**Garbage Collector**

O estudo de **Garbage Collection** é importante para os desenvolvedores que desejam entender o gerenciamento de memória na JVM e que se preocupam em escrever aplicações que tenham um bom desempenho e sejam escaláveis.

Na introdução do documento “The Java Language Specification” – que descreve a especificação completa da sintaxe e semântica do Java –, os autores afirmam que Java é uma linguagem de programação baseada em C/C++, mas que é organizada de maneira bastante diferente.

A especificação da linguagem reforça que na construção do Java foram omitidos alguns aspectos de C/C++ e incluídas algumas ideias de outras linguagens. Por exemplo, um dos recursos incluídos em Java é o gerenciamento automático de memória, o qual utiliza um Coletor de Lixo (Garbage Collector) que tem a finalidade de evitar os problemas de desalocação explícita dos espaços ocupados por objetos não mais referenciados.

Esse gerenciamento explícito é efetivado com a função free em C ou com o operador delete em [**C++**](https://www.devmedia.com.br/curso/curso-de-introducao-a-c/274). A técnica de Coleta de Lixo consiste na recuperação segura do espaço de memória ocupado por um objeto que não é mais referenciado dentro de uma aplicação. Esta técnica é o tema que será estudado nesta matéria.

No entanto, antes de iniciarmos essa discussão, julgamos relevante expor quais são as vantagens desse gerenciamento automático.

Denomina-se gerenciamento de memória em linguagens de programação orientadas a objetos, o processo de reconhecimento quando objetos alocados não são mais referenciados e a posterior liberação da memória usada por tais objetos, tornando-a assim disponível para futuras alocações. Na **Listagem 1** mostramos um exemplo de um objeto alocado que deixa de ser referenciado.

Integer i = new Integer(10);

i = null;

**Listagem 1**. Exemplo de um objeto não mais referenciado

Note na **Listagem 1**, que a variável **i** inicialmente refere-se a um Integer e em seguida a mesma referência recebe null, deixando então de apontar para o objeto.

Ou seja, nesse momento o Integer criado anteriormente não está mais sendo referenciado, permitindo assim que a memória ocupada por ele seja liberada.

Em algumas linguagens, tais como C e C++, como visto antes, a tarefa de gerenciamento de memória – a qual chamamos de gerenciamento explícito – é de responsabilidade do programador.

Nessas linguagens onde o gerenciamento de memória é explícito, podem ocorrer erros, resultados não previstos ou até mesmo o fechamento inesperado (crash) da aplicação.

Um dos problemas que ocorrem quando o gerenciamento de memória é feito pelo programador é o de referências pendentes (dangling references). Esta falha acontece se o programador desaloca o espaço ocupado por um objeto que ainda está sendo referenciado.

Outro problema com o gerenciamento explícito é o [**vazamento de memória**](https://www.devmedia.com.br/memory-leak-eliminando-vazamentos-de-memoria-em-java/28349) (space leaks). O vazamento ocorre quando um objeto deixa de ser referenciado e sua memória não é liberada. Caso aconteça muito vazamento, o aumento de consumo do espaço de armazenamento pode levar ao esgotamento da memória.

Uma alternativa ao gerenciamento explícito é o gerenciamento automático feito por um **programa denominado Garbage Collector**. Este mecanismo é empregado em linguagens orientadas a objetos mais modernas, tais como o Java.

Com o **uso do Garbage Collector** o problema de referências pendentes nunca acontece, pois um objeto que ainda está sendo referenciado jamais será candidato – ou elegível – à coleta de lixo e seu espaço de memória não ficará livre.

Esta maneira de gerenciar o espaço também resolve o problema do vazamento de memória, visto que toda memória não mais referenciada é liberada automaticamente.

Mas antes de começarmos a estudar o funcionamento do Garbage Collector, precisamos aprender onde as variáveis e objetos são alocados em Java. É sobre isso que discutiremos na primeira seção desta matéria.

