Programação em Java com J2SE 5.0

Utilitários para Programação Concorrente

Helder da Rocha

Agosto 2005

Pacote java.util.concurrent

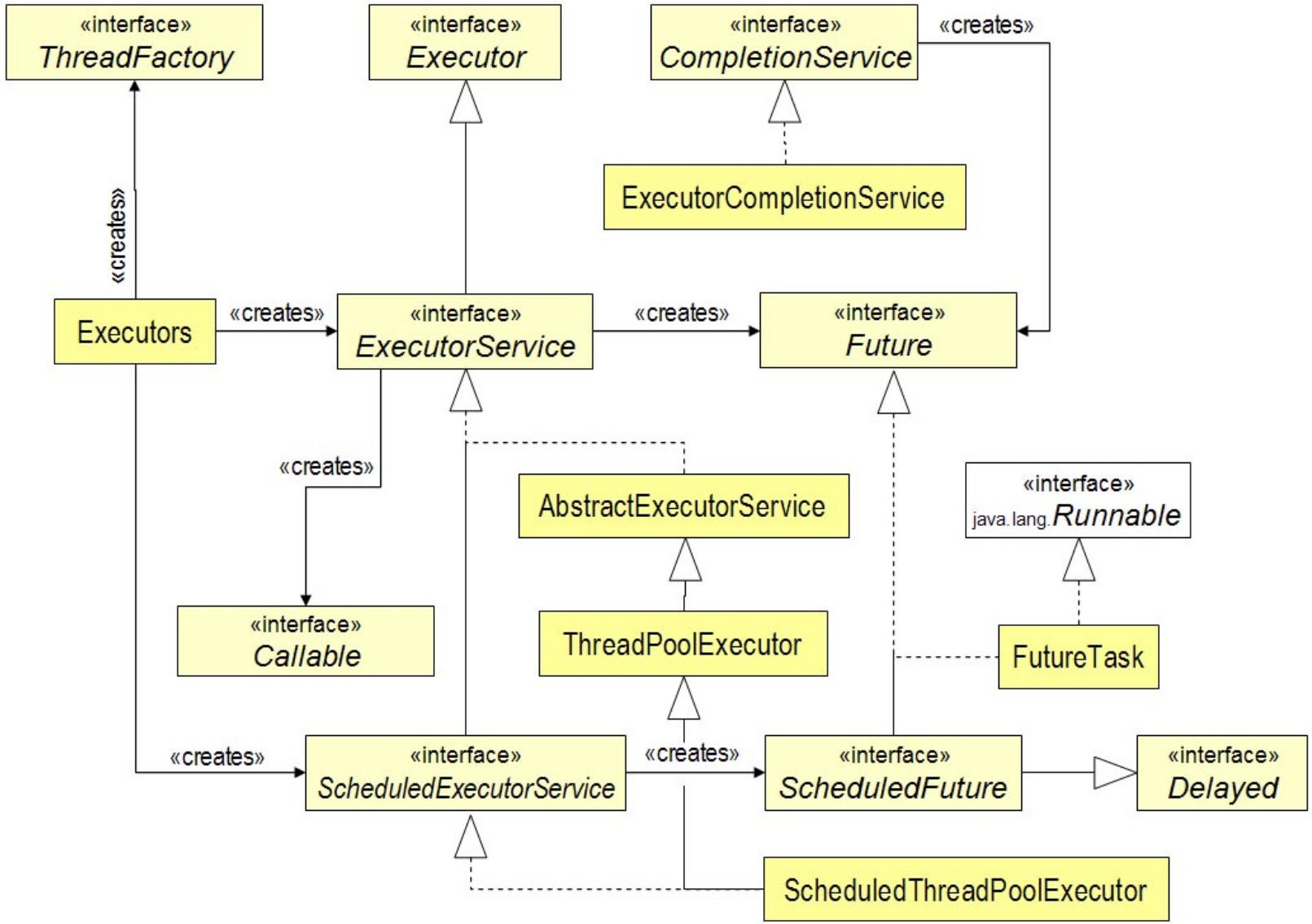
- Classes e interfaces para aplicações concorrentes
  - Ferramentas para facilitar a criação de threads
  - Travas
  - Objetos de sincronização
  - Objetos atômicos
  - Novas coleções
- Alternativa antiga agora é programação em baixo-nível:
  - synchronized
  - wait(), notify() e notifyAll()
  - new Thread(), join(), etc.
- Pacote novo obtém mesmos resultados de forma mais simples, segura e eficiente
  - Primeira opção a ser considerada no desenvolvimento de aplicações concorrentes.

O que contém

- Gerenciamento de tarefas concorrentes
  - Classes e interfaces para criação, execução e agendamento de tarefas assíncronas (threads e grupos).
- Coleções concorrentes
  - Filas, filas bloqueáveis e versões concorrentes de coleções.
- Variáveis atômicas
  - Classes para manipulação atômica de tipos primitivos e referências
- Sincronizadores
  - Semáforos, barreiras, trincos, permutadores
- Travas e monitores
  - Alternativa mais flexível a synchronized, wait() e notify()
- Unidades de tempo
  - Formas de marcar períodos e instantes com mais precisão.

Framework de execução

java.util.concurrent.*



Interface Executor

- Encapsula a execução de tarefas Runnable
- Implementa o padrão de *design* Command
  - Declara um método: **`void execute(Runnable r);`**
  - Cada implementação pode oferecer um comportamento diferente no seu método execute().
- Exemplo de implementação:

```
class UmThreadPorTarefa implements Executor {
   public void execute( Runnable tarefa ) {
       new Thread(tarefa).start();
   }
}

iniciarThreads( new UmThreadPorTarefa() );
```

Interface Executor

- Substitui execução tradicional de tarefas

```
( new Thread( new RunnableUm()   ) ).start();
( new Thread( new RunnableDois() ) ).start();
```

- O mesmo pode ser realizado usando Executor, com mais encapsulamento e possibilidade de maior reuso
  - É preciso usar uma implementação existente de Executor

```
void iniciarThreads( Executor executor ) {
   executor.execute( new RunnableUm() );
   executor.execute( new RunnableDois() );
}
```

  - Veja exemplo no slide anterior

Classe utilitária Executors

- A classe **Executors** contém diversos métodos de fábrica e utilitários para execução de *threads*.
- Métodos que retornam alguma implementação de **ExecutorService**
  - **`newCachedThreadPool()`**
  - **`newFixedThreadPool(int tamanhoFixo)`**
  - **`newSingleThreadExecutor()`**
- Métodos que retornam um **ThreadFactory**
  - **`defaultThreadFactory()`**
- Métodos que retornam objetos **Callable**
  - **`callable(Runnable tarefa)`**

Interface ThreadFactory

- Encapsula a criação de um Thread
- Uma implementação simples de **ThreadFactory** é

```
class SimpleThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable tarefa) {
        Thread t = new Thread(tarefa);
        // configura t (nome, prioridade, etc.)
        return t;
    }
}
```

- É usado para configurar objetos **ExecutorService**.
  - Todos usam um ThreadFactory *default* pré-configurado que pode ser obtido com o método **`defaultThreadFactory()`**
  - Um outro ThreadFactory pode ser passado na construção

Interface ExecutorService

- Estende Executor com métodos para controle do ciclo de vida de uma execução e execuções assíncronas
- Método herdado

```
void execute(Runnable tarefa)
```

- Métodos de execução

```
Future<?> submit(Runnable tarefa)
<T> List<Future<T>>
    invokeAll(Collection<Callable<T>> tarefas)
<T> T invokeAny(Collection<Callable<T>> tarefas)
```

- Métodos de finalização

```
void shutdown()
boolean isShutdown()
boolean isTerminated()
List<Runnable> shutdownNow()
```

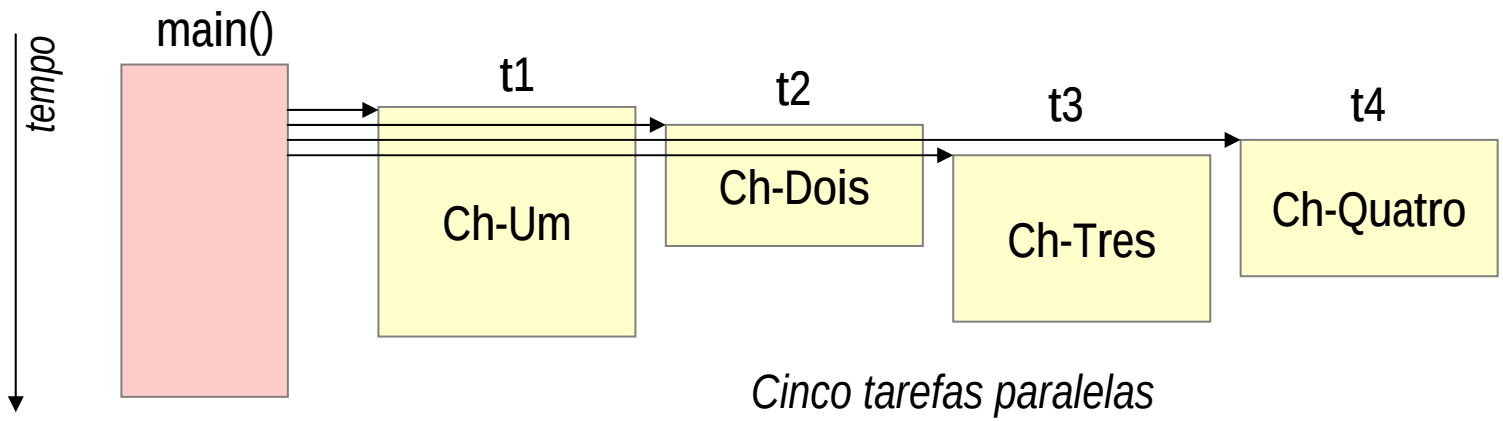
ThreadPoolExecutor

- Principal implementação de ExecutorService
- Implementações pré-configuradas obtidas através dos métodos de Executors
  - **`newCachedThreadPool()`**
  - **`newFixedThreadPool(int tamanhoFixo)`**
  - **`newSingleThreadExecutor()`**
- Pode-se configurar manualmente com ajustes finos para adequação a diversos cenários

Exemplo: cache de threads

- Usando **Executors.newCachedThreadPool()**

```
System.out.println("Execução de 4 threads em paralelo");
ExecutorService e =
     Executors.newCachedThreadPool();
e.execute( new TarefaConcorrente("Ch-Um") );
e.execute( new TarefaConcorrente("Ch-Dois") );
e.execute( new TarefaConcorrente("Ch-Tres") );
e.execute( new TarefaConcorrente("Ch-Quatro") );
e.shutdown(); // finaliza quando todos terminarem
```

