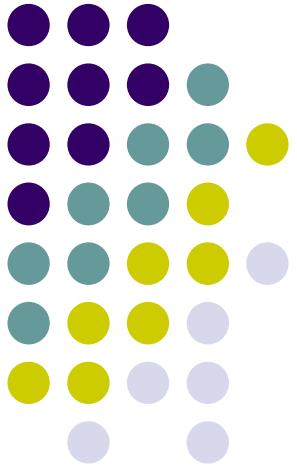
Threads

Ling. Programação Orient. Obj

.:: 2010/1 ::.

Prof. Leandro C. Fernandes



O que é uma Thread?

- Uma *thread* é uma execução seqüencial de um programa.
- Cada programa tem, pelo menos, uma *thread*.
- Cada *thread* tem a sua própria pilha, prioridade e conjunto de registos virtuais.
- Os *threads* subdividem o comportamento de um programa em subtarefas independentes.

Onde é que se usam Threads?

- São usados virtualmente em todos os computadores:
  - Em muitas aplicações (imprimir)
  - Em programas como browsers Internet
  - Em bases de dados
  - Em sistemas operacionais
- As *Threads* são normalmente usados sem serem percebidos pelo utilizador.

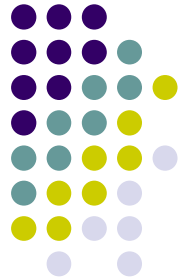
Exemplo de utilização de Thread

- Cada programa corre numa thread.

  Colocar uma thread para dormir é uma técnica que permite que outros threads possam ser executadas.

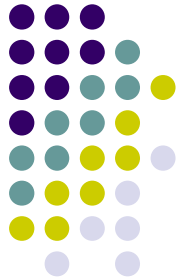
```
public static void main (String args[]) {
   System.out.println("Eu sou a thread " +
      Thread.currentThread().getName());
   try { Thread.sleep(5000) }
   catch (InterruptedException e){}
   …
}
```

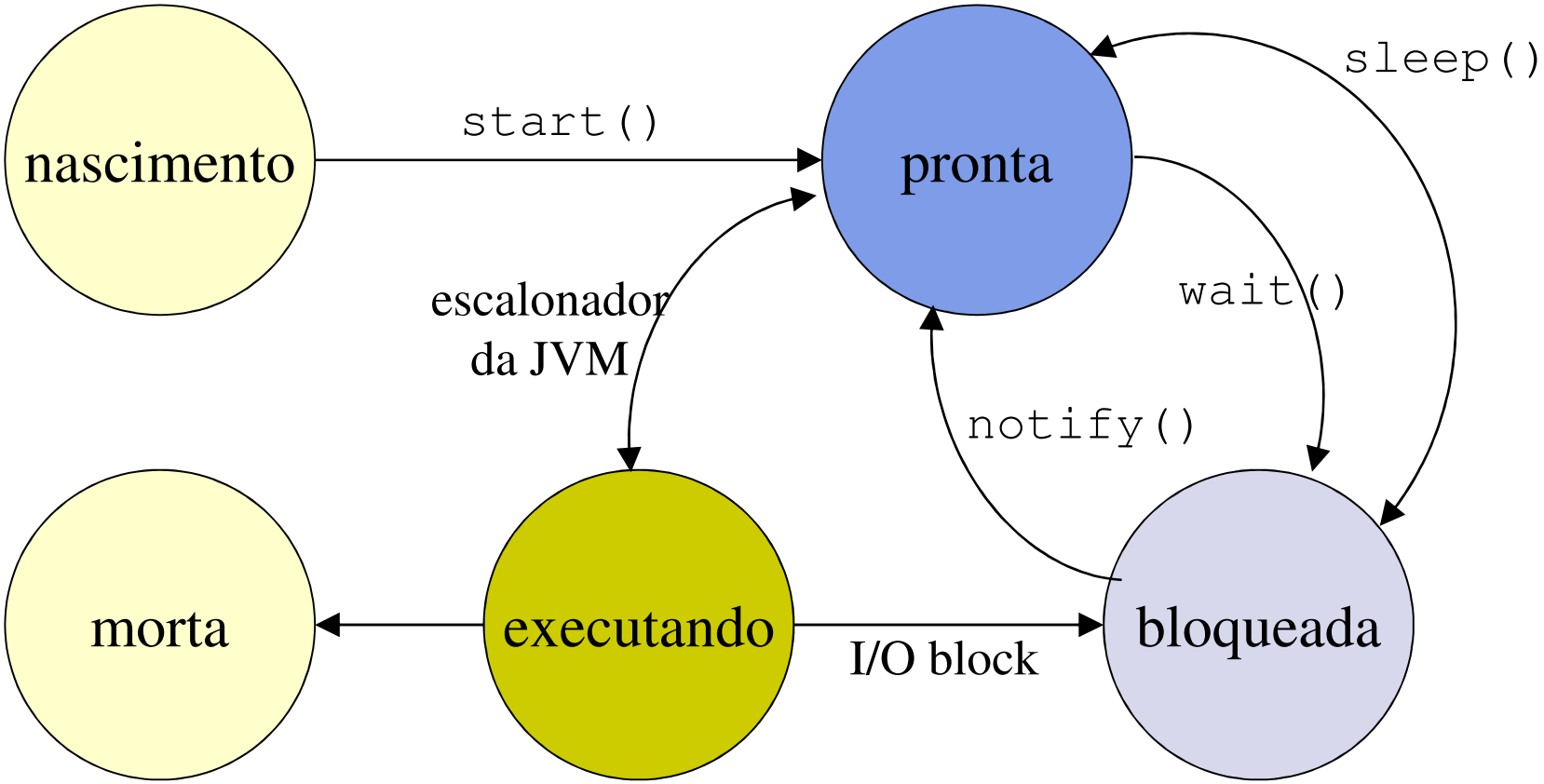
Porque se devem usar Threads?