#### A Pilha (Stack) e o Heap

Em Java o programador precisa se preocupar com duas áreas de memória – a área onde os objetos são armazenados (heap), e a área onde vivem as variáveis locais e as chamadas de métodos (pilha).

As variáveis locais podem ser primitivas ou referências a objetos e são conhecidas como variáveis de pilha, o que reforça o que afirmamos anteriormente – tais variáveis são armazenadas na pilha. As variáveis primitivas possuem um tipo e tamanho, os quais determinam o espaço de memória ocupado pelo valor da variável.

Por exemplo, se declararmos int a = 10, isto indica que o valor 10 será armazenado na pilha como um número inteiro de 32 bits. Por sua vez, as referências não armazenam o objeto propriamente dito, mas uma informação – algo como um endereço ou ponteiro – sobre onde localizar o objeto.

Vejamos então o que acontece quando definimos, por exemplo, Pessoa p = new Pessoa() em um programa:

1. Pessoa p diz à Máquina Virtual Java (JVM) para alocar espaço com esse nome para uma variável de referência na pilha;
2. new Pessoa() diz à JVM para alocar espaço para um novo objeto Pessoa no heap;
3. Finalmente ocorre a ligação do objeto à variável de referência, através do operador de atribuição (=).

Na **Listagem 2** mostramos o código de uma classe, cuja representação de sua execução na pilha e no heap podem ser vistos na **Figura 1**.

public class Retangulo {

private double base;

private double altura;

public double getBase() {

return base;

}

public void setBase(double base) {

this.base = base;

}

public double getAltura() {

return altura;

}

public void setAltura(double altura) {

this.altura = altura;

}

public static void main(String[] args) {

Retangulo r = new Retangulo();

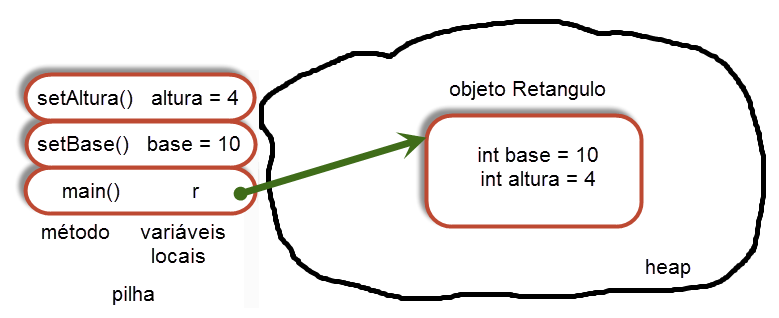
r.setBase(10);

r.setAltura(4);

}

}

**Listagem 2**. Classe para ilustrar a representação de variáveis na pilha e no heap

**Figura 1**. Representação das variáveis na pilha e no heap quando a classe Retangulo é executada

Observando a **Figura 1** notamos que, quando a classe é executada, inicialmente é colocado no topo da pilha o método main(), assim como as variáveis locais e/ou referências declaradas dentro dele. Neste caso, r é uma referência, a qual irá manter a informação de como localizar o objeto Retangulo criado no heap.

Em seguida, cada método que é invocado – setBase() e setAltura(), nessa ordem – vai sendo empilhado com as variáveis locais correspondentes.

Além dessas, existem em Java as variáveis de instância. Tais variáveis são declaradas dentro de uma classe, mas não dentro de um método. Elas são, na verdade, os campos ou atributos de um objeto. As variáveis de instância vivem dentro do objeto ao qual elas pertencem. Na **Listagem 3** ilustramos um exemplo.

public class Pessoa {

// variáveis de instância "nome" e "idade"

String nome;

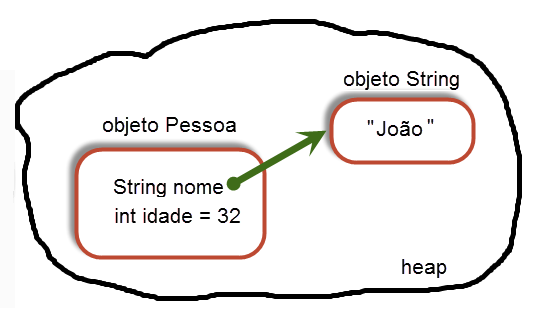
int idade;

}

**Listagem 3**. Exemplo de variável de instância

Note na **Listagem 3** que foram definidas duas variáveis de instância: uma que é primitiva e outra que é um objeto. As variáveis de instância primitivas vivem dentro do objeto, assim como seus valores.