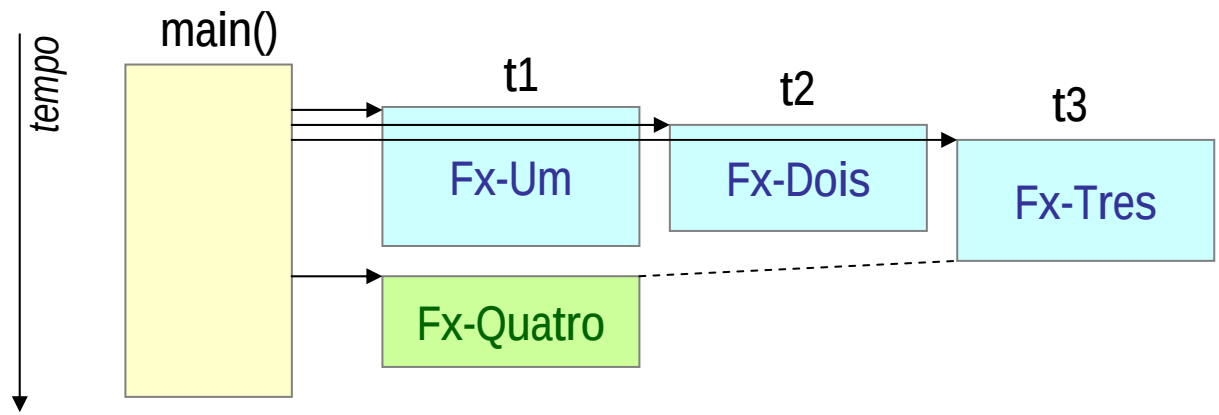


Cinco tarefas paralelas

Exemplo: *pool* fixo de threads

- Usando **Executors.newFixedThreadPool(limite)**
  - Threads que ultrapassarem limite máximo esperam liberação

```
System.out.println("4 tentando rodar em pool de 3");
ExecutorService e =
     Executors.newFixedThreadPool(3);
e.execute( new TarefaConcorrente("Fx-Um") );
e.execute( new TarefaConcorrente("Fx-Dois") );
e.execute( new TarefaConcorrente("Fx-Tres") );
e.execute( new TarefaConcorrente("Fx-Quatro") );
e.shutdown(); // finaliza quando todos terminarem
```



Só "cabem" três tarefas novas executando de cada vez

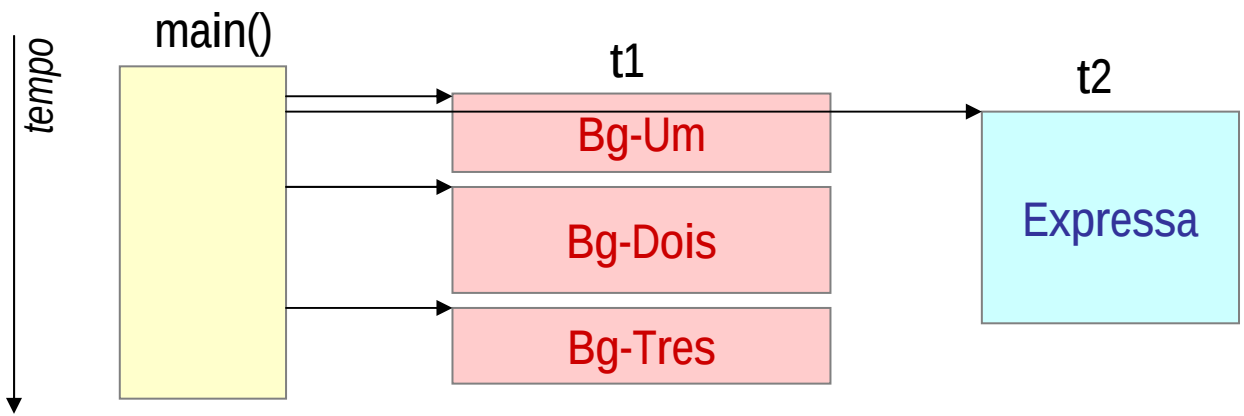
Exemplo: um servidor

```
class ArquivoServer {
  public static void main(String[] args) throws Exception {
    Executor pool = Executors.newFixedThreadPool(5);
    System.out.print("Esperando cliente...");
    ServerSocket socket = new ServerSocket(9999);
    while (true) {
      final Socket cliente = socket.accept();
      String ip = socket.getInetAddress().getHostAddress();
      System.out.println("Cliente de "+ip+" conectado!");
      Runnable tarefa = new Runnable() {
        public void run() {
          processarSocket(cliente);
        }
      };
      pool.execute(tarefa);
    }
  }
  public static void processarSocket(Socket s) {...}
}
```

Exemplo: tarefas em background

- Usando **newSingleThreadExecutor()**
  - Com dois serviços seqüenciais em paralelo

```
System.out.println("3 em seqüência em paralelo com um");
ExecutorService e1 = Executors.newSingleThreadExecutor();
ExecutorService e2 = Executors.newSingleThreadExecutor();
e1.execute( new TarefaExpressa() );
e2.execute( new TarefaConcorrente("Bg-Um"));
e2.execute( new TarefaConcorrente("Bg-Dois"));
e2.execute( new TarefaConcorrente("Bg-Tres"));
e1.shutdown();
e2.shutdown();
```



Só há dois novos threads paralelos

Callable

- Função semelhante à de Runnable, mas com método que pode retornar valor ou exceção

```
interface Callable<V> {
    V call() throws Exception;
}
```

- Callable pode sempre ser usado no lugar de Runnable.
  - Há métodos na classe Executors para encapsular um no outro e permitir o uso de ambos nas mesmas tarefas
- Exemplo de uso

```
Callable um =
   Executors.callable(new Tarefa("Runnable Task"));
ExecutorService e = Executors.newCachedThreadPool();
Future fum = e.submit(um);
```

equivalente a execute() mas retorna valor, se houver

Future

- Representa o resultado de uma computação assíncrona
  - Permite saber se tarefa foi concluída, esperar sua conclusão, recuperar seu resultado e tentar cancelar a computação.
- Método **get()** bloqueia até obter o resultado final.
- Normalmente usa-se Future como resultado de métodos submit() de ExecutorService que rodam tarefas Callable

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<File> future = executor.submit(new Callable<File>() {
    public File call() {
        File arquivo = null;
        // tarefa demorada para achar arquivo
        return arquivo;
    }
});
fazerOutrasCoisasEmParalelo();
File encontrado = future.get();
```

FutureTask

- Implementação de Future que também implementa a interface Runnable
  - Pode assim ser passada como argumento do método execute() de um Executor ou delegar chamadas para run()
- Outra forma de implementar o exemplo do slide anterior

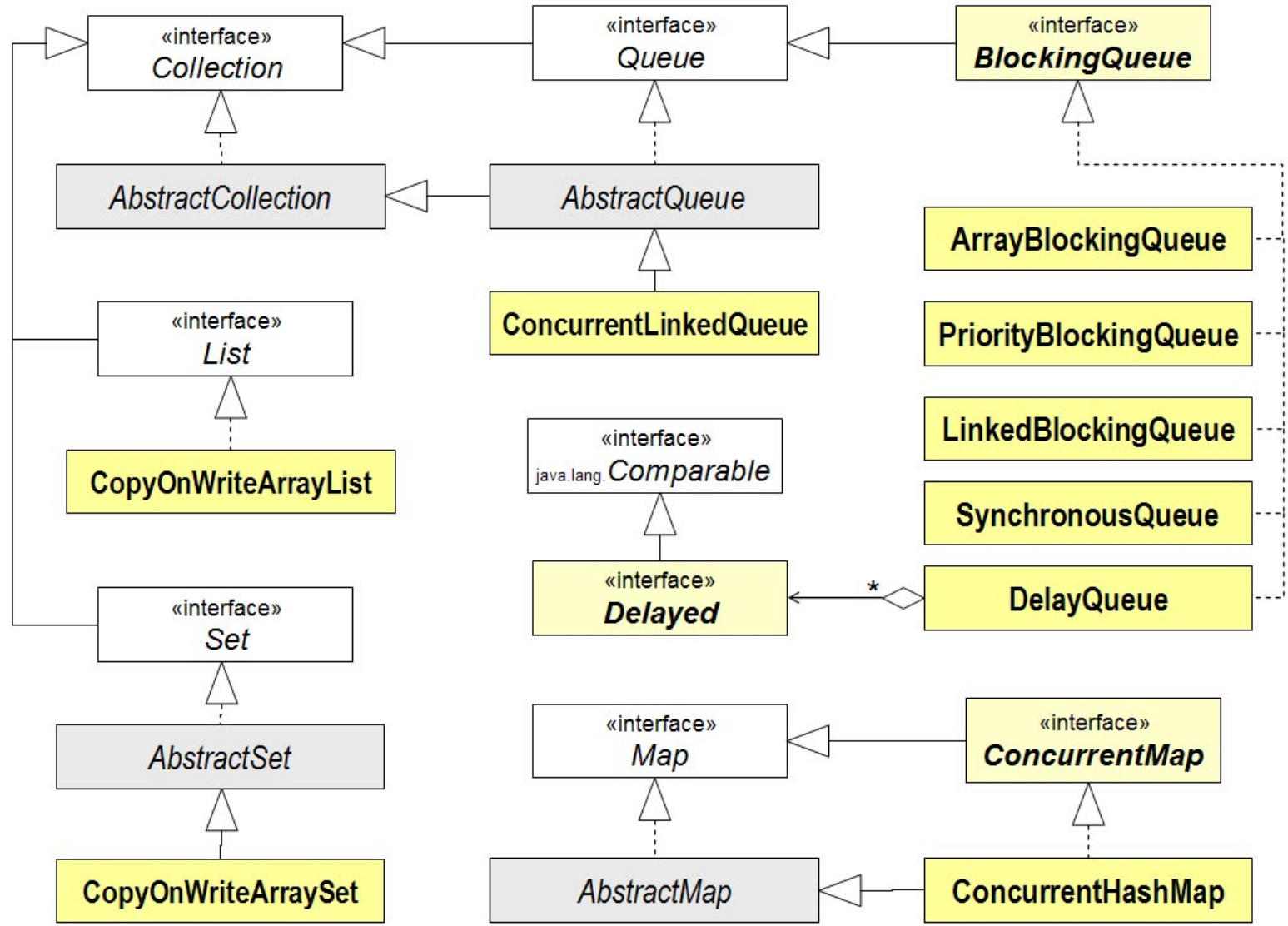
```
ExecutorService executor = Executors.newCachedThreadPool();
FutureTask<File> future =
    new FutureTask<File>(new Callable<File>() {
        public File call() {
            ... // mesmo código
        }
    });
executor.execute(future);
fazerOutrasCoisasEmParalelo();
File encontrado = future.get();
```

Agendamento

- Encapsulam o uso de Timers e TimerTasks
- Interface **Delayed**
  - Método getDelay() retorna atraso associado a objeto que implementa esta interface
- **ScheduledFuture**
  - Objeto Future de um objeto Delayed
- **ScheduledExecutorService**
  - Serviços executados periodicamente ou depois de um período pré-determinado
- **ScheduledThreadPoolExecutor**
  - Principal implementação de ScheduledExecutorService
  - Métodos de Executors fornecem objetos pré-configurados com mesma semântica de pools, caches e seqüências usadas para serviços sem atraso.

Coleções concorrentes

java.util.concurrent.*

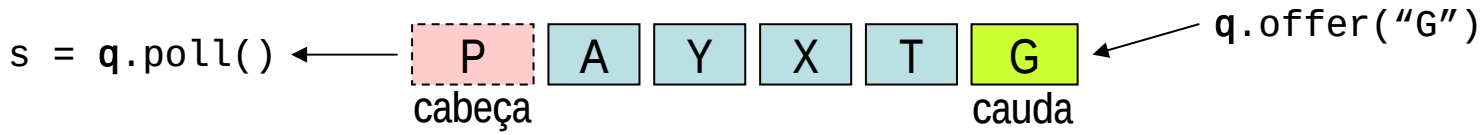


Coleções concorrentes

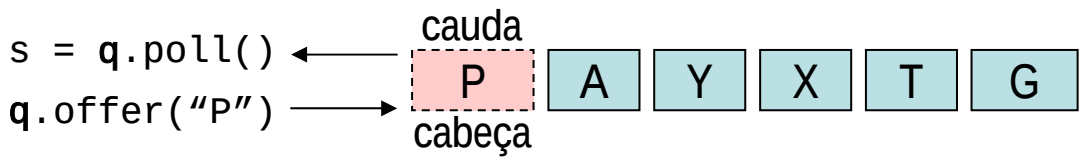
- Filas
  - Java 5.0, Queue (fila) e PriorityQueue no **java.util** (não são concorrentes); LinkedList também implementa Queue
  - **java.util.concurrent**: BlockingQueue (e várias implementações) e ConcurrentLinkedQueue
- Listas e conjuntos
  - CopyOnWriteArrayList
  - CopyOnWriteArraySet
- Mapas
  - Interface ConcurrentMap
  - ConcurrentHashMap
- Iteradores
  - Maioria não congela durante iteração: podem não refletir atualizações realizadas em paralelo
  - Não causam ConcurrentModificationException

Filas (queues)

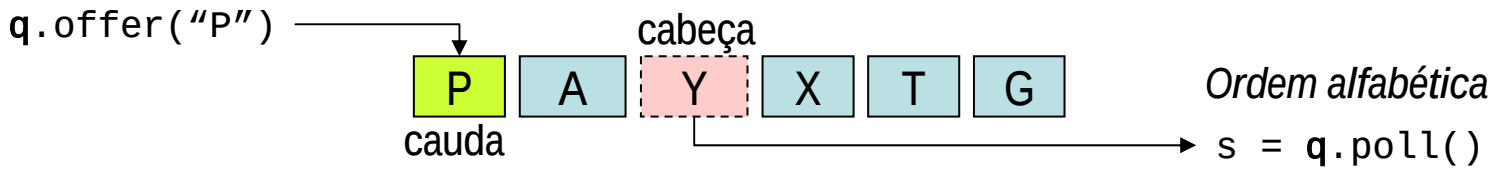
- Estrutura de dados onde objetos são removidos de forma ordenada
  - Ordenação FIFO (first-in-first-out) é a mais comum: primeiro que chega é primeiro que sai



  - Ordenação LIFO (last-in-first-out): pilha



  - Outras ordenações: ordem natural, algoritmos, tempo de vida, etc.



- A cauda (**tail**) da fila é por onde entram novos elementos
- O elemento que é removido da fila é sempre a cabeça (**head**)

java.util.Queue

- Incluída na Collections API a partir do Java 5.0

