

# Abstrações de Alto Nível para Programação Concorrente (em Java)

Fernando Castor

Professor Adjunto

Centro de Informática

Universidade Federal de Pernambuco

Elaborado conjuntamente por:

Benito Fernandes, João Paulo Oliveira e Wesley Torres

# Locks, Monitores, Blocos Guardados, ....

## ■ Abstrações de **baixo nível**

- Resolvem os problemas, mas é fácil errar
- Nem sempre apresentam um **bom desempenho**

## ■ Suposição subjacente:

- Estado compartilhado
- *Threads* precisam **controlar o acesso** a esse estado

## ■ **Alternativa:** estruturas de dados que controlam esse acesso

```
public class Drop {
    private String message;
    private boolean empty = true;
    public synchronized String take() {
        while (empty) {
            try { wait();
            } catch (InterruptedException e) {} }
        empty = true;
        notifyAll();
        return message;
    }
    public synchronized void put(String message) {
        while (!empty) {
            try { wait();
            } catch (InterruptedException e) {} }
        empty = false;
        this.message = message;
        notifyAll();
    }
}
```

Relembrando...

# Algumas estruturas/padrões são muito comuns

## ■ Exemplos:

- Semáforos -> Uma tentativa de decrementar um que já é igual a zero bloqueia a *thread*
- Tabelas *hash* usadas por várias *threads*
- *Threads* que se comunicam usando uma fila de mensagens

# Outro problema: criação de *threads*



- Criar *threads* é caro

- Mas menos que criar processos

- Para aplicações simples, o que já foi visto basta

- Aplicações mais complexas reciclam *threads*
  - Economiza-se memória
  - Economiza-se CPU
- Ou exigem políticas mais flexíveis para a criação de *threads*

Nesta aula, estudaremos algumas abstrações de alto nível

Implementadas pela biblioteca  
**`java.util.concurrent`**

Várias dessas aplicações podem ser encontradas em outras linguagens e em sistemas reais


# Tipos Primitivos Atômicos

# Ações atômicas

- Ações que acontecem como se fossem apenas uma
  - Indivisíveis
- Muitas vezes, **exclusão mútua == ação atômica**
- Nenhum efeito da ação é visível até ela ser completada



# Quais ações são atômicas?

- Ações de incremento como `c++` ? 
  - Expressões simples podem definir operações complexas
- Ações atômicas
  - Ler e escrever variáveis de referência
  - Ler e escrever a maioria das variáveis primitivas (exceto `long` e `double`)
  - Ler e escrever todas as variáveis declaradas como `volatile`

# Por que usar tipos atômicos?

- Mais eficiente que acessar variáveis de tipos comuns através de código sincronizado
- Programador não precisa controlar ao acesso aos dados compartilhados
- Abstração
  - Melhor representação do intento do programador
- Modularidade
  - Concorrência e lógica da aplicação melhor separados

# Tipos atômicos em Java

- `java.util.concurrent.atomic`
- Classes para a criação de valores/variáveis
  - Seguros para *threads* sem usar *locks* ou monitores
  - `AtomicBoolean`, `AtomicInteger`, `AtomicLong`, `AtomicReference`
- Métodos básicos para alteração e obtenção de variáveis
- Possuem também o método **`compareAndSet`**
- Também há ***arrays*** atômicos

# Métodos específicos

- As classes atômicas também fornecem métodos específicos para o tipo
- **AtomicLong** e **AtomicInteger** possuem métodos para incrementar seus valores
  - `getAndIncrement`
  - `getAndDecrement`
  - `getAndAdd`

```
class Sequencer {  
    private final AtomicLong sequenceNumber = new AtomicLong(0);  
  
    public long next() {  
        return sequenceNumber.getAndIncrement();  
    }  
}
```

---

# O bom e velho contador sequencial

```
class Counter {  
    private double c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public double value() {  
        return c;  
    }  
}
```

# Utilizando métodos sincronizados

```
class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {  
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

# Com variáveis atômicas

```
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet();  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

# Coleções Seguras para Threads



# Coleções concorrentes

- Interface **BlockingQueue<E>** - estrutura FIFO
  - bloqueia *threads* que tentem adicionar a uma fila cheia ou retirar de uma fila vazia.
- Interface **ConcurrentMap<K, V>** - operações atômicas para adição, remoção e atualização
  - **ConcurrentHashMap** – implementação padrão de **ConcurrentMap<K, V>**

# Produtor-Consumidor revisitado

```
import java.util.concurrent.*;

public class Produtor extends Thread {
    private BlockingQueue<Integer> cubiculo;
    private int numero;

    public Produtor(BlockingQueue<Integer> c, int num) {
        cubiculo = c;
        numero = num;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                cubiculo.put(i);
                System.out.println("Produtor " + numero + " coloca: " + i);
                sleep((int) (Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

# Produtor-Consumidor revisitado

```
import java.util.concurrent.*;

public class Consumidor extends Thread {
    private BlockingQueue<Integer> cubiculo;
    private int numero;

    public Consumidor(BlockingQueue<Integer> c, int num) {
        cubiculo = c;
        numero = num;
    }

    public void run() {
        int valor = 0;
        for (int i = 0; i < 10; i++) {
            try {
                valor = cubiculo.take();
                System.out.println("Consumidor " + numero + " pega: " + valor);
            } catch (InterruptedException e) { }
        }
    }
}
```

# Produtor-Consumidor revisitado

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class TestaProdutorConsumidor {
    public static void main(String[] args) {

        BlockingQueue<Integer> c = new ArrayBlockingQueue<Integer>(1);
        Produtor produtor = new Produtor(c, 1);
        Consumidor consumidor = new Consumidor(c, 1);

        produtor.start();
        consumidor.start();

    }
}
```

# ConcurrentHashMap

- **Otimiza** operações de **recuperação**.
- A operação de recuperação devolve o valor inserido mais recentemente
  - Para uma dada chave
- Pode retornar um valor acrescentado por uma operação de inserção ainda em andamento.
- **Nunca** retorna um resultado sem sentido.