Por sua vez, as variáveis de instância que são referências também são alocadas dentro do objeto, mas os objetos referenciados por elas são alocados no heap, conforme podemos ver com o auxílio da **Figura 2**, onde é visualizada uma instância da classe Pessoa.

**Figura 2**. Alocação de variáveis de instância no heap

Observe na **Figura 2**, onde representamos um objeto Pessoa e o objeto String correspondente ao nome da pessoa. Este exemplo ilustra o que foi afirmado anteriormente.

A variável de instância nome é alocada dentro do objeto Pessoa – residente no heap – e o objeto String que mantém o nome da pessoa também reside no heap.

Visto que estamos falando sobre o uso da memória, julgamos importante destacar um mecanismo utilizado pela JVM para tornar o uso deste recurso mais eficiente, o “pool de constantes String”.

Neste pool ficam armazenados todos os literais String. Literais são constantes que podem ser expressas no código da aplicação, tal como “abcd” na **Listagem 4**. Quando o compilador encontra um literal String – como na primeira linha da **Listagem 4** –, ele verifica se existe uma **String** idêntica no pool. Se existir, a nova referência é direcionada à String existente e não é criado um novo objeto.

Mas, quando a String é criada usando new, como na segunda linha da **Listagem 4**, este novo objeto é criado no heap, e o literal é inserido no pool de constantes String, se não existir um literal idêntico.

String a = "abcd";

String b = new String("abcd");

**Listagem 4**. Criação de Strings e o uso do pool de constantes String

Após essa visão geral sobre o uso da memória, passaremos então a **falar sobre o Garbage Collector**.

### Garbage Collector

Um Garbage Collector tem a função de:

1. Alocar memória;
2. Assegurar que quaisquer objetos referenciados permaneçam na memória;
3. Recuperar a memória alocada pelos objetos que não são mais alcançáveis pelas referências no código em execução.

Objetos referenciados são denominados vivos, enquanto que são considerados mortos – ou também chamados de lixo – os objetos que não são mais referenciados por qualquer thread em execução. Entende-se por thread como sendo uma linha de execução – um processo – de uma aplicação Java.

Toda aplicação Java possui no mínimo uma thread – o método main(). Este método, necessário para rodar uma classe na JVM, executa em uma thread e é chamado de thread main.

Depois dessa breve explicação sobre threads, vamos prosseguir com a explicação sobre lixo. Diz-se que um objeto X referenciado por x torna-se lixo quando **x** passa a se referir a outro objeto ou a null, ou ainda, se **x** for uma variável local e o programa retorna do método onde x é declarada.

São esses objetos, denominados lixo, que precisam ser localizados e ter seu espaço de memória desalocado no **processo denominado Garbage Collection**.

O controle do Coletor de Lixo é feito pela JVM, que é quem decide quando ele será executado. Mas não existe nada que garanta quando o Garbage Collector será executado, ainda que seja possível ao programador solicitar via código Java que a JVM o execute.

Em geral, quando não há interferência do programador, a máquina virtual executa o Coletor de Lixo quando percebe que a memória está ficando sem espaço. De qualquer maneira, pode ser que uma aplicação encerre sua **execução sem que o Garbage Collector seja executado uma única vez**.

Dessa maneira, pelo que já foi exposto, quando um objeto deixa de ser referenciado, o espaço de memória ocupado por ele não é imediatamente desalocado.

O que se diz de um lixo é que ele – o objeto – se torna qualificado para a coleta de lixo, podendo então ser desalocado na próxima execução do GC (Garbage Collector).

