

# Programação\_Concorrente-Studies/ Aula05: Revisando sobre Threads

- Quando usar e controlar pelo programador as ações de paralelismo e concorrência na Web:

## Quando usar

Processos batch (em lote)

Aplicações que executam no cliente

- Diferenças entre Run e Start:

```
// Nova thread
Thread t1 = new Thread(new MeuRunnable());
//t1.run(); // apenas executando na mesma thread
t1.start(); // executando em uma nova thread | 1
```

- Logo, caso eu use apenas o run na minha main, ele executa o método run da Thread criada por mim na Thread atual, logo na main, não sendo executado na Thread criada por nós, logo, quando executamos o start(), ele inicia a nova Thread e executa o método run dentro dela.
- **Utilizando o Sincronized:** O synchronized deve ser usado sempre que eu tenho uma variável ou um recurso que é compartilhado entre as minhas Thread, então caso eu tenho uma variável que é modificada por Thread, e essa variável é modificada por multiThreads, eu tenho que ter uma certa preocupação e utilizar o método synchronized, para que não ocorra “condições de corrida”:

```
1  import model.MyRunnable;
2
3  public class Main{
4
5      Run | Debug
6      public static void main(String[] args) {
7
8          MyRunnable myRunnable = new MyRunnable();
9
10         Thread t0 = new Thread(myRunnable);
11         Thread t1 = new Thread(myRunnable);
12         Thread t2 = new Thread(myRunnable);
13         Thread t3 = new Thread(myRunnable);
14         Thread t4 = new Thread(myRunnable);
15
16         t0.start();
17         t1.start();
18         t2.start();
19         t3.start();
20         t4.start();
21     }
22 }
```

```

package model;

public class MyRunnable implements Runnable{

    static int count = -1;

    @Override
    public void run() {
        count++;
        System.out.println("Thread " + Thread.
            currentThread().getName() + " is running.
            Count: " + count);
    }
}

```

- Aqui temos um exemplo de 5 Threads que estão compartilhando o mesmo recurso, sendo ele a variável count, logo quando executarmos teremos saídas distintas assim como listado abaixo:

```

n
Thread Thread-1 is running. Count: 2
Thread Thread-4 is running. Count: 4
Thread Thread-3 is running. Count: 3
Thread Thread-2 is running. Count: 2
Thread Thread-0 is running. Count: 2
marcus_cs_pereira@MacBook-Pro-de-Marcus Aula05 %

```

- Quando utilizamos o synchronized, apenas uma Thread vai poder executar esse método por vez, logo enquanto a Thread executa o método as outras não podem interferir na execução do método, controlando assim os erros causados pelas “condições de corrida”:

```

static int count = -1;

@Override
public synchronized void run() {
    count++;
    System.out.println("Thread " + Thread.
        currentThread().getName() + " is running.
        Count: " + count);
}

```

- Aqui temos o método da Runnable com o synchronized, e logo abaixo sua saída:

```

bin Main
Thread Thread-0 is running. Count: 0
Thread Thread-2 is running. Count: 1
Thread Thread-4 is running. Count: 2
Thread Thread-3 is running. Count: 3
Thread Thread-1 is running. Count: 4
marcus_cs_pereira@MacBook-Pro-de-Marcus Aula05 %

```

- Percebe-se que a Thread que executa a contagem não é necessariamente a ordem que damos start nelas, porém agora apenas uma Thread pode executar o método run() por vez, logo o método synchronized elimina a concorrência da variável count.
- Podemos também utilizar o synchronized em um bloco de código da seguinte maneira:

```

public class MyRunnable implements Runnable{

    static int count = -1;
    static Object Lock1 = new Object();
    static Object Lock2 = new Object();

    @Override
    public void run() {
        synchronized(Lock1){
            count++;
            System.out.println("Thread " + Thread.
                currentThread().getName() + " is
                running. Count: " + count);
        }
        synchronized(Lock2){
            count++;
            System.out.println("Thread " + Thread.
                currentThread().getName() + " is
                running. Count: " + count);
        }
    }
}