```
interface Queue<E> extends Collection<E> {
    E element();           // lê mas não remove cabeça
    E peek();              // lê mas não remove cabeça
    E poll();              // lê e remove cabeça
    boolean offer(E o);    // insere elemento da fila
    E remove();            // remove elemento da fila
}
```

- Ordenação depende da implementação utilizada
- Exemplo de uso:

```
Queue<String> fila = new LinkedList<String>();
fila.offer("D"); // cabeça, em fila FIFO
fila.offer("W");
fila.offer("X"); // cauda, em fila FIFO
System.out.println("Decapitando: " + fila.poll()); // D
System.out.println("Nova cabeça: " + fila.peek()); // W
```

Queue: implementações

- **LinkedList**
  - Lista encadeada de propósito geral usada para **pilhas**, **filas** e **deques**
  - Pode ser sincronizada:

```
List list =
   Collections.synchronizedList(new LinkedList(...));
```

- **ConcurrentLinkedQueue**
  - Fila concorrente com nós encadeados (ordem: FIFO)
- **PriorityQueue**
  - Ordenação baseada em algoritmo (Comparable ou Comparator)
- Implementações da subinterface java.util.concurrent.**BlockingQueue**
  - **ArrayBlockingQueue** (ordem: FIFO, tamanho fixo)
  - **LinkedBlockingQueue** (ordem: FIFO, tamanho variável)
  - **PriorityBlockingQueue** (ordem igual a PriorityQueue, tam. variável)
  - **DelayQueue** (ordem: tempo de vida, tamanho variável)
  - **SynchronousQueue** (sem ordenação, tamanho zero)

Interface BlockingQueue

- Acrescenta operações **put()** e **take()** que esperam quando fica está cheia ou vazia, respectivamente

```
public interface BlockingQueue<E> extends Queue<E> {
    boolean add(E o);
    int drainTo(Collection<? super E> c);
    int drainTo(Collection<? super E> c, int maxElements)
    boolean offer(E o);
    boolean offer(E o, long timeout, TimeUnit unit);
    E  poll(long timeout, TimeUnit unit);
    void put(E o);  // método síncrono para colocar
    int remainingCapacity();
    E take();       // método síncrono para remover
}
```

Formas de fazer a mesma coisa?

- **`boolean add(E o)`** (herdado de Collection)
  - Adiciona um objeto na fila se for possível, e retorna *true*
  - Causa *IllegalStateException* se falhar
- **`boolean offer(E o)`**
  - Adiciona um objeto na fila se for possível, e retorna *true*
  - Retorna *false* se falhar
- **`boolean offer(E o, long timeout, TimeUnit unidade) throws InterruptedException`**
  - Adiciona um objeto na fila se for possível, e retorna *true*
  - Se não for, espera o *timeout* e se conseguir, retorna *true*
  - Se o *timeout* expirar retorna *false*
  - A espera pode ser interrompida causando *InterruptedException*
- **`void put(E o) throws InterruptedException`**
  - Adiciona um objeto na fila se for possível
  - Se não for, espera (bloqueia o *thread*) até que seja possível
  - A espera pode ser interrompida causando *InterruptedException*

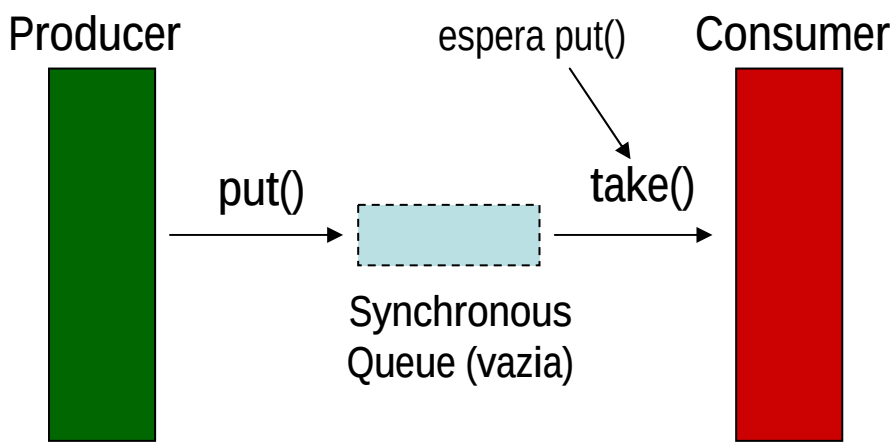
Exemplo: Produtor-Consumidor

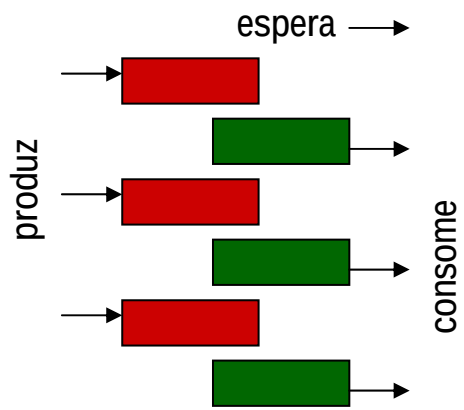
[SDK]

