Tópicos selecionados de programação em

Java

Gerência de Memória em Java

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

argonavis

Helder da Rocha

Setembro 2005

Assuntos abordados

1. Alocação e liberação de memória
   – Ciclo de vida de um objeto
   – Criação de objetos
   – Finalização de objetos
   – Estratégias de controle da coleta de lixo
2. Memory leaks
   – Detecção de memory leaks
3. Objetos de referência
   – Soft, weak e phantom references
   – Finalização com objetos de referência
   – WeakHashMap

1. Alocação e liberação de memória

- A criação de um objeto geralmente envolve
  - Alocação de memória no heap para conter o objeto
  - Atribuição do ponteiro (endereço no heap onde o espaço para o objeto foi alocado) a uma variável de pilha (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 Class
- Apenas objetos String podem ser criados implicitamente
  - Através da definição de um literal ou carga de uma classe que possui literais do tipo String
  - Através da concatenação de literais do tipo String
- Objetos são destruídos automaticamente pela JVM

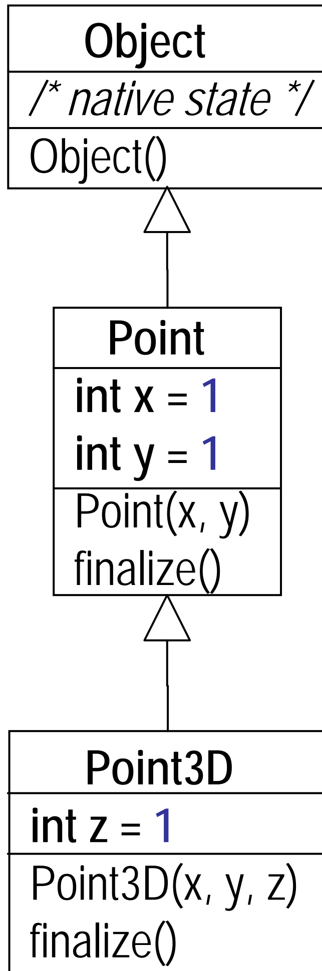
Criação de objetos

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

Processo de construção

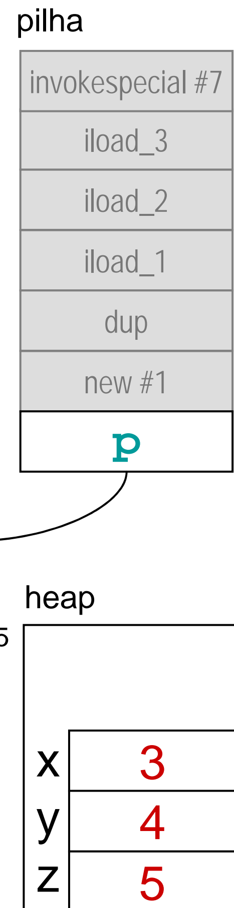
- A primeira instrução do construtor pode ser
  - uma chamada implícita ou explícita a super()
  - uma chamada explícita a this(), que passará o controle para um outro construtor e em algum ponto chamará super()
- O controle sobe a hierarquia através dos construtores chamados pela instrução super()
- Chegando na classe Object realiza os seguintes passos
  1. Inicializa variáveis de instância que têm inicializadores explícitos
  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

Passo-a-passo: construção




```
Point p = new Point3D(3, 4, 5);
```

1. Aloca espaço na memória
   Suficiente para Object + Point + Point3D
2. Empilha parâmetros:
   push 3, push 4, push 5
3. Inicializa variáveis default
   Point.**x** = **0**, Point.**y** = **0**, Point3D.**z** = **0**
4. Chama construtor via super()
   Point3D() → Point() → Object()
5. Executa corpo de Object()
6. Inicializa variáveis de Point
   Point.**x** = **1**, Point.**y** = **1**
7. Executa corpo de Point()
   Point.**x** = **3**, Point.**y** = **4**
8. Inicializa variáveis de Point3D
   Point.z = **1**
9. Executa corpo de Point3D()
   Point.z = **5**
10. Retorna referência do objeto para `p`



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 formas limitadas no processo de destruição através de
  - 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 de lixo
  - Uso de referências fracas

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 ordenação: finalização pode acontecer em paralelo

Finalização é importante?

- Depende.
- Há objetos que não precisam de finalizadores
  - Aqueles cujos recursos são automaticamente liberados pelo coletor de lixo: qualquer tipo de alocação na memória, referências (inclusive circulares) de qualquer tipo, etc.
- Há objetos que precisam de finalizadores
  - Fechar arquivos abertos e sockets (o sistema operacional limita a quantidade de recursos que são abertos; não finalizar depois do uso pode impedir a criação de novos arquivos ou sockets)
  - Fechar streams (fluxos de gravação podem ficar incompletos se buffer não for esvaziado)
  - Fechar threads (threads costumam rodar em loops; finalizadores ligam um flag para terminar o loop ou interrompem o thread e evitar que o programa nunca termine)

Finalizadores automáticos

- Em Java, cada objeto *pode* ter um finalizador chamado automaticamente antes de um objeto ser destruído
- Para implementar, é preciso sobrepor a assinatura:

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

- finalize() é chamado automaticamente e apenas uma vez somente quando o objeto não for mais alcançável através de referências comuns (raiz)
- O método finalize() não será chamado se
  - Não sobrepuser explicitamente o método original (uso é opcional!)
  - Não houver necessidade de liberar memória (GC não executar), mesmo que todas as referências do objeto tenham sido perdidas
- A chamada dos finalizadores automáticos não é garantida
  - Depende de vários fatores e da implementação do GC

Ciclo de vida de um objeto

p = new Ponto()

