Tutorial

Gerência de memória em Java

Edição 1.0 (outubro 2005)

Helder da Rocha
(helder.darocha@gmail.com)

© 2005, *Helder da Rocha*. Os direitos autorais sobre esta obra estão protegidos pela Lei 9.610/98 (*Lei de Direitos Autorais*). Este tutorial pode ser usado para estudo pessoal. O uso como material de treinamentos, cursos e a reprodução para outros fins requer autorização do autor.

Sobre o autor

Helder da Rocha nasceu em Campina Grande, Paraíba, em 1968, viveu o final da infância em Waterloo, Canadá (1975-1980) e reside em São Paulo desde 1995. É instrutor e consultor em tecnologia da informação. Fundador da *Argo Navis* – empresa de treinamento e consultoria especializada em sistemas abertos – realiza pesquisas em *Java* desde 1995. Ocasionalmente escreve para revistas especializadas e está sempre presente como palestrante nos principais eventos nacionais sobre Java. É autor de mais de 20 cursos sobre *Java*, *XML* e tecnologias *Internet*.

Em 1996, começou a escrever um livro sobre Java e nunca terminou, mas acredita que irá conseguir a tempo para o *Java 6*, em 2006.

Além do mundo da informática, o autor também explora diversas outras áreas do conhecimento, como a ecologia, a literatura, a astronomia, a música e o teatro, nem sempre como hobby. Entre suas atividades alternativas está a tradução para o português da *Divina Comédia* de Dante e do poema *O Corvo* de Edgar Allan Poe, adaptado para o teatro. Tem feito também adaptações para teatro de outros autores, pesquisa teatral, dramaturgia e cenografia além de apresentar-se como ator e músico em peças de teatro amador em São Paulo. Quando não está viajando a trabalho (ou nos intervalos) aproveita para acampar nas matas e pantanais do planeta, observar o Universo e vez ou outra arriscar uma foto dos planetas ou da Lua.

Para entrar em contato com o autor, utilize os e-mails ou sites abaixo:

- *helder.darocha@gmail.com* (e-mail)
- *www.argonavis.com.br* (empresa)
- *www.helderdarocha.com.br* (site pessoal e blog)

Sobre a Argo Navis

A *Argo Navis* tem como objetivo explorar, assimilar e divulgar novas idéias do mundo das tecnologias abertas de informática, mapeando o território e indicando os melhores caminhos. Sua missão é difundir a informação e removendo barreiras tecnológicas, culturais e econômicas. Essa ação é realizada através da pesquisa e desenvolvimento de palestras, cursos, artigos, tutoriais, livros e exemplos didáticos em português que tem seu material distribuído gratuitamente e são financiados por atividades de treinamento, consultoria, mentoring, venda de livros e parcerias de treinamento.

A *Argo Navis* realiza treinamentos personalizados em tópicos básicos e avançados de *Java*, *XML* e tecnologias *Web* (*HTML*, *JavaScript*, *CSS* e assuntos relacionados) para grupos ou empresas em qualquer parte do Brasil ou do mundo. Ministramos treinamentos em inglês e português e desenvolvemos material de treinamento sob demanda nos dois idiomas. Para maiores informações e download gratuito de todo o material de treinamento usado nos cursos públicos, visite o site *www.argonavis.com.br*.

R672g Rocha, Helder Lima Santos da, 1968-

Tutorial: gerência de memória em Java. Edição em PDF. Primeira edição concluída em 31 de outubro de 2005. Formato A4. *NÃO REVISADA.*

1. Java (*Linguagem de programação de computadores*) – Gerência de memória. 2. Gerência de memória virtual em computadores (*Administração de sistemas*). 3. Algoritmos de coleta de lixo (Engenharia de Software). I. Título.

CDD 005.133

Índice

Tutorial

Gerência de Memória em Java

Helder da Rocha

ESTE TUTORIAL explora detalhes sobre o uso de memória virtual em aplicações Java. Está dividido em três partes.

A primeira parte explora os detalhes do funcionamento da máquina virtual em relação à execução e à gerência de memória, os tipos de algoritmos usados para coleta de lixo, as diferentes regiões da memória onde eles atuam. São detalhados aspectos da arquitetura da máquina virtual de acordo com a especificação, e não com alguma implementação específica (como a *Sun HotSpot*.) Existem vários diferentes tipos de coletores de lixo e diversas estratégias de coleta que combinam algoritmos. Este *tutorial* concentra-se nos mais importantes. Nem todos são implementados nas máquinas virtuais Java mais populares, porém como as técnicas usadas mudam a cada nova versão dos ambientes de execução Java, vale a pena conhecê-las já que poderão ser opções em máquinas virtuais no futuro.

A segunda parte do *tutorial* aborda a arquitetura de memória das máquinas virtuais *HotSpot*. Essas máquinas virtuais são distribuídas com os ambientes de execução da *Sun* e vários outros fabricantes. Suportam configuração de vários recursos, entre eles estratégias de alocação e coleta de lixo. São discutidos os efeitos da alteração de parâmetros e como configurá-los para obter os melhores resultados, ajustando a organização da memória e algoritmos de coleta de lixo. Para otimizar é preciso medir, e para isto existem várias ferramentas de monitoração e mineração de dados distribuídas como parte dos ambientes de desenvolvimento Java. Será mostrado como usar os dados obtidos com as ferramentas *JConsole*, *Jstat* e *GC Viewer*.

A terceira e última parte discute detalhes sobre a criação e destruição de objetos em Java. Diferentemente das outras duas seções, esta mostra o que o programador pode fazer a respeito da gerência de memória em Java. Algumas das questões abordadas são: como funciona a criação, finalização e remoção de objetos; o que são *memory leaks*, como identificá-los e consertá-los; como controlar eficientemente o coletor de lixo usando objetos de referência; e como construir aplicações robustas que lidam eficientemente com a memória alocada pela máquina virtual.

O leitor deste tutorial deve ser um programador, não necessariamente experiente em Java. A terceira parte requer conhecimentos básicos de Java.

© 2005 Helder da Rocha

Parte I - Gerencia de memória? Em Java?

Por que se preocupar com memória em Java? Diferentemente de C ou C++, programadores Java não têm a responsabilidade e nem a possibilidade de gerenciar a memória do sistema explicitamente. Em Java, é possível desenvolver aplicações programando em *alto nível* sem se preocupar com questões como alocação e liberação de memória, que são realizadas automaticamente pela máquina virtual usando algoritmos. Um programador Java pode desenvolver aplicações preocupando-se apenas com a lógica do programa. Então para que discutir esses detalhes, já que um esforço tão grande foi realizado exatamente para que não fosse necessário discuti-los?

Como tudo o que é feito automaticamente, as soluções foram construídas para os casos mais comuns, mais genéricos. Todas as máquinas virtuais modernas buscam adaptar-se o melhor possível ao ambiente onde suas aplicações irão executar, porém e se seu ambiente for atípico? Se sua aplicação for gigante, usar muita memória, mantiver muitos objetos vivos, ou realizar alguma computação incomum, a configuração default da máquina virtual pode revelar-se inadequada. A maior parte dos algoritmos de coleta de lixo, por exemplo, são otimizados para situações típicas, onde as aplicações ou têm uma vida curta ou contam com uma distribuição previsível de objetos duradouros. O mesmo ocorre com a distribuição de memória que influencia a alocação eficiente. A maior parte das aplicações irão funcionar satisfatoriamente nas máquinas virtuais mais populares sem requerer nenhum ajuste sequer, porém outras aplicações podem se beneficiar de ajustes para melhorar sua performance, escalabilidade, segurança, consumo de memória, etc.

Fazer ajustes não é uma atividade trivial. Muitos ajustes têm efeito colateral. Ajustes no coletor de lixo, por exemplo, geralmente comprometem a eficiência da aplicação na tentativa de reduzir pausas. Quando se busca a maior eficiência, geralmente ganha-se pausas mais longas. Além disso, os ajustes que uma máquina virtual oferece são disponibilizados através de opções que não são padronizadas. Podem mudar de uma versão para outra. Os algoritmos de gerência de memória mudam de uma versão para outra. Assim, cada vez mais é importante que o administrador do sistema tenha conhecimentos sobre a arquitetura das máquinas virtuais e algoritmos de coleta de lixo. Saber o quanto, quando, onde ajustar requer conhecimentos elementares da organização da memória, dos algoritmos de alocação e coleta de lixo empregados pela implementação da JVM[1] usada.

O objetivo desta seção é cobrir os principais tópicos de arquitetura da máquina virtual Java que afetam a performance da gerência automática de memória. A abordagem nesta seção será mais "acadêmica", sem levar em conta nenhuma implementação específica.

---

[1] *JVM = Java Virtual Machine*: máquina virtual Java. É a máquina onde executa qualquer aplicação Java.

1. Anatomia da JVM

A máquina virtual Java (JVM) é uma máquina imaginária implementada como uma aplicação de software [JVMS][2]. Ela executa um código de máquina portável (chamado de Java *bytecode*) armazenado em um formato de arquivo chamado de *class file format* (formato de arquivo *class*). Um arquivo em formato *class* geralmente[3] é gerado como resultado de uma compilação de código-fonte Java, como mostrado na *figura 1*.

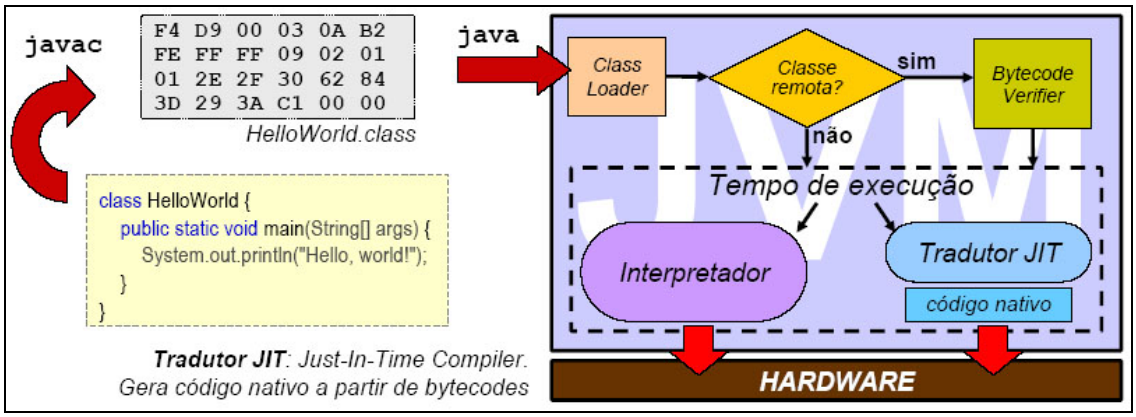

Figura 1 – Processo de construção de aplicações em Java: código-fonte em Java é compilado em linguagem de máquina virtual (arquivo .class) que é lido pelo ambiente de execução (máquina virtual).

Uma das decisões de *design* da plataforma Java foi a de esconder do programador detalhes da memória. A *especificação* da máquina virtual (*Java Virtual Machine Specification* [JVMS]) não especifica detalhes de segmentação da memória (como ocorre o uso de memória virtual, onde fica a pilha, o heap, etc.), o algoritmo de coleta de lixo usado para liberar memória (diz apenas que deve haver um), nem vários outros aspectos de baixo nível como formato de tipos, etc. Diferentes *implementações* da JVM têm a liberdade de organizar a memória diferentemente e escolher algoritmos de coleta de lixo diferentes. Exemplos de implementações de máquinas virtuais Java são:

- *Sun HotSpot JVM*: é a mais popular em *desktops* e servidores; a máquina virtual da IBM é similar, porém usa outras opções de configuração. Há máquinas de outros fabricantes (*Oracle*, *Borland*, etc.) embora compatíveis com a *HotSpot* podem não ter os mesmos comandos de configuração.
- *Sun KVM*: ou máquina virtual K. É usada em dispositivos como *palmtops* e celulares para executar aplicações J2ME.
- *Jikes RVM*: é uma máquina virtual experimental, construída a partir de um projeto da IBM e hoje é um projeto de código aberto. É a máquina virtual mais popular entre cientistas. A maior parte dos artigos científicos sobre coletores de lixo usam como *benchmark* a *Jikes RVM*, mesmo os que têm como alvo outras plataformas, como .NET.

---

[2] As referências entre colchetes estão relacionadas na última seção deste *tutorial*.

[3] É possível gerar bytecode Java a partir de outras linguagens diferentes de Java, apesar de não ser comum nem ser oficialmente suportado (como ocorre com .NET).

A pilha, o heap e a JVM

Existem linguagens em que a alocação de memória é trivial, e não requer gerenciamento complexo. As principais estratégias são:

- Alocação *estática*: áreas de memória são alocadas antes do início do programa; não permite mudanças nas estruturas de dados em tempo de execução (ex: *Fortran*)
- Alocação *linear*: memória alocada em fila ou em pilha; não permite remoção de objetos fora da ordem de criação (ex: *Forth*)
- Alocação *dinâmica*: permite liberdade de criação e remoção em ordem arbitrária; requer gerência complexa do espaço ocupado e identificação dos espaços livres (ex: Java, C++)

Java utiliza alocação dinâmica (*heap*) para objetos e alocação linear (pilha) para procedimentos seqüenciais, mas todo o gerenciamento é feito automaticamente.

A *figura 2* ilustra um diagrama lógico de segmentação de memória virtual. A representação é apenas um modelo genérico e não representa nenhuma implementação real, porém é útil para ilustrar os diferentes papéis assumidos pela memória em linguagens que usam alocação dinâmica e linear. Os blocos no *heap* indicam memória alocada dinamicamente. O espaço entre os blocos ilustra a fragmentação, que é um problema que pode ocorrer em alocação dinâmica. Se o modelo representar uma máquina virtual Java, os blocos na pilha podem representar *frames* (seqüências de instruções de cada método) de *um único thread.* As setas da pilha para o *heap* e de blocos do *heap* para outros blocos do *heap* são ponteiros.

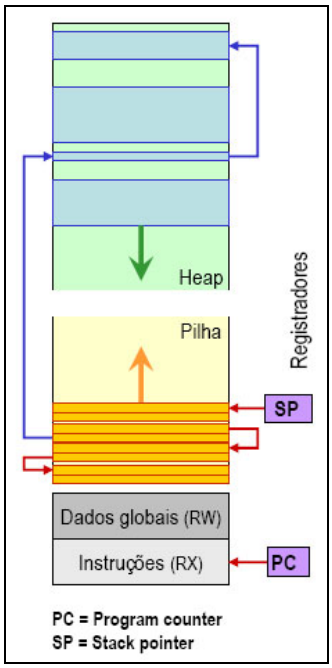

Figura 2 - Esquema lógico de baixo nível. Este diagrama é apenas um modelo genérico (inspirado em modelos de segmentação de memória C++) e não reflete nenhuma implementação real.

Do ponto de vista de um programador Java, as áreas de memória virtual conhecidas como a pilha e o *heap* são lugares imaginários na memória de um computador. Não interessa ao programador nem adianta ele saber onde estão nem os detalhes de como são organizados, uma vez que Java não oferece opções de escolha para alocação no *heap* ou na pilha como ocorre em C ou C++. Além disso, a especificação da máquina virtual garante liberdade ao implementador de máquinas virtuais Java para organizar a memória como bem entender. O que interessa ao programador Java é onde as alocações são feitas: em Java, tipos primitivos ficam sempre na pilha e objetos ficam sempre no *heap*.

Implementações da especificação da JVM, (como a *HotSpot JVM*), oferecem parâmetros que permitem algum controle sobre a gerência de memória virtual.

Conhecer as escolhas de algoritmos e arquitetura da máquina virtual usada é importante para saber como configurá-la e ter uma base para saber quais parâmetros ajustar para obter melhor performance. Ainda assim, o controle é muito limitado, voltado principalmente para administradores e muito pouco pode ser feito por programadores. Portanto não existe, em Java, a disciplina "gerência de memória" da forma como existe em C ou C++. Mas há estruturas e escolhas que um programador pode fazer usando a linguagem que influenciam o coletor de lixo e a alocação de memória. Esses recursos serão vistos na terceira parte deste tutorial.

Anatomia da JVM: áreas de dados

A máquina virtual define várias áreas de dados que podem ser usadas durante a execução de um programa.

- Registradores
- Pilhas e segmentos de pilha (*quadros*)
- Heaps e área de métodos

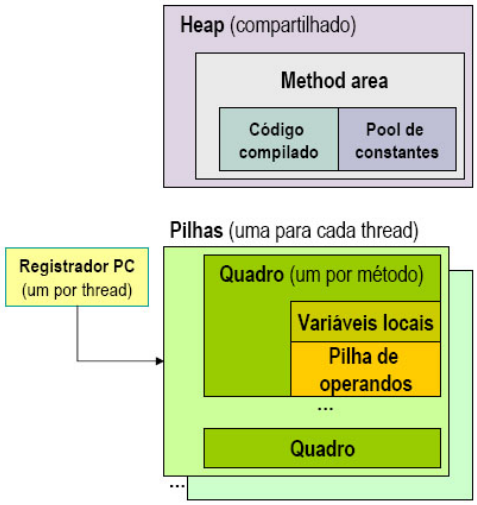

Figura 3 – Áreas de dados usadas pela máquina virtual Java.

Existem áreas de dados que são compartilhadas por operações que executam em paralelo e outras que são privativas. As áreas de dados privativas estão associadas a *threads* e são criadas (alocadas) quando um *thread* novo é criado, sendo destruídas (liberadas) quando o *thread* termina. As áreas ligadas à máquina virtual são compartilhadas entre os *threads* ativos e são criadas quando a JVM é iniciada e destruídas quando a JVM termina.

O registrador PC

Cada *thread* de execução tem um *registrador PC* (*program counter*), que mantém controle sobre as instruções da máquina virtual que estão sendo executadas. Em qualquer momento, cada *thread* estará executando o código de um único método. Um método (em código *bytecode*) consiste de uma lista de instruções executadas em uma seqüência definida. O registrador PC contém o endereço da instrução da JVM que está sendo executada. O valor do registrador PC só não é definido se o método for um método *nativo*, que é um método implementado em linguagem de máquina da plataforma onde roda.

Pilhas

Cada *thread* é criado com *uma* pilha associada que é usada para guardar variáveis locais e resultados parciais. A memória usada pela pilha *pode* ser alocada no *heap*, não precisa ser contígua e é liberada automaticamente depois de usada. Uma ilustração esquemática da pilha é mostrada na *figura 4*. A pilha pode ter um

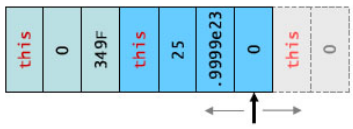

Figura 4 - A pilha de um thread.

tamanho fixo ou expandir-se e contrair-se na medida em que for necessário. As implementações de máquinas virtuais Java podem oferecer controles para ajustar tamanho de pilhas.

Quando a memória acaba em uma operação relacionada à pilha, dois erros podem ocorrer:

- *StackOverflowError* ocorre se a computação de um *thread* precisar de uma pilha maior que a permitida. Métodos que criam muitas variáveis locais ou funções recursivas são a principal fonte causadora desse tipo de erro.
- *OutOfMemoryError* ocorre se não houver memória suficiente para expandir uma pilha que pode crescer dinamicamente. Este erro também pode ocorrer em aplicações com muito *threads* que criam muitas pilhas a ponto de esgotar a memória necessária para alocar o espaço mínimo determinado para a pilha. A solução pode ser diminuir o número do *threads* ou o tamanho inicial (ou fixo) da pilha de cada *thread*.

Quadros de pilha (frames)

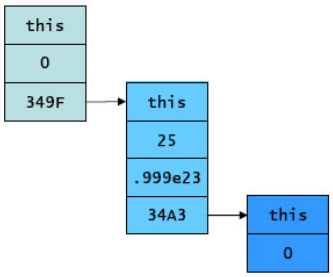

Figura 5 – Um quadro contém a execução de um método.

Um quadro (*frame*) é um segmento alocado a partir da pilha de um *thread*. Um quadro é criado cada vez que um método é chamado e destruído quando a chamada termina (normalmente ou através de exceção). Todo método tem um quadro associado e ele é sempre local ao *thread*, não podendo ser compartilhado com outros *threads*. É usado para guardar resultados parciais, dados temporários, realizar ligação dinâmica, retornar valores de métodos e despachar exceções.

Em um determinado *thread*, apenas um quadro está ativo em um determinado momento: o *quadro corrente*: seu método é chamado de *método corrente* e sua classe é chamada de *classe corrente*. Cada quadro possui um *array* de variáveis locais, uma pilha de operandos e uma referência ao *pool* de constantes de tempo de execução da classe corrente.

Chamadas de métodos continuamente criam e destroem quadros durante a execução de operações. A *figura 6* ilustra esse comportamento. Quando o método corrente *m1*, associado ao quadro *q1*, chama outro método *m2*, um novo quadro *q2* é criado, que passa a ser o quadro corrente. Quando o método m2 retornar, o quadro *q2* retorna o resultado da sua chamada (se houver) ao quadro *q1*. O quadro *q2* é descartado e *q1* volta a ser o quadro corrente.

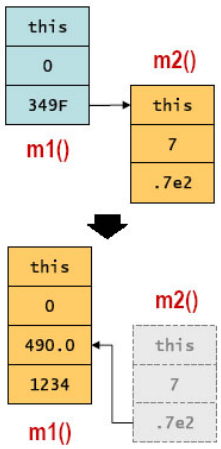

Figura 6 – Criação e destruição de quadros nas chamadas de métodos

Cada quadro possui um vetor de variáveis contendo as *variáveis locais* do seu método associado. Variáveis de até 32 bits ocupam um lugar no *array*. Variáveis de 64 bits

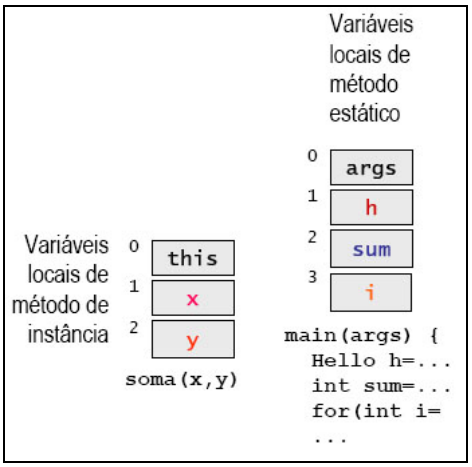

Figura 7 – Arrays de variáveis locais para dois métodos: soma(), método de instância, e main(), método estático.

ocupam dois lugares consecutivos. São usadas para passar parâmetros durante a chamada de métodos.

Na linguagem de máquina Java, as variáveis locais são acessadas pelo seu índice (a partir de 0). Em métodos estáticos, a variável local de índice 0 é o primeiro parâmetro passado ao método, mas em métodos de instância, os parâmetros são passados a partir da variável local de índice 1. Isto ocorre porque a variável local de índice 0 sempre contém o valor do ponteiro para o objeto que o contém. Este ponteiro é a referência que em Java chama-se *this*.

A *figura 7* mostra um diagrama lógico do *array* de variáveis locais e alguns métodos associados.

Cada quadro contém uma pilha LIFO conhecida como *pilha de operandos*. Quando o quadro é criado, a pilha é vazia. Durante a execução do programa, instruções da máquina virtual carregam constantes ou valores de variáveis locais ou campos de dados para a pilha de operandos, e vice-versa.

Figura 8 – Pilha de operandos

A pilha de operandos também serve para preparar parâmetros a serem passados a métodos e para receber seus resultados. Qualquer tipo primitivo pode ser armazenado e lido da pilha de operandos. Tipos *long* e *double* ocupam duas unidades da pilha. Operações sobre a pilha de operandos respeitam os tipos dos dados guardados.

O heap

O *heap* é a área de dados onde todas as instâncias e vetores são alocados. É compartilhada por todos os *threads*.

O *heap* é criado quando a máquina virtual é iniciada. Não precisa ser uma área contígua. O seu espaço ocupado por objetos é reciclado por um sistema automático de gerenciamento de memória – o coletor de lixo – cujo algoritmo depende da implementação da JVM.

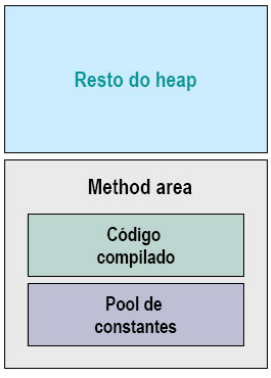

Figura 9 – Áreas do heap.

O *heap* pode ter tamanho fixo ou ser expandido e contraído automaticamente. Diferentes implementações da máquina virtual podem oferecer controles para ajustar tamanho inicial, mínimo, máximo ou fixo do *heap*. Se um programa precisar de mais *heap* que o que foi disponibilizado, a JVM causará *OutOfMemoryError*.

A *área de métodos* (*figura 9*) é a parte do *heap* usada para guardar código compilado de métodos e construtores. É criada quando a máquina virtual inicia

e geralmente armazenada em uma área de alocação permanente (a especificação não determina a localização [4]). Assim como as outras áreas do *heap*, é compartilhada por todos os *threads*. Guarda estruturas que são compartilhadas por todos os métodos de uma classe como: *pool* de constantes de *runtime* (constantes de diversos tipos usados pelo método) e dados usados em campos e métodos. *OutOfMemoryError* pode também ocorrer se em algum momento não houver mais espaço para armazenar o código de métodos.

A ferramenta *javap* permite visualizar o conteúdo de um arquivo de classe. Para obter informações sobre a estrutura de uma classe e instruções da JVM usadas use

```
javap -c nome.da.Classe
```

A sintaxe é

```
javap [-opções] nome.da.Classe
```

Usando opções `-c` e `-verbose` é possível ver a seqüência de instruções da JVM, o tamanho dos quadros de cada método, o conteúdo dos quadros, o *pool* de constantes, etc. A opção `-l` imprime tabela de variáveis locais. Se não for passada nenhuma opção, será mostrada a interface da classe.

O diagrama da *figura 10* ilustra relacionamentos entre o código Java e o código de arquivos *.class* (*bytecode*). As instruções de *bytecode* e outras informações foram obtidos através da ferramenta *javap*.

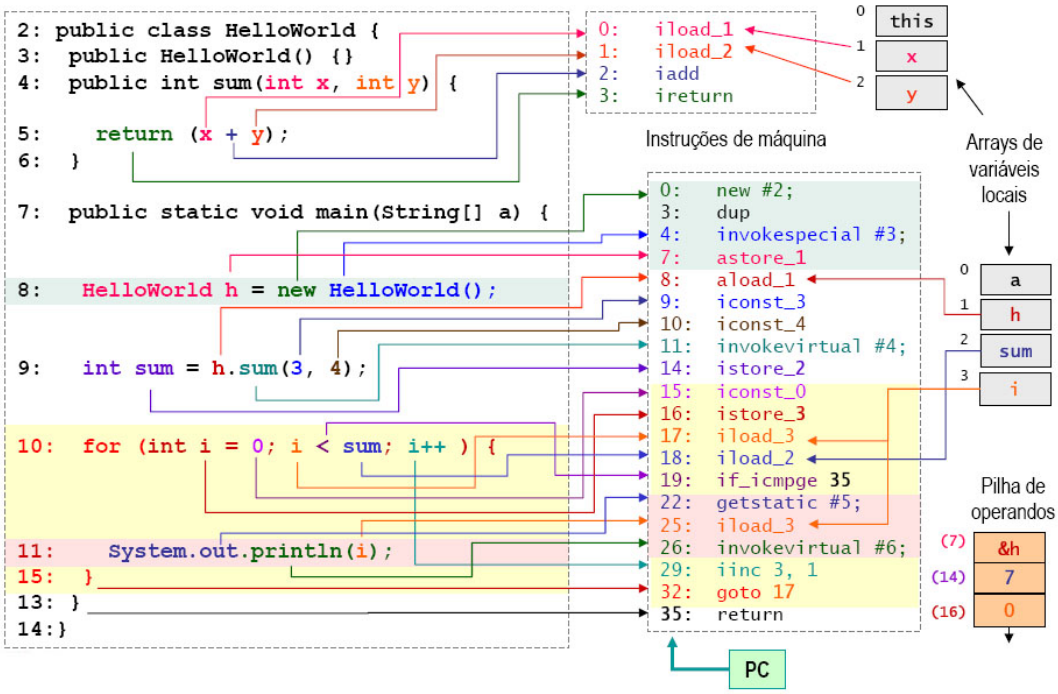

Figura 10 – Diagrama mostrando o relacionamento entre código Java e linguagem bytecode *(código de máquina Java) representado por instruções da máquina virtual.*

[4] A máquina virtual *HotSpot* guarda a área de métodos em uma região do *heap* chamada de *geração permanente*.

2. Algoritmos de coleta de lixo

Em linguagens que usam alocação dinâmica e alocação linear (*heaps* e pilhas), dados armazenados na pilha são automaticamente liberados sempre que a pilha esvazia para ser reutilizada, mas dados armazenados no *heap* precisam ser reciclados através de liberação. A liberação pode ser explícita (*manual*) em linguagens como C e C++ (usando recursos como *freelists* e comandos como *delete*, *free* ou similares), ou implícita (automática), como em Java e maior parte das linguagens dinâmicas[5].

A liberação automática de memória do *heap* é realizada através de algoritmos de coleta de lixo. Há várias estratégias, com vantagens e desvantagens de acordo com a taxa em que objetos são criados e descartados. A coleta de lixo automática têm um considerável impacto na performance, porém a gerência explícita de memória também tem, e é muito mais complicada.

De acordo com a especificação da linguagem Java, a máquina virtual precisa incluir um algoritmo de coleta de lixo para reciclar memória dinâmica do *heap* não utilizada. O principal desafio da coleta de lixo é distinguir o que é lixo do que não é lixo. Na engenharia de software, pode-se classificar os coletores em duas categorias quanto à decisão do que é ou não é lixo: *exatos* e *conservadores*. Algoritmos exatos garantem a identificação precisa de todos os ponteiros e a coleta de todo o lixo. Algoritmos conservadores fazem suposições e podem deixar de coletar lixo que pode não ser lixo (permitindo um possível *memory leak*). Todos os coletores usados nas máquinas virtuais Java são exatos. O critério para definir o que é lixo – o que é memória não utilizada – é o de *alcançabilidade*. Lixo são os objetos inalcançáveis. A especificação não informa *qual* algoritmo deve ser usado – apenas que deve existir um.

A máquina virtual *HotSpot*, da *Sun* – a mais usada em aplicações de servidor e desktops –, usa vários algoritmos e permite diversos níveis de ajuste. O comportamento desses algoritmos é o principal gargalo na maior parte das aplicações de vida longa. Algo em torno de 2% a 20% do tempo de execução de uma aplicação típica é gasto com coleta de lixo. O peso do coletor de lixo é mais evidente em aplicações de vida longa, como servidores. Idealmente, deve manter-se sempre abaixo de 5%, mas existem aplicações onde pode chegar a 40%. Muitas vezes uma solução não é possível sem investimentos de hardware, mas dependendo da natureza da aplicação, uma grande parte da sua ineficiência pode vir da melhor escolha e configuração dos algoritmos de coleta de lixo usados. Conhecer os detalhes do funcionamento desses algoritmos é importante para saber como melhor ajustá-los para obter a melhor performance de um sistema.

A liberação de memória pode influenciar a alocação e degradar a performance depois de várias coletas. Isto é um efeito colateral do algoritmo

[5] *Perl, Python, Rubi, Basic, LISP, Algol, Dylan, Prolog, PostScript, Scheme, Simula, Smalltalk, ML* e *Modula-3* (na maior parte usa coleta de lixo, mas suporta controle manual em alguns módulos).

usado. Como objetos podem ser criados e removidos a qualquer momento e de qualquer lugar do *heap*, a remoção de objetos deixa buracos. Como o coletor de lixo remove muitos objetos de uma vez, ele pode causar fragmentação no *heap* tornando o sistema ficar mais lento com o passar do tempo. Para alocar novos objetos em um *heap* fragmentado, o algoritmo usado precisará procurar nas listas de espaços vazios (*free lists*) um espaço que caiba o próximo objeto. A alocação será mais demorada e mais complexa e o uso do espaço será ineficiente, pois os espaços não usados são desperdiçados. Existem algoritmos que compactam o *heap* depois de realizar a coleta, movendo os objetos para o início do espaço e atualizando os ponteiros. Algoritmos desse tipo tornam a alocação mais simples e eficiente, porém são mais complexos e podem demorar mais.

Algoritmos para coleta de lixo

Existem duas estratégias gerais para coleta de lixo: a *contagem de referências*, que descobre o lixo analisando os objetos ativos, e o *rastreamento de memória*, que varre o *heap* inteiro à procura de objetos inalcançáveis. Existem muitas variações nas técnicas e centenas de algoritmos diferentes que podem ser classificados nessas duas categorias. Os principais algoritmos são:

- *Reference counting algorithm* [Collins 1960]: mantém, em cada objeto, uma contagem das referências que chegam nele. Objetos que têm contagem zero são coletados.
- *Cycle collecting algorithm* [Bobrow 1980]: extensão do algoritmo de contagem de referência para coletar ciclos (referências circulares).
- *Mark and sweep algorithm* [McCarthy 1960]: rastreia objetos do *heap*, marca o que não é lixo e depois varre o lixo (libera memória).
- *Mark and compact algorithm* [Edwards]: extensão do algoritmo *Mark and sweep* que mantém o *heap* desfragmentado depois de cada coleta.
- *Copying algorithm* [Cheney 1970]: divide o *heap* em duas partes. Cria objetos em uma parte do *heap* e deixa outra parte vazia. Recolhe o que não é lixo e copia para a área limpa, depois esvazia a área suja.

Os três últimos são algoritmos de rastreamento.