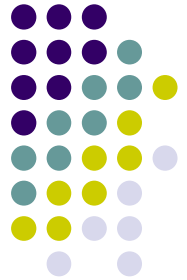
- Para melhor aproveitar as capacidades do computador (utilizar o CPU enquanto se faz entrada/saída de dados)
- Maior produtividade para o utilizador final (uma interface mais interativa)
- Vantagens para o programador (simplificar a lógica aplicacional)

Ciclo de Vida de uma Thread



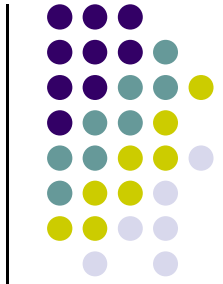


Criando *threads* em Java



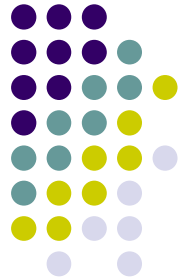
- Existem duas maneiras possíveis para criarmos *threads* em Java:
  - Estendendo a classe Thread.
  - Implementando a interface Runnable.

A Classe Thread



- Mantém o estado de uma thread
- Fornece diversos construtores
- Fornece diversos métodos
  - Thread.currentThread()
  - Thread.sleep()
  - Thread.setName()
  - Thread.isAlive()
- Escalonados pela JVM
- Utiliza o sistema operacional ou um pacote de threads

Métodos da classe Thread



`run()`

Deve estar presente em todas as *threads*.

`start()`

Registra a *thread* no *thread scheduler*.

`getName()/setName()`

Atribui ou retorna o nome de uma *thread*. Por default as *threads* são nomeadas numericamente.

`yield()`

Faz com que a *thread* corrente pause, possibilitando que outra *thread* seja despachada.

Métodos da classe Thread

`sleep()`

Faz com que a *thread* fique em estado de espera uma quantidade mínima de tempo, em ms, possibilitando a CPU executar outras *threads*.

`interrupt()`

Atribui à *thread* o estado de interrompível.

`isInterrupted()`

Verifica se foi atribuído à *thread* o estado de interrompível.

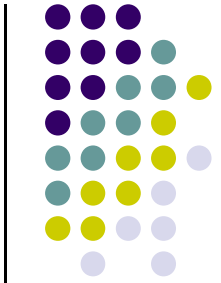
`getPriority()/setPriority()`

Atribui ou retorna a prioridade de despacho de uma *thread*.

`join()`

Condiciona a continuação da execução de uma *thread* ao término de uma outra.