Memória **alocada** no heap

operações recursivas

Variáveis inicializadas: default

Construtor chamado recursivamente

Variáveis inicializadas explicitamente

Corpo do construtor executado

reachable

Retorna referência

p = null

Referência perdida

finalizer-reachable

espera algoritmo GC

Garbage Collector elege objeto

espera fila de finalização

operações recursivas

atributos finalizados/liberados

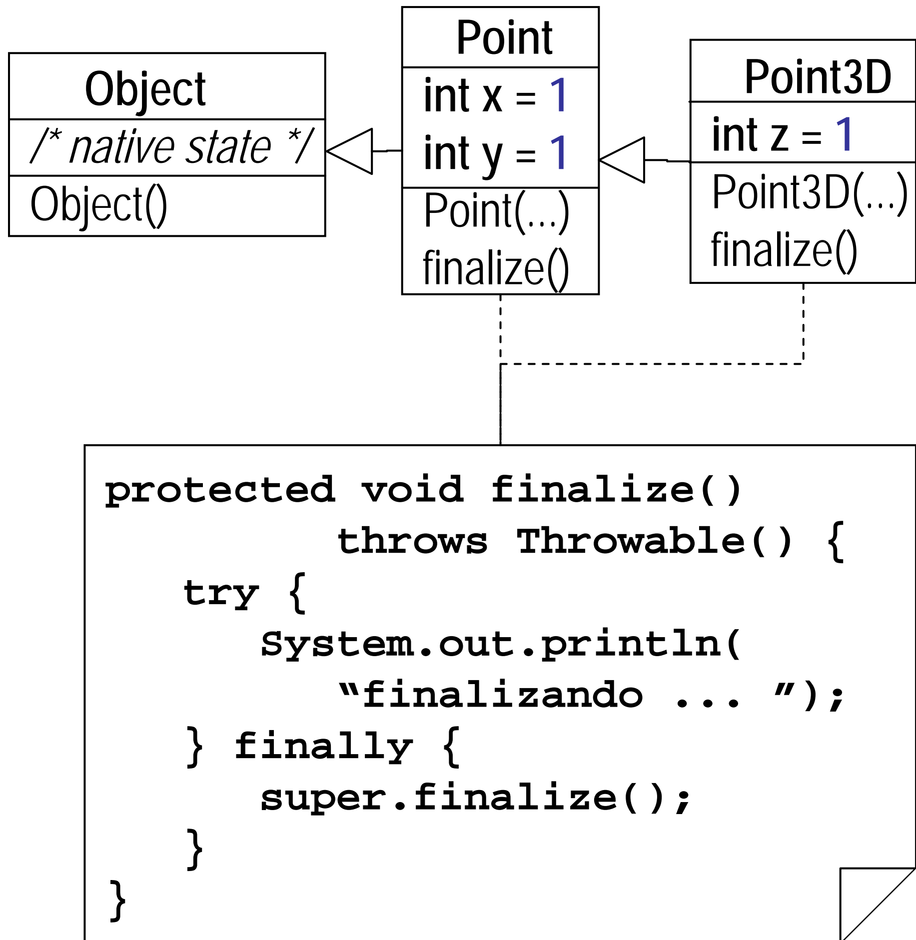
finalizador chamado recursivamente

espera fila de liberação

unreachable

Memória do heap **liberada**

Passo-a-passo: destruição




```
p = null; // p é Point3D
```

1. Espera coleta de lixo
   Eventualmente GC executa
2. Objeto em fila de finalização
   Eventualmente GC tira objeto da fila
3. Executa **finalize()** de Point3D
   Imprime "finalizando ..."
   Chama super.finalize()
4. Executa **finalize()** de Point
   Imprime "finalizando ..."
   Chama super.finalize()
5. Executa **finalize()** de Object
   Termina finalize() de Point3D
6. Objeto finalizado espera liberação
   Eventualmente liberação ocorre
7. Objeto destruído

Objetos alcançáveis

- Objetos que não podem ser destruídos pelo GC
  - 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 tread 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)

Alcançabilidade e finalização

[JLS 12.6]

- Há 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 (não pode ser alcançado normalmente). Pode ser ressuscitado se, após a morte, seu finalizador passar sua referência this para um objeto alcançável.
- Há três estados em que pode estar a finalização
  - **não finalizado** (unfinalized) – nunca teve o finalizador chamado*
  - **finalizado** (finalized) – já teve o finalizador chamado*
  - **finalizável** (finalizable) – pode ter o finalizador chamado automaticamente a qualquer momento (não é mais alcançável)

** automaticamente*

Transição de estados*

Reachable
(utilizável)

Unreachable
(não-utilizável)

1. `MyObject obj = new MyObject();`

Unfinalized

2.
`obj = null;`

`finalize()` não foi definido

Unfinalized

`finalize()` foi definido

3.
`zumbi = this;`

Finalizer-Reachable
& Finalizable

3.
`finalize();`

***Ressurreição**:*
`this` passado para referência alcançável `zumbi` na execução de `finalize()`

Finalização:
Execução normal de `finalize()` sem deixar vazar referências

4.
`zumbi = null;`

Finalized

Morte definitiva:
`finalize()` só pode ser chamado uma vez

Finalized

* *Não leva em conta eventual presença de referências fracas (Soft, Weak, Phantom)*

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
  - Objeto pode ser alcançado por referências externas: volta à vida
  - Se morrer outra vez, vai direto ao estado unreachable: não faz finalize()

```
HauntedHouse h = new HauntedHouse();
new Guest(h);  // cria objeto e mantém referencia em h
h.killGuest(); // mata objeto e finaliza, mas ele ressuscita!
h.killGuest(); // mata objeto de novo... desta vez ele vai
```

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

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

Não ressuscite objetos

- Acordar os mortos geralmente não é uma boa idéia
- Os exemplos mostrados sobre ressurreição de objetos têm finalidade didática (e lúdica ☺)
  - Importante para entender o processo de finalização
- A ressurreição de objetos raramente tem aplicações práticas e geralmente é uma prática a ser evitada
  - Geralmente os problemas que sugerem ressurreição de objetos podem ser implementadas com novos objetos e cópia de seus estados (clonagem, por exemplo)
  - 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 (irão direto para a liberação)
  - Use apenas se for necessário (lembre-se: não é confiável)
- Construtores, automaticamente chamam a sua superclasse; finalizadores não
  - A correta implementação deve sempre chamar super.finalize(), de preferência em um bloco finally
  - Não capture exceções (deixe que elas aconteçam)

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

Técnica padrão para implementar finalize()

