

# Programação Concorrente em Java

Nelson Rosa – nsr@cin.ufpe.br



UNIVERSIDADE  
FEDERAL  
DE PERNAMBUCO

---

CIn.ufpe.br

# Objetivos

---

- **Estudar mecanismos de concorrência em Java de mais alto nível**
  - `java.util.concurrent`
- **Apresentar exemplos de uso desta API**
- **Não pretende ser exaustivo**

# **Java:: java.util.concurrent**

---

- **Disponível a partir da versão Java 1.5**
- **Classes utilitárias para programação concorrente**
- **Conceitos básicos**
  - Lock
  - Executors
  - Concurrent collections
  - Variáveis atômicas

# Java:: Locks

## ■ Lock

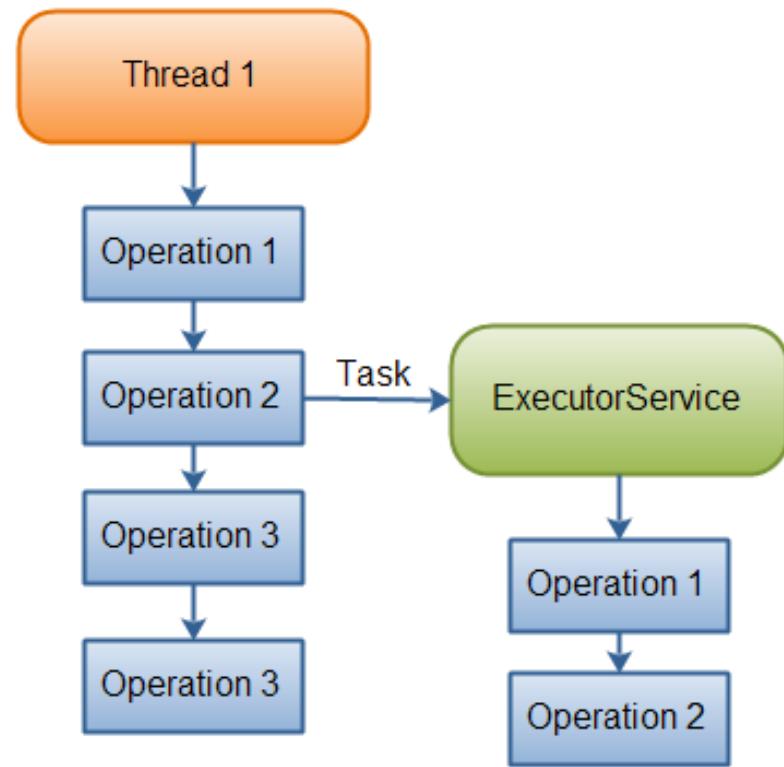
- Similar ao synchronized, mas com a possibilidade de se adicionar uma condição, e.g., timeout
- lock(), unlock(), tryLock()

■ **tryLock retorna imediatamente (ou depois de um tempo) se o lock não estiver disponível**

```
public void run() {  
    try {  
        if(lock.tryLock(10, TimeUnit.SECONDS)){  
            resource.doSomething();  
        }  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }finally{  
        //release lock  
        lock.unlock();  
    }  
    resource.doLogging();  
}
```

# Java:: Executors

- Princípio: gerenciamento do thread **separado** da aplicação
- Objetos Executors encapsulam o gerenciamento do thread
- Threads são executados de diversas formas
  - Criando um novo thread
  - Usando um thread já existente
  - Sequencialmente ou concorrentemente
- Interfaces
  - Executor
  - ExecutorService
  - ScheduledExecutorService

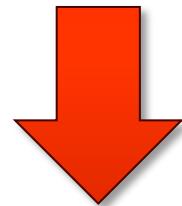


✓ **Thread 1** delega a **Task** para o **ExecutorService** e continua sua execução independente da execução de **Task**.

