MUNDOJ

MundoJ

Java efetivo

mini
LIVRO

Casa do Código
Livros para o programador
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Sumário

Capítulo 1

Cuidando do lixo: coleta seletiva usando Referências

"Uma das características mais atraentes da linguagem Java frente a suas predecessoras, como C/C++, é a existência do Garbage Collector (GC), que é responsável por realizar o gerenciamento automático de memória, descartando objetos não mais utilizados por uma aplicação. Isso se traduz em ganhos de produtividade por parte dos desenvolvedores, ao mesmo tempo em que torna as aplicações menos propensas a erros e problemas de vazamento de memória. No entanto, ainda assim é possível incorrer em problemas desse tipo quando o ciclo de vida dos objetos de uma aplicação não é tratado adequadamente. Neste artigo, discutimos como as classes de objetos de referência podem ser utilizadas para eximir o desenvolvedor de problemas de vazamento de memória, bem como possibilitar a colaboração inteligente do

GC com as aplicações."
– por Alex Marques Campos e Alexandre Gazola

A maioria das linguagens modernas de programação é dotada de mecanismos de gerenciamento automático de memória por meio de um Garbage Collector (GC), que tem como função descartar espaços de memória alocados e não mais necessários em uma aplicação. Em Java, o GC possui hoje uma implementação bastante eficiente, sendo, em muitos casos, superior em desempenho ao gerenciamento manual feito por algumas linguagens (ver links nas referências para artigo com benchmarks Java vs. C++). A presença do GC torna bem mais produtivo o trabalho dos desenvolvedores, reduzindo bastante a possibilidade de problemas relacionados ao gerenciamento de memória.

Mesmo com essa facilidade, no entanto, é necessário que os desenvolvedores compreendam e saibam trabalhar adequadamente com o ciclo de vida dos objetos de uma aplicação. O conhecido erro `OutOfMemoryError`, por exemplo, pode ser lançado por uma aplicação que esteja manipulando muitos objetos de maneira indevida.

Nesse contexto, uma ferramenta bastante útil, presente no JDK desde sua versão 1.2, e, curiosamente, desconhecida da maioria dos desenvolvedores Java, são as classes de objetos de referência do pacote `java.lang.ref`. Essas classes podem ser utilizadas para se administrar de forma mais controlada o ciclo de vida dos objetos em uma aplicação.

Neste artigo, apresentamos as classes de objetos de referência de Java, mostrando como podem ser utilizadas para tratar problemas sutis de vazamento de memória e também mostramos alguns usos interessantes que permitem que o GC colabore com a aplicação. Para isso, faremos uma introdução básica sobre gerenciamento de memória (alocação dinâmica, referências, garbage collection etc.) e, em seguida, explicaremos, com exemplos práticos, cada uma das classes presentes no pacote `java.lang.ref`.

1.1 Gerenciamento de memória

De maneira bastante simplificada, podemos dividir a área de memória de uma aplicação Java em duas áreas principais: a pilha e o heap. A pilha

é utilizada, entre outras coisas, para armazenar as variáveis locais (e os parâmetros) de um método. Já o heap é uma área destinada à alocação dinâmica de memória, e é onde os objetos são armazenados. Como exemplo, considere o trecho de código exibido na listagem a seguir.

Listagem 1.1 - Diferenças sobre variáveis na pilha e no heap em Java.:

```
void agendarReuniao(int a, int b, String obj) {
    criarCalendario();
    //...
}
void criarCalendario() {
    int c = 5;
    Calendar data = new GregorianCalendar(2009, 01, 01);
    //...
}
```

O método `agendarReuniao()`, quando invocado, recebe como parâmetro dois inteiros e uma `String`. Os inteiros passados como parâmetros têm seus valores armazenados na pilha, ao passo que o objeto `String` referenciado pelo parâmetro `obj` está armazenado no heap. Nesse caso, é armazenado na pilha apenas o endereço da `String` que está no heap. Assumindo que `agendarReuniao()` seja chamado com os parâmetros `a = 2`, `b = 3` e `obj = "Sala 152"`, num ponto qualquer da execução do programa, poderíamos ter a configuração de memória ilustrada na figura 1.1. Como está representado na figura, a pilha é dividida em "frames", ou registros de ativação, os quais contêm as variáveis locais (também chamadas de variáveis automáticas) de cada método da pilha de execução.

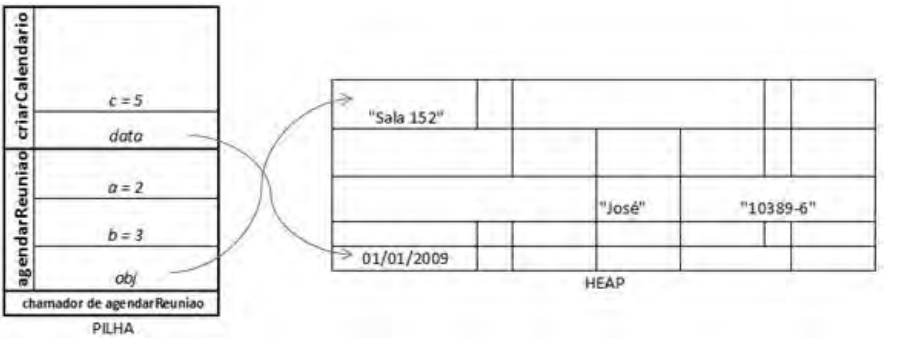

Fig. 1.1: Ilustração de alocação de memória na pilha e no heap.

Quando é solicitada a alocação de um novo objeto (o que é feito por meio do operador `new`), como ocorre na instanciação do `GregorianCalendar` no método `criarCalendario()` da listagem, a Java Virtual Machine (JVM) procura espaço suficiente no heap para armazenar o objeto. Caso não haja espaço suficiente no heap (ou caso a JVM julgue necessário), ela realiza uma chamada ao Garbage Collector (GC) para que este possa eliminar células de memória que, porventura, não estejam mais sendo referenciadas. Explicando de maneira mais detalhada, consideremos o método `criarCalendario()`: ao término da execução desse método, todas as variáveis locais alocadas na pilha perdem seu escopo. Sendo assim, o objeto `GregorianCalendar`, outrora referenciado pela variável `data`, torna-se elegível para ser removido (ou coletado) do heap pelo GC, pois é um objeto que não está mais acessível pela aplicação.

Um detalhe avançado que costuma ser cobrado na certificação de programador Java, e que as pessoas não entendem muito bem o motivo, é que, devido ao fato de as variáveis locais serem alocadas na pilha, classes internas definidas dentro de métodos não podem acessar as variáveis locais desses métodos, a menos que estas sejam especificadas como final, como mostra a próxima listagem. Isso faz todo o sentido, pois o tempo de vida da instância da classe interna pode exceder ao tempo de execução do método da classe externa. Como as variáveis deste último são desalocadas ao término de sua execução, o objeto da classe interna não conseguiria referenciá-las

posteriormente. Uma exceção a isso ocorre quando as variáveis do método externo são finais. Nesse caso, a JVM se encarrega de copiar o valor dessas variáveis para a classe interna e, então, o objeto pode referenciá-las livremente num momento posterior ao término da execução do método externo.

Listagem 1.2 - Acessando variáveis locais externas dentro de uma classe interna.:

```
public class Exemplo {
    public ActionListener getTratadorDeEvento() {
        int x = 5;
        final String str1 = "abc";

        // retorna um objeto de uma classe interna anônima
        return new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                int z = x + 1; // erro de compilacao
                String str2 = str1; // OK, str1 é final.
            }
        };
    }
}
```

A thread do GC executa sempre que se tenta criar um objeto e não existe espaço suficiente no heap (o GC também pode executar em outros momentos, de acordo com a política de coleta de lixo usada, como mencionado anteriormente). Quando isso ocorre, a thread corrente é bloqueada e é realizada uma coleta de lixo. Se o GC não conseguir liberar espaço suficiente e o heap não puder ser expandido pela JVM, então a criação do objeto falha com o lançamento de um `OutOfMemoryError`, que normalmente culmina com o término da thread corrente (se esta não capturar o erro com um bloco try-catch).

O funcionamento básico do Garbage Collector é baseado numa variante do algoritmo Mark-and-Sweep (marcar e varrer), baseado em referências fortes (strong references). De maneira bem simplificada, numa primeira etapa, a etapa de marcação, o GC começa pelas referências raízes (root references),

percorrendo todo o grafo de objetos e marcando todos os objetos que forem alcançados a partir das raízes. Numa etapa posterior, a etapa de varredura, o GC percorre todos os objetos do heap, eliminando os objetos que não tenham sido marcados. Nesse processo, as referências raízes utilizadas pelo GC numa aplicação Java simples são basicamente as variáveis armazenadas na pilha e as variáveis estáticas carregadas pela JVM.

Antes de terminar esta seção, gostaríamos de mencionar a existência do método `finalize()`, presente na classe `java.lang.Object` e, portanto, herdado por todas as classes. Esse método é chamado pelo GC antes de um objeto ser coletado (a implementação herdada de `Object` não faz nada). A idéia do `finalize()` é dar a um objeto a oportunidade de realizar alguma limpeza de recurso (ex.: fechamento de um arquivo) antes de o objeto ser eliminado. Normalmente, não é recomendado confiar nesse método para um correto e eficiente uso da aplicação, pois, como a execução do `finalize()` está atrelada ao GC, pode ser que ele nunca seja executado. Um efeito colateral importante que pode ser desencadeado pelo `finalize()` é a ressurreição de um objeto. Isso pode ser feito passando-se o objeto para uma referência forte ainda em uso pela aplicação (referências fortes são discutidas na seção seguinte).

1.2 Referências

Uma referência forte é o tipo habitual de referência que usamos em Java (ex.: `Calendar data = new GregorianCalendar()`, data é uma referência forte). Toda referência forte mantém em memória o objeto referenciado enquanto a referência forte estiver na pilha (obs.: toda referência raiz é também uma referência forte). Os tópicos a seguir resumem os principais conceitos e um pouco da terminologia que é usada nas seções seguintes:

- **Referência:** Variável que referencia, ou aponta, para um objeto no heap.
- **Objeto referenciado:** Objeto apontado por uma referência.
- **Referência forte:** Referência comum da linguagem Java. Todo objeto referenciado por uma referência forte não é passível de ser coletado

pelo GC.

- **Classe de objeto de referência:** Classes especiais que simulam referências para objetos, mas não são fortes o suficiente para impedir a coleta de seus referenciados pelo GC. Compreendem os tipos `SoftRerence`, `WeakReference` e `PhantomReference`.

Além das referências fortes, também podemos trabalhar com as classes especiais de objetos de referência do pacote `java.lang.ref`: `SoftReference`, `WeakReference` e `PhantomReference` (todas estendem a classe abstrata `Reference`). Instâncias dessas classes são tratadas de maneira diferenciada pelo GC. Diferentemente dos objetos referenciados por referências fortes, objetos referenciados apenas por instâncias das classes `Reference` são elegíveis para coleta pelo Garbage Collector, de acordo com a política vinculada ao tipo de referência. Associados aos tipos de referências, podemos definir quatro níveis de alcance dos objetos referenciados:

- **Fortemente alcançável:** um objeto é dito fortemente alcançável quando for referenciado por, pelo menos, uma referência forte;
- **Suavemente alcançável:** um objeto é dito suavemente alcançável quando não for fortemente alcançável e for referenciado por, pelo menos, uma referência suave (SoftReference);
- **Fracamente alcançável:** um objeto é dito fracamente alcançável quando não for suavemente alcançável e for referenciado por, pelo menos, uma referência fraca (WeakReference);
- **"Fantasmagoricamente" alcançável:** um objeto é dito "fantasmagoricamente" alcançável quando não for fracamente alcançável e for referenciado por, pelo menos, uma referência fantasma (PhantomReference).

A segue uma lista de cada uma das classes encontradas no pacote `java.lang.ref`, as quais são apresentadas em mais detalhes nas próximas seções.