Finalização: como funciona

- Programa de demonstração
  1. Para **ocupar memória** (e forçar o GC)
  2. Executado com **pouca memória** (garantir GC)
  3. Objetos usam **referências fracas** (para que sejam liberados com freqüência)
  4. Contagem de chamadas ao construtor, ao método finalize() e bloco finally

- 1000 objetos são criados: *Quanto deve ser a contagem em cada caso?*

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

Finalização não é garantida!

```
C:\>java -Xmx1M -Xms1M -verbosegc -cp build/classes memorylab.Main
[Criados agora:      447; total criados:      447]
[Finalizados agora: 191; total finalizados:  191]
    [Full GC 1718K->1203K(1984K), 0.0137311 secs]
[Criados agora:      144; total criados:      591]
[Finalizados agora: 146; total finalizados:  337]
    [Full GC 1712K->1636K(1984K), 0.0136167 secs]
[Criados agora:      125; total criados:      716]
[Finalizados agora: 125; total finalizados:  462]
    [Full GC 1979K->1459K(1984K), 0.0134883 secs]
[Criados agora:       84; total criados:      800]
[Finalizados agora: 125; total finalizados:  587]
    [Full GC 1979K->1473K(1984K), 0.0137952 secs]
[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.
```

Execução 1
Heap de 1Mb

```
C:\>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.
```

Execução 2
Heap de 8Mb

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
  - 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 GC (para coleta completa)
  - Não haver necessidade de reusar sua memória
  - Outras razões dependentes de implementação/plataforma

Não é preciso usar finalize()

- Mas finalizadores podem ser importantes!
  – Finalização de arquivos, soquetes, etc. não devem depender da finalização automática do sistema via finalize()
- Problemas dos finalizadores automáticos
  – Não há garantia que serão executados em um tempo razoável (nem que serão executados)
  – A fina de espera pode demorar e consumir memória
  – Execução depende da implementação da JVM
  – Thread de baixa prioridade (GC) pode nunca executar finalizador
  – Exceções ocorridas durante a finalização são ignoradas
- System.gc() aumenta as chances de execução de um finalizador mas não a garante
  – System.gc() também depende de implementação!

O que usar no lugar de finalize?

- Métodos de finalização explícita!
  - close(), destroy(), dispose(), flush() e similares
  - Devem ser chamados pelo cliente (em um bloco try-finally para garantir sua execução)
  - Mudança de design: a responsabilidade pela finalização passa do autor da API para o cliente
  - Esses métodos podem também ser chamados por finalize() como rede de segurança (caso o cliente esqueça de finalizar)
- Há vários finalizadores explícitos na API Java
  - File.close(), Socket.close(), Window.dispose(), Statement.close()
  - A maioria usa finalize() como rede de segurança (para liberar recursos de qualquer maneira, caso o usuário cliente não tenha chamado o método de finalização)
  - Não chamar esses métodos é depender da finalização

Exemplo de finalização

```
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();
  }
...
```

Cliente chama close() para não depender de finalize()

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

```
...
   public void close() {
     done = true;
     if(!queueManager.isInterrupted())
       queueManager.interrupt();
   }
   protected void finalize()
          throws Throwable {
     try { close(); }
     finally { super.finalize(); }
   }
}
```

Rede de segurança:se cliente esqueçer o close(), finalize() é melhor que nada.

Finalizer Guardian

- Havendo necessidade de implementar finalize(), é preciso implementá-lo corretamente
- Proteção contra uso incorreto da API
  - O que fazer se o cliente que sobrepõe a classe não implementar corretamente finalize() (esquecendo de chamar super.finalize())?
- Para aumentar a rede de segurança, 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)

Padrão Finalizer Guardian

- Protege contra implementação incorreta de finalize() por parte das subclasses
  - Classe interna finaliza objeto externo de sua liberação

```
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
    }
}
```

Finalizer Guardian

Quando **guardian** *for finalizado, automaticamente finalizará o objeto externo*

Fonte: Joshua Bloch, Effective Java, Item 6

Finalização de threads

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

```
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 Finalizador();  // implementação
   Thread.currentThread()
       .setUncaughtExceptionHandler(handler);
   // código que pode causar exceção
}
```

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

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

- Torne o objeto **inalcançável**, 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 é garantido para blocos)
- É importante garantir que não haja outras referências para o objeto
  - É comum "esquecer" referências ativas em listas de event handlers e coleções (casos mais comuns de **memory leak**)
- Chamar o **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
- Chamar o método **gc()** sugere à JVM que ela faça um esforço para reciclar objetos não utilizados, para liberar a memória que ocupam para que possa ser reusada
  - Execução pode não acontecer imediatamente
  - Execução pode nunca acontecer (programa pode terminar antes)
- Chamar System.gc() não garante a liberação de memória de todos os objetos inalcançáveis
  - Há algoritmos de GC que, para aumentar a eficiência, podem deixar de recolher objetos (serão recolhidos na próxima coleta)
- Chamar System.gc() repetidamente é muito ineficiente e inútil se não houver objetos disponíveis à remoção
  - Ideal é usar estratégias que não chamem e não usem System.gc(), exceto para depuração

System.runFinalization()

- Executa a finalização de métodos de quaisquer objetos cuja finalização ainda não foi feita
  - Só acontece se objeto já for candidato à liberação através do coletor de lixo (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 destruição, mas cujos métodos de finalização ainda não foram executados
  - Este método é ainda menos previsível que System.gc()
- System.runFinalizersOnExit()
  - Único que garante a execução dos finalizadores, mas é inseguro (e foi deprecado)

Exemplo

- A aplicação abaixo força o GC como meio de garantir a finalização de um objeto
  – O bloco finalize() do objeto imprime o nome passado no construtor (para que possamos saber qual objeto finalizou)
  – Apenas a primeira finalização ocorreu* (comportamento é dependente da plataforma e implementação da JVM)

```
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");
```

Trecho de código

Execução

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

** na minha máquina!*

E então, como controlar o Garbage Collector?

