# Threads e Concorrência em Java (Material de Apoio)

Professor: Lau Cheuk Lung
http://www.inf.ufsc.br/~lau.lung

INE-CTC-UFSC



## Introdução

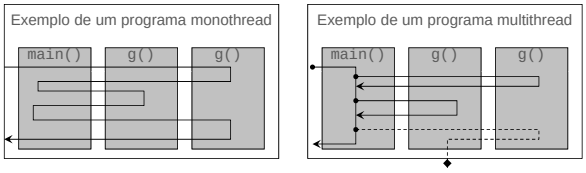
- A maioria dos programas são escritos de modo seqüencial com um ponto de início (método `main()`), uma seqüência de execuções e um ponto de término.
  - Em qualquer dado instante existe apenas uma instrução sendo executada.
- O que são threads ?

  É um simples fluxo seqüencial de execução que percorre um programa.
- Multithreading: o programador especifica que os aplicativos contêm *fluxos de execução* (threads), cada thread designando uma parte de um programa que pode ser executado simultaneamente com outras threads.



## Programação concorrente

- Todo programa, sendo *monothread* ou *multithread*, inicia com a execução da thread principal.




- Mecanismos de sincronização e prioridade podem ser usados para controlar a ordem de execução de threads independentes e colaboradas.



## Programação concorrente

- Por que usar threads ?
  - Permitir que um programa faça mais de uma coisa ao mesmo tempo. Cada thread pode fazer coisas diferentes ao mesmo tempo, por exemplo:
    - Uma thread baixando uma figura da rede e outra renderizando uma imagem;
    - Uma gerenciando a edição de um texto e outra cuidando da impressão de um outro documento em background;
    - Criação de várias threads para processamento de uma tarefa de forma paralela.
- Quando não usar threads ?
  - Quando um programa é puramente seqüencial.



## Threads em Java

- Threads Java são implementadas pela classe `Thread` do pacote `java.lang`:
  - Esta classe implementa um encapsulamento independente de sistema, isto é, a implementação real de threads é oferecida pelo sistema operacional.
  - A classe `Thread` oferece uma interface unificada para todos os sistemas.
    - Portanto, uma mesma implementação do Java Thread pode fazer com que a aplicação proceda de forma diferentes em cada sistema.
- Ver a API Thread Java em `java.lang.Thread`
  - http://java.sun.com



## Threads em Java

- Criando uma nova Thread
  - Para que uma thread possa executar um método de uma classe, a classe deve:
    - Herdar (extend) a classe `Thread` (o qual implementa a classe `Runnable` em si), ou;
    - Implementar a interface `Runnable`.

## Herdando a class Thread

```
class MinhaThread1 extends Thread {
  public void run() {
     System.out.println("Bom Dia !");
  }
}
```

- O método `run()` contém o código que a thread executa.

```
class Teste1 {
  public static void main(String Args[]) {
     new MinhaThread1().start();
  }
}
```

- Para executar a thread é necessário instanciar a classe `MinhaThread` e invocar o método `start()`.



## Implementando a classe Runnable

```
class MinhaThread2 implements Runnable {
  public void run() {
     System.out.println("Bom Dia !");
  }
}
```

- Desta forma, a classe `MinhaThread` pode herdar uma outra classe.

```
class Teste2 {
  public static void main(String Args[]) {
     new Thread(new MinhaThread2()).start();
  }
}
```



## Execução paralela de threads

```
class ImprimirThread_1 implements Runnable {
  String str;
  public ImprimirThread_1(String str) {
      this.str = str;
  }
  public void run() {
      for(;;)
        System.out.print(str);
  }
}
class TesteConcorrente {
  public static void main(String Args[]) {
      new Thread(new ImprimirThread_1("A")).start();
      new Thread(new ImprimirThread_1("B")).start();
  }
}
```



