Universidade Federal da Paraíba

Mestrado em Informática
Fora de Sede
Universidade Tiradentes
Aracaju - Sergipe

Disciplina Sistemas Operacionais

Professor Jacques Philippe Sauvé

Aluno José Maria Rodrigues Santos Júnior

Monografia Threads em Java

Aracaju 08 de março de 2000

Threads em Java

O que é Thread ?

O conceito de thread está intimamente ligado ao conceito de processo, assim é fundamental entender o que são processos, como eles são representados e colocados em execução pelo Sistema Operacional, para em seguida entender as threads. Dessa forma segue uma breve definição de **Processo** e posteriormente a de **Thread**.

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.

Definição de Thread

"Thread, ou processo leve, é a unidade básica de utilização da CPU, consistindo de : contador de programa, conjunto de registradores e uma pilha de execução. Thread são estruturas de execução pertencentes a um processo e assim compartilham os segmentos de código e dados e os recursos alocados ao sistema operacional pelo processo. O conjunto de threads de um processo é chamado de Task e um processo tradicional possui uma Task com apenas uma thread." **Silberschatz**

O conceito de thread foi criado com dois objetivos principais : Facilidade de comunicação entre unidades de execução e redução do esforço para manutenção dessas unidades. Isso foi conseguido através da criação dessas unidades dentro de processos, fazendo com que todo o esforço para criação de um processo, manutenção do Espaço de endereçamento lógico e PCB, fosse aproveitado por várias unidades processáveis, conseguindo também facilidade na comunicação entre essas unidades.

Dessa forma o escalonamento de threads de um mesmo processo será facilitado pois a troca de contexto entre as threads exigirá um esforço bem menor. Sendo que ainda assim, ocorrerá o escalonamento de processos, pois outros processos poderão estar sendo executado paralelamente ao processo que possui as threads. Podemos concluir então que a real vantagem é obtida no escalonamento de threads de um mesmo processo e na facilidade de comunicação entre essas threads.

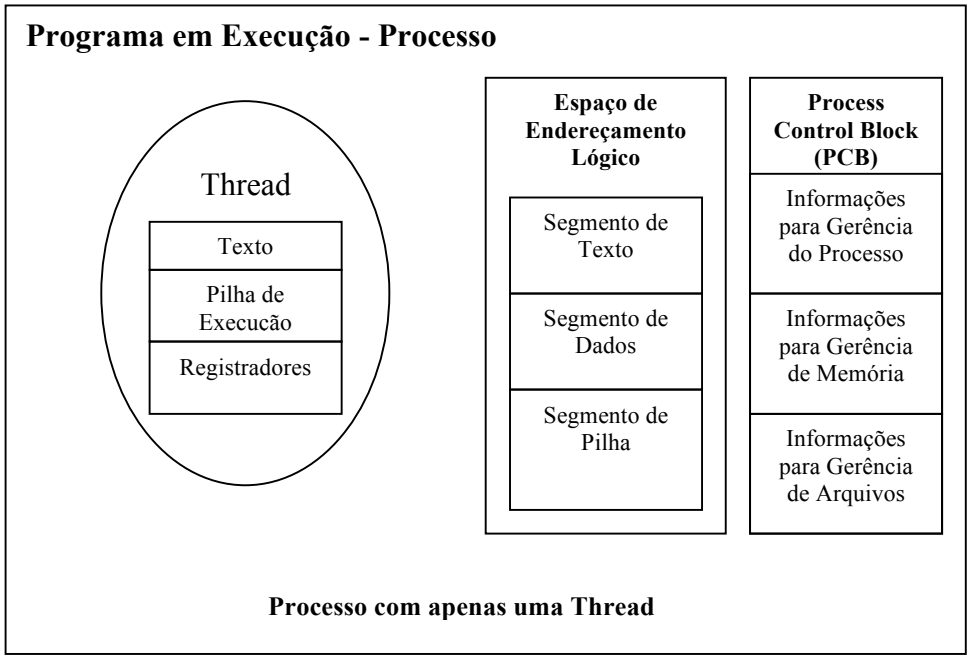
Paralelismo x Concorrência

Threads podem executar suas funções de forma paralela ou concorrente, onde quando as threads são paralelas elas desempenham o seus papeis independente uma das outras. Já na execução concorrente, as threads atuam sobre objetos compartilhados de forma simbiótica necessitando de sincronismo no acesso a esses objetos, assim deve ser garantido o direito de atomicidade e exclusão mútua nas operações das threads sobre objetos compartilhados.

Thread em Java

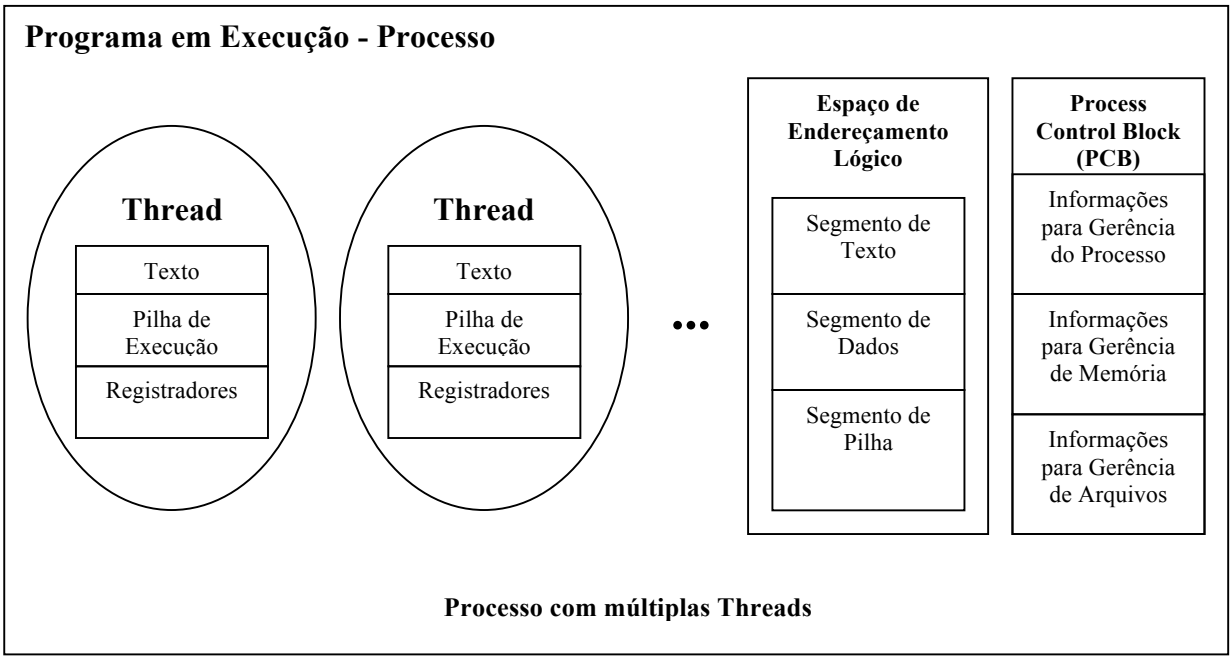
Todo programador está familiarizado com a programação sequencial, pois sem dúvida até o presente momento esta é a forma de programação mais comum. Programas do tipo "Hello World", ou que ordenam uma lista de nome, ou que gera e imprime uma lista de números primos, etc são programas tipicamente sequenciais, onde cada um possui : seu início, sequência de execução e fim e em qualquer momento da sua execução estes programas possuem apenas um ponto de execução.

Uma thread é similar ao programas sequenciais, pois possui um início, sequência de execução e um fim e em qualquer momento uma thread possui um único ponto de execução. Contudo, uma thread não é um programa, ela não pode ser executada sozinha e sim inserida no contexto de uma aplicação, onde essa aplicação sim, possuirá vários pontos de execuções distintos, cada um representado por uma thread. A figura abaixo descreve esse mecanismo de funcionamento da thread dentro de um programa em execução.