- `Reference`: Classe abstrata para as classes de objetos de referência.
- `SoftReference`: Referência suave. Seus objetos podem ser coletados pelo GC quando houver falta de memória, desde que sejam suavemente alcançáveis. Exemplo: `Reference soft = new SoftReference(obj);`
- `WeakReference`: Referência fraca. Seus objetos podem ser livremente coletados pelo GC, desde que sejam fracamente alcançáveis. Exemplo: `Reference weak = new WeakReference(obj);`
- `PhantomReference`: Referência fantasma. Seus objetos podem ser livremente coletados pelo GC, desde que sejam "fantasmagoricamente" alcançáveis. Exemplo: `Reference phantom = new PhantomReference(obj, aReferenceQueue);`
- `ReferenceQueue`: Fila de referências. Exemplo: `ReferenceQueue queue = new ReferenceQueue();`

1.3 SoftReferences

Uma referência suave permite que um objeto suavemente alcançável permaneça no heap até que a memória por ele ocupada seja requisitada pelo sistema. Para criarmos uma referência suave para um objeto `obj` qualquer, fazemos `Reference softReference = new SoftReference(obj)`. Em outra parte do código, para obtermos o objeto referenciado, chamamos `softReference.get()`. Se ele já tiver sido coletado pelo GC, o método `get()` retornará `null`.

As referências suaves para objetos suavemente alcançáveis terão seus objetos referenciados liberados antes que um `OutOfMemoryError` seja lançado pela JVM. Por esse motivo, as referências suaves podem ser vistas como uma espécie de cache simples de memória. Simples, pelo fato de o desenvolvedor não ter controle sobre a política de descarte dos objetos referenciados (são as regras usadas pela JVM que especificam quando e quais objetos serão coletados). Esse fato não desmerece o papel das referências suaves, pois elas podem nos auxiliar a resolver problemas comuns, como veremos a seguir.

Imagine que escrevemos um programa para diagramação visual de elementos, como um programa de editoração de documentos, e é muito custoso carregar as imagens que o usuário necessita para seu documento (as imagens podem estar em disco, na Web, na rede local etc.). Além disso, precisamos de memória disponível para atender a outros requisitos da aplicação, como, por exemplo, objetos que guardam o posicionamento dos elementos, regras de composição, agrupamentos de elementos etc. Gostaríamos de reter as imagens em memória até que esta seja necessária para outros objetos de negócio de maior importância; todavia, controlar a memória manualmente está fora de cogitação. O que fazer?

Esse é um cenário que pode se beneficiar do uso de referências suaves, como demonstrado no código de exemplo da Listagem 3. A classe funciona da seguinte maneira: quando um usuário necessita de uma imagem, ele chama o método `carregarImagem()`, passando a URI para o arquivo que contém a imagem (lembre-se, a imagem pode estar em disco, na Web etc.). O método verifica se existe uma referência para a dada imagem em um cache de referências. Se existir uma referência no cache, significa que a dada imagem já foi requisitada pelo usuário em algum tempo anterior, mas não garante que os dados da imagem estejam em memória. É aqui que está a diferença: como usamos o tipo `SoftReference`, não sabemos se, em algum ponto anterior a essa chamada, o sistema necessitou de memória para outro objeto e, com isso, liberou algum dos objetos que havíamos carregado. Por isso, precisamos verificar se o objeto referenciado ainda está em memória. Fazemos isso ao mesmo tempo em que criamos uma referência forte para o objeto referenciado (`imagem = imagemRef.get()`), caso este exista.

É importante ressaltar que, sempre que necessitamos utilizar o objeto referenciado, precisamos criar uma referência forte temporária para ele, de modo a garantir que o objeto não será coletado pelo GC enquanto estiver sendo utilizado. Isso também se aplica ao tipo `WeakReference`. Para exemplificar:

```
// código ruim! Não emule isto!
if (ref.get() != null) {
   // objeto referenciado pode ser nulo aqui!
   ref.get().aMethod(); // pode lançar uma NullPointerException
```

```
} else {
   return;
}
```

Voltando ao exemplo, se a imagem continua em memória, nosso trabalho está completo e podemos retorná-la. Se a referência for nula (ou se não existir a referência, pois é a primeira vez em que somos solicitados por uma dada URI), chamamos o método `carregarImagem()` para carregar a imagem de seu armazenamento, criamos uma nova `SoftReference` para ela no cache de referências e retornamos a imagem para o chamador. Todo o processo é transparente, e a memória, do ponto de vista da aplicação, sempre está disponível para os objetos mais importantes, de modo que cumprimos nosso objetivo.

Listagem 1.3 - Carregador de imagens implementado com referências suaves.:

```
public class CarregadorDeImagem {
    private final Map<URI,Reference<Image>> cacheDeImagem;

    public CarregadorDeImagem () {
        cacheDeImagem = new HashMap<URI,Reference<Image>>();
    }

    public Image fetchImage(URI imagePath) {
        Image imagem = null;

        Reference<Image> imagemRef =
            getImagemRefDoCache(imagePath);
        if (imagemRef != null) {
            imagem = imagemRef.get();
        }

        if (imagem == null) {
            imagem = carregarImagem(imagePath);
            adicionarImagemAoCache (imagePath, imagem);
        }
```

```
        return imagem;
    }

    private Reference<Image> getImagemRefDoCache(URI imagemPath){
        return cacheDeImagem.get(imagemPath);
    }

    private void adicionarImagemAoCache(URI imagemPath,
            Image imagem) {
        cacheDeImagem.put(imagemPath,
                new SoftReference<Image>(imagem));
    }

    private Image carregarImagem(URI imagemPath) {
        // Recupera a imagem de seu armazenamento...
        return imagem;
    }
}
```

1.4 WeakReferences

Uma referência fraca é uma referência a um objeto que pode ser livremente coletado pelo GC, caso seja fracamente alcançável. Para criarmos uma referência fraca para um objeto `obj` qualquer, fazemos `Reference weakReference = new WeakReference(obj)`. Em outra parte do código, para obtermos o objeto referenciado, chamamos `weakReference.get()`. Se ele já tiver sido coletado pelo GC, o método `get()` retornará `null`.

Para entendermos a utilidade das referências fracas, consideremos o padrão de projeto Observer. O objetivo deste padrão é estabelecer uma dependência de "um-para-muitos" entre objetos, de maneira que, quando um objeto muda de estado, todos os seus dependentes são notificados e podem ser atualizados. Os participantes principais do padrão são os seguintes:

- Subject (sujeito): objeto que mantém uma lista de objetos observers. O Subject fornece uma interface para acrescentar e remover objetos e envia uma notificação para seus observers quando seu estado muda;

- Observer (observador): objeto que é acrescentado a um Subject e que oferece uma interface de atualização. Dizemos que um Observer observa um Subject.

A listagem a seguir exibe uma implementação tradicional de `Subject` (o `Observer` está abstraído em uma interface com um método update()), enquanto a Listagem seguinte exibe uma implementação de `Subject` que usa referências fracas. A implementação tradicional do padrão estabelece a necessidade de que `Subject` possua um método para remoção de observadores, de forma que, quando um objeto `Observer` não precise mais receber notificações de determinado objeto `Subject`, podemos solicitar ao `Subject` a remoção do referido objeto. No entanto, essa implementação acaba adicionando uma preocupação ao desenvolvedor, o qual precisa estar atento em chamar o método para remoção do `Observer` quando este perder seu escopo de utilização na aplicação. Caso contrário, por continuar na lista do `Subject`, esses observadores permanecerão desnecessariamente no heap, gerando vazamentos de memória.

Listagem 1.4 - Subject com implementação tradicional.:

```
public class Subject {
    private Set<Observer> observadores;

    public Subject() {
        observadores = new HashSet<Observer>();
    }

    public void adicionarObservador(Observer observador) {
        observadores.add(observador);
    }

    public void removerObservador(Observer observador) {
        observadores.remover(observador);
    }

    public void notificarObservadores() {
        for (Observer observador : observadores) {
```

```
            observador.atualizar();
        }
    }
}
```

Listagem 1.5 - Subject com implementação usando referências fracas.:

```
public class Subject {
    private Set<Reference<Observer>> observadores;

    public Subject() {
        observadores = new HashSet<Reference<Observer>>();
    }

    public void adicionarObservador(Observer observador) {
        observadores.add(
            new WeakReference<Observer>(observador));
    }

    public void notificarObservadores() {
          removerReferenciasMortas();
        for (Reference<Observer> ref : observadores) {
            //referência forte necessária
            Observer observador = ref.get();
            if (observador != null) {
                observador.update();
            }
        }
    }

    private void removerReferenciasMortas() {
        Set<Reference<Observer>> copiaObservadores =
            new HashSet<Reference<Observer>>();
        copiaObservadores.addAll(observers);

        for (Reference<Observer> ref : copiaObservadores) {
            Observer observador = ref.get();
            if (observador == null) {
                observadores.remove(ref);
            }
```

```
        }
    }
}
```

A versão com referências fracas, ilustrada na segunda implementação, dispensa completamente a necessidade de que o `Subject` tenha um método `removerObservador()`. Nessa implementação, mantemos um conjunto de referências fracas para os observadores. Dessa forma, quando um `Observer` não é mais necessário em uma aplicação, o simples fato de perder suas referências fortes já o torna elegível para coleta pelo GC. Na próxima seção, veremos como o método `removerReferenciasMortas()` pode ser melhorado por meio do uso da classe `ReferenceQueue`.

1.5 ReferenceQueue

Antes de entender o funcionamento do tipo `PhantomReference`, é necessário entender o funcionamento da `ReferenceQueue`. Se o leitor consultar a documentação, notará que os construtores das referências que vimos até agora, `SoftReference` e `WeakReference`, vêm em dois "sabores": um que só recebe o objeto referenciado e outro que recebe também uma `ReferenceQueue`.

A `ReferenceQueue` (ou fila de referências) é uma estrutura de dados utilizada para suporte ao uso de referências. Seu fim é facilitar a descoberta das referências cujos objetos referenciados já tenham sido marcados para coleta pelo GC e, portanto, já não sejam mais passíveis de serem utilizados (a menos que os objetos sejam ressuscitados pelo método `finalize()`, o que pode ocorrer apenas com referências suaves e fracas).

Uma alternativa ao uso de uma `ReferenceQueue` é percorrer a estrutura de dados em que estão armazenadas as referências e, em seu tempo, remover aquelas cujos objetos referenciados sejam nulos (`reference.get() == null`). Buscar referências inválidas é menos eficiente do que saber de antemão quais referências foram invalidadas. É exatamente esse o propósito da `ReferenceQueue`.

Como exemplo de uso, vejamos como poderíamos adaptar o código do `Subject` discutido na seção anterior de modo a utilizar uma

`ReferenceQueue`, como mostra a listagem a seguir.

Listagem 1.6 - Subject com implementação usando ReferenceQueue.:

```
public class Subject {
    private ReferenceQueue<Observer> referenciasMortas;
    private Set<Reference<Observer>> observadores;

    public Subject() {
        observadores = new HashSet<Reference<Observer>>();
        referenciasMortas = new ReferenceQueue<Observer>();
    }

    public void registrarObservadores(Observer observador) {
        observadores.add(
            new WeakReference<Observer>(
                observador, referenciasMortas));
    }

    public void notificarObservadores() {
        removerReferenciasMortas();
        for (Reference<Observer> ref : observadores) {
            Observer observador = ref.get();
            if (observador != null) {
                observador.atualizar();
            }
        }
    }

    private void removerReferenciasMortas() {
        Reference<?> ref = referenciasMortas.poll();
        while ( ref != null ) {
            observadores.remove(ref);
            ref = referenciasMortas.poll();
        }
    }
}
```

Neste exemplo atualizado, adicionamos uma `ReferenceQueue`, cha-

mada referenciasMortas, na qual registramos cada uma das referências para observadores, `new WeakReference<Observer>(observador, referenciasMortas)`. Quando esses observadores perdem seu escopo e se tornam elegíveis para serem coletados, automaticamente suas referências são postas na `ReferenceQueue` associada. Vale ressaltar que as referências, e não os objetos (os objetos já foram coletados pelo GC), é que são postos na `ReferenceQueue`. O método `removerReferenciasMortas()` tem a responsabilidade de remover todas as referências inválidas da nossa lista de observadores. Esse método é chamado sempre que precisamos notificar os observadores de um `Subject`.

Neste ponto temos uma decisão de projeto muito comum. Por simplicidade, optamos por realizar a limpeza das referências inválidas sempre que o código precisa notificar seus observadores, utilizando o modo de leitura "não-bloqueante" da `ReferenceQueue` o método `poll()`. Poderíamos, entretanto, ter escolhido implementar uma thread que monitorasse a `ReferenceQueue`, de modo a realizar esse trabalho. Ambas são opções válidas, suportadas pela `ReferenceQueue` e qual delas escolher depende muito do contexto de utilização (embora implementações de uma única thread, como a do exemplo, sejam mais comuns). A lista a seguir traz um resumo da API da `ReferenceQueue`, que dá suporte a esses modos de uso. Todos os métodos retiram e retornam um objeto do tipo `Reference<? extends T>` da fila ou retornam `null` quando não existem mais elementos na fila.