```
class Producer implements Runnable {
   private final BlockingQueue queue;
   Producer(BlockingQueue q) { queue = q; }
   public void run() { // try-catch omitido
     while(true) { queue.put("X"); }
   }
}
class Consumer implements Runnable {
   private final BlockingQueue queue;
   Consumer(BlockingQueue q) { queue = q; }
   public void run() { // try-catch omitido
     while(true) { System.out.println(queue.take()); }
   }
}
class Setup {
   void main() {
     BlockingQueue q = new SomeQueueImplementation();
     ExecutorService e = Executors.newCachedThreadPool();
     e.execute(new Producer(q));
     e.execute(new Consumer(q));
     e.execute(new Consumer(q));
   }
}
```

SynchronousQueue

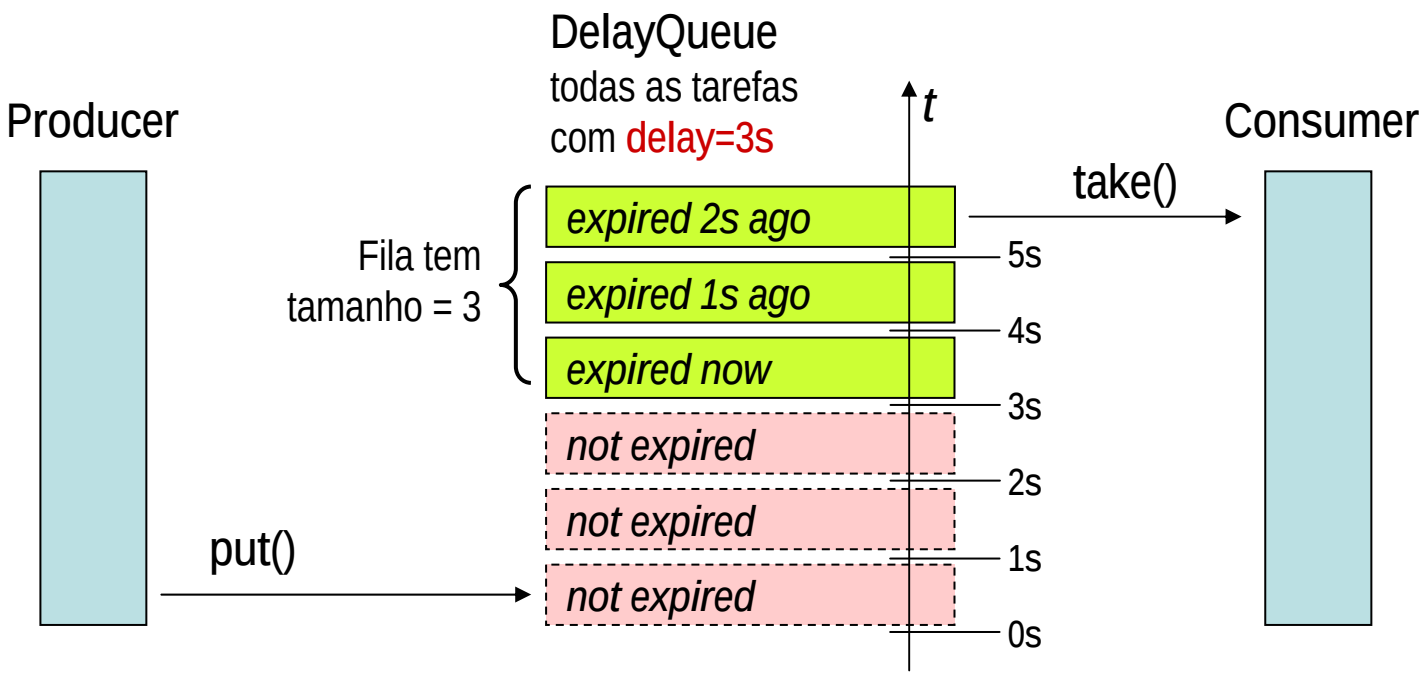
- Implementa uma pilha/fila sincronizada
  - Um objeto só é adicionado (**put**) quando houver outro thread querendo removê-lo (**take**)
  - Elementos são inseridos e removidos alternadamente
  - Fila é mero ponto de troca (*rendezvous*)
  - Não há elementos na fila: é uma fila vazia! Consequentemente peek() não funciona (nem Iterator)





DelayQueue

- Ordenação de objetos cujo atraso expirou há mais tempo
  - Objetos têm que implementar a interface **Delayed** (e seu método getDelay(TimeUnit) que retorna o atraso)
  - Objeto só pode ser removido quando atraso tiver expirado
  - Se nenhum objeto da fila tiver atraso expirado, fila comporta-se como se estivesse vazia



ConcurrentHashMap

- Implementação de Map e ConcurrentMap
  - Melhor escolha quando for preciso usar um Map em típicas aplicações concorrentes
  - Mesma especificação funcional que a classe Hashtable, porém permite maior concorrência
- Acesso exclusivo na gravação (remove, put), livre na leitura: nunca bloqueia o *thread* ao recuperar elementos (get)
  - Conseqüência: resultados das leituras podem não estar em dia com as alterações realizadas.
- Iteradores (Iterator ou Enumeration) comportam-se diferente
  - Retornam elementos que refletem o estado do mapa em algum ponto
  - Não causam ConcurrentModificationException
- O nível de concorrência máximo definido na construção
  - O *default* é 16
  - Só afeta atualizações e inserções

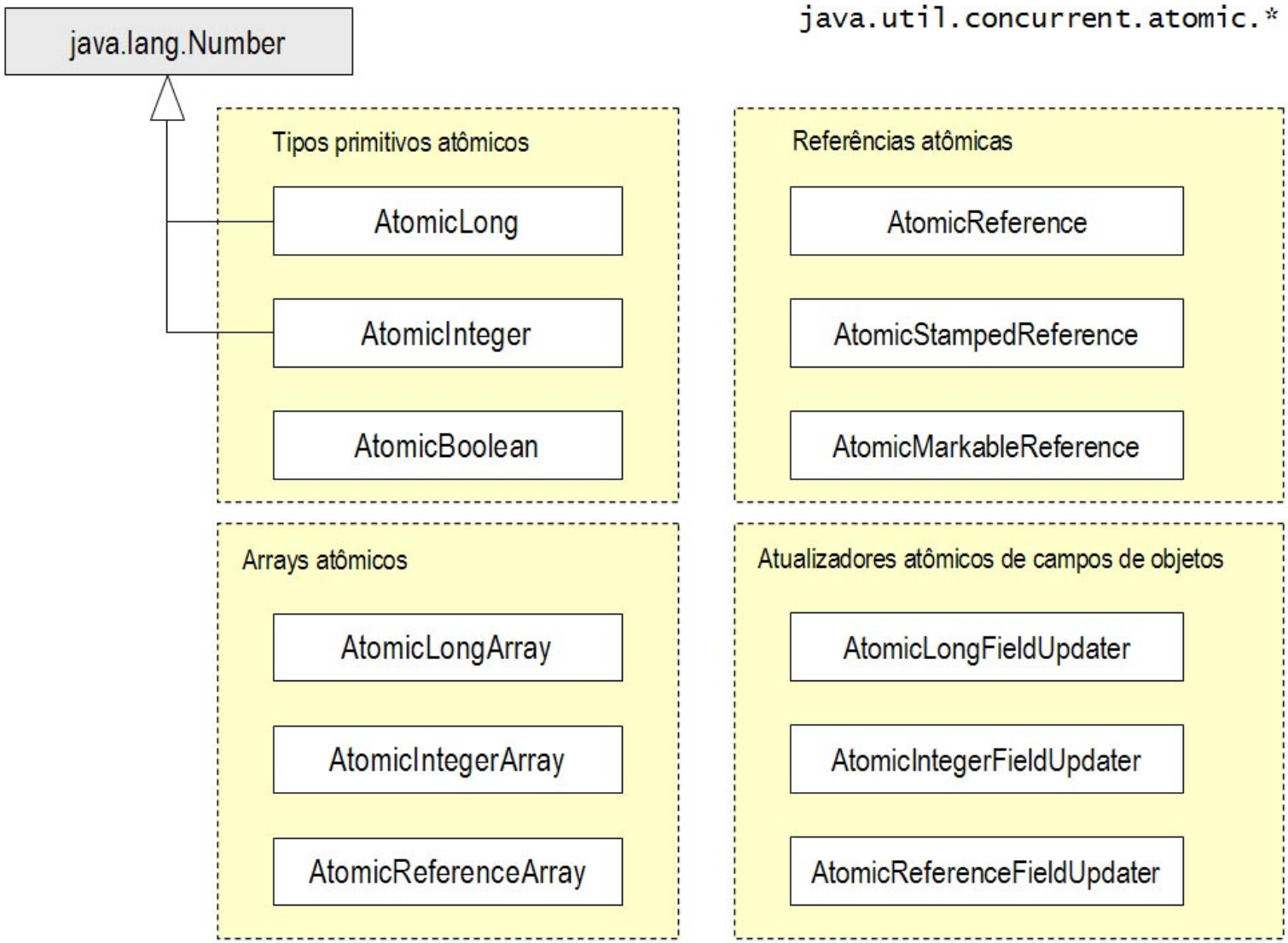
Sincronizadas vs. Concorrentes

- Sincronizadas (pré-Java 5) (thread-safe)
  - Controladas por única trava
  - Ex: java.util.Hashtable, Collections.synchronizedCollection(...) e os métodos similares para coleções específicas
  - Menos eficientes que concorrentes
- Concorrentes (thread-safe)
  - Têm travas diferenciadas para leitura e gravação: ex: ConcurrentHashMap
  - Mais escaláveis
  - Menos eficientes que coleções não sincronizadas
- Coleções não sincronizadas (não são thread-safe)
  - Mais eficientes que todas as outras
  - Devem ser preferidas quando não forem compartilhadas ou acessíveis apenas em trechos já sincronizados

Copy on Write Arrays

- **CopyOnWriteArrayList**
  - Implementação de List onde as operações de inserção, remoção e atualização causam a criação de uma nova cópia do *array* usado internamente para implementar a coleção.
  - Nenhuma sincronização é necessária
  - Iteradores nunca causam ConcurrentModificationException
- **CopyOnWriteArraySet**
  - Set que usa CopyOnWriteArrayList para todas as operações.
- Aplicações: listas de *event-handlers*
  - Alterações levam um tempo proporcional ao tamanho do *array*
  - Ideal para aplicações onde atualizações são raras mas a pesquisa é freqüente
  - Ideal para listas pequenas

Variáveis atômicas



Variáveis atômicas

- Suportam programação *thread-safe* sem travas sobre variáveis escalares
  - Usadas como variáveis mutáveis em ambientes concorrentes
- Representam objetos atômicos voláteis
- Estendem o conceito de atomicidade a operações unárias de comparação, atribuição, alteração condicional e incremento/decremento pré- e pós-fixados
- Uso

```
AtomicInteger aInt = new AtomicInteger(4);

int[] array = {5, 6, 7};
AtomicIntegerArray aIntArray =
                    new AtomicIntegerArray(array);

Coisa pri = new Coisa(6, "primeira");
AtomicReference aref = new AtomicReference(pri);
```

Métodos comuns

- Todas as classes possuem quatro métodos em comum
- **`get()`**
  - Retorna o valor armazenado (mesmo que ler campo volátil)
  - Em *arrays*, **get()** requer parâmetro correspondente ao índice
- **`set(valor)` e `getAndSet(valor)`**
  - Inicializam o objeto com valor (mesmo que gravar campo volátil)
  - **getAndSet()** retorna o valor anterior.
- **`compareAndSet(esperada, recebida)`**
  - Compara a referência atual com uma esperada (==) e se for a mesma, muda o valor para a referência recebida atomicamente.
- **`weakCompareAndSet(esperada, recebida)`**
  - **compareAndSet()** como uma operação não volátil

Operações atômicas

- Booleanos e inteiros (int e long), arrays, referências
- Exemplo: elementos escalares (inteiros)

```
int x = 4;
AtomicInteger aInt = new AtomicInteger(x);
int past = aInt.getAndIncrement(); // x++
int pres = aInt.incrementAndGet(); // ++x;
```

- Exemplo: vetores (de inteiros)
  - É a única forma de manipular de forma completa com *arrays* voláteis, já que declarar um *array* volatile não garante a semântica de acesso volátil para os seus elementos.

