Multithreading – Java

LPOO
Prof. Fabrício Martins Lopes
fabricio@utfpr.edu.br

Objetivos da aula

- O que são threads e sua utilidade
- Gerenciamento de atividades concorrentes
- Ciclo de vida de uma thread
- Prioridades e agendamentos
- Sincronização
- Exemplos e aplicações

Definição

- A classe java.lang.Thread é definida como:

  public class Thread extends Object implements Runnable

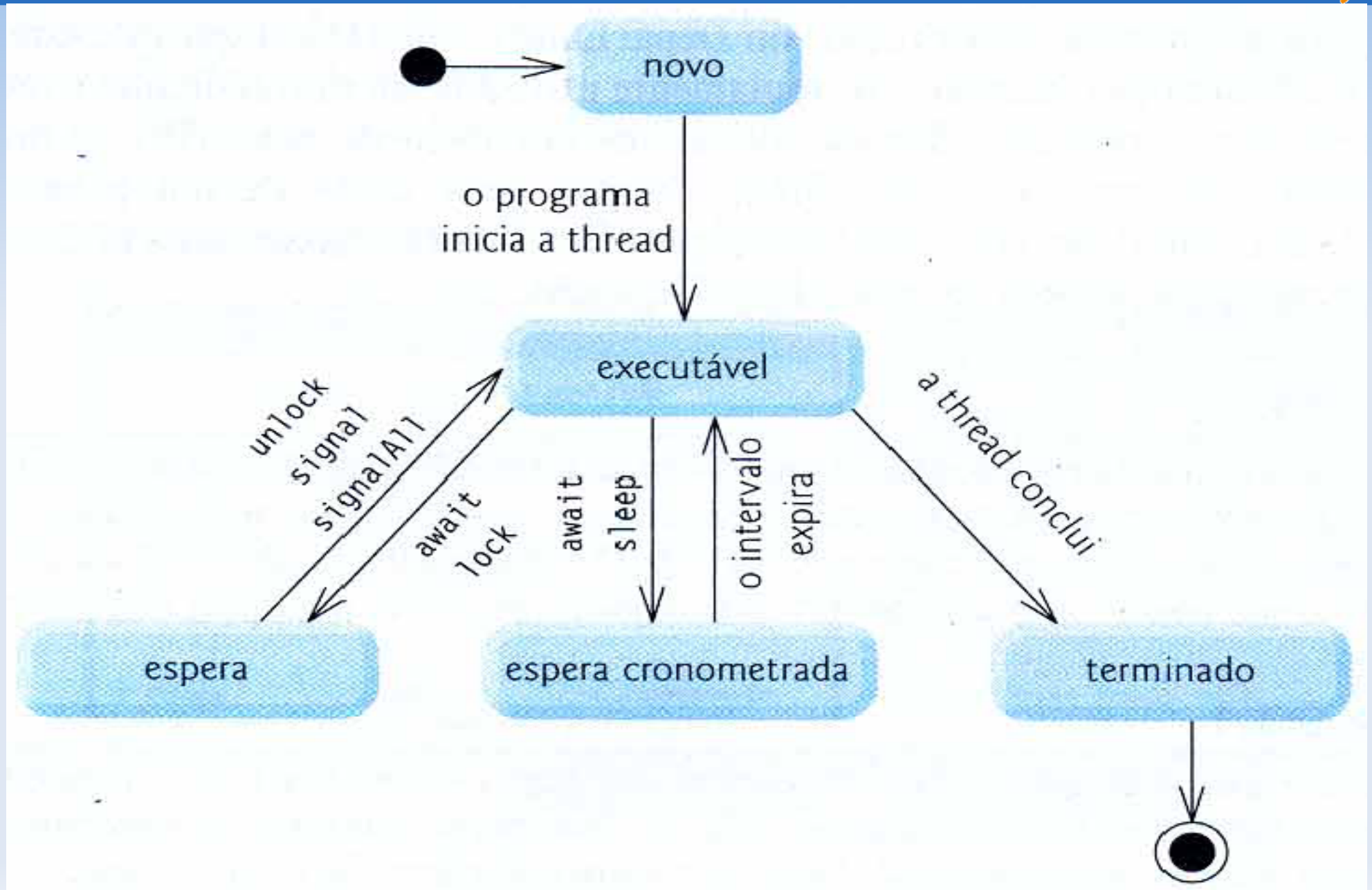
- A interface **Runnable** deve ser implementada por qualquer classe cujas instâncias são destinadas a serem executadas por uma thread.
- A **classe deve implementar** um método sem argumentos chamado **run**.
- A especificação completa da classe Thread está disponível em:

  http://download.oracle.com/javase/6/docs/api/java/lang/Thread.html

Threads e sua utilidade

- Executar operações de forma **paralela / concorrente.**
- A maioria das LPs **não permitem implementações paralelas,** como o C e C++.
- O Java disponibiliza a programação concorrente por meio da **Thread.**
- Esse recurso do Java, chamado **multithreading**, permite um programa executar concorrentemente com outras threads.