# Java:: Executors

Gerenciamento  
realizado  
pela aplicação

```
public static void main(String args[]) {  
    Thread thread = new Thread(new ThreadDemo());  
  
    thread.start();  
}
```



Gerenciamento  
realizado  
pelo Executor

```
public static void main(String[] args) {  
    Executor executor = Executors.newSingleThreadExecutor();  
    ThreadDemo thread = new ThreadDemo();  
  
    executor.execute(thread);  
}
```

# Java:: Executors:: Interface Executor

---

- Usada para execução de tarefas
- Fornece um único método
  - execute: cria e executa um novo thread

```
public class ThreadExecutor {  
  
    private static class ThreadDemo01 implements Runnable {  
        public void run() {  
  
            System.out.println("Tarefa a ser executada");  
        }  
    }  
  
    public static void main(String[] args) {  
        Executor executor = Executors.newSingleThreadExecutor();  
  
        executor.execute(new ThreadDemo01());  
    }  
}
```

# Java:: Executors:: Interface Executor

---

```
public class ThreadServiceExecutorPool {  
  
    private static class ThreadDemo implements Runnable {  
        public void run() {  
  
            System.out.println("Tarefa executada por um pool de threads");  
        }  
    }  
  
    public static void main(String[] args) {  
  
        Executor executor = Executors.newFixedThreadPool(10);  
  
        executor.execute(new ThreadDemo());  
    }  
}
```

# Java:: Executors:: Interface ExecutorService

## ■ Estende Executor

## ■ Adiciona o método submit

- Aceita objetos Runnable, Callable
  - Callable permite a tarefa retornar um valor
- Retorna um objeto Future
  - Future é usado para recuperar o valor do objeto Callable
- Future gerencia o status das tarefas Callable e Runnable

```
public class ThreadExecutor {  
  
    private static class ThreadDemo01 implements Runnable {  
        public void run() {  
            System.out.println("Tarefa assincrona");  
        }  
    }  
  
    public static void main(String[] args) {  
        ExecutorService executor0 =  
            Executors.newSingleThreadExecutor();  
  
        executor.submit(new ThreadDemo01());  
  
        executor.shutdown();  
    }  
}
```

- ✓ Não é possível obter o resultado da tarefa executada se objetos Runnable são usados.
- ✓ Nestes casos, usam-se objetos Callable.

# Java:: Executors:: Interface ExecutorService

```
public class ThreadRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from thread \'"+Thread.currentThread().getName()+"\'");
    }
    public static void main(String args[]) {
        Thread thread = new Thread(new ThreadRunnable());
        thread.start();
    }
}
```

```
private static class CallableDemo implements Callable{
    @Override
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Retorno do Callable";
    }
}
public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(10);
    Future future = executor.submit(new CallableDemo());
    System.out.println("future.get() = " + future.get());
}
```

# Java:: Executors:: Interface ExecutorService:: invokeAny & invokeAll

invokeAny: quando uma das tarefas termina (ou gera exceção), as outras são canceladas.

```
public static void main(String args[]) {  
    ExecutorService executor = Executors.newFixedThreadPool(10);  
    Set<Callable<String>> callables = new HashSet<Callable<String>>();  
  
    callables.add(new CallableDemo01());  
    callables.add(new CallableDemo02());  
    callables.add(new CallableDemo03());  
  
    System.out.println(executor.invokeAny(callables));  
  
    executor.shutdown();  
}
```

invokeAll: invoca todas as tarefas (em uma ordem qualquer)

```
public static void main(String args[]) {  
    ExecutorService executor = Executors.newFixedThreadPool(10);  
    Set<Callable<String>> callables = new HashSet<Callable<String>>();  
    List<Future<String>> futures = null;  
  
    callables.add(new CallableDemo01());  
    callables.add(new CallableDemo02());  
    callables.add(new CallableDemo03());  
  
    futures = executor.invokeAll(callables);  
    executor.shutdown();  
}
```