Diversas *estratégias* usam ou baseiam-se em um ou mais dos algoritmos citados para obter melhores resultados em um dado cenário. Vale a pena destacar duas tendências de classificação: quando à organização de memória e *idades* dos objetos, e quando ao nível de *paralelismo*.

Algoritmos que organizam áreas de memória diferentes para classificar objetos de acordo com a sua idade são chamados de algoritmos baseados em gerações ou em idade. As principais estratégias são: *generational garbage collection*, onde objetos são transferidos para áreas de memória diferentes conforme sobrevivem a várias coletas de lixo, e *age-oriented garbage collection*, onde algoritmos diferentes são usados conforme a idade dos objetos. As duas são muito semelhantes (*age-oriented GC* pode ser considerada um tipo de *generational GC*), mas organizam a memória diferentemente.

Quanto ao nível de paralelismo há os coletores *seriais*, que executam em série parando a aplicação para executar em um único *thread* da CPU; *incrementais* (também chamados de *on-the-fly*), que rodam em *threads* de baixa prioridade concorrendo com a aplicação sem interrompê-la; e *concorrentes*, que executam em vários *threads* em paralelo, mas não necessariamente eliminam totalmente as pausas.

A escolha de um coletor depende das características de uma aplicação. As principais metas de ajuste são *eficiência* e *pausas*. Eficiência (*throughput*) é a relação entre o tempo em que uma aplicação passa fazendo sua função útil dividido pelo tempo que passa fazendo coleta de lixo. O ideal é que seja a maior possível. As pausas são os momentos em que a aplicação inteira (todos os *threads*) da aplicação param para executar o coletor de lixo e liberar memória. O ideal é que as pausas sejam mínimas, ou mesmo zero em sistemas de tempo real que não admitem pausas.

Freqüentemente, essas metas podem ser alcançadas através da escolha de um coletor de lixo adequado, já que diferentes estratégias usam algoritmos diferentes, de formas diferentes, e causam impactos diferentes no sistema.

É possível também configurar parâmetros que modificam o espaço usado, influenciando a forma como um mesmo coletor reage ao ambiente (a maior parte dos coletores reage ao espaço disponível.) Nas seções a seguir explicaremos o funcionamento dos principais algoritmos de coleta de lixo.

Contagem de referências

É o algoritmo mais simples. Cada objeto possui um campo extra que conta quantas referências apontam para ele. O compilador precisa gerar código para atualizar esse campo sempre que uma referência for modificada.

Descrição do algoritmo:

1. Objeto criado em *thread* ativo: *contagem* = 1
2. Objeto ganha nova referência para ele (atribuição ou chamada de método com passagem de referência): *contagem*++.
3. Uma das referências do objeto é perdida (saiu do escopo onde foi definida, ganhou novo valor por atribuição, foi atribuída a *null* ou objeto que a continha foi coletado): *contagem*--.
4. Se *contagem* cair a zero, o objeto é considerado lixo e pode ser coletado a qualquer momento.

As *figuras 11 e 12* ilustram o funcionamento do algoritmo de contagem de referências em várias etapas. Cada seta que chega em um objeto é contada como uma referência para ele (independente de onde tenha vindo). Observe que as referências circulares impedem que contagem caia para zero quando deveria. Essa é uma das principais desvantagens do algoritmo de contagem de referências, e requer tratamento por via de outros algoritmos para que não ocorram *memory leaks*.

O conjunto raiz são as referências iniciais acessíveis através de variáveis locais de métodos em execução, constantes, variáveis globais, etc.

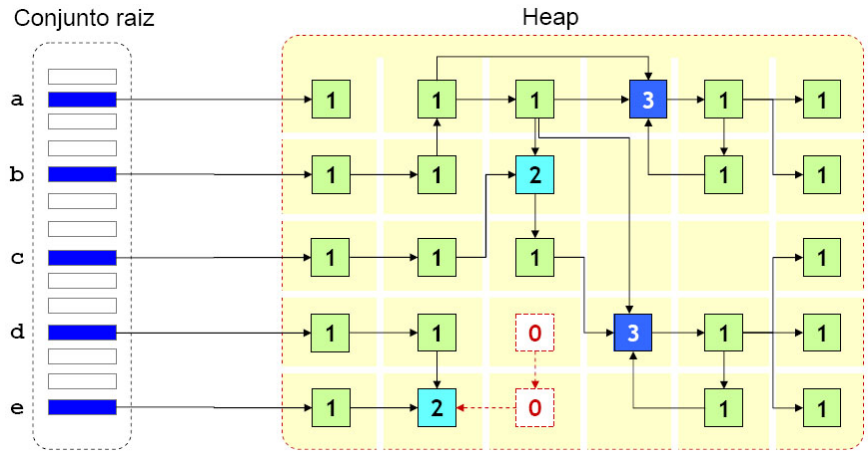

Figura 11 - Cada objeto possui uma contagem de quantas setas chegam nele (referências).

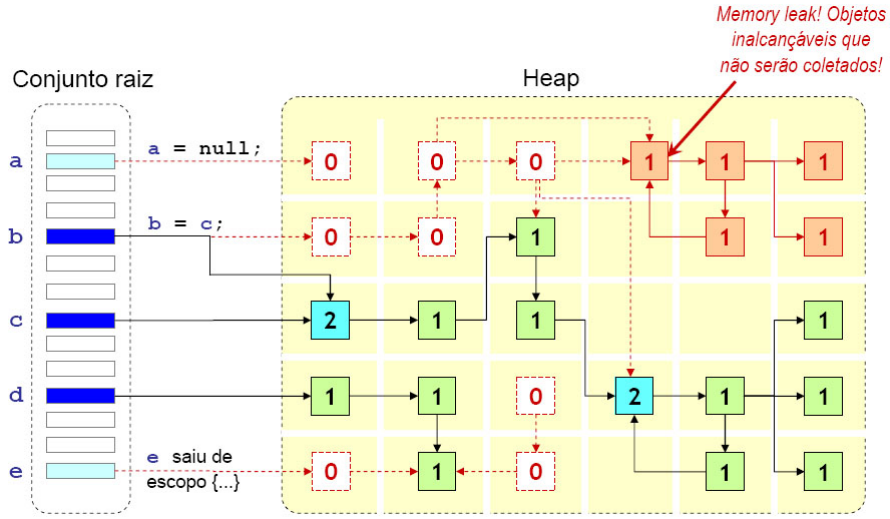

Figura 12 – Quando um objeto perde suas referências, a contagem é alterada, e objetos que têm contagem zero serão coletados. O ciclo não é coletado porque seus objetos ainda recebem referências.

O algoritmo de contagem de referências não precisa varrer o *heap* inteiro. Varre apenas espaço ocupado. Pode executar em paralelo com a aplicação e assim é considerado um algoritmo incremental. Impõe um *overhead* alto já que precisa varrer as referências recursivamente e incrementar um contador. O suporte a paralelismo também implica em custos adicionais para garantir a sincronização. Mas sua principal desvantagem é a incapacidade de recuperar ciclos (objetos que mantém referências circulares entre si). Em implementações de contagem de referências, é comum usar um outro algoritmo (geralmente de rastreamento) como *backup* para limpar os ciclos não coletados.

Apesar de simples, a contagem de referências tem sido pouco usada em coletores de lixo comerciais. As pesquisas têm ressurgido com o aumento do tamanho dos *heaps*, que torna os algoritmos atuais – baseados em rastreamento – menos eficientes. Existem propostas eficientes para coletores incrementais (*on-*

the-fly) [Levanoni-Petrank 2001] que reduzem *overhead*, custo do paralelismo, eliminando também totalmente as pausas (o que o torna viável para sistemas de tempo-real). Existem também algoritmo eficientes de coleta de ciclos, cujo processo será descrito a seguir.

Coleta de ciclos

Resolve o principal problema do algoritmo de contagem de referências. Baseia-se em duas observações: 1) *ciclos-lixo* só podem ser criados quando uma contagem cai para valor diferente de zero, e 2) em *ciclos-lixo*, toda a contagem é devido a ponteiros internos.

Objetos que tem contagem decrementada para valores diferente de zero são *candidatos* (*observação 1*) a serem lixo. O algoritmo realiza três passos locais nos candidatos

1. *Mark*: marca apenas objetos que têm ponteiros externos (*observação 2*)
2. *Scan*: varre o ciclo a partir do objeto candidato com ponteiro externo e restaura a marcação de objetos que forem alcançáveis.
3. *Collect*: coleta os nós cuja contagem for zero.

As *figuras 13, 14* e *15* ilustram um algoritmo de coleta de ciclos descrito em [Paz-Petrank 2003], compatível com máquinas virtuais executando em ambientes multiprocessados. A *figura 13* ilustra o estado do *heap* depois de uma coleta. Dois objetos são candidatos à remoção: objetos cuja contagem foi decrementada para valor diferente de zero. A *figura 14* ilustra o passo seguinte, onde o algoritmo navega nas referências a partir do candidato e conta apenas as referências externas ao ciclo. Nesta etapa, todos os objetos que são lixo estão marcados, porém existem objetos que não são lixo marcados também. Na *figura 15* foi restaurada a contagem dos nós que puderam ser alcançados através das referências externas. Os objetos que continuarem com contagem zero nesta etapa serão coletados.

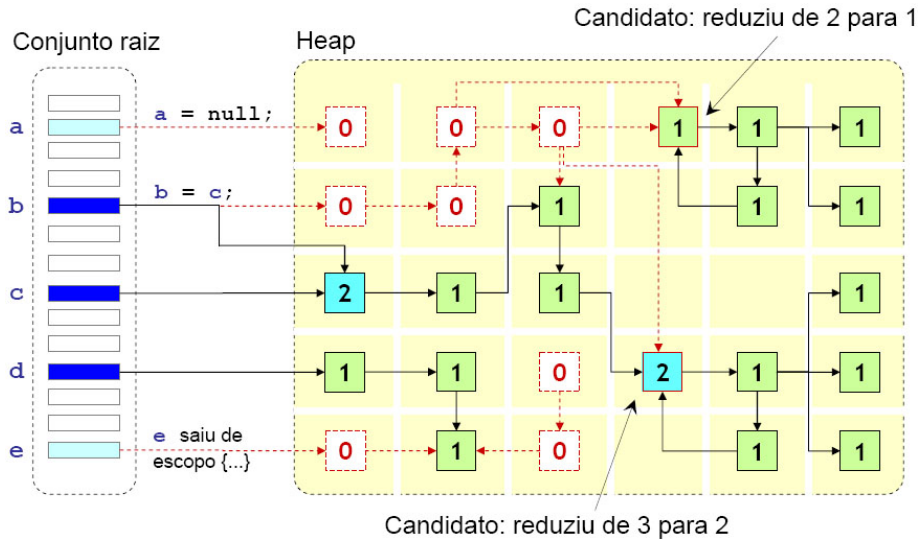

Figura 13 - Fase de identificação dos objetos candidatos, após uma coleta de contagem de referências normal. Os candidatos são os objetos cuja contagem diminuiu mas não a zero.

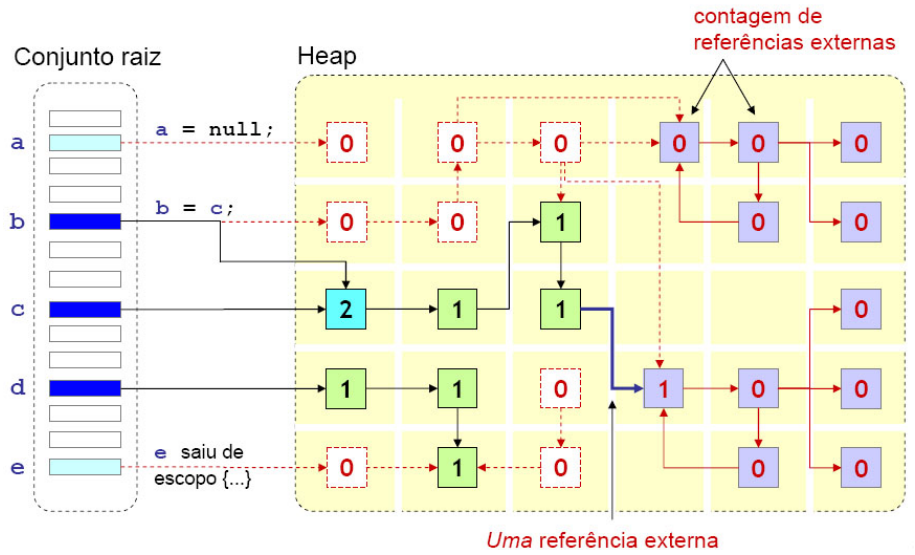

Figura 14 – Fase de marcação. Os ponteiros internos a partir dos objetos candidatos não são contados.

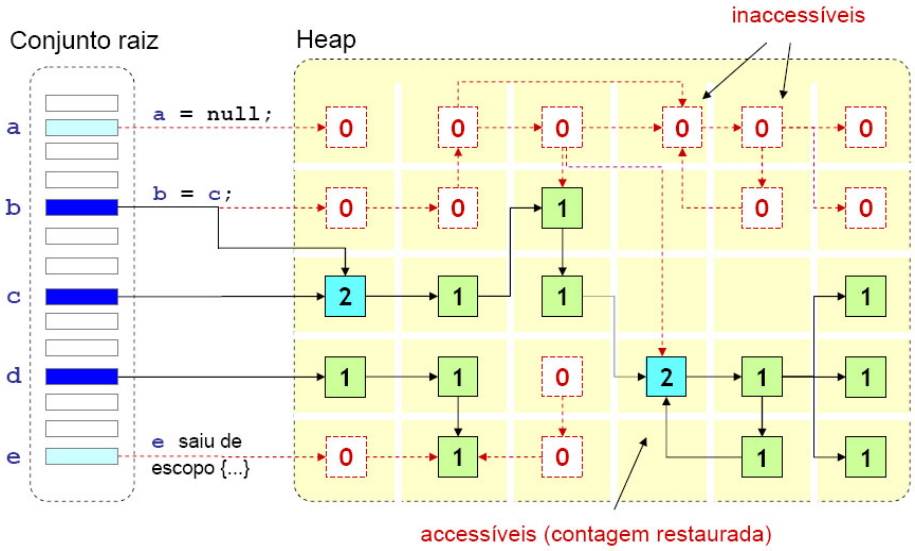

Figura 15 – Fase de varredura: se um objeto do ciclo for acessível através de referência externa, sua contagem é restabelecida.

O algoritmo trabalha apenas com objetos ativos e não precisa pesquisar todo o *heap*. Isto é uma vantagem que o torna um forte candidato para *heaps* grandes, já que as alternativas usadas atualmente (algoritmos de rastreamento) precisam pesquisar o *heap* inteiro. Além disso, por ser um algoritmo incremental, pode trabalhar em paralelo sem interromper a aplicação principal.

Por outro lado pode ser muito ineficiente se houver muitos ciclos, já que precisa passar três vezes por cada um deles, o que tornará a aplicação mais lenta mesmo não havendo pausas. Também precisará garantir a atomicidade das etapas de coleta de ciclos, caso venha a ser usado em sistemas paralelos.

O algoritmo de coleta de ciclos não é usado nas máquinas virtuais Java comerciais (até a versão 5.0). Tem sido usado com sucesso em máquinas virtuais experimentais (*Jikes RVM*) e fundamental em estratégias como coletores *age-oriented* paralelos [Paz et al. 2005] que assumem *heaps* grandes.

Algoritmos de rastreamento (tracing algorithms)

Marcam as referências que são alcançáveis (navegando a partir do conjunto raiz), e remove todas as referências que sobrarem. A *figura 16* ilustra um caminho de referências alcançáveis marcadas como ativas. As que sobrarem serão removidas.

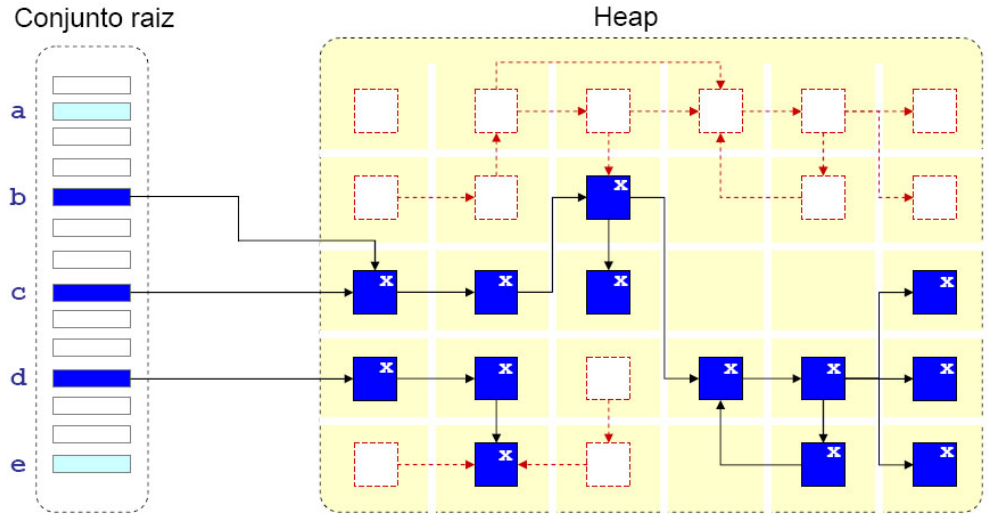

Figura 16 – Objetos ativos marcados por um algoritmo de rastreamento.

Algoritmo Mark and Sweep

O algoritmo *mark and sweep* (ou *mark-sweep*) é o mais simples algoritmo de rastreamento. É geralmente disparado quando a memória do *heap* atinge um nível crítico (ou acaba) e então todos os *threads* da aplicação param para executá-lo. Esse comportamento é chamado de "*stop-the-world*". Difere do modo incremental possibilitado pela contagem de referências, que não precisa parar a aplicação.

O algoritmo *mark-sweep* foi originalmente projetado para a linguagem LISP pelo seu criador [McCarthy 1960]. Tem duas fases (*ilustradas na figura 17*)

- *Mark*: navega pelos objetos alcançáveis a partir do conjunto raiz e deixa uma marca neles.
- *Sweep*: varre o *heap* inteiro para remover os objetos que não estiverem marcados (lixo), liberando a memória.

A principal vantagem do algoritmo *mark-sweep* está na sua simplicidade. Remove todo o lixo sem complicação. Não importa se há referências circulares ou não. Desta forma, pode ser mais rápido que a contagem de referências *se* o *heap* não for excessivamente grande e se objetos morrerem com freqüência. Em *heaps* grandes com objetos longevos a contagem de referências com coleta de ciclos tende a ser mais vantajosa.

Por outro lado o algoritmo *mark-sweep* (e todos os algoritmos de rastreamento) precisa interromper todos os *threads* da aplicação principal para poder executar. Um desafio dos coletores de lixo modernos é garantir que essa pausa seja imperceptível. Outro problema é a fragmentação do *heap*, que pode aumentar rapidamente se houver coletas freqüentes. Uma desfragmentação

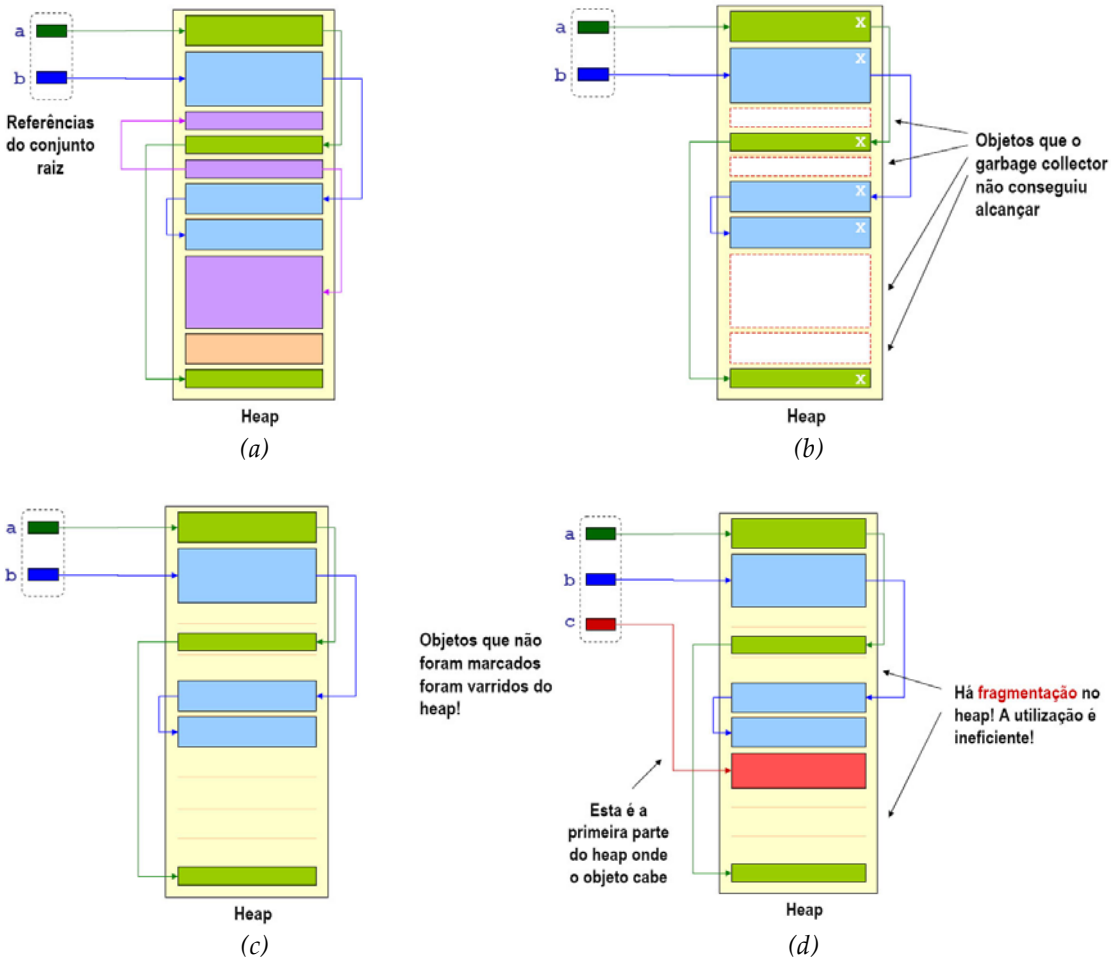

Figura 17 – Algoritmo Mark and Sweep mostrando o heap (a) antes da marcação; (b) após a marcação dos objetos alcançáveis; (c) após a liberação da memória; (d) durante a alocação de um novo objeto.

necessária (após várias coletas) é cara e requer a interrupção de todos os *threads* da aplicação por um tempo maior que as pausas de coleta. Um *heap* muito fragmentado diminui a disponibilidade de memória e pode aumentar a freqüência em que as coletas ocorrem, com passar do tempo. Finalmente, em *heaps* grandes esse algoritmo tem baixa performance. Embora precise visitar apenas os objetos alcançáveis na fase de marcação, depois precisa varrer o *heap* inteiro para localizar objetos não marcados e liberar a memória.

Algoritmo Mark and Compact

Um dos problemas do algoritmo *mark-sweep* é solucionado pelo algoritmo *mark-compact*. É um algoritmo de rastreamento baseado no algoritmo *mark-sweep* que acrescenta um algoritmo de compactação que elimina a fragmentação de memória. Assim, depois de cada coleta os objetos estão todos juntos e a memória livre é contígua, tornando a alocação mais simples e dispensando a necessidade de um algoritmo de alocação baseado em *free lists*. Para alocar memória para um novo objeto, basta localizar o final do último objeto alocado e usar a memória que for necessária.

O algoritmo consiste de duas fases. A primeira é idêntica ao *mark-sweep*, que marca os objetos alcançáveis. A fase seguinte move os objetos alcançáveis sobreviventes para frente até que a memória que eles ocupam seja contígua. O funcionamento do algoritmo *mark-compact* está ilustrado na *figura 18* a partir da compactação do *heap*.

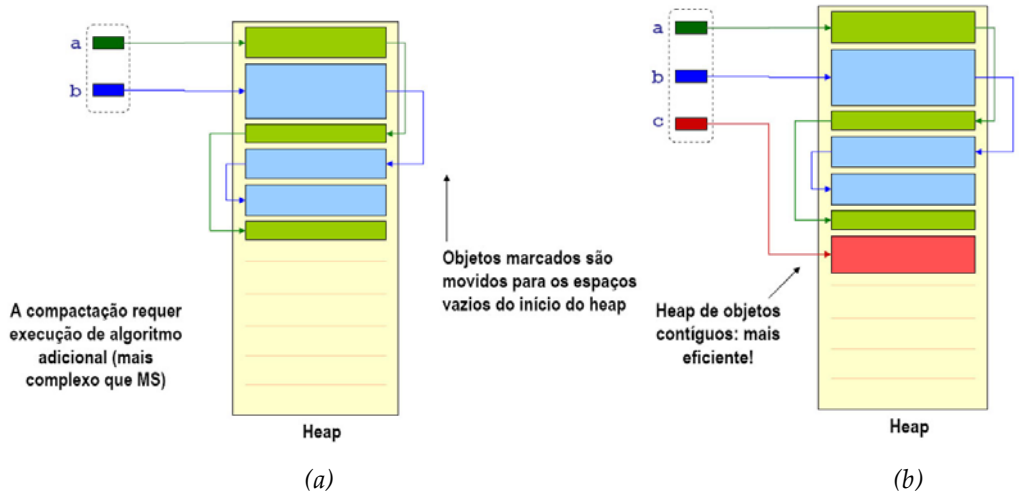

Figura 18 – Algoritmo Mark and Compact: (a) estado do heap após uma coleta; (b) alocação de um novo objeto no heap não-fragmentado.

A principal vantagem deste algoritmo em relação ao algoritmo *mark-sweep* é não causar fragmentação da memória. Isto torna a alocação rápida e sua performance não se degrada com o tempo devido ao aumento das coletas. A alocação rápida é importante não apenas na criação de objetos novos, mas também durante a coleta. Estratégias de coleta de lixo que dividem o *heap* em áreas chamadas de gerações realizam freqüentes coletas em áreas dedicadas a objetos jovens seguidas por alocações em áreas para objetos sobreviventes. Um *heap* não fragmentado é essencial para que essas coletas sejam eficientes.

O *mark-compact* continua sendo um algoritmo *stop-the-world* e seu algoritmo de compactação introduz um *overhead* maior, já que requer várias visitas aos objetos. As pausas, portanto, tendem a ser maiores que as pausas em *mark-sweep*. Também é mais difícil implementar uma versão concorrente (existem versões experimentais mas nenhuma foi ainda (até a versão 5.0) utilizada nas máquinas virtuais Java comerciais).

Algoritmo de cópia

O algoritmo de cópia (*Copying algorithm* [Chenney 1970]) divide o *heap* em duas áreas iguais chamadas de espaço origem (*from space*) e espaço destino (*to space*). Funciona da seguinte maneira:

1. Objetos são alocados na área "from space".
2. Quando o coletor de lixo é executado, ele navega pela corrente de referências e copia os objetos alcançáveis para a área "to space".
3. Quando a cópia é completada, os espaços "to space" e "from space" trocam de papel.

As *figuras 19 a 22* mostram o funcionamento do algoritmo de cópia.

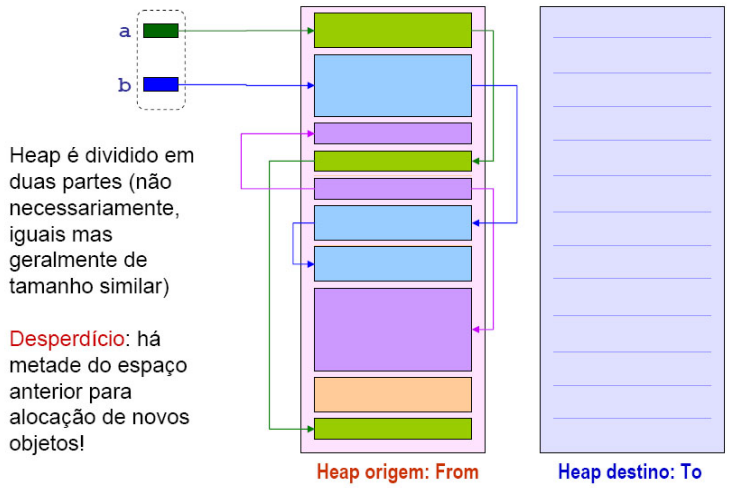

Figura 19 – O heap origem enche e dispara a coleta de lixo.

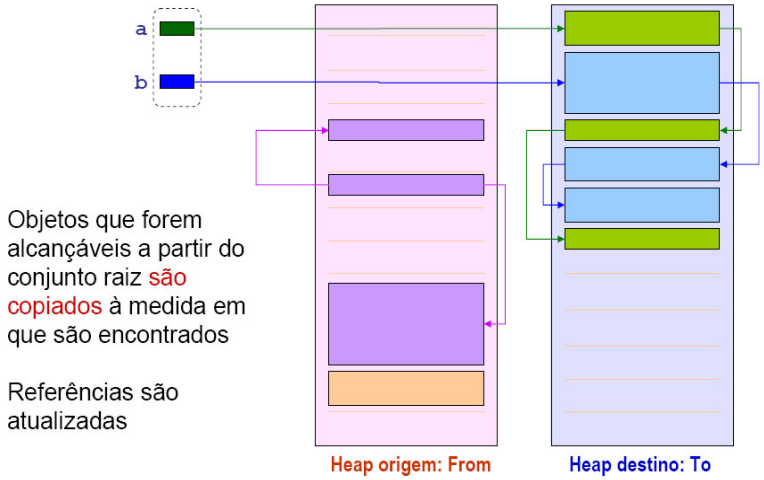

Figura 20 – Objetos alcançáveis são copiados e suas referências são varridas e atualizadas.

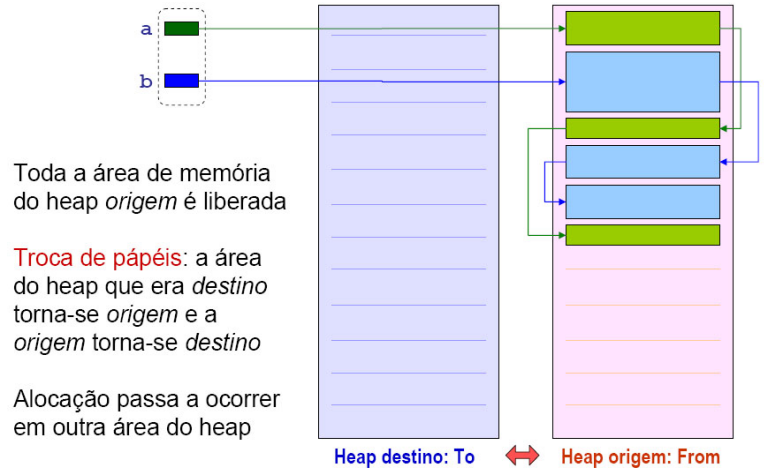

Figura 21 – Os objetos que restarem na origem são eliminados. A origem torna-se o destino e o destino torna-se a origem. O destino permanece vazio até a próxima coleta.

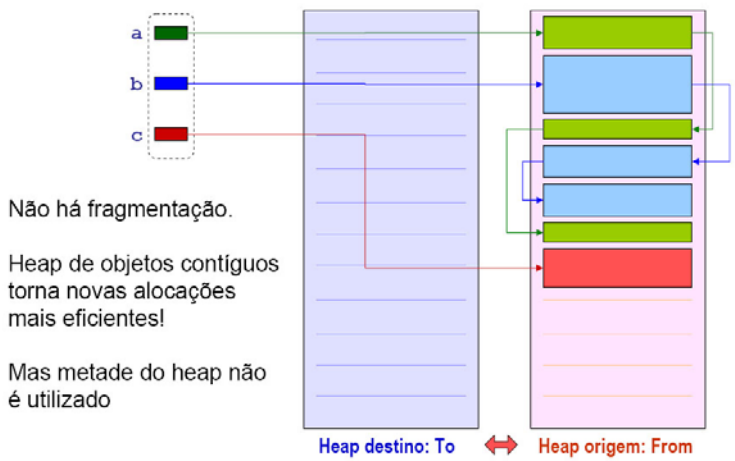

Figura 22 – Todas as alocações são feitas em área não fragmentada.

Este algoritmo possui várias vantagens. É mais simples, a cópia é rápida (principalmente se a quantidade de objetos alcançáveis for pequena, o que é comum), e não precisa visitar o *heap* inteiro: apenas os objetos alcançáveis. Também não fragmenta a memória do *heap*.

Por outro lado a aplicação ainda precisa parar (*stop-the-world*) enquanto o algoritmo está sendo executado (como em qualquer algoritmo de rastreamento). É possível reduzir bastante as pausas usando versões concorrentes (proposto em [Baker 78]) nas plataformas com vários processadores.

A principal desvantagem é o consumo e desperdício de memória. Esse algoritmo dobra a necessidade de memória do *heap*, e ainda mantém metade sem uso[6]. Há dois problemas relacionados a isto. O primeiro é a possível falta de memória se o *heap* necessário for muito grande. O segundo é a freqüência de coletas que pode aumentar ao reduzir o *heap* a um tamanho menor (com metade do tamanho normal) diminuindo a eficiência da coleta de lixo.

Em vermelho, geralmente um critério negativo

	precisa varrer heap inteiro	coleta todo o lixo	precisa parar a aplicação*	fragmenta o heap	permite uso total do heap	usado no HotSpot JVM**	
Contagem de referências	não	não	não	sim	sim	não	
Coleção de ciclos	não	sim	não	sim	sim	não	
Mark-sweep	sim	sim	sim	sim	sim	sim	
Mark-compact	sim	sim	sim	não	sim	sim	
Copying	não	sim	sim	não	não	sim	

* não funciona de forma incremental (stop-the-world) ** até versão 5.0

Tabela 1 – Quadro comparativo entre os algoritmos de coleta de lixo elementares.