Classe Thread e seus estados



Prioridades

- Os algoritmos **preemptivos** são algoritmos que permitem que um processo seja interrompido durante sua execução.

- Já os algoritmos **não preemptivos**, por serem utilizados exclusivamente em sistemas monoprocessados, esse fato não ocorre, sendo cada programa executado até o fim.

Prioridades

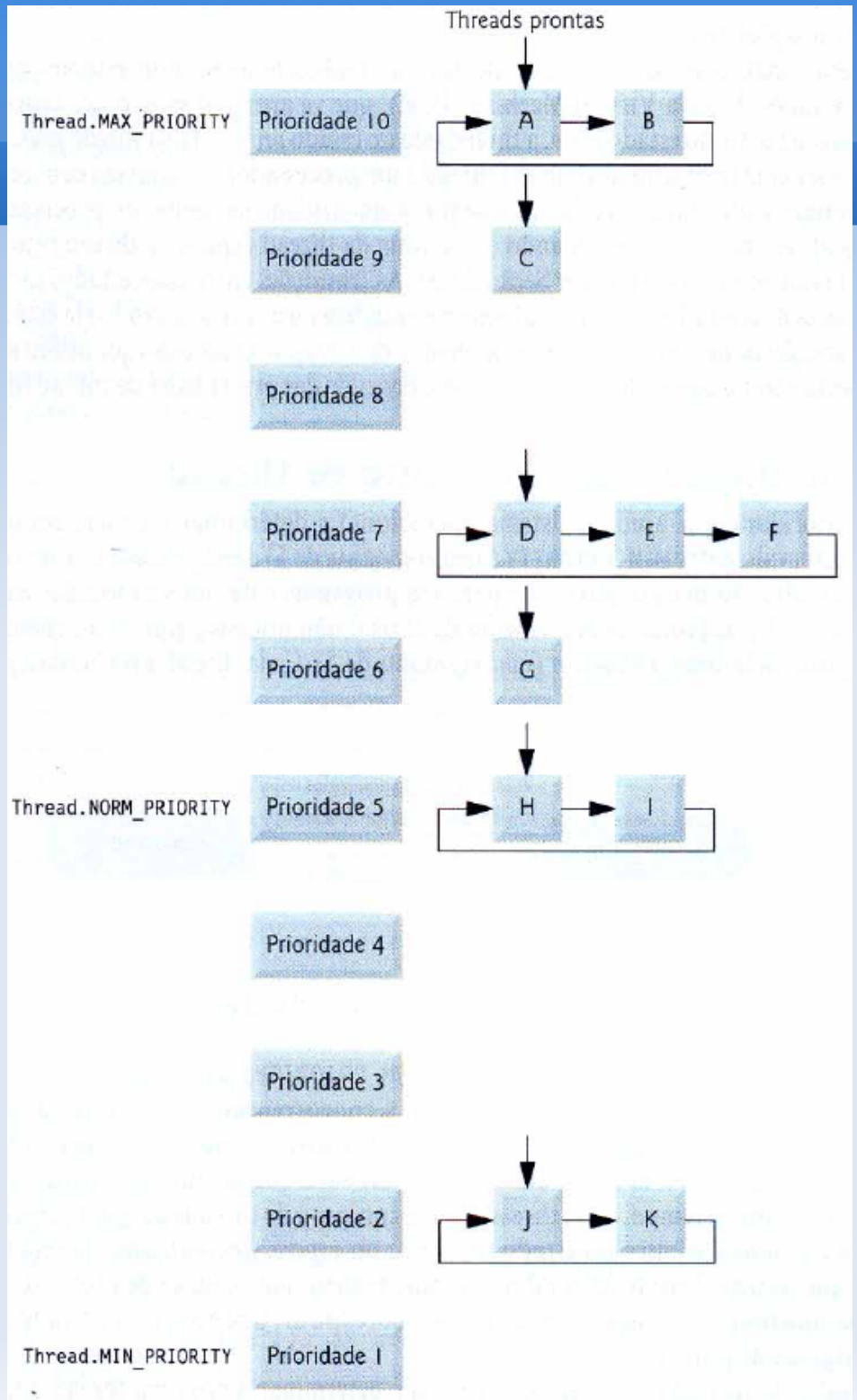
- Cada thread Java te sua própria prioridade.
- A prioridade ajuda o SO a determinar a ordem de execução das threads.
- **Prioridades usando atributos da classe Thread:**
  - **MIN_PRIORITY** (constante == 1)
  - **MAX_PRIORITY** (constante == 10)
  - **NORM_PRIORITY** (**default**, constante == 5)

Prioridades

- Threads com valor de prioridade mais alta recebem maior tempo de execução do processador.
- Cada nova thread **herda a prioridade** da thread que a criou.
- O método **setPriority(int i)** pode ser usado para atribuir um valor de prioridade para a thread.

