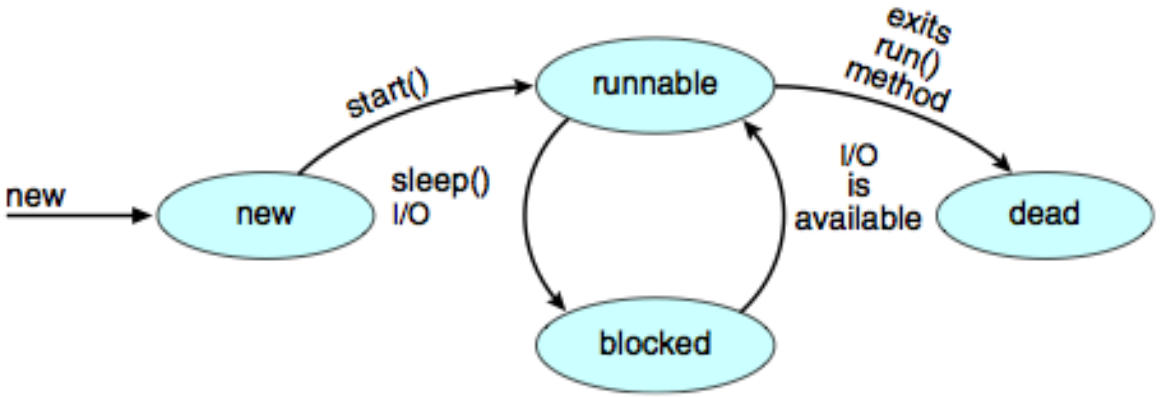
Java Threads

Marcio Seiji Oyamada
msoyamada@gmail.com

Threads em Java

- Gerenciada pela JVM
  - Mapeamento para o sistema operacional depende da implementação da JVM
  - Java Threads
    - Classe Thread ou
    - Interface Runnable



Interface Runnable

- Método necessário para descrever uma thread

```
public interface java.lang.Runnable {
    // Methods
   public abstract void run();
}
```

- Exemplo:

```
class ThreadInterface implements Runnable{
    public void run(){
       for (int i=0; i <20;i++){
      System.out.println("Thread["+Thread.currentThread().
  getName()+"]="+i);
        }
 }
}
```

Classe Thread

- Classe principal que representa uma thread em Java
- Métodos para gerenciar threads
  - Obter nome da thread
  - Alterar a prioridade
  - Interromper uma thread
  - Liberar o processador

Criando uma thread com a classe Thread

```
public class ThreadClasse extends Thread {
   public ThreadClasse(){
      super();

    }

   public void run(){
       for (int i=0; i <20;i++){
             System.out.println("Thread["+
             Thread.currentThread().getName()+
   "]="+i);
         }

    }
}
```

Executando threads (Interface Runnable)

- Utilizando a interface Runnable

```
public class ExecutaThread {
  public static void main(String args[]) throws
   InterruptedException{
        Thread t1;
        Thread t2;
        t1= new Thread(new ThreadInterface());
        t2= new Thread(new ThreadInterface());
        t1.start();  // inicia a execução da Thread
        t2.start();  // inicia a execução da Thread
        System.out.println("Thread inicializadas");
        t1.join();   // Aguarda a thread t1 finalizar
        t2.join();   // Aguarda a thread t1 finalizar
        System.out.println("Thread finalizadas");
  }

}
```

Executando threads (Classe Thread)

- Utilizando a Classe Thread

```
public class ExecutaThread {
   public static void main(String[] args) throws
        InterruptedException {
         // TODO code application logic here
         ClasseThread t1= new ClasseThread();
         ClasseThread t2= new ClasseThread();
         t1.start();
         t2.start();
         System.out.println("Thread inicializadas");
         t1.join();
         t2.join();
         System.out.println("Thread finalizadas");

     }
}
```

Executors

- Gerencia um conjunto de Threads
  - FixedThreadPool: número fixo de threads
  - CachedThreadPool: aloca dinamicamente
- FixedThreadPool
  - Evita o overhead com a criação de thread
  - Maior controle dos recursos utilizados durante a execução do sistema

Exemplo Executors (1)

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class Main {

public static void main(String[] args) {
        // TODO code application logic here
        ExecutorService executor=
  Executors.newCachedThreadPool();
        for (int i=0; i < 5; i++){
          executor.execute(new ThreadInterface());
        }
System.out.println("Threads executando");
        executor.shutdown();
    }