# Escalabilidade

## Hashtable vs ConcurrentHashMap

Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

Tempo em milissegundos para 10.000.000 de iterações

# Escalabilidade

## Hashtable vs ConcurrentHashMap

Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

Por que essa diferença de **desempenho** tão grande?

# Sincronizando coleções

- As coleções **padrão** de Java (`ArrayList`, ...) não são sincronizadas.
  - **`Vector`** e **`Hashtable`** são exceções
- Para criar coleções sincronizadas, pode-se criar um “decorador” da coleção
  - sincroniza os métodos.
  - **Não tão eficiente** quanto `java.util.concurrent`
  - Java já fornece tais decoradores na classe `Collections`
    - `Collection`, `List`, `Map`, `Set`



# Iterators podem ser um problema neste caso...

```
synchronized(list) {  
    Iterator i = list.iterator(); // Precisa estar em um bloco sincronizado  
    while (i.hasNext())  
        foo(i.next());  
}
```

# Executores e *Thread Pools*

# Executors

- Separam gerenciamento e criação de *threads* do resto da aplicação
- Executors encapsulam essas funções
  - Interfaces – Definem os tipos dos objetos
  - Thread Pools – o tipo mais comum de implementação de executor

# Interfaces

## ■ `java.util.concurrent`

### ■ **Executor**

- Criação e gerenciamento de *threads*

### ■ **ExecutorService**

- Subinterface de **Executor**
- Gerenciamento de ciclo de vida, para tarefas individuais ou para o executor em si

### ■ **ScheduledExecutorService**

- Subinterface de **ExecutorService**
- Execução de tarefas futuras e/ou programadas

# Interface **Executor**

- Provê um só método

- **execute** ( )

- Substitui a abordagem padrão para criação de threads

The diagram illustrates the transition from a traditional thread creation approach to an Executor-based approach. On the left, the code `(new Thread(r)).start();` is shown, with a green box around `(r)` and a green arrow pointing to the text "Runnable object". A large green arrow points to the right, where the code `e.execute(r);` is shown. A green box around `e` has a green arrow pointing to the text "Executor object".

```
(new Thread(r)).start(); → e.execute(r);
```

Runnable object

Executor object

- Pode executar na mesma *thread*, numa nova *thread* ou num *pool*

```

public class SerialExecutor implements Executor {
    final Queue<Runnable> tarefas = new ArrayDeque<Runnable>();
    final Executor executor;
    Runnable active;

    public SerialExecutor(Executor executor) {
        this.executor = executor;
    }

    public synchronized void execute(final Runnable r) {
        tarefas.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();
                }
            }
        });
        if (active == null) {
            scheduleNext();
        }
    }

    protected synchronized void scheduleNext() {
        if ((active = tarefas.poll()) != null) {
            executor.execute(active);
        }
    }
}

```

# Thread Pools

- **Coleção de threads** disponíveis para realizar tarefas
  - Criar muitas threads é **caro**!
- A maioria das implementações de executores em Java
  - *Worker threads*
  - As threads existem separadamente dos objetos **Runnable** que executam
  - Responsáveis por **múltiplas tarefas**

# Thread Pools, porque usar?

- *Worker threads* minimizam o **overhead da criação de threads**
- Threads usam uma quantidade significativa de memória
- Melhor desempenho quando se executa um grande número de tarefas
  - overhead reduzido de chamadas por tarefa
- Uma forma de limitar recursos consumidos



# Tamanho do pool

- É comum utilizar uma quantidade **fixa** de *threads* no pool
- Tarefas submetidas ao pool são enviadas para uma fila interna
- Armazena as tarefas quando existem mais tarefas do que threads

# Implementações de Pool

- **ThreadPoolExecutor**
- **ScheduledThreadPoolExecutor**
- Implementação permite que se estabeleça:
  - O numero de *threads*
  - O tipo de estrutura que será armazenada nas tarefas
  - Como criar e finalizar as *threads*
- Abordagem mais comum: fábrica **Executors**

# ExecutorService

- Estende a interface **Executor**
- Mais versátil
- Inclui métodos para gerenciamento de ciclo de vida
- Permite que a tarefa retorne um valor

# Finalizando um **ExecutorService**

- Rejeita novas tarefas
- **shutdown ()**
  - Permite executar as tarefas anteriormente submetidas antes de ser finalizado
- **shutdownNow ()**
  - Não permite que tarefas em espera inicializem
  - Finaliza as tarefas que estão sendo executadas

Mostrar código das classes  
**MuitasThreads,**  
**PoucasThreads** e  
**UmaThread**