# Java:: Executors:: Interface

## ScheduledExecutorService

- Escalona tarefas para executar depois de um certo tempo ou repetidamente com um intervalo fixo de tempo
- Adiciona o método **schedule**
  - Executa objetos `Runnable` e `Callable` depois de um tempo determinado

```
private static class CallableDemo implements Callable {  
  
    @Override  
    public String call() throws Exception {  
        System.out.println("Tarefa Executada!!");  
        return "Task 1";  
    }  
}  
public static void main(String args[]) {  
    ScheduledExecutorService scheduledExecutor =  
        Executors.newScheduledThreadPool(5);  
  
    ScheduledFuture scheduledFuture =  
        scheduledExecutor.schedule(new CallableDemo(), 5, TimeUnit.MILLISECONDS);  
  
    scheduledExecutor.shutdown();  
}
```

# Java:: Executors:: Pool de Threads

---

- Usado para executar múltiplos threads (worker threads)
- Minimiza o overhead da criação de threads
- Tarefas são submetidas ao pool através de uma fila interna

```
public class ThreadServiceExecutorPool {  
  
    private static class ThreadDemo implements Runnable {  
        public void run() {  
  
            System.out.println("Tarefa executada por um thread do pool");  
        }  
    }  
  
    public static void main(String[] args) {  
  
        Executor executor = Executors.newFixedThreadPool(10);  
  
        executor.execute(new ThreadDemo());  
    }  
}
```

# Java:: Executors:: Fork/Join

## ■ Princípio

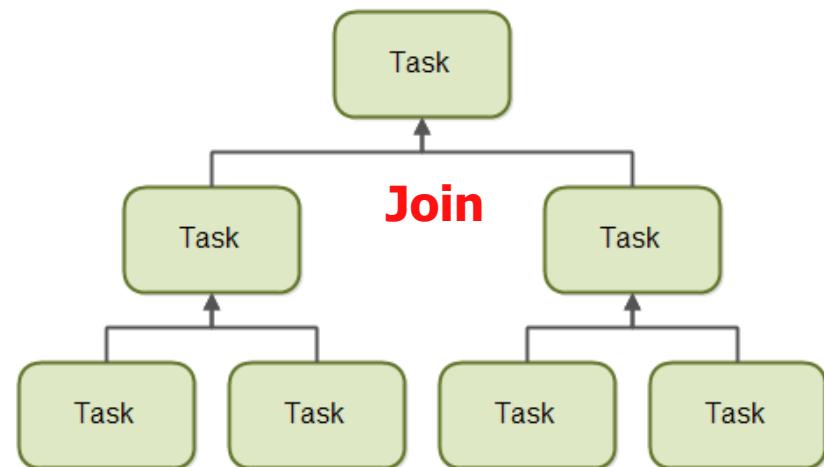
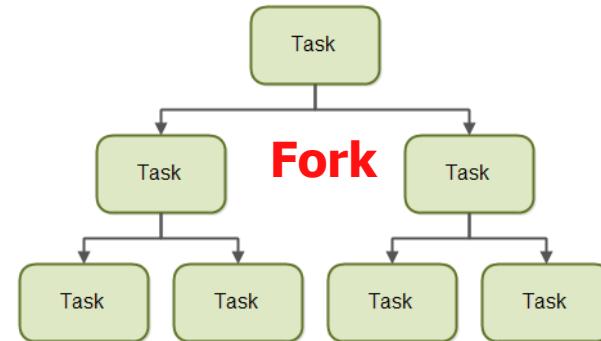
- “primeiro fork e depois join”

## ■ Fork

- Uma tarefa pode usar o fork para “quebrá-la” em pequenas sub-tarefas que podem ser executadas concurrentemente
- **Importante:** a tarefa tem que ser grande suficiente para justificar a “quebra”

## ■ Join

- Quando a tarefa foi “quebrada” em sub-tarefas, o join faz com que a tarefa espere até que todas as suas sub-tarefas tenham terminado as suas execuções