Definição : Uma thread representa um fluxo de controle de execução dentro de um programa

Não há nada de novo nesse conceito de processo com uma única thread, pois o mesmo é idêntico ao conceito tradicional de processo. O grande benefício no uso de thread é quando temos várias thread num mesmo processo sendo executadas simultaneamente e podendo realizar tarefas diferentes. A figura abaixo representa um processo com múltiplas threads (Programação Multi-Thread.)



Definição : Com múltiplas theads um programa possui múltiplos pontos de execução

Dessa forma podemos perceber facilmente que aplicações multithreads podem realizar tarefas distintas ao "mesmo tempo", dando idéia de paralelismo. Veja um exemplo do navegador web HotJava, o qual consegue carregar e executar applets, executar uma animação, tocar um som, exibir diversas figuras, permitir rolagem da tela, carregar uma nova página, etc e para o usuário todas essas atividades são simultâneas, mesmo possuindo um único processador. Isso é possível, por que dentro da aplicação do navegador HotJava várias threads foram executadas, provavelmente, uma para cada tarefa a ser realizada.

Alguns autores tratam threads como processos leves. Uma thread é similar a um processo no sentido que ambos representam um único fluxo de controle de execução, sendo considerada um processo leve por ser executada dentro do contexto de um processo e usufruir dos recursos alocados pelo processo. Cada thread necessita possuir apenas as informações (contador de programa e pilha de execução) necessárias a sua execução, compartilhando todo o contexto de execução do processo com todas a demais threads do mesmo processo.

A linguagem Java possui apenas alguns mecanismos e classes desenhadas com a finalidade de permitir a programação Multi-Thread, o que torna extremamente fácil implementar aplicações Multi-Threads, sendo esses :

- A classe `java.lang.Thread` utilizada para criar, iniciar e controlar Threads;
- As palavras reservadas `synchronized` e `volatile` usadas para controlar a execução de código em objetos compartilhados por múltiplas threads, permitindo exclusão mútua entre estas;
- Os métodos `wait`, `notify` and `notifyAll` definidos em `java.lang.Object` usados para coordenar as atividades das threads, permitindo comunicação entre estas.

Criando Threads em Java

A criação de threads em java é uma atividade extremamente simples e existem duas formas distintas de fazê-lo: uma através da herança da classe `Thread`, outra através da implementação da interface `Runnable`, e em ambos os casos a funcionalidade (programação) das threads é feita na implementação do método `run`.

Implementando o Comportamento de uma Thread

O método `run` contém o que a thread faz, ele representa o comportamento (implementação) da thread, é como um método qualquer podendo fazer qualquer coisa que a linguagem Java permita. A classe `Thread` implementa uma thread genérica que por padrão não faz nada, contendo um método `run` vazio, definindo assim uma API que permite a um objeto runnable prover uma implementação para o método `run` de uma thread.

Criando uma subclasse de Thread

A maneira mais simples de criar uma thread em Java é criando uma subclasse de `Thread` e implementando o que a thread vai fazer sobrecarregando o método `run`.

Exemplo de criação de threads estendendo a classe `Thread`

```
/** Criação de NovaThread através da herança da classe Thread */
public class NovaThread extends Thread {
    /** Construtor de NovaThread*/
    public NovaThread (String threadName) {
        /* construtor de Thread passando o nome da Thread como parâmetro */
        super(threadName);
    }
    /** método run o qual representa o comportamento de NovaThread */
    public void run () {
        /* Implementação do comportamento da thread */
    }
}
```

Implementando a Interface `Runnable`

A outra forma de criar threads em Java é implementando a interface `Runnable` e implementando o método `run` definido nessa interface. Sendo que a classe que implementa a thread deve declarar um objeto do tipo `Thread` e para instanciá-lo deve chamar o construtor da classe `Thread` passando como parâmetro a instância da própria classe que implementa `Runnable` e o nome da thread. Como o objeto do tipo `Thread` é local a classe que vai implementar thread, e normalmente privado, há a necessidade de criação de um método `start` para permitir a execução do objeto thread local. O mesmo deve ser feito para qualquer outro método da classe `Thread` que dever ser visto ou usado fora da classe que implementa `Runnable`.

Exemplo de thread implementando a interface `Runnable`

```
/** Criação de NovaThread através da implementação da Interface Runnable */
public class NovaThread implements Runnable {
    /* Objeto local do tipo Thread*/
    private Thread thread = null;

    /* Construtor da classe */
    public NovaThread (String threadName) {
        /* Caso o objeto thread local não tiver sido criado ainda */
        if (thread == null) {
            /* A mesma deve ser instanciada com o nome recebido no construtor
            *  Detalhe importante é que uma instância da própria classe (this)
            * também é passada ao construtor da classe Thread permitindo assim
            * a associação do método run com a thread
            */
            thread = new Thread(this, threadName);
        }
    }

    /* Método para iniciar o objeto thread local */
    public void start() {
        thread.start();
    }

    public void run () {
        /* Comportamento da thead */
    }
}
```

Escolhendo entre os dois métodos de criação de threads

O principal fator a ser levado em consideração na escolha entre um dos dois métodos de criação de thread é que a linguagem Java não permite herança múltipla, assim quando uma classe que já for sub-classe de outra precisar implementar thread é obrigatório que isso seja feito através da implementação da interface `Runnable`, caso contrário pode-se utilizar sub-classes da classe `Thread`. Um exemplo claro desse fator de escolha é na implementação de Applet as quais obrigatoriamente devem ser sub-classes da classe Applet, assim applets só podem implementar threads através da implementação da interface `Runnable`.

Um exemplo mais interessante de threads em Java

Simulação de uma Corridas de Sapos, onde cada sapo na corrida é representado por uma thread.

```
/* Aplicação que simula uma corrida de sapos usando threads */
public class CorridaDeSapos {
    final static int NUM_SAPOS = 5;   // QTE. de sapos na corrida
    final static int DISTANCIA = 500; // Distância da corrida em cm
    public static void main (String[] args) {
        /* colocando sapos na corrida */
        for (int i = 1; i <= NUM_SAPOS; i++) {
            new SapoCorrendoThread("SAPO_" + i, DISTANCIA).start();
        }
    }
}
```

