## Introdução

A finalidade principal do artigo é demostrar o funcionamente de thread na linguagem java.

As threads permitem que computadores executem varios processos simultanemente, por consequência o desempenho aumenta razoavekmente e despedico de hardware é minimizado.

A lingugem java disponibiliza varias formas de impplementação de threads, sera demostrado a seguir como as mesmas funcionan e são criadas.

**Java**

é uma [linguagem de programação](http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o) [orientada a objeto](http://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos) desenvolvida na [década de 90](http://pt.wikipedia.org/wiki/D%C3%A9cada_de_1990) por uma equipe de programadores chefiada por [James Gosling](http://pt.wikipedia.org/wiki/James_Gosling), na empresa [Sun Microsystems](http://pt.wikipedia.org/wiki/Sun_Microsystems). Diferentemente das linguagens convencionais, que são [compiladas](http://pt.wikipedia.org/wiki/Compilador) para[código nativo](http://pt.wikipedia.org/wiki/C%C3%B3digo_nativo), a linguagem Java é compilada para um[*bytecode*](http://pt.wikipedia.org/wiki/Bytecode_Java) que é executado por uma [máquina virtual](http://pt.wikipedia.org/wiki/M%C3%A1quina_virtual_Java), sendo assim pode ser executada em varias plataformas.

## Definição de Processo

“Um processo é basicamente um programa em execução, sendo constituído do código executável, dos dados referentes ao código, da pilha de execução, do valor do contador de programa (registrador PC), do valor do apontador do apontador de pilha (registrador SP), dos valores dos demais registradores do hardware, além de um conjunto de outras informações necessárias à execução dos programas.” **Tanenbaum**

Podemos resumir a definição de processo dizendo que o mesmo é formado pelo seu espaço de endereçamento lógico e pela sua entrada na tabela de processos do Sistema Operacional. Assim um processo pode ser visto como uma unidade de processamento passível de ser executado em um computador e essas informações sobre processos são necessárias para permitir que vários processos possam compartilhar o mesmo processador com o objetivo de simular paralelismo na execução de tais processos através da técnica de escalonamento, ou seja, os processos se revezam (o sistema operacional é responsável por esse revezamento) no uso do processador e para permitir esse revezamento será necessário salvar o contexto do processo que vai ser retirado do processador para que futuramente o mesmo possa continuar sua execução.

**Conceitos de threads**

Uma thread java é um fluxo de controle, um objeto do tipo Thread é equivalente a um fluxo de controle de um processo java contendo: dados, prioridades, métodos de pacotes e da aplicação. Uma thread permite um programa ter mais de um fluxo de execução. Um simples estruturado ou orientado a objetos é considerado um processo que por sua vez é uma processo com uma unica thread.

Métodos de uma thread:

run() - código principal do fluxo.

setPriority() - prioridade da thread.

uso de constantes

# máxima: MAX\_PRIORITY = 10

# mímima: MIN\_ \_PRIORITY = 1

# normal: NORM \_PRIORITY = 5

start() - inicia a execução.

**Criação de threads**

Existem dois possiveis métodos.Em ambos os casos a thread somente foi criada para ativar a mesma é necessário o método start().

Estender a classe Thread definindo uma nova subclasse, reescrever o método run() e criar um objeto da nova subclasse considerado a maneira mais simples.

Exemplo de criação:

|  |
| --- |
| public class MyThread extends Thread {  private String whoami;  private int delay;  public MyThread(String name, int d) {  whoami = name;  delay = d;  }  }  public void run() {  try {  sleep(delay);  } catch(InterruptedExcetion e) {}  System.out.println(“Hello, this is  “+whoami+” ! ”);  }  public class TestThreads {  public static void main(String[] args) {  MyThread t1;  t1 = new MyThread(“First”, 1000);  t1.start();  }  } |

**Figura 1. Exemplo de criação de thread, estendendo a classe Thread**

Outra maneira é implementando a interface *Runnable* e o método *run* definido nessa interface. A classe que implementa a *thread* deve declarar um objeto do tipo *Thread* e depois, para instanciá-lo, deve chamar o construtor da classe *Thread* passando como parâmetro a instância da própria classe que implementa *Runnable*, mais o nome da *thread*.