- `pool()`: Verifica e retorna uma referência da fila (se existir alguma) ou null se a fila estiver vazia. Este método não bloqueia a thread corrente, retornando imediatamente.

- `remove()`: Verifica e retorna uma referência da fila (se existir alguma) ou faz a thread corrente bloquear, esperando que algum objeto seja nela posto.

- `remove(long timeout)`: Verifica e retorna uma referência da fila (se existir alguma) ou faz a thread corrente bloquear por timeout (em milissegundos) ou até que um objeto seja nela posto o que acontecer

primeiro (um valor de timeout igual a zero faz a thread bloquear indefinidamente, como a chamada de remove sem argumentos faz).

1.6 PhantomReferences

A referência fantasma é o tipo mais fraco de referência disponibilizado pela linguagem Java. Para criarmos uma referência fantasma para um objeto `obj` qualquer, fazemos `Reference phantomReference = new PhantomReference(obj, referenceQueue)`. A referência fantasma possui uma peculiaridade: não podemos nem mesmo consultar qual é o objeto referenciado, ou seja, chamadas ao método `get()` em referências desse tipo sempre retornam `null`. Além disso, uma referência fantasma está sempre associada a uma `ReferenceQueue`. Sendo assim, para quê exatamente servem as referências fantasmas, e como utilizá-las?

Diferentemente das referências suaves e fracas, que, quando associadas a uma `ReferenceQueue` são colocadas nessa fila tão logo sejam marcadas para coleta pelo GC (o que pode ocorrer antes da execução do método `finalize()` dos objetos referenciados), as referências fantasmas só são colocadas na fila de referências após a execução do método `finalize()` dos objetos referenciados e desde que o `finalize()` não ressuscite os objetos. Logo, podemos garantir que os objetos outrora referenciados por referências fantasmas que estejam numa `ReferenceQueue` foram, de fato, eliminados. Em outras palavras, referências fantasmas são a única maneira segura de garantir que um objeto não mais existe. Com isso, constituem-se como uma alternativa (superior) ao método `finalize()` para limpeza de recursos não mais necessários (ex.: fechamento de um arquivo). Referências fantasmas também poderiam ser úteis para testes "caseiros" na busca de vazamentos de memória na aplicação, em casos em que se exige mais agilidade do que a provida pelos profilers.

A listagem a seguir traz um exemplo ilustrando o uso de referências fantasmas. Implementamos um gerenciador para fluxos de entrada a partir de arquivos. Sua finalidade é garantir que o método `close()` do `FileInputStream` será chamado quando não houver mais referências para a URI que define o caminho para o arquivo. Isso permite que o desenvolvedor

não se preocupe em fechar o fluxo explicitamente, bastando apenas manter vivo o objeto URI pelo tempo em que necessitar do fluxo relacionado a ele. Esta idéia pode ser extrapolada para um gerenciador de recursos que forneça handles para os recursos que ele (gerenciador) manipule, simplificando a utilização destes e garantindo sua correta finalização.

Listagem 1.7 - Gerenciador de fluxos de entrada com implementação usando referências fantasmas.:

```
public class GerenciadorDeFluxosDeEntrada {
    private final Map<Reference<URI>, FileInputStream>
                                            fluxosDeEntrada;
    private final ReferenceQueue<URI> referenciasMortas;

    public GerenciadorDeFluxosDeEntrada() {
        fluxosDeEntrada =
            new HashMap<Reference<URI>, FileInputStream>();
        referenciasMortas = new ReferenceQueue<URI>();
    }

    public FileInputStream obterFluxoDeEntrada(URI caminho)
            throws FileNotFoundException {
        removerReferenciasMortas();

        File arquivo = new File(caminho);
        FileInputStream fluxo = new FileInputStream(arquivo);
        Reference<URI> referencia =
            new PhantomReference<URI>(caminho,referenciasMortas);
        fluxosDeEntrada.put(referencia,fluxo);
        return fluxo;
    }

    private void removerReferenciasMortas() {
        Reference<?> ref = referenciasMortas.poll();

        while ( ref != null ) {
            FileInputStream fluxo = fluxosDeEntrada.get(ref);
            fluxosDeEntrada.remove(ref);
```

```
            try {
                fluxo.close();
            } catch (IOException e) {}
            ref.clear();
            ref = referenciasMortas.poll();
        }
    }
}
```

1.7 Exemplo de uso na biblioteca padrão - WeakHashMap

Na biblioteca padrão de Java, temos um exemplo de uso de referências para produzir uma estrutura de dados que junta o desempenho e a facilidade do `HashMap` com a habilidade das `WeakReference`s de não forçar um objeto a permanecer em memória. É a `WeakHashMap`. Essa estrutura de dados utiliza `WeakReference`s para as chaves (e não os valores) que armazena no mapa. Os valores associados a cada chave são referências comuns (fortes).

Esse comportamento é útil quando temos, por exemplo, uma situação em que queremos guardar metadados sobre objetos, mas não queremos que o fato de guardar esses metadados interfira no ciclo de vida normal destes. Outro exemplo é quando queremos associar metadados a objetos, mas as classes desses objetos não podem ser modificadas nem estendidas.

Considere um sistema fictício, a ser utilizado em locadoras, para um quiosque de exibição de vídeos. O sistema deverá exibir aos seus usuários, entre outras informações, vídeos de propaganda e os trailers dos filmes que a locadora possui. Durante o projeto, foram identificadas duas interfaces para o sistema, exibidas nas duas próximas listagens. Uma será utilizada para abstrair o vídeo e a outra para abstrair seus metadados.

Listagem 1.8 - Interface de abstração para um vídeo.:

```
public interface Video {
    public void tocar();
    public void pausar();
    public void parar();
```

```
    public void avancar(double multiplicador);
    public void retroceder(double multiplicador);
    public void buscar(long tempo);
}
```

Listagem 1.9 - Interface de abstração para os metadados de um vídeo.:

```
public interface MetadadosDeVideo {
    public String getTitulo();
    public String getAutor();
    public List<String> getPalavrasChave();
    public long getDuracao();
    public List<String> getElenco();
    public Date getDataDeLancamento();
}
```

A responsabilidade de obter os dados do vídeo será de um subsistema especializado, já que também se pensa em permitir que um servidor de streaming de vídeo seja usado em algum ponto do projeto. Por sua vez, os metadados desses vídeos serão carregados de servidores de banco de dados, arquivos (locais ou na rede), ou ainda por um serviço Web, dependendo da configuração. De modo a uniformizar a experiência do usuário e facilitar o controle dos metadados, foi proposto o cache de metadados dos arquivos sendo exibidos, como exemplificado na listagem a seguir.

Listagem 1.10 - Cache de metadados dos arquivos em exibição.:

```
public class MetadadosDeVideosEmExibicao {
    private final Map<Video, MetadadosDeVideo> informacoesVideos;

    public MetadadosDeVideosEmExibicao() {
        informacoesVideos =
            new WeakHashMap<Video, MetadadosDeVideo>();
    }

    public void registrarVideo(Video video,
            MetadadosDeVideo metadados) {
        informacoesVideos.put(video,metadados);
    }
```

```
    public MetadadosDeVideo obterMetaDados(Video video) {
        return informacoesVideos.get(video);
    }

    public int obterNumeroDeVideosRegistrados() {
        return informacoesVideos.size();
    }
}
```

A classe `MetadadosDeVideosEmExibicao` utiliza um `WeakHashMap` como seu componente central. Quando um vídeo é posto em exibição, seus metadados são carregados e registrados junto à classe `MetadadosDeVideosEmExibicao`, com o método `registrarVideo()`. Esse método, por sua vez, insere uma nova entrada no `WeakHashMap` interno, usando a referência forte do objeto Video passada como parâmetro. A chave dessa nova entrada, contudo, não será a referência forte do objeto Video, mas sim uma referência fraca, criada e mantida internamente pela classe `WeakHashMap`.

Como o leitor deve ter notado, não existe um método para "desregistrar" um metadado de um vídeo. Basta o objeto `Video` em questão se tornar fracamente alcançável (e o objeto ser coletado) para que seus metadados sejam automaticamente eliminados de nosso registro de metadados.

O processo ocorre da seguinte forma: quando o objeto `Video` é coletado, a referência fraca que serve de chave para o metadado desse vídeo é posta em uma `ReferenceQueue` interna ao `WeakHashMap`. Sempre que um `WeakHashMap` é manipulado, um método interno é chamado para eliminar as referências inválidas (as que estão na `ReferenceQueue` interna). Com isso, da próxima vez em que o usuário tentar manipular o mapa, essa chave será eliminada e, junto com ela, a referência forte que o mapa mantém para o metadado desse vídeo. A menos que o usuário tenha criado explicitamente uma outra referência para o objeto de metadados de vídeo, o valor associado à chave eliminada também se torna passível de ser coletado.

Por meio do uso de um `WeakHashMap`, não só simplificamos a interface e a utilização do cache de metadados, como também evitamos vazamentos de memória no caso de o desenvolvedor não se lembrar de "desregistrar" o

metadado quando o vídeo relacionado não for mais necessário.

1.8 Considerações Finais

A presença do Garbage Collector na plataforma Java impulsiona a produtividade no desenvolvimento de aplicações, eximindo os desenvolvedores do compromisso de fazer desalocação explícita de objetos não mais necessários em suas aplicações. No entanto, os desenvolvedores ainda precisam estar atentos ao ciclo de vida dos objetos de suas aplicações, pois, apesar de todo o suporte oferecido pela linguagem, problemas como vazamentos de memória ainda podem ocorrer.

Neste artigo, apresentamos as classes de objetos de referência oferecidas pela linguagem Java, destacando como podem ser usadas para contornar problemas sutis de gerenciamento de memória, e fazendo com que o próprio Garbage Collector possa ser um colaborador nessa tarefa. Por incrível que pareça, mesmo presente desde os primórdios da linguagem Java, a maioria dos desenvolvedores desconhece essa API, que, como vimos, pode ser uma ferramenta bastante útil.

Não obstante, é importante frisar que as classes de objetos de referência devem ser usadas parcimoniosamente numa aplicação, pois podem acarretar impactos em termos de eficiência. Além do overhead ocasionado pelo nível de indireção no acesso aos objetos, existe o trabalho extra que o GC deve realizar durante as coletas de lixo. Mesmo o uso das classes mais úteis (ex.: `WeakHashMap`) para caches simples pode estressar bastante o GC, quando uma implementação de cache LRU de tamanho fixo poderia ser muito mais apropriada e performática (como faz o Hibernate, por exemplo).

Em tempos de globalização e desenvolvimento sustentável, devemos fazer a nossa parte. A memória é um recurso muito importante e devemos utilizá-la com consciência, portanto, trate bem de seus objetos e separe seu lixo!

1.9 Referências

- http://weblogs.java.net/blog/enicholas/archive/2006/05/understanding_w.html

- http://www.kdgregory.com/index.php?page=java.refobj
- Design patterns; Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- A Linguagem de Programação Java, 4 Ed.; Gosling, James; Arnold, Ken; Holmes, David.
- http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ref/package-summary.html
- http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html
- http://www.ibm.com/developerworks/java/library/j-jtp01246.html
- http://chaoticjava.com/posts/how-does-garbage-collection-work/
- http://www.javaranch.com/campfire/StoryInner.jsp
- http://www.janeg.ca/scjp/gc/finalize.html
- Performance Java vs. C++: http://www.idiom.com/\char126zilla/Computer/javaCbenchmark.html, http://www-128.ibm.com/developerworks/java/library/j-jtp09275.html

Capítulo 2

Conhecendo os Parâmetros de Configuração da JVM

"Muitas vezes, ao criarmos uma aplicação JAVA nos deparamos com erros relacionados a estouro de memória, mesmo ainda havendo memória livre. O que realmente aconteceu? Para entendermos perfeitamente as questões envolvidas com esse problema, que eventualmente nos tira o sono, devemos primeiramente observar como a memória é organizada na máquina virtual JAVA. Neste artigo, veremos como a memória é organizada, os parâmetros utilizados para inferir nos tamanhos de alocação de cada região e as causas mais comuns de estouro de memória. Ao final, serão mostradas algumas formas de monitorarmos a memória e como alterar os parâmetros para algumas IDEs e servidores de aplicação."
– por Rodrigo de Azevedo

A linguagem JAVA é dotada de mecanismos singulares existentes em sua gerência de memória. Neste artigo, buscaremos entender tais mecanismos e como a Máquina Virtual Java (JVM) trata cada região que compõe sua memória.