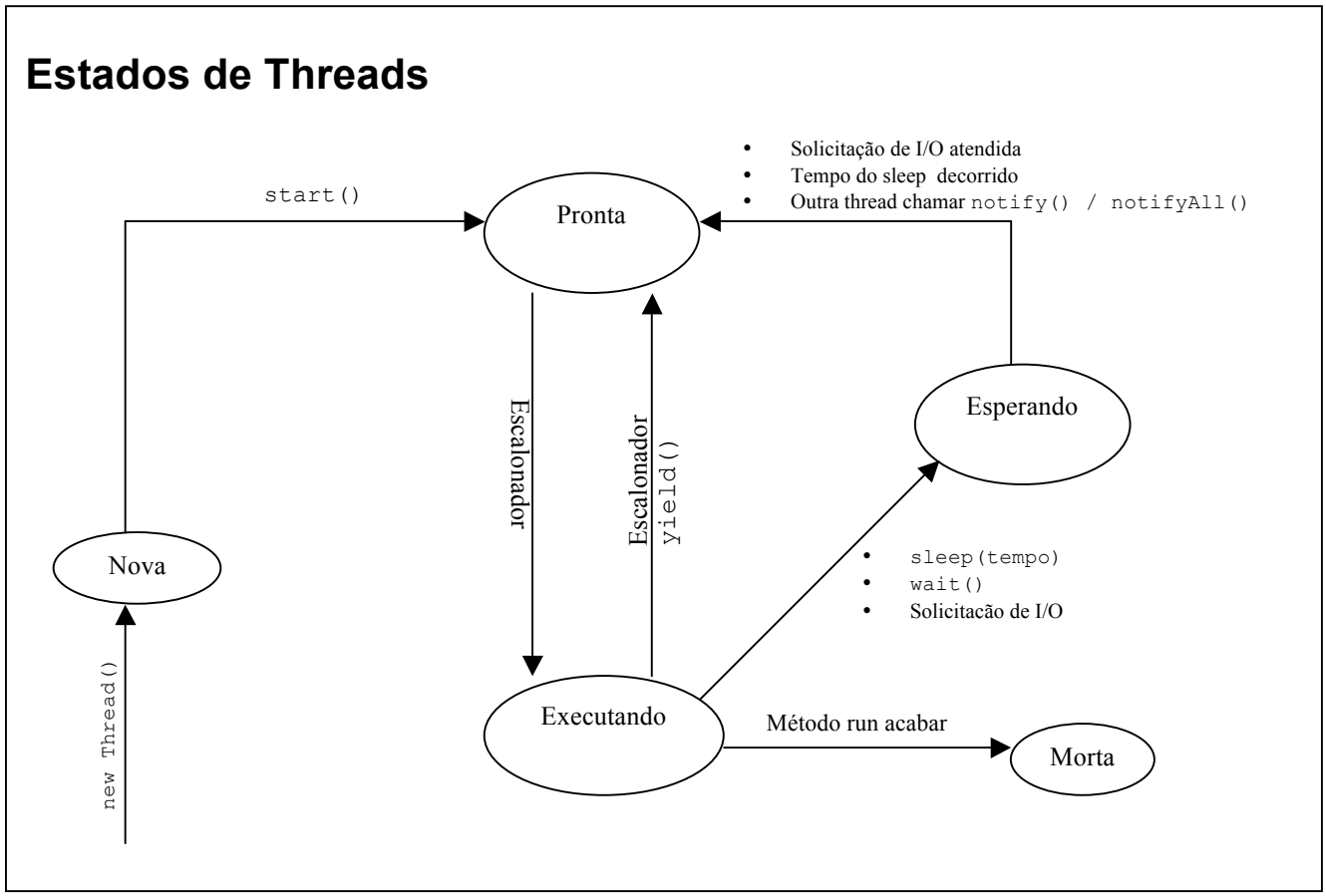
```
/* Classe usando Thread que simula a corrida de um sapo */
class SapoCorrendoThread extends Thread {

    String nome;                    // nome do sapo
    int distanciaCorrida = 0;       // distância já corrida pelo sapo
    int distanciaTotalCorrida;      // distância a ser corrida pelo sapo
    int pulo = 0;                   // pulo do sapo em cm
    int pulos = 0;                  // quantidades de pulos dados na corrida
    static int colocacao = 0;       // colocação do sapo ao final da corrida
    final static int PULO_MAXIMO = 50; // pulo máximo em cm que um sapo pode dar

    /** Construtor da classe. Parâmtros : Nome do Sapo e Distância da Corrida */
    public SapoCorrendoThread (String nome, int distanciaTotalCorrida) {
        /* chamando o construtor de Thread passando o nome do sapo como parâmetro */
        super(nome);
        this.distanciaTotalCorrida = distanciaTotalCorrida;
        this.nome = nome;
    }
    /** Imprime o último pulo do sapo e a distância percorrida */
    public void sapoImprimindoSituacao () {
        System.out.println("O " + nome +  " pulou " + pulo + "cm \t e já percorreu " +
                           distanciaCorrida + "cm");
    }
    /** Faz o sapo pular */
    public void sapoPulando() {
        pulos++;
        pulo = (int) (Math.random() * PULO_MAXIMO);
        distanciaCorrida += pulo;
        if (distanciaCorrida > distanciaTotalCorrida) {
            distanciaCorrida = distanciaTotalCorrida;
        }
    }
    /** Representando o descanso do sapo */
    public void sapoDescansando () {
        /* Método que passa vez a outras threads */
        yield();
    }
    /** Imprime a colocação do sapo ao final da corrida */
    public void colocacaoSapo () {
        colocacao++;
        System.out.println(nome + " foi o " + colocacao +
                           "° colocado com " + pulos + " pulos");
    }
    /** Método run da thread Corrida de Sapos */
    public void run () {
        while (distanciaCorrida < distanciaTotalCorrida) {
            sapoPulando();
            sapoImprimindoSituacao();
            sapoDescansando();
        }
        colocacaoSapo();
    }
}
```

Resultado da Execução – Corrida de Sapos

```
O SAPO_01 pulou 21cm       e já percorreu 21cm
O SAPO_02 pulou 21cm       e já percorreu 21cm
O SAPO_03 pulou 47cm       e já percorreu 47cm
O SAPO_04 pulou 4cm        e já percorreu 4cm
O SAPO_05 pulou 46cm       e já percorreu 46cm
O SAPO_01 pulou 34cm       e já percorreu 55cm
O SAPO_02 pulou 30cm       e já percorreu 51cm
O SAPO_03 pulou 42cm       e já percorreu 89cm
O SAPO_04 pulou 33cm       e já percorreu 37cm
O SAPO_05 pulou 14cm       e já percorreu 60cm
O SAPO_01 pulou 33cm       e já percorreu 88cm
O SAPO_02 pulou 25cm       e já percorreu 76cm
O SAPO_03 pulou 33cm       e já percorreu 122cm
O SAPO_04 pulou 29cm       e já percorreu 66cm
O SAPO_02 pulou 13cm       e já percorreu 89cm
O SAPO_03 pulou 8cm        e já percorreu 130cm
O SAPO_04 pulou 37cm       e já percorreu 103cm
O SAPO_05 pulou 5cm        e já percorreu 65cm
O SAPO_01 pulou 17cm       e já percorreu 105cm
O SAPO_02 pulou 25cm       e já percorreu 114cm
O SAPO_03 pulou 10cm       e já percorreu 140cm
O SAPO_04 pulou 11cm       e já percorreu 114cm
O SAPO_05 pulou 48cm       e já percorreu 113cm
O SAPO_01 pulou 3cm        e já percorreu 108cm
O SAPO_02 pulou 44cm       e já percorreu 158cm
O SAPO_03 pulou 38cm       e já percorreu 178cm
O SAPO_04 pulou 37cm       e já percorreu 151cm
O SAPO_05 pulou 20cm       e já percorreu 133cm
O SAPO_01 pulou 40cm       e já percorreu 148cm
O SAPO_02 pulou 13cm       e já percorreu 171cm
O SAPO_03 pulou 45cm       e já percorreu 200cm
O SAPO_04 pulou 10cm       e já percorreu 161cm
O SAPO_05 pulou 18cm       e já percorreu 151cm
O SAPO_01 pulou 19cm       e já percorreu 167cm
O SAPO_02 pulou 18cm       e já percorreu 189cm
SAPO_03 foi o 1° colocado com 7 pulos
O SAPO_05 pulou 49cm       e já percorreu 200cm
O SAPO_01 pulou 44cm       e já percorreu 200cm
O SAPO_02 pulou 49cm       e já percorreu 200cm
O SAPO_04 pulou 14cm       e já percorreu 175cm
SAPO_05 foi o 2° colocado com 7 pulos
SAPO_02 foi o 3° colocado com 9 pulos
SAPO_01 foi o 4° colocado com 8 pulos
O SAPO_04 pulou 31cm       e já percorreu 200cm
SAPO_04 foi o 5° colocado com 9 pulos
```

O ciclo de vida de uma Thread

A melhor forma de analisar o ciclo de vida de uma thread é através das operações que podem ser feitas sobre as mesmas, tais como : criar, iniciar, esperar, parar e encerrar. O diagrama abaixo ilustra os estados os quais uma thread pode assumir durante o seu ciclo de vida e quais métodos ou situações levam a estes estados.



Ciclo de vida de uma thread
(Nova/Executando/Pronta/Esperando/Morta)

Criando Threads

A criação de uma thread é feita através da chamado ao seu construtor colocando a thread no estado **Nova**, o qual representa uma thread vazia, ou seja, nenhum recurso do sistema foi alocado para ela ainda. Quando uma thread está nesse estado a única operação que pode ser realizada é a inicialização dessa thread através do método `start()`, se qualquer outro método for chamado com a thread no estado **Nova** irá acontecer uma exceção (`IllegalThreadStateException`), assim como, quando qualquer método for chamado e sua ação não for condizente com o estado atual da thread.

