Mecanismos de Sincronização em Java

Praticando os conceitos da aula passada

Exemplo de sincronização estática

- Abrir os arquivos: Table.java, MyThread1.java e TestSynchronization4.java
- Executar a classe MyThread1

Exemplo de sincronização estática

```
class Table{
  synchronized static void printTable(int n){
  for(int i=1;i<=10;i++){
    System.out.println(n*i);
    try{
      Thread.sleep(400);
    }catch(Exception e){}
  }
 }
}
```

Exemplo de sincronização estática

```
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
```

Exemplo de sincronização estática

```
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
} }
  class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
} }
```

Exemplo de sincronização estática

```
public class TestSynchronization4{
  public static void main(String t[]){
      MyThread1 t1=new MyThread1();
      MyThread2 t2=new MyThread2();
      MyThread3 t3=new MyThread3();
      MyThread4 t4=new MyThread4();
      t1.start();
      t2.start();
      t3.start();
      t4.start();  } }
```

Executar o programa e ver a saída!

Mais exemplos

Abra o arquivo S.java e siga o roteiro abaixo.

- Leia o programa para entender o que ele faz
  - Criadas duas threads, cada uma incrementa de um em um ate 10.000.000
- Qual é a seção crítica do código? Qual é a saída esperada para o programa (valor final de s)?
- Compile o programa, execute-o várias vezes e observe os resultados impressos na tela. Os valores impressos foram sempre o valor esperado?
- A seguir, ainda no arquivo S.java, comente as linhas 15-17; e descomente as linhas 19-23.
- Observe o uso de synchronized em Java.
- Compile o programa, execute-o várias vezes e observe os resultados impressos na tela. Os valores impressos foram sempre o valor esperado? Por que?

Observação importante sobre a aula passada: métodos wait e notify

- Uma thread que chama wait() em um objeto torna-se inativa (bloqueada) até que outra thread chame notify() naquele objeto
  - Liberando o acesso ao objeto (libera o lock que ela detém no objeto do monitor)
  - ou seja, permite a execução de outra thread no objeto
- Para poder chamar ou wait() ou notify, a thread chamadora precisa primeiro obter o lock daquele objeto
  - Ou seja, a thread chamadora TEM que chamar wait() ou notify() DE DENTRO de um bloco sincronizado. ISSO É OBRIGATÓRIO!
  - Uma thread não pode chamar wait(), notify() ou notifyAll() sem ter adquirido o lock sobre o objeto a partir dos quais tais métodos são chamados
  - Se isso for feito, será lançada uma exceção do tipo IllegalMonitorStateException
- Então não teria como no exemplo da aula passada chamar o notify sem ter sido de dentro do bloco sincronizado (método unlock)

Observação importante sobre a aula passada: métodos wait e notify

- Uma vez que uma thread é despertada, ela não pode sair da chamada wait () até que a thread chamando notify () tenha deixado o seu bloco sincronizado
- Em outras palavras: a thread despertada tem que obter de novo o lock do objeto monitor antes que ela possa sair da chamada wait (), porque a chamada wait () está dentro de um bloco sincronizado.
- Se várias threads forem despertadas usando notifyAll (), apenas uma de cada vez pode sair do método wait (), uma vez que cada thread deve obter o bloqueio no objeto do monitor por vez antes de sair de wait ()

Um Lock Simples

- A classe Counter escrita usando um Lock em vez de um bloco synchronized:

```
public class Counter{
  private Lock lock = new Lock();
  private int count = 0;
  public int inc(){
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
  }}
```

O método lock() **bloqueia a instância de Lock** de modo que todas as threads chamando lock() são bloqueadas até unlock() ser executado

Um Lock Simples

- Implementação simples da classe Lock:

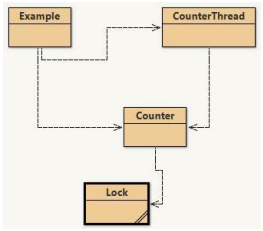
```
public class Lock{
  private boolean isLocked = false;
  public synchronized void lock()   throws InterruptedException{
    while(isLocked){
      wait();
    }
    isLocked = true;
  }
  public synchronized void unlock(){
    isLocked = false;
    notify();
  }}
```

Locks em Java

- Abra os arquivos:

Lock, Counter, CounterThread e Example

- Execute Example e observe o resultado
- Você pode criar mais threads (novas instâncias da classe CounterThread, inicializa-las e ver o novo resultado



Outro Exemplo

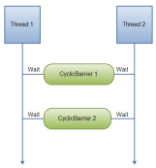
- Criar uma classe chamada Banheiro para representar um banheiro com apenas um vaso sanitário e apenas uma pia
  - A classe deve ter um método representando o uso do vaso e outro representando o uso da pia
- A seguir, criar uma classe chamada Aluno do tipo Thread
  - A classe possui uma variavel estática que é um objeto do tipo Banheiro
  - Ao ser inciada a thread, deve ser feita uma chamada ao método para uso do vaso da classe Banheiro, depois a thread dorme por algum tempo e a seguir feita uma chamada ao método para uso da pia da classe Banheiro.
- Por fim, criar uma classe executável contendo um laço para criar 10 objetos da classe aluno, iniciando as respectivas threads.