[6] Isto depende da implementação. Várias implementações reservam espaços de tamanho desigual baseado na distribuição de ciclo de vida dos objetos.

3. Estratégias de coleta de lixo

Coletores modernos combinam vários algoritmos em estratégias mais complexas, aplicando algoritmos diferentes conforme as idades e localização dos objetos, e utilizando técnicas que possibilitem a coleta de lixo paralela (algoritmos incrementais e concorrentes). Nesta seção apresentaremos as principais estratégias usadas (e propostas) para coletores seriais e paralelos:

- *Generational garbage collection* (usada na JVM *HotSpot*)
- *Age-oriented garbage collection* (usada em implementações experimentais)

Ambas baseiam-se na *idade dos objetos* para tornar as coletas mais eficientes. Fundamentam-se em três observações empíricas:

- Se um objeto tem sido alcançável por um longo período, é provável que continue assim;
- Em linguagens funcionais, a maior parte dos objetos (95%) morre pouco depois de criados *(figura 23)*;
- Referências de objetos velhos para objetos novos são incomuns.

Conclusão: pode-se tornar mais eficiente o coletor de lixo analisando-se os objetos jovens com mais freqüência que os objetos mais velhos.

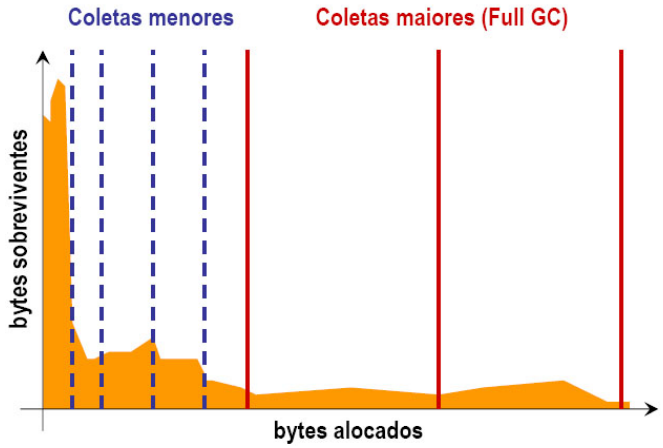

Figura 23 – Objetos morrem jovens! Fundamento para o tratamento diferenciado de objetos com base na sua idade. Coletas menores liberam memória de objetos jovens. Coletas maiores liberam memória do heap inteiro. Fonte [Sun 05].

Generational garbage collection

A estratégia de coleta de lixo com base em gerações classifica objetos em diferentes grupos de acordo com a sua idade. Os grupos são chamados de gerações. Se considerarmos n gerações G_0, G_1, ..., G_n, a geração G_0 será a geração que contém os objetos mais jovens (recém-criados.) A geração seguinte conterá objetos que sobreviveram a uma coleta de lixo e assim por diante[7]. A coleta de lixo é realizada separadamente em cada geração, e será mais freqüente nas gerações mais jovens que nas gerações mais velhas, ou seja, G_n será varrida mais freqüentemente que G_{n+1}. Pressupõe-se que a maior parte dos objetos jovens

[7] A forma de implementação do algoritmo pode variar.

(90%) já seja lixo antes da próxima coleta. Os objetos sobreviventes são promovidos para a geração seguinte.

As gerações mais velhas devem ser maiores que as gerações mais novas. Tipicamente são exponencialmente maiores. Implementações típicas dessa estratégia usam apenas duas gerações chamadas de *geração jovem* (G_0) e *geração estável ou velha* (G_1).

As gerações representam áreas do *heap*. A geração jovem é a área menor, onde é inicialmente alocada a memória para novos objetos. A geração antiga, ou estável, é uma área maior onde o espaço alocado não é para novos objetos mas para acomodar objetos que sobrevivem a uma ou mais coletas de lixo na área menor. Na transferência, todos os ponteiros entre objetos precisam ser atualizados.

Quando um objeto é criado, suas referências *geralmente* apontarão para objetos mais antigos. Se houver ponteiros entre gerações, provavelmente serão da geração nova para a geração velha. Mas pode acontecer de um objeto antigo receber referência para um objeto novo algum tempo depois de criado. Neste caso o sistema precisa interceptar modificações em objetos antigos e manter uma lista de referências. Isto deve ocorrer raramente (se ocorrer com freqüência, as coletas menores serão demoradas).

Na *HotSpot JVM*, é usada uma tabela de referências (*card table*) para controlar ponteiros entre gerações (*inter-generational pointers*). A geração antiga é dividida em blocos de 512kb (chamadas de *cards*). Alterações são interceptadas e blocos onde elas ocorrem são marcados. As coletas menores (coletas realizadas na geração jovem) verificam apenas os blocos marcados.

A coleta de lixo baseada em gerações usa mais de um algoritmo para realizar as suas coletas de lixo, uma vez que cada geração possui tamanhos e comportamentos diferentes.

Na geração jovem, aproximadamente 90% dos objetos já estão mortos e a área total do *heap* usado é pequena (geralmente bem menor que a geração estável). Neste caso, o *algoritmo de cópia* é a melhor opção pois seu custo é proporcional aos objetos ativos. Como a maior parte dos objetos está morto, há poucos objetos a copiar.

Na geração estável pode haver muitos objetos ativos e área é grande. Nessa situação o ideal seria usar um algoritmo como contagem de referência com coleta de ciclos já que é eficiente com *heaps* maiores e pode ser incremental. O *HotSpot* prefere usar uma implementação de *mark-sweep* concorrente ou *mark-compact* serial. Ambos precisam varrer o *heap* inteiro e não são totalmente incrementais (são *stop-the-world*), embora a versão concorrente consiga reduzir significativamente a duração das pausas em sistemas com muitos processadores.

Nas implementações mais comuns de coleta de lixo baseada em gerações que usam duas gerações (G_0: jovem e G_1: estável), usa-se um algoritmo de cópia na geração jovem. Um algoritmo de coleta de lixo é disparado sempre que uma

geração enche. As coletas podem ser *parciais* (apenas na geração jovem) ou *completas* (no *heap* inteiro.)

A *coleta parcial, ou menor* ocorre quando a geração jovem enche. Ela sempre enche primeiro, já que acumula objetos mais rapidamente. Quando isto acontece, ela dispara uma coleta menor, que é rápida (proporcional ao número de objetos ativos). Os sobreviventes da coleta serão copiados para a geração antiga.

A *coleta completa, ou maior* ocorre quando a geração antiga enche. Ela cresce ao receber os sobreviventes da geração jovem. Vários objetos irão morrer na geração antiga. Depois de várias coletas menores, a geração antiga enche, e quando isto acontecer haverá uma coleta maior (mais lenta) no *heap* inteiro, que irá não só remover objetos velhos como eventuais objetos jovens que forem encontrados.

As ilustrações da *figura 24* mostram o funcionamento.

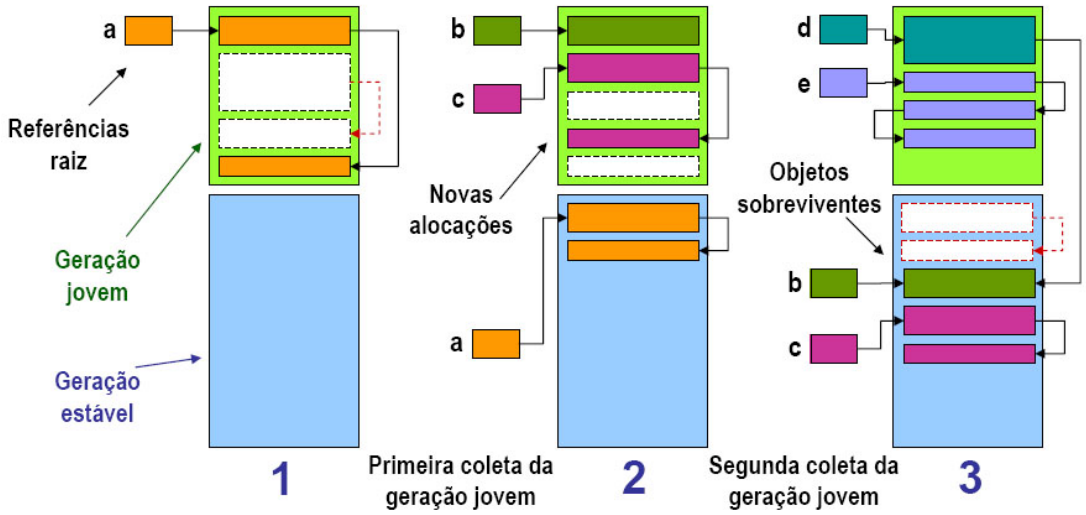

Figura 24 – Ilustração de duas coletas menores (parciais) usando algoritmo de cópia. Neste modelo [8], o to_space sempre é a geração estável. As coletas ocorreram apenas na geração jovem. Quando a geração estável enchar, haverá uma coleta no heap inteiro.

A estratégia de coleta de lixo baseada em gerações consegue obter pausas menores, já que coletas rápidas e freqüentes distribuem as pausas de tal maneira que podem tornar-se imperceptíveis. Também consegue aumentar a eficiência (*throughput*), concentrando a coleta nas áreas de memória onde o lixo se encontra, gastando menos tempo.

A pequena geração jovem pode causar um início mais lento devido a muitas coletas curtas, o que causa baixa eficiência. Essas coletas provocarão pausas que podem acontecer no início da aplicação se a geração jovem encher várias vezes quando a aplicação estiver sendo iniciada. Esse é um comportamento possível e talvez comum em várias aplicações. As pausas curtas podem não ser percebidas,

[8] O modelo usado no *HotSpot* difere um pouco deste modelo mais simples. Veja a *seção 2* para detalhes sobre a implementação no *HotSpot*.

mas se várias ocorrerem em uma curta seqüência, pode parecer que houve uma longa pausa.

Por usar uma área maior, a coleta na geração antiga ainda é lenta, principalmente com algoritmos de rastreamento (usados no *HotSpot*). Os algoritmos usados atualmente nas máquinas virtuais comerciais ainda não conseguem eliminar totalmente as pausas.

Na JVM *HotSpot*, a geração antiga permite a escolha entre diversos algoritmos: *mark-sweep*, *mark-compact* e um algoritmo incremental (*train*). Existem pesquisas usando contagem de referências com coleta de ciclos para coletar geração antiga eficientemente (veja, por exemplo, [Azatchi-Petrank 03], no qual implementações foram testadas na JVM experimental *Jikes RVM* com bons resultados.)

Age-oriented garbage collection

A estratégia de *coleta de lixo baseado na idade dos objetos* também divide objetos em gerações, mas estas podem ocupar espaços de tamanho variável do *heap*. As gerações estão associadas não a um lugar específico no *heap*, mas a cada objeto. As coletas sempre varrem o *heap* inteiro, o que pode provocar pausas muito longas. A solução proposta busca diminuir as pausas com concorrência (seria ineficiente em ambientes monoprocessados).

Ainda é uma solução experimental. As implementações recomendadas usam um algoritmo de rastreamento (*cópia*) na geração jovem (da mesma forma que implementações típicas do *generational garbage collection*) e um algoritmo de *contagem de referências com coleta de ciclos* na geração antiga.

Inicialmente geração jovem ocupa todo o espaço (*figura 25*), o que garante alta eficiência (demora a ocorrência da primeira coleta). O espaço reservado para a geração antiga cresce à medida em que ocorrem coletas na geração jovem, mas ela é sempre menor que a geração jovem. Uma pequena geração antiga com mais objetos ativos que mortos e pouca atividade permite eficiência máxima do algoritmo de coleta de ciclos.

Figura 25 – Gerações de objetos novos e antigos em duas estratégias baseadas na idade dos objetos. Fonte: *[Paz-Petrank-Blackburn 05] (slides).*

A *tabela 2* compara a estratégia de coleta de lixo *age-oriented* com a estratégia *generational*.

Generational	Age-oriented
Geração jovem menor que geração velha.	Geração jovem maior que geração velha.
Faz coletas freqüentes apenas na geração jovem. Após várias coletas da geração jovem, faz coleta do *heap* inteiro, com algoritmos diferentes para cada geração.	Sempre coleta o *heap* inteiro, usando algoritmos diferentes para cada geração.

Tabela 2 – Comparação entre estratégias de coleta de lixo baseadas em idade dos objetos.

A estratégia *age-oriented* procura manter a *maior* geração jovem possível, o que torna as coletas de lixo mais raras e pausas menos freqüentes. Também permite um início mais rápido da aplicação. A eficiência é buscado da mesma forma que na estratégia *generational*: tratando cada geração diferentemente.

A principal desvantagem dessa estratégia é não funcionar bem para pequenos sistemas. É ideal para sistemas com muita memória e muitos processadores paralelos. Foi concebida tendo esse tipo de ambiente em vista. As pausas serão longas na geração jovem se não for implementado em um coletor paralelo, executando um algoritmo de cópia concorrente. E como a geração antiga é coletada freqüentemente, é importante que se use um algoritmo incremental como contagem de referências. A coleta será ineficiente se for usado algoritmo de rastreamento e as pausas introduzidas poderão ser excessivas.

Atualmente esse algoritmo é usado apenas experimentalmente e não é suportado por nenhuma máquina virtual comercial como a *HotSpot*, mas algumas de suas idéias poderão influenciar algoritmos de coleta de lixo nas máquinas virtuais do futuro. O artigo [Paz-Petrank-Blackburn 05] documenta *benchmarks* realizados com implementações desse algoritmo na máquina virtual experimental *Jikes RVM* que obtiveram uma performance média melhor que a implementação do *generational GC* que é atualmente usada no *HotSpot*.

4. Coleta de lixo em paralelo

Os maiores problemas da coleta de lixo: as pausas e a redução da eficiência da aplicação podem ser minimizados usando coleta de lixo em paralelo, principalmente em sistemas com mais de um processador. Coletores paralelos geralmente combinam os algoritmos básicos já vistos e implementam extensões para torná-los seguros e eficientes nesses ambientes.

No que se refere ao paralelismo, as principais estratégias são:

- *Coleta serial*: o coletor ocorre em série com a aplicação, parando o mundo (*stop-the-world*) quando precisar liberar memória. Às vezes coletas seriais são realizadas em estratégias paralelas para realizar tarefas mais raras, buscando maior eficiência em detrimento de possíveis pausas longas.
- *Coleta incremental* (*on-the-fly*): o coletor executa em paralelo realizando coletas pequenas (não necessariamente completas) sempre que possível, usando vários *threads* buscando menos (ou zero) pausas.
- *Coleta concorrente*: o coletor realiza suas principais tarefas em um processador ou *thread* exclusivo (pode parar todos os *threads* para marcar, se necessário) buscando maior eficiência.

Coletores incrementais

Os algoritmos seriais de rastreamento precisam parar todos os *threads* para realizar coleta de memória. Isto é inaceitável para aplicações de tempo real. Uma solução são os algoritmos de coleta incremental (*on-the-fly*.)

Um algoritmo incremental permite que a aplicação execute enquanto a coleta de lixo acontece. Uma das soluções é o algoritmo de marcação tricolor (*tri-colour marking - TCM*). TCM é um algoritmo de rastreamento que atribui um entre três estados (cores) a um nó do grafo de objetos. É o principal algoritmo de rastreamento incremental (considerando a engenharia de software como um todo, e não apenas Java) e a base para outras implementações populares. Classifica os nós em diferentes tipos marcando-os com as cores: branco, cinza e preto. Executa os seguintes passos:

1. Inicialmente todos os nós são *brancos* (inalcançáveis) e o conjunto de referências raiz é marcada *cinza*.
2. Quando o coletor encontra um caminho entre um nó cinza e um nó branco, pinta o nó branco de cinza. Depois prossegue recursivamente até encontrar todos os objetos alcançáveis a partir dele, pintando cada objeto encontrado de cinza.
3. Quando todos os caminhos de um nó cinza levam a nós cinza ou pretos, o nó é pintado de *preto*.
4. Quando não houver mais nós cinzas, todos os nós alcançáveis foram encontrados. Os nós brancos restantes são reciclados.

O processo está ilustrado nas *figuras 26 a 30*.

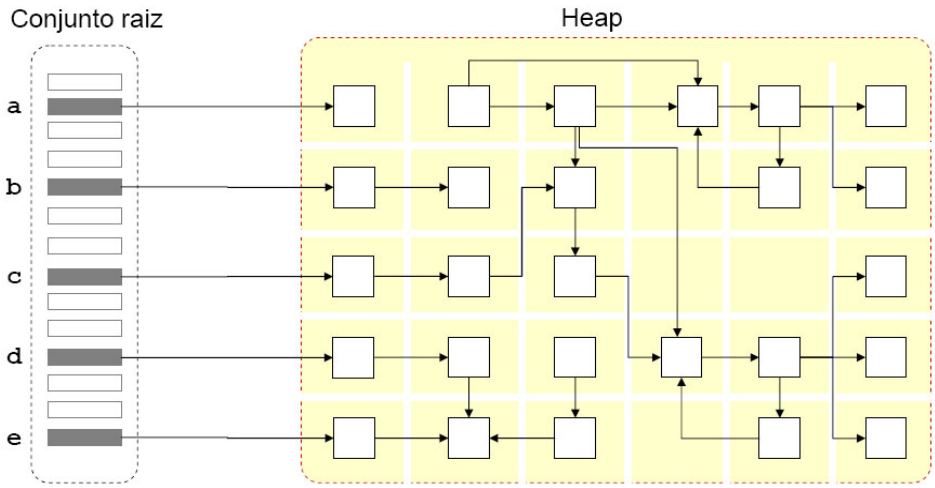

Figura 26 – Inicialmente todas as referências são brancas. As referências raiz são marcadas cinza.

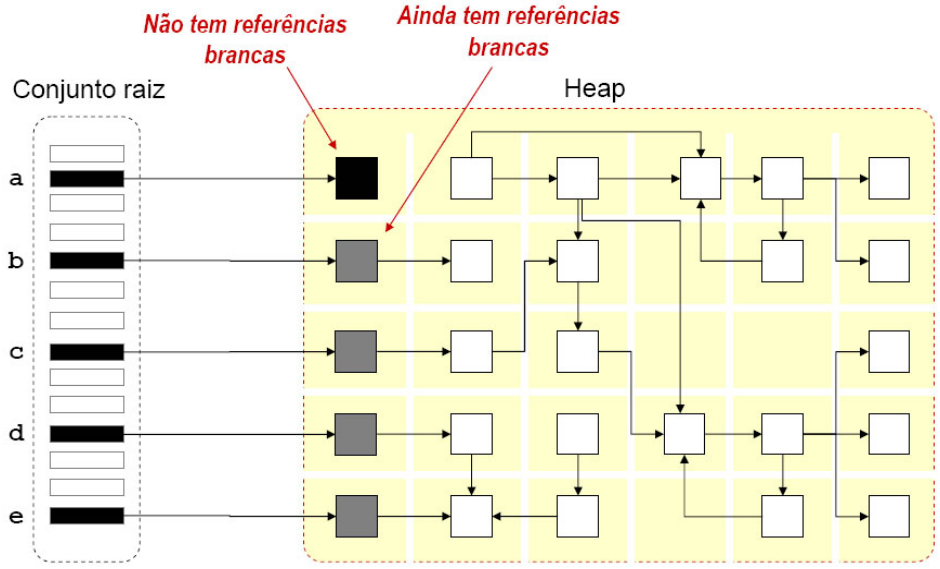

Figura 27 – A partir das referências raiz, as referências alcançadas são marcadas cinza.

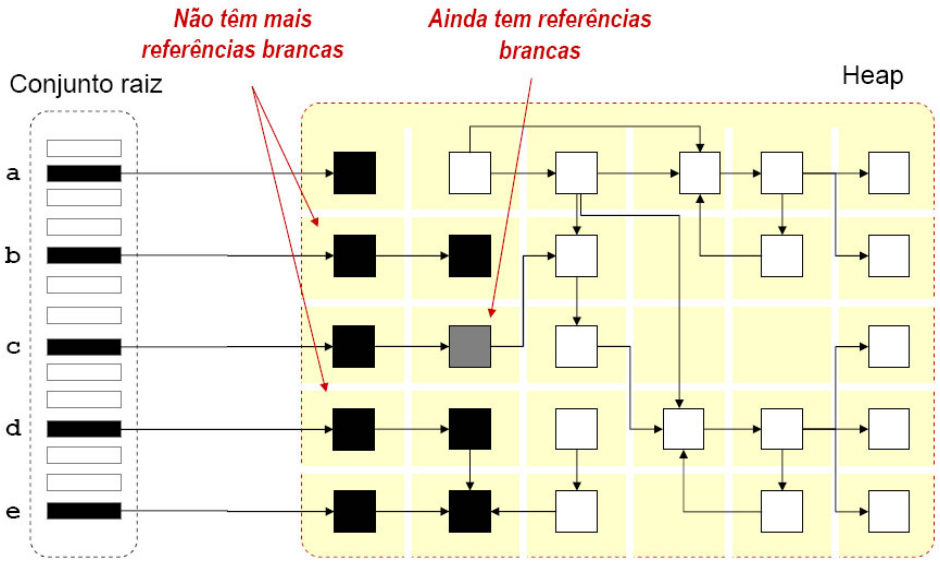

Figura 28 – Objetos que não tem referências brancas são marcados como pretos.

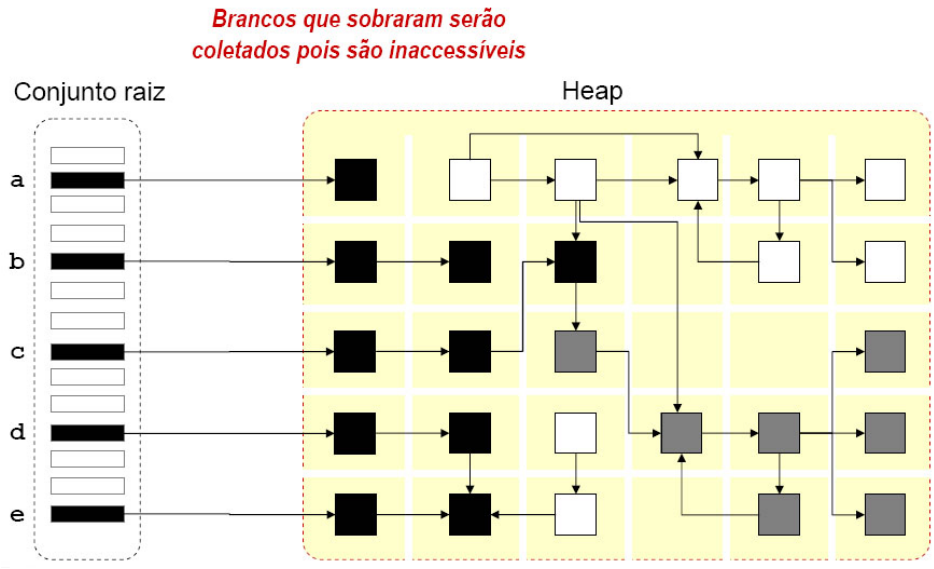

Figura 29 – Objetos alcançáveis são marcados como cinzas.

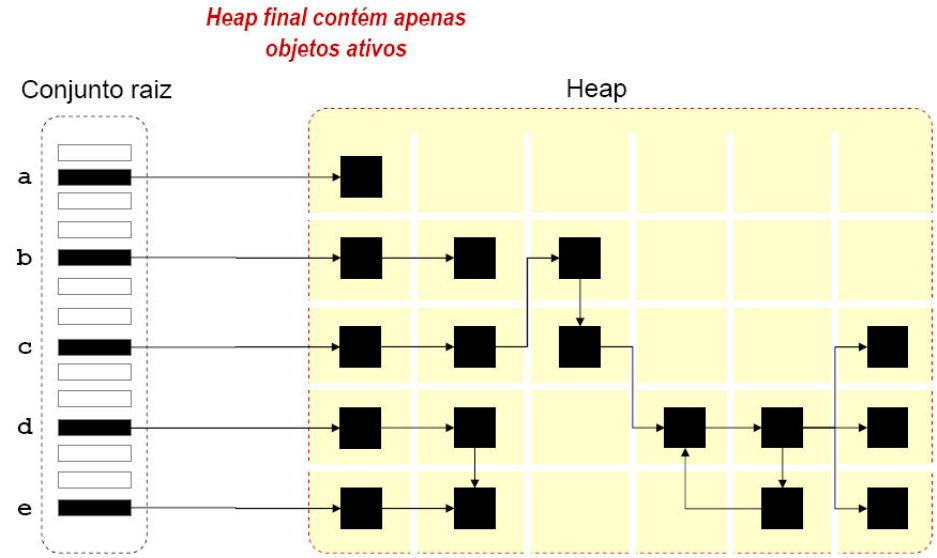

Figura 30 – Memória é liberada.

Um objeto preto nunca poderá ter referências para objetos brancos. Quando aplicação gravar uma referência entre um nó preto e um branco, o coletor precisará pintar ou o nó pai ou o nó filho de cinza. Quando a aplicação quiser ler um nó branco, ele tem que ser pintado de cinza. Para realizar isto, o sistema precisa:

1. Rastrear gravações em nós pretos (através de uma barreira de gravação – *write barrier*) e
2. Rastrear leituras em nós brancos (através de uma barreira de leitura – *read barrier*).

A principal vantagem do algoritmo TCM é a possibilidade de uso incremental e eliminação de pausas na coleta de lixo, permitindo o seu uso em aplicações de tempo real. Tem, pelas mesmas razões, uma melhor performance

aparente. Por outro lado é complexa a sincronização é complexa entre a aplicação e o coletor de lixo. Barreiras podem dificultar a implementação em diferentes sistemas e diminuir a eficiência.

Train algorithm

As máquinas *HotSpot* não usam TCM mas um outro popular algoritmo incremental chamado de *algoritmo do trem* (*train algorithm*). Este algoritmo aplica alguns dos princípios das estratégias de classificação dos objetos por idade. Divide a memória em blocos de tamanho fixo (no *HotSpot* são blocos de 512kB) apelidados de *vagões*. Coleções de tamanho arbitrário de vagões interligados são chamados de *trens*. Trens e vagões são ordenados por idade; os mais antigos são coletados enquanto novos trens e vagões se formam. Entre a formação e coleta, atualiza-se referências entre objetos.

Este algoritmo é muito ineficiente com objetos *populares* (objetos que têm muitas referências) que podem ocorrer com freqüência nas gerações estáveis. Versões eficientes deste algoritmo lidam com esse problema mas podem ter pausas. As pausas são pequenas mas não são previsíveis.)

É um dos algoritmos usados no *HotSpot* e será explorado em mais detalhes na próxima seção.

Snapshots e Sliding Views

Coletores paralelos precisam trabalhar com *heaps* que mudam durante a coleta e ainda assim garantir a coleta de todo o lixo, mas em sistemas paralelos enquanto um *thread* marca os objetos outro *thread* pode estar liberando referências (gerando lixo). Para realizar uma coleta completa é preciso trabalhar com modelos estáticos do *heap* (*snapshots* ou *sliding views*) e coletar de forma incremental.

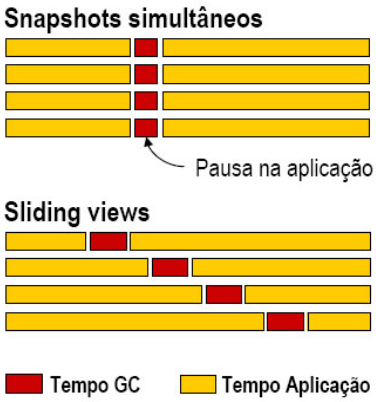

Figura 31 – Coleta concorrente vs. coleta incremental

Snapshots e *sliding views* são *a mesma coisa* (devem representar os mesmos dados). A diferença está na forma como são obtidos e usados.

Snapshots são as visões estáticas do *heap* usadas por coletores de lixo *concorrentes*, mas não necessariamente incrementais sem pausas (veja *figura 31*). Um coletor de lixo concorrente usa vários *threads* para executar os algoritmos de coleta, mas pode ainda ser do tipo *stop-the-world* e sincronizar as interrupções de todos os *threads* da aplicação para realizar a coleta mais rápida e eficientemente. Mesmo que realize parte da coleta incrementalmente, para obter um modelo do *heap* completo: o *snapshot*, o coletor de lixo precisa parar em um determinado momento todos os *threads*.

Sliding views são visões estáticas do *heap* usadas por coletores *incrementais*, ou *on-the-fly*. Esses coletores param um *thread* de cada vez, em tempos

desencontrados para obter visões completas do *heap*. Desta maneira conseguem evitar pausa na aplicação, porém podem afetar a eficiência da aplicação, uma vez que vários *sliding views* terão dados repetidos e o tempo de processamento da aplicação será disputado com o coletor de lixo.

Apesar das máquinas virtuais *HotSpot* atualmente não usarem algoritmos incrementais completamente sem pausas (possuem alternativas de pausas mínimas), é provável que os utilizem no futuro. É uma área de pesquisa emergente fundamental para sistemas de tempo real. Essa necessidade tem trazido de volta a possibilidade de uso de algoritmos de contagem de referência (com coleta de ciclos ou com *backup* de rastreamento), pois eles são eficientes com *heaps* grandes e sistemas paralelos.

Coletores concorrentes

Os algoritmos de cópia concorrente usados no *HotSpot* são todos algoritmos de rastreamento com pausas e não totalmente incrementais. Existem algoritmos de cópia incrementais (inicialmente propostos em [Baker 78]). A implementação é simples: ponteiros são lidos apenas em *to_space*; se ponteiro estiver em *from_space* na leitura, primeiro copia objeto depois obtém ponteiro. Um algoritmo similar é usado pela *HotSpot* JVM para coletar paralelamente a geração jovem. Veja [Flood et al 2001].

Um algoritmo *mark-sweep* concorrente é usado pelo *HotSpot* JVM para coletar paralelamente a geração antiga [Printezis 00]. Não é completamente incremental, porém reduz pausas. Há pausa pequena para obter *snapshot* (pára todos os *threads* ao mesmo tempo). Por não compactar, causa fragmentação. Existe uma versão com compactação em desenvolvimento [Flood 01] mas ela não é usada em nenhuma máquina virtual atual (até Java 5.0).

Conclusões

Existem muitas estratégias de coleta de lixo. Há muito, muito mais do que foi exposto aqui. Embora o *programador* Java não tenha a opção de escolher qual usar, as máquinas virtuais podem permitir essa escolha e configuração pelo administrador do sistema ou usuário. Muito pode mudar nas próximas versões das máquinas virtuais existentes atualmente: há muitas estratégias experimentais que poderão ser usadas em versões futuras, em diferentes plataformas; há estratégias antigas caindo em desuso. Conhecer o funcionamento dos principais algoritmos ajudará a configurar e ajustar a performance da máquina virtual Java em diferentes tipos de aplicações.

© 2005 Helder da Rocha

Parte II - Monitoração e configuração da máquina virtual *HotSpot*

A máquina virtual *HotSpot* é a máquina virtual da *Sun* para a plataforma Java. Foi concebida com o objetivo de obter a melhor performance para aplicações Java executando tanto em ambientes servidores como em ambientes cliente. É uma máquina virtual, ou seja, emula um processador com interface de execução e programação uniforme, através de diferentes plataformas de hardware. Oferece um modelo de consistência de memória próprio, altamente flexível e facilmente adaptável a ambientes mono- e multiprocessados, um otimizador adaptativo para compilação de suas instruções de máquina e um sistema de gerenciamento automático de memória.

Apesar da máquina virtual *HotSpot* ser previamente configurada para as situações mais comuns, ela permite a configuração de vários aspectos relacionados à gerência de memória. Entre eles:

- Escolha entre dois tipos de máquina virtual, previamente configuradas e otimizadas para ambientes distintos: 1) ambientes servidores e grandes aplicações de longa duração, ou 2) aplicações cliente de curta duração geralmente executando em desktops;
- Escolha de diferentes algoritmos e estratégias de coleta de lixo, permitindo a escolha e combinação de diferentes algoritmos em diferentes regiões do *heap*;
- Configuração de diversos parâmetros dos algoritmos de coleta de lixo como redimensionamento de gerações e políticas de ativação;
- Parâmetros de ajuste absoluto, relativo ou automático do tamanho total e gerações distribuídas no espaço do *heap;*
- Ajuste de tamanho da pilha (para todos os *threads*);
- Configuração automática (*ergonomics*) baseada em metas de eficiência e pausas máximas;
- Política de tratamento de referências fracas (*soft references*[9]);
- Geração de relatórios e *logs* contendo informações e estatísticas que podem ser usadas para auxiliar a configuração.

Esta seção explora esses recursos, mostra como usar as opções da máquina *HotSpot* e aponta estratégias de ajuste visando melhor performance na execução de aplicações Java.

[9] Mecanismo que pode ser usado pelo programador para influenciar a coleta de lixo.

5. Arquitetura da HotSpot JVM

A máquina virtual é configurada para as situações mais comuns da plataforma usada. Há duas opções básicas de máquina virtual a escolher:

- *Java HotSpot Client VM*: minimiza tempo de início da aplicação e memória utilizada. Para iniciar a máquina virtual com esta opção, use:

  ```
  java -client [outras opções] nome.da.Classe
  ```

- *Java HotSpot Server VM* (opcional): maximiza velocidade de execução da aplicação. Para iniciar a máquina virtual com esta opção, use

  ```
  java -server [outras opções] nome.da.Classe
  ```

As configurações são otimizadas para ambientes específicos. A *Server VM* não está disponível em todas as instalações do Java *HotSpot.* Ambientes desktop com um processador e menos de 2 GB de RAM não suportam ou serão ineficientes se usarem a *Server VM*.

A máquina virtual a ser usada como *default* é selecionada *automaticamente*, de acordo com a plataforma usada durante a instalação, que decide que máquinas grandes multiprocessadas usam *Server VM* e demais usam *Client VM* (a menos que o usuário mude através das opções.). A escolha durante a execução é feita também automaticamente, caso nenhum parâmetro seja passado para selecionar a máquina virtual. Normalmente a configuração escolhida automaticamente é a melhor opção.