```

- O synchronized dessa forma necessita receber um objeto para servir como trava, logo eu posso passar qualquer objeto ali que será minha trava de execução daquela parte do código, podendo ser o this ou qualquer outro objeto, nesse caso criamos 2 objetos que servirão como travas dos métodos dentro do bloco synchronized.
- Caso utilizemos 2 Locks diferentes vamos obter:

```
Thread Thread-0 is running. Count: 0
Thread Thread-0 is running. Count: 2
Thread Thread-3 is running. Count: 2
Thread Thread-3 is running. Count: 3
Thread Thread-4 is running. Count: 4
Thread Thread-2 is running. Count: 6
Thread Thread-4 is running. Count: 6
Thread Thread-1 is running. Count: 7
Thread Thread-2 is running. Count: 8
Thread Thread-1 is running. Count: 9
marcus_cs_pereira@MacBook-Pro-de-Marcus
```

- Porém caso a gente utilize o mesmo lock para ambos os blocos de código synchronized:

```
er/workspaces/storage/297c48ed010c7d19ee
n
Thread Thread-0 is running. Count: 0
Thread Thread-3 is running. Count: 1
Thread Thread-3 is running. Count: 2
Thread Thread-0 is running. Count: 3
Thread Thread-2 is running. Count: 4
Thread Thread-2 is running. Count: 5
Thread Thread-1 is running. Count: 6
Thread Thread-1 is running. Count: 7
Thread Thread-4 is running. Count: 8
Thread Thread-4 is running. Count: 9
marcus_cs_pereira@MacBook-Pro-de-Marcus
```

- Porém é importante ressaltar o uso de ambas as formas possíveis de trava, caso utilizemos uma mesma trava, travamos a execução daqueles blocos de código, porém caso utilizemos travas diferentes conseguimos permitir que enquanto uma Thread está executando um bloco com a posse daquele objeto, outra Thread também está com a posse de outro objeto executando o outro bloco de código synchronized.
- Executando synchronized em um método static. Primeiramente devemos ressaltar que em métodos statics não temos como referenciar usando this, logo, temos 2 opções, criar Objetos Lock que não é a convencional, ou a opção abaixo:

```
public static void imprimi(){
    synchronized(MyRunnable.class){
        System.out.println("Thread " + Thread.
            currentThread().getName() + " is
            running. Count: " + count);
    }
}
```

- Aqui referenciamos a própria classe como objeto lock do synchronized, por conta que esse método é static.
- Quando utilizamos dessa forma, estamos sincronizando completamente a thread, pois estamos utilizando o synchronized na nossa classe, logo quando antes utilizávamos o this, estamos nós referindo a instância de uma classe, e não a toda a Classe em si, então se eu tivesse 2 Runnables declaradas para 2 Threads, elas não seriam sincronizadas, por conta que

meu método synchronized iria procurar o synchronized na sua instância, e dessa forma do static, temos o Synchronized na nossa classe.

- Desvantagens de utilizar o Synchronized:
- Acaba com o paralelismo caso eu uso o synchronized em todo bloco de código do meu runnable.
- Exemplo de uso racional do synchronized:

```
public class MyRunnable implements Runnable{

    static int count = 0;

    //static Object Lock1 = new Object();
    //static Object Lock2 = new Object();

    @Override
    public void run() {
        int i = 0;
        synchronized(this){
            count++;
            i = count*2;
        }

        double iElevadoA10 = Math.pow(i, 10);
        double sqrt = Math.sqrt(iElevadoA10);
        System.out.printf(format:"%.2f\n",sqrt);
    }
}
```