```
int[] array = {5, 6, 7};
AtomicIntegerArray a = new AtomicIntegerArray(array);
a.set(1, 5); // mude a[1] para 5
a.compareAndSet(2, 7, 5); // se a[2] tiver 7, mude para 5
```

Exemplo [DL2]

- Único objeto *Random* compartilhado por muitas classes
  - Menos contenção
  - Mais rápido

```
class Random { // snippets
     private AtomicLong seed;
     Random(long s) {
          seed = new AtomicLong(s);
     }
     long next(){
          for(;;) {
               long s = seed.get();
               long nexts = s * ... + ...;
               if (seed.compareAndSet(s,nexts))
                    return s;
          }
     }
}
```

Fonte: [DL2] Doug Lea, slides sobre JSR-166

AtomicReference

- Referências que podem ser atualizadas atomicamente

```
class Coisa {
   int variavel;
   String nome;   ... }
Coisa pri = new Coisa(6, "primeira");
Coisa seg = new Coisa(0, "segunda");
AtomicReference aref = new AtomicReference(pri);
aref.compareAndSet(pri, new Coisa(9, "circular"));
```

- Referências rotuladas

```
AtomicMarkableReference m1 =
    new AtomicMarkableReference(pri, false);
AtomicMarkableReference m2 =
    new AtomicMarkableReference(seg, false);
boolean expired = m2.attemptMark(seg, true);

AtomicStampedReference r1 = new AtomicStampedReference(pri, -1);
AtomicStampedReference r2 = new AtomicStampedReference(seg, -1);
r1.attemptStamp(pri, 245);
r2.attemptStamp(seg, 244);
```

Exemplo: algoritmo otimista [DL2]

```
public class OptimisticLinkedList { // incomplete
    static class Node {
        volatile Object item;
        final AtomicReference<Node> next;
        Node(Object x, Node n) {
            item = x;
            next = new AtomicReference(n);
        }
    }
    final AtomicReference<Node> head = new AtomicReference<Node>(null);
    public void prepend(Object x) {
        if (x == null) throw new IllegalArgumentException();
        for(;;) {
            Node h = head.get();
            if (head.compareAndSet(h, new Node(x, h))) return;
        }
    }
    public boolean search(Object x) {
        Node p = head.get();
        while (p != null && x != null && !p.item.equals(x))
            p = p.next.get();
        return p != null && x != null;
    }
}
```

Fonte: [DL2] Doug Lea, slides sobre JSR-166

Updaters

- Realizam atualização atômica em campos voláteis de objetos
  - Estilo similar a *reflection*

```
Coisa pri =
    new Coisa(6, "primeira");
Coisa seg =
    new Coisa(0, "segunda");
```

```
class Coisa {
   String getNome() {...}
   void setNome(String n) {...}
}
```

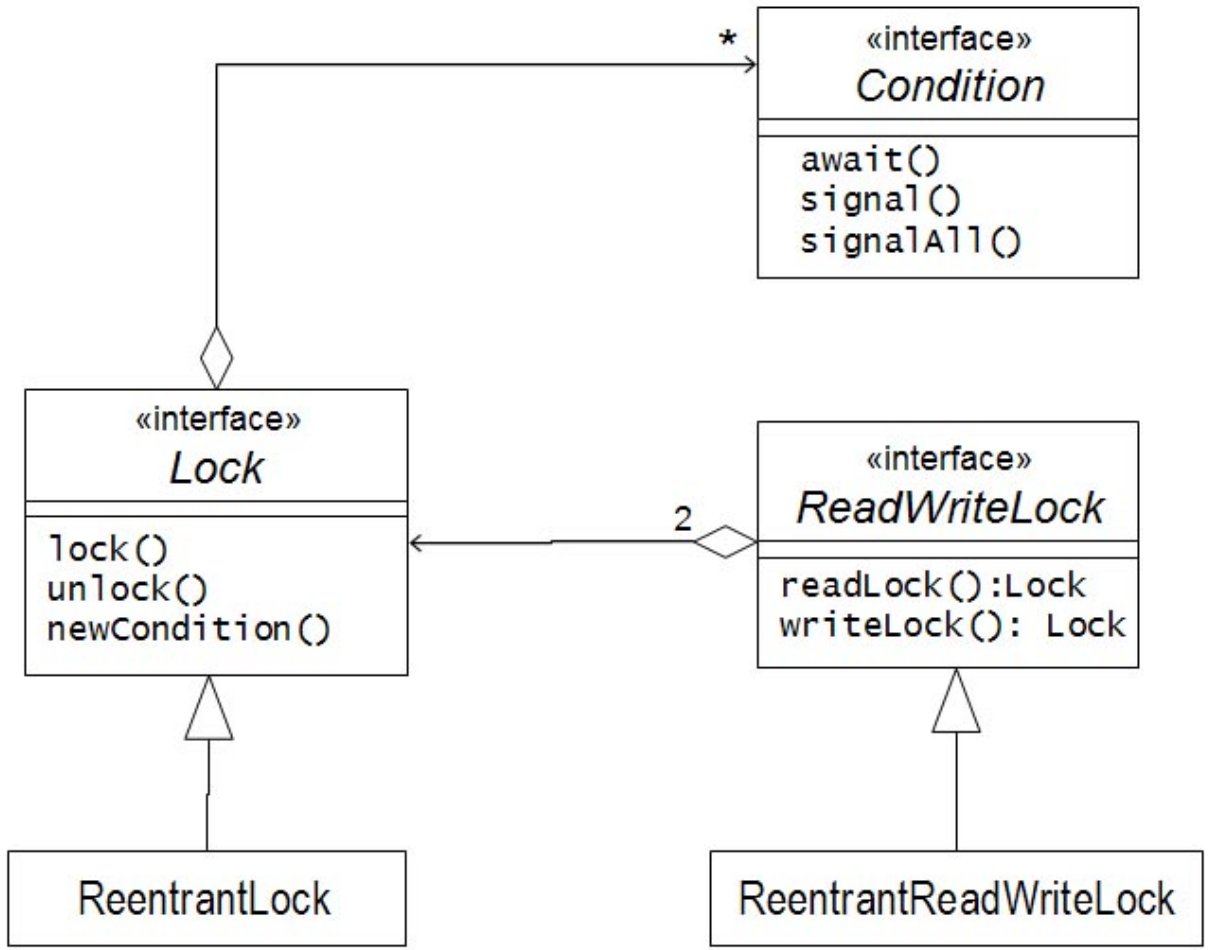
```
AtomicReferenceFieldUpdater<Coisa,String> nomeUpdater =
        AtomicReferenceFieldUpdater.newUpdater(Coisa.class,
                                               String.class,
                                               "nome");
String nomeAntigo = nomeUpdater.getAndSet(pri, "first");
```

Por que usar?

- Por que usar variáveis atômicas?
  - Para escrever código mais eficiente em multiprocessadores: máquinas virtuais irão utilizar a melhor implementação disponível na plataforma
- Para que servem?
  - Estruturas de dados não bloqueantes
  - Algoritmos otimistas
  - Redução de overhead e contenção quando alterações concentram-se em único campo
- Não abuse
  - Uso desnecessário tem impacto na performance

Travas

java.util.concurrent.locks.*



Travas de exclusão mútua

- A interface **Lock** é uma trava de exclusão mútua (mutex)
  - Permite a implementação de estratégias de travamento que diferem em semântica (travas reentrantes, travas justas, etc.)
  - Principal implementação (única disponível) é ReentrantLock que tem semântica de memória equivalente ao bloco *synchronized*.
- A interface **ReadWriteLock** contém um par de travas
  - Através de seus dois métodos obtém-se travas (Lock) para leitura e gravação.
  - Uma implementação disponível: ReentrantReadWriteLock.
- **Condition** é variável de condição associada a uma trava
  - A semântica de uso é similar a wait() e notify(),
  - Permite associar múltiplas condições a uma única trava

Trava simples

- A técnica básica consiste em chamar o método **lock()**, segui-lo por um bloco **try**, e terminar com um bloco **finally** para liberar a trava: **unlock()**
  - Mais flexível que **synchronized**
  - Menos compacto que **synchronized**

```
Lock trava = new ReentrantLock();
trava.lock();
try {
    // acessar recurso protegido pela trava
} finally {
    trava.unlock();
 }
```

ReentrantLock mantém uma contagem do aninhamento das travas: getHoldCount()

Trava condicional

- O método **tryLock()** “tenta” obter uma trava em vez de ficar esperando por uma
  - Se a tentativa falhar, o fluxo do programa segue (não bloqueia o *thread*) e o método retorna false

```
Lock trava = new ReentrantLock();
if (trava.tryLock(30, TimeUnit.SECONDS)) {
    try {
        // acessar recurso protegido pela trava
    } finally {
        trava.unlock();
    }
} else {
    // tentar outra alternativa
}
```

Obtenção que se pode interromper

- O método *lock()* não permite interrupção na tentativa de se obter uma trava
- Com **lockInterruptibly()** isto é possível

```
Lock trava = new ReentrantLock();
try {
    trava.lockInterruptibly();
    try {
        // acessar recurso protegido pela trava
    } finally {
        trava.unlock();
    }
} catch (InterruptedException e) {
    // tentativa de obter a trava foi interrompida
}
```

Condições

- Qualquer trava pode obter uma ou mais condições através do método de Lock **newCondition()**
  - Condition cheio = lock.newCondition();
- Condições contém métodos para espera e notificação: await() e signal()/signalAll()

```
interface Condition {
    void await()
    boolean await(long time, TimeUnit unit)
    long awaitNanos(long nanosTimeout)
    void awaitUninterruptibly()
    void signal()
    void signalAll()
}
```

Exemplo: monitor

[SDK]

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();
   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       lock.unlock();
     }
   }
   public Object take() throws InterruptedException { ... }
}
```

ReadWriteLock

- Uma trava para leitura e gravação
  - Mantém um par de travas associadas: uma para leitura apenas e a outra para gravação.
  - A trava de leitura pode ser usada simultaneamente por múltiplos *threads*, desde que não haja um *thread* com trava de gravação, que é exclusiva.
- Maior nível de concorrência e escalabilidade
- Uso:

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();
Lock readLock = rwLock.readLock();
Lock writeLock = rwLock.writeLock();
```

Exemplo

[SDK]

```
class RWDictionary {
    private final Map<String, Data>  m =
                                new TreeMap<String, Data>();
    private final ReentrantReadWriteLock rwl =
                                new ReentrantReadWriteLock();

    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();