Todas as máquinas virtuais *HotSpot* possuem um compilador adaptativo de *bytecode*. Aplicações são iniciadas usando o interpretador, mas, ao longo do processo o código é analisado para localizar gargalos de performance. Trechos ineficientes são compilados e outras otimizações (ex: *inlining*) são realizadas.

Todas as máquinas virtuais *HotSpot* também realizam alocação rápida de memória, liberação de memória automática usando algoritmos eficientes de coleta de lixo adaptados ao ambiente usado, e sincronização de *threads* escalável.

Há várias diferenças entre as duas máquinas virtuais. Os tamanhos *default* do *heap* e das gerações permanentes diferem nas duas opções. O compilador usado no *Server VM* faz otimizações mais agressivas, fazendo *inlining* inclusive de código que pode ter referências alteradas dinamicamente. Caso uma alteração ocorra, o compilador desfaz a otimização. Como as alterações são raras, é possível obter uma performance maior.

Opções de linha de comando

A máquina virtual pode ser configurada e ajustada por administradores através de opções de linha de comando. As opções apresentadas neste tutorial valem para as distribuições *HotSpot* da *Sun*. Como não são padronizadas, podem ser diferentes ou não existir em outras implementações.

Existem diversas opções –*X*. Elas são documentadas na linha de comando através do comando de ajuda:

```
java -X
```

Existem também várias opções –*XX*. Elas *não* são documentadas na linha de comando e podem não estar disponíveis (verifique na sua instalação.) A sintaxe das opções –*XX* difere das opções –*X*:

```
java -XX:+Opção1 -XX:Opção2=5 ... [+opções] pacote.Classe
```

Nem as opções -*X* nem as opções –*XX* são padronizadas. Elas não fazem parte da especificação da máquina virtual e podem mudar em versões futuras. Também diferem entre diferentes implementações do *HotSpot* e podem ter sintaxe e resultados diferentes em outras máquinas virtuais, como as da IBM. Elas estão documentadas em *http://java.sun.com/docs/hotspot/VMOptions.html*.

Há dois tipos de opções –*XX*: *booleanas* e *inteiras*. As *booleanas* possuem uma chave para ligar e desligar usando os símbolos + e -. As *inteiras* recebem um parâmetro.

- Opção booleana `-XX:<+/-><nome>`
- Opção inteira `-XX:<nome>=<valor>`

Nas opções booleanas, o **+** liga e o **-** desliga uma opção. Por exemplo:

`-XX:+Opcao` *(liga uma opção que estava desligada por default)*

`-XX:-Opcao` *(desliga uma opção que estava ligada por default)*

As opções inteiras recebem o valor diretamente através de **=**:

```
-XX:Valor=8
```

Breve história da coleta de lixo em Java

Até a versão 1.1 da plataforma Java, as máquinas virtuais da *Sun* usavam um único coletor *mark-sweep* para coletar todo o lixo. Esse coletor não só causava fragmentação de memória, como significava um alto custo de alocação e liberação (o *heap inteiro* precisava ser varrido em cada coleta). O resultado era pausas longas quando ocorriam as coletas e baixa eficiência da aplicação que perdia CPU para o coletor de lixo.

Com o *HotSpot*, a partir da versão 1.2, adotou-se o *generational collector* que emprega na geração jovem um *algoritmo de cópia* e na geração antiga um algoritmo *mark-compact*. Ambos eliminam a fragmentação de memória, tornando as alocações mais eficientes. O resultado foi uma melhoria considerável na execução de aplicações Java, tornando a plataforma Java extremamente eficiente. O *HotSpot* também inovou ao fazer a compilação seletiva do código Java, diminuindo o tempo de início das aplicações.

Novas implementações e opções foram acrescentadas a cada lançamento novo da linguagem. Entre as versões 1.3 e 1.5 surgiram diversas soluções mais eficientes de coletores seriais, paralelos de alta eficiência, concorrentes e incrementais. Todos foram adaptados ao modelo de separação de objetos em gerações. Com tantas opções, a configuração e escolha do coletor de lixo tornou-se uma tarefa árdua. Finalmente, na versão 1.5 foram introduzidos mecanismos de auto-ajuste, escolha e otimização baseado em análise ergonômica.

Nas seções seguintes exploraremos a arquitetura dos coletores usados no *HotSpot*, começando com o coletor mais simples: o serial.

O coletor de lixo serial do HotSpot

Em uma *HotSpot Client VM* o coletor serial é o coletor *default*. Não é preciso configurar nada para usá-lo. Ele é otimizado para a maior parte das aplicações, portanto, raramente será necessário alterar suas configurações.

Utiliza um *heap* dividido em gerações (estratégia *generational garbage collection*) implementado com uma *geração jovem* e uma *geração estável*. A geração jovem ainda é dividida em três áreas, para uma implementação mais eficiente do algoritmo de cópia.

Possui uma área do *heap* chamada de geração permanente, que apesar do nome, não faz parte das gerações usadas pela estratégia *generational garbage collection* usada no *HotSpot*. É uma área de memória alocada à parte do *heap* total usada para armazenar dados estáticos como métodos e classes. É raramente coletada (geralmente não é coletada).

Os algoritmos de coleta de lixo usados na implementação do *generational garbage collection* do *HotSpot* são:

- *Geração jovem*: usa algoritmo de cópia com duas origens e dois destinos (um destino temporário e um destino permanente, que é a geração estável). A geração jovem é menor que a geração estável e realiza coletas pequenas e freqüentes chamadas de coletas menores (*minor collections*).
- *Gerãção estável* (ou velha): usa algoritmo *mark-compact*. É uma área menor que recebe os sobreviventes da geração jovem. As coletas quando ocorrem são completas e abrangem todo o *heap*. São chamadas de coletas maiores (*major collections*).
- *Geração permanente*: usa um algoritmo *mark-sweep*. A coleta é rara e é disparada quando essa área enche.

A *figura 32* ilustra a organização do *heap* no *HotSpot*. As opções –X permitem alterar a organização *default* nos servidores que as suportarem.

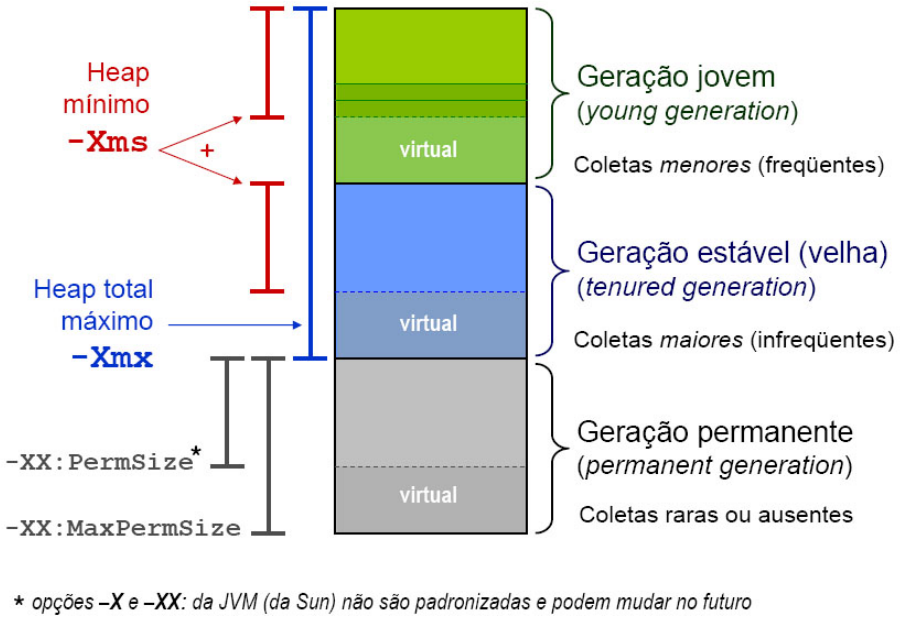

Figura 32 – Heap dividido em gerações gerenciado pela máquina virtual HotSpot.

Geração jovem

A geração jovem é dividida em três partes. Uma parte maior, sempre usada para alocação de novos objetos, e duas partes menores que revezam os papéis de origem e destino de um algoritmo de cópia.

A parte maior é chamada de *Éden*. É onde novos objetos são criados. No algoritmo de cópia, o *Éden* é sempre origem e nunca muda de papel. Sobreviventes de uma coleta esvaziam o *Éden* e são copiados para as áreas menores, chamadas de *espaços sobreviventes*. As áreas da geração jovem estão mostradas na *figura 33*.

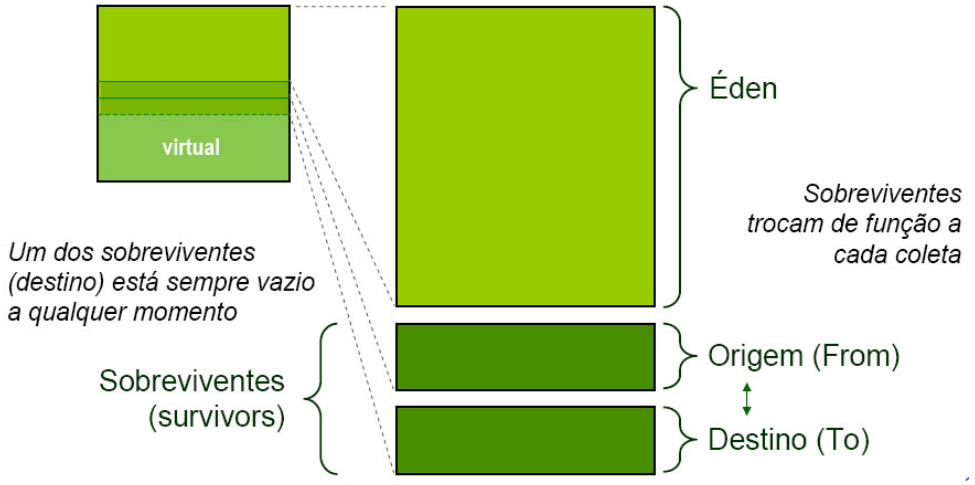

Figura 33 – Anatomia da geração jovem na máquina virtual HotSpot.

As coletas menores, ou *parciais*, são freqüentes e rápidas. Acontecem sempre que o *Éden* enche. Executam um algoritmo de cópia que trabalha com duas áreas de *origem* e duas áreas de *destino*. As áreas sobreviventes alternam função de origem e destino. Uma das duas está sempre vazia, como no coletor de cópia elementar. A área *Éden* é sempre origem. Objetos são criados e alocados no *Éden*, mas deixam o *Éden* na primeira coleta. A geração estável é sempre destino.

Quando o coletor de lixo executa uma coleta menor, todos os objetos alcançáveis que existirem no *Éden* e *sobrevivente origem* são copiados para a área *sobrevivente destino* ou geração estável. Se um objeto não couber no sobrevivente, será copiado diretamente para geração estável. O coletor de lixo pode *promover* um objeto que já foi copiado várias vezes entre as regiões sobreviventes e torná-lo estável.

No final da coleta, o *Éden* e área sobrevivente origem estão vazios. A origem muda de função e passa a ser destino. Coletas seguintes copiarão objetos entre os dois sobreviventes ou para a geração estável, quando tiverem idade suficiente. Um objeto nunca mais volta ao *Éden*. Ou morre no *Éden* ou sobrevive à coleta e é transferido para um espaço sobrevivente. Objetos que foram promovidos para a geração estável nunca mais voltam à geração jovem. A *figura 34* ilustra a organização do *heap* antes e depois de duas coletas.

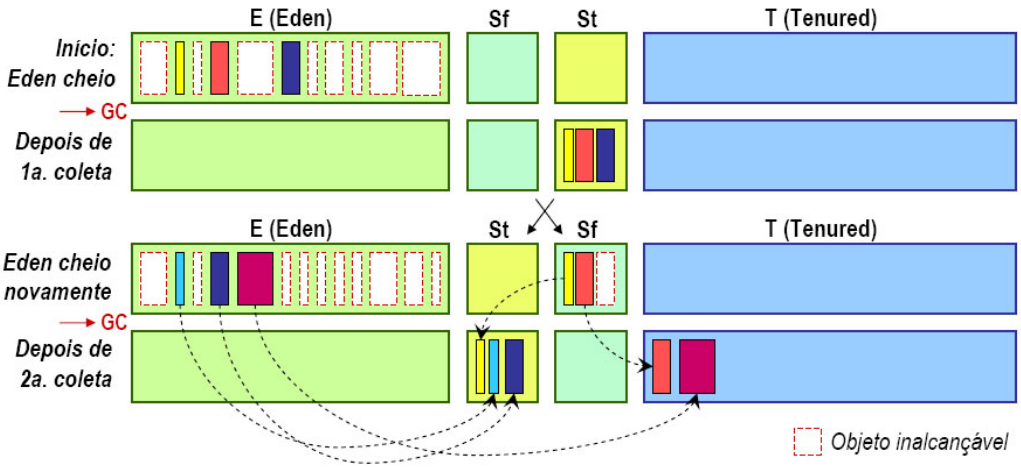

Figura 34 - Quando o Éden enche, coletor de lixo copia objetos alcançáveis do Éden (E) para sobrevivente To (St) – sempre esvazia o Éden; do sobrevivente From (Sf) para St – sempre esvazia o Sf; de Sf para a geração estável (T) (dependente de algoritmo); do Éden ou Sf para T (se não cabe em St).

Quando os objetos são copiados entre gerações, suas referências precisam ser atualizadas para conter os novos endereços. A figura 35 ilustra as alterações nos endereços dos objetos após duas coletas de lixo, cópia de objetos entre sobreviventes e promoção de um objeto para a geração estável. A mudança é completamente transparente ao programador. Como Java não realiza aritmética de ponteiros, não existe risco algum das mudanças de endereço causarem algum defeito em um programa.

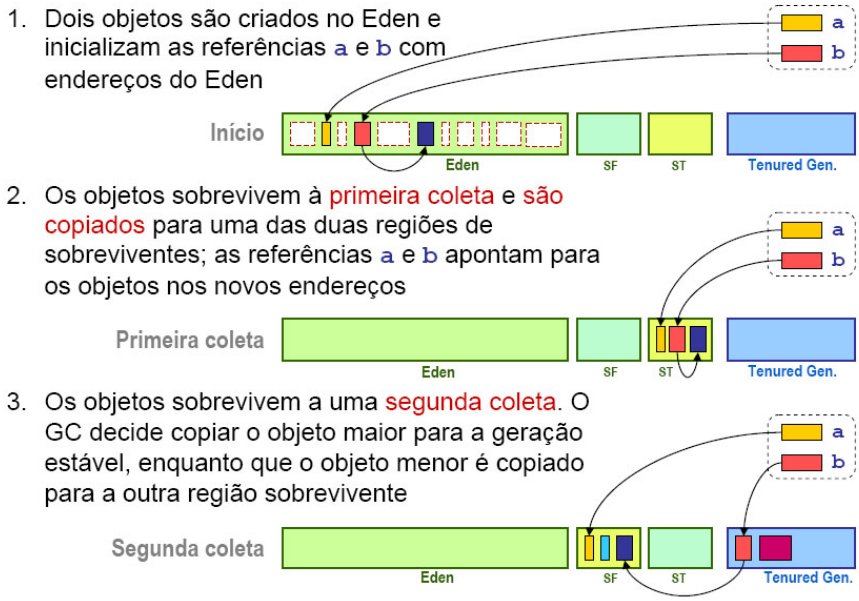

Figura 35 – A dança das referências: os endereços das referências mudam várias vezes entre as coletas, mas isto é totalmente transparente ao programador.

Geração estável

A *geração estável* (ou velha) consiste principalmente de objetos que sobreviveram a várias coletas menores, sendo copiados várias vezes de um espaço sobrevivente para o outro. O algoritmo de coleta de lixo decide quando

promover um objeto. Basicamente, depende de quantas vezes o objeto foi copiado. O valor pode variar mas tem um teto. Se sobrevive a um certo número de coletas, um objeto é necessariamente promovido, mas pode ser promovido antes. Objetos jovens que recebem referências de objetos estáveis podem ser emancipados para evitar ponteiros entre gerações, objetos que não cabem nos sobreviventes podem ser promovidos na primeira cópia e um objeto muito grande que não cabe no *Éden* pode ser criado diretamente na geração estável.

A geração estável pode estar sujeita à fragmentação. Isto depende do algoritmo de coleta de lixo usado. No coletor serial não ocorre fragmentação devido ao uso do algoritmo *mark-compact*, mas coletores paralelos podem apresentar esse problema.

No coletor serial, uma coleta maior ou *completa* acontece quando a região estável enche. Coletas menores vão gradualmente enchendo a região estável. Quando a geração estável está cheia, é executada uma coleta envolvendo todos os objetos de todas as gerações. Uma coleta maior sempre demora bem mais que uma coleta menor, porém é menos freqüente e pode nunca acontecer.

A coleta maior pode acontecer antes se o algoritmo do coletor escolhido for incremental ou se uma coleta menor não for possível devido à falta de espaço. Isto pode acontecer, por exemplo, se os sobreviventes estiverem cheios e houver mais objetos ativos no *Éden* que caberiam na região estável.

Geração permanente

A geração permanente consiste de memória alocada por processos que não estão relacionados à criação de objetos como:

- Carga de classes (*ClassLoader*),
- Área de métodos (área de código compilado),
- Classes geradas dinamicamente (JSP e reflexão),
- Objetos nativos (JNI).

As coletas de lixo na geração permanente são muito raras. Quando acontecem usam a algoritmo *mark-sweep* (com compactação quando cheio). Pode-se desligar a coleta de lixo nesta geração usando a opção `-Xnoclassgc`.

É comum haver duplicação de classes na geração permanente quando se usa múltiplas máquinas virtuais em uma mesma máquina rodando aplicações de longa duração[10].

A geração permanente não faz parte do *heap* total cujo tamanho é controlado pelas opções *–Xmx* e *–Xms* da máquina virtual. Para dimensioná-la é preciso usar opções próprias como *–XX:MaxPermSize* e *–XX:PermSize* (opções usadas no *HotSpot*).

---

[10] Em sistemas MacOS, que possuem várias aplicações de desktop implementadas em Java, isto motivou a criação de uma geração adicional compartilhada pelas máquinas virtuais. Chama-se geração "imortal". É na verdade uma parte da geração permanente que é compartilhada e não afetada por coleta de lixo. Essa geração não é a mesma coisa que o *compartilhamento de classes realizado* pelas *JVM Client*, disponível na maior parte dos sistemas (veja *capítulo 11*).

6. Configuração de memória

Existem diversas opções da máquina virtual *HotSpot* para configurar o tamanho das gerações, do *heap* total e da geração permanente. Também é possível determinar o tamanho das pilhas de cada *thread*. Os ajustes podem ser realizados de forma *absoluta*, com valores em bytes; de forma *relativa*, com percentagens ou relações de proporcionalidade 1:n; e de forma *automática* usando análise ergonômica baseada em metas de performance.

Definição de limites absolutos para o heap

O *heap total* consiste do espaço ocupado na geração jovem mais o espaço ocupado pela geração estável. Não inclui a geração permanente, que é configurada à parte.

Para alterar o limite *máximo* do *heap* total, utilize a opção:

```
-Xmx<número>[k|m|g]
```

que define um limite superior ao *heap* total. Se essa opção não estiver presente, o sistema usará valores *default*. O *default* para máquinas cliente é 64 MB. Para máquinas servidoras, o valor é calculado via ergonômica: ¼ memória física ou 1GB. Valores *default* podem variar entre plataformas, fabricantes, implementações e versões diferentes de máquinas virtuais, portanto não se deve depender deles em situações críticas.

O valor *inicial* do heap é determinado pela opção

```
-Xms<número>[k|m|g]
```

que também estabelece o mínimo de espaço que deve ser alocado para o *heap* (mesmo que não seja usado). O *default* em máquinas cliente é 4MB e em máquinas servidores 1/64 memória física ou 1 GB.

Por exemplo, para um *heap* ocupando entre 128 e 256 megabytes, pode-se chamar o interpretador da forma:

```
java -Xmx256m -Xms128m ...
```

O exemplo seguinte configura a ocupação do *heap* em exatamente 256 megabytes (tamanho fixo). Essa configuração evita que a JVM tenha que calcular se deve ou não deve aumentar o *heap*.

```
java -Xmx256m -Xms256m ...
```

A *geração permanente* (onde classes compiladas são guardadas) não faz parte do *heap* total. Pode ser necessário aumentá-la em situações em que há muito uso de reflexão e geração de classes (ex: aplicações EJB e JSP). Duas opções existem no *HotSpot* da Sun para configurar seus limites:

```
-XX:PermSize=<valor>[k,m,g]
```

que define o tamanho inicial da geração permanente, e

```
-XX:MaxPermSize=<valor>[k,m,g]
```

que define o seu tamanho máximo. Caso o sistema precise de mais espaço que o permitido nesta opção, acontecerá *OutOfMemoryError*.

Chamando o interpretador Java com as opções abaixo, serão alocados inicialmente 32 megabytes para a geração permanente, e o espaço será expandido até o limite de 64 megabytes, se for preciso.

```
java -XXPermSize=32m -XX:MaxPermSize=64m ...
```

Uma *geração jovem* menor causa coletas pequenas mais freqüentes. Uma geração jovem maior é mais eficiente pois coletas serão mais raras, mas se ocorrerem irão demorar mais, além de reduzirem o espaço para a geração velha, o que pode causar coletas demoradas freqüentes.

Para alterar o tamanho inicial e máximo da geração jovem, existem também duas opções. A opção

```
-XX:NewSize=<valor>[k,m,g]
```

define o tamanho inicial da geração jovem, e

```
-XX:MaxNewSize=<valor>[k,m,g]
```

o seu limite superior. Como o limite máximo do *heap* é fixo, aumentar a geração jovem tem o efeito de reduzir a geração estável.

Se a máquina virtual for chamada com as opções abaixo

```
java -XX:NewSize=64m -XX:MaxNewSize=64m ...
```

a geração jovem terá um tamanho fixo de 64 megabytes. Para especificar um valor inicial de 64 megabytes e permitir que a geração jovem varie até ocupar 128 megabytes, deve-se chamar a máquina virtual da forma:

```
java -XX:NewSize=64m -XX:MaxNewSize=128m ...
```

Tamanho fixo da pilha

Cada *thread* tem uma pilha. A pilha é dividida em quadros (*frames*) para cada método cujos dados não são compartilhados. Uma pilha grande evita *StackOverflowError*, porém se a aplicação tiver muitos *threads* (ex: servidores) a memória total pode ser consumida de forma ineficiente levando a *OutOfMemoryError*. Nessas situações pode-se reduzir o tamanho da pilha até um tamanho ideal que não cause *StackOverflowError*.

O tamanho de cada pilha pode ser definido com a opção

```
-Xss=<valor>[k,m,g]
```

que define um tamanho fixo para a pilha de cada *thread*. Por exemplo, iniciar a máquina virtual com a opção

```
java -Xss128k ...
```

altera o tamanho da pilha de cada *thread* para 128 quilobytes.

A *figura 36* resume os ajustes de tamanho fixo que podem ser realizados através de opções da máquina virtual HotSpot.

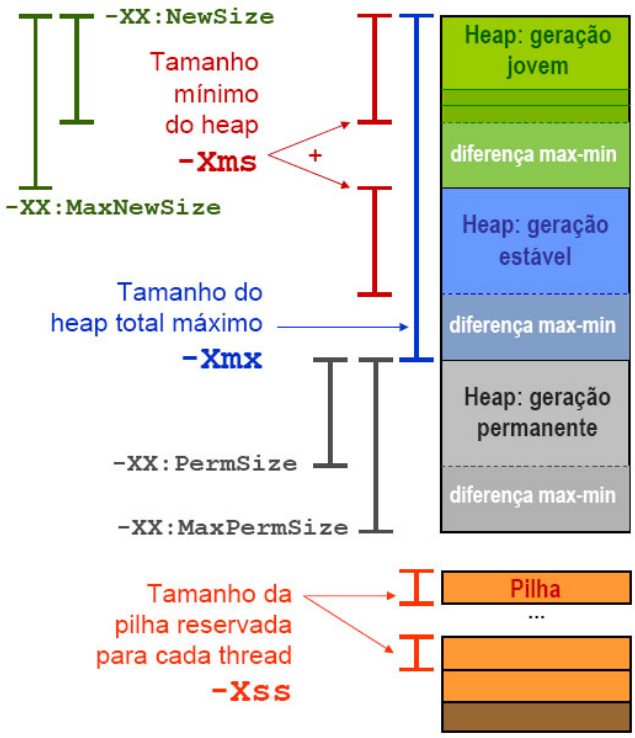

Figura 36 - Resumo: ajustes de memória; valores de ajuste absolutos

Variação do tamanho do heap

A ocupação do *heap* na máquina virtual *HotSpot* varia de tamanho durante a execução de uma aplicação. Os valores fixos atribuídos ao *heap* inicial e *heap* máximo são valores limite: a máquina virtual irá procurar utilizar a memória da forma mais eficiente, só ocupando o espaço realmente necessário. Existem cinco medidas associadas ao tamanho do *heap*; duas fixas e três variáveis:

- *Heap inicial*: é o espaço reservado para a aplicação quando a máquina virtual inicia. O valor usado será o *default* ou fornecido pela opção –Xms.
- *Heap máximo*: é o maior espaço que pode ser reservado para a máquina virtual. O valor usado será o *default* ou o fornecido pela opção –Xmx.
- *Heap reservado (committed)*: inicialmente é igual ao *heap* inicial, mas à medida em que a aplicação consome memória, aumenta gradualmente até o valor máximo estipulado pelo *heap máximo*. Uma coleta de lixo pode diminuir o *heap reservado* se o *heap utilizado* cair abaixo de certo limite.
- *Heap utilizado*: é o espaço realmente ocupado pelos objetos. É sempre menor que o *heap reservado*. Uma coleta de lixo reduz o *heap utilizado*.
- *Heap disponível (free)*: é a diferença entre o *heap reservado* e *heap utilizado*.

Se os valores para *heap inicial* e *máximo* forem iguais, o *heap reservado* terá o mesmo valor e não irá mudar. Se forem diferentes, a máquina virtual irá acompanhar o aumento e diminuição do *heap utilizado* e ajustar o espaço reservado em cada caso. O aumento ou diminuição ocorre quando o *heap utilizado* atinge um determinado valor.

É possível alterar os valores que forçam alteração do *heap reservado* através de duas opções da máquina virtual, que estabelecem as percentagens do *heap*

utilizado em relação ao *heap reservado* que irão causar as mudanças dentro da faixa *-Xms/-Xmx*. Se *–Xms* for igual a *–Xmx* elas não serão consideradas.

A opção:

```
-XX:MinHeapFreeRatio=<percentagemMinima>
```

define a percentagem mínima do *heap reservado* que precisa estar disponível após uma coleta. O *default* é geralmente 40% na *Client JVM*. Se após uma coleta o *heap disponível* não corresponder a no mínimo esse valor, a máquina virtual aumentará o espaço do *heap reservado* proporcionalmente até alcançar a meta ou atingir o limite.

A percentagem máxima do *heap reservado* que pode estar disponível após uma coleta pode ser alterado com a opção:

```
-XX:MaxHeapFreeRatio=<percentagemMaxima>
```

O valor *default* geralmente é 70% na *Client JVM*. Se após uma coleta o *heap reservado* for maior que este valor, a máquina virtual irá reduzir o espaço reservado do *heap* até alcançar a meta ou atingir o limite mínimo.

As *figuras 37 e 38* ilustram o crescimento e diminuição do *heap reservado* para a seguinte configuração de linha de comando:

```
java -Xms30m -Xmx100m
     -XX:MinHeapFreeRatio=50 -XX:MaxHeapFreeRatio=60 ...
```

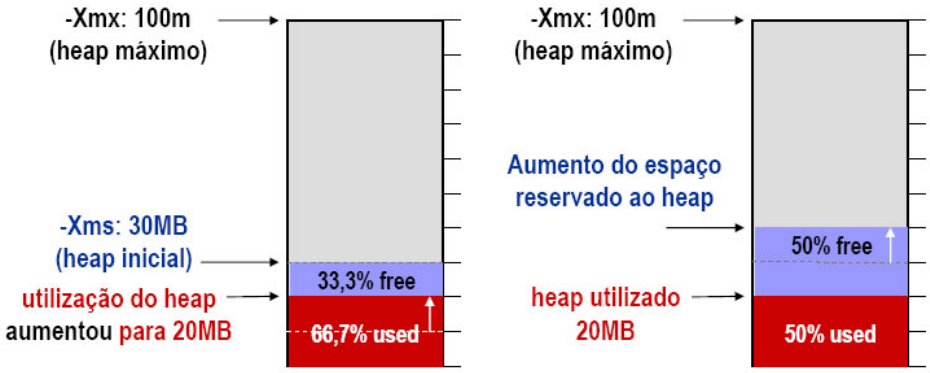

Figura 37 – Aumento do heap reservado após aumento do heap utilizado que passou a ocupar um espaço maior que o estabelecido em MinHeapFreeRatio, fazendo o heap disponível cair abaixo de 50%.

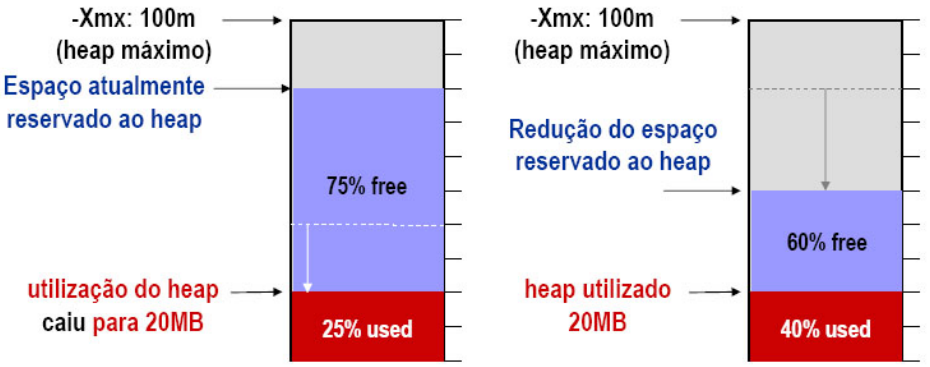

Figura 38 – Redução do heap reservado após a redução do heap utilizado que passou a ocupar um espaço inferior ao MaxHeapFreeRatio, fazendo com que houvesse mais de 60% de heap disponível.

O ajuste do tamanho do *heap* é o fator que tem o maior impacto na performance da coleta de lixo geral. Aplicações que variam o *heap* com freqüência poderão melhorar sua performance ajustando os parâmetros de redimensionamento para adequar-se ao comportamento da aplicação. Pode-se também definir *-Xms* and *-Xmx* para o mesmo valor, e assim evitar o redimensionamento a cada coleta. Isto evita que a máquina virtual recalcule o uso do *heap* a cada coleta. Por outro lado ela não compensará escolhas malfeitas.

Proporção geração jovem/estável

Se um *heap* tem tamanho limitado, o aumento da geração jovem diminui a geração estável. As opções *–XX:NewSize* e *–XX:MaxNewSize* definem um tamanho absoluto para a nova geração. O tamanho do espaço reservado à geração jovem irá variar proporcionalmente ao *heap total* dentro desses limites. Uma outra alternativa é usar a opção relativa:

```
-XX:NewRatio=n
```

que define a proporção 1:*n* entre a geração jovem e a geração estável. A geração jovem passará a ocupar 1/(*n*+1) do espaço total do *heap*.

Na *Client JVM* a relação *default*[11] é 1:**8** (*NewRatio*=8) e a geração jovem ocupará 1/9 do *heap*, como mostra a *figura 39*.

Figura 39 – Distribuição do heap para NewRatio=8.

Na *Server JVM* a relação *default* é 1:2 (*NewRatio*=2) e a geração jovem ocupará 1/3 do *heap*, como mostra a *figura 40*.

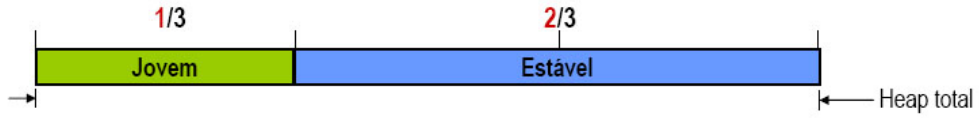

Figura 40 – Distribuição do heap para NewRatio=2.

Por exemplo, se for o interpretador Java for chamado da forma

```
java -XX:NewRatio=3 ...
```

n será 3, a relação será 1:3 e a geração velha terá 3 vezes o tamanho a geração jovem, ocupando 75% do espaço. A geração jovem ocupará 25% do *heap*.

Cada coleta na geração jovem pode necessitar de alocações na geração estável. Esse espaço precisa ser garantido ou a coleta irá falhar. Na hipótese de uma coleta improvável em que todos os objetos estão ativos, será preciso reservar memória livre suficiente na geração estável para acomodar *todos* os objetos existentes no *Éden* e espaços sobreviventes. A *Young Generation Guarantee* [Sun 05] (YGG) é definida como a reserva prévia de espaço na geração estável

[11] Valores *default* dependem do servidor usado.

nessas situações. É um espaço que poderá nunca ser usado, mas é uma garantia necessária nos coletores seriais.

Idealmente, os objetos serão copiados do *Éden* e sobrevivente origem para o sobrevivente destino, mas não há garantia que todos os objetos caberão no sobrevivente. O espaço reservado pelo YGG pressupõe o pior caso: é igual ao tamanho do *Éden* mais os espaços ocupado pelos objetos no sobrevivente origem. Não havendo como reservar esse espaço, ocorrerá uma coleta completa.