- Aqui foi usado o synchronized apenas onde existe a concorrência entre as Threads, nesse caso quando ambas estão necessitando do acesso da variável count, logo quando as Threads executam, elas vão parar no synchronized para realizar o count++ e a modificação da variável i, e após isso elas seguem executando sem depender uma das outras, um exemplo seria se caso uma Thread já tenha passado do bloco synchronized e outra tenha entrado, a Thread que saiu pode continuar sua execução enquanto a outra Thread está no bloco synchronized.
- Considerações finais:
- Use o synchronized com a palavra reservada this, em métodos, variáveis e recursos que não são static.
- Use o synchronized com a classe como Lock, em métodos, variáveis e recursos static.
- Logo a correção da runnable acima seria:

```

public class MyRunnable implements Runnable{

    static int count = 0;


    //static Object Lock1 = new Object();
    //static Object Lock2 = new Object();

    public void run() {
        int i;
        synchronized(MyRunnable.class){
            count++;
            i = count*2;
        }

        double iElevadoA10 = Math.pow(i, 10);
        double sqrt = Math.sqrt(iElevadoA10);
        System.out.printf(format:"%.2f%n", sqrt);
    }
}

```

Clique para recolher o intervalo.



- Sincronizando collections em várias Threads:

O problema que vamos enfrentar agora é quando usamos certas coleções e essas coleções vão ser o recurso compartilhado entre nossas Threads, pode ocorrer uma “condição de corrida”, sendo ela um problema de sobreposição de dados na lista, ou a falta de algum dado na lista, pois ambas as Threads tentam acessar uma coleção e acaba que as vezes uma Thread tenta usar a lista e ela já está sendo usada, dessa forma as vezes ocorre da lista ficar vazia, ou uma Thread sobrescrever o que a outra Thread colocou na lista.



```

public class SincronizarColecoes {

    private static List<String> lista = new
    ArrayList<>();
    Run | Debug
    public static void main(String[] args) throws
    InterruptedException{
        MeuRunnable meuRunnable = new MeuRunnable
        ();
        Thread t0 = new Thread(meuRunnable);
        Thread t1 = new Thread(meuRunnable);
        Thread t2 = new Thread(meuRunnable);

        t0.start();
        t1.start();
        t2.start();
        System.out.println(lista);
        Thread.sleep(millis:1000);
    }

    public static class MeuRunnable implements
    Runnable {

        @Override
        public void run() {
            lista.add(e:"teste");
        }
    }
}

```

- Aqui temos um exemplo, onde a execução pode ocorrer perfeitamente e sem erros, assim como a execução pode não ser perfeita como esperamos e acabar sobrescrevendo itens já

adicionados ou a lista pode até ficar com espaços vazios, isso ocorre pois objetos List não são adequados para uso como recurso compartilhado entre Thread.

- Agora vamos as formas de sincronizar o nosso objeto List, mesmo ele não sendo o mais recomendado para uso de MultiThreading:

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3  import java.util.List;
4
5  public class SincronizarColecoes {
6
7      private static List<String> lista = new
      ArrayList<>();
      Run | Debug
8      public static void main(String[] args) throws
      InterruptedException{
9
10         lista = Collections.synchronizedList
            (lista);
11
12         MeuRunnable meuRunnable = new MeuRunnable
            ();
13         Thread t0 = new Thread(meuRunnable);
14         Thread t1 = new Thread(meuRunnable);
15         Thread t2 = new Thread(meuRunnable);
16
17         t0.start();
18         t1.start();
19         t2.start();
20         System.out.println(lista);
21         Thread.sleep(millis:1000);
22     }
```

- Para isso tivemos que adicionar um método que sincroniza nossa lista e permite que apenas uma Thread faça a operação dela por vez, evitando as “condições de corrida”.
- Além do método SynchronizedList(), nós também temos os métodos:

```
Collections.synchronizedCollection  
(c:null);  
Collections.synchronizedList(list:null);  
Collections.synchronizedMap(m:null);  
Collections.synchronizedSet(s:null);  
Collections.synchronizedSortedMap(m:null);  
Collections.synchronizedSortedSet(s:null);
```

- Logo, utilize a versão do synchronized de acordo a sua coleção.
- Formas de evitar o uso do synchronized, afinal ele quebra o paralelismo das Threads:
- Utilizaremos classes Thread-safe: são classes que são seguras para utilizar em cenários de multiThreading
- A primeira que podemos utilizar é substituir o ArrayList por CopyOnWriteArrayList<>();