- System.gc()
  - Chama o garbage collector (assim que possível), mas só elimina objetos que já estiverem inalcançáveis
  - É ineficiente: pára o sistema para remover os objetos
  - Comportamento depende da JVM: use raramente (depuração)
- Runtime.getRuntime().gc()
  - Mesmo que System.gc()
- ref = null
  - Declarar a última referência para um objeto como null, vai torná-lo elegível à coleta de lixo (inalcançável ou finalizer-reachable)
  - É mais rápido que reutilizar a referência, ou fechar o bloco (método) onde o objeto foi declarado (mas JVMs podem otimizar)
- Referências fracas

2. 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
  - Não ocorre em aplicações 100% Java (se acontecer é bug na JVM, o que não é responsabilidade do programador)
- Memory leaks em Java são considerados em um sentido mais abrangente: um objeto que não é coletado depois que não é mais necessário, ou não está mais ativo
  - São causados por objetos não mais usados que não são liberados porque ainda são alcançáveis
  - Uma interface que impede ou que não exige que o cliente libere uma referência depois do uso tem potencial para memory leak
  - O critério para definir um memory leak nem sempre é claro: pode ser subjetivo, depender de contexto ou de algum evento (memória sendo consumida rapidamente, OutOfMemoryError)

Considere a seguinte classe...

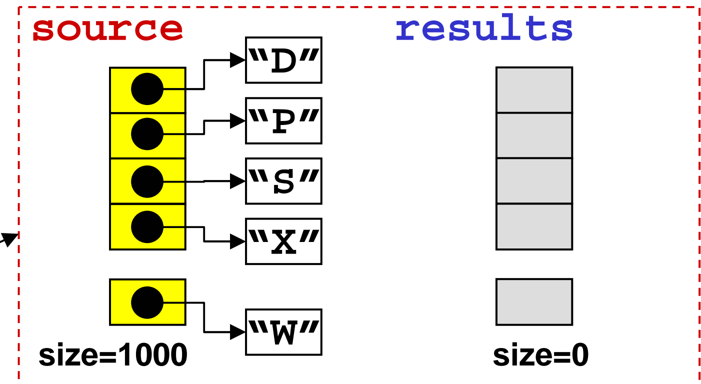
```
public class BadStack { // não é thread-safe!
    private Object[] elements;
    private int size = 0;
    public BadStack(int initialCapacity) {
        this.elements = new Object[initialCapacity];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    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);
        }
    }
}
```

Única interface garante consistência (sem threads) da leitura e gravação

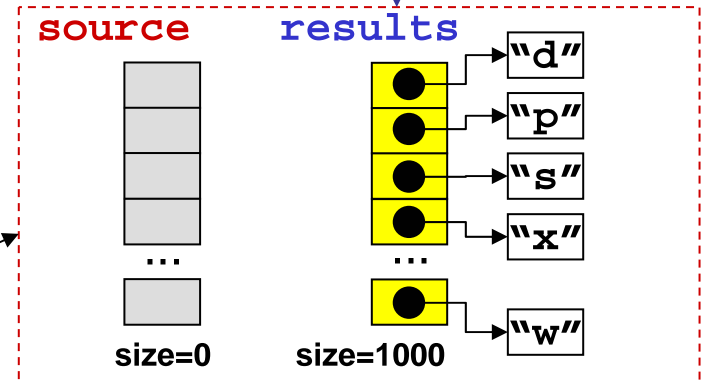
Fonte: Joshua Bloch, Effective Java, Item 5

Transferindo dados

```
BadStack results = new BadStack(1000);
BadStack source = new BadStack(1000);

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

System.out.println("BEFORE PROCESSING");
// imprime source.size(), results.size()
// conta quantas instâncias existem em cada pilha

try {
    while(true) {
       char c = Character.toLowerCase(
           (Character)source.pop());
       results.push(new Character(c));
    }
} catch (EmptyStackException e) {}

System.out.println("AFTER PROCESSING");
// Imprime mesmas informações
```



Comportamento funcional

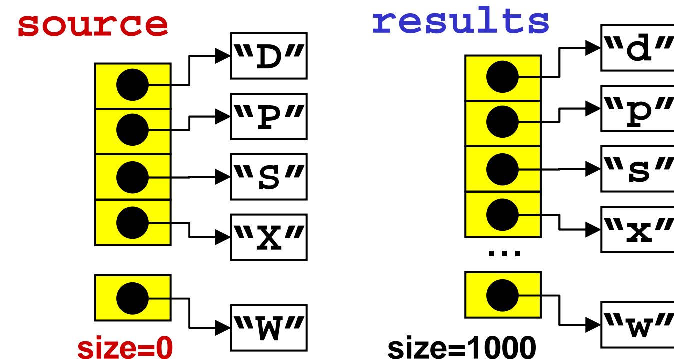


Há algum problema com este programa?

Esvaziamento não ocorre!

- 1000 objetos foram transferidos de uma pilha para outra
  - No modelo funcional de memória do programa, uma pilha foi esvaziada e a outra foi preenchida (não é possível acessar objetos em source)
  - Mas no que se refere ao coletor de lixo, os 1000 objetos da pilha que foi esvaziada (`source`) continuam accessíveis

```
BEFORE PROCESSING
source.size(): 1000
result.size(): 0
Instances in source: 1000
Instances in results: 0

AFTER PROCESSING
source.size(): 0
result.size(): 1000
Instances in source: 1000
Instances in results: 1000
```



Terminamos de usar o objeto, no entanto, ainda há 1000 instâncias que podem ser alcançadas! Elas não terão sua memória liberada pelo GC!