Analisaremos cada uma destas regiões, seu papel e como inferir em sua alocação e outras facilidades proporcionadas pelos parâmetros de iniciação da JVM. Também será possível entender como e porque ocorrem os redimensionamentos de memória.

Evidenciaremos a razão da maioria dos erros relacionados com estouro de memória (`java.lang.OutOfMemoryError`) e algumas formas de evitá-lo.

Veremos algumas formas de monitorar a utilização da memória na JVM utilizando as classes disponíveis no pacote `java.util.management`, acompanhando e verificando a eficiência dos ciclos do Garbage Collector (GC) entre outras formas disponíveis.

Ao final do artigo, teremos visto informação suficiente para entendimento do macroprocesso que a JVM utiliza para gerenciar a memória aumentando a capacidade do leitor em evitar os erros mais comuns que ocasionam o estouro de memória em diferentes regiões da memória.

2.1 Organização da memória na JVM

Antes de verificarmos as possíveis causas de um estouro de memória, vamos primeiro procurar entender como a memória na máquina virtual é organizada e aprender um pouco sobre os conceitos e nomenclaturas envolvidas.

A memória da JVM é dividida em três partes principais:

- Young gen (eden, survivor);
- Old gen (tenured);
- Perm gen (permanent).

Além disso, há ainda a uma área denominada "code cache". A região denominada survivor é dividida em duas subregiões:

- From-space;

- To-space.

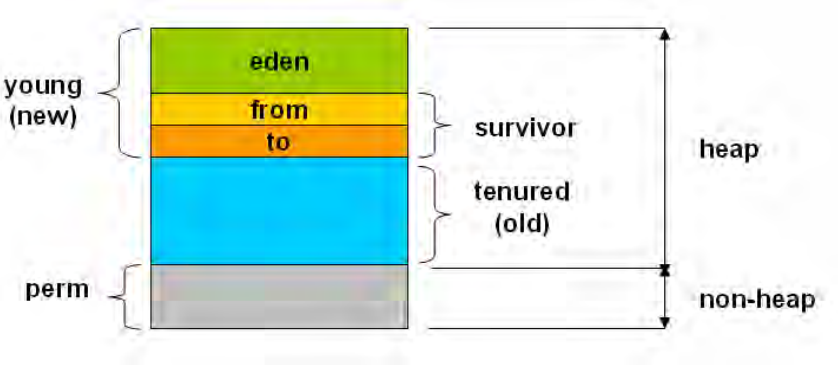

Fig. 2.1: Organização da memória na JVM sem margem para redimensionamentos.

Podemos ainda dividir em dois grandes grupos:

- Heap;
- Non-heap

Grupo heap:

- Young generation (eden, from-space, to-space);
- Old generation (tenured).

Grupo non-heap

- Code cache;
- Permanent generation.

2.2 Young generation e Tenured generation

A grande maioria das alocações realizadas em memória é realizada na região da young generation. Esta região da memória é voltada e otimizada para lidar com objetos que têm um ciclo de vida curto. Objetos que sobrevivem a vários ciclos do Garbage Collector são movidos para a região conhecida como Tenured Generation.

Normalmente a Young Generation é menor que a Tenured e os ciclos de Garbage Collector (GC) são mais frequentes. Na Tenured Generation os ciclos de GC ocorrem com muito menos frequência.

A região denominada Young Generation é dividida em três subregiões:

- eden;
- to-space;
- from-space.

As alocações de objeto ocorrem primeiramente na eden e com o tempo e principalmente depois de alguns ciclos de GC movem-se (parte dos objetos, pois o restante fora recolhido pelo coletor) em direção a região from-space.

Novamente quando estas subregiões ficam cheias ou próximas de sua capacidade, há novo ciclo de GC. Neste momento, a expectativa é que a maioria dos objetos seja recolhida pelo Garbage Collector e aqueles que sobrevivem são agora movidos para a subregião to-space. Após alguns novos ciclos do GC, finalmente os objetos sobreviventes são movidos para a região denominada Old ou Tenured Generation. Na região Tenured, os ciclos do Garbage Collector são menos frequentes e mais lentos, pois é uma região rica em objetos ativos.

Além das regiões citadas, há ainda outra grande região conhecida como Permanent Generation. Nesta, os ciclos de GC ocorrem com muito menos frequência. Nela se encontram, entre outras coisas, os metadados das classes utilizadas.

A região denominada Code Cache é onde ficam alocadas classes compiladas pertencentes à JVM e que dão suporte a sua gestão de alocação de memória, otimizações arquiteturais (RISC x CISC) etc.

Memória heap e problemas com estouro de memória

A memória heap é a área onde os dados normalmente são alocados. Todas as instâncias de classes e os primitivos são alocados nesta área. Uma aplicação JAVA normalmente inicia com o heap utilizando o valor definido pelo parâmetro -Xms. Quando os objetos são criados, estes são alocados nesta região. Se a memória heap chegar ao seu limite, é automaticamente redimensionada até chegar ao valor máximo fornecido pelo parâmetro -Xmx.

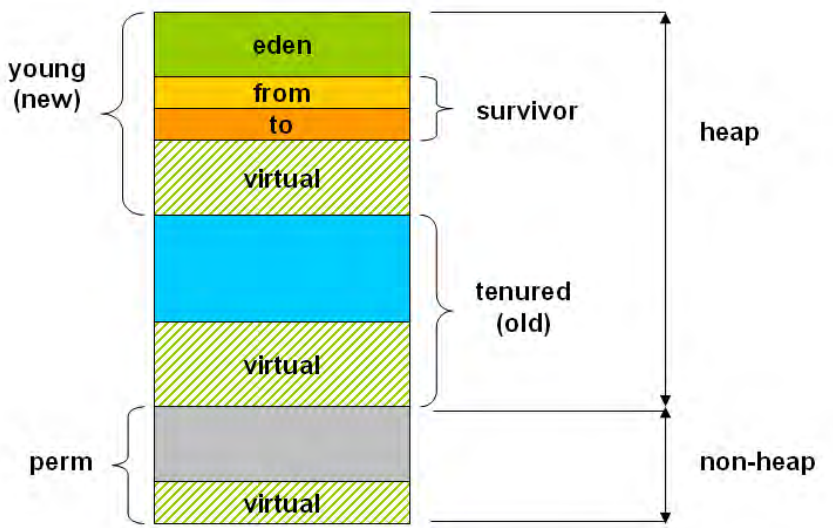

Fig. 2.2: Organização da memória na JVM com margem para ocorrer redimensionamentos quando necessário.

Ao iniciar a Máquina Virtual JAVA todo o espaço definido pelo `-Xmx` é reservado. Se o limite inferior e superior estiver definido com valores diferentes, haverá margem para a JVM realizar redimensionamentos (expansão ou encolhimento) quando necessário. Se ao iniciar JVM, os valores fornecidos de mínimo e máximo forem iguais para algumas das regiões (heap ou perm gen, vide figura 2.1), não será possível ocorrer redimensionamento nesta parte da memória.

O espaço total disponível para o redimensionamento é a diferença entre o limite superior e inferior. Este espaço não é disponibilizado, apesar de alocado, imediatamente para a JVM. Na figura 2.2, este espaço alocado pode ser identificado como "virtual" e esta diferença entre o valor mínimo e máximo é observável tanto na heap quanto na perm gen.

Outras terminologias são utilizadas, como espaço inicial (normalmente definido pelo parâmetro `-Xms`), espaço usado (o espaço realmente utilizado naquele instante), espaço comitado (o tamanho da região naquele instante disponível para JVM) e espaço máximo (normalmente definido pelo parâmetro `-Xmx`).

Entendendo quando acontecem os redimensionamentos da heap

Existem algumas verificações que são realizadas pela JVM na memória heap que identificam se há necessidade de redimensionamento. Os critérios utilizados para realizar qualquer ação de redimensionamento são fornecidos pelos parâmetros `-XX:MinHeapFreeRatio` e `-XX:MaxHeapFreeRatio`.

Estes parâmetros aceitam valores percentuais, isto é, valores que variam entre 0 a 100. Caso estes parâmetros não sejam informados (o que normalmente acontece), existem valores padrões definidos pela JVM. Estes valores percentuais padrões dependem da arquitetura (32-bit ou 64-bit) e do S.O. utilizado.

A dinâmica de funcionamento basicamente segue o seguinte princípio: se for verificado que o espaço livre da memória é superior ao percentual definido no parâmetro `-XX:MaxHeapFreeRatio` (isto é, muita memória livre), a mesma é redimensionada (encolhimento) buscando ficar abaixo do valor definido por este parâmetro, respeitando sempre o limite inferior definido pelo `-Xms`. Por outro lado, se após a passagem de um ciclo de GC, a heap estiver com a memória livre abaixo do percentual mínimo definido pelo parâmetro `-XX:MinHeapFreeRatio` (isto é, escassez de memória), a heap será redimensionada (expansão) buscando ficar acima do valor definido, limitado pela fronteira superior dada pelo parâmetro `-Xmx`.

Quando a memória heap chega ao seu valor máximo e não há mais nenhuma forma de recuperação, mesmo após ter sido executado os processos do Garbage Collector, será lançado pela JVM o erro que in-

dica o estouro de memória (out of memory) representado pela classe `java.lang.OutOfMemoryError`. Normalmente a mensagem que acompanha este erro é *"java.lang.OutOfMemoryError: Java heap space"*.

Motivos para o estouro de memória

A razão mais simples para este incidente está relacionada a uma aplicação grande demais para o tamanho da heap fornecida. Neste caso, a solução é simples, basta aumentar os limites inferiores e superiores da memória. Outro motivo poderia estar relacionado com "memory leak" (vazamento de memória), isto é, os objetos são alocados em memória e após não serem mais utilizados nos processos da aplicação ainda continuam sendo referenciados, impedindo desta forma que o Garbage Collector possa retirá-los da memória. Com o tempo, pode gerar o estouro da memória.

Uma aplicação que, por exemplo, não está de acordo com as boas práticas de programação, má definição de escopo e outras práticas não recomendadas ainda encontradas no mercado torna-se bastante vulnerável a este erro. Podemos dizer que uma aplicação rodeada de más práticas é uma bomba-relógio. Na maioria das vezes, o simples aumento de processamento e uso fará o sistema que até então supostamente funcionava perfeitamente começar a apresentar erros de estouro de memória.

Aumentar o espaço alocado para a heap

Para aumentar o espaço alocado na heap, podemos utilizar dois parâmetros na iniciação da JVM.

```
-Xms<inteiro>[ (padrão byte) | k | m | g]
        Define o tamanho inicial da heap
Exemplo: -Xms4096m
-Xmx<inteiro>[ (padrão byte) | k | m | g]
        Define o tamanho máximo da heap
Exemplo: -Xmx4g
```

Como utilizar os parâmetros da JVM

Opções boleanas são habilitadas utilizando `-XX:+<opção>` e desabilitadas utilizando `-XX:-<opção>`.

Opções numéricas são definidas utilizando `-XX:<opção>=<inteiro>`. Os números podem ser incluídos utilizando "`m`" ou "`M`" para megabytes, "`k`" ou "`K`" para kilobytes e "`g`" ou "`G`" para gigabytes. Se nenhuma letra for definida será utilizada a configuração padrão em bytes. Não é aceito números fracionários, apenas inteiros.

Opções em texto são definidas utilizando `-XX:<opção>=<texto>`; Geralmente especificam um arquivo, um caminho ou uma lista de comandos.

Opções que iniciam com `-X` não são padronizadas e, portanto, podem ser retiradas em futuras implementações da JVM.

Opções que são especificadas como `-XX` podem ser retiradas ou modificadas em futuras versões.

Apesar das duas afirmações acima, é possível notar que nas VMs implementadas a maioria dos parâmetros tem sobrevivido às novas versões.