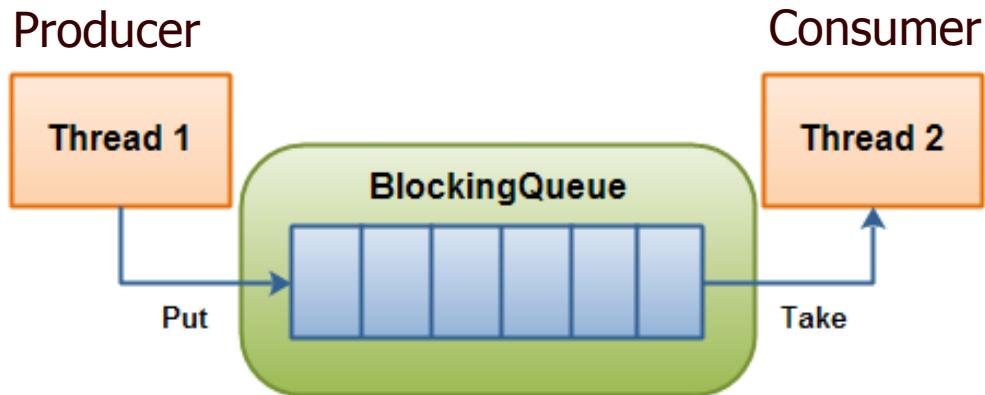


# Java:: Executors:: Concurrent Collections:: BlockingQueue

## ■ BlockingQueue

- Define uma estrutura de FIFO que bloqueia ou timeout quando tenta-se adicionar um novo elemento à uma fila cheia ou remover um elemento de uma fila vazia, i.e., queue thread safe

## ■ Tipicamente usada quando precisamos de uma fila onde um thread produz objetos enquanto outro thread consome



- ✓ **Produtor** produz objetos e coloca-os na fila até a fila ficar cheia
- ✓ Quando a fila enche, o **produtor** é bloqueado quando tenta inserir um novo objeto, ele fica bloqueado até que o **consumidor** remova um objeto da fila
- ✓ **Consumidor** remove elementos da fila até que ela esvazie
- ✓ Quando o **consumidor** tenta remover um objeto de uma fila vazia, ele é bloqueado até o **produtor** coloque algum objeto na fila

# Java:: Executors:: Concurrent Collections:: BlockingQueue

```
class Producer implements Runnable {  
    protected BlockingQueue queue = null;  
  
    public Producer(BlockingQueue queue) {  
        this.queue = queue;  
    }  
  
    public void run() {  
        queue.put("1"); Thread.sleep(1000);  
        queue.put("2"); Thread.sleep(1000);  
        queue.put("3");  
    }  
}
```

```
class Consumer implements Runnable {  
    protected BlockingQueue queue = null;  
  
    public Consumer(BlockingQueue queue) {  
        this.queue = queue;  
    }  
  
    public void run() {  
        System.out.println(queue.take());  
        System.out.println(queue.take());  
        System.out.println(queue.take());  
    }  
}
```

```
public class ThreadBlockingQueue {  
  
    public static void main(String[] args) throws Exception {  
  
        BlockingQueue queue = new ArrayBlockingQueue(1024);  
  
        Producer producer = new Producer(queue);  
        Consumer consumer = new Consumer(queue);  
  
        new Thread(producer).start();  
        new Thread(consumer).start();  
  
        Thread.sleep(4000);  
    }  
}
```

# Java:: Executors:: Concurrent Collections

---

## ■ **ConcurrentMap**

- Define operações atômicas para `java.util.Map`
- Operações que removem/trocam um chave-valor se a chave já existe ou adiciona um par se a chave não existe
- Bloqueia apenas a parte do Map que está sendo atualizada

## ■ **ConcurrentNavigableMap**

- Sub interface de **ConcurrentMap** com suporte a match parcial
- Acesso concorrente a submaps