A fim de que o GC execute suas funções adequadamente, é necessário que ele possua algumas características. Por exemplo, o GC deve ser seguro e compreensivo.

Seguro significa que objetos vivos nunca devem ter seu espaço liberado, e para que o GC seja compreensivo, os objetos não mais referenciados não deveriam permanecer sem ser coletados por mais que um pequeno número de ciclos de coleta.

Entende-se por ciclo de coleta o processo que inicia com a marcação dos objetos elegíveis e vai até suas exclusões definitivas.

Também é desejável que um GC opere com eficiência, sem introduzir longas pausas, durante as quais a aplicação parará de executar. No entanto, como acontece com a maioria dos sistemas computacionais, muitas vezes existem conflitos entre tempo, espaço e frequência.

Por exemplo, se o tamanho do heap é pequeno, a coleta vai ser rápida, mas a pilha vai encher mais rapidamente, exigindo, portanto, coletas mais frequentes. Por outro lado, uma pilha de tamanho maior vai demorar mais tempo a encher e assim, as coletas serão menos frequentes, mas podem demorar mais tempo.

Outra característica importante é a limitação da fragmentação. A fragmentação ocorre quando a memória é liberada e o espaço livre pode aparecer em pequenos pedaços em diversas áreas, tais que pode não haver espaço suficiente em uma área contígua qualquer para ser usada para alocação de um determinado objeto. Uma abordagem para eliminar a fragmentação é chamada de compactação.

Devido ao quesito desempenho, uma característica importante é a escalabilidade em aplicações multithread, pois a alocação de espaço de memória não deve se tornar um gargalo para o aumento do número de tarefas concorrentes sendo executadas.

Isso também é válido para aplicações em sistemas com múltiplos processadores. Nas aplicações multithread e em hardware com múltiplos processadores, pressupõe-se a execução de várias tarefas concorrentemente.

Nesses casos, escalabilidade significa aumento de desempenho à medida que se aumenta o número de linhas de execução (threads). Posto que cada tarefa ou linha de execução necessita alocar/desalocar espaço para seus objetos, o Garbage Collector não deve ter influência na queda de performance da aplicação.

Outra questão a respeito do GC é que não se sabe ao certo como ele funciona, visto que na própria especificação da linguagem isto não é descrito. Mas, para que seja possível entender melhor como se dá a coleta de lixo, é necessário que seja utilizado no estudo um modelo simples de algoritmo. Sendo assim, nos próximos parágrafos abordaremos dois desses algoritmos, os quais podem ser empregados para a compreensão do funcionamento do GC.

O processo de coleta de lixo é basicamente dividido em duas fases. A primeira é a separação entre objetos vivos e objetos mortos – já conceituados anteriormente nesta matéria. E a segunda fase é a liberação da memória ocupada pelos mortos.

Um modelo de coleta de lixo normalmente apresentado na literatura é a contagem de referências. Nesse modelo, quando um objeto X referencia outro objeto Y, o sistema incrementa um contador de referências em Y, e quando X deixa de referenciar Y, o contador é decrementado.

Quando o contador chega a zero, Y não está mais vivo, e torna-se qualificado para a coleta de lixo. Dessa maneira, os contadores dos objetos aos quais Y eventualmente se refere também serão decrementados.

No entanto, este modelo não funciona quando X e Y referenciam-se mutuamente. Nesta situação nenhum dos contadores de referência se tornará zero, e nenhum dos dois objetos poderá ser coletado.

E, portanto, também não serão coletados nenhum dos demais objetos que porventura sejam referenciados por X e Y, direta ou indiretamente. Este é um forte motivo para que muitos coletores de lixo não empreguem esta abordagem.

Um modelo também simples e que não apresenta o problema citado anteriormente é o chamado marcar-e-varrer. Este nome é usado devido à maneira que as duas fases de coleta de lixo são implementadas.

Para encontrar objetos vivos, o Coletor de Lixo determina inicialmente um conjunto de raízes que contém aqueles objetos que são referenciados diretamente, tais como aqueles referenciados por variáveis locais.