Iniciando Threads

A inicialização de uma thread é feita através do método `start()` e nesse momento os recursos necessários para execução da mesma são alocados, tais como recursos para execução, escalonamento e chamada do método `run` da thread. Após a chamada ao método `start` a thread está pronta para ser executada e será assim que for possível, até lá ficará no estado **Pronta**. Essa mudança de estado (Pronta/Executando) será feito pelo escalonador do Sistema de Execução Java. O importante é saber que a thread está pronta para executar e a mesma será executada, mais cedo ou mais tarde de acordo com os critérios, algoritmo, de escalonamento do Sistema de Execução Java.

Fazendo Thread Esperar

Uma thread irá para o estado **Esperando** quando :

- O método `sleep` (faz a thread esperar por um determinada tempo) for chamado;
- O método `wait` (faz a thread esperar por uma determinada condição) for chamado;
- Quando realizar solicitação de I/O.

Quando um thread for para estado Esperando ela retornará ao estado **Pronta** quando a condição que a levou ao estado **Esperando** for atendida, ou seja :

- Se a thread solicitou dormir por determinado intervalo de tempo (`sleep`), assim que este intervalo de tempo for decorrido;
- Se a thread solicitou esperar por determinado evento (`wait`), assim que esse evento ocorrer (outra thread chamar `notify` ou `notifyAll`);
- Se a thread realizou solicitação de I/O, assim que essa solicitação for atendida.

Finalizando Threads

Uma Thread é finalizada quando acabar a execução do seu método `run`, e então ela vai para o estado Morta, onde o Sistema de Execução Java poderá liberar seus recursos e eliminá-la .

Verificando se Threads estão Executando/Pronta/Esperando ou Novas/Mortas

A classe `Thread` possui o método `isAlive`, o qual permite verificar se uma thread está nos estado **Executando/Pronta/Esperando** ou nos estados **Nova/Morta**. Quando o retorno do método for `true` a thread esta participando do processo de escalonamento e o retorno for `false` a thread está fora do processo de escalonamento. Não é possível diferenciar entre **Executando**, **Pronta** ou **Esperando**, assim também como não é possível diferenciar entre **Nova** ou **Morta**.

Threads Daemon

São threads que rodam em background e realizam tarefas de limpeza ou manutenção, as quais devem rodar enquanto a aplicação estiver em execução, as threads daemons somente morrerem quando a aplicação for encerrada. Sendo que o Sistema de Execução Java identifica que uma aplicação acabou quando todas as suas threads estão morta, assim para que seja possível que uma aplicação seja encerrada ainda possuindo threads, tais threads devem ser daemon. Em outras palavras, o Sistema de Execução Java irá terminar uma aplicação quando todas as suas threads **não** daemon morrerem. Threads daemons são denominadas de **Threads de Serviço** e as não daemon são denominadas de **Threads de Usuário**.

Os métodos para manipulação de threads daemon são:

`public final void setDaemon(boolean)`	Torna ou não uma thread daemon
`public final boolean isDaemon()`	Verifica se uma thread é daemon

Escalonamento de Threads

O escalonamento é fundamental quando é possível a execução paralela de threads, pois, certamente existirão mais threads a serem executadas que processadores, assim a execução paralela de threads é simulada através de mecanismos do escalonamento dessas threads, onde os processadores disponíveis são alternados pelas diversas threads em execução. O mecanismo de escalonamento utilizado pelo Sistema de Execução Java é bastante simples e determinístico, e utiliza um algoritmo conhecido como **Escalonamento com Prioridades Fixas**, o qual escalona threads baseado na sua prioridade. As threads escalonáveis são aquelas que estão nos estados **Executando** ou **Pronta**, para isso toda thread possui uma prioridade, a qual pode ser um valor inteiro no intervalo [MIN_PRIORITY ... MAX_PRIORITY], (estas são constantes definidas na classe Thread), e quanto maior o valor do inteiro maior a prioridade da thread. Cada thread Nova recebe a mesma prioridade da thread que a criou e a prioridade de uma thread pode ser alterada através do método `setPriority(int priority)`.

O algoritmo de escalonamento com Prioridades Fixas utilizado pela Sistema de Execução Java funcionada da seguinte forma :

1. Quando várias threads estiverem Prontas, aquela que tiver a maior prioridade será executada.
2. Quando existir várias threads com prioridades iguais, as mesmas serão escalonadas segundo o algoritmo Round-Robin de escalonamento.
3. Uma thread será executada até que : uma outra thread de maior prioridade fique Pronta; acontecer um dos eventos que a faça ir para o estado Esperando; o método `run` acabar; ou em sistema que possuam fatias de tempo a sua fatia de tempo se esgotar.
4. Threads com prioridades mais baixas terão direito garantido de serem executadas para que situações de starvation não ocorram.

Exemplo de aplicação usando threads com diferentes prioridades

Observe que essa corrida não é justa pois os sapos 4 e 5 possuem prioridades máximas e com certeza ganharão a corrida.

```
/* Aplicação que simula uma corrida de sapos usando threads */
public class CorridaDeSapos {
    final static int NUM_SAPOS = 5;       // QTE. de sapos na corrida
    final static int DISTANCIA = 500;     // Distância da corrida em cm

    public static void main (String[] args) {
        /* Criando as threads para representar os sapos correndo */
        SapoCorrendoThread sapo_1 = new SapoCorrendoThread("SAPO_01", DISTANCIA);
        SapoCorrendoThread sapo_2 = new SapoCorrendoThread("SAPO_02", DISTANCIA);
        SapoCorrendoThread sapo_3 = new SapoCorrendoThread("SAPO_03", DISTANCIA);
        SapoCorrendoThread sapo_4 = new SapoCorrendoThread("SAPO_04", DISTANCIA);
        SapoCorrendoThread sapo_5 = new SapoCorrendoThread("SAPO_05", DISTANCIA);
        /* Estabelecendo prioridades para as threads */
        sapo_1.setPriority(Thread.MIN_PRIORITY);
        sapo_2.setPriority(Thread.MIN_PRIORITY);
        sapo_3.setPriority(Thread.NORM_PRIORITY);
        sapo_4.setPriority(Thread.MAX_PRIORITY);
        sapo_5.setPriority(Thread.MAX_PRIORITY);
        /* Iniciando as threads */
        sapo_1.start();
        sapo_2.start();
        sapo_3.start();
        sapo_4.start();
        sapo_5.start();
    }
}
```

Sincronizando Threads (Concorrência)

Até o momento temos abordado threads como unidades de processamento independentes, onde cada uma realiza a sua tarefa sem se preocupar com o que as outras threads estão fazendo (paralelimos), ou seja, abordamos threads trabalhando de forma assíncrona e embora a utilização de threads dessa maneira já seja de grande utilidade, muitas vezes precisamos que um grupo de threads trabalhem em conjunto e agindo sobre objetos compartilhados, onde várias threads desse conjunto podem acessar e realizar operações distintas sobre esses objetos, sendo que, muitas vezes a tarefa de uma thread vai depender da tarefa de outra thread, dessa forma as threads terão que trabalhar de forma coordenada e simbiótica, ou seja, deverá haver uma sincronização entre as threads para que problemas potenciais advindos do acesso simultâneo a esses objetos compartilhado não ocorram.

Esses problemas são conhecidos como *condições de corrida* (Race Conditions) e os trechos de códigos, das threads, que podem gerar condições de corrida são chamados de *regiões críticas* (critical sections), assim as condições de corridas podem ser evitadas através da exclusão mútua das regiões críticas das threads e sincronização entre as threads envolvidas.

O mecanismo para prover exclusão mútua entre threads em Java é bastante simples, necessitando apenas da declaração dos métodos que contém regiões críticas como `synchronized`. Isso garante que quando uma das threads estiver executando uma região crítica em um objeto compartilhado nenhuma outra poderá fazê-lo. Ou seja, quando uma classe possuir métodos sincronizados, apenas um deles poderá estar sendo executado por uma thread.