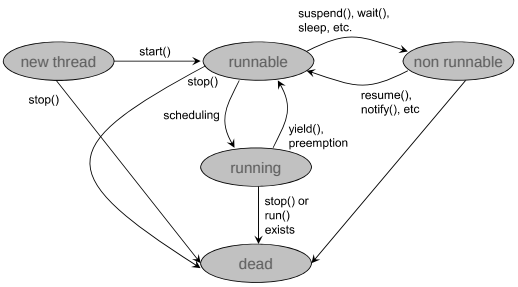
## Execução paralela de threads

- Método `yield()`, cede o processamento para outra thread.

```
class ImprimirThread_2 implements Runnable {
  String str;
  public ImprimirThread_2(String str) {
      this.str = str;
  }
  public void run() {
     for(;;) {
        System.out.print(str);
        Thread.currentThread().yield();
     }
  }
}
class TesteConcorrente2 {
  public static void main(String Args[]) {
      new Thread(new ImprimirThread_2("A")).start();
      new Thread(new ImprimirThread_2("B")).start();
  }
}
```

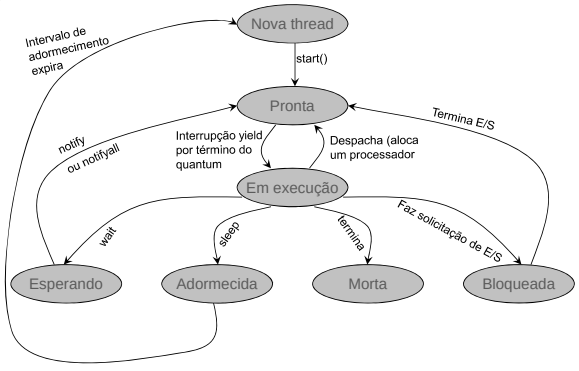


## Thread: diagrama de estado






## Thread: diagrama de estado



## Prioridade de thread

- Cada thread possui uma prioridade de execução que vai de `Thread. MIN_PRIORITY` (igual a 1) a `Thread.MAX_ PRIORITY` (igual a 10).
  - Importante: uma thread herda a prioridade da thread que a criou.
- O algoritmo de escalonamento sempre deixa a thread (runnable) de maior prioridade executar.
  - A thread de maior prioridade preempta as outras threads de menor prioridade.
  - Se todas as threads tiverem a mesma prioridade, a CPU é alocada para todos, um de cada vez, em modo round-robin.
    - getPriority(): obtém a prioridade corrente da thread;
    - setPriority(): define uma nova prioridade.



## Prioridade de thread

```
class BaixaPrioridade extends Thread {
   public void run() {
      setPriority(Thread.MIN_PRIORITY);
       for(;;) {
           System.out.println("Thread de baixa prioridade
                               executando -> 1");
       }
   }
}

class AltaPrioridade extends Thread {
   public void run() {
      setPriority(Thread.MAX_PRIORITY);
       for(;;) {
          for(int i=0; i<5; i++)
              System.out.println("Thread de alta prioridade
                                  executando -> 10");
          try {
             sleep(100);
          } catch(InterruptedException e) {
             System.exit(0);
          }
       }
   }
}
```

InterruptedException lança exceção se a thread é interrompida pelo método interrupt();



## Prioridade de thread

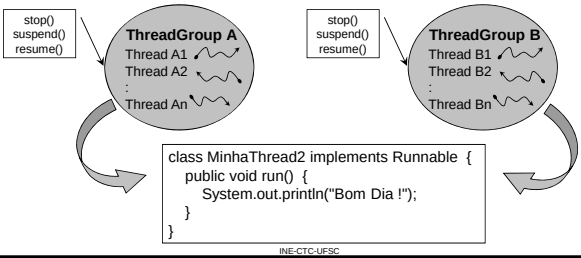
```
class Lançador {
   public static void main(String args[]) {
        AltaPrioridade  a = new AltaPrioridade();
        BaixaPrioridade b = new BaixaPrioridade();
        System.out.println("Iniciando threads...");
        b.start();
         a.start();
        // deixa as outras threads iniciar a execução.
        Thread.currentThread().yield();
        System.out.println("Main feito");
   }
}
```

```
Iniciando threads...
Main feito
Thread de alta prioridade executando -> 10
Thread de alta prioridade executando -> 10
Thread de baixa prioridade executando -> 1
Thread de baixa prioridade executando -> 1
Thread de baixa prioridade executando -> 1
Thread de alta prioridade executando -> 10
Thread de alta prioridade executando -> 10
Thread de baixa prioridade executando -> 1
:
```