2.3 JVM 32-bit e sua limitação de uso de memória

Atualmente quando falamos de máquinas extremamente rápidas, além de estarmos normalmente nos referindo a máquinas com múltiplos processadores, estamos nos referindo a memórias extremamente grandes, muito superiores a 4 gigabytes. Como sabemos, uma arquitetura de 32-bits consegue alocar no máximo 4 gigabytes de memória (é uma limitação matemática, afinal 232= 4g).

Um JVM de 32 bits rodando neste sistema conseguirá alocar um pouco menos do que os 4 gigabytes de memória. A maioria das implementações da Máquina Virtual Java, além de sofrerem restrições provenientes do S.O. utilizado, conseguem utilizar algo em torno de 2 a 3.5 gigabytes.

Para conseguirmos utilizar 4 gigabytes ou mais de memória, devemos utilizar uma JVM de 64 bits e um Sistema Operacional e processador cuja máquina virtual possui esse suporte. Ao iniciar a JVM deve-se informar o desejo de utilizar sua função de 64 bits. Isto é feito através do parâmetro -d64.

2.4 Memória non-heap e problemas de estouro de memória

Existem duas regiões (Code Cache e Perm Gen) na memória pertencentes ao grupo non-heap. Normalmente a que apresenta maior incidência de problemas relacionados a estouro de memória é a Perm Gen.

Non-heap Permanent Generation

A Perm Gen é usada para armazenar as instâncias de `Class` (`java.lang.Class`) e metadados (instâncias de classes do pacote `java.lang.reflect`). Aplicações que possuem muitas classes carregadas (classloading) ou geração dinâmica de classes podem rapidamente encher esta região. Outra característica marcante é que os ciclos do Garbage Collector ocorrem com muito menos frequência.

Outra preocupação que deve existir nesta região está relacionada com o "pool de strings". Este pool é basicamente uma tabela hash onde as referências dos objetos String são mantidas. Quando criamos uma String utilizando um literal, esta é alocada automaticamente neste pool. Outra forma disso acontecer é utilizando o método de instância denominado `intern()` da classe `String`.

O método `intern()` utiliza os conceitos do padrão de projeto Flyweight, pois verifica se o objeto `String` responsável pela chamada já existe no pool, se já existir retorna este objeto, e senão o inclui no pool e retorna o próprio objeto.

Antes de continuarmos, é importante entender a listagem a seguir. Note que a declaração utilizada na `s1` será alocada no pool, `s2` não será, pois não é utilizada apenas literais. A `s3` somente utiliza literais, portanto será alocado no pool. A instância da `s4` utiliza a palavra reservada `new` e, portanto, não será alocada no pool. Já os objetos do tipo `String` referenciados pelas

variáveis `s5` e `s6` são inseridos no pool por causa do método `intern()` apesar de na instância inicial utilizando `new` isso não ocorrer.

Listagem 2.1 - Trecho de código que demonstra vários exemplos de alocação de String e sua relação com o pool.:

```
String ola = "Ola ";
String s1 = "Ola Mundo";
String s2 = ola + "Mundo";
String s3 = "Ola " + "Mundo";
String s4 = new String("Ola Mundo");
String s5 = s3.intern();
String s6 = new String("Ola Mundo").intern();
System.out.println(s1 == s2); // Valor exibido: false
System.out.println(s1 == s3); // Valor exibido: true
System.out.println(s1 == s4); // Valor exibido: false
System.out.println(s1 == s5); // Valor exibido: true
System.out.println(s1 == s6); // Valor exibido: true
```

Motivos para utilizar o método `intern()`:

- poupa a alocação dos objetos `String` na memória heap;
- a comparação entre objetos `String` utilizando `==` é normalmente mais rápida que utilizando o método `equals()`. Ex.: `s1.intern() == s2.intern()`.

Motivos para não utilizar:

- Se a alocação não for realizada com cuidado, a Perm Gen poderá sofrer com um estouro de memória.

Listagem 2.2 - Código que ocasionará estouro de memória na Perm Gen depois de alocar Strings demasiadamente no pool.:

```
public class TestPoolString {

    public static void main(String[ ] args) {
```

```
        // Utiliza o padrão de projeto FlyWeight.
        String texto = "Uma pobre String na Perm Gen...";

        List<String> lista = ArrayList<String>();

        for (int i=0; true; i++) {
            String textoFinal = texto + i;
            // Utiliza o padrão de projeto FlyWeight.
            textoFinal = textoFinal.intern();

            // Manter referenciado para que não
            // seja coletado pelo GC.
            lista.add(textoFinal);
        }
    }
}
```

Listagem 2.3 - Código que ocasionará estouro de memória na heap depois de alocar Strings demasiadamente sem utilizar o pool.:

```
public class TestStringSemPool {

    public static void main(String[ ] args){

         // Somente a String abaixo é alocada no pool
        String texto = "Não fui para o pool, fiquei na heap...";

        List<String> lista = new ArrayList<String>();

        for (int i=0; true; i++) {
            // Sem utilizar o pool
            String textoFinal = texto + i;

            // Manter referenciado para que não
            // seja coletado pelo GC.
            lista.add(textoFinal);
        }
    }
}
```

Vale ressaltar que na maioria das vezes a necessidade de ajuste do espaço alocado pode ser ignorada, pois o tamanho padrão definido é adequado. Contudo, conforme mencionado, programas com muitas classes carregadas e alocando `String`s demais no pool podem necessitar de um espaço maior de alocação na Perm Gen.

Deve-se observar que o espaço da Perm Gen é alocado fora da heap. A conseqüência direta disso é que podemos verificar que os parâmetros `-Xms` e `-Xmx` somente alteram o tamanho da heap e em nada afetam o espaço alocado para a Perm Gen. Como vimos, é possível através de parâmetros de iniciação da Máquina Virtual JAVA (JVM), redimensionar vários segmentos da memória, inclusive a "permament generation".

Motivos para o estouro da memória

A principal razão para ocorrer um problema de alocação na Perm Gen diz respeito à quantidade de classes carregadas na JVM. Se a aplicação for muito extensa, provavelmente será necessário utilizar o parâmetro apropriado para aumentar o seu tamanho.

Outro motivo para ocorrer o estouro de memória na Perm Gen está relacionado com os servidores de aplicação que não realizam o redeploying corretamente, acabando por duplicar as classes carregadas (classloaders) em memória. Este é o motivo pelo qual alguns servidores de aplicação não recomendam o redeploy sem restart da aplicação (hot deploy) em ambientes de produção. Normalmente a mensagem que acompanha o estouro de memória na Perm Gen é s"*java.lang.OutOfMemoryError: PermGen space*".

Java HotSpot Server e Client

A Java HotSpot Client VM substituiu a máquina virtual clássica e os compiladores JIT (Just in Time) utilizados em versão anteriores ao Java 2. Está melhor ajustada para ambientes-cliente, pois em essência visa reduzir o tempo de start-up das aplicações e o footprint da memória. É o modo de funcionamento utilizado como padrão das JVMs.

O Java HotSpot Server VM objetiva em essência o uso em sistemas contínuos instalados em servidores de aplicações. A preocupação com a velocidade é crítica, porém a preocupação com o tempo de start-up da aplicação está em

segundo plano.Para executar a JVM neste modo, deve-se no momento da iniciação fornecer o parâmetro `-server`. Exemplo:

```
java -server MinhaAplicacao
```

Aumentar o espaço alocado para a Perm Gen

É possível aumentar o tamanho desta região utilizando na iniciação da Máquina Virtual JAVA (JVM) o parâmetro `-XX:MaxPermSize`.

As JVMs costumam aceitar o parâmetro `-XX:PermSize` que define o tamanho inicial da Perm Gen. Se ao utilizar `-XX:PermSize` e o `-XX:MaxPermSize` for detectado que o valor inicial não é o suficiente para aplicação, recomenda-se colocar o valor inicial e máximo iguais, pois desta forma evita-se alguns ciclos de full do GC que poderiam ocorrer quando a Perm Gen tiver que ser redimensionada.

> **Uso real de memória**
>
> Por exemplo, ao utilizar o parâmetro `-Xms1g` para alterar o tamanho da heap e também a Permament Generation para `128m` utilizando `-XX:MaxPermSize=128m`, tem-se ao final no mínimo o consumo de `1152mb`. É importante mencionar que ainda faltaria ser contabilizada a memória Code Cache, alguma memória que a própria JVM utiliza internamente e a utilizada para stack de cada Thread.

Espaço alocado no Code Cache e possibilidades de estouro

Normalmente, o espaço destinado ao Code Cache é utilizado para compilar e armazenar código nativo. O processo de geração de código nativo a partir do Java varia dependendo da arquitetura da plataforma.

Geralmente é dividido entre a arquitetura x86 e a RISC. Durante a fase de compilação, os métodos são traduzidos em instruções de máquina que podem ser otimizadas dependendo da característica arquitetural. O código compilado é colocado em parte neste espaço.

São raras as situações que motivam a ocorrência de estouro de memória nesta região, entretanto, se após monitoramento for identificado que será necessário aumentá-la, existe a possibilidade de solicitar à Máquina Virtual este aumento.

Para aumentar o espaço alocado no Code Cache é necessário utilizar o parâmetro `-XX:ReservedCodeCacheSize` na iniciação da JVM. `-XX:ReservedCodeCacheSize=<numero>[ (padrão byte) | k | m | g]` define o tamanho do Code Cache.

Segue um exemplo utilizando todos os parâmetros vistos até agora

```
java -server -d64 -Xms4g -Xmx4g -XX:PermSize=128m
    -XX:MaxPermSize=256m -XX:ReservedCodeCacheSize=92m
    MeuServidorDeAplicacao
```

2.5 A memória e as chamadas diretas o Garbage Collector

Iniciarei de fato com uma frase polêmica. É normal que o consumo da memória continuamente aumente e isto não necessariamente significa que há um problema. O Garbage Collector do Java intencionalmente somente é executado quando necessário, pois os ciclos de execução costumam ser custosos. É verdade que vários foram os avanços na versão 5, 6 e 7. Finalizando, um alto consumo da heap é muitas vezes normal.

Os algoritmos utilizados pelo Garbage Collector são, em grande parte das vezes, senão na maioria absoluta, muito mais eficientes se gerenciados pela própria JVM. Não se recomenda que chamemos o `System.gc()` diretamente. A Máquina Virtual JAVA (JVM) tem total condição de determinar quando é mais apropriado para realizar a coleta.

Se for detectável que sua aplicação está tendo problemas constantes no processamento ocasionando pausas em momentos que o Gabage Collector está nitidamente sendo executado sem necessidade, é provável que exista inúmeras chamadas ao `System.gc()`. De fato, o que se vê, e inclusive é matéria da certificação de programador, que a chamada direta ao Garbage Collector não necessariamente fará com que a JVM execute o ciclo do GC, entretanto, é

possível notar que há sim uma indução gerada que muitas vezes interfere na decisão do momento de execução.

Estas chamadas podem ser oriundas de sua aplicação ou simplesmente de alguma API utilizada. Recordo-me que em minha vida profissional tive que utilizar uma API para construção de arquivos compatíveis com o formato Excel que internamente realizava inúmeras chamadas ao GC causando nítidas "paradas" no processamento normal da aplicação. É possível solucionar também este problema, pois as JVMs possuem um parâmetro, `-XX:+DisableExplicitGC`, que pode ser fornecido na iniciação que desabilita toda e qualquer chamada direta ao GC.

2.6 Outros Parâmetros

Seguem outros parâmetros que alteram as dimensões da memória, porém são menos utilizados:

- `-XX:MaxNewSize=<número>` Tamanho máximo da young (new generation). Desde a JDK 1.4 este valor é calculado a partir do NewRatio.
- `-XX:NewRatio=<número>` Razão da young (eden + survivor) sobre a tenured (old generation).
- `-XX:NewSize=<número>` Tamanho padrão da young (new generation).
- `-XX:SurvivorRatio=<inteiro>` Razão da memória eden sobre a survivor.

A maioria dos parâmetros tem valores padrões dependentes do sistema operacional e da arquitetura (32-bit ou 64-bit).

A memória Young Generation é definida pelo parâmetro que define o limite inferior (NewSize) e o parâmetro que define o limite superior (MaxNewSize). Como é composta pela eden e a survivor (to e from space), ao ser redimensionada, acaba por aumentar proporcionalmente todas as regiões envolvidas.

2.7 Como verificar o uso da memória na JVM

Pode ser realizado utilizando o parâmetro -XX:+PrintGCDetails para mostrar informações sobre toda a memória através de cada ciclo de "garbage collection" enquanto o programa estiver em execução, dando ao final um "snapshot" da memória.