Implementando Exclusão Mútua de Regiões Críticas

A sincronização entre threads, que acessam concorrentemente um mesmo objeto compartilhado, é feita da seguinte forma : a classe do objeto compartilhado deve possuir métodos que realizem as operações de interesse das threads que o acessam e esses métodos devem ser declarados como `synchronized`, garantido que somente uma thread por vez poderá executar um desses métodos. Assim é fácil perceber que o mecanismo descrito acima fornece, de forma simples, a exclusão mútua de regiões críticas a qual é garantida pelo Sistema de Execução Java. Fazendo uma relação com a teoria de Sistemas Operacionais a linguagem de programação Java fornece um **MONITOR**, onde o **Compilador e o Sistema de Execução** garantem a exclusão mútua e sincronia, provendo assim um mecanismo de alto nível para facilitar a criação de aplicações Multi-Threads.

Comunicação Entre Threads

Um outro aspecto muito importante na implementação de threads que trabalham concorrentemente sobre um objeto compartilhado é que muitas vezes a atividade de uma thread depende da atividade de outra thread, assim é necessário que as threads troquem mensagem afim de avisarem umas as outras quando suas tarefas forem concluídas ou quando precisam esperar por tarefas de outras. Em Java isso é feito através dos métodos `wait`, `notify`, `notifyAll`, onde :

`wait`	Faz a thread esperar (coloca no estado Esperando) em um objeto compartilhado, até que outra thread a notifique ou determinado intervalo de tempo seja decorrido.
`notify`	Notifica (retira do estado Esperando) uma thread que esta esperando em um objeto compartilhado.
`notifyAll`	Notifica todas as threads que estão esperando em um objeto compartilhado, onde uma delas será escalonada para usar o objeto e as outras voltarão ao estado Esperando.

Observação : Esses métodos devem ser usados na implementação das operações dos objetos sincronizados, ou seja, nos métodos synchronized dos objetos sincronizados. Pois só faz sentido uma thread notificar ou esperar por outra em objetos sincronizados e o Sistema de Execução Java tem mecanismos para saber qual thread irá esperar ou receber a notificação, pois somente uma thread por vez pode estar executando um método sincronizado de um objeto compartilhado.

Evitando Starvation e Deadlock

Com a programação concorrente de diversas threads que trabalham em conjunto e acessam objetos compartilhados surge o risco potencial de Starvation e Deadlock, onde Starvation acontece quando uma thread fica esperando por um objeto que nunca lhe será concedido e o Deadlock é quando duas ou mais threads esperam, de forma circular, por objetos compartilhados e nenhuma delas será atendida pelo fato de esperar umas pelas outras.

Na programação Java Starvation e Deadlock devem ser evitados pelo controle lógico de acesso aos objetos compartilhados, fazendo com que o programador deva identificar situação que possam gerar Starvation e Deadlock e evitá-las de alguma forma e com certeza essa é a tarefa mais desafiadora e trabalhosa na programação concorrente.

Agrupando Threads

Toda thread Java é membro de um grupo . Grupo de threads é um mecanismo para agrupar várias threads em um único objeto e permitir realizar operações sobre todas elas mais facilmente. O Sistema de Execução Java coloca threads em grupos no momento da sua criação. Quando uma thread é criada a mesma é colocada num grupo padrão pelo Sistema de Execução Java ou o grupo pode ser especificado pelo programador. Uma vez criada, a thread é membro permanente do seu grupo e a mesma não pode ser movida para outro grupo.

O Grupo Padrão de Threads

Quando uma thread é criada sem a especificação de um grupo no seu construtor, o Sistema de Execução Java a coloca no mesmo grupo da thread que a criou. Quando uma thread for criada em uma aplicação sem a especificação de um grupo a mesma será colocada no grupo padrão "main" o qual é criado na inicialização da aplicação pelo Sistema de Execução Java.

É muito comum que o programa não se preocupe com grupos no uso de threads em Java deixando que o Sistema de Execução Java faça a tarefa de agrupar as threads. Mas, quando uma aplicação for trabalhar com diversas threads é mais conveniente agrupá-las, pois algumas operações deverão ser feitas sobre todas as threads e a criação de grupos de threads irá facilitar tais operações.

Criando Grupos de Threads e Inserindo Threads

A criação de um grupo de threads é feita através do construtor da classe ThreadGroup, onde basta informar o nome do grupo :

```
GroupThread grupoDeThreads = new GroupThread("Nome Grupo de Threads")
```

A colocação de threads em grupos é realizada no momento da sua criação para isso, existem três construtores onde é possível informar o grupo ao qual a thread deve ser inserida.

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String name)
```

Operações sobre Grupos de Threads

As principais operações que podem ser realizadas sobre grupos de threads são :

Método	Descrição
`getName()`	Retorna o nome do grupo
`toString()`	Retorna uma representação do grupo em string
`list()`	Imprime informações sobre o grupo no dispositivo padrão de saída

O Exemplo Produtor/Consumidor ou Buffer Limitado

```
/**
 * @author José Maria Rodrigues Santos Junior
 * zemaria@unitnet.com.br / http://www.unit.br/zemaria
 *
 * Aplicação para simular o problema clássico Produtor/Consumidor
 * de comunicação entre processos usando threads em Java
 */
class ProdutorConsumidor {
    public static void main (String[] args) {

        /* Criando o Buffer Limitado */
        BufferLimitado buffer = new BufferLimitado();
        /* Quantidade de itens a serem produzidos/consumidos pelas threads */
        final int QUANT_ITENS = 10;
        /* Quantidade de threads */
        final int QUANT_THREADS = 5;
        /* Criando os grupos de threads Produtoras/Consumidoras */
        ThreadGroup threadsProdutoras   = new ThreadGroup("Produtores");
        ThreadGroup threadsConsumidoras = new ThreadGroup("Consumidores");
        /* Criando e iniciando e inserindo nos grupos as threads Produtoras */
        for (int i = 1; i <= QUANT_THREADS; i++){
            new ProdutorThread(threadsProdutoras,"Produtor_" + i,
                               buffer,QUANT_ITENS).start();
        }
        /* Criando, iniciando e inserindo nos grupos as threads Consumidoras */
        for (int i = 1; i <= QUANT_THREADS; i++){
            new ConsumidorThread(threadsConsumidoras,"Consumidor_" + i,
                                 buffer,QUANT_ITENS).start();
        }
    }
}
```

```java
/**
  * @author José Maria Rodrigues Santos Junior
  * zemaria@unitnet.com.br / http://www.unit.br/zemaria
  *
  * Classe de thread Produtora
  */

class ProdutorThread extends Thread {
    /* objeto para referenciar o buffer */
    BufferLimitado buffer;
    /* quantidade de itens a serem produzidos */
    int quantItens;

    /**
     * Construtor
     * @param threadGroup Grupo da thread
     * @param nomeThread Nome da thread
     * @param buffer buffer a ser utilizado
     * @param quantItens quantidade de itens a serem produzidos
     */
    public ProdutorThread (ThreadGroup threadGroup,String nomeThread,
                           BufferLimitado buffer, int quantItens)  {
        super(threadGroup, nomeThread);
        this.buffer = buffer;
        this.quantItens = quantItens;
    }

    /*
     * Método run da thread, o qual contém o ser comportamento
     */
    public void run () {
        for (int i = 1; i <= quantItens; i++) {
            /* Gera um novo item aleatoriamente */
            int novoItem = (int) (Math.random() * 10);
            /* Insere o novo item no buffer */
            buffer.insere(novoItem);
            /* dorme pelo intervalo de tempo aleatório [0 ... 100]mseg */
            int tempo = (int) (Math.random() * 1000);
            dorme(tempo);
        }
    }

    private void dorme(int tempo) {
        try {
            sleep(tempo);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```java
/**
  * @author José Maria Rodrigues Santos Junior
  * zemaria@unitnet.com.br / http://www.unit.br/zemaria
  *
  * Classe de thread consumidora
  */
class ConsumidorThread extends Thread {
    /* objeto para referenciar o buffer */
    BufferLimitado buffer;
    /* quantidade de itens a serem consumidos */
    int quantItens;

    /**
     * Construtor
     * @param threadGroup Grupo da thread
     * @param nomeThread Nome da thread
     * @param buffer buffer a ser utilizado
     * @param quantItens quantidade de itens a serem consumidos
     */
    public ConsumidorThread (ThreadGroup threadGroup, String nomeThread,
                             BufferLimitado buffer, int quantItens) {
        super(threadGroup, nomeThread);
        this.buffer = buffer;
        this.quantItens = quantItens;
    }