## Grupo de Threads

- Cada thread pertence a um grupo de threads. Um **thread group** é um conjunto de threads com mecanismos para desempenhar operações em todos membros do conjunto.
  - Se não definido, toda thread criada pertence ao grupo 'main'.






## Grupo de Threads

- Criando um grupo de threads:
  - `ThreadGroup(String name)`
  - `ThreadGroup(ThreadGroup parent, String name)`
- Criando uma thread dentro de um **thread group** específico:

```
ThreadGroup meuGrupo = new ThreadGroup("Grupo A");
Thread novaThread = new Thread(meuGrupo, umObjRunnable, "Thread 11");
```

  - `umObjRunnable`: objeto alvo cujo método `run()` é executado.
- Para saber qual grupo uma thread pertence use o método `getThreadGroup()` e depois `getName()`.



## Sincronização de Threads

- Quando muitas threads são executadas muitas vezes é necessário sincronizar suas atividades, por exemplo:
  - Prevenir o acesso concorrente a estruturas de dados no programa que são compartilhados entre as threads;

```
class MeuDado {
   private int Dado;
   public void armazenar(int Dado) {
        this.Dado=Dado;
   }
   public int carregar() {
        return this.Dado;
   }
}
class Main {
   public static void main(String argv[]) {
      MeuDado dado = new MeuDado();
      new Thread(new Produtor(dado)).start();
      new Thread(new Consumidor(dado)).start();
    }
}
```

## Sincronização de Threads: Problema

```
class Produtor implements Runnable {
   MeuDado dado;
   public Produtor(MeuDado dado) {
      this.dado=dado;
   }
   public void run() {
      int i;
      for(i=0;i<30;i++) {
         dado.armazenar(i);
         System.out.println ("Produtor: "+i);
         try {
            // cochila por um tempo randômico (0 to 0.5 sec)
            Thread.sleep((int)(Math.random()*500));
         } catch (InterruptedException e) { }
      }
   }
}
```



## Sincronização de Threads: Problema

```
class Consumidor implements Runnable {
   MeuDado dado;
   public Consumidor(MeuDado dado) {
      this.dado=dado;
   }
   public void run() {
      for(int i=0;i<30;i++) {
         System.out.println("Consumidor: "+dado.carregar());
         try {
            // cochila por um tempo randômico (0 to 0.5 sec)
            Thread.sleep((int)(Math.random()*500));
         } catch (InterruptedException e) { }
      }
   }
}
```



## Sincronização de Threads: Solução 1

```
class MeuDado {
   private int Dado;
   private boolean Pronto;
   private boolean Ocupado;
   public MeuDado() {
      Pronto = false;
      Ocupado = true;
   }
   public void armazenar(int Dado) {
      while(!Ocupado);
      this.Dado=Dado;
      Ocupado = false;
      Pronto = true;
   }
   public int carregar() {
      int Dado;
      while(!Pronto);
      Dado = this.Dado;       // armazena o valor porque uma vez
      Pronto = false;         // ocupado é "true" ele pode mudar
      Ocupado = true;         // em qualquer tempo.
      return Dado;
   }
}
```



## Sincronização usando Monitores

- Java suporta sincronização de threads através do uso de monitores.
- Um monitor é um inibidor, como um *token* que uma thread pode adquirir e liberar:
  - Um monitor é associado com um objeto específico e funciona como uma tranca nesse objeto;
  - Uma thread pode adquirir o monitor (ou *token*) de um objeto se nenhuma outra thread tiver ele, e pode liberá-lo quando quiser.
- Região crítica: é um segmento de código, dentro de um programa, que não deve ser acessado concorrentemente.
  - Em Java, esse segmento de código é declarado usando a palavra chave `synchronized`.



