Garbage Collection

ISEL/LEIC – Inverno 2008/2009

2.º ano, 2.º Semestre

Nuno Leite

AVE: Ambientes Virtuais de Execução – Semestre de Inverno 2008/09

http://www.deetc.isel.ipl.pt/programacao/ave/

Sumário

- Garbage collection
- O tipo System.GC
- Implementação de tipos que armazenam recursos nativos (*unmanaged*)
  - Utilização do método virtual System.Object.Finalize() e da interface IDisposable

Garbage collection (1)

- O CLR gere a memória dos objectos alojados num programa através de um processo designado *garbage collection*

- Os objectos .NET são alojados numa zona de memória designada *managed heap*, onde são automaticamente destruídos pelo *garbage collector* "algures no futuro"
  - O *garbage collector* remove um objecto do *heap*, libertando a sua memória, quando este deixar de ser necessário, isto é, quando este é *inalcançável* por qualquer parte do código base

Garbage collection (2)

- Exemplo:

```
static void MakeACar()
{
  // Dado que myCar é a única referência para o objecto Car,
  // "pode ser" detruído quando o método retornar.
  Car myCar = new Car();
}
```

- Apenas se pode assumir que o objecto referido por myCar é destruído quando o CLR realizar a próxima recolha

Garbage collection (3)

MSIL do método MakeACar():

```
.method public hidebysig static void MakeACar() cil managed
{
      // Code size 7 (0x7)
      .maxstack 1
      .locals init ([0] class SimpleGC.Car c)
      IL_0000: newobj instance void Car::.ctor()
      IL_0005: stloc.0
      IL_0006: ret
}
```

Garbage collection (5)

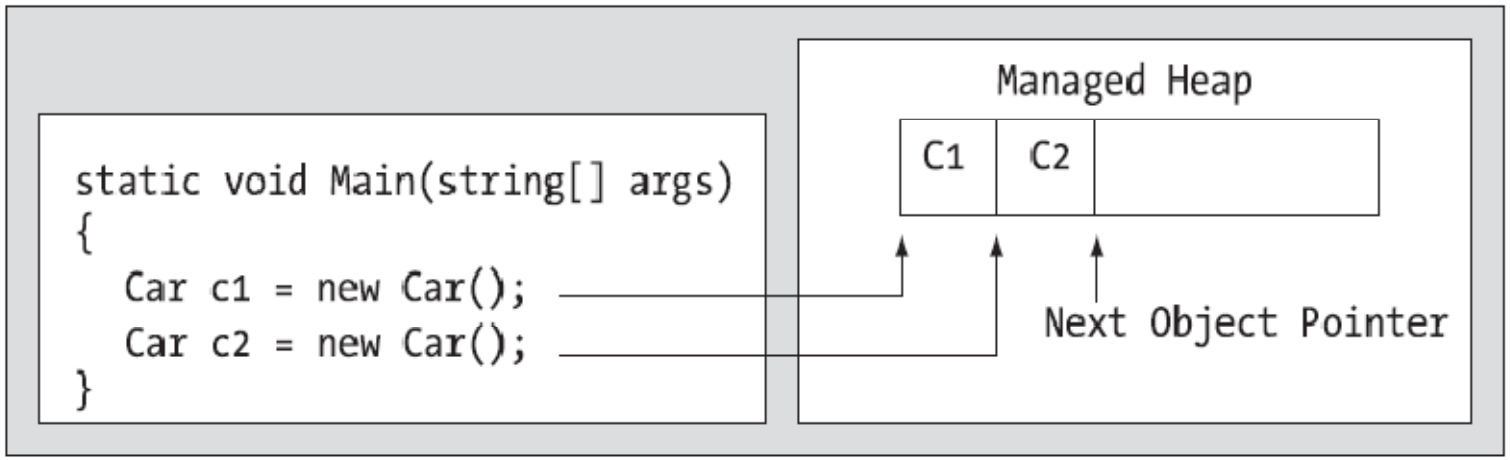
- A instrução newobj envolve a realização das seguintes tarefas pelo CLR:
  - Calcular o total de memória necessária para alojar o objecto (memória requerida pelos membros de dados do tipo e dos tipos base, mais membros *sync* e *type object pointer*)
  - Examinar o *managed heap* para garantir que existe espaço suficiente para alojar o objecto.
  - Se for este o caso, o construtor é invocado, sendo retornada a referência para o novo objecto na memória, cujo endereço é a posição do ponteiro *next object pointer*
  - Finalmente, antes de retornar a referência, avança o ponteiro *next object pointer* para a próxima posição livre no *managed heap*