Prioridades

- **Agendamento de prioridades**
- Threds com valores de prioridades + altas por meio do **agendamento preemptivo**, podem **adiar indefinidamente** as threads com menor prioridade (**inanição**).



Criando e Executando threads

- O modo **preferido** de criar um aplicativo com **múltiplas threads** é **implementar a interface Runnable** (java.lang)**.**
- **Runnables** são executadas por um objeto de uma classe que implementa a interface **Executor** (java.util.concurrent).
- O Executor declara um único método chamado **execute**.
- Um objeto Executor cria e gerencia um grupo de threads denomidado **pool de threads.**

Exemplos

- class PrintTask implements Runnable
- class RunnableTester
- As duas classes estão disponíveis no moodle.

Sincronização de threads

- Frequentemente, múltiplas threads de execução manipulam um objeto compartilhado na memória.
- É importante **definir quando e como um objeto compartilhado será acessado** pelas múltiplas threads.
- O Java utiliza **bloqueios** para realizar a sincronização.

Sincronização de threads

- Uma thread chama o método **lock para obter o bloqueio**.
- Uma vez que o lock foi obtido, outra thread não poderá obter o bloqueio novamente até que a thread o libere, chamando o método **unlock**.
- Somente uma thread pode obter o bloqueio por vez.

Sincronização de threads

- Classe **ReentrantLock** implementa a interface **Lock** (java.util.concurrent.locks).
- O construtor de **ReentrantLock** aceita um parâmetro booleano que especifica se o bloqueio tem uma **diretiva de imparcialidade**.
- A **diretiva de imparcialidade** (parâmetro true) determina que a thread na espera mais longa vai obter o bloqueio quado estiver disponível.

Sincronização de threads

- Pode ser definida uma **variável de condição** para uma thread para determinar o bloqueio de uma thread.
- As **variáveis de condição** **devem ser associadas com um Lock e, são criadas a partir do método newCondition da interface Lock** que retorna um objeto **Condition** (java.util.concurrent.locks.Condition).

Sincronização de threads

- Para **esperar** uma **variável de condição** a **thread pode chamar o método await** de Condition.
- **A chamada do método await coloca a thread no estado de espera** dessa Condition.
- Quando a thread em execução completar a tarefa dependente, pode determinar que a **thread na espera pode continuar a execução**, é chamado o método **signal**.

Sincronização de threads

- Se múltiplas **threads estiverem na espera de uma Condition** quando **signal** for chamado, a thread de espera mais longa irá se tornar executável.
- Se uma thread chamar o método Condition **signalAll**, **todas as threads que esperam essa Condition** mudam para o **estado executável**.
- Quando uma thread **concluir sua tarefa** com um objeto compartilhado, ela **deve chamar o método unlock** para liberar o objeto Lock.

Sincronização de threads

- O **Impasse (deadlock)** ocorre quando uma thread em espera não pode prosseguir porque está esperando outra thread e, simultaneamente a segunda thread nào pode prosseguir porque está esperando a primeira.
- **É um erro** se uma thread tentar chamar um **await, signal ou signalAll** em uma variável de condição **sem adquirir o bloqueio** dessa variável de condição. Isso causa uma **IllegalMonitorStateException**.

Exemplos

- interface Buffer.java
- classe Producer.java
- classe Consumer.java
- classe UnsynchronizedBuffer.java
- classe SharedBufferTest.java

- **Versão sincronizada:**
- classe SynchronizedBuffer.java
- classe SharedBufferTest2.java

Referências Consultadas

- DEITEL, P.J. Java - Como Programar. Porto Alegre: Bookman, 2003.
- HORSTMANN, Cay. Big Java. Porto Alegre: Bookman, 2004.
- HORSTMANN, Cay, S. e CORNELL, Gary. Core Java 2. São Paulo: Makron Books, 2001 v.1. e v.2.
- MORGAN, Michael. Java 2 para Programadores Profissionais. Rio de Janeiro: Ciência Moderna, 2000.