}
```

Exemplo: Executors (2)

```
public class ThreadInterface implements Runnable{

    public void run() {
        for (int i=0; i <20;i++){
      System.out.println("Thread["+Thread.currentThr
  ead().getName()+"]="+i);
        }
    }

}
```

Executors

- Métodos
  - *Execute(Interface Runnable)*: submete uma nova interface para ser executada
  - *Shutdown()*: previne que outras threads sejam submetidas ao *ExecutorService*
  - *ShutdownNow():* tentar finalizar a execução das Threads

Interrompendo Threads

```
public class MainInterruptTest {
   public static void main(String[] args) throws InterruptedException {
   ExecutorService executor= Executors.newCachedThreadPool();
        for (int i=0; i < 10; i++)
            executor.execute(new MyThread());
        System.out.println("Sleeping....");
        TimeUnit.SECONDS.sleep(10);
        executor.shutdownNow();
        System.out.println("Shutdown solicitado");
    }
}

class MyThread implements Runnable{

    public void run() {
        boolean sair=false;
        while (!sair){
            System.out.println(Thread.currentThread().getName());
            if (Thread.currentThread().isInterrupted()){
                System.out.println("Interrupção solicitada, finalizando a
   Thread"+ Thread.currentThread().getName());
                sair= true;
            }
        }
    }
}
```

Thread.yield()

- Para a execução da Thread atual e libera o processador para que uma outra Thread possa executar
  - Forçar a troca de contexto e conseqüentemente uma melhor distribuição do processador

```
public class ThreadInterface implements Runnable{

    public void run() {
        for (int i=0; i <10;i++){
       System.out.println("Thread["+Thread.currentThread().
   getName()+"]="+i);
            Thread.yield();
        }
    }
}
```

Threads:Sleep

```
public class SleepingTest implements Runnable {
   public void run() {
         try {
                  for (int i=0; i <10; i++) {
                           System.out.print("Sleep
   …"+Thread.currentThread().getName()) ;
                           // Old-style: // Thread.sleep(100);
                           // Java SE5/6-style:
                           TimeUnit.MILLISECONDS.sleep(100);
                  }
         } catch(InterruptedException e) {
                  System.err.println("Interrupted");
         }
   }


   public static void main(String[] args) {
         ExecutorService exec = Executors.newCachedThreadPool()
         for (int i = 0; i < 5; i++)
                  exec.execute(new SleepingTest());
         exec.shutdown();
   }
}
```

Inicialização de atributos

```
public class SleepingTest implements Runnable{
    boolean setYield= false;
    public SleepingTest (boolean _setYield){
       this.setYield= _setYield;
    }
    public void run() {
        for (int i=0; i <10;i++){
       System.out.println("Thread["+Thread.currentThread().getNam
   e()+"]="+i);
             if (setYield)
               Thread.yield();
        }
    }
   public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool()
        exec.execute(new SleepingTest(true));
         exec.execute(new SleepingTest(true));
         exec.execute(new SleepingTest(false));
         exec.execute(new SleepingTest(false));
        exec.shutdown();
   }
}
```

Threads: Retornando valores

- A interface Runnable não define um método para retornar valores
  - Solução 1: utilizar a interface *Callable*
  - Solução 2: retornar os valores em um objeto compartilhado
    - Problemas de sincronização em dados compartilhados (veremos mais adiante)

Interface Callable

- Callable: *generic* possibilitando definir o tipo de retorno
- Método call: ponto de entrada para a Thread (ao invés do método run()), retornando o tipo definido no *generic* Callable

Exemplo: Callable(1)