*Mas o programa **está correto** (foi necessário quebrar o encapsulamento para obter esses dados)*

Consertando o vazamento

- O problema é que o programa mantém referências obsoletas para objetos
  - O vazamento poderia ser ainda maior, se os objetos da pilha tivessem referências para outros objetos, e assim por diante
  - Poderia ocorrer OutOfMemoryError
- 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 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
- Use ferramentas de monitoração
  - O jconsole traça gráficos do heap e de suas regiões
  - O consumo médio de memória deve manter-se constante através do tempo

Programa com memory leaks

Alocação média de memória aumenta longo do tempo após várias coletas

J2SE 5.0 Monitoring & Management Console

Connection Window

1328@localhost

Summary | Memory | Threads | Classes | MBeans | VM

Chart: Heap Memory Usage Time Range: All Perform GC

3,0 Mb
2,5 Mb
2,0 Mb
1,5 Mb
1,0 Mb
0,6 Mb

23:46 23:47 23:48 23:49 23:50 23:51 23:52 23:53 23:54 23:55 23:56 23:57 23:58 23:59 00:00 00:01 00:02 00:03
2005-09-15

Used 1.998.104

Details

Time: 2005-09-15 00:03:57
Used: 1.951 kbytes
Committed: 1.984 kbytes
Max: 1.984 kbytes
GC time: 4,466 seconds seconds on Copy (283 collections)
1,044 seconds seconds on MarkSweepCompact (22 collections)

100% 75% 50% 25% 0%

Heap Non-Heap

jconsole faz parte do SDK do Java 1.5

Programa sem memory leaks

Alocação média de memória mantém-se constante ao longo do tempo após várias coletas

J2SE 5.0 Monitoring & Management Console: 3312@localhost

Connection

Summary | Memory | Threads | Classes | MBeans | VM

Chart: Heap Memory Usage Time Range: 10 min Perform GC

2,0 Mb
1,5 Mb
1,0 Mb
0,7 Mb

13:54 13:55 13:56 13:57 13:58 13:59 14:00 14:01 14:02 14:03

Used
1.145.576

Details

Time: 2005-08-17 14:03:52
Used: 1.118 kbytes
Committed: 1.984 kbytes
Max: 1.984 kbytes
GC time: 6 minutes seconds on Copy (17.080 collections)
2,435 seconds seconds on MarkSweepCompact (68 collections)

100 %
75 %
50 %
25 %
0 %

Heap Non-Heap

start | 2 Windows Comman... | J2SE 5.0 Monitoring ... | memoryleak.jpg - Win... | PT | 14:03

Como consertar vazamentos

- Não adianta chamar System.gc()
  - Tem impacto absurdamente negativo na performance,
  - Força a execução do Garbage Collector, que recolherá apenas objetos inalcançáveis (memory leaks são objetos alcançáveis)
- Eliminar todas as referências para o objeto
  - Procure-as usando ferramentas, se necessário
- Alternativas de eliminação de referências
  - Declarar a referência como null quando não for mais usada (não abuse: polui o código desnecessariamente)
  - Manter referências no menor escopo possível (melhor opção): o escopo mínimo deve ser o de método
  - Reutilizar a referência (melhor opção): a liberação poderá não ocorrer tão cedo quanto null em JVMs antigas
- Outra solução é usar referências fracas

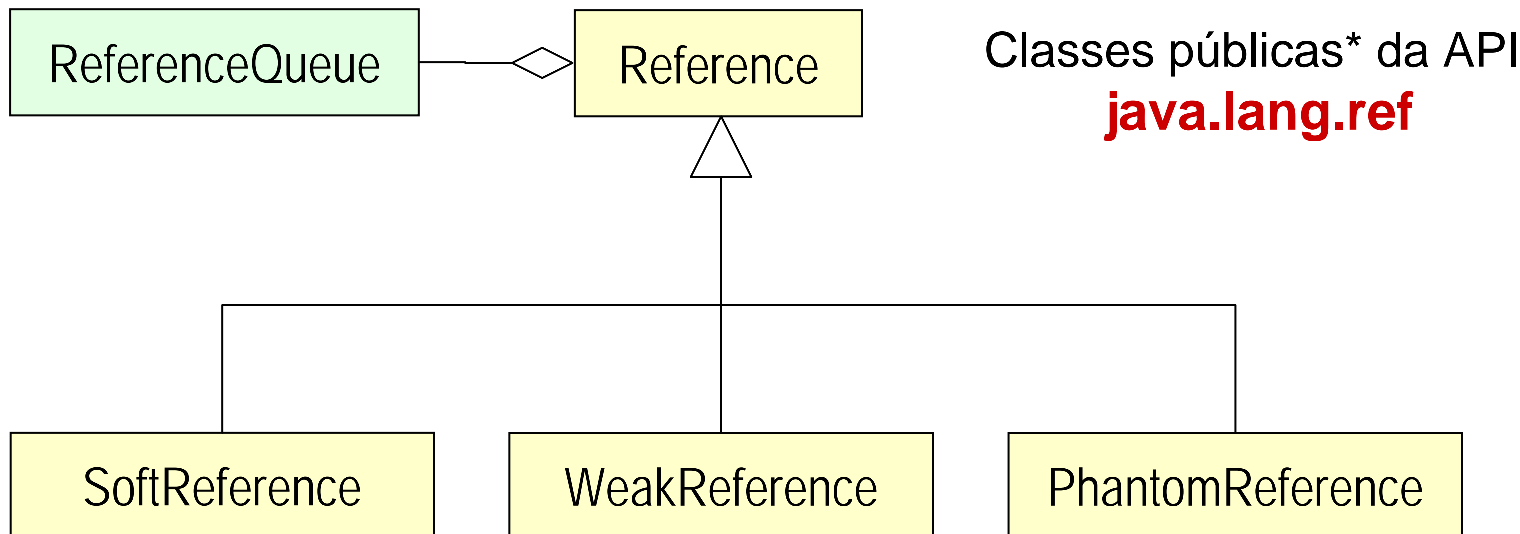
3. O que são referências fracas?

- Referências cuja ligação com o objeto ao qual se refere é **fraca**: pode ser perdida a qualquer momento
  - Permitem que um programa refira-se a um objeto sem impedir sua eventual coleta, caso seja necessário
  - O coletor de lixo considera os objetos que só são alcançáveis via referências fracas como objetos que podem ser removidos
- A API de reference objects (java.lang.ref) permite que um programa mantenha referências fracas para 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

Hierarquia dos objetos de referência

- Os objetos de referência são descendentes da classe abstrata java.lang.ref.Reference<T>

Todos os tipos são genéricos!



** Há mais duas subclasses package-private usadas internamente para realizar finalização*

Para que servem?