Garbage collection (4)

- O *garbage collector* mantém a memória *managed heap* organizada, compactando os blocos de memória quando os objectos alojados deixam de ser alcançáveis

- O *managed heap* mantém um ponteiro (referido comummente como *next object pointer* ou *new object pointer*) que identifica a posição onde o próximo objecto vai ficar situado

Garbage collection (6)



- Quando, ao processar a instrução newobj, o CLR determinar que no *managed heap* não existe memória suficiente para alojar o objecto, realiza uma recolha (*garbage collection*) para tentar libertar memória

Raízes da aplicação ("Application Roots")

- Uma raiz (*root*) é uma localização de memória contendo uma referência para um objecto no *heap*
- Podem ser raízes:
  - Referências estáticas para objectos
  - Referências para objectos locais no código da aplicação
  - Referências para objectos passados como parâmetros num método
  - Referências para objectos à espera de serem finalizados (*finalized)*, presentes na lista *freacheable*

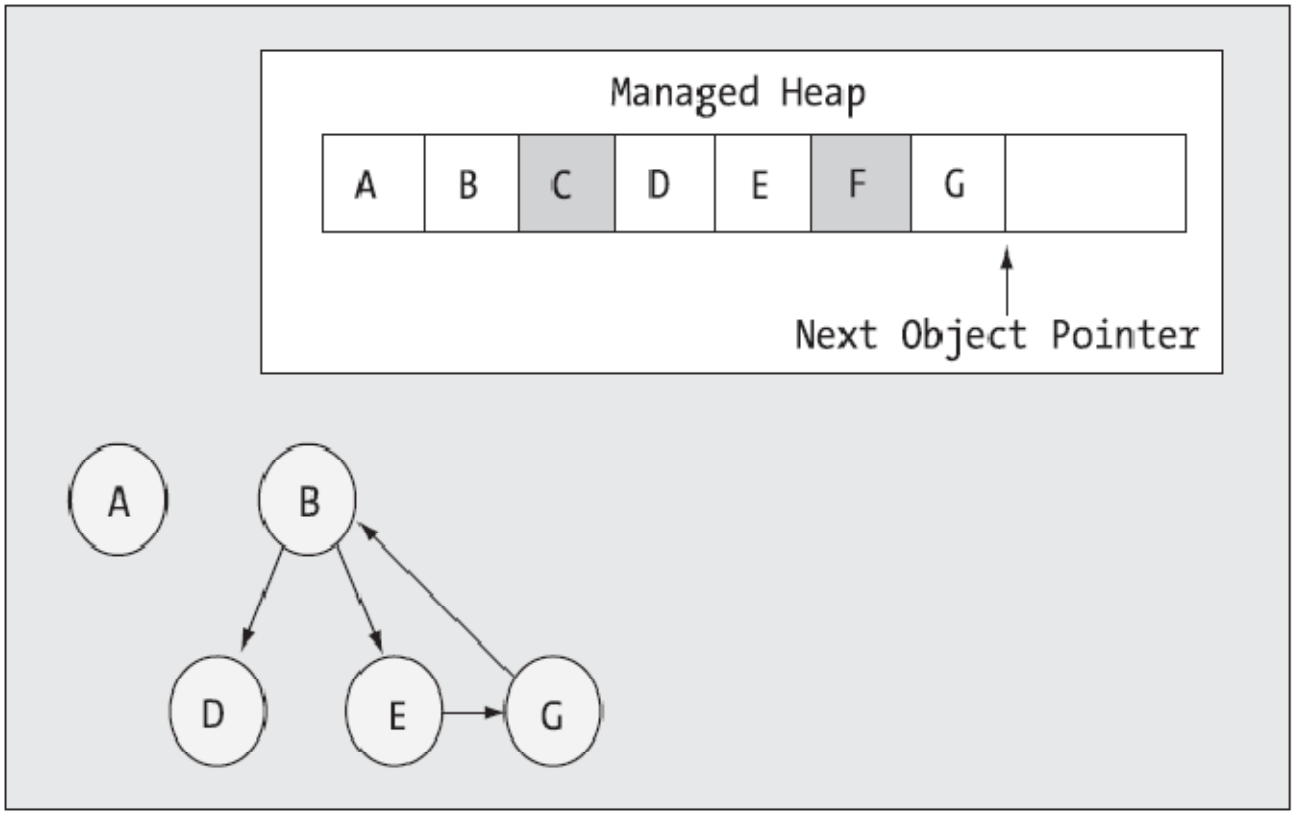
- Durante o processo de *garbage collection*, o AVE irá investigar os objectos no *managed heap* para determinar se ainda são alcançáveis (*rooted*) pela aplicação.
  - O CLR constrói um *grafo de objectos*, que representa cada objecto alcançável no *heap*