```
public class ThreadCallable implements
   Callable<Integer>{
      private static Random generator= new Random();
      public Integer call(){
          return generator.nextInt(1000);
      }
}
```

Exemplo: Callable(2)

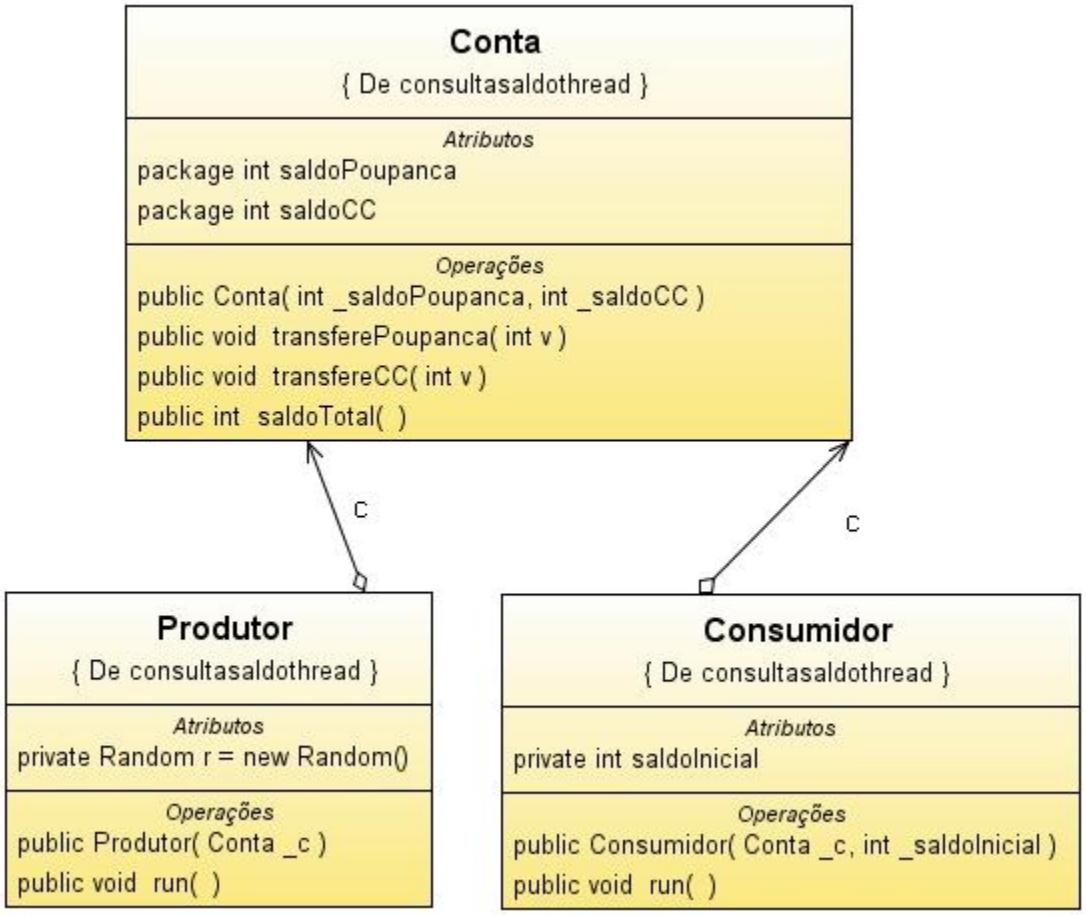
```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.ArrayList;
public class MainCallable {

public static void main(String[] args) {
        ExecutorService executor= Executors.newCachedThreadPool();
        ArrayList<Future<Integer>> resultados= new ArrayList<Future<Integer>>();
        for (int i=0; i < 10; i++)
            resultados.add(executor.submit(new ThreadCallable()));
         executor.shutdown();
        for (int i=0; i< resultados.size(); i++){
            try {
                Future<Integer> result;
                result = resultados.get(i);
                System.out.println(result.get());
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            } catch (ExecutionException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Sincronização entre threads

Acesso a variáveis compartilhadas

Acesso concorrente a dados



Classe Conta

```
public class Conta {
    int saldoPoupanca;
    int saldoCC;

    public Conta(int _saldoPoupanca, int _saldoCC){
        saldoPoupanca= _saldoPoupanca;
        saldoCC= _saldoCC;
    }
    public void transferePoupanca(int v){
        saldoCC -= v;
        saldoPoupanca +=v;
    }
    public void transfereCC(int v){
        saldoPoupanca -=v;
        saldoCC +=v;
    }
    public int saldoTotal(){
        return (saldoPoupanca + saldoCC);
    }
}
```

Produtor

```
public class Produtor implements Runnable{
    private Conta c;
    private Random r= new Random();
    public Produtor(Conta _c){
        c= _c;
    }
    public void run(){
        while (true){
            c.transfereCC(r.nextInt(1000));
            c.transferePoupanca(r.nextInt(500));
        }
    }
}
```