|  |
| --- |
| **public** **class** NewThread **implements** Runnable {  **private** Thread thread = **null**;    //Construtor da classe  **public** NewThread(String threadName) {  **if** (thread == **null**) {  thread = **new** Thread(**this**, threadName);  }  }    //inicia thread  **public** **void** start() {  thread.start();  }    **public** **void** run () {  //execução da thread(oque ela ira fazer)  }  } |

**Figura 2. Exemplo de criação de thread, usando a interface Runnable**

No momento da escolha de qual dos dois métodos sera utilizado deve-se levar em conta que que java não permite herança multipla, quando uma clase for sub-classe de outra classe e é preciso implementar deve-se usar a implemetação da interface Runnable.

**Métodos sincronizados - Sincronizando *Threads* em Java(monitores)**

O controle de execução de sessões críticas em Java é garantido por um mecanismo de monitores. Este mecanismo garante que determinada sessão de código será executada por apenas uma *thread* em um determinado instante de tempo boqueado o acesso de uma ou amis threads concorretnes. Caso uma *thread* deseje executar algum serviço do monitor, o primeiro passo é tentar obter a permissão de execução: caso obtenha a permissão, é garantido que o monitor não esta sendo utilizado por nenhuma outra *thread*. Caso a permissão seja negada, é sinal que uma outra *thread* esta sendo servida pelo monitor; neste caso a *thread* requisitante permanece bloqueada aguardando que o monitor seja liberado. Caso duas ou mais *threads* estejam bloqueadas aguardando o acesso ao monitor, no momento em que o monitor for liberado, uma das *threads* será escalonada a executar, enquanto a outra continuará aguardando. A regra mais comum é de permitir acesso ao monitor à *thread* que está aguardando a mais tempo, contudo esta regra não deve nunca ser considerada para provar a correção de um programa.

A sincronização pode ser aplicada a diferentes níveis de granularidade: **objeto**, **classe** e **bloco**.

**Objeto:**A sincronização a nível de objeto garante que apenas uma thread possa executar métodos dentro do escopo de um objeto.

Como foi visto, cada objeto possui um ponteiro interna: o ponteiro this, que é uma referência ao próprio objeto.

**Classe:**A sincronização a nível de classe garante que apenas uma *thread* possa executar métodos dentro do escopo de uma classe.

Uma classe pode possuir atributos de classe, ou seja, membros cujo escopo é a classe, necessitando portanto de um controle de sincronização a nível de classe. Para garantir a exclusividade de execução de thr*e*ads no escopo de uma classe, faz-se uso de métodos static.

**Bloco**:A sincronização a nível de bloco garante que apenas uma *thread* possa executar o conjunto de instruções definidas dentro do escopo de um bloco.

Para controlar a execução de uma sessão crítica definida por um bloco, é utilizado um mecanismo de sincronização apoiado em um objeto auxiliar.

[**Pools de threads**](http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/threads/pool.html)

Pool de threads funciona de um modo onde é agrupado um determinado némero de threads onde os processos são iniciados paralelamente e jogados no pool, um exemplo, quando todos os processos (threads) tiverem concluído suas tarefas, a execução do programa gerenciador do pool de threads continua normalmente.

Um pool de threads nada mais é do que uma coleção de threads.

Mas porque criar coleções de threads qual é afinalidade de pool de threads? Serve para realizar várias tarefas paralelamente, mas quando a execução do programa que lançou as threads depende da realização de todas as threads antes de continuar.

Este é o comportamento deste exemplo, porém o pool de threads não necessariamente deve aguardar o término de todas as threads para continuar.

Por exemplo, realizar a cópia de vários arquivos ao mesmo tempo, ou seja, o programa gerenciador do pool não pode simplesmente continuar o processo sem que a cópia de todos os arquivos tenha sido realizada, para isso a utilização do pool de threads seria ideal.

Em geral, ligeiramente mais rápido para atender uma solicitação com uma thread ja existente que criar uma nova thread.