Devido à YGG, um *Éden* maior que metade do espaço do *heap reservado* inutiliza as vantagens da *Generational GC*: apenas coletas maiores iriam ocorrer. Como o *Éden* ocupa a maior parte da geração jovem, a opção *-XX:NewRatio* tem forte influência nessa configuração.

Uma *geração jovem maior* terá como conseqüência menos coletas menores mas pausas maiores se ocorrerem. Uma geração estável menor como devido ao aumento da geração jovem também aumentará a freqüência de coletas maiores, já que o espaço será preenchido mais rapidamente. Uma *geração jovem menor* causará maior freqüência de coletas menores, principalmente no início da aplicação. Mas as pausas serão curtas, já que haverá poucos objetos ativos e o algoritmo tem custo proporcional aos objetos vivos. Uma conseqüência da geração jovem menor é uma geração estável maior, causando o adiamento de coletas maiores, mas que dependendo da aplicação, podem nunca ocorrer.

Para escolher o melhor tamanho para a geração jovem é preciso analisar a distribuição dos objetos alocados e estabilizados (*tenured*) durante a vida da aplicação. Para a maior parte das aplicações, as configurações *default* são suficientes. A menos que haja pausas muito longas ou coletas maiores excessivas, deve-se alocar o máximo de memória à geração jovem, observando se a garantia da geração jovem (YGG) continua valendo: não aumente além da metade do espaço usado do *heap*. A alocação de objetos pode ocorrer em paralelo, portanto deve-se aumentar o tamanho da geração jovem à medida em que houver mais processadores, para otimizar a eficiência do sistema.

Proporção Éden/sobreviventes

Utilizando a opção

```
-XX:SurvivorRatio=n
```

é possível alterar a proporção entre os sobreviventes e o *Éden*. O número refere-se ao espaço ocupado pelos *dois* espaços sobreviventes, ou seja, uma relação 1:*n* reserva 1/(*n*+2) do espaço da geração jovem para *cada* sobrevivente. O *default*[12] para *n* é 25 (1/27) para o *Client JVM* e 30 (1/32) para o *Server JVM (figura 41)*.

Figura 41 – Proporção entre o Éden e geração jovem para n=25.

[12] Varia entre plataformas e versões do *HotSpot*.

Se o interpretador Java for chamado da forma

```
java -Xmx=100m -XX:NewRatio=3 -XX:SurvivorRatio=3
```

a distribuição do *heap máximo* será de 15MB para o *Éden*, 5MB para cada sobrevivente e 75MB para a geração estável, como ilustrado na *figura 42*.

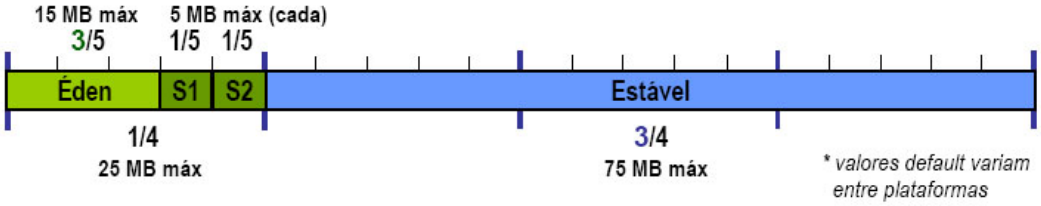

Figura 42 – Exemplo de distribuição do heap entre gerações estável, jovem, Éden e sobreviventes.

Um sobrevivente muito grande causa desperdício de espaço, já que um dos espaços estará sempre vazio. Também reduzirá o tamanho do *Éden* que terá coletas mais freqüentes. Se for muito pequeno poderá encher muito rápido ou não conseguir sequer acomodar os objetos, que serão copiados diretamente para a geração estável aumentando a freqüência das coletas maiores.

A cada coleta, a JVM define o número de vezes que um objeto pode ser copiado entre sobreviventes antes de ser promovido à geração estável. Este valor é chamado de *tenuring threshold*. Esse comportamento pode ser modificado com duas opções:

```
-XX:TargetSurvivorRatio=percentagem
```

é a percentagem do espaço sobrevivente que deve estar cheia antes da coleta. O *default* é 50%, que mantém os sobreviventes cheios pela metade. Valores menores irão fazer com que os objetos sejam copiados mais vezes e alcancem o seu *tenuring threshold* mais rapidamente.

O número máximo de cópias necessária para que ocorra a promoção de um objeto para a geração estável pode ser modificado através da opção:

```
-XX:MaxTenuringThreshold=n
```

O valor *default* para *n* é 31. Se *n* for zero os objetos sempre serão promovidos na primeira coleta. Mesmo com *n* maior que zero um objeto pode ainda ser promovido na primeira coleta ou após menos de *n* coletas, pois *n* representa um valor limite. A promoção após *n* cópias é garantida, mas pode ocorrer antes.

Para obter informações sobre a distribuição dos objetos estáveis, a opção:

```
-XX:+PrintTenuringDistribution
```

gera um relatório dos objetos na geração estável distribuídos por idade, e contém, dentre outras informações, o *tenuring threshold* de cada um. Os dados listados são importantes para auxiliar no redimensionamento das gerações.

7. Seleção do coletor de lixo

A coleta de lixo nos servidores *HotSpot* é realizada por uma coleção de algoritmos diferentes que atuam em diferentes partes do *heap*. Uma forma de evitar ter que configurar essas combinações é escolher uma máquina virtual através das opções *–server* ou *–client*. Existem quatro coletores pré-configurados que podem ser selecionados através de opções da máquina virtual.

O coletor serial, ou *serial collector*, é *default* no *Client JVM*. Pode ser ativado ou desativado (se necessário) através das opções

`-XX:+UseSerialGC` *ou* **`-XX:-UseSerialGC`**

O coletor de alta eficiência ou *throughput collector* é *default* no *Server JVM*. Pode ser ativado ou desativado (se necessário) através das opções

`-XX:+UseParallelGC` *ou* **`-XX:-UseParallelGC`**

O coletor semi-concorrente de pausas curtas, chamado de *mostly-concurrent low pause collector* ou ainda *concurrent mark-sweep collector* (CMS) pode ser ativado ou desativado através das opções

`-XX:+UseConcMarkSweepGC` *ou* **`-XX:-UseConcMarkSweepGC`**

Finalmente o coletor incremental de pausas curtas, chamado de *incremental low pause collector* ou ainda de *train collector* pode ser ativado ou desativado através das opções

`-XX:+UseTrainGC` *ou* **`-XX:-UseTrainGC`**

Cada coletor usa uma combinação de algoritmos disponíveis otimizados para situações distintas. É possível configurar e combinar algoritmos diferentes, mas não misturar os coletores pré-configurados. Há opções de configuração próprias que permitem que os coletores compartilhem algumas funcionalidades. A ativação explícita de um coletor implica na desativação do coletor *default*. As opções de desativação não são necessárias. Combinações entre as opções de ativação geralmente são ilegais.

Algoritmos utilizados

Algoritmos diferentes são usados para as diferentes gerações de cada coletor. A geração jovem possui três algoritmos:

1. Coletor serial (*copying algorithm*) - *default*
2. Coletor paralelo (*concurrent copying algorithm*)
3. Coletor paralelo de alta eficiência (*scavenging algorithm*)

A geração estável possui outros três:

4. Coletor *mark-compact* serial - *default*
5. Coletor *mark-sweep* concorrente
6. Coletor *train* incremental

Os coletores pré-configurados combinam esses algoritmos e permitem ajustes e alterações na configuração *default*.

Todos os três algoritmos usados para coletar a *geração jovem* são algoritmos de cópia. No coletor *default*, todos os *threads* são interrompidos e um *thread* executa o algoritmo de cópia serial. Nos coletores paralelos, todos os *threads* são interrompidos ao mesmo tempo (comportamento *stop-the-world*) e um ou mais *threads* executam um dos dois algoritmos de cópia concorrentes. A pausa provocada em um coletor paralelo diminui com o aumento do número de processadores paralelos, como mostrado na *figura 43(a)*.

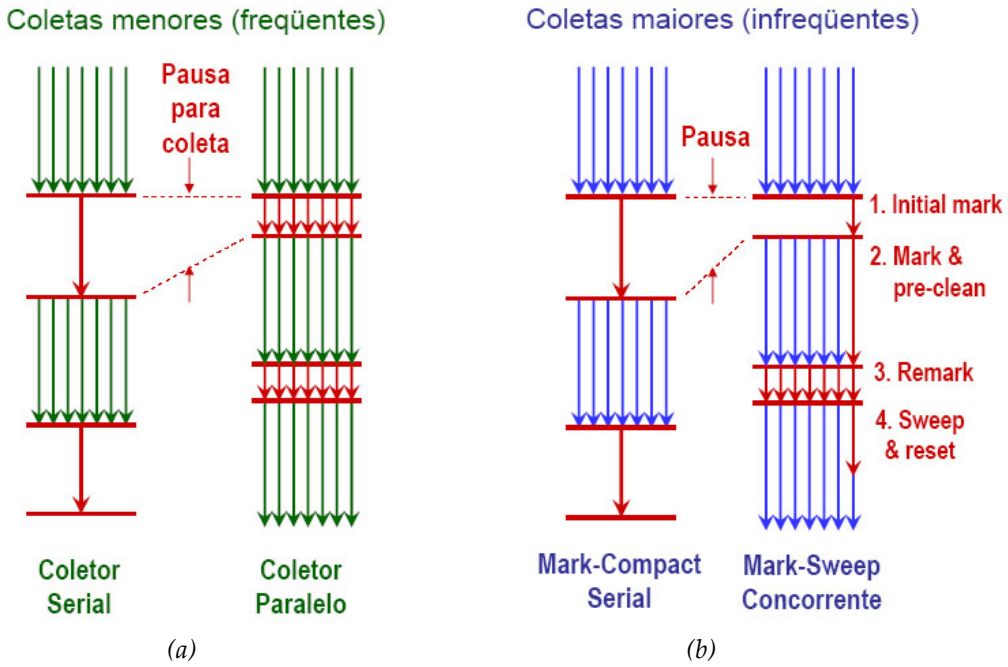

Figura 43 – Coletores de lixo: (a) das gerações jovens; (b) das gerações estáveis

Na *geração estável*, o coletor serial *mark-compact* compacta o espaço a cada coleta. O coletor concorrente *mark-sweep* não faz compactação, que só é realizada se espaço acabar. Como conseqüência, a alocação na geração estável, que é realizada durante coletas menores, será mais demorada e irá aumentar as pausas das coletas menores. O coletor concorrente faz a maior parte do trabalho em paralelo, como mostra a *figura 43(b)*, dividindo as tarefas em quatro etapas[13]:

1. *Initial mark*: marca todos os objetos diretamente alcançáveis de fora do *heap*, parando a aplicação e fazendo a marcação em um *thread*.
2. *Mark/pre-clean*: marca os objetos alcançáveis recursivamente a partir das referências achadas na primeira fase. Esta fase é realizada em paralelo por um *thread* e pode deixar de marcar alguns objetos, já que a aplicação pode estar criando objetos enquanto eles são marcados.
3. *Remark*: pára a aplicação e usa todos os *threads* disponíveis para revisitar todos os objetos e marcar os que escaparam de ser marcados na fase *mark*. A pausa é minimizada usando vários *threads*.

[13] A rigor são seis, mas as especificações descrevem quatro, combinando *mark/pre-clean*, e *sweep/reset* que são sempre realizadas em seqüência.

4. *Sweep/reset*: enquanto a aplicação executa em paralelo, usa um *thread* para varrer do *heap* os objetos inalcançáveis.

Coleta incremental

Usado apenas na geração estável, o *algoritmo do trem*, ou *train algorithm*, realiza *coleta incremental* em paralelo com a execução da aplicação. Quando este algoritmo é ativado, a geração estável é dividida em blocos de memória de 512kB, chamados de *vagões*, que são ordenados de acordo com a ordem de criação em *trens*, criados a cada coleta. Objetos sobreviventes de coletas na geração jovem são alocados nos vagões de trens existentes, ou em vagões novos engatados nesses trens se não couberem. São coletados sempre os vagões e trens mais antigos e alocações podem ser feitas em qualquer trem que não esteja sendo coletado. Durante uma coleta, objetos em um vagão que está será coletado são transferidos para outro trem até que o vagão só contenha lixo. A coleta remove vagões-lixo sem parar a aplicação.

A *figura 44* ilustra o *heap* da geração estável dividido em trens e vagões.

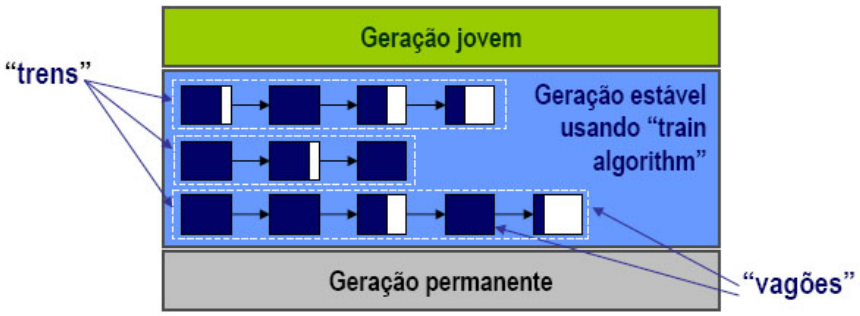

Figura 44 – Divisão do heap da geração estável em trens e vagões no uso do coletor incremental.

O *coletor do trem* não é um algoritmo de tempo real[14], pois não consegue evitar totalmente a não ocorrência de pausas, nem determinar um limite máximo para elas, nem saber quando ocorrem e nem como impedir que todos os *threads* parem ao mesmo tempo. É um algoritmo que coloca a redução das pausas acima de todas as outras prioridades. Dos algoritmos disponíveis para a geração estável é menos eficiente e só deve ser usado quando a ausência total de pausas realmente for essencial.

Para ativar a coleta incremental do *heap* usando este coletor há duas opções que fazem a mesma coisa:

`-XX:+UseTrainGC` *ou* **`-Xincgc`**

[14] Não existe *nenhum* algoritmo que seja verdadeiramente de tempo real (*hard real time*) no *HotSpot*. A maior parte das soluções propostas e testadas até o momento tem um alto custo sobre a eficiência. Os algoritmos usados em máquinas virtuais comuns são chamados de *soft real time*. As APIs e máquinas virtuais Java que implementam a especificação de tempo real (*JSR-1: Real Time Specification for Java*) requerem *hard real time* e não usam a mesma arquitetura do *HotSpot*. Utilizam uma parte do *heap* que *não faz coleta de lixo* (chamada de geração *imortal*) e outra onde *o programador explicitamente gerencia o ciclo de vida de objetos* através de escopos (devolve parte da responsabilidade de gerência de memória ao programador.)

Como a coleta afeta apenas a geração estável, a geração jovem continuará sendo coletada usando o coletor de cópia serial *default*. É possível trocá-lo por um coletor paralelo na geração jovem usando a opção *–XX:+ParNewGC*, para que toda a coleta seja realizada em paralelo.

O coletor incremental parou de ser atualizado desde a versão 1.4.2 do *HotSpot* e poderá não estar presente em versões futuras.

Opções de paralelismo

As opções descritas a seguir permitem são usadas em conjunto com a escolha de um determinado coletor.

Os coletores paralelos que concentram suas otimizações na geração estável utilizam por *default* o coletor serial para a geração jovem. Isto pode ser mudado com a opção:

```
-XX:+UseParNewGC
```

Que faz com que a máquina virtual use um coletor de cópia paralelo **(2)** para a geração jovem. Esta opção só pode ser usada com os coletores que não especificam um algoritmo para a geração jovem: *XX+:UseTrainGC* ou *XX:+UseConcMarkSweepGC*. Não é compatível com *XX+:UseParallelGC* que tem um algoritmo próprio para a nova geração.

O nível de paralelismo dessa operação pode ser controlada com a opção

`-XX:ParallelGCThreads=n` *(default: número de. threads disponíveis)*

que especifica quantos *threads* o coletor usará para coletar a geração jovem.

Várias opções são utilizadas apenas no coletor CMS (*Concurrent Mark-Sweep*) e requerem o uso da opção *-XX:+UseConcMarkSweepGC* uma vez que este coletor não é *default* em nenhuma configuração de máquina virtual.

A opção

```
-XX:+CMSParallelRemarkEnabled
```

usada apenas no coletor CMS, faz com que a etapa de remarcação (*remark*) seja realizada em paralelo usando quantos *threads* estiverem disponíveis, diminuindo as pausas. Apesar de referir-se apenas à geração estável, requer o uso da opção *-XX:+UseParNewGC* e também de *-XX:+UseConcMarkSweepGC*.

Uma coleta concorrente deve sempre iniciar e terminar *antes* que a geração estável fique cheia. Isto difere do comportamento do coletor serial que inicia quando a geração enche. Para saber quando iniciar, o coletor mantém estatísticas para estimar o tempo que falta antes da geração estável encher e o tempo necessário para realizar a coleta. As suposições são conservadoras. Uma coleta concorrente também iniciará assim que a ocupação da geração estável passar de um certo limite. Este valor pode ser alterado com a opção

`-XX:CMSInitiatingOccupancyFraction=`*`n`*

onde *n* é a % do espaço ocupado antes da coleta (0-100). O valor inicial é aproximadamente 68% do *heap*.

É possível diminuir as pausas do CMS, através do seu modo incremental. As principais opções são:

`-XX:+CMSIncrementalMode` *(default: desabilitado)*

que habilita modo incremental,

`-XX:+CMSIncrementalPacing` *(default: desabilitado)*

que permite ajuste automático do ciclo (*pacing*) com base em estatísticas,

`-XX:CMSIncrementalDutyCycle=n` *(default: 50)*

que especifica a percentagem de tempo (0-100) entre coletas menores em que o coletor concorrente pode executar. Se o *pacing* automático habilitado, este é o valor inicial. Finalmente

`-XX:CMSIncrementalDutyCycleMin=n` *(default: 10)*

define a percentagem (0-100) que será o limite inferior do ciclo caso o *pacing* esteja habilitado.

Maiores detalhes sobre as várias outras opções do CMS podem ser encontrados na documentação oficial e nos artigos listados no final, particularmente [Gupta 02] e [Nagarajayya 02].

Como escolher um coletor de lixo?

Quando a escolha de um coletor de lixo importa para o usuário? Para muitas aplicações ele não faz diferença. O coletor de lixo previamente instalado e configurado na máquina virtual geralmente é suficiente. Ele realiza pausas de pouca duração e freqüência que em geral são desprezíveis. Mas, em sistemas grandes, essas pausas ou a CPU consumida pelo coletor podem ter importância significativa. Nessas situações, pode compensar escolher corretamente o melhor coletor de lixo e ajustá-lo.

Para maior parte das aplicações o coletor serial, ou *Serial GC*, é adequado. Os outros têm *overhead*, são mais complexos e podem piorar a performance de uma aplicação que realmente não precise deles. Se uma aplicação não necessita do comportamento especial de um coletor alternativo, deve usar o coletor serial. Em geral, computadores com menos de 2 gigabytes de memória RAM e menos de dois processadores executam bem aplicações típicas com um coletor serial. Em grandes aplicações com muitos *threads*, alto requerimento de memória, comportamento incomum, rodando em máquinas com *heaps* grandes e muitos processadores, o coletor serial provavelmente não será a melhor escolha. Neste caso, a escolha deve inicialmente recair sobre o coletor paralelo de alta eficiência, ou *Parallel GC*.

O *coletor paralelo de alta eficiência*, também chamado de *Parallel Collector* ou *Throughput Garbage Collector (TGC)* tem como objetivo a máxima eficiência com eventuais pausas. Aplicações que usam esse coletor raramente realizam coletas maiores, e quando realizam, não se incomodam muito se o sistema parar por

alguns segundos. Consideram importante que coletas menores sejam rápidas (são sempre realizadas em paralelo) e que a eficiência (taxa entre o tempo usado pela aplicação pelo tempo usado na coleta de lixo) seja a melhor possível. A performance de aplicações que usam este coletor aumenta proporcionalmente ao número de processadores existentes.

A segunda alternativa a se considerar é o *coletor concorrente de baixa latência (Low Latency Collector)* também chamado de *Mostly-Concurrent Collector* ou *Concurrent Mark-Sweep (CMS) garbage collector*. Seu objetivo é alcançar o mínimo de pausas em troca da eventual redução da eficiência. As coletas maiores, apesar de pouco freqüentes, podem impor pausas muito longas (principalmente com *heaps* grandes). Porém este coletor diminui as pausas da coleta maior, rodando em paralelo com a aplicação principal, que fica um pouco mais lenta. As pausas não são totalmente eliminadas. Ocorrem duas pequenas pausas, porém são da mesma ordem das pausas que ocorrem nas coletas menores. É indicado em aplicações que têm muitos dados de vida longa (grande geração estável) e requerimento de pausas mínimas. *Pode* haver vantagens para aplicações desse tipo mesmo em máquinas com um processador.

A *figura 45* compara duas configurações típicas usando esses dois coletores paralelos.

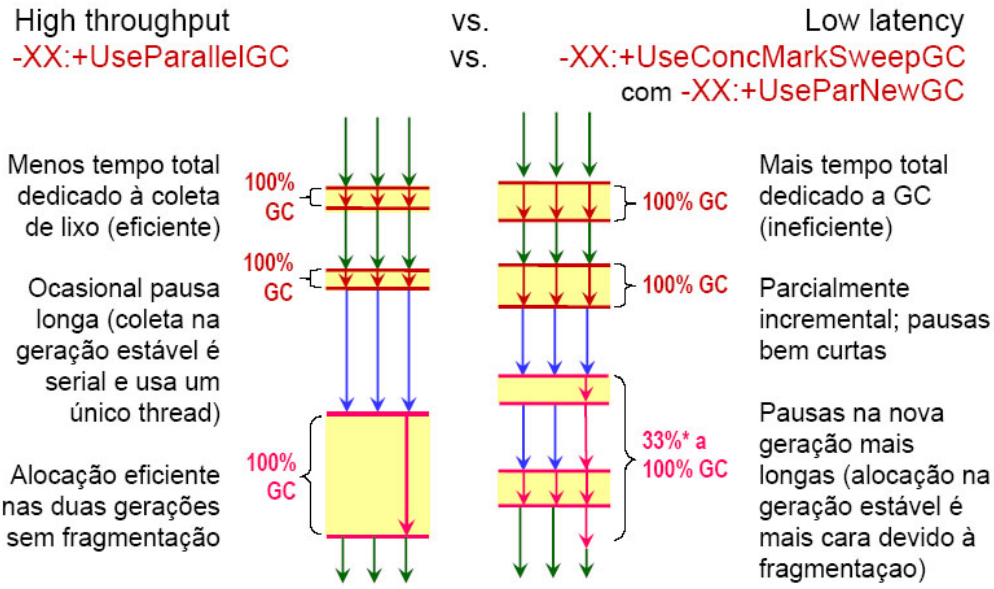

Figura 45 – Comparação entre algoritmos do coletor paralelo (TGC) e do coletor concorrente (CMS).

Por fim, há o *coletor incremental*, também chamado de *train garbage collector*. Como ele não elimina totalmente as pausas, sua vantagem em relação ao CMS em um sistema com muitos processadores poderá não ser grande, devido à sua baixa ineficiência. O CMS no modo incremental pode alcançar algumas de suas vantagens. Deve ser usado quando houver um requerimento de pausas mínimas, e quando uma eficiência mais baixa não fizer tanta diferença. Esse coletor poderá reduzir pausas em sistemas com menos (ou até um) processadores,

sendo uma alternativa possível em aplicações com requerimento de pausas mínimas que rodam em sistemas menores.

A *tabela 3* ilustra algumas das diferenças entre as opções de coletor de lixo existentes do *HotSpot* até o Java 5.0.

Coletor	Opção de ativação	Algoritmos utilizados	
		Geração Jovem	Geração Estável
Coletor serial	-XX:+UseSerialGC	Coletor de cópia serial (1) (*default*)	Coletor *mark-compact* serial (4) (*default*)
Coletor paralelo com eficiência máxima	-XX:+UseParallelGC	Coletor de cópia concorrente de alta eficiência (3)	
Coletor paralelo com pausas mínimas	-XX:+UseConcMarkSweepGC	Coletor *default* (1); Coleta concorrente (2) *pode* ser ativada com a opção -XX:+UseParNewGC	Coletor *mark-sweep* concorrente (5) (sem compactação)
Coletor incremental	-XX:+UseTrainGC		Algoritmo do trem (*train*) incremental (6)

Tabela 3 – Coletores de lixo usados no HotSpot.

8. Monitoração de aplicações

Para ajustar os parâmetros configuráveis da máquina virtual, é preciso realizar medições. Vários parâmetros da máquina virtual *HotSpot* fornecem informações úteis. Além disso, há ferramentas gráficas que mostram o comportamento da máquina virtual e sua alocação/liberação de memória.

É preciso saber: 1) o que ajustar e como ajustar; 2) o objetivo do ajuste – se menos pausas ou mais eficiência; e 3) as conseqüências do ajuste.

Pode-se também utilizar ajustes automáticos usando o recurso do Java 5.0 chamado de *Ergonomics*. Mesmo para usar ergonômica, é preciso conhecer como funciona o coletor de lixo.

As metas desejáveis geralmente envolvem obter menos pausas e mais eficiência de processamento (*throughput*). É preciso avaliar qual das duas é prioritária, já que melhorar uma pode piorar a outra.

Eficiência (capacidade de processamento) é a percentagem de tempo total não gasta com coleta de lixo. Isto inclui tempo gasto com alocação. Se a *eficiência* for maior que 95%, geralmente não vale a pena fazer ajustes na máquina virtual.

As *pausas* são o tempo em que uma aplicação parece não responder porque está realizando coleta de lixo. Em alguns sistemas interativos elas devem ser mínimas. Em sistemas que realizam processamento demorado elas são toleradas.

Como obter informações sobre as coletas

Pode-se obter informações sobre quando ocorrem coletas e como isto afeta a memória usando a opção

```
-verbose:gc
```

que imprime informações básicas sobre as coletas maiores e coletas menores. As estatísticas são redirecionadas para a saída padrão, mas usando a opção

```
-Xloggc:<arquivo>
```

junto com –verbose:gc os dados serão gravados no arquivo especificado. O formado dos dados é lido por várias ferramentas de análise de logs. Por exemplo, a chamada

```
java –verbose:gc –Xloggc:aplicacao.gc aplicacao.Main
```

Imprime informações de coleta da aplicação *Main* no arquivo de texto *aplicacao.gc*.

Uma saída típica de *-verbose:gc* (em uma grande aplicação servidora) está mostrada a seguir:

```
[GC      325407K->83000K(776768K), 0.2300771 secs]
[GC      325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

A saída mostra duas coletas menores e uma coleta maior. Os números antes e depois da seta (325.407K->83.000K) indicam o tamanho total de objetos alcançáveis antes e depois da coleta. Depois de pequenas coletas, a contagem

inclui objetos que não estão necessariamente alcançáveis mas que não puderam ser coletados. O número entre parênteses (776.768K) é o total de espaço disponível (*heap* total usado menos um dos espaços de sobreviventes, sem contar o espaço da geração permanente). No exemplo, as coletas menores levaram em média 0,24 segundos. A coleta maior levou quase dois segundos.

Pode-se imprimir mais informações com

```
-XX:+PrintGCDetails
```

que faz com que a máquina virtual mostre mais detalhes sobre a coleta de lixo, como variações sobre o tamanho das gerações após uma coleta. É útil para obter *feedback* sobre freqüência das coletas e para ajustar os tamanhos das gerações. Há mais detalhes porém não é completo:

```
java -XX:+PrintGCDetails

GC [DefNew: 64575K->959K(64576K), 0.0457646 secs]
196016K-133633K (261184K), 0.0459067 secs]]
```

Para obter mais informações é preciso acrescentar mais opções. Para informar o tempo transcorrido e distribuição de objetos durante a aplicação – importantes para tomada de decisões de ajuste, há duas opções:

```
-XX:+PrintGCTimeStamps
```

que imprime carimbos de tempo relativos ao início da aplicação, e

```
-XX:+PrintTenuringDistribution
```

que acrescenta ao relatório detalhes da distribuição de objetos transferidos para a área estável. Pode ser usado para estimar as idades dos objetos que sobrevivem à geração jovem e para descrever a vida de uma aplicação. O exemplo abaixo ilustra um uso típico dessas opções.

```
java -verbose:gc -XX:+PrintGCDetails
     -XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution ...

  5.350: [GC Desired survivor size 32768 bytes,
  new threshold 1 (max 31)
  age 1: 57984 bytes, 57984 total
  age 2: 7552 bytes, 65536 total
  756K->455K(1984K), 0.0097436 secs]
```

Monitoração com o *jconsole*

O próprio ambiente de desenvolvimento Java possui uma ferramenta simples que fornece informações gráficas sobre a memória usando a tecnologia JMX (*Java Management Extensions*): o *jconsole*. Para habilitar o agente JMX e configurar sua operação, é preciso definir algumas propriedades do sistema ao iniciar a máquina virtual. As propriedades podem ser passadas em linha de comando da forma

```
java –Dpropriedade ...
java –Dpropriedade=valor ...
```

Se um valor não for fornecido, a propriedade utilizará um valor *default* (se houver e se for aplicável).

As duas principais propriedades JMX da máquina virtual são

```
com.sun.management.jmxremote[=true|false]
com.sun.management.jmxremote.port=valor
```

A primeira habilita o agente JMX *localmente* e permite monitoração *local* através do conector JMX usado pela ferramenta *jconsole*. Se o valor for omitido, será considerado *true*. O valor *false* é o mesmo que omitir a propriedade.

A segunda propriedade habilita o agente *remoto* JMX. Permite monitoração remota através de um conector JMX de interface pública disponibilizada através de uma porta TCP/IP. O *valor* passado como argumento deve ser o número da porta. Esta opção poderá requerer outras propriedades[15].

Para habilitar o agente JMX para *monitoração local* é preciso primeiro executar a classe ou JAR da aplicação via JVM passando a propriedade *jmxremote*:

```
java –Dcom.sun.management.jmxremote pacote.MainClass
java –Dcom.sun.management.jmxremote –jar Aplicacao.jar
```

Depois, é preciso obter o número do processo JVM usando o *jps*[16]:

```
> jps
3560 Programa                 (máquina virtual)
3740 pacote.MainClass         (use este número!)
3996 Jps
```

Finalmente, inicia-se o *jconsole* com o número do processo. O comando deve ser iniciado pelo mesmo usuário que iniciou a aplicação:

```
> jconsole 3740
```

Para monitoração em tempo de produção, recomenda-se uso remoto (devido ao *overhead* da aplicação). Para configurar, é preciso obter uma porta de rede livre. A porta será usada para configurar acesso remoto via RMI. O sistema também criará no registro RMI um nome *jmxrmi*.

Além da porta de acesso, é preciso configurar propriedades de autenticação, dentre outras. Para configuração do acesso remoto e outras informações, consulte a documentação.

Para executar o *jconsole* para *monitoração remota*, deve-se informar o nome da máquina e número do processo a ser monitorado:

```
> jconsole alphard:3740
```

Se o número do processo for omitido no acesso local, o *jconsole* irá oferecer uma lista de processos ativos para escolha. A *figura 46* ilustra a tela do *jconsole*.

---

[15] Veja *tabela 1* em */docs/guide/management/agent.html* (documentação J2SE 5.0)

[16] Ferramenta distribuída no J2SDK.

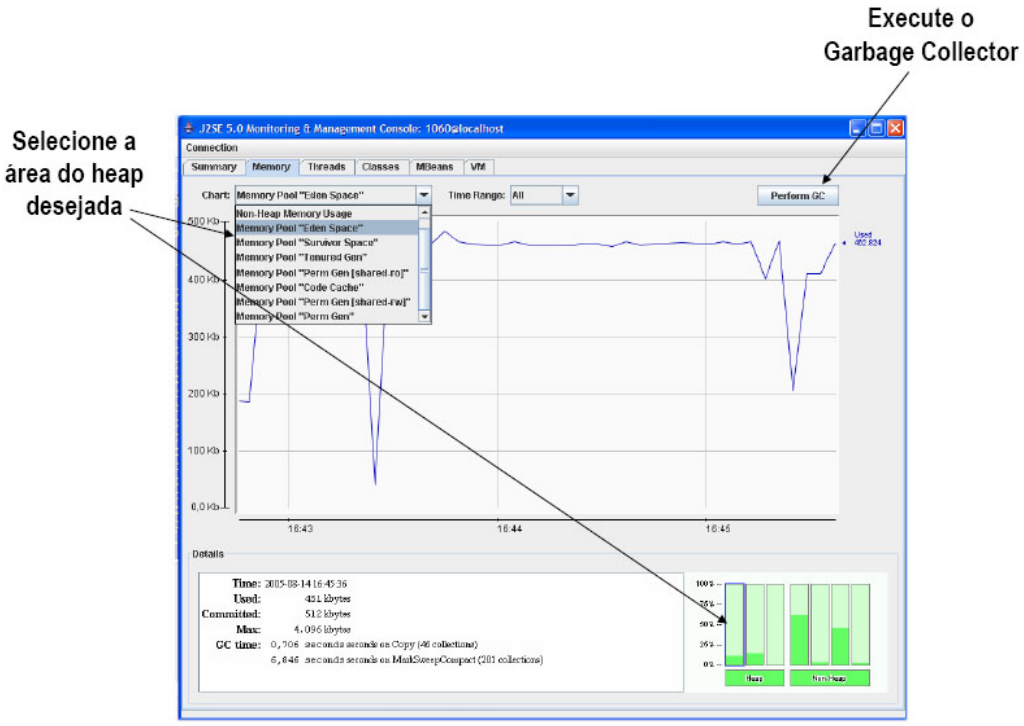

Figura 46 – Tela do jconsole*, mostrando aba de memória. Esta aba contém um gráfico mostrando utilização de qualquer geração em relação ao tempo, ícones ilustrando percentagem de utilização de memória nas gerações, e área de texto contendo quantidades de memória usada, reservada* (committed)*, máxima, tempos e quantidades das coletas maiores e menores.*

Monitoração com as ferramentas do *jvmstat*

Outra ferramenta disponível no ambiente de desenvolvimento é o *jstat* – uma ferramenta do pacote experimental *jvmstat*. Ela obtém dinamicamente estatísticas de uso das gerações, de compilação, de carga de classes em tempo real, sem precisar de *JMX*. Para executar, é preciso também ter o *id* do processo do JVM. A sintaxe é