Consumidor

```
public class Consumidor implements Runnable{
    private Conta c;
    private int saldoInicial;
    public Consumidor(Conta _c, int _saldoInicial){
        c= _c;
        saldoInicial=_saldoInicial;
    }

   public void run(){
        int saldo;
        while (true){
            saldo= c.saldoTotal();
            if (saldo != saldoInicial){
                System.out.println("Saldo errado = "+saldo);
                Thread.currentThread().yield();
            }
        }
    }
}
```

Sincronizando acessos concorrentes

- *Synchronized*
  - Evita a execução concorrente das *threads*
- Como definir a sincronização
  - Métodos
    - public synchronized transfereCC(int v);
    - public synchronized transferePoupanca(int v);
    - public synchronized saldoTotal();
- Quando um método é definido como *synchronized*, ocorre um bloqueio, evitando a execução
  - *No método*
  - *Entre métodos synchronized da classe*

Exercício

1) Altere o código da transferência de conta para que o mesmo funcione corretamente
2) Verifique o funcionamento da aplicação SerialNumberGenerator
   a) Identifique o problema
   b) Faça as alterações necessárias

Utilizando tipos de dados sincronizados

- AtomicInteger, AtomicLong
  - boolean compareAndSet(expectedValue, updateValue);
  - addAndGet (int delta)
  - incrementAndGet()
- AtomicReference<V>
  - boolean compareAndSet(expectedValue, updateValue);
  - getAndSet(V newValue)

Tipos de dados sincronizados

- Os tipos Queue (fila) LinkedQueue (lista) no pacote java.util não são sincronizados
- Tipos de dados *thread-safe* são definidos no pacote java.util.concurrent
- Interface BlockingQueue()
  - Classe: ArrayBlockingQueue
    - void put(E e);
    - E take();

ArrayBlockingQueue

```
class Main() {
 // Instanciando
   private ArrayBlockingQueue<Integer> buffer= new
   ArrayBlockingQueue<Integer>(5); // buffer de 5 posições
}

 // inserindo elementos no buffer
    buffer.put(new Integer(5));


 //removendo elementos do buffer
   valor= buffer.take();
```

Semafóros em Java

```
import java.util.concurrent.Semaphore;
public class ThreadInterface implements Runnable{
    Semaphore sem;
    public ThreadInterface(Semaphore _s){
       sem=_s;
    }
    public void run() {
       sem.acquire();
         for (int i=0; i <10;i++){
       System.out.println("Thread["+Thread.currentThread().
  getName()+"]="+i);
              Thread.yield();
         }
       sem.release();
    }
    public static void main(String args[]) {
       Semaphore sem= new Semaphore(1);
       ExecutorService executor= Executors.newCachedThreadPool();
       for (int i=0; i < 5; i++)
           executor.execute(new ThreadInterface(sem));
    }
}
```

Exercício

- Implemente duas Threads, uma produtora e uma consumidora
  - O Produtor deverá gerar 1000 números e colocar no buffer compartilhado para ser consumido pela Thread Consumidor

Bloqueios e variáveis de condição

- Menor overhead que semáforos e synchronized
- Implementação mais eficiente no Java
- Variáveis de condição são utilizadas caso seja necessário bloquear a execução no meio da seção crítica

Exemplo: Lock

```
import java.util.concurrent.locks.*;
public class ThreadInterface implements Runnable{
   Lock mutex= new Lock();
     public void run() {
        mutex.lock();
          for (int i=0; i <10;i++){
        System.out.println("Thread["+Thread.currentThread().
   getName()+"]="+i);
                Thread.yield();
          }
        mutex.unlock();
     }
     public static void main(String args[]) {
        ExecutorService executor= Executors.newCachedThreadPool();
        for (int i=0; i < 5; i++)
            executor.execute(new MyThread());
     }
```

Exemplo: Variáveis de condição

```
class ProducerConsumer {
   Lock mutex= new Lock();
   Condition cond= mutex.newCondition();
     static class Produtor extends Runnable{
        public void run() {
          while (true) {
             mutex.lock();
              while (count == BUFFER_SIZE)
                     cond.await();
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
                cond.signal();
            }
        }
     }
     static class Consumidor extends Runnable{
        public void run() {
          while (true) {
             mutex.lock();
              while (count == 0)
                     cond.await();
                nextConsumed= buffer[out];
                out = (out + 1) % BUFFER_SIZE;
                count--;
             cond.signal();
            }
        }
     }
     ..
     ..// método Main
}
```