    public Data get(String key) {
        r.lock();
        try { return m.get(key); }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
}
```

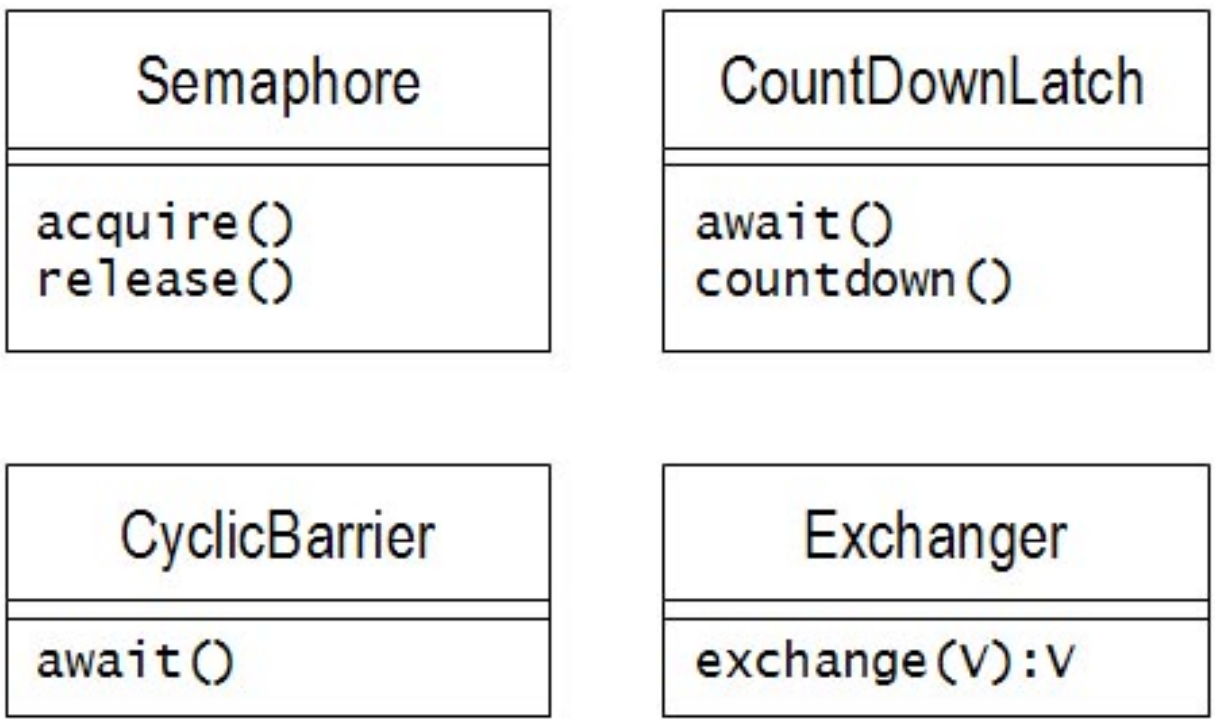
Sincronizadores

- Implementam algoritmos populares de sincronização
  - Facilita o uso da sincronização: evita a necessidade de usar mecanismos de baixo nível como métodos de Thread, wait() e notify()
- Estruturas disponíveis
  - Barreira cíclica: **CyclicBarrier**
  - Barreira de contagem regressiva: **CountDownLatch**
  - Permutador: **Exchanger**
  - Semáforo contador: **Semaphore**

Sincronizadores

- Interface essencial

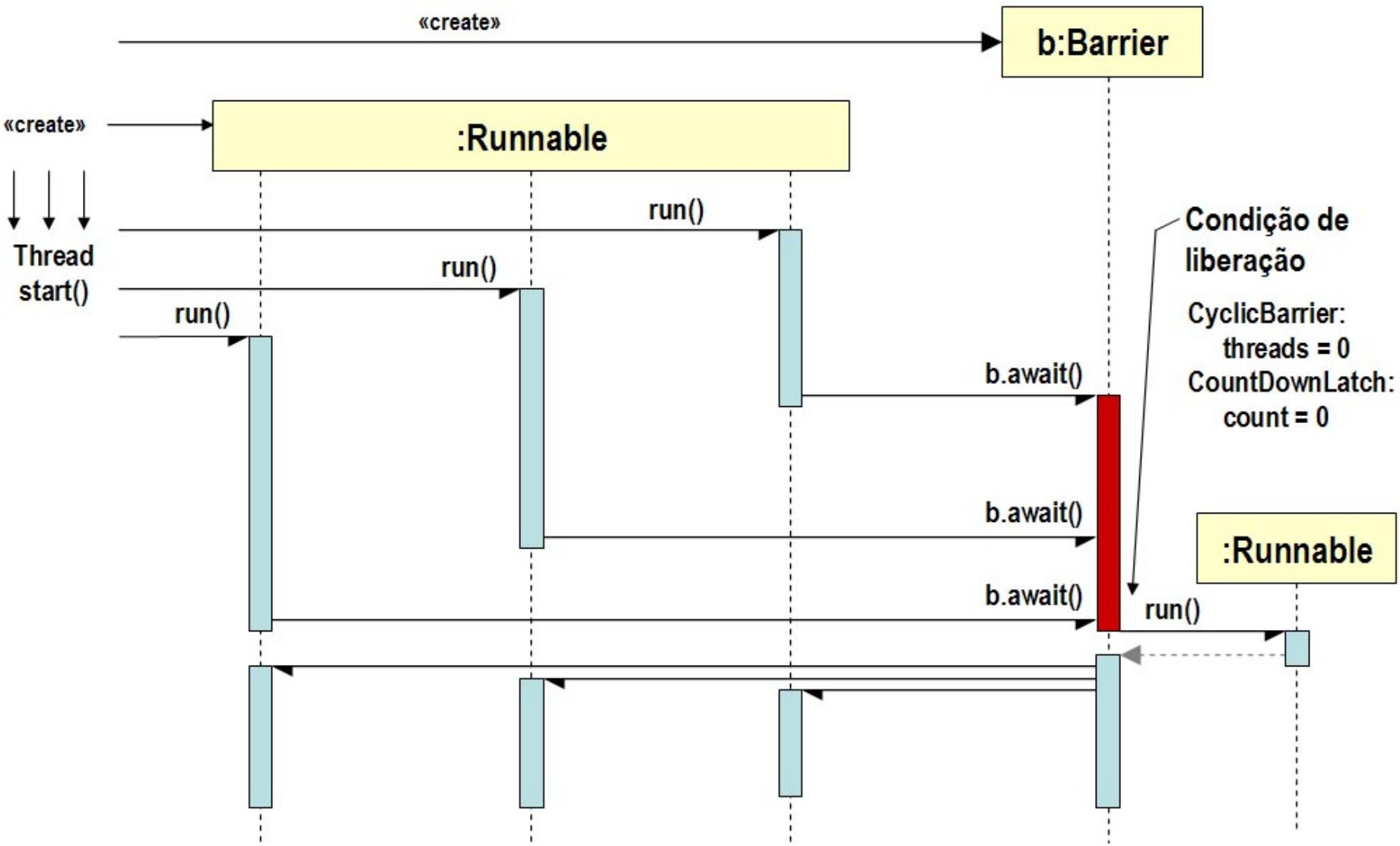
`java.util.concurrent.*`



Barreira

- Uma barreira é um mecanismo de sincronização que determina um ponto na execução de uma aplicação onde vários *threads* esperam os outros
  - Quando o *thread* chega no ponto de barreira, chama uma operação para indicar sua chegada e entra em estado inativo
  - Depois que um certo número de *threads* atinge a barreira, ou certa contagem zera, ela é vencida e os *threads* acordam
  - Opcionalmente, uma operação pode sincronamente executada na abertura da barreira antes dos *threads* acordarem
- Como implementar
  - Em baixo nível, usando o método **join()** para sincronizar com threads que terminam, ou usar **wait()** no ponto de encontro e **notify()** pelo último *thread* para vencer a barreira
  - Em java.util.concurrent: **CyclicBarrier** e **CountDownLatch**

Barreira


«create»
b:Barrier
«create»
:Runnable
Thread
start()
run()
run()
run()
b.await()
Condição de
liberação
CyclicBarrier:
threads = 0
CountDownLatch:
count = 0
b.await()
:Runnable
b.await()
run()

Interface de CyclicBarrier

```
public class CyclicBarrier {
    public CyclicBarrier(int parties);
    public CyclicBarrier(int parties,
                         Runnable barrierAction);

    public int await( ) throws InterruptedException,
                                BrokenBarrierException;
    public int await(long timeout, TimeUnit unit)
                        throws InterruptedException,
                                BrokenBarrierException,
                                TimeoutException;
    public void reset( );
    public boolean isBroken( );
    public int getParties( );
    public int getNumberWaiting( );
}
```

Barreira cíclica: **CyclicBarrier**

- Como criar
  - `CyclicBarrier b = new CyclicBarrier(n)`
  - `CyclicBarrier b = new CyclicBarrier(n, ação)`
- A barreira é criada especificando-se
  - **Número de *threads n*** necessários para vencê-la
  - **Uma *ação*** (*barrier action*) para ser executada sincronamente assim que a barreira for quebrada e antes dos *threads* retomarem o controle: implementada como um objeto **Runnable**
- Cada *thread* tem uma referência para uma instância ***b*** da barreira e chamar ***b.await()*** quando chegar no ponto de barreira
  - O *thread* vai esperar até que todos os *threads* que a barreira está esperando chamem seu método ***b.await()***
- Depois que os *threads* são liberados, a barreira pode ser reutilizada (por isto é chamada de cíclica)
  - Para reutilizar, chame o método ***b.reset()***

Exemplo (parte 1)

```
public class BarrierDemo {
  volatile boolean done = false;
  final Double[][] despesas = ...;
  volatile List<Double> parciais =
        Collections.synchronizedList(new ArrayList<Double>());

  public synchronized double somar(Double[] valores) { ... }

  class SomadorDeLinha implements Runnable {
     volatile Double[] dados; // dados de uma linha
     CyclicBarrier barreira;
     SomadorDeLinha(Double[] dados, CyclicBarrier barreira) {...}
     public void run() {
        while(!done) { // try-catch omitido
            double resultado = somar(dados);
            parciais.add(resultado); // guarda em lista
            System.out.printf("Parcial R$%(.2f\n", resultado);
            barreira.await();
        }
     }
  }    ...
```

Ponto de barreira

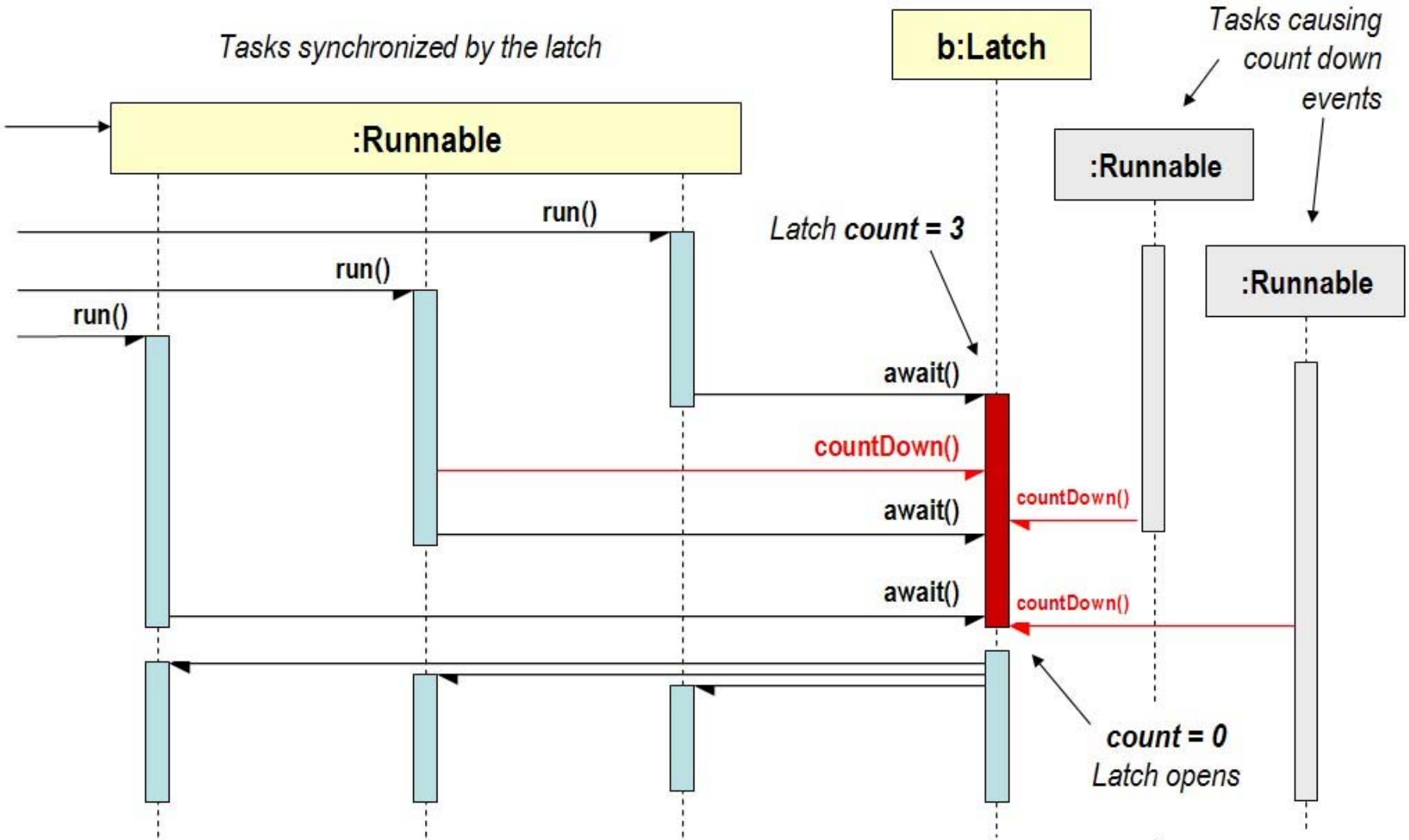
Exemplo (parte 2)