    /*
     * Método run da thread, o qual contém o ser comportamento
     */
    public void run () {
        for (int i = 1; i <= quantItens; i++) {
            /* Retita um elemtno do buffer */
            buffer.retira();
            /* dorme pelo intervalo de tempo aleatório [0 ... 100]mseg */
            int tempo = (int) (Math.random() * 1000);
            dorme(tempo);
        }
    }

    private void dorme(int tempo) {
        try {
            sleep(tempo);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }

}
```

```java
/**
 * @author José Maria Rodrigues Santos Junior
 * zemaria@unitnet.com.br / http://www.unit.br/zemaria
 *
 * Classe representando o buffer limitado a ser compartilhado pelas threads
 * Produtoras e Consumidoras
 */
class BufferLimitado {
    /* tamano do buffer */
    private final static int TAMANHO = 5;
    /* buffer */
    private int[] buffer = new int[TAMANHO];
    /* posição para inserção de novo item (in) */
    private int posicaoInsere = 0;
    /* posição para remoção do item (out) */
    private int posicaoRetira = 0;
    /* quantidades de itens no buffer */
    private int quantItens = 0;

    /* Construtor */
    public BufferLimitado () {
    }

    /**
     * Insere um novo item no buffer
     * @param novoItem item a ser inserido no buffer
     */
    synchronized public void insere (int novoItem) {
        /* enquanto o buffer estiver cheio a thread deve esperar */
        while (quantItens == TAMANHO) {
            try {
                wait();
            } catch (InterruptedException e){
                System.out.println("Erro de Estado da Thread "
                                   + e.getMessage());
            }
        }
        /* colocando novo item no buffer */
        buffer[posicaoInsere] = novoItem;
        /* exibindo informções sobre a inserção do novo item */
        System.out.print("Inserindo " + novoItem + "\tna posição " + posicaoInsere + "\t");
        /* atualizando a posição de inserção */
        posicaoInsere = proximaPosicao(posicaoInsere);
        /* incrementando a quantidade de itens no buffer */
        quantItens++;
        /* imprimindo o buffer */
        imprime();
        /* caso o buffer estivesse vazio, acordar as threads consumidoras */
        if (quantItens == 1) {
            notifyAll();
        }
    }
```

```java
/**
 * Retira um item do buffer
 * @return Item retirado do buffer
 */
synchronized public int retira () {
    int itemRetirado;  // item retirado
    /* enquanto o buffer estiver vazio a thread deve esperar */
    while (quantItens == 0) {
        try {
            wait();
        } catch (InterruptedException e){
            System.out.println("Erro de Estado da Thread "
                               + e.getMessage());
        }
    }
    /* armazendo o item a ser retirado */
    itemRetirado = buffer[posicaoRetira];
    /* atualizando a quantidade de itens */
    quantItens--;
    /* exibindo informações sobre a retirado do item do buffer */
    System.out.print("Retirando " + itemRetirado
                     + "\tda posição " + posicaoRetira + "\t");
    /* atualizando a  posição de retirado */
    posicaoRetira = proximaPosicao(posicaoRetira);
    /* imprimindo o buffer */
    imprime();
    /* caso o buffer estivesse cheio, acordar as threads produtoras */
    if (quantItens == TAMANHO - 1) {
        notifyAll();
    }

    /* retorna o item retirado */
    return itemRetirado;
}

/**
 * Obtem a próxima posição no buffer
 * @param posicaoAtual a atual posicao no buffer
 * @return a prómixa posição no buffer
 */
private int proximaPosicao (int posicaoAtual) {
    /* obtem a nova posição */
    int novaPosicao = ++posicaoAtual;
    /* caso ultrapasse o tamanho do buffer,irá para o 1ª posição (0) */
    if (novaPosicao > TAMANHO - 1) {
        novaPosicao = 0;
    }

    return novaPosicao;
}

/**
 * Obtem a posição anterior no buffer
 * @param posicaoAtual a atual posicao no buffer
 * @return a posição anterior no buffer
 */
private int posicaoAnterior (int posicaoAtual) {
    /* obtem a nova posição */
    int novaPosicao = --posicaoAtual;
    /* caso seja menor que a posição inicial, irá para a última posição */
    if (novaPosicao < 0) {
        novaPosicao  = TAMANHO-1;
    }

    return novaPosicao;
}
```

```
    /*
     * Imprime o buffer
     */
    synchronized private void imprime () {
        /* posição inicial da impressão */
        int inicio = posicaoRetira;
        /* posição final da impressão */
        int fim = posicaoAnterior(posicaoInsere);
        /* posição impressa */
        int i = inicio;
        /* quantidade de itens impressos */
        int quantItensImpressos = 0;
        System.out.print("\tBuffer[" + quantItens + "] \t= ");
        while (quantItensImpressos < quantItens) {
            System.out.print(buffer[i] + " ");
            i = proximaPosicao(i);
            quantItensImpressos++;
        }
        System.out.println();
    }
}
```

Resultado da Execução – Produtor/Consumidor

```
Inserindo 5  na posição 0        Buffer[1]    = 5
Inserindo 0  na posição 1        Buffer[2]    = 5 0
Inserindo 1  na posição 2        Buffer[3]    = 5 0 1
Inserindo 5  na posição 3        Buffer[4]    = 5 0 1 5
Inserindo 9  na posição 4        Buffer[5]    = 5 0 1 5 9
Retirando 5  da posição 0        Buffer[4]    = 0 1 5 9
Retirando 0  da posição 1        Buffer[3]    = 1 5 9
Retirando 1  da posição 2        Buffer[2]    = 5 9
Retirando 5  da posição 3        Buffer[1]    = 9
Retirando 9  da posição 4        Buffer[0]    =
Inserindo 8  na posição 0        Buffer[1]    = 8
Inserindo 6  na posição 1        Buffer[2]    = 8 6
Inserindo 5  na posição 2        Buffer[3]    = 8 6 5
Retirando 8  da posição 0        Buffer[2]    = 6 5
Inserindo 3  na posição 3        Buffer[3]    = 6 5 3
Inserindo 2  na posição 4        Buffer[4]    = 6 5 3 2
Inserindo 6  na posição 0        Buffer[5]    = 6 5 3 2 6
Retirando 6  da posição 1        Buffer[4]    = 5 3 2 6
Inserindo 3  na posição 1        Buffer[5]    = 5 3 2 6 3
Retirando 5  da posição 2        Buffer[4]    = 3 2 6 3
Inserindo 2  na posição 2        Buffer[5]    = 3 2 6 3 2
Retirando 3  da posição 3        Buffer[4]    = 2 6 3 2
Retirando 2  da posição 4        Buffer[3]    = 6 3 2
Inserindo 1  na posição 3        Buffer[4]    = 6 3 2 1
Retirando 6  da posição 0        Buffer[3]    = 3 2 1
Retirando 3  da posição 1        Buffer[2]    = 2 1
Inserindo 5  na posição 4        Buffer[3]    = 2 1 5
Inserindo 8  na posição 0        Buffer[4]    = 2 1 5 8
Inserindo 6  na posição 1        Buffer[5]    = 2 1 5 8 6
Retirando 2  da posição 2        Buffer[4]    = 1 5 8 6
Inserindo 1  na posição 2        Buffer[5]    = 1 5 8 6 1
Retirando 1  da posição 3        Buffer[4]    = 5 8 6 1
Inserindo 5  na posição 3        Buffer[5]    = 5 8 6 1 5
Retirando 5  da posição 4        Buffer[4]    = 8 6 1 5
Inserindo 9  na posição 4        Buffer[5]    = 8 6 1 5 9
Retirando 8  da posição 0        Buffer[4]    = 6 1 5 9
Inserindo 7  na posição 0        Buffer[5]    = 6 1 5 9 7
Retirando 6  da posição 1        Buffer[4]    = 1 5 9 7
Inserindo 9  na posição 1        Buffer[5]    = 1 5 9 7 9
Retirando 1  da posição 2        Buffer[4]    = 5 9 7 9
Inserindo 7  na posição 2        Buffer[5]    = 5 9 7 9 7
Retirando 5  da posição 3        Buffer[4]    = 9 7 9 7
Retirando 9  da posição 4        Buffer[3]    = 7 9 7
Retirando 7  da posição 0        Buffer[2]    = 9 7
Inserindo 3  na posição 3        Buffer[3]    = 9 7 3
Retirando 9  da posição 1        Buffer[2]    = 7 3
Inserindo 6  na posição 4        Buffer[3]    = 7 3 6
Inserindo 9  na posição 0        Buffer[4]    = 7 3 6 9
Retirando 7  da posição 2        Buffer[3]    = 3 6 9
Inserindo 8  na posição 1        Buffer[4]    = 3 6 9 8
Retirando 3  da posição 3        Buffer[3]    = 6 9 8
Inserindo 2  na posição 2        Buffer[4]    = 6 9 8 2
Inserindo 2  na posição 3        Buffer[5]    = 6 9 8 2 2
Retirando 6  da posição 4        Buffer[4]    = 9 8 2 2
.
.
.
```

O Exemplo do Jantar dos Filósofos Glutões

Exemplo para demonstrar o controle e esforço necessário para prevenção de Starvation/Deadlocks