```

2  import java.util.Collections;
3  import java.util.List;
4
5  public class SincronizarColecoes {
6
7      private static List<String> lista =
        Collections.synchronizedList(new
        ArrayList<String>());
        Run | Debug
8      public static void main(String[] args) throws
        InterruptedException{
9
10         MeuRunnable meuRunnable = new MeuRunnable
            ();
11         Thread t0 = new Thread(meuRunnable);
12         Thread t1 = new Thread(meuRunnable);
13         Thread t2 = new Thread(meuRunnable);
14
15         t0.start();
16         t1.start();
17         t2.start();
18         System.out.println(lista);
19         Thread.sleep(millis:1000);
20     }
21

```

- Porém o uso dessa classe gera um problema, ela é muito pesada, pois sempre que você modifica ela, ele copia o Array inteiro na memória, logo não é tão útil em termo de performance.
- Logo não usaremos essa classe quando tivermos muitas operações de escrita como add e remove.
- Usaremos essa classe quando tivermos operações de leitura e etc.
- Quando formos trabalhar com HashMap devemos substituir o HashMap por ConcurrentHashMap:

```

ArrayList<String>());
private static ConcurrentHashMap<Integer,
String> lista = new ConcurrentHashMap<>();
Run | Debug
public static void main(String[] args) throws
InterruptedException{

    MeuRunnable meuRunnable = new MeuRunnable
    ();
    Thread t0 = new Thread(meuRunnable);
    Thread t1 = new Thread(meuRunnable);
    Thread t2 = new Thread(meuRunnable);

    t0.start();
    t1.start();
    t2.start();
    System.out.println(lista);
    Thread.sleep(millis:1000);
}

```

- A desvantagem de usar esse concurrent é o fator que ela é mais lenta que o próprio HashMap.
- E por último temos uma classe pouco utilizada mais muito útil que é a LinkedBlockingQueue, trabalhando com filas:
-

```
private static BlockingQueue<String> fila =  
new LinkedBlockingQueue<>();
```

Run | Debug

```
public static void main(String[] args) throws  
InterruptedException{
```

```
    MeuRunnable meuRunnable = new MeuRunnable  
    ();
```

```
    Thread t0 = new Thread(meuRunnable);
```

```
    Thread t1 = new Thread(meuRunnable);
```

```
    Thread t2 = new Thread(meuRunnable);
```

```
    t0.start();
```

```
    t1.start();
```

```
    t2.start();
```

```
    System.out.println(fila);
```

```
    //System.out.println(lista);
```

```
    Thread.sleep(millis:1000);
```

```
}
```

```
public static class MeuRunnable implements
```

- Temos várias formas de se manipular os dados nessa fila:

```
fila.add(e:"teste");
fila.poll();
fila.offer(e:"teste");
try {
    fila.put(e:"teste");
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- O add padrão que adiciona elementos na fila
  - O poll retorna e remove o elemento da head da fila.
  - O offer tenta adicionar o elemento na fila caso haja espaço ainda nessa fila, pois podemos determinar o tamanho dessa fila quando instenciamos ela.
  - O put coloca o elemento na fila quando possível, logo ele aguarda até o momento que seja possível ele adicionar o elemento caso haja espaço na fila.
  - Todas essas operações só são possíveis pois essa fila é Thread-safe, ela foi criada pensando no uso do multithreading, por isso temos essas características nesses métodos.
- Formas de evitar o uso do Synchronized utilizando operações atômicas:
- Utilizaremos classes atômicas para evitar o uso do synchronized, dessa forma manteremos o paralelismo sem a ocorrência de “condições e corrida”.
- A primeira que iremos usar é o AtomicInteger:

```
static AtomicInteger i = new AtomicInteger  
(-1);
```

Run | Debug

```
public static void main(String[] args) {  
    MeuRunnable meuRunnable = new MeuRunnable  
    ();  
  
    Thread t0 = new Thread(meuRunnable);  
    Thread t1 = new Thread(meuRunnable);  
    Thread t2 = new Thread(meuRunnable);  
  
    t0.start();  
    t1.start();  
    t2.start();  
}
```

```
public static class MeuRunnable implements  
Runnable {  
    public void run() {  
        String name = Thread.currentThread().  
        getName();  
        System.out.println(name + " " + i.  
        incrementAndGet());  
    }  
}
```

- Declaramos ele com o nosso valor inicial, e dentro do sout temos a forma de fazer incremento na nossa variável.