```
...

  class SomadorTotal implements Runnable {
     public void run() {
        Double[] array = parciais.toArray(new Double[5]);
        System.out.printf("Total R$%(.2f\n", somar(array));
        done = true;
     }
  }

  public void executar() {
     ExecutorService es =
          Executors.newFixedThreadPool(despesas.length);
     CyclicBarrier barrier =
          new CyclicBarrier(despesas.length, new SomadorTotal());
     for(Double[] linha: despesas)
       es.execute(new SomadorDeLinha(linha, barrier));
     es.shutdown();
  }
}
```

Trinco de contagem regressiva: **CountDownLatch**

- Um **CountDownLatch** é um tipo de barreira
  - É inicializado com valor inicial para um contador regressivo
  - *Threads* chamam **await()** e esperam a contagem chegar a zero
  - Outros *threads* podem chamar **countDown()** para reduzir a contagem
  - Quando a contagem finalmente chegar a zero, a barreira é vencida (o trinco é aberto)
- Pode ser usado no lugar de CyclicBarrier
  - Quando liberação depender de outro(s) fator(es) que não seja(m) a simples contagem de *threads*

Barreira com CountDownLatch



Interface de CountDownLatch

```
public class CountDownLatch {
    public CountDownLatch(int count);

    public void await( )
                throws InterruptedException;
    public boolean await(long timeout,
                         TimeUnit unit)
                throws InterruptedException;

    public void countDown( );
    public long getCount( );
}
```

*Retorna **true** se o retorno ocorre porque o trinco foi aberto*

*Retorna **false** se o retorno foi devido a timeout*

Mesmo exemplo usando Latch (1)

```
public class LatchDemo {
  ...
  private CountDownLatch trinco;
  public synchronized double somar(Double[] valores) {
     double subtotal = 0;
     for(Double d: valores) subtotal += d;
     if (trinco.getCount() != 0) trinco.countDown();
     return subtotal;
  }
  class SomadorDeLinha implements Runnable {
     volatile Double[] dados; // dados de uma linha
     SomadorDeLinha(Double[] dados) {...}
     public void run() {
        while(!done) { // try-catch omitido
            double resultado = somar(dados);
            parciais.add(resultado); // guarda em lista
            System.out.printf("Parcial R$%(.2f\n", resultado);
            trinco.await();
            done = true;
        }
     }
   }    ...
```

Exemplo usando Latch (parte 2)

```
...

class SomadorTotal implements Runnable {
   public void run() {
      Double[] array = parciais.toArray(new Double[5]);
      System.out.printf("Total R$%(.2f\n", somar(array));
   }
}

public void executarDemo() {
   ExecutorService es =
       Executors.newFixedThreadPool(despesas.length);
   trinco = new CountDownLatch(despesas.length);
   for(Double[] linha: despesas)
       es.execute(new SomadorDeLinha(linha));
   es.shutdown();
   trinco.await();  // try-catch omitido
   done = true;
   (new Thread(new SomadorTotal())).start(); // calculo final
}
```

A chamada final a este método fará a contagem chegar a zero, liberando a barreira

Permutador: **Exchanger**

- Serve para trocar objetos de um mesmo tipo entre threads em determinado ponto
  - Um *thread* chama método **exchange()** com um objeto;
  - Se outro *thread* tiver chamado **exchange()** e estiver esperando, objetos são trocados ambos os threads são liberados
  - Não havendo outro *thread* esperando no **exchange()**, o thread torna-se inativo e espera até que um apareça, a menos que haja interrupção (ou *timeout*)

```
public class Exchanger<V> {
    public Exchanger( );
    public V exchange(V x) throws InterruptedException;
    public V exchange(V x, long timeout, TimeUnit unit)
                        throws InterruptedException,
                               TimeoutException;
}
```

- A classe **Exchanger** pode ser usada em aplicações que usariam um SynchronousQueue

Exemplo (parte 1)

[SDK]

```
class FillAndEmpty {
   Exchanger<DataBuffer> exchanger = new Exchanger();
   DataBuffer initialEmptyBuffer = ... a made-up type
   DataBuffer initialFullBuffer = ...

   class FillingLoop implements Runnable {
     public void run() {
       DataBuffer currBuffer = initialEmptyBuffer;
       try {
         while (currBuffer != null) {
           addToBuffer(currBuffer);
           if (currBuffer.full())
             currBuffer = exchanger.exchange(currentBuffer);
         }
       } catch (InterruptedException ex) { ... }
     }
   } ...
```

Espera ou liberа
(depende de quem chegar primeiro)

Exemplo (parte 2)

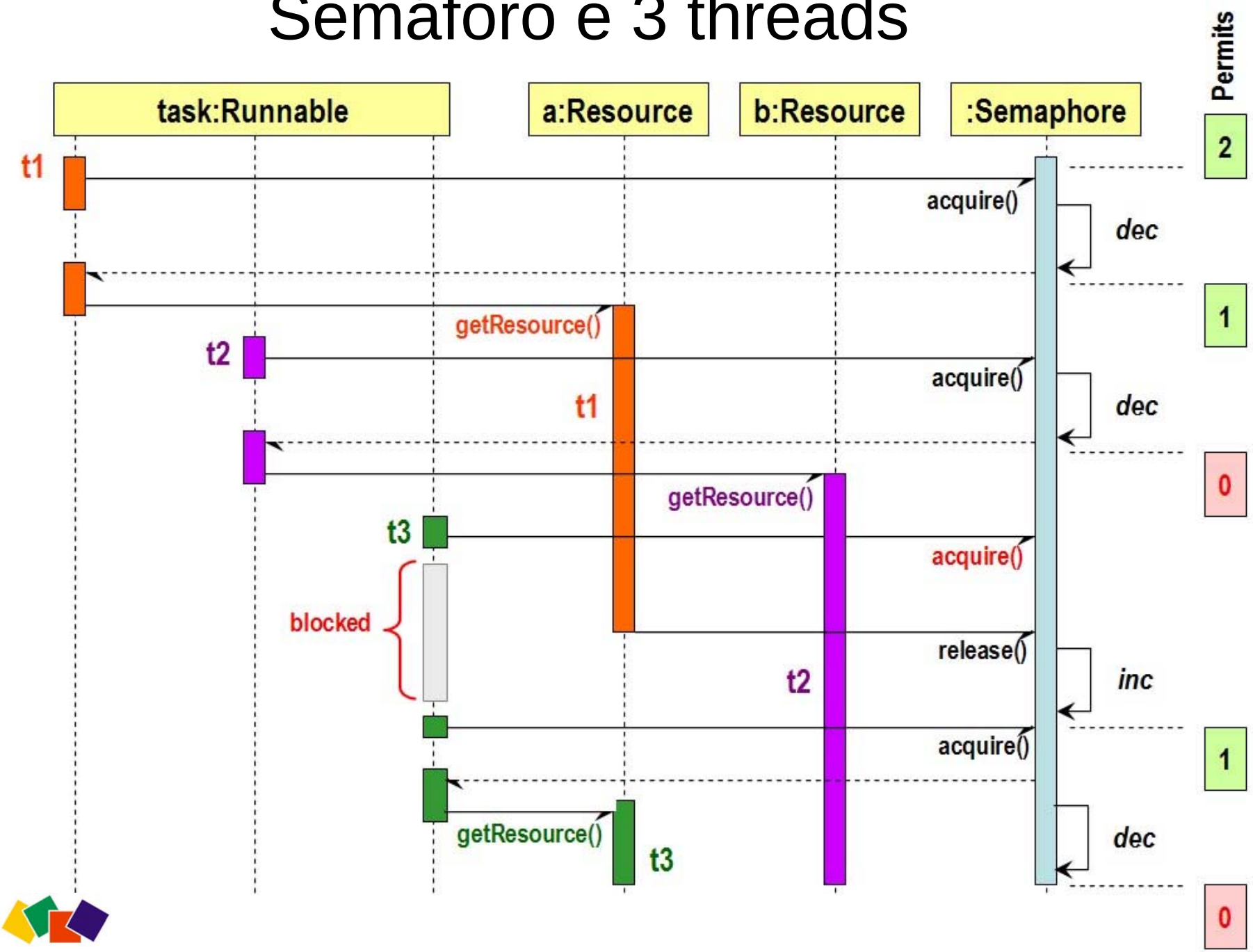
[SDK]

```
...
   class EmptyingLoop implements Runnable {
     public void run() {
       DataBuffer currBuffer = initialFullBuffer;
       try {
         while (currBuffer != null) {
           takeFromBuffer(currBuffer);
           if (currBuffer.empty())
             currBuffer = exchanger.exchange(currBuffer);
         }
       } catch (InterruptedException ex) { ... handle ...}
     }
   }
   void start() {
     new Thread(new FillingLoop()).start();
     new Thread(new EmptyingLoop()).start();
   }
 }
```

Semáforo

- Mecanismo de sincronização entre *threads* usado para distribuir um número limitado de permissões de acesso.
  - Usado para distribuir permissões de acesso para *threads* que podem acessar algum recurso compartilhado
- Funciona como uma trava com contador
  - Inicializado com um número fixo de permissões
  - Se número de permissões for 1 (semáforo binário), comporta-se como uma trava **mutex** (mutual exclusion)
- Possui duas operações básicas
  - **aquisição**: cede uma permissão (e decrementa o contador)
  - **liberação**: devolve a permissão (aumentando o contador)
  - Se a contagem chegar a zero, novas permissões não são distribuídas até que haja pelo menos uma liberação

Semáforo e 3 threads



Interface de Semaphore