```
/**
 * Aplicação implementando o clássico problema sobre Deadlock :
 * "O Jantar dos Filósofos Glutões"
 *
 * @author José Maria Rodrigues Santos Junior
 * zemaria@unitnet.com.br
 * http://www.unit.br/zemaria
 */

class JantarDosFilosofos {

   public static void main (String[] args) {
       /* Mesa de jantar para os filósofos */
       MesaDeJantar mesa = new MesaDeJantar ();
       /* Criação das threads representando os cinco filósofos */
       for (int  filosofo = 0; filosofo < 5; filosofo++) {
           new Filosofo("Filosofo_" + filosofo, mesa, filosofo).start();
       }
   }
}
```

```java
/**
 * Classe para representar os filosofos
 *
 * @author José Maria Rodrigues Santos Junior
 * zemaria@unitnet.com.br
 * http://www.unit.br/zemaria
 */
class Filosofo extends Thread {
    /* Tempo máximo (em milesegundos) que o filósofo vai comer ou pensar */
    final static int TEMPO_MAXIMO = 100;
    /* Referência à mesa de jantar */
    MesaDeJantar mesa;
    /* Filósofo na mesa [0,1,2,3,4] */
    int filosofo;

    public Filosofo (String nomeThread, MesaDeJantar mesa, int filosofo) {
        /* construtor da classe pai */
        super(nomeThread);
        this.mesa = mesa;
        this.filosofo = filosofo;
    }

    public void run () {
        int tempo = 0;
        /* Laço representando a vida de um filósofo : pensar e comer */
        while (true) {
            /* sorteando o tempo pelo qual o filósofo vai pensar */
            tempo = (int) (Math.random() * TEMPO_MAXIMO);
            /* filósofo pensando */
            pensar(tempo);
            /* filósofo pegando garfos */
            pegarGarfos();
            /* sorteando o tempo pelo qual o filósofo vai comer */
            tempo = (int) (Math.random() * TEMPO_MAXIMO);
            /* filósofo comendo */
            comer(tempo);
            /* filósofo devolvendo garfos */
            devolverGarfos();
        }
    }

    /* simula o filósofo pensando */
    private void pensar (int tempo) {
        try {
            /* filósofo dorme de tempo milisegundos */
            sleep(tempo);
        } catch (InterruptedException e){
            System.out.println("Filófoso pensou demais, morreu");
        }
    }

    /* simula o filósofo comendo */
    private void comer (int tempo) {
        try {
            sleep(tempo);
        } catch (InterruptedException e){
            System.out.println("Filófoso comeu demais, morreu");
        }
    }

    /* pega os garfos */
    private void pegarGarfos() {
        mesa.pegandoGarfos(filosofo);
    }
    /* devolve os garfos */
    private void devolverGarfos() {
        mesa.devolvendoGarfos(filosofo);
    }
}
```

```java
/**
 * Classe para representar a mesa de jantar, talheres
 * e estados dos fislósofos
 *
 * @author José Maria Rodrigues Santos Junior
 * zemaria@unitnet.com.br
 * http://www.unit.br/zemaria
 */

class MesaDeJantar {
    /* filósofo pensando */
    final static int PENSANDO = 1;
    /* filósofo comendo */
    final static int COMENDO  = 2;
    /* filósofo com fome */
    final static int COM_FOME = 3;
    /* Quantidade de filósofos */
    final static int QUANT_FILOSOFOS   = 5;
    /* Número do primeiro filósofo */
    final static int PRIMEIRO_FILOSOFO = 0;
    /* Número do último filósofo */
    final static int ULTIMO_FILOSOFO = QUANT_FILOSOFOS - 1;
    /* array[0...QUANT_FILOSOFOS - 1] representando os garfos na mesa :
     * true = garfo na mesa; false = garfo com filósofo
     */
    boolean[] garfos = new boolean[QUANT_FILOSOFOS];
    /* array [0...QUANT_FILOSOFOS - 1]
       representando o estado de cada um dos filósofos
     */
    int[] filosofos = new int[QUANT_FILOSOFOS];
    /* Quantas vezes cada filósofo tentou comer e não conseguiu,
     * serve para identificar situações de Starvation
     */
    int[] tentativasParaComer = new int[QUANT_FILOSOFOS];

    /* Construtor */
    public MesaDeJantar() {
        /* Preenchendo os vetores de Garfos e filósofos à mesa */
        for (int i = 0; i < 5; i++) {
            /* Todos os garfos estão na mesa */
            garfos[i] = true;
            /* Todos os filósofos sentam à mesa pensando */
            filosofos[i] = PENSANDO;
            /* Nenhum filósofo tentou comer ainda */
            tentativasParaComer[i] = 0;
        }
    }

    /* filosofo pegando os garfos */
    synchronized void pegandoGarfos (int filosofo) {
        /* filósofo com fome */
        filosofos[filosofo] = COM_FOME;
        /* Deve esperar enquanto algum filósofo vizinho estive comendo */
        while (filosofos[aEsquerda(filosofo)] == COMENDO
            || filosofos[aDireita(filosofo)]  == COMENDO) {
                try {
                    /* Filósofo tentou comer e não conseguiu */
                    tentativasParaComer[filosofo]++;
                    /* colocando o filosofo para esperar */
                    wait();
                } catch (InterruptedException e) {
                    System.out.println("Filósofo morreu de fome");
                }
        }
        /* Filósofo conseguiu comer */
        tentativasParaComer[filosofo] = 0;
        /* retirando os garfos esquerdo e direito da mesa */
        garfos[garfoEsquerdo(filosofo)] = false;
        garfos[garfoDireito(filosofo)]  = false;
        /* Filósofo comendo */
```

```
        filosofos[filosofo] = COMENDO;
        imprimeEstadosFilosofos();
        imprimeGarfos();
        imprimeTentativasParaComer();
    }

    /* Filosofo devolvendo os garfos */
    synchronized void devolvendoGarfos (int filosofo) {
        /* Devolvendo os garfos esquerdo e direito da mesa */
        garfos[garfoEsquerdo(filosofo)] = true;
        garfos[garfoDireito(filosofo)]  = true;
        /* Verificando se há algum filósofo vizinho com fome */
        if (filosofos[aEsquerda(filosofo)] == COM_FOME ||
            filosofos[aDireita(filosofo)]  == COM_FOME) {
            /* Notifica (acorda) os vizinhos com fome */
            notifyAll();
        }
        /* Filósofo pensando */
        filosofos[filosofo] = PENSANDO;
        imprimeEstadosFilosofos();
        imprimeGarfos();
        imprimeTentativasParaComer();
    }