A seguir vemos um exemplo de ocorrência de dois ciclos de garbage normal e outro full. O importante é entender a informação. Veja que neste caso a Tenured não foi aliviada após a GC. Note também que o ciclo Full da GC também não conseguiu liberar memória alguma e provavelmente em breve teremos um estouro de memória.

Os ciclos Full do GC normalmente ocorrem antes de qualquer redimensionamento ou de atingir o limite da memória. É um processo mais custoso que um ciclo normal da GC e visa evitar o redimensionamento liberando maior quantidade de memória, pois é mais minucioso (consequentemente mais lento) que o ciclo de GC normal.

```
[ GC [ DefNew: 959K->64K(960K), 0.0014005 secs]
    12289K->11633K(12728K), 0.0014619 secs]
    [ Times: user=0.00 sys=0.00, real=0.00 secs]

[ GC [ DefNew: 960K->64K(960K), 0.0008898 secs]
    [ Tenured: 11808K->11872K(11896K), 0.0367320 secs]
    12529K->11872K(12856K), [ Perm : 18K->18K(12288K)],
    0.0396587 secs] [ Times: user=0.05 sys=0.00, real=0.05 secs]

[ Full GC [ Tenured: 483968K->483968K(483968K), 1.5431730 secs]
     520256K->520255K(520256K), [ Perm : 1691K->1691K(12288K)],
    1.5433371 secs] [ Times: user=1.50 sys=0.03, real=1.53 secs]
```

Ao iniciar a VM com esta opção, será exibido algo semelhante ao abaixo: (neste exemplo foi utilizado a JDK 6 u10).

```
Heap
  def new generation   total 36288K, used 36287K
   [ 0x02990000, 0x050f0000, 0x050f0000)
  eden space 32256K, 100% used
   [ 0x02990000, 0x04910000, 0x04910000)
```

```
  from space 4032K,  99% used
   [ 0x04d00000, 0x050effe8, 0x050f0000)
  to   space 4032K,   0% used
   [ 0x04910000, 0x04910000, 0x04d00000)

 tenured generation   total 483968K, used 483968K
   [ 0x050f0000, 0x22990000, 0x22990000)
  the space 483968K, 100% used
   [ 0x050f0000, 0x22990000, 0x22990000, 0x22990000)

 compacting perm gen  total 12288K, used 1691K
  [ 0x22990000, 0x23590000, 0x26990000)
 the space 12288K,  13% used
  [ 0x22990000, 0x22b36c98, 0x22b36e00, 0x23590000)

No shared spaces configured.
```

O "snapshot" exibido neste artigo pertence a uma aplicação Java finalizada através do método `System.exit()` um pouco antes de ocorrer o estouro de memória na heap. É importante entender o comportamento ao utilizar o parâmetro `-XX:+PrintGCDetails`. Repare que se o estouro da memória ocorresse, a visão da memória exibida seria completamente diferente, pois as regiões já apareceriam esvaziadas.

2.8 Utilizando as classes JAVA do pacote java.lang.management

Programaticamente é possível verificar a memória como um todo utilizando classes do pacote `java.lang.management`. As classes necessárias podem ser visualizadas no diagrama de classes contido na figura 2.3.

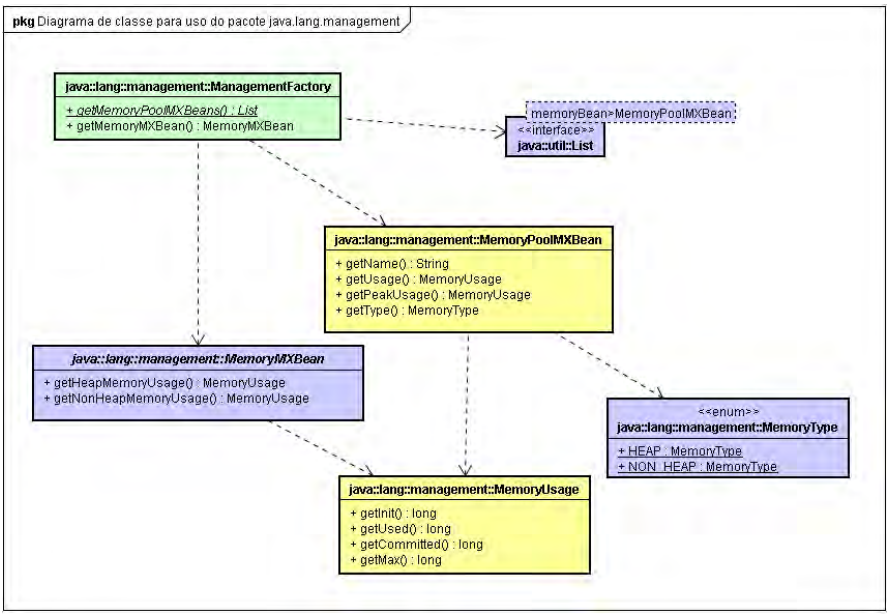

Fig. 2.3: Diagrama de Classes que fornece informações sobre o uso da memória na JVM do pacote java.lang.management.

O código presente na listagem a seguir é utilizado para exemplificar as informações de memória que podem ser obtidas com algumas das classes do pacote `java.lang.management`.

Listagem 2.4 - Implementação da classe que obtém informações da memória.:

```
public class Main {

    public static int MEGA = 1024 * 1024;
    public static String FORMAT = " (%.2fmb)";

    public static void main(String... args) {

        MemoryMXBean memoryBean =
            ManagementFactory.getMemoryMXBean();
```

```
out.println("::: Memória heap :::");
out.println();
MemoryUsage heapMemory = memoryBean.getHeapMemoryUsage();
out.println("Tamanho Inicial :: " + heapMemory.getInit()
    + String.format(Locale.US, FORMAT,
        (double) heapMemory.getInit() / MEGA));
out.println("Tamanho Atual :: " +
    heapMemory.getCommitted()
    + String.format(Locale.US, FORMAT,
        (double) heapMemory.getCommitted() / MEGA));
out.println("Usado :: " + heapMemory.getUsed()
    + String.format(Locale.US, FORMAT,
        (double) heapMemory.getUsed() / MEGA));
out.println("Máximo :: " + heapMemory.getMax()
    + String.format(Locale.US, FORMAT,
        (double) heapMemory.getMax() / MEGA));
out.println();

out.println("::: Memória non-heap :::");
out.println();
MemoryUsage nonheapMemory =
    memoryBean.getNonHeapMemoryUsage();
out.println("Tamanho Inicial :: " +
    nonheapMemory.getInit()
    + String.format(Locale.US, FORMAT,
        (double) nonheapMemory.getInit() / MEGA));
out.println("Tamanho Atual :: " +
    nonheapMemory.getCommitted()
    + String.format(Locale.US, FORMAT,
        (double) nonheapMemory.getCommitted() / MEGA));
out.println("Usado :: " + nonheapMemory.getUsed()
    + String.format(Locale.US, FORMAT,
        (double) nonheapMemory.getUsed() / MEGA));
out.println("Máximo :: " + nonheapMemory.getMax()
    + String.format(Locale.US, FORMAT,
        (double) nonheapMemory.getMax() / MEGA));
out.println();
```

```
        out.println(":: Pool de memórias (organização JVM) ::");
        out.println();
        List<MemoryPoolMXBean> list =
            ManagementFactory.getMemoryPoolMXBeans();

        for (MemoryPoolMXBean m : list) {

            out.println("Nome do Pool :: " + m.getName());
            out.println("Grupo :: " + m.getType());
            out.println();

            MemoryUsage usage = m.getUsage();
            out.println("Inicial: " + usage.getInit()
                + String.format(Locale.US, FORMAT,
                    (double) usage.getInit() / MEGA));
            out.println("Atual: " + usage.getCommitted()
                + String.format(Locale.US, FORMAT,
                    (double) usage.getCommitted() / MEGA));
            out.println("Usado: " + usage.getUsed()
                + String.format(Locale.US, FORMAT,
                    (double) usage.getUsed() / MEGA));
            out.println("Maximo: " + usage.getMax()
                + String.format(Locale.US, FORMAT,
                    (double) usage.getMax() / MEGA));
            out.println();
        }
    }
}
```

2.9 Desmistificando o pool de Strings

Várias confusões de entendimento estão relacionadas ao pool de `Strings`, muitos acreditam que após uma String ser alocada nesta área ficará até que a JVM tenha sua execução interrompida. Outros acreditam que não ocorrem ciclos de GC na Perm Gen. É importante notar que os objetos `String` que são alocados no pool ali permanecem enquanto forem referenciados. Ao perderem a referência, o próximo ciclo de GC irá retirá-los da Perm Gen nor-

malmente.

Em resumo, não acredite que o objeto `String` existirá no pool até o fim da aplicação, pois de fato não é esse o comportamento. Analisemos a classe mencionada na listagem a seguir utilizando as informações que obtemos até agora. Esta classe é responsável pela saída visualizada na listagem posterior.

Listagem 2.5 - Classe que demonstra a ocorrência do GC na região Perm Gen para desalocar objetos String armazenados no pool.:

```
public class TestePermGenString {

    public static final int MEGA = 1024 * 1024;
    public static final String FORMAT = "%.2f mb";

    public static void main(String... args) {
        String str = "Ola mundo";

        List textos = new ArrayList();
        MemoryPoolMXBean mp = getPermGenMemory();

        for (int i = 0; true; i++) {
            String str2 = (str + i).intern();

            if (i % 100000 == 0) {
                MemoryUsage mu = mp.getUsage();
                System.out.println("PermGen inicial: "
                 + String.format(FORMAT,
                        (double) mu.getInit() / MEGA)
                 + " commitada:" + String.format(FORMAT,
                     (double) mu.getCommitted() / MEGA)
                 + " utilizada: " + String.format(FORMAT,
                     (double) mu.getUsed() / MEGA)
                 + " max: " + String.format(FORMAT,
                     (double) mu.getMax() / MEGA));
            }

            if (i == 200000) {
                System.out.println("Retirar a " +
```

```
                    "referência das Strings do pool.");
                textos = new ArrayList();
            }

            textos.add(str2);
        }
    }

    public static MemoryPoolMXBean getPermGenMemory() {
        MemoryPoolMXBean mp = null;
        List<MemoryPoolMXBean> lista =
            ManagementFactory.getMemoryPoolMXBeans();
        for (MemoryPoolMXBean m : lista) {
            if ((m.getType() == MemoryType.NON_HEAP) &&
    (m.getName().toUpperCase().indexOf("PERM") != -1)) {
                mp = m;
                break;
            }
        }
        return mp;
    }
}
```

Nesta classe, utilizamos alguns "números mágicos" para visualizarmos em determinados pontos a situação da Perm Gen. A listagem a seguir nos permite observar várias questões referentes à gerência de memória da JVM. Repare que começamos a aplicação com a Perm Gen inicialmente dimensionada para 4mb e no decorrer da execução este tamanho é constantemente redimensionado (memória disponibilizada para JVM ou commitada) até seu limite de 16mb para que possa suportar as novas alocações no pool de `String`s.