Métodos da classe Object



`wait()`

Utilizado para sincronizar acesso a um objeto. Coloca a *thread* corrente em estado de espera até que outra *thread* chame os métodos notify ou notifyAll liberando o objeto.

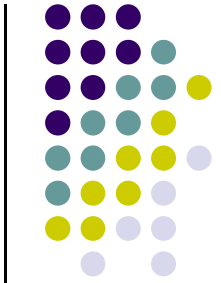
`notify()`

Acorda a *thread* que, voluntariamente, estava esperando a liberação de um objeto.

`notifyAll()`

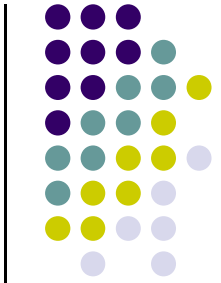
Acorda todas as *threads* que estavam esperando a liberação de um objeto.

Criação de um novo Thread



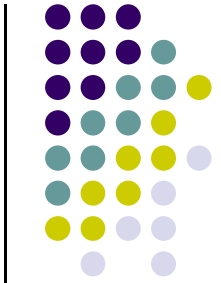
1. Criar a nova classe.
    1. Definir uma subclasse de Thread.
    2. Redefinir o seu método `run()`.
2. Instanciar e executar o thread.
    1. Criar uma instância da classe.
    2. Invocar o método `start()`.
3. O escalonador invoca o método `run()`.

Criar a Classe



```
public class Dorminhoca extends Thread {
  public void run () {
    Date horaInicio = new Date();
    try {
         Thread.currentThread().sleep(
                          (int)(1000*Math.random()) );
    }
    catch (Exception es) {}
    long tempoDecorrido = new Date().getTime() -
                                  horaInicio.getTime();
   System.out.println( Thread.currentThread().getName()
    +": Eu dormi por "+tempoDecorrido+"milliseconds");
  }
}
```

Instanciar e Executar



```
public static void main(String[] args) {
   new Dorminhoca().start();
   new Dorminhoca().start();
   System.out.println("Duas threads iniciadas…");
}
```

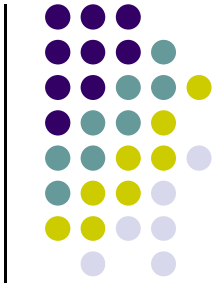
- **Saída:**

```
Duas threads iniciadas…
Thread-1: Eu dormi por 78 milliseconds
Thread-2: Eu dormi por 428 milliseconds
```

Regiões críticas

- Quando duas *threads* precisam utilizar ao mesmo tempo um objeto existe a possibilidade de haver corrupção de dados.
- As seções de um programa que têm o potencial de provocar este dano são chamadas de "*regiões críticas*". Para evitar esta situação, o acesso por estas "*regiões críticas*" deve ser sincronizado.
- As *threads* devem estabelecer um acordo de forma que antes que qualquer instrução de uma região crítica seja executada um *lock* do objeto deve ser adquirido.

Acesso a Recursos Compartilhados

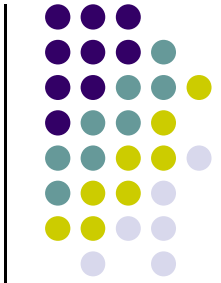


- Os dados podem ficar corrompidos se acessados por vários threads:

```
public class ContaBancaria {
  private double saldo;
  public void saque(double valor) {
    saldo -= valor;
  }
```

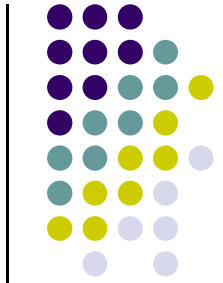
- Utilizar a palavra *synchronized* para evitar conflitos de recursos partilhados.

Sincronizando *threads* em Java



- Estando o objeto *locked* qualquer outra *thread* fica impossibilitada de acessá-lo até que o objeto fique liberado(*unlocked*).
- Cada objeto tem seu próprio *lock*.
- O *lock* pode ser adquirido ou liberado através do uso de métodos ou instruções *synchonized*.
- O objeto fica atomicamente bloqueado quando o seu método *synchronized* é invocado.