- ReferenceQueue
  - Usada com Weak ou SoftReference permite tratar eventos na mudança da alcançabilidade: realizar pré-finalização
  - Com PhantomReference guarda objetos finalizados para pós-finalização
- SoftReference
  - Para implementar caches sensíveis à memória (que são esvaziados apenas quando a memória está muito escassa)
- WeakReference
  - Implementar mapeamentos que permitam que chaves ou valores sejam removidos do heap (ex: listas de handlers para eventos)
  - Construir caches de serviços, que mantém referência para o serviço quando em uso (quando referência perder-se, objeto pode ser removido)
- PhantomReference
  - Para implementar ações de finalização de uma forma mais flexível que o mecanismo de finalização do Java.

Como criar e como usar

- Como criar (uso típico)
  - Passe a referência de um objeto (*referente*) como argumento na construção de um objeto de referência

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

  - Elimine todas as referências fortes do objeto

```
fraca = null;
```

- Uma vez criada, a referência fraca é imutável
  - Não pode apontar para outro objeto
  - Pode ser esvaziada (conter null) chamando o método **clear()**
- Como usar a referência (uso típico)
  - Chame o método **get()** para obter o referente

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

  - **get()** retorna **null** se objeto já foi coletado ou **clear()** foi chamado

Referências fracas e fortes

```
Referente fraca = new Referente();
SoftReference forte = new SoftReference(fraca);
fraca = null;
fraca = (Objeto)forte.get();
```

ligação fraca

forte

:SoftReference

:Referente

referent

referências fortes

referência fraca

fraca

Quando esta referência for anulada,
Referente será fracamente alcançável

API essencial: Reference

- Todos os Reference Objects possuem duas operações básicas (herdadas da classe Reference<T>)
  - `T get()` : retorna o objeto referente. Este método é sobreposto em todas as subclasses para prover o comportamento distinto
  - `void clear()` : elimina objeto referente (faz get() retornar null)
- Métodos usados pelo coletor de lixo para gerenciar fila de objetos de referência (classe ReferenceQueue<T>)
  - `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 fila ReferenceQueue à qual foi registrado.
- O coletor de lixo acrescenta um objeto na fila quando clear() é chamado

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)
  - Objeto que pode tornar-se inalcançável a qualquer momento
- 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

Precedência da alcançabilidade

Conjunto raiz
de referências

Heap

Objeto de referência

Objeto *fracamente* alcançável

Objeto inalcançável (lixo)

Objeto fortemente alcançável

Fonte: [Pawlan]

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): acessíveis através de uma **SoftReference**: objetos podem ser finalizados e coletados quando o GC decide é preciso liberar memória
  - **Weakly reachable** (fracamente alcançável): acessíveis através de uma **WeakReference**: objetos podem ser finalizados e coletados a qualquer momento
  - **Phantomly reachable** (alcançável após a finalização): acessíveis através de uma **PhantomReference**; objetos já finalizados que esperam autorização para liberação (não são mais utilizáveis)
  - **Unreachable** (inalcançável): objetos que não têm mais referência alguma para eles, e que serão coletados

Transição de estados com objetos de referência

SoftReference e WeakReference

- Estratégias similares: diferem apenas na forma do tratamento recebido pelo Garbage Collector

SoftReference	WeakReference
Mantém objetos ativos desde que haja memória suficiente, mesmo que não estejam em uso	Mantém objetos ativos enquanto estiverem em uso (alcançáveis, com uma referência forte)
O GC só terá que liberar todas os objetos que só tenham referências desse tipo antes de lançar um OutOfMemoryError	O coletor de lixo poderá liberar objetos que só tenham referências desse tipo a qualquer momento (próxima passada do GC)
O coletor de lixo primeiro removerá os objetos mais antigos	O coletor de lixo não toma quaisquer decisões antes de liberar a memória
Use quando existir a possibilidade do cliente voltar e tentar reaver o objeto depois de algum tempo	Use para objetos que têm vida curta (o cliente ou decide reaver o objeto logo ou não volta mais)

Política do GC para referências do tipo Soft e Weak

- WeakReference: não usam algoritmo para decidir ou não pela liberação de memória
  - Se GC rodar e houver WeakReferences, seus objetos referentes serão removidos
- SoftReferences são avaliadas pelo GC
  - Algoritmo só os remove se não tiver outra opção
  - Mais antigos são liberados primeiro
  - Pode-se ajustar comportamento do algoritmo via opções do JVM
- Opções de configuração: JVM da Sun (JRE 5.0)

  `-XX:SoftRefLRUPolicyMSPerMB=`<*ms por Mb livre do heap*>
  - Taxa *(milissegundos por Mb)* em que VM remove referências Soft
  - VM **`-client`** considera Mb relativo ao tamanho atual do heap.
  - VM **`-server`** considera Mb relativo ao heap máximo (**`-Xmx`**)
  - *Exemplo:* **`java -XX:SoftRefLRUPolicyMSPerMB=1000`** ...
    - Referências tipo Soft irão durar 1 segundo para cada Mb livre

Exemplo: pilha com WeakReference

- Manutenção das referências é responsabilidade do cliente
  - **Cuidado**: 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();
    }
    public int size() { return size; }
    private void ensureCapacity() { ... }
}
```

Pode retornar null se cliente já tiver perdido referências usadas no push()

Exemplo: pilha com SoftReference

- Objetos duram muito mais (ainda dependem de cliente e GC)
  - Mesmo que cliente perca as referências, 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() { ... }
}
```

Pode retornar null se GC precisar da memória e cliente tiver perdido as referências do push()

Exemplo: cache de dados

- SoftReferences são a escolha ideal para caches: manterão um objeto ativo o máximo de tempo possível.

```
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 { ... }
   ...
}
```

Se o objeto ainda estiver ativo, economiza-se leitura dos dados do arquivo

ReferenceQueue

- Uma fila de objetos de referência preenchida pelo GC
  - Recebe uma referência weak ou soft algum tempo depois que o referente tornar-se inalcançável; phantom depois de finalizado
  - Pode ser usada como mecanismo de notificação, e de pré- ou pós-finalização
- Sempre passada na criação do objeto