Listagem 2.6 - Visão da dinâmica de execução dos ciclos Full do GC na Perm Gen com o intuito de evitar o redimensionamento até que ocorre o estouro de memória provocado pelo excesso de Strings armazenadas no pool.:

```
PermGen inicial: 4,00 mb commitada:4,00 mb
```

```
    utilizada: 1,72 mb max: 16,00 mb
[ Full GC [ Tenured: 1563K->665K(4096K), 0.0559573 secs]
    2166K->665K(5056K), [ Perm : 8191K->8191K(8192K)],
    0.0560523 secs] [ Times: user=0.06 sys=0.00, real=0.06 secs]
PermGen inicial: 4,00 mb commitada:8,25 mb
    utilizada: 8,24 mb max: 16,00 mb
[ Full GC [ Tenured: 1441K->924K(4096K), 0.0785669 secs]
    1612K->924K(5056K), [ Perm : 12287K->12287K(12288K)],
    0.0786638 secs] [ Times: user=0.08 sys=0.00, real=0.08 secs]
PermGen inicial: 4,00 mb commitada:15,25 mb
    utilizada: 15,11 mb max: 16,00 mb
Retirar a referência das Strings alocadas no pool.
[ Full GC [ Tenured: 2087K->216K(4096K), 0.0481254 secs]
    2933K->216K(5056K), [ Perm : 16384K->4017K(16384K)],
    0.0492858 secs] [ Times: user=0.05 sys=0.00, real=0.05 secs]
[ Full GC [ Tenured: 1046K->493K(4096K), 0.0678745 secs]
    1715K->493K(5056K), [ Perm : 8191K->7276K(8192K)],
    0.0680447 secs] [ Times: user=0.06 sys=0.00, real=0.06 secs]
PermGen inicial: 4,00 mb commitada:9,25 mb
    utilizada: 9,00 mb max: 16,00 mb
[ Full GC [ Tenured: 1010K->924K(4096K), 0.1010554 secs]
    1841K->924K(5056K), [ Perm : 11519K->11519K(11520K)],
    0.1011260 secs] [ Times: user=0.09 sys=0.02, real=0.11 secs]
[ Full GC [ Tenured: 924K->924K(4096K), 0.1642016 secs]
    1518K->924K(5056K), [ Perm : 15615K->15615K(15616K)],
    0.1642776 secs] [ Times: user=0.16 sys=0.00, real=0.16 secs]
PermGen inicial: 4,00 mb commitada:16,00 mb
    utilizada: 15,87 mb max: 16,00 mb
[ Full GC [ Tenured: 2087K->1312K(4096K), 0.1679544 secs]
    2660K->1312K(5056K), [ Perm : 16383K->16383K(16384K)],
    0.1680335 secs] [ Times: user=0.16 sys=0.02, real=0.17 secs]
[ Full GC [ Tenured: 1312K->1306K(4096K), 0.2110858 secs]
    1312K->1306K(5056K), [ Perm : 16383K->16382K(16384K)],
    0.2111585 secs] [ Times: user=0.22 sys=0.00, real=0.22 secs]
[ Full GC [ Tenured: 1306K->1306K(4096K), 0.1619602 secs]
    1324K->1306K(5056K), [ Perm : 16384K->16384K(16384K)],
    0.1620658 secs] [ Times: user=0.14 sys=0.01, real=0.16 secs]
[ Full GC [ Tenured: 1306K->1306K(4096K), 0.1750233 secs]
    1306K->1306K(5056K), [ Perm : 16384K->16384K(16384K)],
```

```
    0.1751015 secs] [ Times: user=0.17 sys=0.00, real=0.17 secs]
[ Full GC [ Tenured: 1306K->143K(4096K), 0.0573083 secs]
    1325K->143K(5056K), [ Perm : 16384K->2186K(16384K)],
    0.0584201 secs] [ Times: user=0.06 sys=0.00, real=0.06 secs]
Exception in thread "main" java.lang.OutOfMemoryError:
    PermGen space
        at java.lang.String.intern(Native Method)
```

Também é possível notar que existe uma relação direta entre a ocorrência de um ciclo Full de GC e o redimensionamento da memória nesta região. Outra informação interessante, ao pararmos de referenciar os objetos String já alocadas no pool, o ciclo Full de GC é eficaz e consegue liberar boa parte da memória.

Continuando a análise da listagem pode-se notar que após a liberação inicial dos objetos não referenciados, continuamente realizamos novas alocações e fatalmente, mesmo apesar de vários ciclos Full de GC (ineficazes pois há referência as estas `String`s alocadas no pool) a memória é totalmente preenchida e ocorre o estouro de memória nesta região.

Intencionalmente foram retiradas da listagem os ciclos de GC não full da heap, pois ocorrem com muito mais frequência e prejudicavam o entendimento do exemplo.

2.10 Considerações Finais

O objetivo deste artigo é aumentar o nível de conhecimento dos profissionais com a organização da memória utilizada na máquina virtual JAVA. Também é intenção apresentar algumas técnicas de verificação de toda a memória buscando evitar um estouro de memória por má configuração. Existem inúmeras aplicações que monitoram a JVM. Há duas que se destacam: JConsole e JVisualVM. Recomendo aos desenvolvedores darem uma olhada nessas ferramentas e buscar também outras alternativas.

2.11 Referências

- Java Application no Solaris http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/java.html

- Organização da memória e GC http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
- Página da SUN sobre os parâmetros de iniciação da JVM http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp
- Perguntas e respostas sobre Java Hotspot VM http://java.sun.com/docs/hotspot/HotSpotFAQ.html
- Perguntas e respostas sobre GC na Java Hotspot VM http://java.sun.com/docs/hotspot/gc1.4.2/faq.html
- Artigo sobre estouro de memória na Permament Generation http://blogs.sun.com/fkieviet/entry/classloader_leaks_the_dreaded_java
- Excelente compilação sobre os parâmetros aceitos pelas JVMs SUN http://blogs.sun.com/watt/resource/jvm-options-list.html

Capítulo 3

Serialização de objetos: da memória para o disco

"Aprenda a utilizar os recursos que a linguagem Java provê ao programador para realizar de maneira simples, o armazenamento de objetos em formato mais simples, a persistência dos dados em formato sequencial."
– por Leandro Yung

Guardar informações contidas nos objetos pode ser muito útil para aplicações. Essas informações podem ser usadas para manter a memória das configurações atuais para um uso futuro, guardar os dados de uma agenda telefônica, cálculos ou qualquer outra coisa. Além disso, podemos usar este mesmo mecanismo para exportar dados entre mesmas aplicações sendo executadas em locais distintos. Ou ainda mais além, transferir informações entre aplicações através da rede.

Tudo isso é possível muito facilmente na tecnologia Java pela existência de mecanismos de conversão de atributos e estados de objetos em formas de dados mais simples, números e bits em um processo chamado de serialização de dados.

3.1 Serialização e Desserialização

Serialização é o processo de armazenar os dados de um objeto e envolve a transformação das variáveis ou atributos de um objeto em memória em bytes ou caracteres, ou seja, converter dados inteligíveis em números para podermos guardar em um arquivo. Desserialização é o processo contrário, que realiza a leitura dos dados em forma de números ou caracteres e constrói um objeto a partir do mesmo.

Felizmente, para o programador Java, a linguagem provê mecanismos simples e automatizados para realizarmos essa tarefa sem a necessidade de escrevermos ou traduzirmos para o computador a maneira que os dados serão escritos ou lidos.

3.2 java.io.Serializable

Serialização de dados é o processo mais simples de armazenamento ou persistência de objetos. Para tanto, um objeto que desejamos armazenar, precisa apenas implementar uma interface chamada `java.io.Serializable`. Essa interface não possui nenhum método, apenas serve para indicar ao Java que os objetos da classe e subclasses são possíveis de serem persistidos de maneira serializada.

Vamos criar uma classe simples chamada `Pessoa` e iremos permitir a gravação desta no disco, conforme listagem a seguir.

Listagem 3.1 - Classe que será serializada:

```
import java.io.Serializable;

public class Pessoa implements Serializable {
    private String nome;
```

```
    private String telefone;

    public Pessoa() {
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getTelefone() {
        return telefone;
    }
    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}
```

Pode-se notar que a classe `Pessoa` não possui métodos distintos, apenas a marcação `implements Serializable` e isso já basta. Na classe, definimos os atributos `nome` e `telefone` do tipo `String`. Em seguida é definido um construtor, usado durante a criação do objeto, que no caso é um construtor vazio. Nas linhas de código seguintes são definidos os métodos de acesso para os atributos.

Métodos de acesso são uma boa prática para evitar que os dados de um objeto sejam vistos e alterados de maneira indevida. Dessa maneira, o acesso direto para uma variável é bloqueada e só é permitida se o programador tiver escrito o método para ter acesso. Na prática, então, só se pode ler e escrever aquilo que a classe permite. Assim, por exemplo, não deveria ser acessível para leitura o valor de uma senha, caso contrário, qualquer programa poderia ir ao objeto e recuperar a senha do mesmo. Da mesma forma, não pode ser acessível para modificação o campo com o total da compra de um carrinho de compras, caso contrário, um programa poderia ir e alterar o mesmo, gerando uma inconsistência.

Temos então um objeto que pode ser escrito e guardado de maneira 'Serializada'. Precisamos agora desenvolver um programa que irá criar um

objeto e guardar esse objeto no disco. Esse programa está apresentado na listagem a seguir:

Listagem 3.2 - Classe que armazena uma Pessoa com serialização:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class CriaEArmazenaPessoas {
    public static void escreve(Pessoa p) {
        try {
            FileOutputStream fout =
                new FileOutputStream("Grava.dados");
            ObjectOutputStream oos =
                new ObjectOutputStream(fout);
            oos.writeObject(p);
            oos.close();
            fout.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        System.out.println("Criando Pessoa.");
        Pessoa p1 = new Pessoa();
        p1.setNome("Leandro");
        p1.setTelefone("3xxx-xxxx");
        System.out.println("Escrevendo Pessoa");
        escreve(p1);
        System.out.println("Terminado");
    }
}
```

No método `main()`, inicialmente é impressa a mensagem na tela da mensagem e em seguida é criado um objeto do tipo `Pessoa`. Em seguida os valores dos atributos são inseridos no objeto e por fim é executado o método `escreve()` que recebe como parâmetro o objeto do tipo `Pessoa`.

Será executado então o método `escreve()`, que inicia com um bloco de tratamento de erros obrigatório, pois todo acesso ao disco pode gerar erros durante a execução. Primeiramente é criado um fluxo de acesso ao disco (`FileOutputStream`) para o arquivo chamado `Grava.dados`, que pode ainda não existir (caso o arquivo exista, ao executarmos o programa, iremos sobrescrever os dados que existiam perdendo os dados antigos). Em seguida, cria-se um filtro sobre o fluxo acesso ao disco para objetos serializáveis, chamado `ObjectOutputStream`. Feita toda esta preparação, estamos prontos para poder escrever o objeto para o disco.

O objeto inteiro é enviado para o fluxo de saída de objetos através do método `writeObject()`. Feita a escrita, podemos fechar os fluxos e terminar o método, voltar para o método principal e terminar o programa.

Se listarmos o conteúdo do arquivo 'Grava.dados', podemos ver os dados do objeto armAzenado. Para fazer isso, abra uma janela do console e digite o comando de listagem (no Windows, digite `type` e, no Unix, digite o comando `cat`). Algo como a figura 3.1 será mostrado.

Fig. 3.1: Listagem do arquivo Grava.dados.

Para restaurar, ou desserializar o conteúdo na memória, precisaremos fazer a leitura dos dados novamente, para tanto, veja o código da listagem a seguir.

Listagem 3.3 - Lendo um objeto serializado:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LeERestauraPessoas {
    public static Pessoa le() {
        try {
```

```
            FileInputStream fin =
                new FileInputStream("Grava.dados");
            ObjectInputStream ois =
                new ObjectInputStream(fin);
            Pessoa p = (Pessoa) ois.readObject();
            ois.close();
            fin.close();
            return p;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            return null;
        }
    }

    public static void main(String[] args) {
        System.out.println("Lendo Pessoa.");
        Pessoa objetoLido = le();
        System.out.println("Executado.");
        System.out.println("Nome: "
            + objetoLido.getNome());
        System.out.println("Telefone: "
            + objetoLido.getTelefone());
    }
}
```

Novamente, o programa inicia a execução pelo método `main()`. Apresenta a mensagem na tela e chama o método `le()`. Depois do bloco de tratamento de exceções, é criado um fluxo de leitura do arquivo, apontando para o arquivo de nome `Grava.dados` gerado anteriormente. Em seguida, cria-se um fluxo de leitura de objetos sobre o fluxo de leitura de arquivo, para montar objetos a partir dos dados lidos do arquivo. Por fim, é feita a leitura de um objeto a partir da fonte e fazemos uma conversão (type casting) do objeto lido para o objeto da classe `Pessoa`. O método é finalizado fechando o leitor de objetos e o arquivo lido.

Caso tudo esteja correto, é retornada a referência para o objeto recons-

tituído na leitura. O programa volta para a execução do método principal, colocando o valor da referência do objeto lido na variável `objetoLido`. Nas linhas de código seguinte, os dados são apresentados na tela.

3.3 Nem tudo é perfeito

Ótimo, conseguimos guardar os dados e depois lê-los e utilizá-los novamente. Ficamos então tentados a usar o mecanismo de serialização-desserialização para armazenar dados de nossas aplicações, exportar, etc. Vamos alterar nossa classe `Pessoa` para incluir um novo campo, `sobrenome`, conforme listagem a seguir.