## Sincronização usando Monitores

- Declarando uma seção de código sincronizada:

```
void minhaClasse() {
   // código não sincronizado
   ...
   synchronized(objCujoMonitorÉParaSerUsado) {
      // bloco sincronizado
      ...
   }
   // mais código não sincronizado
   ...
}
```

- Declarando um método todo sincronizado:

```
synchronized meuMetodo(...) {
   // bloco sincronizado
   ...
}
```



## Sincronização usando Monitores

```
class MeuDado {
   private int Dado;
   private boolean Pronto;
   private boolean Ocupado;

   public MeuDado() {
      Pronto=false;
      Ocupado=true;
   }

   public synchronized void armazenar(int Dado) {
      while(!Ocupado);
      this.Dado=Dado;
      Ocupado=false;
      Pronto=true;
   }

   public synchronized int carregar() {
      while(!Pronto);
      Pronto=false;
      Ocupado=true;
      return this.Dado;
   }
}
```

## Sincronização usando Monitores

- Código corrigido

```
class MeuDado {
   private int Dado;
   private boolean Pronto;
   private boolean Ocupado;

   public MeuDado() {
      Pronto=false;
      Ocupado=true;
   }
   public void armazenar(int Dado) {
      while(!Ocupado);
      synchronized(this) {
         this.Dado=Dado;
         Ocupado=false;
         Pronto=true;
      }
   }
   public int carregar() {
      while(!Pronto);
      synchronized(this) {
         Pronto=false;
         Ocupado=true;
         return this.Dado;
      }
   }
}
```

Comentário:
- Tem que fornecer um objeto para o synchronized;
- Não é adequado usar loop busy-wait (ver while), pois usa o processador.



## Sincronização usando eventos

- Faz uso dos métodos `wait()` e `notify()` para gerar eventos de espera e notificação para parar de esperar.

```
class MeuDado {
   private int Dado;
   private boolean Pronto;

   public MeuDado() {
      Pronto=false;
   }

   public synchronized void
      armazenar(int Data) {
      while (Pronto)
         try {
            wait();
         } catch (InterruptedException e) { }
      this.Dado=Data;
      Pronto=true;
      notify();
   }
 public synchronized int carregar() {
   while (!Pronto)
         try {
            wait();
         } catch
 (InterruptedException e) { }
       Pronto=false;
       notify();
       return this.Dado;
   }
}
```



## Barreira de sincronização

- Permite sincronizar varias threads em um ponto específico do código.
  - Usada em aplicações onde todas as threads devem terminar uma fase antes de mover *todas juntas* para a próxima fase.
- Uma thread que alcança a barreira automaticamente `wait()`;
- Quando a última thread alcança a barreira ele `notifyAll()` todas threads em espera.



## Barreira de sincronização

```
import java.util.*;
class Barreira {
   private int ThreadsParticipante;
   private int EsperandoNaBarreira;

   public Barreira(int num) {
      ThreadsParticipante = num;
      EsperandoNaBarreira = 0;
   }

   public synchronized void Alcançado() {
      EsperandoNaBarreira++;
      if(ThreadsParticipante != EsperandoNaBarreira) {
         try {
            wait();
         catch (InterruptedException e) {}
      } else {
         notifyAll();
         EsperandoNaBarreira=0;
      }
   }
}
```



## Esperando uma thread terminar

- Algumas vezes é necessário esperar a terminação de uma thread, por exemplo, a thread principal (main) pode lançar uma thread secundária para processar uma tarefa