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

- Métodos: todos retornam Reference

  `remove()` e **`remove(long timeout)`**

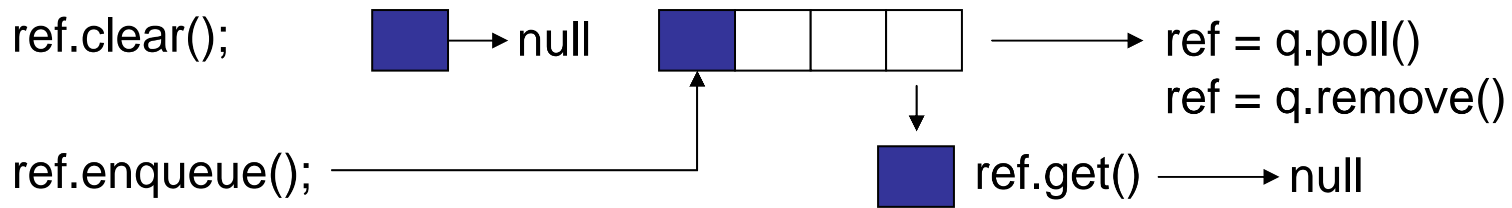
  Bloqueiam o *thread* enquanto não houver elementos para retirar; podem ser interrompidos (InterruptedException)

  `poll()`

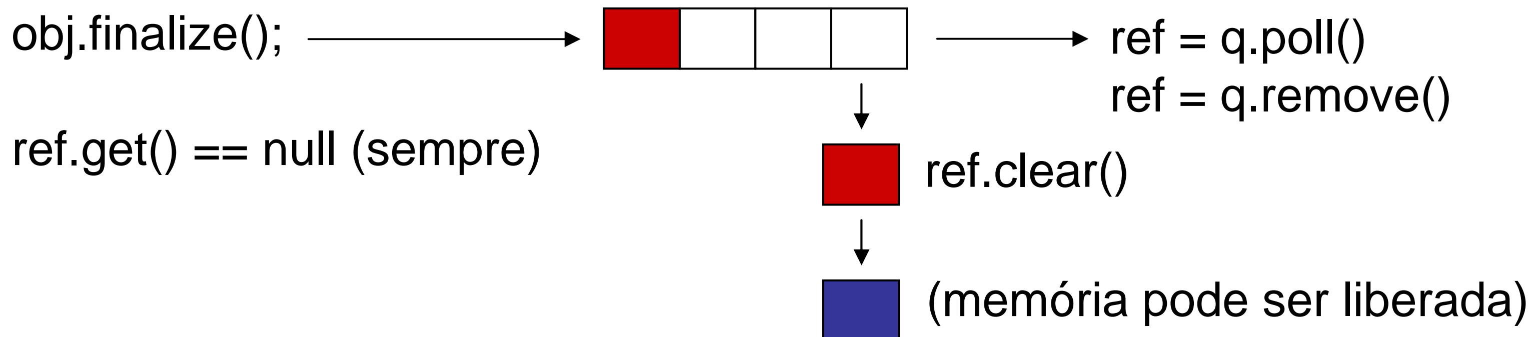
  Retorna null (obj. referência) enquanto não houver objetos na fila
- Métodos não servem para recuperar referente
  - get() em objeto de referência tirado da fila *sempre* retorna null

ReferenceQueue: funcionamento

- Com referências Weak e Soft
  - Chamar clear(), coloca objeto na fila (depois de algum tempo)



- Com referências Phantom
  - Objeto “nasce” na fila. Chamar clear(), tira objeto da fila



Uso de ReferenceQueue

- *Thread* abaixo remove entradas de um Map quando referências weak tornam-se inalcançáveis

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

Bloqueia até que apareça um Reference na fila

Se valor guardado for igual ao da referência que chegou na fila, o referente (ref.get()) já está inalcançável, então remova chave (e valor) do mapa

Finalização com referencias fracas

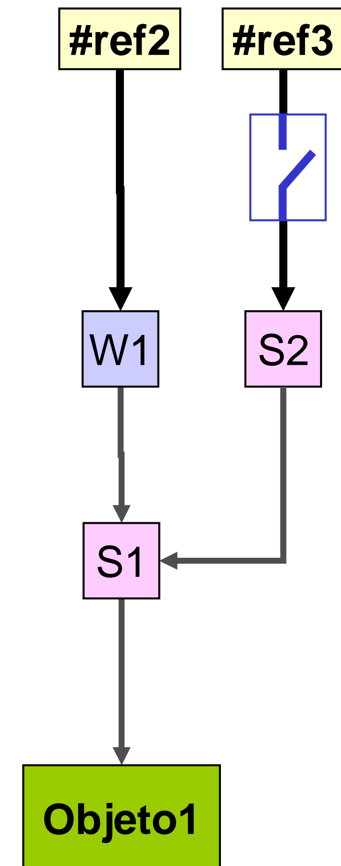
- Permite três comportamentos
  - Quando a memória estiver no limite (soft): pré-finalização
  - Quando o GC rodar (weak): pré-finalização
  - Depois que objeto estiver finalizado (phantom): pós-finalização
- Como implementar
  - Crie um *thread* que use poll() ou remove() para saber quando um objeto perdeu sua referência fraca