```
> jstat <opções> jvmid
```

A *figura 47* ilustra um exemplo de execução do *jstat* e diferentes informações apresentadas.

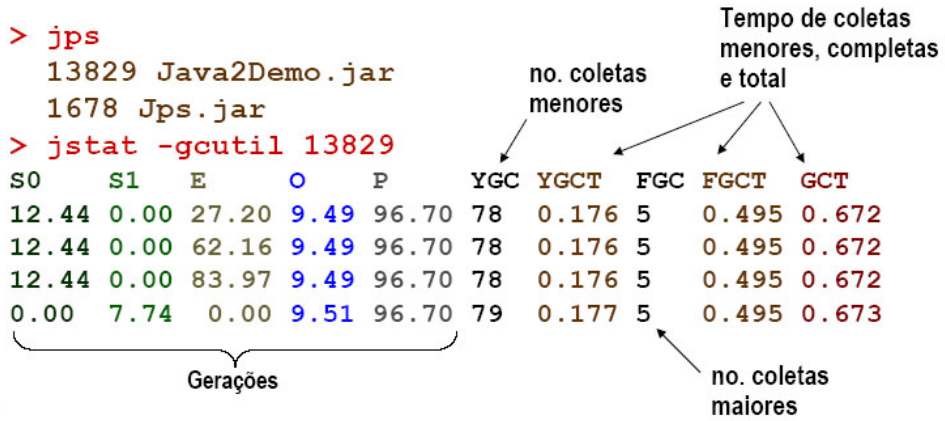

Figura 47 – Exemplo de utilização da ferramenta jstat.

O *Visual GC* é a ferramenta visual do pacote experimental *jvmstat*, mas não é distribuída com o SDK 5.0. É preciso fazer um download separado[17]. Mostra gerações, coletas, carga de classes, etc. graficamente.. Para rodar, é preciso obter o número do processo da aplicação a monitorar e o período de coleta de amostras (em milissegundos). O exemplo a seguir mostra como executar:

```
> jps
21891 Java2Demo.jar
1362 Jps.jar
> visualgc 21891 250
```

O *Visual GC* também permite monitoramento remoto. Para isto, é preciso obter o número do processo remoto e acrescentar o nome de domínio:

```
visualgc 21891@remota.com 250
```

A *figura 48* ilustra uma tela de saída do *Visual GC*.

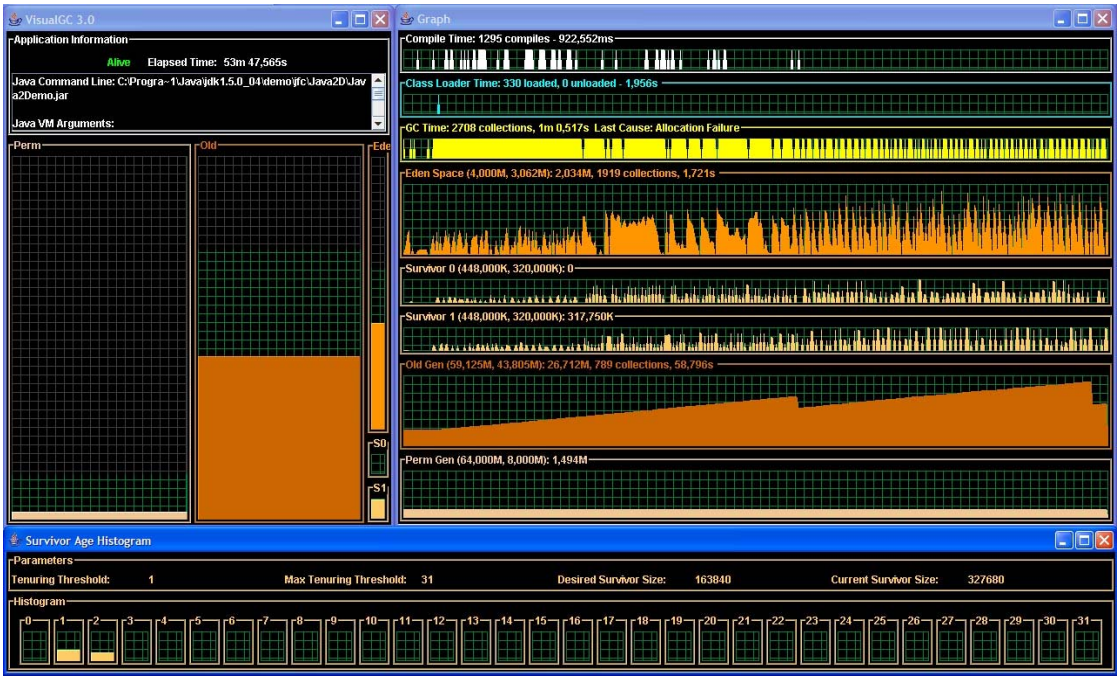

Figura 48 – Monitoração com o Visual GC. *A janela da esquerda contém dados estatísticos sobre o sistema e as coletas, e espaços ocupados pelas gerações permanente (*Perm*), estável (*Old*), e jovem (*Eden, S0 *e* S1*). A janela da direita (*Graph*) mostra tempos de compilação do JIT, tempos do* ClassLoader *para carregar classes, tempo total das coletas e distribuição do uso de memória nas gerações. A janela inferior é um histograma que destaca a idade dos objetos sobreviventes, até o valor máximo de* tenuring threshold, *que na imagem acima é 31.*

Outras ferramentas

Existem várias outras ferramentas de monitoração de memória que podem ser usadas para obter informações sobre gerações, coletas de lixo, uso do *heap*, promoção, etc. Algumas das mais populares são:

[17] http://java.sun.com/performance/jvmstat/visualgc.html

- *GC Portal*[18] é uma aplicação J2EE que gera gráficos e estatísticas. Requer instalação em um servidor e configuração. No site há uma versão online que pode ser usada livremente.
- *GCViewer*[19] analisa documentos de texto criados com *–Xloggc:arquivo*, mostra comportamento das gerações e outras informações. Pode também executar em tempo real (veja *figura 49*.)

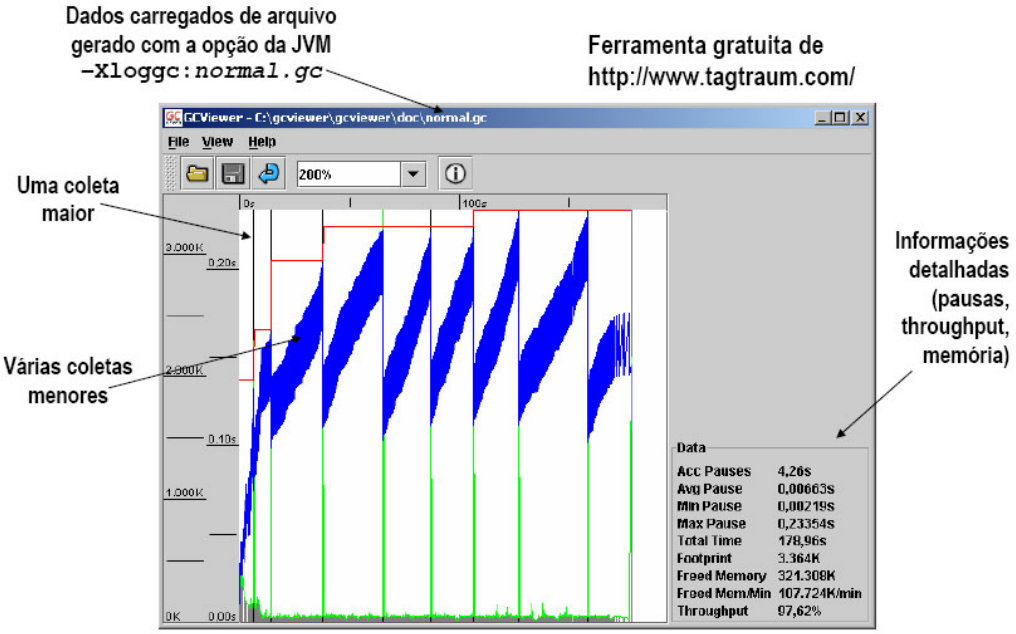

Figura 49 – Monitoração com o GCViewer.

Vários *profilers* comerciais e gratuitos também oferecem informações e capacidade de monitoração em tempo real da JVM. Alguns exemplos são os *profilers* comerciais: *JProbe*, *OptimizeIt*, *JProfiler* e também os gratuitos como: *NetBeans Profiler*, *Eclipse Profiler*, *JRat*, *Cougaar*, etc.

Seja qual for a ferramenta usada, é importante entender o significado das informações obtidas antes de tentar qualquer tipo de ajuste de performance.

[18] *java.sun.com/developer/technicalArticles/Programming/GCPortal*
[19] *www.tagtraum.com*

9. Ajuste automático: ergonomics

O objetivo da ergonômica é obter a melhor performance da JVM com o mínimo de ajustes de linha de comando. Busca obter, para uma aplicação, as melhores seleções de tamanho de *heap*, coleta de lixo e compilador de tempo de execução (JIT).

Os ajustes são baseados em duas *metas*: pausa máxima e capacidade de processamento mínima. Têm como alvo aplicações que executam em servidores grandes (*Server JVM*). Os primeiros ajustes automáticos são realizados na instalação e na execução. Na instalação, o sistema tentará descobrir se está em uma máquina de classe "servidor" ou "cliente". Se tiver pelo menos duas CPUs e pelo menos 2GB de memória física, será considerada servidora, caso contrário, é "cliente".

A máquina virtual instalada será *default*, passando a ser sempre usada a não ser que seja especificada outra, durante a execução, através das opções *–server* ou *–client*. Se uma aplicação estiver em uma máquina servidora, iniciará a *Server JVM*, que começa mais lentamente, mas com o tempo executa mais rapidamente. Se a aplicação estiver em máquina cliente, usará a *Client JVM*, que é configurada para melhor performance em ambientes cliente.

A parte mais interessante da ergonômica, porém, é o ajuste automático das gerações e parâmetros de coleta de lixo. O ajuste é baseado em *metas*. Para realizá-lo, o usuário especifica uma meta de *pausa máxima*, uma meta de *eficiência mínima* e configura uso mínimo/máximo de *memória* do *heap*. A partir dessas informações, o coletor de lixo ajusta automaticamente vários parâmetros para tentar alcançar as metas. Nem sempre consegue. Se não conseguir, o usuário poderá ajustar outros parâmetros até obter uma configuração aceitável. Os ajustes feitos pelo coletor incluem o tamanho e proporcionalidade entre a geração jovem, espaços sobreviventes, geração estável e outros valores, como a alteração da política de promoção para geração estável.

A ergonômica trabalha com *metas*, não com *garantias*. Não há como garantir que as metas serão alcançadas. Algumas metas podem ser incompatíveis com os parâmetros ou com o ambiente disponível. Mesmo falhando, podem fornecer um *feedback* importante, e indicar necessidade de software e hardware. Deve-se realizar ajustes até chegar o mais próximo possível das metas desejadas.

Controles de ergonômica no coletor paralelo

As opções relativas à ergonômica referem-se a ajustes realizáveis no coletor paralelo de alta eficiência. Todas as opções abaixo requerem o uso do parâmetro *-XX:+UseParallelGC* ou um *Server JVM* com o coletor *default*.

A opção

```
-XX:MaxGCPauseMillis=valor
```

estabelece a meta de pausas máximas. O valor representa uma quantidade em milissegundos, que é o tempo máximo que o coletor poderá parar a aplicação

para realizar coleta de lixo. A máquina virtual tentará garantir pausas mais curtas que o valor especificado. Esta opção tem precedência sobre a opção:

```
-XX:GCTimeRatio=n
```

que define uma meta de eficiência (*throughput*). A eficiência é

$$\frac{\textit{tempo de aplicação}}{\textit{tempo de coleta de lixo}} = 1 - \frac{1}{1+n}$$

Onde *n* é um valor normalizado que mede a proporção de tempo dedicado à aplicação da forma 1:*n*. Se *n* for 19, por exemplo, a máquina virtual reservará à aplicação 20 (19 + 1) vezes mais tempo que a coleta de lixo (coleta terá 5% do tempo). Esta opção tem menos precedência que *-XX:MaxGCPauseMillis*, ou seja, se a meta de pausas estiver presente, ela será buscada em detrimento da eficiência.

A opção

```
-XX:+UseAdaptiveSizePolicy
```

é automaticamente ligada se a opção *–XX:+UseParallelGC* estiver presente ou se a *Server JVM* estiver sendo usada com seu coletor *default*. Com esta opção presente, a máquina virtual coleta dados e se baseia neles para redimensionar as gerações jovem e antiga.

Servidores dedicados com pelo menos 256MB de memória física podem usar a opção

```
-XX:+AggressiveHeap
```

que leva a máquina virtual a utilizar informações como quantidade de memória e número de processadores para configurar vários parâmetros buscando otimizar tarefas que fazem uso intenso de memória. Seu uso implica no uso das opções *–XX:+UseParallelGC* e *–XX:+UseAdaptiveSizePolicy*. É uma opção exclusiva do coletor de paralelo (*ParallelGC*) e não pode ser usada em conjunto com *–XX:+UseConcMarkSweepGC*.

Como utilizar a ergonômica

Inicialmente, não escolha um valor máximo para o *heap* (*-Xmx*). Escolha uma meta de eficiência (*throughput*) que seja suficiente para sua aplicação. Em uma situação ideal, o sistema aumentará o *heap* até atingir um valor que permitirá alcançar a meta de eficiência desejada.

Se o *heap* alcançar o limite e o *throughput* não tiver sido alcançado, então escolha um valor máximo para o *heap* (menor que a memória física da máquina) e rode a aplicação de novo. Se ainda assim a meta de eficiência não for atingida, é alta demais para a memória disponível na plataforma.

Se a meta de eficiência foi alcançada mas as pausas ainda forem excessivas, estabeleça uma meta de tempo máximo para pausas. Isto pode fazer com que a meta de eficiência não seja mais alcançada. Escolha valores que garantam um *tradeoff* aceitável.

Conclusões

Máquinas virtuais *HotSpot* implementam diversos algoritmos clássicos de coleta de lixo. Todos são fundamentados em um *heap* dividido em gerações e são pré-configurados para situações, plataformas e usos diferentes. Todos permitem ajustes manuais ou automáticos.

O ajuste correto da máquina virtual em grandes aplicações pode trazer ganhos dramáticos de performance. Um *ajuste* pode ser simplesmente a seleção da máquina virtual (servidora ou cliente) ou a definição manual de diversos e complexos parâmetros. As versões mais recentes da JVM permitem ajustes automáticos, mas para ajustar quaisquer parâmetros (mesmo os automáticos) é preciso conhecer um pouco sobre o funcionamento dos algoritmos. A configuração manual (ex: tamanho de *heap*) impõe conseqüências que têm impactos em outras áreas da performance, inclusive no ajuste automático. Metas de ajuste automático também afetam outras metas ou parâmetros.

10. Apêndice: Class data sharing (CDS)

Class data sharing (compartilhamento dos dados de classes) é um recurso das *JVM Client*, versão 5.0 para reduzir o tempo de início de pequenas aplicações. Durante a instalação, é criado um arquivo de classes que serão compartilhadas pelas máquinas virtuais que estiverem executando, evitando ter que carregá-las novamente em outras execuções.

Para suportar este recurso, é preciso usar uma plataforma que *não* seja *Windows 95/98/ME* e usar o *JVM Client* e coletor de lixo serial (*default* em *desktops*).

As opções da máquina virtual *HotSpot* relacionadas com *CDS* são:

```
-Xshare:[on|off|auto]
```

que liga/desliga ou usa *CDS* automaticamente, se possível, e

```
-Xshare:dump
```

que gera novamente o arquivo de classes compartilhadas. O arquivo fica armazenado na área do ambiente de execução. Para que o arquivo seja gerado de novo, é preciso primeiro apagá-lo. No Java 5.0, este arquivo está localizado em *$JAVA_HOME/client/classes.jsa*.

Parte III - Finalização, *memory leaks* e objetos de referência

A maior parte deste tutorial tratou de assuntos que interessam mais ao administrador de sistema que o programador Java. Por não precisar se preocupar com a liberação de memória, tampouco lidar com algoritmos complexos de alocação, um programador pode criar seus programas e sequer lembrar da existência de heap, pilha, coletas de lixo e outras questões relacionadas à memória. Esses temas geralmente não são parte das preocupações de um programador Java e surgem normalmente numa fase mais avançada do desenvolvimento ou na fase de otimização.

Mas conhecer em algum detalhe o processo de criação e destruição de objetos é importante pois a linguagem Java é flexível o suficiente para permitir que um programador inadvertidamente sobreponha métodos chamados de um construtor, esqueça de anular referências de coleções não utilizadas, crie muitos objetos em um tempo muito curto, chame o coletor de lixo explicitamente, chame os finalizadores ou use métodos da API como *finalize()* sem entender completamente como funcionam. Programas que usam inadequadamente recursos que interferem na alocação e liberação de memória poderão não fazer diferença, poderão eventualmente rodar mais rápido, porém têm grande chance de causar problemas que levarão a uma performance indesejável ou até mesmo seu funcionamento incorreto.

Esta seção explora o processo de construção e destruição de objetos do ponto de vista de um programador, tratando principalmente da finalização, que é um tema menos abordado. Mostra como implementar corretamente o método finalize(), como evitar vazamentos de referências em construtores, e como evitar finalize(). Trata de um problema freqüentemente ignorado por grande parte dos programadores Java: os vazamentos de memória, ou *memory leaks*. Eles existem em Java. Embora bem mais benignos que suas variações em C ou C++, ainda provocam problemas como esgotamento dos recursos da máquina virtual. Algumas estratégias para achá-los e consertá-los serão discutiudas. Por fim, uma API para controlar diversas etapas da finalização dos objetos foi introduzida a partir do Java 1.2: os objetos de referência. Apesar de não ser nova, é pouco usada. Podem não só permitir maior controle nas etapas de finalização de objetos, como permitir a criação de objetos com referências fracas que podem ser recolhidos quando a memória está escassa ou a cada coleta.

Como esta é uma seção voltada para programadores, haverá mais exemplos de código que argumentos de linha de comando e algoritmos de coleta de lixo, mas os assuntos abordados nas seções anteriores serão úteis pois o ambiente que iremos controlar é exatamente o que foi explorado anteriormente.

11. Alocação e liberação de memória

A criação de um objeto geralmente envolve a alocação de memória no *heap* para conter o objeto e a atribuição do ponteiro – endereço no *heap* onde o espaço para o objeto foi alocado – a uma variável de pilha que guardará a referência. Objetos podem ser criados explicitamente de duas formas [JVM 2.17.6]:

- através de uma expressão *new Classe()*;
- através do método *newInstance()* da classe *java.lang.Class*.

Apenas objetos *String* podem ser criados implicitamente. A criação de Strings pode ser realizada de três maneiras:

- através da definição de um literal,
- através da carga de uma classe que possui literais do tipo *String*, ou
- através da concatenação de literais do tipo *String*.

Quaisquer objetos criados são destruídos automaticamente pela JVM através do sistema de gerenciamento de memória por coleta de lixo automática.

Criação de objetos

Quando uma nova instância de uma classe é criada, memória é alocada para todas as variáveis de instância declaradas na classe e superclasses, inclusive variáveis ocultas. Não havendo espaço suficiente para alocar memória para o objeto, a criação terminará com um *OutOfMemoryError*.

Se a alocação de memória terminar com sucesso, todas as variáveis de instância do novo objeto, inclusive aquelas declaradas nas superclasses, serão inicializadas a seus valores *default* (0, *null*, *false*, '\u0000'). No passo seguinte, os valores passados como argumentos para o construtor são copiados às variáveis de parâmetro locais e a construção é iniciada.

A execução de um construtor envolve a chamada de operações exclusivas da criação de objetos: *super()*, que faz uma chamada de subrotina ao construtor da superclasse, e *this()*, que chama um outro construtor da mesma classe. A primeira instrução do construtor não pode ser outro código exceto uma chamada implícita (oculta) ou explícita a *super()*, ou uma chamada explícita a *this()* – que também passará o controle para um outro construtor que em algum ponto chamará *super()*. O controle *sobe* a hierarquia através da cadeia de construtores chamados pela instrução *super()*. Chegando na classe *Object* – o único que não possui *super()* – realiza os seguintes passos:

1. Inicializa variáveis de instância que possuem inicializadores explícitos (atribuições no local da declaração)
2. Executa o corpo do construtor
3. Retorna para o próximo construtor da hierarquia (*descendo* a hierarquia), e repete esses três passos até terminar no construtor que foi chamado pela instrução *new*. Quando o último construtor for terminado, retorna a referência de memória do novo objeto.

É mais fácil entender o processo com uma ilustração. A *figura 50* mostra os vários passos da criação de um objeto.

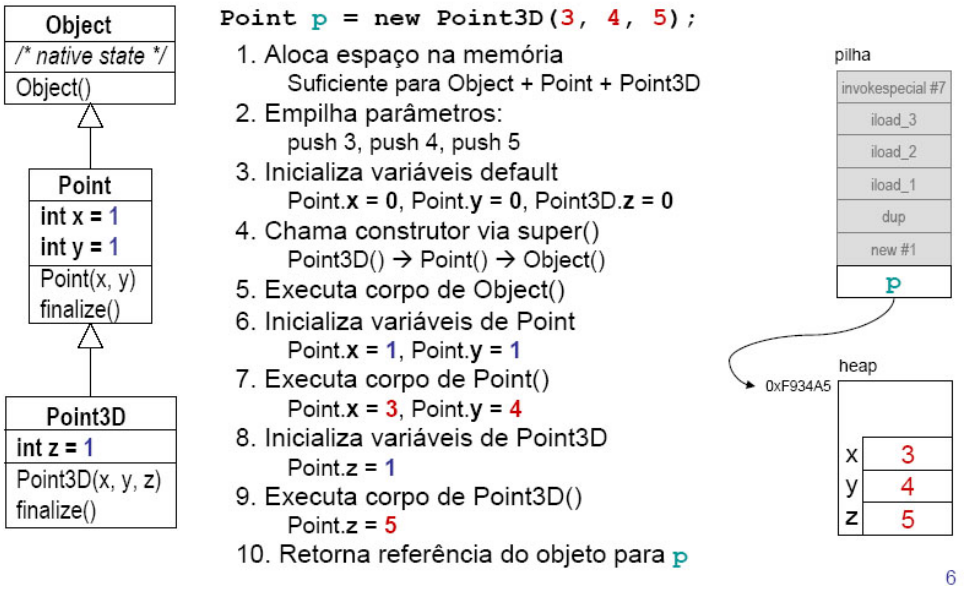

Figura 50 – Passo a passo da construção de um objeto

Destruição de objetos

Em Java, o coletor de lixo realiza a destruição de objetos liberando a memória que foi alocada para ele. Não é responsabilidade do programador preocupar-se com a remoção de qualquer objeto individual. O instalador ou usuário da aplicação pode interferir ajustando as configurações do coletor de lixo para o ambiente onde a aplicação irá executar. O programador pode interferir de maneira limitada no processo de destruição de várias maneiras:

- rotinas de finalização inseridas antes da liberação de memória,
- chamadas explícitas ao coletor de lixo,
- remoção das referências para um objeto para torná-lo elegível à coleta,
- uso de referências fracas, ou
- finalização.

Antes que a memória de um objeto seja liberada pelo coletor de lixo, a máquina virtual chamará o *finalizador* desse objeto [JLS 12.6].

A linguagem Java não determina em que momento um finalizador será chamado. A única garantia é que ele será chamado antes que a memória do objeto seja liberada para reuso (pode nunca acontecer). Também é garantido que o construtor de um objeto completará *antes* que a finalização do objeto tenha início.

A linguagem também não especifica qual *thread* chamará o finalizador, mas garante que esse *thread* não estará usando travas acessíveis pelo usuário. Não garante nenhuma ordenação: a finalização pode acontecer em paralelo com outros processos.

A finalização é importante? Depende. Há objetos que *não precisam* de finalizadores. São aqueles cujos recursos são automaticamente liberados pelo coletor de lixo, por exemplo, alocação na memória e referências (inclusive

circulares) de qualquer tipo. E há objetos que *precisam* de finalizadores. São os que precisam liberar recursos externos. Eles precisam

- Fechar arquivos abertos e *soquetes*: o sistema operacional limita quantos recursos podem ser abertos simultaneamente; não finalizar depois do uso pode impedir a criação de novos arquivos ou *soquetes*;
- Fechar *streams*: fluxos de gravação podem ficar incompletos se o *buffer* não for esvaziado.
- Terminar *threads*: eles costumam rodar em *loops*; finalizadores ligam um *flag* para terminar o *loop* ou interrompem o *thread* para evitar que o programa nunca termine.

A *figura 51* ilustra o processo de destruição de objetos.

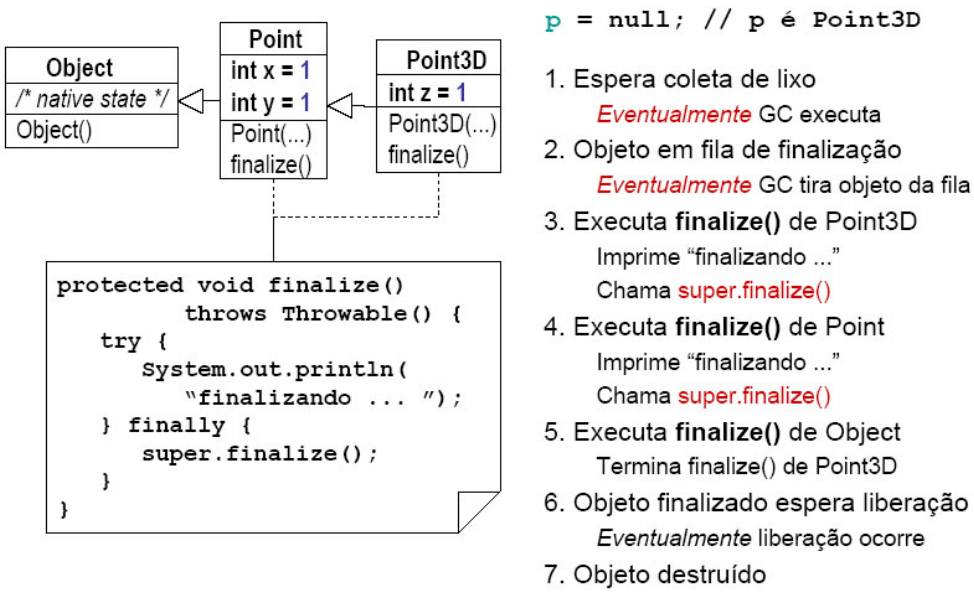

Figura 51 – Passo-a-passo da destruição de objetos

Os objetos da *figura 51* possuem finalizadores automáticos. Em Java, qualquer objeto *pode* ter um finalizador chamado automaticamente antes de ser destruído. Finalizadores em Java são opcionais e não são tão importantes quanto finalizadores em C ou C++. Para implementar, é preciso sobrepor a assinatura:

```
protected void finalize() throws Throwable {...}
```

O método *finalize()* é chamado automaticamente e *apenas uma vez* somente quando o objeto não for mais alcançável através de referências comuns (as referências raiz). O método *finalize()* não será chamado se

- Não sobrepuser explicitamente o método original: o uso é opcional!
- Não houver necessidade de liberar memória (o coletor não executar), mesmo que todas as referências para o objeto já tenham sido perdidas.

A chamada dos finalizadores automáticos, portanto, não é garantida durante a vida da aplicação. Sua chamada também depende de vários outros fatores e da implementação do coletor de lixo usado.

O ciclo de vida completo de um objeto, envolvendo sua criação, finalização e destruição está ilustrado na *figura 52*.

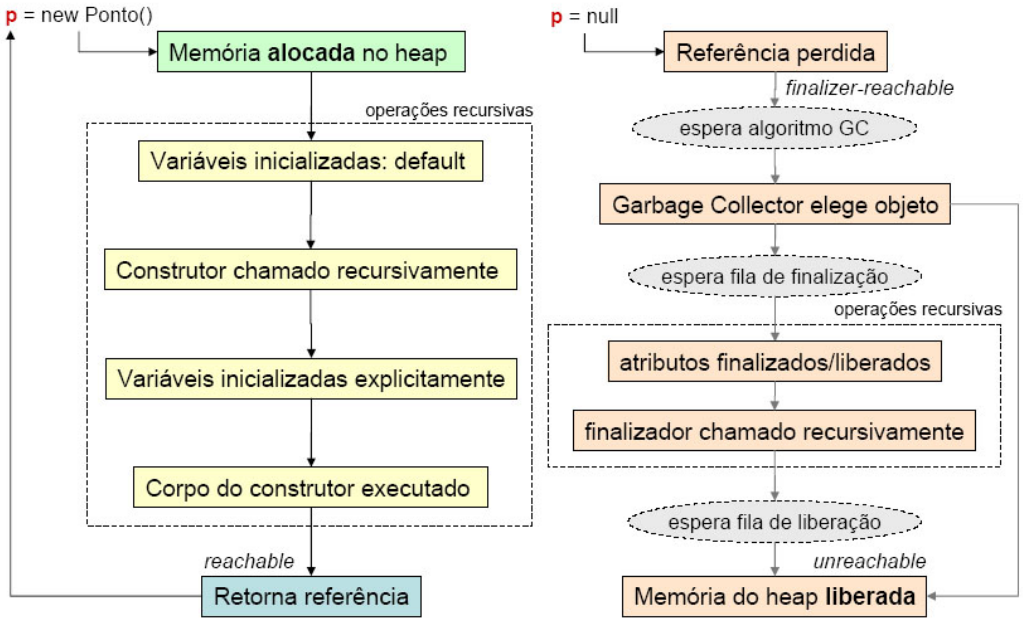

Figura 52 – Ciclo de vida de um objeto

Alcançabilidade

Objetos alcançáveis são objetos que não podem ser destruídos pelo coletor de lixo. Podem ser *alcançados* através de uma *corrente de referências* partindo de um *conjunto raiz* de referências. O conjunto raiz contém referências imediatamente acessíveis ao programa, em determinado momento. São referências do conjunto raiz:

- *Variáveis locais* e argumentos dos métodos quando estão executando um thread ativo (referências armazenadas na pilha);
- *Variáveis de referência estáticas*, depois que suas classes forem carregadas;
- Variáveis de referência registradas através da *Java Native Interface*, implementadas em outras linguagens.

Existem três estados elementares de *alcançabilidade*:

- *alcançável* (*reachable*): pode ser acessado através de um *thread* ativo; existem quatro *forças* diferentes de alcançabilidade;
- *inalcançável* (*unreachable*): não pode ser acessado por nenhum meio e está elegível à remoção;
- *alcançável por finalizador* (finalizer-reachable): é um objeto quase inalcançável, pois não pode ser alcançado através das vias normais. Pode ser ressuscitado se, após a morte, seu finalizador passar sua referência *this* para algum objeto alcançável.

E há três estados em que a *finalização* de um objeto pode se encontrar:

- *não finalizado* (*unfinalized*): nunca teve seu finalizador chamado;
- *finalizado* (*finalized*): já teve seu finalizador chamado;

- *finalizável* (*finalizable*): seu finalizador pode chamá-lo automaticamente a qualquer momento. É um objeto que não é mais alcançável.

O diagrama da *figura 53* mostra a transição entre estados durante a vida de um objeto. Se um objeto não tem finalizador explícito, o diagrama tem apenas dois estados de alcançabilidade: *alcançável* e *inalcançável*, e um estado de finalização: *não finalizado*.

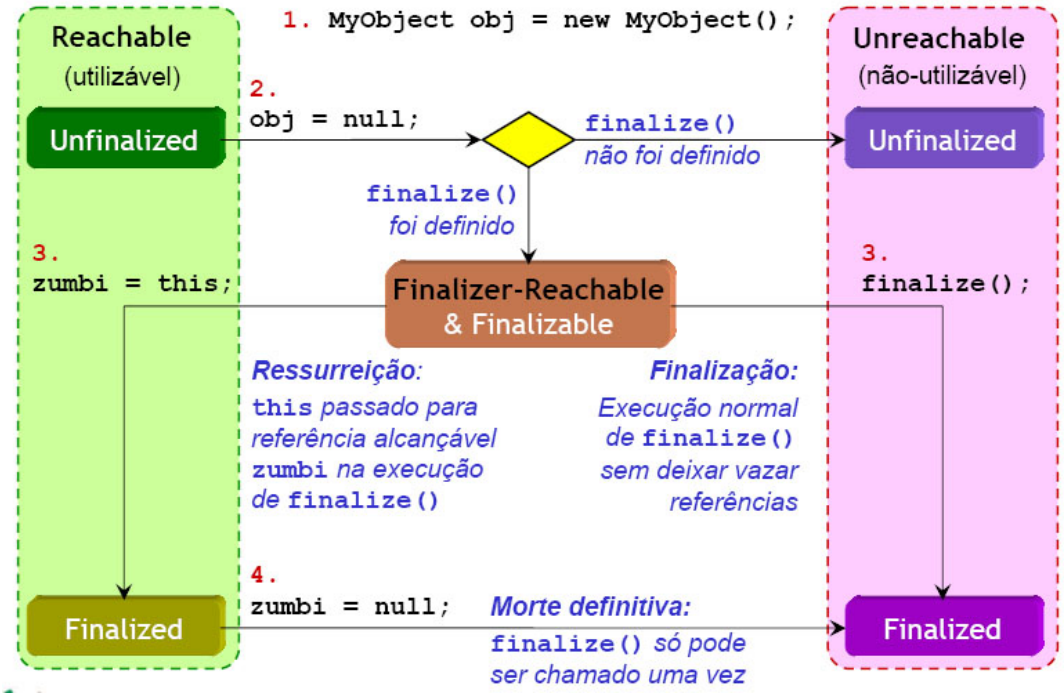

Figura 53 – Transição entre estados de finalização e de alcançabilidade. O diagrama não leva em conta a existência de referências fracas.

Ressurreição de objetos

Um objeto *finalizer-reachable* não tem mais referências entre os objetos vivos, mas, durante sua finalização pode copiar sua referência *this* para uma referência ativa. O objeto poderá então ser alcançado por referências externas, e assim "volta à vida". Nesse estado, se morrer outra vez, vai direto ao estado *unreachable* e não passa mais pelo método finalize() já que esse método só é executado uma vez.

Considere as duas classes abaixo. *HauntedHouse* (casa mal-assombrada) permite acomodar um único *Guest* (visitante). Há métodos para aceitar um visitante e para matá-lo:

```
public class HauntedHouse {
  private Guest guest;
  public void addGuest(Guest g) {
    guest = g;
  }
  public void killGuest() {
    guest = null;
  }
}
```

O visitante, por sua vez, possui uma referência para *HauntedHouse*, que é passada na sua criação. O construtor a sua própria referência (*this*) como argumento do método *addGuest()* da *HauntedHouse*.