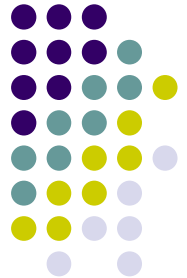
Sincronizando *threads* em Java



- A sincronização força com que as execuções de duas ou mais *threads* sejam mútuamente exclusivas no mesmo espaço de tempo.

- O *lock* é automaticamente liberado assim que o método *synchronized* termina.

Exemplo de utilização do synchronized



- Se uma thread invoca um método *synchronized*, nenhuma outra thread pode executar qualquer método sincronizado (no mesmo objeto) até o primeiro thread completar a sua tarefa.

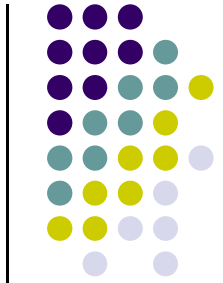
```
public class ContaBancaria {
  private double saldo;
  public synchronized void saque(double valor) {
    saldo -= valor;
  }
  public synchronized void deposito(double valor) {
    saldo += valor;
  } …
```

Synchonized Statements

- Permite que a sincronização seja feita apenas em uma porção do código.
- Como a sincronização afeta a performance, este processo é mais eficaz.
- Com *synchronized statements* somente fica *locked* o absolutamente necessário.
- Sintaxe:

```
synchonized (objeto que será locked) {
  statements
}
```

Exemplo de utilização do synchronized



```
public static void abs(int[] valores) {
 synchronized (valores) {
   for (int i=0; i < valores.length; i++)
   {
     if (valores[i] < 0)
        valores[i] = - valores[i];
   }
 }
}
```

Cuidado com synchronized!

- Cuidado para evitar *deadlocks*, evitando que todos os métodos sejam synchronized.

```
void BotaoEmCasoDePanico(ActionEvent e) {
  …
  finished = true;
  while(elapsedTime == 0) {}
    jText.setText("…");
  }
```

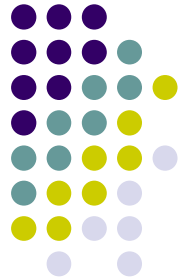
Comunicação entre *Threads*

- O mecanismo de sincronização é suficiente para evitar que as *threads* interfiram umas com as outras, mas pode ocorrer a necessidade de as *threads* se comunicarem.
- Os métodos `wait()`, `notify()` e `notifyAll()` têm o propósito de permitir a comunicação entre as *threads.*
- O método `wait()` faz com que a *thread* fique em estado de espera até que determinada condição aconteça.
- Os métodos `notify()` e `notifyAll()` informam às *thread*s, em estado de espera, que alguma coisa ocorreu e que pode satisfazer àquela condição.

Formas de uso: `wait()`

```
synchronized void facaQuandoCondicao() {
    while (!condicao) {
      try{
         wait();
      }
      catch(InterruptedException e) {}
    }
    /* instruções quando a
     * condição for verdadeira
     */
}
```

Formas de uso:
`notify()` e `notifyAll()`



```
synchronized void trocaCondicao() {
   /* troca os valores usados para
    * o teste da condição
    */
   notifyAll(); // ou notify()
}
```

Comunicação entre *Threads*

- O método que testa a condição precisa ser *synchronized* pois ninguém pode garantir que após o *while* a condição já não tenha sido alterada por outra *thread*.
- O wait() suspende a execução da *thread* e libera o *lock* do objeto. Quando a *thread* é reiniciada o *lock* é readquirido.
- O teste da condição deve estar sempre em *loop.* Nunca podemos assumir que se fomos notificados implica em condição satisfeita. Não podemos trocar o *while* por um *if*.

Outra forma de criar Threads

- Implementar a interface *Runnable*.
- Definir o método abstrato `run()`.
- Criar uma instância da classe (objeto alvo).
- Criar uma instância do Thread, passando o objeto alvo como um parâmetro.
- Invocar `start()` no objeto Thread.
- Aguardar o escalonador invoca `run()` sobre o objeto alvo

Exemplo com *Runnable*

```
public class MyApplet extends Applet
                      implements Runnable {
  private Thread t;
  public void startApplet(){
     t = new Thread(this); // Cria uma nova
                           //   instancia de runnable
     t.start();            // Inicia a Thread
  }
  public void run() { // A nova thread runnable
    …                 // invoca run() e o método
                      // é executado como uma
  }                   // thread separada
  …
}
```