Algoritmo de GC (1)

- O algoritmo de GC executa duas fases: uma de *Marcação* e outra de *Compactação*

- Inicialmente, assume que todos os objectos são "lixo"
  - Começa por verificar as referências *roots* e que referem objectos em memória
    - Cada um destes objectos alcançáveis é marcado num *bit* do campo *sync block index* presente no objecto
    - Se o objecto contiver campos do tipo referência, estes objectos, se existirem, são também marcados
    - O GC continua a percorrer recorrentemente todos os objectos alcançáveis

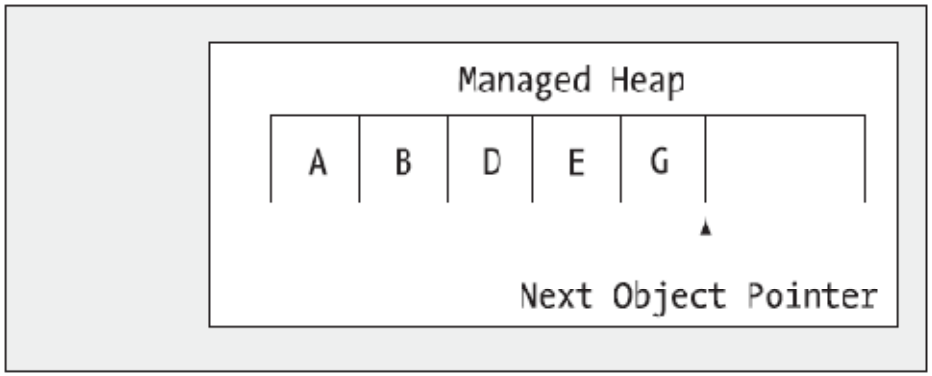
Algoritmo de GC (2)



- G depende de E e indirectamente de B
- A não depende de ninguém

Algoritmo de GC (3)

- Os objectos C e F não foram marcados pelo GC, sendo considerados “lixo”
  - São removidos da memória
  - O espaço existente no *heap* é compactado, invalidando todas as variáveis e registos do CPU que contêm ponteiros para objectos
    - O CLR tem que revisitar todas as raízes da aplicação e modificar as referências para as novas localizações
- Finalmente, o ponteiro *next object pointer* é reajustado para apontar o próximo *slot* livre



Gerações de objectos (1)

- Quando o CLR procura objectos inalcançáveis, não examina literalmente cada objecto situado no *managed heap*
  - Fazer isto envolveria um tempo considerável, especialmente em aplicações reais de grande dimensão
- Para optimizar o processo, cada objecto no *heap* é atribuído a uma "geração" específica
- A ideia por detrás das gerações é a seguinte:
  - quanto mais antigo for um objecto no *heap*, maior é a probabilidade de se manter no *heap*; ex: o objecto que implementa a aplicação manter-se-á na memória até que o programa termine
  - Por outro lado, objectos recentemente colocados no *heap* (ex: objecto alojado no código dum método) têm maior probabilidade de se tornarem inalcançáveis (se não forem armazenados algures, ficam inalcançáveis assim que o método terminar)

Gerações de objectos (2)