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

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

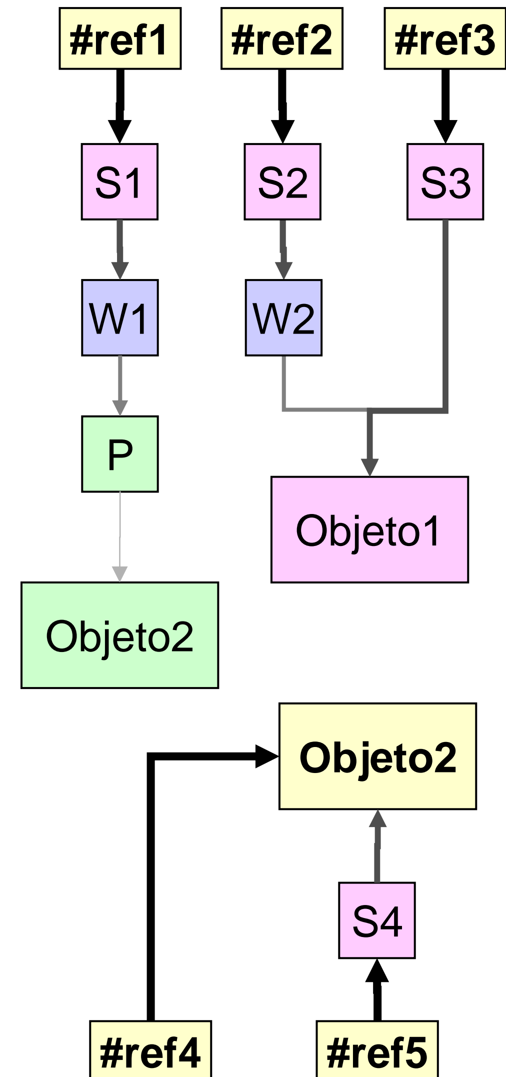
Controle do algoritmo de liberação

- Pode-se ter algum controle sobre a liberação de memória usando referências encadeadas
- Exemplo (veja figura)
  - **#ref2**: uma WeakReference W1 contém uma SoftReference S1
  - **#ref3**: uma SoftReference S2 referencia S1
- Enquanto existir a referência **#ref3**, o objeto será tratado como tendo uma SoftReference (só será removido se faltar memória)
- Em algum momento, se **#ref3** for perdida, o único caminho para **Objeto1** é através de uma WeakReference, portanto passará a ser tratado como tal (poderá ser removido a qualquer momento)



Referências encadeadas

- Pode haver diversos caminhos paralelos de referências encadeadas em série para um objeto
  - Dentre os caminhos **paralelos**, a alcançabilidade do objeto é determinada pela referência mais forte que houver para o objeto
  - Em uma **série** de referências interligadas, a referência mais fraca determina a alcançabilidade do objeto através daquele caminho
- O processamento (sempre pelo caminho mais forte) acontece na ordem abaixo
  1. Soft references
  2. Weak references
  3. Finalização de objetos
  4. Phantom references
  5. Liberação de memória
- Não há garantia de quando o processamento em cada etapa ocorrerá



Referências fantasma

- Objetos do tipo PhantomReference já foram finalizados (finalize() foi chamado) mas ainda não foram liberados
  – Estão mortos. Não podem mais ser usados nem ressuscitados!
  – Permitem realizar operações pós-morte associadas à objetos já finalizados (identificáveis através de suas referências fracas).
- ReferenceQueue é obrigatório
  – Fantasmas são colocados no seu ReferenceQueue logo que se tornam phantomly reachable (pouco depois de criados)
  – Pode-se pesquisar a fila, retirar os objetos de referência e através deles identificar os referentes (já mortos)
  – Chamar clear() em um PhantomReference, retira-o da fila
- É preciso retirar o objeto da fila chamando clear() ou sua memória nunca será liberada (memory leak!)

Finalização com PhantomReference

- Não há garantia que isto seja muito mais confiável que finalize():

```
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();
    close(); // finalization
  }
};
new Thread(finalizer).start();
```

Depois que o objeto referente estiver finalizado, ele irá aparecer na fila

Libera memória

Pós-finalização

- Neste exemplo, finalize() guarda arquivo serializado com objeto morto que é trazido de volta à vida (como cópia) na pós-finalização

```
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();
    }
  }
}
```

Pós-finalização não tem mais referência para objeto, mas é disparada pelo evento de finalização do mesmo objeto

```
Reference found = queue.remove();
if (found != null) { // uma Reference
try {
   ObjectInputStream openMummy =
     new ObjectInputStream(
       new FileInputStream("/tmp/mummy"));
   Guest ressurected =
       (Guest)openMummy.readObject(); ...
} catch (Exception e) {...}
```

java.util.WeakHashMap

- Um Map onde o par chave/valor é uma WeakReference
  - Depois que o objeto referenciado pela chave fraca torna-se fracamente alcançável, o GC 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
  - Use para caches, listas de event handlers, etc.
  - Evita memory leaks mais comuns
- Há risco de perda de dados!
  - Usa WeakReferences (GC pode liberar a qualquer momento)
  - Considere construir um SoftHashMap (não existe na API) se volatilidade do WeakHashMap for um problema

WeakHashMap

<K,V>

«interface»
Map

entrySet()

*

«interface»
Map.Entry

java.util.ref

WeakReference

AbstractMap

private
implementation

WeakHashMap.Entry

getKey()

1

K

getValue()

1

V

WeakHashMap

«weak»

Aplicação usando HashMap com memory leak

- O exemplo (didático) abaixo não pára de acrescentar novos objetos em um HashMap
  - Eventualmente causará 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);
    }
  }
}
```

Corrigindo o memory leak com WeakHashMap

- Simplemente mudando para WeakHashMap pode-se garantir que a memória não acabará por excesso de elementos no HashMap

```
public class MemoryLeak {
  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);
    }
  }
}
```

Execute as aplicações usando o jconsole para monitorar

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 funcionamento do GC usando
  - finalizadores automáticos: não confiáveis
  - chamadas explícitas ao GC: não garantidas
  - objetos de referência
- Objetos de referência flexibilizam a ligação forte de um objeto com suas referências e oferecem o maior controle sobre o comportamento do GC. Há três tipos
  - SoftReferences: adia a coleta o máximo possível
  - WeakReferences: coleta objeto no próximo GC
  - PhantomReferences: notifica finalização de objeto

Fontes de referência

[Pawlan] 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/

[SDK] Documentação do J2SDK 5.0

[BJ] Bruce Tate, Bitter Java, Manning, 2002

- Discussão interessante sobre memory leaks

[EJ] Joshua Bloch, Effective Java, Addison-Wesley, 2001

- Padrão finalizer guardian, discussão sobre finalize e memory leaks

[Friesen] Trash Talk part 2: Reference Objects. JavaWorld, Jan 2002.

- http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-java101.html

© 2005, Helder da Rocha
www.argonavis.com.br