Depois disso, o Garbage Collector marcará os objetos que são referenciados – alcançáveis – por essas raízes. Após essa etapa, o GC examinará as referências em cada um desses objetos.

Caso um objeto aqui referenciado já tenha sido marcado no passo anterior, ele é ignorado. Caso contrário, o objeto é marcado como alcançável e suas referências são examinadas.

Esse processo continua até que objetos não mais referenciados permaneçam desmarcados. Após concluir essa fase, o Coletor de Lixo pode reclamar os objetos não marcados, varrendo-os da memória.

Durante o processo de marcar-e-varrer, qualquer modificação na interconexão dos objetos interferirá na coleta de lixo. Por exemplo, se um objeto não referenciado passar a ser alcançável enquanto acontece a execução da marcação, tal objeto poderá ser coletado indevidamente. Por esse motivo, a execução do programa deverá ser pausada durante o processo de marcação.

É válido destacar que o algoritmo marcar-e-varrer é apenas um modelo mental para que possamos entender o funcionamento do Coletor de Lixo. As máquinas virtuais utilizam estratégias muito mais complexas para realizar este processo.

### Tornando objetos explicitamente elegíveis à coleta de lixo

Com a finalidade de reforçar o entendimento do que foi discutido nas seções anteriores, vejamos alguns exemplos de código que tornam objetos elegíveis à coleta de lixo.

O primeiro exemplo, e o mais simples, consiste em atribuir null à variável de referência, conforme podemos ver na **Listagem 5**.

public class TesteColetor {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("abcd");

// O StringBuffer é alcançável pela referência sb

// e não é elegível para coleta de lixo

System.out.println(sb);

sb = null;

// Agora o StringBuffer deixa de ser referenciado

// por sb e torna-se candidato à coleta de lixo

}

}

**Listagem 5**. Anulando uma referência para tornar o objeto elegível ao gc

Outra maneira é fazer uma variável de referência a um objeto A, “apontar” para outro objeto B, como na **Listagem 6**.

public class TesteColetor {

public static void main(String[] args) {

StringBuffer sb1 = new StringBuffer("abcd");

StringBuffer sb2 = new StringBuffer("xwyz");

// Os StringBuffer estão sendo referenciados

// e não são elegíveis para coleta de lixo

System.out.println(sb1);

sb1 = sb2;

// Agora o StringBuffer "abcd" deixa de ser referenciado

// por sb1 e torna-se candidato à coleta de lixo

// O StringBuffer "xwyz" por outro lado, está sendo referenciado

// por duas variáveis de referência

}

**Listagem 6**. Tornando um objeto elegível ao gc por meio da reatribuição de uma referência

Objetos que são instanciados dentro de métodos também precisam ser avaliados para verificar se são elegíveis, pois como sabemos, as variáveis locais declaradas em um método – sejam primitivas ou de referência – existem apenas durante a sua execução.

Dessa maneira, objetos criados e referenciados por variáveis locais, tornam-se elegíveis para a coleta de lixo após o retorno da execução do método. A exceção ocorre quando um objeto é retornado pelo método e atribuído a uma variável de referência no método que chamou, conforme o exemplo da **Listagem 7**.

import java.util.Date;

public class TesteColetor {

public static void main(String[] args) {

Date d = getDate();

System.out.println("d = " + d);

}

public static Date getDate() {

Date d1 = new Date();

StringBuffer now = new StringBuffer(d1.toString());

System.out.println(now);

return d1;

}

}

**Listagem 7.** Exemplo de objeto retornado por um método que não se torna elegível para o gc

No código da **Listagem 7**, o método getDate() cria dois objetos – um Date e um StringBuffer. Visto que getDate() retorna o objeto Date, o qual é atribuído a d no método main(), então tal objeto não se torna candidato ao Coletor de Lixo.

Entretanto, o StringBuffer torna-se elegível, pois a referência now deixa de existir fora de getDate().