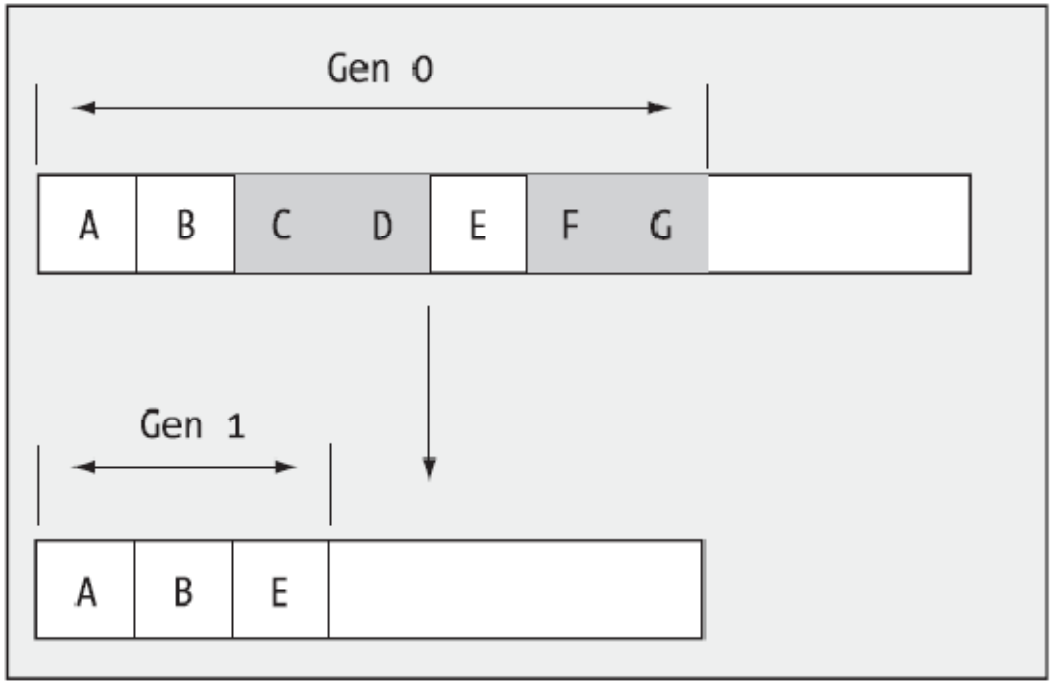
- Cada objecto no *heap* pertence a uma das seguintes gerações:
  - *Geração 0*: Identifica novos objectos recentemente alojados que nunca foram marcados numa recolha
  - *Geração 1*: Identifica objectos que sobreviveram a um *garbage collection* (objectos alcançáveis que sobreviveram ao *garbage collection* feito na geração 0)
  - *Geração 2*: Identifica objectos que sobreviveram a mais do que um varrimento do *garbage collector*
- Dimensões das gerações criadas por omissão (contudo, podem variar ao longo da execução do programa)
  - Geração 0 ~ 256 KB
  - Geração 1 ~ 2 MB
  - Geração 2 ~ 10 MB

Gerações de objectos (3)

- Nota: geralmente, só é feito um GC quando a geração 0 estiver cheia
- O *garbage collector* começa por investigar primeiro os objectos da geração 0
  - Os objectos são avaliados e marcados para remoção (se forem inalcançáveis)
  - Os objectos marcados são removidos; os objectos que sobreviveram são promovidos para a geração 1
- Quando a geração 1 ficar preenchida, O GC irá varrer esta
- Os objectos que sobreviverem à geração 1 são promovidos à geração 2.
- Quando a geração 2 ficar preenchida, é também avaliada. Os objectos que sobreviverem, permanecem na geração 2

Gerações de objectos (4)

- Os objectos A, B e E, da geração 0, sobreviveram à recolha e passam à geração 1



Gerações de objectos (5)

- Podem existir gerações contendo objectos inalcançáveis
- Por factores de optimização, o GC varre a geração que contém mais memória ocupada, pois é provável que exista nessa geração vários objectos que entretanto ficaram inalcançáveis
  - o GC varre igualmente a geração 0 (não esquecer que, usualmente, o GC é desencadeado devido à Ger 0 estar totalmente preenchida)
- Como os objectos são alojados na geração 0, esta é varrida com maior frequência que as restantes