**Arquiteturas para pool de Threads**

Única thread o pool vai ser de tamanho 1(um) uma unica threads é utilizada para realizar todas as tarefas.

Tamanho fixo usuário especifica o tamanho do pool e as treads podem ser reaproveitadas para executar as tarefas em uma fila por exemplo.

Tamanho Ilimitado as threads são criadas na medida que são necessarias e as threads que ja existem são reaproveitadas no momento que estiverem disponiveis.

Utilizaremos a interface ExecutorService, A implementação permitem que seja estabelecido:

· O número básico e máximo de pool (número de threads)

· O tipo de estrutura de dados para armazenar as tarefas

· Como tratar tarefas rejeitadas

· Como criar e terminar threads

|  |
| --- |
| package br.com.upf.pool01;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  public class PoolThreads {  public static void main(String[] args) {  ExecutorService executor = Executors.newFixedThreadPool(5);  for (int i = 0; i < 10; i++) {  Runnable worker;  worker = new ThreadWorking("" + i);  executor.execute(worker);  }  executor.shutdown();  while (!executor.isTerminated()) {  }  System.out.println("Finished all threads");  }  } |

**Figura 3. Exemplo de criação de pool de thread**

|  |
| --- |
| package br.com.upf.pool01;  public class ThreadWorking implements Runnable {  private String command;  private static Integer conta = 100;  public ThreadWorking(String s){  this.command=s;  }  @Override  public void run() {  System.out.println(Thread.currentThread().getName()+" Start. Command = "+command);  processCommand();  System.out.println(Thread.currentThread().getName()+" End.\n");  System.out.println("Valor conta: " + conta);  }  private void processCommand() {  try {  conta -= 10;  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  @Override  public String toString(){  return this.command;  }  } |

**Figura 4. Exemplo de criação de pool de thread**

**Estudo de caso - Conta Bancaria simples**

É um sistema bancário acessado por multiplas pessaos simultaneamente, para que os mesmos tenham maior comodidade para realização de consultas, transferências, aplicações sem ter que sair de casa.

Com a implantação do sistema, espera-se obter uma diminuição na demanda de serviços nas agências do banco, o que implica em melhor qualidade do atendimento nas mesmas e maior produtividade interna. O sistema também diminuirá a possibilidade de erros humanos. Os resultados são a satisfação do cliente e redução dos custos nas agências.

O sistema permite aos seus usuários realizar consultas bancárias (saldo), depósitos e transferenica entre contas.

A consulta pode verificar somente a quantia que ele tem no banco solicitando o seu saldo da conta (conta corrente).

Os clientes do banco têm acesso ao sistema através de um cadastramento realizado nos caixas eletrônicos ou no próprio banco, onde informam (criam) seus logins e senhas de acesso.

Dada a heterogeneidade de seus usuários, o sistema precisa ter uma interface agradável e de fácil assimilação. Além disso, o esforço de implantação do sistema nos clientes deve ser o menor possível, de forma a não requisitar recursos extras do banco para a disponibilização de novas versões.

O processo de desenvolvimento envolve a modelagem de todo o sistema bancário para atualizar tecnologicamente os sistemas internos. Além disso, as funcionalidades do sistema serão implantadas de forma gradual – numa primeira etapa serão disponibilizadas as funcionalidades de login, saldo e transferências sendo as operações relacionadas a emissão e desbloqueio de cheques postergadas para uma segunda etapa de implantação do sistema.

ftp://[ftp.inf.ufrgs.br/pub/geyer/PDP-CIC-ECP/slidesAlunos/ProvaP1-2012-1/j01-javaThreadsMonitores-v30-mac-mar2012-red.pdf](http://ftp.inf.ufrgs.br/pub/geyer/PDP-CIC-ECP/slidesAlunos/ProvaP1-2012-1/j01-javaThreadsMonitores-v30-mac-mar2012-red.pdf)

<http://www.dotclass.com.br/portal/?p=702>

bom link para adicionar conteudo

<http://rrmartins.github.io/blog/2012/10/28/threads-e-concorrencia-threads-e-variaveis-part-ii-number-ruby-1-dot-9/>