```
public class Guest {
  private HauntedHouse home;
  Guest(HauntedHouse h) {
    home = h;
    home.addGuest(this);
  }
  protected void finalize() ... {
    home.addGuest(this);
  }
}
```

Para executar, criamos uma instância de *HauntedHouse* para passar como argumento do construtor na criação de um *Guest*. Não guardamos uma referência externa para o *Guest*, de forma que a única referência é a mantida dentro de *HauntedHouse*.

```
HauntedHouse h = new HauntedHouse();
new Guest(h);  // cria objeto e mantém referencia em h
```

Ao chamar *killGuest()* na tentativa de livrar-se da visita inoportuna, *HauntedHouse* destrói a última e única referência restante de *Guest*, deixando-o candidato à coleta de lixo. Porém, antes do objeto ser coletado, seu finalizador será executado, e quando isto acontecer, a referência *this* mais uma vez será obtida e passada ao método *addGuest()*, causando a resssurreição do objeto já dado como morto.

```
h.killGuest(); // mata objeto e finaliza, mas ele ressuscita!
```

O único jeito de livrar-se do fantasma é matar o objeto de novo. Como a finalização só ocorre uma vez, não há risco do objeto ressuscitar outra vez através do finalizador.

```
h.killGuest(); // mata objeto de novo... desta vez ele vai
```

Os exemplos mostrados sobre ressurreição de objetos têm finalidade didática e foram usados para facilitar o entendimento do processo de finalização. Mas acordar os mortos geralmente não é uma boa idéia. A ressurreição de objetos raramente tem aplicações práticas e geralmente é uma prática a ser evitada, portanto, *não ressuscite objetos*. Os problemas que sugerem a ressurreição de objetos como solução geralmente podem ser melhor implementadas com novos objetos e cópia de seus estados (clonagem, por exemplo). Além disso, objetos de referência permitem práticas envolvendo finalização que são mais seguras e previsíveis para problemas similares.

Como escrever finalize()

O método *finalize()* é opcional. Objetos que não tenham declarado finalizadores explícitos, não serão finalizados e irão direto para o lixo, portanto, use finalização automática apenas se for necessário.

Se precisar mesmo usar um finalizador, então escreva-o corretamente. Construtores automaticamente chamam a sua superclasse, mas finalizadores não chamam automaticamente os finalizadores das superclasses. A correta implementação deve sempre chamar *super.finalize()* explicitamente, de preferência em um bloco *finally* para garantir sua execução:

```
protected void finalize() throws Throwable {
   try {
      // código de finalização
   } finally {
      super.finalize();
   }
}
```

O trecho acima é a técnica padrão para escrever *finalize()*. Exceções não devem ser capturadas dentro dos finalizadores, já que não serão úteis e poderão atrasar a finalização.

A finalização é uma operação que só ocorre se houver coleta de lixo, portanto, para demonstrá-la, criaremos um programa inútil cuja principal finalidade será *ocupar memória* para forçar a coleta de lixo.

Considere a classe e o trecho de código a seguir, executado com *pouca memória* (1 megabyte de heap) para garantir uma coleta de lixo mais freqüente. Os objetos usam *referências fracas*[20] para que sejam liberados com freqüência. O construtor, o método *finalize()* e o bloco *finally* fazem contagem de chamadas.

```
public class FinalizingObject {
    private int[] state;
    public FinalizingObject(String state) {
        this.state = new int[1000];
        creationCount++;
    }
    public void finalize() throws Throwable {
        finalizationCount++;
        super.finalize();
    }
}
...
WeakHashMap fp = new WeakHashMap();
for (int i = 0; i < 1000; i++) {
   try {
      fp.put(-i, new FinalizingObject());
   } finally {
      ++finallyCount;
   }
}
...
```

Pergunta: quanto deve ser a contagem de cada um, se *mil* objetos forem criados e depois destruídos? O *for* acima irá alocar mil entradas no mapa *fp*, mas

[20] Referências fracas serão abordadas mais adiante neste tutorial.

se a memória acabar, ela será recuperada pois o *WeakHashMap* irá liberar suas referências automáticamente em caso de memória escassa.

A execução, com –verbose:gc e heap limitado a 1 megabyte produziu os seguintes resultados:

```
> java -Xmx1M –Xms1M -verbosegc -cp build/classes memorylab.Main
...
[Criados agora:     200; total criados:     1000]
[Finalizados agora:  83; total finalizados:  670]
```

- Construtor foi executado 1000 vezes.
- Bloco finally foi executado 1000 vezes.
- Finalizador foi executado 670 vezes.

Observe que a máquina virtual terminou antes que todos os finalizadores fossem executados. Se havia 1000 objetos, era de se esperar que houvesse 1000 finalizações. Mas nem todos os objetos foram coletados.

Se a memória for aumentada, tudo muda:

```
> java –Xmx8M –Xms8M -verbosegc -cp build/classes memorylab.Main
...
[Finalizados agora:  0; total finalizados:    0]
[Criados agora:   1000; total criados:     1000]
```

- Construtor foi executado 1000 vezes.
- Bloco finally foi executado 1000 vezes.
- Finalizador foi executado 0 vezes.

Nenhum objeto foi finalizado! Por que? Simples: não foi necessária a execução do coletor de lixo. Os objetos não foram coletados.

Conclusão: *não dependa da finalização*! Nunca dependa de uma chamada automática a *finalize()*. Uma aplicação em ambiente com muita memória pode nunca chamar os *finalize()* dos objetos que perderam suas referências, e assim deixar de executar código importante. A mesma aplicação em um ambiente igual mas com menos memória faria chamadas ao *finalize()* de vários objetos. Para *finalize()* ser chamado, é necessário que o objeto esteja prestes a ser coletado. Se objetos são criados e suas referências são sempre alcançáveis, nunca serão finalizados nem coletados.

O método *finalize()* pode nunca ser chamado por não haver necessidade de rodar o coletor de lixo (em coleta completa), não haver necessidade de reusar sua memória, ou outras razões dependentes de implementação/plataforma. Além dessa desvantagem, os finalizadores automáticos também podem contribuir para o consumo de memória e baixa performance, não são previsíveis, funcionam diferentemente entre plataformas e ignoram exceções. Nem o comando *System.gc()* – que chama o coletor de lixo – garante a execução de um finalizador. Por que usar um finalizador, então?

Finalizadores são importantes! Não finalizadores *automáticos*, como *finalize()*, mas finalizadores *explícitos*! Finalizadores são quaisquer métodos que ajudam a encerrar o uso de um objeto. A finalização de arquivos, soquetes, e outros

recursos não-relacionados à liberação de memória não serão feitas pelo sistema de coleta de lixo e têm que ser realizadas pelo programador. Por serem tão importantes, não devem *depender* da finalização automática do sistema através de *finalize()*, mas devem estar presentes através de métodos de finalização explícita!

File.close(), *Socket.close()*, *Window.dispose()*, *Statement.close()* e outros métodos existentes nas APIs Java são métodos de finalização explícita. Devem ser chamados pelo cliente – geralmente em um bloco *try-finally* para garantir sua execução. Isto implica em uma mudança na atribuição de responsabilidades: a finalização de um recurso após o seu uso é uma responsabilidade do cliente da API, e não do seu autor.

Métodos de finalização explícita podem também ser chamados por *finalize()* como uma *rede de segurança*, caso o cliente esqueça de finalizar. A maioria dos métodos de finalização explícita da API Java usa *finalize()* como rede de segurança para liberar recursos de qualquer maneira, mesmo que o usuário cliente não tenha chamado o método de finalização. Eventualmente, quando a memória acabar, *finalize()* será chamado e recursos que o usuário esqueceu de finalizar poderão ser recuperados.

Um exemplo de classe usando *finalize()* como rede de segurança está mostrado a seguir:

```
class Cache { ...
  Thread queueManager;
  void init() {
    Runnable manager = new Runnable() {
      public void run() {
        while(!done) {
          try { blockingOperation(); }
          catch (InterruptedException e) {
             done = true; return;
          }
        }
      }
    };
    queueManager = new Thread(manager);
    queueManager.start();
  }
   public void close() { // FINALIZADOR EXPLÍCITO
     done = true;
     if(!queueManager.isInterrupted())
       queueManager.interrupt();
   }
   protected void finalize()
        throws Throwable {    // FINALIZADOR AUTOMÁTICO
     try { close(); }
     finally { super.finalize(); }
   }
}
```

A forma correta de usar a classe *Cache* é:

```
Cache c = new Cache();
try {
  c.init();
  // usar o cache
} finally {
  c.close();
}
```

Se o cliente esquecer de chamar *close()*, existe a possibilidade do *Cache* ser liberado se a memória acabar e o coletor de lixo for chamado. Se isto não acontecer, o *Cache* não será finalizado.

Finalizer Guardian

Havendo necessidade de implementar finalize(), é preciso implementá-lo corretamente. O que fazer se o cliente que sobrepõe a classe não implementar corretamente *finalize()*, esquecendo, por exemplo. de chamar *super.finalize()*? Pode-se usar o padrão *Finalizer Guardian* para garantir que o finalizador de uma superclasse será chamado quando o objeto de uma subclasse for finalizado.

O *Finalizer Guardian* é um atributo do objeto protegido que funciona porque antes de um objeto ter sua memória liberada, seus atributos serão liberados (e finalizados se preciso). É um objeto que implementa seu próprio *finalize()* com uma chamada ao *finalize()* da classe que o contém (e guarda). Protege contra implementação incorreta de *finalize()* por parte das subclasses.

O código abaixo ilustra o uso do padrão *Finalizer Guardian* implementado como uma classe interna.

```
public class Recurso { ...
    private final Object guardian = new Object() {
        protected void finalize() throws Throwable {
            Frase.this.close(); // finaliza Recurso
        }
    };
    public void finalize() throws Throwable {
        try {
            close(); // chama finalizador explícito
        } finally {
            super.finalize();
        }
    }
    public void close() throws Throwable {
        // finalização explícita
    }
}
```

(Fonte: Joshua Bloch, Effective Java, Item 6)

Finalização de threads

A Interface *Thread.UncaughtExceptionHandler*[21], é usada para lidar com exceções que não foram capturadas. É uma interface interna da classe *Thread*:

[21] Em versões anteriores a Java 1.5, use *ThreadGroup.uncaughtException()*

```
public class Thread ... { ...
   public interface UncaughtExceptionHandler {
       void uncaughtException(Thread t, Throwable e);
   }
}
```

Pode-se implementar a interface com código a ser executado antes que o *thread* termine devido a uma exceção não capturada.

```
public static void main(String args[]) {
  Thread.UncaughtExceptionHandler handler =
      new Thread.UncaughtExceptionHandler () {
          void uncaughtException(Thread t, Throwable e) {
              // fazer finalização
          }
  };
  Thread.currentThread().setUncaughtExceptionHandler(handler);
  // segue código que pode causar exceção
}
```

Como tornar um objeto elegível à remoção pela coleta de lixo?

Apara que um objeto seja coletado, é preciso primeiro que se torne *inalcançável*. Isto é feito eliminando *todas* as suas referências a partir dos nós raiz do *thread* principal (variáveis locais e estáticas).

- Declarar a última referência como *null* torna-o inalcançável imediatamente (ou *finalizer-reachable*, se tiver finalizador).
- Atribuir outro objeto à última referência do objeto não o torna *imediatamente* inalcançável (porém atuais implementações de JVMs garantem o mesmo efeito que *null*).
- Objetos criados dentro de um método tornam-se inalcançáveis pouco depois que o método termina. Não basta sair do escopo de um bloco. É preciso sair do escopo do método.

É importante garantir que não haja outras referências para o objeto. É comum "esquecer" referências ativas em listas de *event handlers* e coleções. Esses são os casos mais comuns de *memory leak*.

Chamar o método *System.gc()* após eliminar todas as referências para um objeto *pode* liberar a memória dos objetos inalcançáveis. *System.gc()* executa o garbage collector assim que possível. *Sugere* à JVM que ela faça um esforço para reciclar objetos não utilizados, para liberar a memória que ocupam para que possa ser reusada, porém existe um nível de incerteza associado à execução desse comando. A execução pode não acontecer imediatamente ou nunca se o programa pode terminar antes.

Uma chamada a *System.gc()* também não garantirá a liberação de memória de todos os objetos inalcançáveis. Os algoritmos de coleta de lixo podem, para aumentar a eficiência, deixar de recolher todos os objetos encontrados como lixo que serão recolhidos em coletas posteriores. Pode-se resolver esse problema chamando *System.gc()* várias vezes até que a memória pare de diminuir.

Pode-se descobrir se a memória parou de diminuir usando o método *freeMemory()* da classe Runtime:

```
Runtime rt = Runtime.getRuntime();
do {
      long memLivreAntes = rt.freeMemory();
      System.gc();
      long memLivreDepois = rt.freeMemory();
while (memLivreAntes != memLivreDepois);
```

Chamar *System.gc()* repetidamente é muito ineficiente e só deve ser usado em casos extremos. É inútil se não houver objetos disponíveis à remoção. O ideal é encontrar estratégias que não precisem chamar *System.gc()*, exceto em raros casos e para depuração.

Um método associado é *System.runFinalization()*, que executa a finalização de métodos de quaisquer objetos cuja finalização ainda não foi feita. Isto só acontece se objeto já for candidato à liberação através do coletor de lixo (se for *finalizable*). Uma chamada a *System.runFinalization() sugere* à máquina virtual que realize o melhor esforço para executar os métodos *finalize()* de objetos que estão marcados para remoção, mas cujos métodos de finalização ainda não foram executados.

Este método é ainda menos previsível que *System.gc()*. O antigo método *System.runFinalizersOnExit()* é o único que garante a execução dos finalizadores, mas é inseguro e foi deprecado.

O trecho de código abaixo força o coletor de lixo como meio de garantir a finalização de um objeto. O bloco *finalize()* imprime o seu nome passado no construtor para que possamos saber qual objeto finalizou.

```
public static void main(String[] args) {
   System.out.println("Creating object...");
   Citacao cit = new Citacao("Primeiro objeto...");
   cit = null;
   System.out.println("Forcing GC...");
   System.gc();
   cit = new Citacao("Segundo!");
   cit = null;
   System.out.println("Forcing GC again...");
   System.gc();
   System.out.println("Done");
}
```

Na execução, apenas a primeira finalização ocorreu[22].

```
Creating object...
Forcing GC...
Forcing GC again...
finalize(): Primeiro objeto...; Done
```

[22] Na minha máquina! Na sua pode funcionar diferente. O comportamento é dependente da plataforma e implementação da JVM

Se o finalizador realizasse alguma tarefa crítica, como por exemplo, o fechamento de arquivos, e a aplicação continuasse por mais tempo, esses arquivos ficariam bloqueados usando recursos da máquina. Portanto, mesmo usando S*ystem.gc()* não há como garantir a execução dos finalizadores.

Resumo

Na API e linguagem Java, as alternativas que o programador possui para chamar ou induzir a coleta de lixo são:

- *System.gc():* chama o garbage collector assim que possível, mas só elimina objetos que já estiverem inalcançáveis. É ineficiente, pois pára o sistema para remover os objetos, e tem comportamento dependente da máquina virtual.
- *Runtime.getRuntime().gc()*: faz o mesmo que *System.gc()*.
- *ref = null*: declarar a última referência para um objeto como *null*, vai torná-lo elegível à coleta de lixo (estado *inalcançável* ou *finalizer-reachable*). É mais rápido que reutilizar a referência, ou fechar o bloco do método onde o objeto foi declarado.
- *Referências fracas*: permitem um controle mais eficiente; serão abordadas mais adiante.

12. Memory leaks

Um vazamento de memória, ou *memory leak*, no sentido C++, ocorre quando um objeto não pode ser alcançado e não é liberado através da coleta de lixo. Isto não pode ocorrer em aplicações 100% Java[23].

Memory leaks em Java são considerados em um sentido mais abrangente. É considerado um *memory leak* um objeto que não é coletado depois que não é mais necessário. Apesar de não serem mais usados, não são liberados porque ainda são alcançáveis. Uma interface que impede ou que não garante que o cliente irá liberar uma referência depois do uso tem potencial para *memory leak*.

Já que baseia-se no período em que um objeto é útil, o critério para definir um *memory leak* nem sempre é muito claro: pode ser subjetivo, depender de um contexto ou ainda de algum evento, por exemplo: o fato da memória estar sendo consumida muito rapidamente, ou um *OutOfMemoryError*.

Considere a classe abaixo[24]. É uma pilha com método *pop()* para inserir dados e *push()* para extrair. Possui um método *ensureCapacity()* que aumenta o tamanho da pilha caso seja necessário. Há algum problema com esta classe?

```
public class BadStack { // não é thread-safe!
    private Object[] elements;
    private int size = 0;
    public BadStack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0) throw new EmptyStackException();
        return elements[--size];
    }
    public int size() { return size; }
    private void ensureCapacity() {
        if (elements.length == size) {
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

Para testá-la, escrevemos um programa que cria duas pilhas e transfere objetos de uma para a outra:

```
BadStack res = new BadStack(1000);
BadStack src = new BadStack(1000);

for (int i = 0; i < 1000; i++)
   src.push(new Character((char)((Math.random()*26) + 'A')));
```

[23] Se acontecer é bug na máquina virtual, o que não é responsabilidade do programador.

[24] Fonte: Joshua Bloch, *Effective Java*, Item 5.

```
System.out.println("ANTES");
// imprime src.size(), res.size()

try {
   while(true) {
      char c = Character.toLowerCase(
          (Character)source.pop());
      res.push(new Character(c));
   }
} catch (EmptyStackException e) {}
System.out.println("DEPOIS");
// Imprime mesmas informações
```

O programa imprime os dados de ANTES e DEPOIS:

```
ANTES
src.size(): 1000
res.size(): 0

DEPOIS
src.size(): 0
res.size(): 1000
```

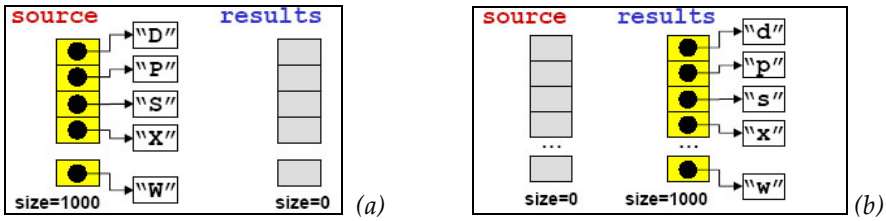

Figura 54 – Situação aparente dos arrays source e results.

Aparentemente o programa funciona corretamente, como mostram os dados e a *figura 54*. Do ponto de vista do usuário, uma pilha foi esvaziada e a outra foi preenchida. Mas no que se refere ao coletor de lixo, os 1000 objetos da pilha que foi esvaziada (*src*) continuam accessíveis. Como a variável está encapsulada, o usuário não consegue vê-la. Se imprimíssemos também a contagem de objetos em *src*, teríamos:

```
ANTES
Instancias em src: 1000
Instancias em res: 0

DEPOIS
Instancias em src: 1000
Instancias em res: 1000
```

E veríamos que na verdade, o resultado final é, na verdade, o da *figura 55*.

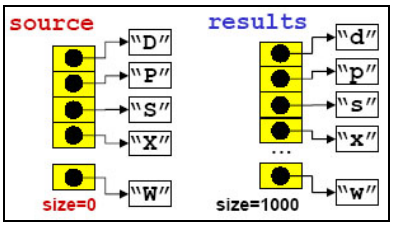

Figura 55 – Situação real dos arrays source e results.

Terminamos de usar o objeto, no entanto, ainda há 1000 instâncias que podem ser alcançadas! Elas não terão sua memória liberada pelo coletor de lixo. Do ponto de vista funcional, porém, o programa *está correto*. Foi necessário quebrar o encapsulamento para obter esses dados.

Para consertar o vazamento temos que eliminar as referências obsoletas para objetos que o programa mantém. O vazamento poderia ser ainda maior se os objetos da pilha tivessem referências para outros objetos, e assim por diante. Poderia ocorrer *OutOfMemoryError* em uma execução curta.

A forma mais simples de resolver o problema, é eliminar a referência, declarando-a *null*.

```
public Object pop() {
    if (size == 0) throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null;
    return result;
}
```

Como achar e consertar vazamentos?

Analise o código. Procure os lugares mais prováveis: coleções, *listeners*, *singletons*, objetos atrelados a campos estáticos. Desconfie de objetos com longo ciclo de vida em geral.

Teste, e force a coleta de lixo entre *test cases* repetidos. Exercite um segmento de código para examinar o *heap* e descobrir se ele está crescendo irregularmente. Use grafos de referência de objetos.

Use um *profiler* para achar objetos alcançáveis que não deviam ser alcançáveis: alguns usam cores para mostrar objetos muito usados e outros menos usados – preste atenção também nos objetos pouco utilizados.

Finalmente, use ferramentas de monitoração. O *jconsole)*, por exemplo, traça gráficos do heap e de suas regiões. O consumo médio de memória deve manter-se constante através do tempo, como mostra a *figura 56*.

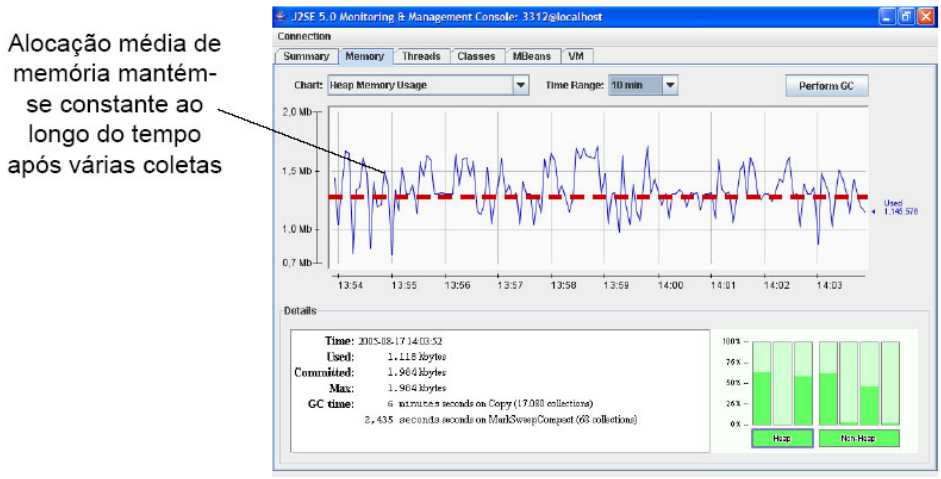

Figura 56 – Programa sem memory leaks.

Se acontecer do gráfico de consumo de memória crescer linearmente descontando-se as coletas de lixo, há um vazamento de memória (*figura 57*):

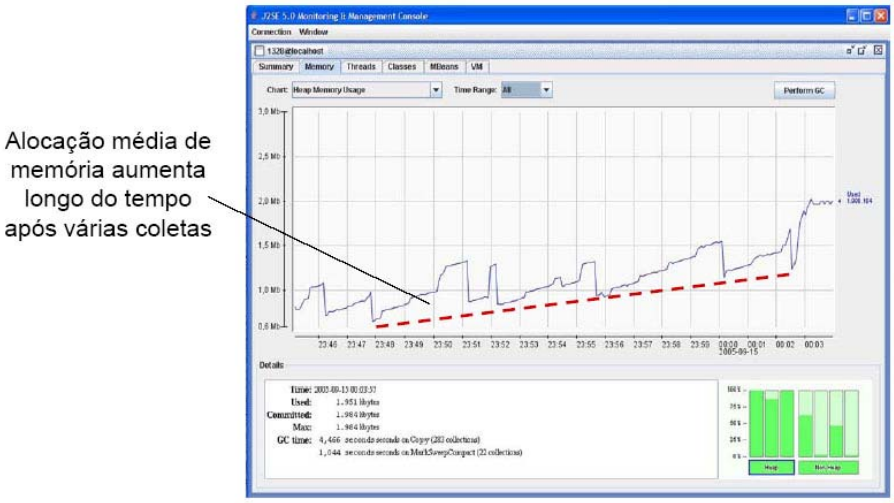

Figura 57 – Programa com memory leaks.

Para consertar o vazamento não adianta chamar *System.gc()*. Esse método além de ter impacto negativo na performance (executa uma coleta em todas as gerações) irá obter-se a coleta apenas dos objetos inalcançáveis, mas *memory leaks* são objetos *alcançáveis*.

É preciso eliminar todas as referências para o objeto. Procure-as usando ferramentas, se necessário. Alternativas para eliminação de referências incluem declarar a referência como *null* quando não for mais usada. Mas não abuse dessa alternativa. O ideal é manter as referências no menor escopo possível (o escopo mínimo deve ser o de método), ou reutilizar a referência.

Uma outra solução é utilizar objetos de referência para criar referências fracas. Isto será discutido na seção seguinte.

13. Referências fracas

Referências fracas são ponteiros cuja ligação com o objeto ao qual se referem é *fraca*: pode ser perdida a qualquer momento. Elas permitem que um programa aponte para um objeto sem impedir sua eventual coleta, caso seja necessário.

O coletor de lixo considera os objetos que são alcançáveis apenas via referências fracas como objetos que podem ser removidos. A API de *reference objects* (*java.lang.ref*) ou *objetos de referência* permite que um programa mantenha referências fracas para quaisquer objetos.

Típicas aplicações para esse tipo de referência são:

- Programas que mantém muitos objetos na memória, e não precisaria tê-los todos disponíveis a qualquer momento;
- Programas que usam muitos objetos por um curto período;
- Programas que precisam realizar operações de finalização nos objetos e outros objetos associados antes da liberação.

API dos objetos de referência

Objetos de referência são descendentes da classe *java.lang.ref.Reference*. A *figura 58* ilustra sua hierarquia.

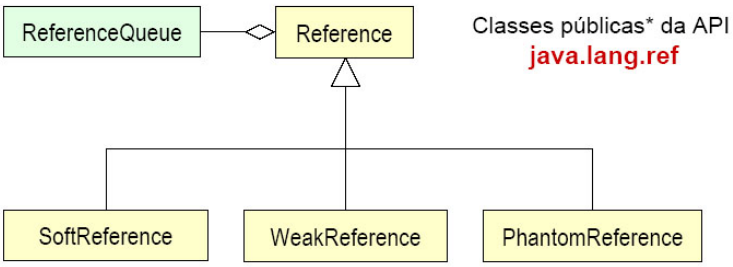

Figura 58 – Hierarquia dos objetos de referência.

A classe *ReferenceQueue*, como diz o nome, é uma fila. Quando usada com *WeakReference* ou *SoftReference* possibilita o tratamento de eventos durante a mudança da alcançabilidade. Pode ser usada para realizar pré-finalização. Usada com *PhantomReference* para guardar objetos já finalizados para a realização de tarefas pós-finalização.

SoftReference serve para implementar caches sensíveis à memória, que são esvaziados apenas quando a memória está muito escassa. Referências desse tipo sobrevivem a várias coletas mas são perdidas quando a máquina virtual precisar de mais memória do *heap*.

WeakReference é usada para implementar mapas nos quais chaves ou valores podem ser removidos do *heap* a qualquer momento. Referências desse tipo não sobrevivem a uma coleta de lixo.

A classe *PhantomReference* representa objetos já finalizados que ainda não foram recolhidos. Serve para implementar ações de finalização de uma forma mais flexível que o mecanismo de finalização automático do Java, ou para realizar tarefas adicionais depois da finalização.

Todas as classes aceitam parâmetros de tipo desde a versão 5.0 do Java, que encapsula o tipo do objeto para o qual mantém a referência fraca. Esse objeto é chamado de *objeto referente* (*referent*). Todos os objetos de referência possuem duas operações básicas, que são herdadas da classe *Reference<T>*, onde *T* é o tipo do referente.

- *T get()*: retorna o objeto referente. Este método é sobreposto em todas as subclasses para prover o comportamento distinto de cada tipo de objeto de referência.
- *void clear()* : elimina o objeto referente. Se o método *get()* for chamado depois de um *clear()*, retornará *null*.

Além das operações básicas, há dois outros métodos usados pelo coletor de lixo para gerenciar filas de objetos de referência. Esses métodos requerem que um objeto de referência receba um *ReferenceQueue* no momento da criação.

- *boolean enqueue()*: acrescenta este objeto de referência à fila no qual está registrado, se tiver sido registrado em uma fila no momento da criação.
- *boolean isEnqueued()*: retorna *true* se este objeto estiver sido enfileirado na ao qual foi registrado. O coletor de lixo automaticamente acrescenta um objeto na sua fila quando *clear()* é chamado.

Como usar objetos de referência

A *figura 59* ilustra como um objeto de referência mantém uma referência fraca para um objeto.

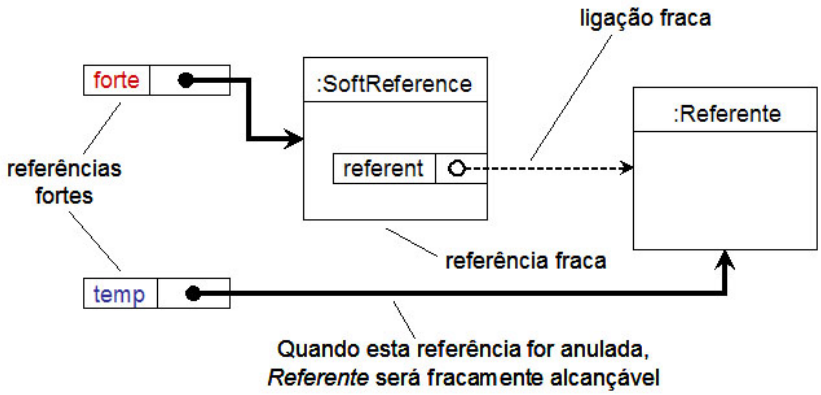

Figura 59 – Uso típico de um objeto de referência.

Inicialmente, o objeto deve ser criado da forma usual, através de uma referência forte, chamada de *temp* na *figura 59*. Após a criação do objeto, um objeto *SoftReference* é criado (poderia ser *WeakReference* também) recebendo no momento da criação a referência forte *temp* para o objeto referente.

```
Objeto temp = new Objeto();
SoftReference<Objeto> forte = new SoftReference<Objeto>(temp);
```

Em seguida, eliminamos todas as referências fortes que referem-se ao objeto diretamente:

```
temp = null;
```

Agora só temos uma referência forte, e ela aponta para um objeto *SoftReference*. Não há mais referências fortes que apontem diretamente ao objeto criado. A única forma de obter a instância que criamos – o objeto referente – é através da referência *forte* para o objeto *SoftReference*, que está ligado a ele através de uma referência fraca.

Para obter o e poder usar a instância do objeto referente, utilizamos o método *get()*[25]:

```
Objeto temp = forte.get();
```

Se o objeto já tiver sido coletado, ou se o método *clear()* tiver sido chamado, *get()* retornará *null*.

Uma vez criada, uma referência fraca é imutável. Não pode apontar para outro objeto. Pode ser esvaziada chamando o método *clear()* mas não pode ser reutilizada.

Alcançabilidade fraca e forte

Referências fracas redefinem estados de alcançabilidade. Um objeto é fortemente alcançável (strongly reachable) quando, a partir do *conjunto raiz de referências*, ele é alcançável através de uma corrente de referências comuns.

Se a *única forma* de alcançar um objeto envolver a passagem por *pelo menos uma* referência fraca, ele é chamado informalmente de fracamente alcançável (*weakly reachable*), como mostra a *figura 60*. Um objeto fracamente alcançável é um objeto que pode tornar-se inalcançável a qualquer momento.