No livro Sun Certified Programmer for Java 6 – Study Guide, Kathy Sierra e Bert Bates referem-se ao conceito de ilha de objetos ou ilha de isolação. Este é um caso onde objetos são elegíveis mesmo que ainda tenham referências válidas.

Um exemplo citado pelos autores é quando uma classe tem uma variável de instância que é referência para outra instância da mesma classe, tal como na classe Ilha, mostrada na **Listagem 8**.

public class Ilha {

Ilha i;

public static void main(String[] args) {

Ilha i1 = new Ilha();

Ilha i2 = new Ilha();

Ilha i3 = new Ilha();

i1.i = i2;

i2.i = i3;

i3.i = i1;

i1 = null;

i2 = null;

i3 = null;

}

}

**Listagem 8**. Exemplo de ilha de objetos

Note no exemplo da **Listagem 8** que os objetos criados passam a referir-se mutuamente e depois suas referências são anuladas. Assim, mesmo que os objetos possuam referências válidas – as variáveis de instância –, eles não podem mais ser alcançados por uma thread em execução, e tornam-se portanto elegíveis para a coleta de lixo. Felizmente o Garbage Collector é capaz de descobrir essas ilhas de objetos e então removê-las.

### Interagindo com o Garbage Collector

Não existe uma maneira explícita de liberar a memória ocupada por objetos não mais referenciados, porém pode-se invocar diretamente o Garbage Collector para procurar objetos inalcançáveis.

A classe Runtime, com alguns métodos da classe System, permite que o Coletor de Lixo seja chamado, que finalizadores pendentes sejam executados ou que se consulte o estado atual da memória.

Mais adiante, dedicaremos uma seção ao método finalize() – finalizador – que é executado pelo Coletor de Lixo depois que ele determina que um objeto não é mais alcançável, tornando-se assim elegível à coleta de lixo.

Os métodos que Java oferece para interagir com o Coletor de Lixo são membros de Runtime, a qual é uma classe especial que tem uma única instância para cada programa. Para obter a instância de Runtime, pode-se usar o método Runtime.getRuntime().

Após esta instância ter sido definida, pode-se invocar o Garbage Collector utilizando o método gc(). Ao executar esse método, a coleta de lixo deveria ser efetivada e o programa teria mais memória livre disponível.

No entanto, não existe garantia de que o método fará realmente isso. Primeiro, a especificação de Java permite que ele não faça nada. Segundo, outra thread – linha de execução – pode ter requisitado muita memória imediatamente após a chamada a gc().

Dessa maneira, mesmo que gc() tenha executado, o efeito de liberar memória não será obtido, pois a requisição da outra thread mascarará a liberação de espaço feita pelo Garbage Collector. Um terceiro motivo é que pode não haver objetos a serem coletados.

Com o objetivo de tentar ver o efeito do Coletor de Lixo, tente executar o programa da **Listagem 9** – adaptado do livro Sun Certified Programmer for Java 6 – Study Guide.

Neste programa podemos visualizar a memória disponível para a JVM, a memória livre antes da criação e depois da criação de 100.000 objetos do tipo Date. Em seguida o método gc() é chamado e novamente fazemos a consulta à memória livre.

import java.util.Date;

public class VerificaGC {

public static void main(String[] args) {

Runtime rt = Runtime.getRuntime();

System.out.println("Memória total da JVM: " + rt.totalMemory());

System.out.println("Memória antes da criação dos objetos: " + rt.freeMemory());

Date d = null;

for (int i = 0; i < 100000; i++) {

d = new Date();

d = null;

}

System.out.println("Memória depois da criação dos objetos: " + rt.freeMemory());

rt.gc();

System.out.println("Memória depois executar o gc: " + rt.freeMemory());

}

}

**Listagem 9**. Programa para interagir com o Garbage Collector

Após a execução do programa, ele retornou as seguintes informações no console:

Memória total da JVM: 61210624

Memória antes da criação dos objetos: 60570816

Memória depois da criação dosobjetos: 58208432

Memória depois de executar o gc: 60630904

Como podemos constatar pelos dados retornados, o Coletor de Lixo realmente deletou os objetos, visto que a memória livre aumentou após a execução do método gc().