Outro Exemplo

- Olha as classes Aluno, Banheiro e ExemploMonitor
- Execute a classe ExemploMonitor e veja o resultado
  - Ilustra claramente as thdreas em FILA esperando para entrar na região crítica
    - Ou seja, estacionados na chamada a wait ()

Barreiras: Exemplo de código

```
//criando comandos/ações de barreira
Runnable barrier1Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 1 executed ");
    }
};
Runnable barrier2Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 2 executed ");
    }
};
```



Barreiras: Exemplo de código

```
//criando duas barreiras com ações associadas
CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

//criando duas threads do tipo CyclicBarrierRunnable
CyclicBarrierRunnable barrierRunnable1 =
        new CyclicBarrierRunnable(barrier1, barrier2); //essa classe será declarada
                                                       // a seguir
CyclicBarrierRunnable barrierRunnable2 =
        new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();
new Thread(barrierRunnable2).start();
```

Barreiras

- A seguir a classe CyclicBarrierRunnable: representa uma thread sincronizada por 2 barreiras

```
public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable( CyclicBarrier barrier1, CyclicBarrier barrier2) {

        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }
```

Barreiras

```
public void run() {
    try {
        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() + " waiting at barrier 1");
        this.barrier1.await();

        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() +  " waiting at barrier 2");
        this.barrier2.await();

        System.out.println(Thread.currentThread().getName() +     " done!");
    } catch (InterruptedException e) {
        e.printStackTrace();
} catch (BrokenBarrierException e) {
        e.printStackTrace();
    }   } }
```

Barreiras

- Olhe os arquivos CyclicBarrierRunnable e TestBarrier
- Execute o programa TestBarrier
- A seguir mostramos a saída do console para uma execução do código acima. Observe que a sequência em que as threads podem escrever na tela pode variar de execução para execução. Às vezes, Thread-0 imprime primeiro, às vezes Thread-1 imprime primeiro etc.

```
Thread-0 waiting at barrier 1
Thread-1 waiting at barrier 1
BarrierAction 1 executed
Thread-1 waiting at barrier 2
Thread-0 waiting at barrier 2
BarrierAction 2 executed
Thread-0 done!
Thread-1 done!
```

CountDownLatch (Trancas)

- Similar a barreiras, a diferença é a condição para liberação:
  - não é o número de threads que estão esperando, mas sim quando um contador específico chega a zero
- threads que executarem o wait após o contador já ter atingido o zero são liberadas automaticamente

CountDownLatch (Trancas)

- Exemplo: Após o objeto Decrementer ter chamado countDown() 3 vezes no CountDownLatch, o objeto Waiter aguardando é liberado a partir da chamada de await()

```
CountDownLatch latch = new CountDownLatch(3);
Waiter      waiter      = new Waiter(latch);
Decrementer decrementer = new Decrementer(latch);
new Thread(waiter).start();
new Thread(decrementer).start();
Thread.sleep(4000);
```

CountDownLatch (Trancas)

- Olhe os arquivos Decrementer, Waiter e TestLatch
- Execute o programa TestLatch

Exercício 1

- Implemente um programa concorrente com 5 threads. Todas as threads devem executar o mesmo trecho de código contendo um laço de repetição (controlado por uma constante)
- A cada iteração (passo), cada thread deve incrementar sua variável contadora, imprimir o seu ID, o valor da sua variável contadora e **o passo atual**, e só pode continuar sua execução DEPOIS que todas as threads completarem esse passo.

Exercício 1

- A saída do programa deverá ser como mostrado abaixo. Apenas a ordem das threads na coluna 1 pode variar de uma execução para outra.

```
Thread 0: cont=1, passo=0
Thread 1: cont=1, passo=0
Thread 2: cont=1, passo=0
Thread 3: cont=1, passo=0
Thread 4: cont=1, passo=0
```

Exercício 1

Thread 4: cont=2, passo=1
Thread 0: cont=2, passo=1
Thread 2: cont=2, passo=1
Thread 1: cont=2, passo=1
Thread 3: cont=2, passo=1

Exercício 1

Thread 3: cont=3, passo=2
Thread 4: cont=3, passo=2
Thread 0: cont=3, passo=2
Thread 2: cont=3, passo=2
Thread 1: cont=3, passo=2
Thread 1: cont=4, passo=3
Thread 3: cont=4, passo=3
Thread 4: cont=4, passo=3
Thread 0: cont=4, passo=3
Thread 2: cont=4, passo=3
Thread 2: cont=5, passo=4
Thread 1: cont=5, passo=4
Thread 4: cont=5, passo=4
Thread 0: cont=5, passo=4
Thread 3: cont=5, passo=4

Exercício 2

- Criar uma classe chamada `Caixas` com os seguintes atributos e métodos:
- - uma variável inteira chamada `itens`, inicializada com zero
- - uma constante chamada `capacidade` inicializada com 10
- - um método `retirar()` que decrementa o valor de itens em uma unidade (tem que fazer um teste aqui!)
- - um método `colocar()` que, caso o valor de itens esteja abaixo da capacidade, incrementa itens em uma unidade
- - um método `main` que cria uma instância da classe `Caixas` e duas threads, ambas recebendo o objeto `Caixas` criado. Uma das threads (a consumidora), ao executar, deve fazer chamadas ao método `retirar()` e outra (a thread produtora) ao método `colocar()` do objeto Caixas
  - Crie mensagens na tela para observamos o valor atual da variável `itens` depois de cada operação e ao final da execução do programa