    /* Retorna o número do filósofo a direita */
    private int aDireita (int filosofo) {
        int direito;
        /* Caso seja o filósofo n°5, a sua direita está o filósofo n°1 */
        if (filosofo == ULTIMO_FILOSOFO) {
            direito = PRIMEIRO_FILOSOFO;
        } else {
            /* Caso contrário */
            direito = filosofo + 1;
        }

        return direito;
    }

    /* Retorna o número do filósofo a esquerda */
    private int aEsquerda (int filosofo) {
        int esquerdo;
        /* Caso seja o primeiro filósofo a sua esquerda está o último */
        if (filosofo == PRIMEIRO_FILOSOFO) {
            esquerdo = ULTIMO_FILOSOFO;
        } else {
            esquerdo = filosofo - 1;
        }

        return esquerdo;
    }

    /** Retorna o número do garfo a esquerda do filósofo */
    private int garfoEsquerdo (int filosofo) {
        /* O filósofo 1 possui o garfo 1 a sua esquerda e assim por diante */
        int garfoEsquerdo = filosofo;

        return garfoEsquerdo;
    }

    /** Retorna o número do garfo a direita do filósofo */
    private int garfoDireito (int filosofo) {
        int garfoDireito;
        /* O último filósofo possui o garfo 0 a sua direita*/
        if (filosofo == ULTIMO_FILOSOFO) {
            garfoDireito = 0;
        } else {
            garfoDireito = filosofo + 1;
        }

        return garfoDireito;
    }
```

```
    /* Imprimindo os estados dos filósofos */
    private void imprimeEstadosFilosofos () {
        String texto = "*";
        System.out.print("Filósofos = [ ");
        for (int i = 0; i < QUANT_FILOSOFOS; i++) {
            switch (filosofos[i]) {
                case PENSANDO :
                    texto = "PENSANDO";
                    break;
                case COM_FOME :
                    texto = "COM_FOME";
                    break;
                case COMENDO :
                    texto = "COMENDO";
                    break;
            }
            System.out.print(texto + " ");
        }
        System.out.println("]");
    }

    /* Imprimindo os que estão na mesa */
    private void imprimeGarfos () {
        String garfo = "*";
        System.out.print("Garfos    = [ ");
        for (int i = 0; i < QUANT_FILOSOFOS; i++) {
            if (garfos[i]) {
                garfo = "LIVRE";
            } else {
                garfo = "OCUPADO";
            }
            System.out.print(garfo + " ");
        }
        System.out.println("]");
    }

    /* Imprimindo as tentativas de comer dos dos filósofos */
    private void imprimeTentativasParaComer () {
        System.out.print("Tentou comer = [ ");
        for (int i = 0; i < QUANT_FILOSOFOS; i++) {
            System.out.print(filosofos[i] + "  ");
        }
        System.out.println("]");
    }
}
```

Resultado da Execução – Jantar do Filósofos Glutões

```
Filósofos = [ PENSANDO COMENDO PENSANDO PENSANDO PENSANDO ]
Garfos    = [ LIVRE OCUPADO OCUPADO LIVRE LIVRE ]
Tentou comer = [ 1  2  1  1  1  ]
Filósofos = [ PENSANDO COMENDO COM_FOME COMENDO PENSANDO ]
Garfos    = [ LIVRE OCUPADO OCUPADO OCUPADO OCUPADO ]
Tentou comer = [ 1  2  3  2  1  ]
Filósofos = [ COM_FOME PENSANDO COM_FOME COMENDO COM_FOME ]
Garfos    = [ LIVRE LIVRE LIVRE OCUPADO OCUPADO ]
Tentou comer = [ 3  1  3  2  3  ]
Filósofos = [ COMENDO PENSANDO COM_FOME COMENDO COM_FOME ]
Garfos    = [ OCUPADO OCUPADO LIVRE OCUPADO OCUPADO ]
Tentou comer = [ 2  1  3  2  3  ]
Filósofos = [ COMENDO PENSANDO COM_FOME PENSANDO COM_FOME ]
Garfos    = [ OCUPADO OCUPADO LIVRE LIVRE LIVRE ]
Tentou comer = [ 2  1  3  1  3  ]
Filósofos = [ COMENDO PENSANDO COMENDO PENSANDO COM_FOME ]
Garfos    = [ OCUPADO OCUPADO OCUPADO OCUPADO LIVRE ]
Tentou comer = [ 2  1  2  1  3  ]
Filósofos = [ PENSANDO PENSANDO COMENDO COM_FOME COM_FOME ]
Garfos    = [ LIVRE LIVRE OCUPADO OCUPADO LIVRE ]
Tentou comer = [ 1  1  2  3  3  ]
Filósofos = [ PENSANDO PENSANDO COMENDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO LIVRE OCUPADO OCUPADO OCUPADO ]
Tentou comer = [ 1  1  2  3  2  ]
Filósofos = [ PENSANDO COM_FOME PENSANDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO LIVRE LIVRE LIVRE OCUPADO ]
Tentou comer = [ 1  3  1  3  2  ]
Filósofos = [ PENSANDO COMENDO PENSANDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO OCUPADO OCUPADO LIVRE OCUPADO ]
Tentou comer = [ 1  2  1  3  2  ]
Filósofos = [ PENSANDO COMENDO PENSANDO COM_FOME PENSANDO ]
Garfos    = [ LIVRE OCUPADO OCUPADO LIVRE LIVRE ]
Tentou comer = [ 1  2  1  3  1  ]
Filósofos = [ PENSANDO COMENDO PENSANDO COMENDO PENSANDO ]
Garfos    = [ LIVRE OCUPADO OCUPADO OCUPADO OCUPADO ]
Tentou comer = [ 1  2  1  2  1  ]
Filósofos = [ PENSANDO PENSANDO PENSANDO COMENDO PENSANDO ]
Garfos    = [ LIVRE LIVRE LIVRE OCUPADO OCUPADO ]
Tentou comer = [ 1  1  1  2  1  ]
Filósofos = [ PENSANDO PENSANDO PENSANDO PENSANDO PENSANDO ]
Garfos    = [ LIVRE LIVRE LIVRE LIVRE LIVRE ]
Tentou comer = [ 1  1  1  1  1  ]
Filósofos = [ COMENDO PENSANDO PENSANDO PENSANDO PENSANDO ]
Garfos    = [ OCUPADO OCUPADO LIVRE LIVRE LIVRE ]
Tentou comer = [ 2  1  1  1  1  ]
Filósofos = [ COMENDO PENSANDO COMENDO PENSANDO COM_FOME ]
Garfos    = [ OCUPADO OCUPADO OCUPADO OCUPADO LIVRE ]
Tentou comer = [ 2  1  2  1  3  ]
Filósofos = [ PENSANDO COM_FOME COMENDO COM_FOME COM_FOME ]
Garfos    = [ LIVRE LIVRE OCUPADO OCUPADO LIVRE ]
Tentou comer = [ 1  3  2  3  3  ]
Filósofos = [ PENSANDO COM_FOME COMENDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO LIVRE OCUPADO OCUPADO OCUPADO ]
Tentou comer = [ 1  3  2  3  2  ]
Filósofos = [ PENSANDO COM_FOME PENSANDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO LIVRE LIVRE LIVRE OCUPADO ]
Tentou comer = [ 1  3  1  3  2  ]
Filósofos = [ PENSANDO COMENDO PENSANDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO OCUPADO OCUPADO LIVRE OCUPADO ]
Tentou comer = [ 1  2  1  3  2  ]
Filósofos = [ PENSANDO PENSANDO PENSANDO COM_FOME COMENDO ]
Garfos    = [ OCUPADO LIVRE LIVRE LIVRE OCUPADO ]
.
.
.
```

Bibliografia

LEA, DOUG. “Concurrent Programming in Java - Design Principles and Patterns”. The Java Series, Sun Microsystems, 1996.

Fontes de Pesquisa

http://web2.java.sun.com/docs/books/tutorial/essential/threads/index.html

http://matter.ccet.umc.br/~paulo/homepagejava/threads.html

http://www.inf.unisinos.tche.br/~chris/Paradigma.html#Concorrente

http://www.di.ufpe.br/~java/curso/aula7/threads.htm

http://www.ime.usp.br/~gubi/local/mac431/index.html

http://www.ucs.tche.br/so/javaos/index.htm

http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html

http://www.eng.uiowa.edu/~skaliann/books/jthreads/index.htm

http://java.sun.com/docs/books/vmspec/index.html

http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html