Gerações de objectos (6)

- Com os objectos organizados em gerações, novos objectos (como variáveis locais temporárias) irão ser removidos rapidamente
- Enquanto objectos mais antigos (como o objecto que representa a aplicação) não são avaliados tão frequentemente

O tipo System.GC (1)

System.GC Member	Meaning in Life
`AddMemoryPressure()` `RemoveMemoryPressure()`	Allow you to specify a numerical value that represents the calling object's "urgency level" regarding the garbage collection process. Be aware that these methods should alter pressure *in tandem* and thus never remove more pressure than the total amount you have added.
`Collect()`	Forces the GC to perform a garbage collection. This method has been overloaded to specify a generation to collect, as well as the mode of collection (via the `GCCollectionMode` enumeration).
`CollectionCount()`	Returns a numerical value representing how many times a given generation has been swept.
`GetGeneration()`	Returns the generation to which an object currently belongs.
`GetTotalMemory()`	Returns the estimated amount of memory (in bytes) currently allocated on the managed heap. The Boolean parameter specifies whether the call should wait for garbage collection to occur before returning.
`MaxGeneration`	Returns the maximum of generations supported on the target system. Under Microsoft's .NET 3.5, there are three possible generations (0, 1, and 2).
`SuppressFinalize()`	Sets a flag indicating that the specified object should not have its `Finalize()` method called.
`WaitForPendingFinalizers()`	Suspends the current thread until all finalizable objects have been finalized. This method is typically called directly after invoking `GC.Collect()`.

O tipo System.GC (2)

- Raramente há necessidade de manipular o *garbage collector* programaticamente

- Eis três situações em que é benéfico forçar um *garbage collection* usando GC.Collect() :

  - A aplicação vai executar um bloco de código onde não pretende ser interrompida com um possível *garbage collection*
  - A aplicação vai alojar um número grande de objectos e deseja-se limpar a memória de outros objectos que já não são referenciados
  - Uma aplicação gráfica mantém em memória vários *handles* para *bitmaps* (recursos nativos *unmanaged*) e deseja fazer uma recolha para provocar a finalização e consequente libertação destes recursos

O tipo System.GC (3)

```
static void Main(string[] args)
{
    ...
    // Força um garbage collection e espera
    // que cada objecto seja finalizado
    GC.Collect();
    GC.WaitForPendingFinalizers();
    ...
}
```

O tipo System.GC (4)

- Quando se força manualmente a recolha, deve chamar-se sempre o método GC.WaitForPendingFinalizers()

- Com esta abordagem, dá-se a hipótese de objectos que necessitem de finalização (*finalizable objects*) de realizar a libertação de recursos nativos antes do programa continuar

- O método GC.WaitForPendingFinalizers() suspende a *thread* invocadora durante o processo de recolha

Construir objectos finalizáveis (1)

- System.object define um método virtual (vazio) para ser redefinido por objectos que necessitem de finalização
  - Objectos que armazenem recursos nativos *unmanaged* ex: handles de ficheiros, sockets, ligações a base de dados

```
// System.Object
public class Object
{
    ...
    protected virtual void Finalize() { }
}
```

Construir objectos finalizáveis (2)

- A chamada ao método Finalize() irá (eventualmente) ocorrer durante uma recolha

- Adicionalmente, este método é chamado automaticamente sobre todos os objectos finalizáveis assim que o *application domain* (AppDomain) que alberga a aplicação seja descarregado da memória
  - De forma semelhante a um destrutor em código C++

Construir objectos finalizáveis (3)

- A redefinição directa de Finalize() provoca um erro de compilação:

```
public class MyResourceWrapper
{
    // Compile-time error!
    protected override void Finalize(){ }
}
```

- Deve usar-se a sintaxe alternativa dos destrutores em C++ dado que o comp. C# adiciona mais código extra à redefinição (chamar o Finalize da classe base)

Construir objectos finalizáveis (4)

```
// Override System.Object.Finalize() via finalizer syntax.
class MyResourceWrapper
{
    ~MyResourceWrapper()
    {
        // Clean up unmanaged resources here.
        // Beep when destroyed (testing purposes only!)
        Console.Beep();
    }
}
```