Note, porém, que o comportamento de gc() pode ser diferente em outras máquinas virtuais, sem garantia de que os objetos elegíveis serão realmente removidos da memória.

Para finalizar esta seção, destacamos que o método gc() também pode ser chamado na classe System, a qual possui alguns métodos estáticos que criam uma instância de Runtime.

Assim, pode-se invocar o Coletor de Lixo simplesmente chamando System.gc().

### O método finalize()

Este método foi idealizado para ser chamado pelo Garbage Collector em um objeto que é elegível para a coleta de lixo. Ele é declarado originalmente na classe Object e, portanto, herdado por todas as classes Java. Por esse motivo, essas classes devem sobrescrever este método porque, semelhante aos métodos toString(), equals() e hashCode(), não existe uma implementação de finalize() em Object. Nessas implementações de finalize() – feitas nas subclasses – devem ser liberados quaisquer recursos utilizados pelo objeto que não sejam memória – visto que a memória é liberada pelo próprio Coletor de Lixo.

O contrato – recomendações da linguagem Java – de finalize() informa que ele só deve ser executado se e quando a JVM determinar que não existe mais qualquer meio pelo qual o objeto seja referenciado por qualquer thread.

A exceção ocorre por conta de alguma ação de um finalize() de outro objeto que esteja pronto para ser coletado. Isto se explica porque este método pode realizar qualquer ação, inclusive tornar o objeto disponível novamente para outras threads.

Dessa forma, se um objeto A for referenciado por um objeto B, o qual está elegível para o GC, então A também é elegível. Entretanto, se no método finalize() de B, este tornar-se disponível novamente, A também deixará de ser elegível.

No entanto, como já dissemos, seu objetivo principal é liberar recursos antes que o objeto seja irreversivelmente descartado.

Por exemplo, o método finalize() de um objeto que implementa uma conexão de Entrada/Saída (I/O), pode explicitamente encerrar a conexão antes da coleta definitiva.

Um problema que devemos nos preocupar em relação à implementação de finalize() é que, como estudamos até agora, não podemos contar que o objeto será removido pelo Coletor de Lixo.

Portanto, também não é assegurado que a ação implementada no código de finalize() seja efetivada. Por esses motivos concluímos que não se deve escrever código em finalize() que seja fundamental para a aplicação, visto que não se pode ter certeza que ele será executado.

Na verdade, recomenda-se que, em geral, o método finalize() não seja sobrescrito.

Finalmente, destacamos que finalize(), para um dado objeto, é executado uma única vez pelo Garbage Collector. Suponha, por exemplo, que para um objeto **A** – elegível para coleta de lixo – esse método seja implementado de maneira que a referência ao objeto A seja passada a outro objeto B, tornando assim A novamente vivo.

Se algum tempo depois este mesmo objeto A tornar-se novamente elegível, o Garbage Collector não executará mais o método finalize() e A finalmente será coletado.

A ausência de um gerenciamento adequado de memória pode causar uma exceção do tipo OutOfMemoryException. Esse é um dos principais motivos pelos quais o programador precisa conhecer o funcionamento do Coletor de Lixo, pois mesmo que o desenvolvedor não precise se preocupar em liberar explicitamente a memória ocupada pelos objetos candidatos à coleta, ele ainda necessita tratar de tornar elegíveis os objetos que não forem mais úteis à aplicação.

Por fim, fazemos uma ressalva de que este texto é introdutório. Por isso, recomendamos àqueles que pretendem avançar mais neste tema que estude o documento “Memory Management in the Java HotSpot Virtual Machine”, relacionado nos links no final da matéria.

Nele são descritos o funcionamento dos quatro tipos de Garbage Collection disponíveis na JVM e como o desenvolvedor pode escolher aquele que melhor se enquadra nas necessidades da sua aplicação.