- Dessa forma garantimos que não ocorra “condições de corrida”, pois essa classe garante que nossa operação seja realizada atomicamente, sendo assim ela garante que o método `incrementAndGet` só pode ser executado um por vez.
  - Logo operações atômicas são operações que só podem ser executadas por uma Thread, sem interrupções causadas por outros meios.
  - Temos também o `AtomicLong`.
- Agora veremos o `AtomicBoolean`:

```
static AtomicBoolean b = new AtomicBoolean  
(initialValue:false);
```

Run | Debug

```
public static void main(String[] args) {  
    MeuRunnable meuRunnable = new MeuRunnable  
    ();  
  
    Thread t0 = new Thread(meuRunnable);  
    Thread t1 = new Thread(meuRunnable);  
    Thread t2 = new Thread(meuRunnable);  
  
    t0.start();  
    t1.start();  
    t2.start();  
}
```

```
public static class MeuRunnable implements  
Runnable {  
    public void run() {  
        String name = Thread.currentThread().  
        getName();  
        //System.out.println(name + " " + i.  
        incrementAndGet());  
        System.out.println(name + " " + b.  
        compareAndSet(expect:false,  
        update:true));  
    }  
}
```

- Dessa forma podemos controlar os booleanos, com métodos atômicos e suas propriedades específicas.

- Agora iremos ver o AtomicReference:

```
(false);  
  
static AtomicReference<Object> s = new  
AtomicReference<>(new Object());
```

Run | Debug

```
public static void main(String[] args) {  
    MeuRunnable meuRunnable = new MeuRunnable  
    ();
```

```
    Thread t0 = new Thread(meuRunnable);
```

```
    Thread t1 = new Thread(meuRunnable);
```

```
    Thread t2 = new Thread(meuRunnable);
```

```
    t0.start();
```

```
    t1.start();
```

```
    t2.start();
```

```
}  
  
public static class MeuRunnable implements  
Runnable {
```

```
    public void run() {
```

```
        String name = Thread.currentThread().  
        getName();
```

```
        //System.out.println(name + " " + i.  
        incrementAndGet());
```

```
        //System.out.println(name + " " + b.  
        compareAndSet(false, true));
```

```
        System.out.println(name + " " + s.  
        getAndSet(new Object()));
```

```
    }
```

```
}
```

- Essa classe recebe como parâmetro um objeto e cria uma referência atômica para ele, logo, ela transforma um objeto passado em atômico, assim podendo usar os métodos `getAndSet()`, etc..., dessa forma podendo transformar uma classe em Thread-safe.
- Uso do `Volatile`:
  - O `volatile` é uma palavra reservada que resolve um problema grave no multithreading, pois a execução de um código na prática não é como ocorre em nível de máquina, e quando tratamos de Multithreading a ordem do nosso programa não é importante tanto, e sim devemos dar prioridade ao paralelismo.
  - Para isso usamos o `volatile` em variáveis e abrimos mão do cache local do processador, dessa forma permitindo a prevenção de erros ocasionados por conta do paralelismo entre Threads.

```
public class Volatile {  
    private static volatile int number = 0;  
    private static volatile boolean ready = false;  
  
    private static class MeuRunnable implements  
    Runnable {  
        public void run() {  
            while (!ready) {  
                Thread.yield();  
            }  
  
            if (number != 42) {  
                System.out.println("number = " +  
                number);  
            }  
        }  
    }  
}
```

#Concurrent