- Métodos finalizadores nunca têm modificador de acessibilidade (são implicitamente *protected*), não recebem parâmetros e não podem ser *overloaded* (apenas um finalizador por classe)

Construir objectos finalizáveis (5)

```
.method family hidebysig virtual instance void
Finalize() cil managed
{
      // Code size 13 (0xd)
      .maxstack 1
      .try
      {
         IL_0000: ldc.i4 0x4e20
         IL_0005: ldc.i4 0x3e8
         IL_000a: call
            void [mscorlib]System.Console::Beep(int32, int32)
         IL_000f: nop
         IL_0010: nop
         IL_0011: leave.s IL_001b
      } // end .try

      ...
```

Construir objectos finalizáveis (6)

```
    finally
    {
        IL_0013: ldarg.0
        IL_0014:
          call instance void [mscorlib]System.Object::Finalize()
        IL_0019: nop
        IL_001a: endfinally
     } // end handler
     IL_001b: nop
     IL_001c: ret
} // end of method MyResourceWrapper::Finalize
```

Construir objectos finalizáveis (7)

```
static void Main(string[] args)
{
      Console.WriteLine("***** Fun with Finalizers *****\n");
      Console.WriteLine("Hit the return key to shut down this app");
      Console.WriteLine("and force the GC to invoke Finalize()");
      Console.WriteLine("for finalizable objects created in this
      AppDomain.");
      Console.ReadLine();
      MyResourceWrapper rw = new MyResourceWrapper();
}
```

Processo de finalização (1)

- Quando um objecto é alojado no *managed heap*, o AVE determina automaticamente se o objecto suporta um método Finalize()
- Se sim, o objecto é marcado como *finalizable*, e é armazenado numa queue interna (designada *finalization queue*) um pointer para este objecto
  - Esta queue mantém todos os objectos que necessitam de finalização antes de ser removidos

- Quando o *garbage collector* determina que está na altura de remover o objecto da memória,
  - examina cada entrada da finalization queue e copia o objecto do heap para outra estrutura *managed* designada *finalization reachable table* (conhecida pela abreviação *freachable*)

Processo de finalização (2)