# Java:: Executors:: Variáveis atômicas

---

- Permite operações atômicas sobre variáveis
- Possui métodos get/set

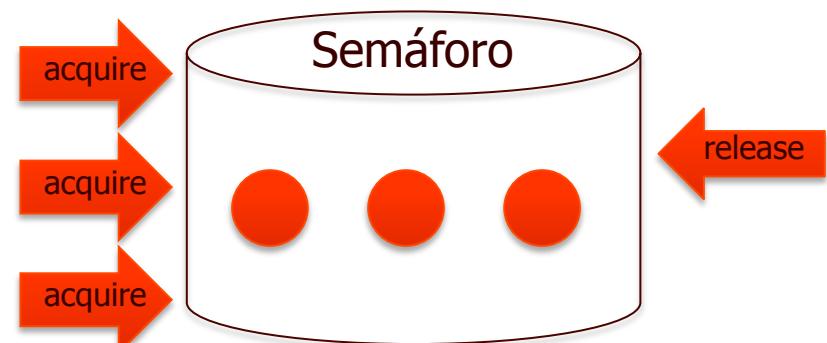
```
class AtomicCounter {  
    private AtomicInteger c = new AtomicInteger(0);  
  
    public void increment() {  
        c.incrementAndGet();  
    }  
  
    public void decrement() {  
        c.decrementAndGet();  
    }  
  
    public int value() {  
        return c.get();  
    }  
}
```

# Java:: Semáforo

---

- Restringe o número de acessos simultâneos a um recurso compartilhado a um certo limite

- e.g., threads solicitam o acesso a um determinado recurso compartilhado (decrementa o semáforo) e sinalizam o fim do uso do recurso (decrementa o semáforo)



- Variável inteira com operações incrementa/decrementa atômicas

# Java:: Semáforo

---

## ■ Usos de semáforo

- Guardar um região crítica permitindo que N threads acessem a região
- Enviar sinalização entre dois threads

## ■ O semáforo é inicializado com um certo número de permissões (N)

- `acquire`: thread obtém uma permissão
- `release`: uma permissão é retornada ao semáforo

## ■ Sinalização entre threads

- Um thread “`acquire`” e o outro “`release`”
- Se as permissões “acabaram”, `acquire` bloqueia até a liberação do outro thread
- Se todas as permissões foram liberadas, o `release` bloqueia o thread

```
public class ThreadSemaphoreSimple {  
  
    public static void main(String[] args) {  
        Semaphore semaphore = new Semaphore(1);  
  
        // região crítica  
        semaphore.acquire();  
  
        semaphore.release();  
    }  
}
```

## Java:: Semáforo

---

- Não há garantias de que o primeiro thread bloqueado pelo `acquire` será o primeiro a obter a permissão (fairness)
- Fairness pode ser obtida informando isto na criação do semáforo
  - FIFO



```
Semaphore semaphore = new Semaphore(1,true);
```

# Java:: Semáforo

```
class SendingThread extends Thread {  
    Semaphore semaphore = null;  
  
    public SendingThread(Semaphore semaphore) {  
        this.semaphore = semaphore;  
    }  
  
    public void run() {  
        while (true) {  
            System.out.println("Hello from Sending");  
            try {  
                this.semaphore.acquire();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
class ReceivingThread extends Thread {  
    Semaphore semaphore = null;  
  
    public ReceivingThread(Semaphore semaphore) {  
        this.semaphore = semaphore;  
    }  
  
    public void run() {  
        while (true) {  
            this.semaphore.release();  
            System.out.println("Hello from Receiving");  
        }  
    }  
}
```

```
public class ThreadSemaphoreSync {  
  
    public static void main(String args[]) {  
        Semaphore semaphore = new Semaphore(1);  
        SendingThread sender = new SendingThread(semaphore);  
        ReceivingThread receiver = new ReceivingThread(semaphore);  
  
        receiver.start();  
        sender.start();  
    }  
}
```

---

# **Fim dos Slides**