```
public class Semaphore {
    public Semaphore(long permits);
    public Semaphore(long permits, boolean fair);

    public void acquire( ) throws InterruptedException;
    public void acquireUninterruptibly( );
    public void acquire(long permits)
                              throws InterruptedException;
    public void acquireUninterruptibly(long permits);
    public boolean tryAcquire( );
    public boolean tryAcquire(long timeout, TimeUnit unit);
    public boolean tryAcquire(long permits);
    public boolean tryAcquire(long permits,
                                long timeout, TimeUnit unit);
    public void release(long permits);
    public void release( );
    public long availablePermits( );
}
```

Semaphore

- Para criar. Exemplo: **new Semaphore(5, true)**
  - **5** é número de permissões
  - **true** sinaliza uso de algoritmo justo
- Uso típico: Thread chama **acquire()** para obter acesso, e
  - Passa se houver permissões, semáforo decrementa, usa recurso, chama **release()**
  - Bloqueia a *thread* se a contagem estiver em zero): espera liberação
- Algumas variações de acquire()
  - **acquire**(): pode ser interrompido, bloqueia thread
  - **acquireUninterruptibly**(): não pode ser interrompido
  - **tryAcquire**(): retorna false se não houver permissões
  - **tryAcquire**(timeout): espera timeout e retorna false de expirar

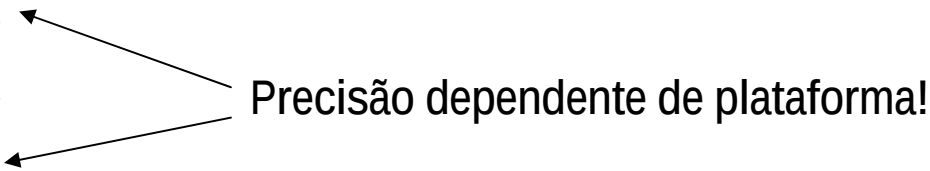
Exemplo: Semaphore

[SDK]

```
class Pool {
   private static final MAX_AVAILABLE = 100;
   private final Semaphore available =
                     new Semaphore(MAX_AVAILABLE, true);
   public Object getItem() throws InterruptedException {
     available.acquire();
     return getNextAvailableItem();
   }
   public void putItem(Object x) {
     if (markAsUnused(x))
       available.release();
   }
   Object[] items = ...
   boolean[] used = new boolean[MAX_AVAILABLE];
   synchronized Object getNextAvailableItem() {...}
   synchronized boolean markAsUnused(Object item) {...}
 }
```

Unidades de tempo

- **TimeUnit** é um Enum
- Elementos
  - **MICROSECONDS**
  - **MILLISECONDS**
  - **NANOSECONDS**
  - **SECONDS**



- Vários métodos
- Uso típico em métodos que aceitam argumentos de tempo

```
Lock lock = ...;
if ( lock.tryLock(50L, TimeUnit.MILLISECONDS) )
```

Performance e Escalabilidade

- Os principais objetivos do JSR-166, que resultou nos utilitários de concorrência do Java 5.0 foram
  - Facilitar a criação de aplicações paralelas corretas e seguras
  - Aumentar a escalabilidade das aplicações
- Escalabilidade mede a performance da aplicação quando o número de threads paralelos aumenta
  - Uma aplicação escalável é aquela cuja degradação em performance é logarítmica ou linear (e não exponencial)
- ReentrantLocks, ConcurrentHashMaps, variáveis atômicas, etc. são mais escaláveis que blocos synchronized, Hashtables, travas atômicas
- Algoritmos de justiça (fairness) são pouco escaláveis

Concurrent vs. Synchronized [IBM]

- Dados de tutorial da IBM (veja referências)
  - Em cada execução, N threads concorrentemente executam em um loop onde recuperavam chaves aleatórias ou de um Hashtable ou ConcurrentHashMap, com 60% das recuperações que falhavam realizando uma operação put() e 2% das recuperações de sucesso realizando uma operação remove()
  - Testes realizados em um sistema Xeon de dois processadores rodando Linux
  - Dados: 10,000,000 iterações normalizados a 1 thread para ConcurrentHashMap

Threads	ConcurrentHashMap	Hashtable
1	1.0	1.51
2	1.44	17.09
4	1.83	29.9
8	4.06	54.06
16	7.5	119.44
32	15.32	237.2

50% pior

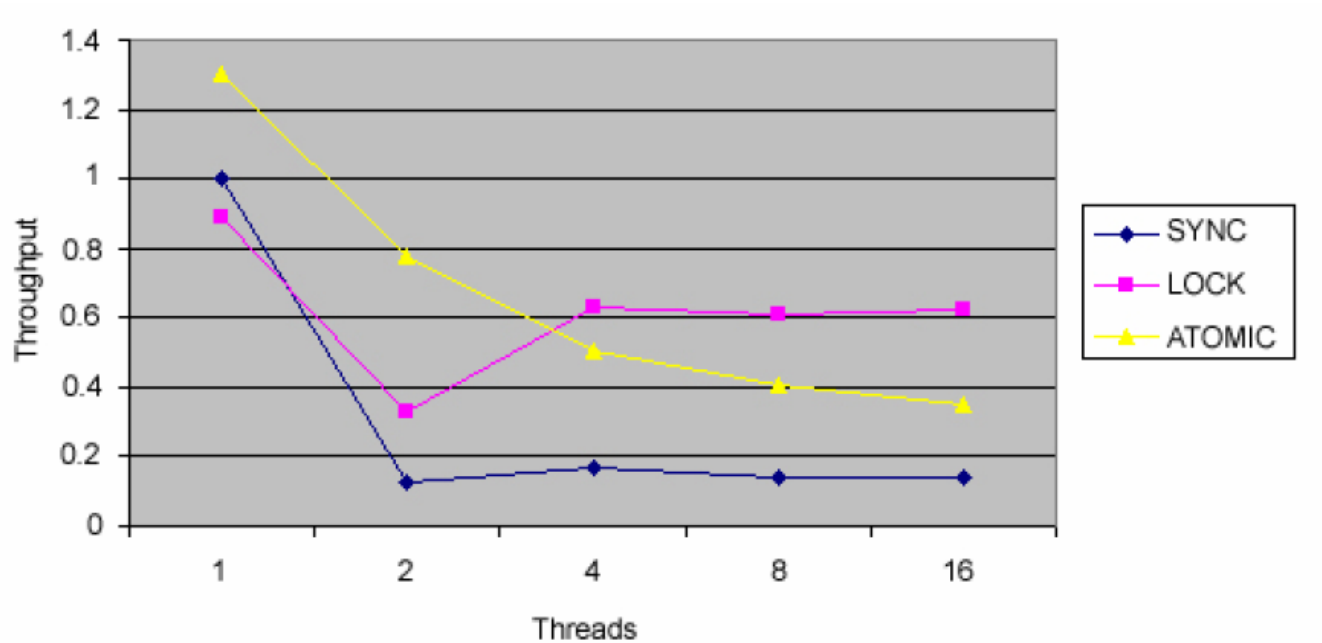
mais de 12x pior

16x pior

Fonte: IBM

Lock vs. sync vs. Atomic [IBM]

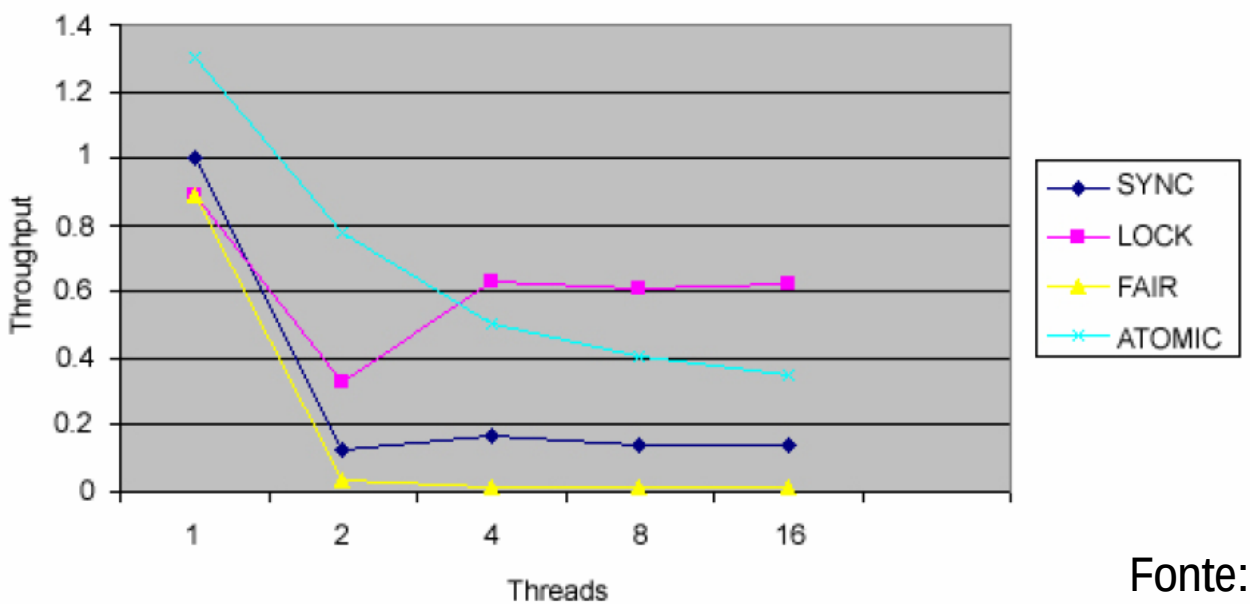
- Dados de tutorial da IBM (veja referências)
  - Simula lançamento de um dado usando gerador de números aleatórios de congruência linear.
  - Três implementações: usando synchronized, usando ReentrantLock, usando AtomicLong
  - Throughput relativo com múltiplos threads em UltraSparc3 8-way



Fair vs. Unfair

[IBM]

- Critério de justiça (fairness): garantia FIFO; é cara
  - synchronized não tem critério de fairness
  - ReentrantLock, Semaphore e ReentrantReadWriteLock permitem ser declaradas como fair (justas)
  - Travas justas são as menos escaláveis



Fonte: [IBM]

Conclusões

- Os utilitários de concorrência do Java 5.0 tornam a programação paralela mais simples, mais segura e mais escalável
- Antes de pensar em usar métodos da classe Thread, wait(), notify(), join(), synchronized, veja se não existe alguma estrutura no pacote java.util.concurrent que resolva o problema

Fontes de referência

[SDK] Documentação do J2SDK 5.0

[DL] Doug Lea, Concurrent Programming in Java, Second Edition, Addison-Wesley, 1996

[DL2] Doug Lea, JSR-166 Components, PDF slides. gee.cs.oswego.edu

gee.cs.oswego.edu/dl/concurrency-interest/jsr166-slides.pdf

[IBM] IBM DeveloperWorks. Java 5.0 Concurrency Tutorial

www.ibm.com/developerworks/edu/j-dw-java-concur-i.html