Listagem 3.4 - Adicionando um novo atributo a classe Pessoa:

```
import java.io.Serializable;

public class Pessoa implements Serializable {
    private String nome;
    private String sobrenome;
    private String telefone;

    public Pessoa() {
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getSobrenome() {
        return sobrenome;
    }
    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }
    public String getTelefone() {
        return telefone;
```

```
    }
    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }
}
```

Incluímos à classe já existente, o campo `sobrenome` do tipo `String` e adicionamos mais dois métodos, `getSobrenome()` e `setSobrenome()`. O resto da classe permaneceu sem alteração. Se rodarmos novamente a classe `LeERestauraPessoas`, o resultado será algo como mostrado na figura 3.2.

```
Lendo Pessoa.
java.io.InvalidClassException: Pessoa; local class incompatible: stream classdesc serialVersionUI
        at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at LeERestauraPessoas.le(LeERestauraPessoas.java:10)
        at LeERestauraPessoas.main(LeERestauraPessoas.java:25)
Executado.
Exception in thread "main" java.lang.NullPointerException
        at LeERestauraPessoas.main(LeERestauraPessoas.java:27)
```

Fig. 3.2: Falha na execução de LeERestauraPessoas após alteração da classe serializada.

O programa apresentou erro ao ler os dados armazenados. Isso porque os dados foram guardados segundo uma estrutura seqüencial, ao ler novamente e tentar colocar os dados seqüenciais nos campos correspondentes do objeto, o programa encontra outro objeto de mesmo nome, mas incompatível por ser estruturalmente diferente, não conseguindo assim restaurar os dados originais.

O custo da facilidade e do uso de mecanismos padronizados de persistir os dados de um objeto tem em contrapartida uma menor flexibilidade a mudanças. Um objeto serializado poderá ser lido e escrito sem problemas se a classe originária não foi alterada, caso seja alterada, teremos problemas para lê-lo.

3.4 Persistindo atributos: normais, transientes e estáticos

O comportamento dos atributos são distintos. Nem sempre gostaríamos de gravar todos os dados sob forma serializada, pois algumas vezes não queremos guardar algumas informações no arquivo.

As razões podem ser de diversas naturezas. Por exemplo, pelo motivo dos dados do arquivo serializado poderem ser facilmente lidos com um editor de texto ou binário, temos razões de segurança para não armazenarmos campos de senha e número de cartões de crédito nestes tipos de arquivos. Ou por questões de que os dados no momento que forem recuperados podem não ser válidos. Seria o caso de atributos como o de hora atual, lista de usuários presentes no sistema, cálculos, etc.

Algumas vezes, objetos referenciados em um dado momento podem não existir em um outro momento no futuro, como por exemplo, objetos de dados, uma janela específica de apresentação, um campo de texto, entre outras coisas. Outras vezes, um objeto depende de recursos fora do Java, uma `Thread` irá depender de recursos do sistema operacional; um objeto de acesso a rede irá depender de conexões e outras coisas mais.

Para esses casos, existe o modificador `transient`, que diz que um determinado atributo não deve ser armazenado em disco. Para verificar um exemplo, analise o funcionamento da listagem a seguir.

Listagem 3.5 - Adicionando um novo atributo a classe Pessoa:

```
import java.io.Serializable;

public class Item implements Serializable {
    private static long numeroSequencia;
    private String usuario;
    private long matricula;
    private transient String senha;

    public Item() {
        numeroSequencia = System.currentTimeMillis();
        usuario = "Nenhum";
```

```
        matricula = System.currentTimeMillis();
        senha = "Nenhuma";
    }

    public String toString() {
        return "Item: numeroSequencia=" + numeroSequencia +
               ", usuario=" + usuario +
               ", matricula=" + matricula +
               ", senha=" + senha;
    }
    public static long getNumeroSerial() {
        return numeroSequencia;
    }
    public static void setNumeroSerial(long numeroSerial) {
        Item.numeroSequencia = numeroSerial;
    }
    public long getMatricula() {
        return matricula;
    }
    public void setMatricula(long matricula) {
        this.matricula = matricula;
    }
    public String getSenha() {
        return senha;
    }
    public void setSenha(String senha) {
        this.senha = senha;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
}
```

Observe que os atributos da classe Item possuem características diferentes. O modificador de visiblidade (`private`, `protected` ou `public`) não altera o comportamento da serialização, e nessa classe todos os atributos são

`private`.

Temos o atributo `numeroSequencia`, uma variável de classe (estática) do tipo long. Variáveis estáticas são aquelas que pertencem à classe e terão um único valor para todos os objetos daquela classe, ou seja, se modificada diretamente ou em um objeto, essa mudança será vista por todos os objetos.

Temos o atributo `usuario`, uma variável de objeto (não-estática) da classe `String`. Variáveis de objeto são variáveis de instância, seu conteúdo não é visto pela classe diretamente, nem por outros objetos existentes. Temos a `matricula`, outra variável de objeto, do tipo `long`. Por fim, temos `senha`, uma variável de objeto `String`, no entanto, que esta está sendo modificada pelo marcador `transient`.

Vejamos então como estas variáveis distintas se comportam no processo de serialização. Vamos criar uma classe utilitária que possui um método para gravação de um objeto e outro para recuperação do objeto, conforme a listagem a seguir.

Listagem 3.6 - Classe utilitária para gravação e recuperação de objetos em arquivos:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class FileUtilities {
    public static void escreva(String nomeArq, Item item) {
        try {
            FileOutputStream fout =
                new FileOutputStream(nomeArq);
            ObjectOutputStream oos =
                new ObjectOutputStream(fout);
            oos.writeObject(item);
            oos.close();
            fout.close();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }

    public static Item leia(String nomeArq) {
        Item item = null;
        try {
            FileInputStream fin =
                new FileInputStream(nomeArq);
            ObjectInputStream ois =
                new ObjectInputStream(fin);
            item = (Item) ois.readObject();
            ois.close();
            fin.close();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        return item;
    }
}
```

O método `escreva()` recebe o nome do arquivo para onde o objeto será escrito e um objeto do tipo `Item` para escrever no arquivo. Inicialmente, abrimos um fluxo para este arquivo, e em seguida um fluxo de escrita de objetos sobre o fluxo criado para o arquivo. Por fim, escrevemos o arquivo no fluxo de objetos e fechamos as saídas.

O método `leia()` recebe um nome de arquivo e lê o objeto `Item` armazenado dentro dele. Primieramente, criamos uma variável que recebe o objeto do tipo `Item` lido. Em seguida, cria-se um fluxo de leitura de arquivo apontando para o nome passado como parâmetro, e um fluxo de leitura de objetos sobre o arquivo de leitura. Em sequencia tentamos ler o objeto do arquivo e realizar a conversão do objeto lido para o tipo `Item`. Se a leitura ocorreu corretamente, será retornado o objeto restaurado, e caso ocorra algum problema será retornado o valor nulo (`null`).

Temos nossa classe de dados e a classe que irá nos ajudar a escrever e ler objetos. Falta agora fazer as classes de aplicação para podermos fazer a

execução, gravação e leitura dos objetos. Vamos começar pela gravação, veja a classe na listagem a seguir.

Listagem 3.7 - Classe que cria um item e o grava em um arquivo:

```
public class CriaItem {
    public static void main(String[] args) {
        System.out.println("Criando Objeto");
        Item item = new Item();
        System.out.println(item);
        System.out.println("Alterando Valores");
        item.setMatricula(123456);
        item.setSenha("Nova Senha");
        item.setUsuario("Fulano");
        item.setNumeroSerial(67890);
        System.out.println(item);
        System.out.println("Gravando Objeto");
        FileUtilities.escreva("itemArq", item);
        System.out.println("Finalizado");
    }
}
```

Temos na classe o método `main()` por onde iremos iniciar a execução do código. O programa cria um objeto novo do tipo Item e é impresso no console o objeto criado com os valores inicializados pelo construtor. Em seguida alteramos os valores das variáveis e imprimimos novamnete no console. Por fim, realizamos a gravação do objeto para o arquivo chamado `itemArq` e, para isso utilizamos o método `escreva()` que criamos anteriormente na classe `FileUtilities`.

A execução da classe `CriaItem` gera a seguinte saída no console:

```
Criando Objeto
Item: numeroSequencia=1415361296961, usuario=Nenhum,
    matricula=1415361296961, senha=Nenhuma
Alterando Valores
Item: numeroSequencia=67890, usuario=Fulano, matricula=123456,
    senha=Nova Senha
Gravando Objeto
Finalizado
```

Agora, precisamos recuperar os dados gravados. Será criada a classe `RecuperaItem` conforme listagem a seguir.

Listagem 3.8 - Classe qeu lê um item do arquivo e imprime seus dados no console:

```
public class RecuperaItem {
    public static void main(String[] args) {
        System.out.println("Recuperando Objeto");
        Item item = FileUtilities.leia("itemArq");
        System.out.println("Apresentando Valores");
        System.out.println(item);
    }
}
```

Também temos na classe `RecuperaItem` o método `main()` por onde iremos iniciar a execução do código. O programa declara uma referência do tipo Item e utiliza o método `leia()` da classe `FileUtilities` para obter um objeto deste tipo. Esta classe irá se encarregar de realizar a leitura do arquivo `itemArq`, passado como parâmetro, e de retornar um objeto reconstituído a partir dos dados lidos. A seguir imprimimos no console os dados do objeto restaurado.

Na execução da classe RecuperaItem, teremos algo como apresentado a seguir:

```
Recuperando Objeto
Apresentando Valores
Item: numeroSequencia=0, usuario=Fulano, matricula=123456,
    senha=null
```

Então, como podemos ver, os valores de atributos estáticos e de atributos marcados com `transient` não são serializados e ao recuperar o objeto serializado eles assumem valores-padrão (null para objetos e o para números).

3.5 Escrevendo XML

Uma funcionalidade interessante acrescentada no Java 2 SE 1.4 foi a possibilidade de serializar objetos em formato XML. Para fazer isso, uti-

lizamos as classes `XMLEncoder` e `XMLDecoder`. Vamos utilizar nossa classe Pessoa criada no começo deste artigo e gravá-la em formato XML. O código da classe `EscrevePessoaXML` está apresentado na listagem a seguir.

Listagem 3.9 - Exemplo de serialização em XML:

```
import java.beans.XMLEncoder;
import java.io.FileOutputStream;
import java.io.IOException;

public class EscrevePessoaXML {
    public static void main(String[] args) {
        Pessoa pessoa = new Pessoa();
        pessoa.setNome("Leandro");
        pessoa.setSobrenome("Yung");
        pessoa.setTelefone("3743-xxxx");

        System.out.println("Gravando como XML");
        try {
            FileOutputStream fout =
                new FileOutputStream("Pessoa.xml");
            XMLEncoder xe = new XMLEncoder(fout);
            xe.writeObject(pessoa);
            xe.close();
            fout.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Ok");
    }
}
```

Inicialmente, criamos um objeto `Pessoa` que iremos serializar e abrimos um fluxo de saída para arquivo apontando para o arquivo `Pessoa.xml`. Em seguida, criamos um fluxo de saída XML sobre o fluxo de saída para arquivo. Por fim, escrevemos para o fluxo de saída XML o objeto pessoa criado.

Abrindo o arquivo XML gerado, ele deverá possuir a seguinte estrutura:

Listagem 3.10 - XML gerado pela serialização:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0" class="java.beans.XMLDecoder">
 <object class="Pessoa">
  <void property="nome">
   <string>Leandro</string>
  </void>
  <void property="sobrenome">
   <string>Yung</string>
  </void>
  <void property="telefone">
   <string>3743-xxxx</string>
  </void>
 </object>
</java>
```

3.6 Considerações Finais

Os fundamentos de serialização são bastante simples, porém são importantes para tecnologias mais avançadas. O processo é muito importante para converter uma estrutura em memória em dados facilmente armazenáveis e transportáveis, por esse motivo, vastamente utilizado em mecanismos de persistência, armazenamento e transporte.

Para o leitor, os próximos passos seriam estudar outros aspectos da serialização, tais como o número serial, personalização do processo de leitura e gravação e o uso para envio de dados no contexto da programação distribuída.

3.7 Referências

- The Java Programming Language, Third Edition - KeN Arnold / James Gosling / David Holmes
- Java Tutorial – Lesson I/O: http://docs.oracle.com/javase/tutorial/essential/io/index.html