o objeto, os comandos no bloco sincronizado são executados, e a seguir o lock é liberado

Blocos X métodos sincronizados

- Qualquer método de instância sincronizado pode ser convertido em um bloco sincronizado
- Suponha o seguinte método de instância sincronizado:

```
public synchronized void xMethod() {
  // method body
}
```

- Este método é equivalente a

```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

MAIS SOBRE ISSO NA AULA QUE VEM!

45