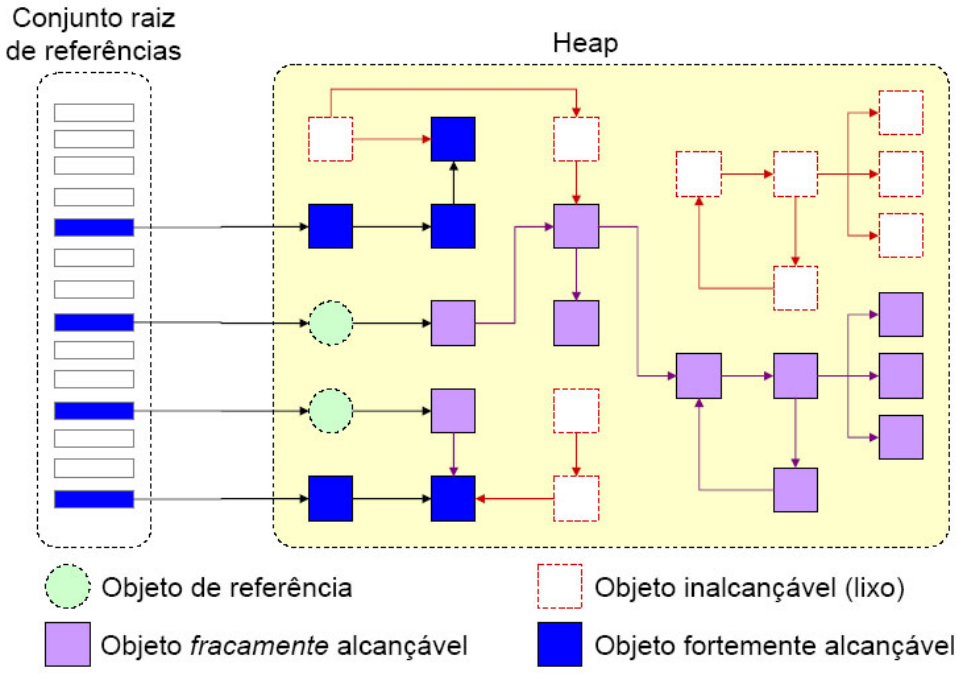

Figura 60 – Objetos alcançáveis apenas via objetos de referência são fracamente alcançáveis.

[25] Caso não se utilize genéricos, é necessário fazer o cast para converter a referência devolvida pelo método *get()*, que é do tipo *Object*.

O termo *fracamente alcançável* é um termo genérico para qualquer referência criada através das subclasses de *Reference*. Formalmente, a API define *três níveis de força* para a alcançabilidade fraca com base no uso das classes *SoftReference*, *WeakReference*, ou *PhantomReference*.

Força da alcançabilidade

Objetos podem ser classificados quanto à força da sua alcançabilidade em

- *Strongly reachable* (fortemente alcançável): objetos que têm referências normais e que não estão elegíveis à coleta de lixo;
- *Softly reachable* (levemente alcançável): objetos que são acessíveis através de uma *SoftReference* e podem ser finalizados e coletados quando o coletor de lixo precisar liberar memória;
- *Weakly reachable* (fracamente alcançável): objetos que acessíveis através de uma *WeakReference* e podem ser finalizados e coletados a qualquer momento (assim que ocorrer uma coleta menor).
- *Phantomly reachable* (alcançável após a finalização): objetos acessíveis através de uma *PhantomReference*; são objetos já finalizados que esperam autorização para que o espaço que ocupam seja reciclado pelo coletor de lixo. O referente de um *PhantomReference* não é mais utilizável.
- *Unreachable* (inalcançável): são objetos que não têm mais referência alguma para eles, e que serão coletados.

Com objetos de referência, o diagrama de transição de estados mostrado na *figura 53* precisa ser redesenhado para levar em conta os novos estados intermediários criados pelas referências fracas. Esse diagrama está mostrado na *figura 61*.

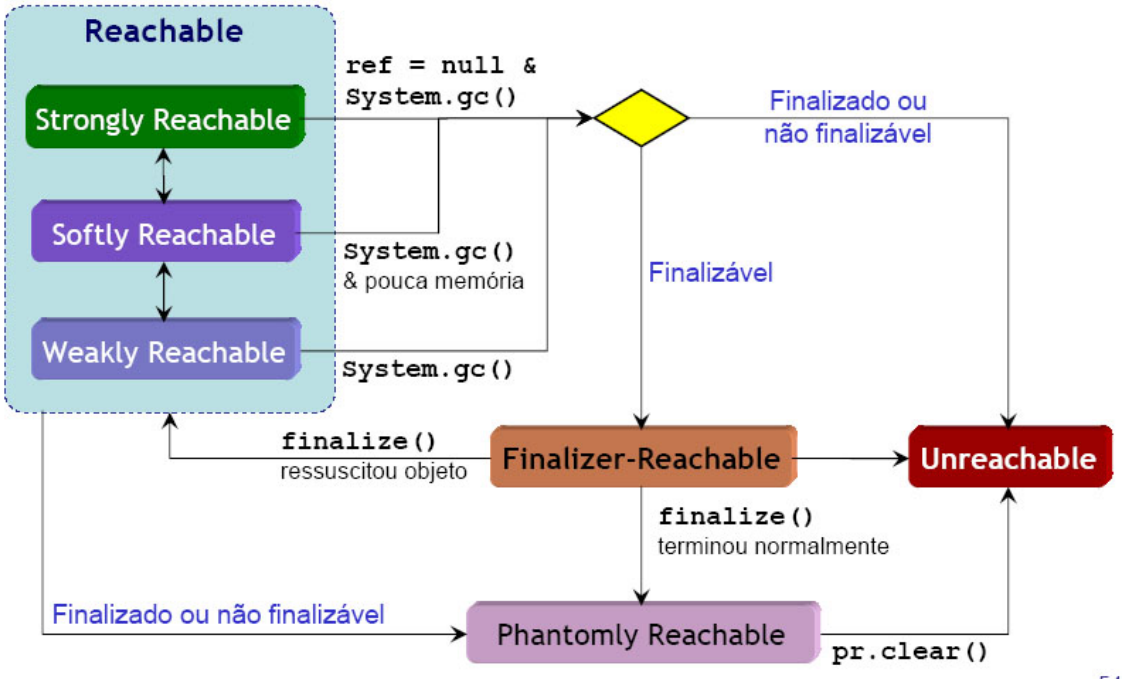

Figura 61 – Transição de estados com objetos de referência. Compare com a figura 53.

SoftReference e WeakReference

Essas duas classes são estratégias muito similares. Elas diferem apenas na forma do tratamento recebido pelo coletor de lixo. O coletor de lixo sempre coleta objetos fracamente acessíveis via *WeakReference*, mas só coleta objetos fracamente acessíveis via *SoftReference* quando não houver mais como alocar memória sem removê-lo. Em ambos os casos, os objetos mais antigos são removidos primeiro, embora isto não faça diferença no caso dos objetos referenciados via *WeakReference*.

A *tabela 4* compara as duas estratégias.

WeakReference	SoftReference
Mantém referência para objetos ativos somente enquanto estiverem em uso (alcançáveis, tendo uma referência forte).	Mantém referência para objetos ativos desde que haja memória suficiente, mesmo que não estejam em uso.
O coletor de lixo poderá liberar objetos que só tenham referências desse tipo a qualquer momento (sempre que executar.)	O coletor de lixo só terá que liberar objetos que só tenham referências desse tipo antes de lançar um *OutOfMemoryError*.
O coletor de lixo não toma decisões antes de liberar a memória usada por seus referentes.	O algoritmo do coletor de lixo obedece a uma política de liberação de de seus referentes.
Se o coletor rodar e houver *WeakReferences*, seus referentes serão removidos.	O coletor de lixo só remove *SoftReferences* se não tiver outra opção.
Use para objetos que têm vida curta: cliente decide reaver objeto logo ou não volta mais.	Use quando existir a possibilidade do cliente voltar e tentar reaver objeto após algum tempo.

Tabela 4 – Comparação entre SoftReference e WeakReference

Pode-se ajustar a política de liberação da memória ocupada por *SoftReferences* quando a memória estiver no fim, através de opções disponíveis em máquinas virtuais *HotSpot*, da *Sun* a partir da versão 5.0. A opção é

`-XX:SoftRefLRUPolicyMSPerMB=`*`taxa`*

A *taxa* é o valor em milissegundos por megabyte do *heap* na qual a máquina virtual remove referentes acessíveis via *SoftReference* quando necessário. A *HotSpot Client VM* considera o valor relativo ao tamanho atual (utilizado) do heap, e a *HotSpot Server VM* considera o valor relativo ao heap máximo (parâmetro *-Xmx*). Por exemplo, a chamada:

```
java -XX:SoftRefLRUPolicyMSPerMB=1000 ...
```

configura a máquina virtual para que referentes fracamente acessíveis via *SoftReference*, usados há mais tempo (*LRU = Least Recently Used*) durem pelo menos um segundo para cada megabyte livre. Se houver 60 megabytes livres, a liberação será adiada um minuto.

O exemplo abaixo mostra uma pilha cujos objetos são guardados em *WeakReferences*. A manutenção das referências é de responsabilidade do cliente, uma vez que na primeira coleta de lixo os dados guardados não estarão mais

disponíveis. Ou seja, depois que as referências do cliente forem perdidas (depois do *push*), existe a possibilidade de perda de dados.

```
public class VolatileStack { // não é thread-safe!
    private Reference[] elements;
    private int size = 0;
    public VolatileStack(int initialCapacity) {
        this.elements = new Reference[initialCapacity];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = new WeakReference(e);
    }
    public Object pop() {
        if (size == 0)
              throw new EmptyStackException();
        Reference ref = elements[--size];
        return ref.get();   // pode retornar null!!
    }
    public int size() { return size; }
    private void ensureCapacity() { ... }
}
```

Uma implementação mais segura seria usando *SoftReferences*. Com eles, objetos duram muito mais, embora ainda dependam do cliente e do coletor de lixo. Neste caso, mesmo que o cliente perca as referências, os elementos só serão coletados se faltar memória, e os mais novos serão os últimos.

```
public class LessVolatileStack { // não é thread-safe!
    private Reference[] elements;
    private int size = 0;
    public LessVolatileStack(int initialCapacity) {
        this.elements = new Reference[initialCapacity];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = new SoftReference(e);
    }
    public Object pop() {
        if (size == 0)
              throw new EmptyStackException();
        Reference ref = elements[--size];
        return ref.get();
    }
    public int size() { return size; }
    private void ensureCapacity() { ... }
}
```

SoftReferences são a escolha ideal para *caches* pois manterão um objeto ativo o máximo de tempo possível. O exemplo abaixo ilustra a implementação de um cache que guarda os dados de arquivos lidos. Se o objeto não estiver mais disponível, ele será novamente carregado do disco.

```
public class FileDataCache {
   private Map map = new HashMap();//<String, SoftReference<Object>>
   private Object getFromDisk (String fileName) {
       Object data = null;
       try {
           data = readFile(fileName);
       } catch (IOException e) { ... }
       map.put(fileName, new SoftReference(data));
       return data;
   }
   public Object getFromCache(String fileName) {
       Reference ref = map.get(name);
       if (ref.get() == null)
           return getFromDisk(fileName);
       else return ref.get();
   }
   private Object readFile(String fileName)
                throws IOException { ... }
   ...
}
```

ReferenceQueue

A classe *ReferenceQueue* pode ser usada para responder a eventos causados pela coleta de objetos referentes. *ReferenceQueue* implementa uma fila de objetos de referência normalmente preenchida pelo coletor de lixo. A fila recebe uma referência *weak* ou *soft* algum tempo depois do referente tornar-se inalcançável. Referências *phantom* são adicionadas à fila depois que o objeto referente foi finalizado.

ReferenceQueue pode ser usada como mecanismo de notificação, e de pré- ou pós-finalização. É sempre passada na criação do objeto.

```
ReferenceQueue q = new ReferenceQueue();
Reference ref = new SoftReference(referent, q);
```

A classe *ReferenceQueue<T>* possui três métodos que fazem a mesma coisa: removem objetos da fila. Todos retornam *Reference<T>*.

- *remove()* e *remove(long timeout)*: bloqueiam o *thread* no qual executam enquanto não houver elementos para retirar. Podem ser interrompidos (*InterruptedException*) através de *Thread.interrupt()*.
- *poll()*: retorna *null* enquanto não houver objetos na fila. Quando um referente for coletado, seu objeto de referência aparecerá na fila e poderá ser removido através deste método.

Os métodos de *ReferenceQueue* não servem para recuperar o referente, pois quando um objeto de referência cai na fila, seu referente já foi coletado, portanto uma chamada ao método *get()* em um objeto de referência retirado da fila *sempre* retorna *null*.

ReferenceQueue funciona de forma semelhante com *WeakReference* e *SoftReference*, mas bastante diferente se for usado com *PhantomReference*. As *figuras 62* e *63* ilustram as diferenças.

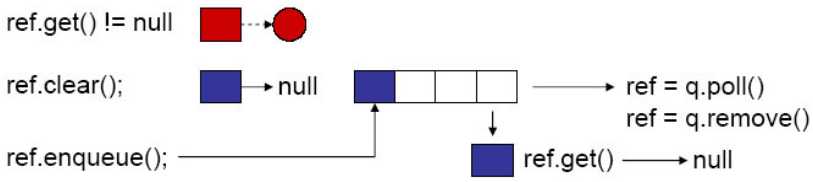

Figura 62 – Funcionamento de ReferenceQueue*: com referências* Weak *e* Soft *chamar* clear()*, coloca objeto na fila depois de algum tempo.*

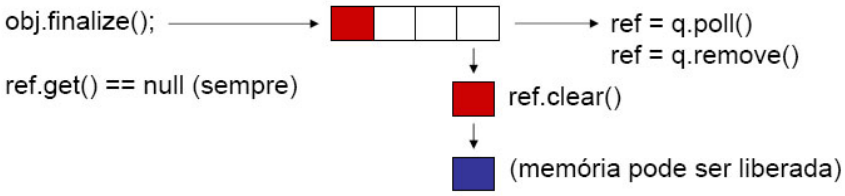

Figura 63 – Funcionamento de ReferenceQueue*: com referências* Phantom*, o objeto "nasce" na fila; chamar* clear() *sobre o objeto, tira-o da fila.*

O próximo exemplo usa *ReferenceQueue* para saber quando um objeto foi coletado em um mapa onde as chaves são *Strings* e os valores são objetos *Reference*. O *thread* remove entradas de um *Map* quando suas referências *weak* tornam-se inalcançáveis. Se um objeto foi coletado, seu objeto *Reference* será colocado na fila, ativando o método *remove()* que retorna o *Reference*. O objeto é localizado no mapa e usado para achar sua chave, que é removida.

```
Map map = new HashMap(); // <String, Reference<Object>>
ReferenceQueue queue = new ReferenceQueue();

Runnable queueThread = new Runnable() {
   public void run() {
     while(!done) {
       Reference ref = null;
       try { ref = queue.remove(); // blocks
       } catch (InterruptedException e) {done = true;}
       Set entries = map.entrySet();
       for (Map.Entry entry: entries) {
         if(entry.getValue() == ref) {
           String key = entry.getKey();
           key = null;
           map.remove(key);
         } // if
        } // for
      } // while
   } // run()

};
new Thread(queueThread).start();
```

Finalização com referencias fracas

Objetos de referência podem ser usados como uma alternativa ou complemento à finalização através da captura de eventos de alcançabilidade usando *ReferenceQueue*. Duas alternativas são possíveis:

- *Pré-finalização*, realizada quando a referência do objeto estiver perdida, que pode ocorrer quando a memória estiver no limite (*soft reference*) ou quando o coletor de lixo executar (*weak reference*);
- *Pós-finalização*, realizada depois que objeto estiver finalizado (*phantom reference*).

Para implementar, a solução padrão é criar um *thread* que use *poll()* ou *remove()* para descobrir quando um objeto perdeu sua referência fraca. O trecho de código abaixo cria um *thread* que chama um finalizador explícito quando o objeto *obj* (ou qualquer outro que estiver na fila) for coletado.

```
Runnable finalizer = new Runnable() {
  public void run() {
    while(q.poll() == null) { // espera que objeto apareça
      try {Thread.sleep(32);} catch(...) {}
    }
    close(); // finalização
  }
};
new Thread(finalizer).start();

ReferenceQueue q = new ReferenceQueue();
Reference ref = new WeakReference(obj, q);
```

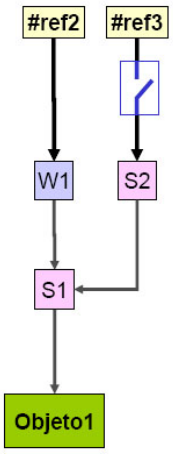

Figura 64

Pode-se também ter controle adicional sobre a liberação de memória usando referências encadeadas e suas regras de precedência. Por exemplo, a *figura 64* ilustra um objeto com dois caminhos passando por referências fracas: *#ref2* é uma *WeakReference* e *#ref3*, uma *SoftReference*.

Enquanto existir *#ref3*, o objeto será tratado como levemente (*softly*) acessível e só será removido se faltar memória. Mas se *#ref3* for perdida, o único caminho para *Objeto1* é fracamente acessível via *#ref2*, podendo ser removido a qualquer momento.

Em um único caminho, contendo uma *série* de referências interligadas, a referência *mais fraca* determina a alcançabilidade do objeto. Se houver vários caminhos *paralelos* de referências encadeadas em série para um objeto, sua alcançabilidade é determinada pelo caminho *mais forte* que houver. Isto é ilustrado na *figura 65*.

Na figura, como *Objeto2* possui apenas um caminho: via *#ref1*, é a referência mais fraca que irá determinar sua alcançabilidade, que é *phantomly reachable*.

Já o *Objeto1* possui dois caminhos: um passando por uma *Weak Reference* (*#ref2*), e outro passando por uma *Soft Reference* (*#ref3*). Prevalece o caminho mais forte (*#ref3*) e o objeto é *softly reachable*.

Finalmente o *Objeto3* é fortemente acessível pois dois seus dois caminhos, um deles (#ref4) não passa por nenhum objeto de referência. Se *#ref4* for perdida, o objeto ainda será fracamente acessível através da referência *#ref5*.

Em aplicações típicas, referências fracas são usadas de forma bem mais simples. Não é comum encontrar programas usando várias referências de tipos diferentes encadeadas.

O processamento do objeto durante uma coleta de lixo é realizado sempre pelo caminho mais forte e acontece na ordem abaixo:

1. *Soft references*
2. *Weak references*
3. Finalização de objetos
4. *Phantom references*
5. Liberação de memória

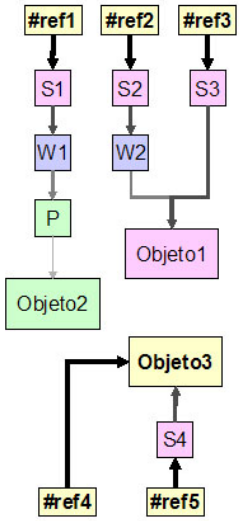

Figura 65

Não há garantia de *quando* o processamento em cada etapa irá ocorrer, já que tanto a liberação de memória quanto a finalização de objetos dependem de agendamento próprio do algoritmo de coleta de lixo, cuja ocorrência e comportamento não são controláveis através de programação.

Referências fantasma

Objetos acessíveis através de referências do tipo *PhantomReference* já foram finalizados (seu método *finalize()* já foi chamado, se existir) mas ainda não foram liberados. Estão mortos. Não podem mais ser usados e nem ressuscitados! Permitem, porém, disparara a realização de operações *pós-morte*, pois podem ser identificados através de suas referências fracas.

Todo *PhantomReference* tem um *ReferenceQueue*. Não há como criar um *PhantomReference* sem passar seu *ReferenceQueue* na construção. Fantasmas são colocados no seu *ReferenceQueue* logo que se tornam *phantomly reachable*. Pode-se, então, pesquisar a fila, retirar os objetos de referência que forem se acumulando e através deles identificar os referentes já mortos.

PhantomReferences na verdade sequer são referências. São apenas vestígios de um objeto finado. Porém, é preciso eliminar esses fantasmas depois que não forem mais úteis. Chamar *clear()* em um *PhantomReference* é necessário para retira-o da fila e permitir que ele seja finalizado (o próprio fantasma, não o referente que já é finado). O método *clear()* de *PhantomReference* faz o contrário de *clear()* em *WeakReference* ou *SoftReference*. Nos últimos, é chamado para por a referência na fila enquanto que em *PhantomReference* serve para retirá-la da fila. A *figura 63* ilustra este comportamento. Se *clear()* não for chamado, o *PhantomReference* nunca será retirado da fila e sua memória nunca será liberada, o que representa um *memory leak*.

PhantomReferences podem ser usados como alternativa à finalização automática, mas não há garantia usá-los seja muito mais confiável que simplesmente usar *finalize()*, já que ainda dependem da finalização e agendamento de liberação de objetos pelo coletor de lixo.

O trecho de código abaixo mostra como poderia ser implementado um mecanismo de finalização automática com *PhantomReferences*:

```
ReferenceQueue q = new ReferenceQueue();
Reference ref    = new PhantomReference(obj, q);
Runnable finalizer = new Runnable() {
  public void run() {
    Reference ref = null;
    while( (ref = q.poll()) == null) {
      try {Thread.sleep(32);} catch(...) {}
    }
    ref.clear(); // libere o fantasma!
    close(); // finalize o objeto
  }
};
new Thread(finalizer).start();
```

Depois que o objeto referente *obj* estiver finalizado, ele irá aparecer na fila *q* e será retornado pelo método *poll()*. A linha *ref.clear()* é necessário para liberar a memória ocupada pelo fantasma. Em seguida, o método de finalização é chamado.

Neste outro exemplo[26], retornamos à casa mal-assombrada do *capítulo 11* e reescrevemos o finalizador para que ele guarde uma cópia serializada do objeto morto para uma possível ressurreição pós-finalização (uma múmia!). Assim, o morto, mesmo que não mais possa ressuscitar após duas mortes, pode ser trazido de volta à vida, quem sabe, uma terceira vez, na forma de uma cópia, durante a pós-finalização.

Primeiro, reescrevemos o finalizador (veja o original no *capítulo 11*), que grava uma cópia do objeto em */tmp/mummy*:

```
public class RessurectableGuest extends Guest { ...
  protected void finalize() ... {
    try {
      ObjectOutputStream mummy = new ObjectOutputStream(
         new FileOutputStream("/tmp/mummy"));
      mummy.writeObject(this);
      mummy.close();
    } finally {
      super.finalize();
    }
  }
}
```

Depois, em uma aplicação usando *PhantomReference*, pesquisamos sua fila queue e esperamos que o fantasma apareça depois qude o objeto for finalizado. Como a finalização é garantida antes que um *PhantomReference* apareça na fila, sabemos que ele deve ter sido serializado, então procuramos o arquivo e criamos uma nova cópia do objeto, que mais uma vez volta à vida.

---

[26] Este é mais um exemplo meramente educativo para explicar o funcionamento das referências fantasma. Não tente fazer algo parecido em aplicações "sérias".

```
Reference found = queue.remove();
if (found != null) { // uma Reference foi encontrada!
try {
   ObjectInputStream openMummy =
     new ObjectInputStream(
       new FileInputStream("/tmp/mummy"));
   Guest ressurected =
       (Guest)openMummy.readObject(); // objeto criado!
   hauntedHouse.addGuest(ressurrected); // volta à casa!
} catch (Exception e) {...}
```

WeakHashMap

A classe *java.util.WeakHashMap* é a implementação de um *java.util.Map* onde o par chave/valor é uma *WeakReference*. É uma classe utilitária que implementa o uso mais comum de *WeakReferences*. A *figura 66* ilustra a estrutura da classe *WeakHashMap*.

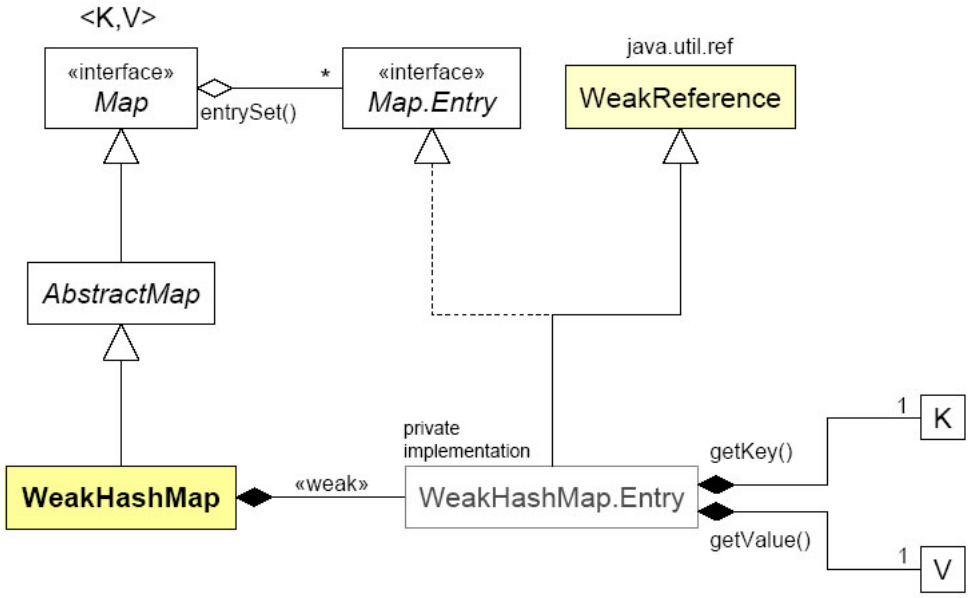

Figura 66 – Hierarquia de classes associadas à classe java.util.WeakHashMap.

Depois que o objeto referenciado pela chave fraca torna-se fracamente alcançável, o coletor de lixo pode limpar a referência interna. A chave e seu valor associado tornam-se elegíveis à finalização.

WeakHashMap é a escolha ideal para mapas onde objetos podem ficar obsoletos rapidamente. Pode ser usado para implementar caches sensíveis à memória, listas de event handlers, etc. É uma forma de evitar os *memory leaks* mais comuns, porém há risco de perda de dados. Como usa *WeakReferences*, o coletor de lixo pode liberar sua memória a qualquer momento. Se houver um grande consumo de memória em outra parte da aplicação e isto causar coletas de lixo freqüentes, as chaves do *WeakHashMap* serão continuamente perdidas. Deve-se considerar a construção de um *SoftHashMap*[27] se volatilidade do *WeakHashMap* for um problema.

[27] Não existe *SoftHashMap* na API.

WeakHashMap pode ser usada em qualquer lugar que se usa um *HashMap*. A aplicação abaixo, por exemplo, possui um *memory leak*. Ela não pára de acrescentar novos objetos em um *HashMap*, o que irá, eventualmente levar a ocorrência de um *OutOfMemoryError*.

```
public class MemoryLeak {
  public static void main(String[] args) {
    Map<Integer, String> map =
          new HashMap<Integer, String>();
    int i = 0;
    while( true ) {
      String objeto = new String("ABCDEFGHIJKLMNOQRSTUVQWXYZ");
      System.out.print(".");
      try {Thread.sleep(100);} catch (InterruptedException e) {}
      map.put(++i, objeto);
    }
  }
}
```

É fácil corrigir o *memory leak*. Simplesmente mudando o *HashMap* para *WeakHashMap* pode-se garante-se que a memória não acabará por excesso de elementos no *HashMap*. A desvantagem é que se outros processos causarem uma coleta de lixo, os objetos também serão perdidos (será preciso incluir um mecanismo para gerenciar esse risco.)

```
public class FixedMemoryLeak {
  public static void main(String[] args) {
    WeakHashMap<Integer, String> map =
          new WeakHashMap<Integer, String>();
    int i = 0;
    while( true ) {
      String objeto = new String("ABCDEFGHIJKLMNOQRSTUVQWXYZ");
      System.out.print(".");
      try {Thread.sleep(100);} catch (InterruptedException e) {}
      map.put(++i, objeto);
    }
  }
}
```

Conclusões

A finalização e destruição de objetos em Java é controlada por algoritmos de coleta de lixo. É possível ter um controle limitado sobre o coletor de lixo usando finalizadores automáticos, chamadas explícitas (*System.gc*) e objetos de referência. Das três opções, objetos de referência são a que oferece mais controle. Há três diferentes tipos de objetos de referência: *WeakReference*, que oferece uma ligação que se perde na primeira coleta de lixo, *SoftReference*, que dura pelo menos até que a memória acabe, e *PhantomReference*, que não está mais ligado ao objeto mas permite a realização de tarefas ligadas à sua destruição. Os três tipos de objetos de referência permitem construir aplicações que gerenciam um pouco o uso de memória ao flexibilizar a ligação de objetos com suas referências, e permitir a captura de eventos ligados à liberação de memória e finalização.

© 2005 Helder da Rocha

Referências

[Collins 60] G. Collins. *A Method for Overlapping and Erasure of Lists*, IBM, CACM, 1960. Algoritmo de contagem de referências.

[McCarthy 60] J. McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, MIT, CACM, 1960. Artigo original do Mark-Sweep algorithm (em Lisp).

[Edwards] D.J. Edwards. *Lisp II Garbage Collector. MIT. AI Memo 19. ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-019.ps*. Mark-Compact.

[Cheney 70] C. J. Cheney. *A Nonrecursive List Compacting Algorithm*. CACM, Nov 1970. Artigo original do copying algorithm.

[Baker 78] H. G. Baker. *List processing in real time on a serial computer*. CACM, Apr 1978. Uma versão concorrente do copying algorithm.

[Lieberman-Hewitt 83] H. Lieberman, C. Hewitt. *A Real Time Garbage Collector Based on the Lifetimes of Objects*. CACM, June 1983. Artigo principal do Generational GC.

[Dijkstra 76] E. W. Dijkstra, L. Lamport, et al. *On-the-fly Garbage Collection: An Exercise in Cooperation*. Lecture Notes in Computer Science, Vol. 46. 1976. Tri-color marking (citado em [Jones & Lins 95]).

[Bobrow 80] D. Bobrow. *Managing reentrant structures using reference counts.* ACM/PLS, Jul 1984. Contagem de referências com coleta de ciclos.

[Ungar 84] David Ungar. Generation Scavenging: *A Non-disruptlve High Performance Storage Reclamation Algorithm*. ACM, 1984. Um dos artigos do *Generational GC*.

[Hudson & Moss 92] R. Hudson, J.E.B. Moss. *Incremental Collection of Mature Objects*, ACM/IWMM, Sep 1992. Artigo do Train algorithm.

[Domani 00] T. Domani et al. *A Generational On-The-Fly Garbage Collector for Java*, IBM 2000.

[Printezis 00] Tony Printezis and David Detlefs. *A Generational Mostly-concurrent Garbage Collector*, 2000. Algoritmo usado no HotSpot.

[Flood et al 02] Christine Flood et al. *Parallel Garbage Collection for Shared Memory Multiprocessors*. Sun Microsystems. Usenix, 2001. Algoritmos usados no HotSpot.

[Bacon-Rajan 01] D. Bacon, V. T. Rajan. *Concurrent Cycle Collection in Reference Counted Systems.* IBM, 2001.

[Levanoni-Petrank 01] Y. Levanoni, E. Petrank. *An On-the-fly Reference Counting Garbage Collector for Java*, IBM, 2001.

[Azatchi 03] H. Azatchi et al. *An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views*. OOPSLA 03, ACM, 2003.

[Paz 05] H. Paz et al. *Efficient On-the-Fly Cycle Collection*. IBM (Haifa), 2005.

[Paz-Petrank-Blackburn 05] H. Paz, E. Petrank, S. Blackburn. *Age-Oriented Concurrent Garbage Collection*, 2005.

[Memory] *The Memory Management Reference*. *http://www.memorymanagement.org/*. Várias referências e textos sobre gerência de memória em geral.

[JVMS] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification, second edition*, Sun Microsystems, 1999. Formato de memória, pilha, heap, registradores na JVM.

[Sun 05] Sun Microsystems. *Tuning Garbage Collection with the 5.0 Java[tm] Virtual Machine*. 2005. Generational GC e estratégias paralelas no HotSpot.

[HotSpot] Sun Microsystems. *The Java HotSpot™ Virtual Machine, v1.4.1, Technical White Paper*. Sept. 2002. Algoritmos usados no *HotSpot*.

[SDK] *Documentação do J2SDK 5.0*. Sun Microsystems, 2005.

[Apple 04] *Java Development Guide for MacOS X*. Apple, 2004

[Printezis 05] Tony Printezis. *Garbage Collection in the Java HotSpot Virtual Machine*. *http://www.devx.com/Java/Article/21977*, DevX, 2005.

[Jones & Lins 96] R. Jones, R.Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley 1996. Várias estratégias de GC explicadas.

[Venners] Bill Venners, *Inside the Virtual Machine*. Applet Heap of Fish: *http://www.artima.com/insidejvm/applets/HeapOfFish.html*

[Gotry 02] K. Gottry. *Pick up performance with generational garbage collection*. JavaWorld *www.javaworld.com*. Jan 2002

[Gupta 02] A. Gupta, M. Doyle. *Turbo-charging Java HotSpot Virtual Machine, v1.4 to Improve the Performance and Scalability of Application Servers*. Sun, 2002. *http://java.sun.com/developer/technicalArticles/Programming/turbo*

[Nagarajayya 02] N.Nagarajayya, J.S. Mayer. *Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1*. Sun Microsystems. Nov 2002

[Holling 03] G. Holling. *J2SE 1.4.1 boosts garbage collection*. JavaWorld. Mar 2003.

[Goetz 03] B. Goetz. *Java theory and practice: Garbage collection in the HotSpot JVM*. IBM Developerworks. Nov 2003.

[Pawlan 98] Monica Pawlan, *Reference Objects and Garbage Collection*, Sun Microsystems, JDC, August 1998. Um tutorial abrangente sobre objetos de referência. *http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/*

[Tate 02] [BJ] Bruce Tate, *Bitter Java*, Manning, 2002. Contém discussão interessante sobre *memory leaks.*

[Bloch 01] [EJ] Joshua Bloch, *Effective Java*, Addison-Wesley, 2001. Contém padrão *finalizer guardian*, discussão sobre *finalize* e *memory leaks.*

[Friesen 02] *Trash Talk part 2: Reference Objects*. JavaWorld, Jan 2002. *http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-java101.html*

www.argonavis.com.br