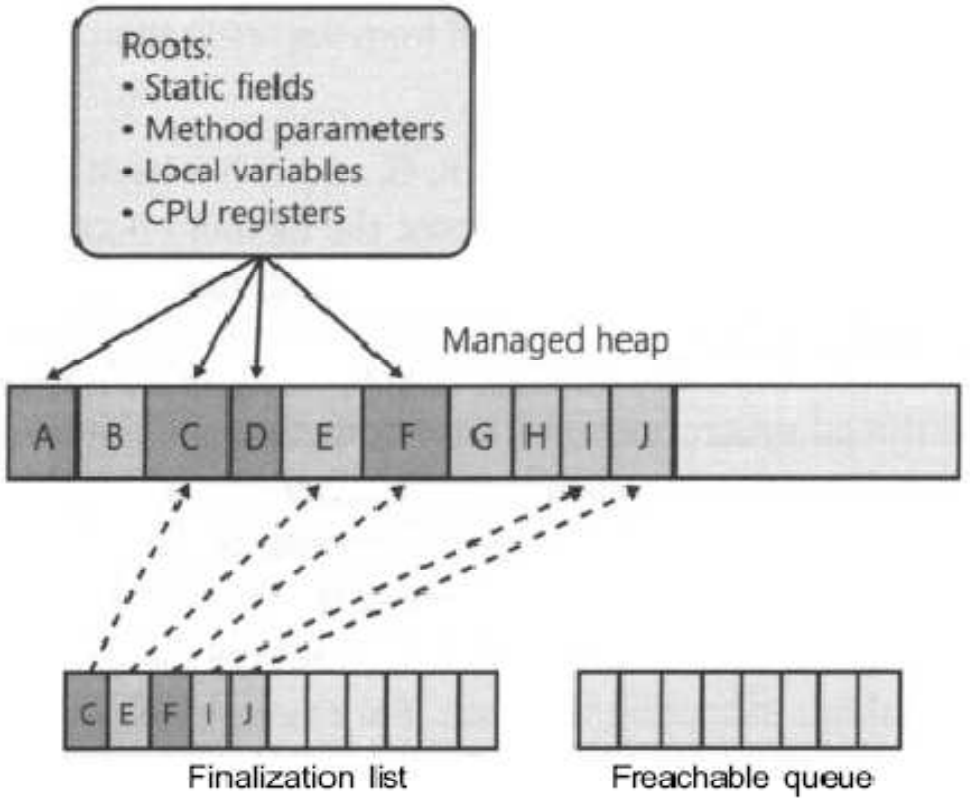


Figure 20-5 The managed heap showing pointers in its finalization list

Processo de finalização (3)

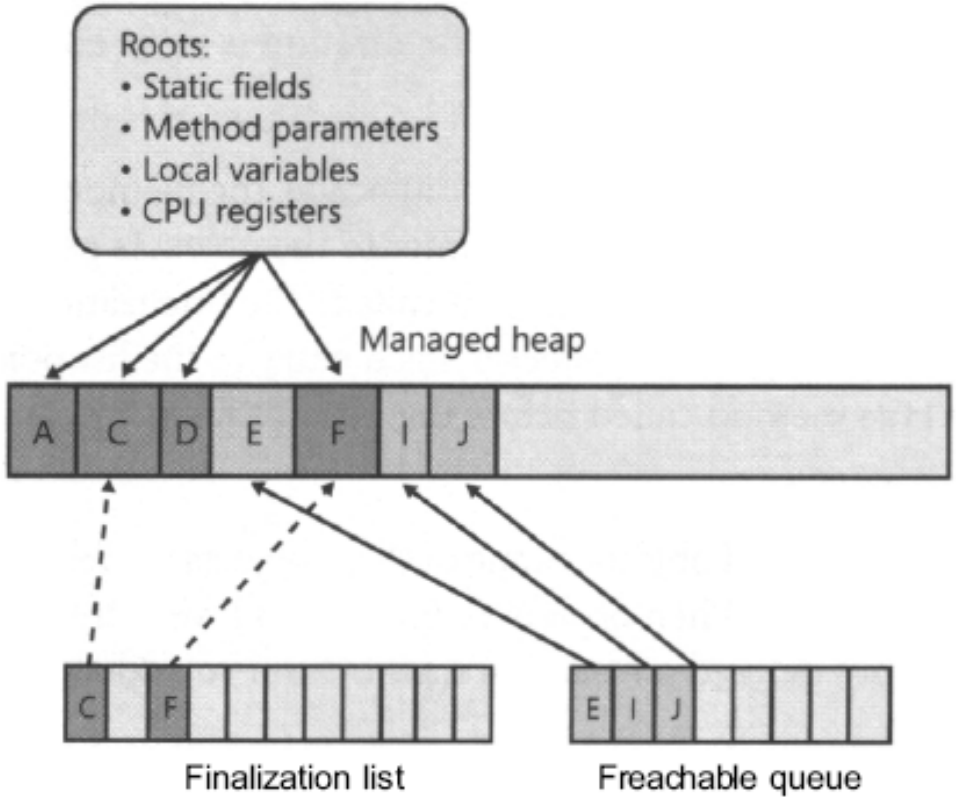


Figure 20-6 The managed heap showing pointers that moved from the finalization list to the freachable queue

Processo de finalização (4)

- Neste ponto, a *thread* responsável pelo processo de finalização acorda para invocar o método Finalize() para cada objecto na tabela *freachable*
  - só na próxima recolha, se já tiver sido finalizado (não constando, deste modo, na *freachable list*) é que o objecto poderá ser limpo
- são necessárias, no mínimo, duas recolhas para limpar um objecto finalizável
- Enquanto a finalização de um objecto garante a libertação dos seus recursos nativos *unmanaged*, é um processo não determinístico por natureza, e devido ao processamento extra, é consideravelmente mais lento
- A *thread* responsável pela finalização não deve ser bloqueada
  - Sob pena de não prosseguir com a finalização dos restantes objectos
  - Assim, o código do Finalize() de um objecto não deve ser bloqueante

Processo de finalização (5)

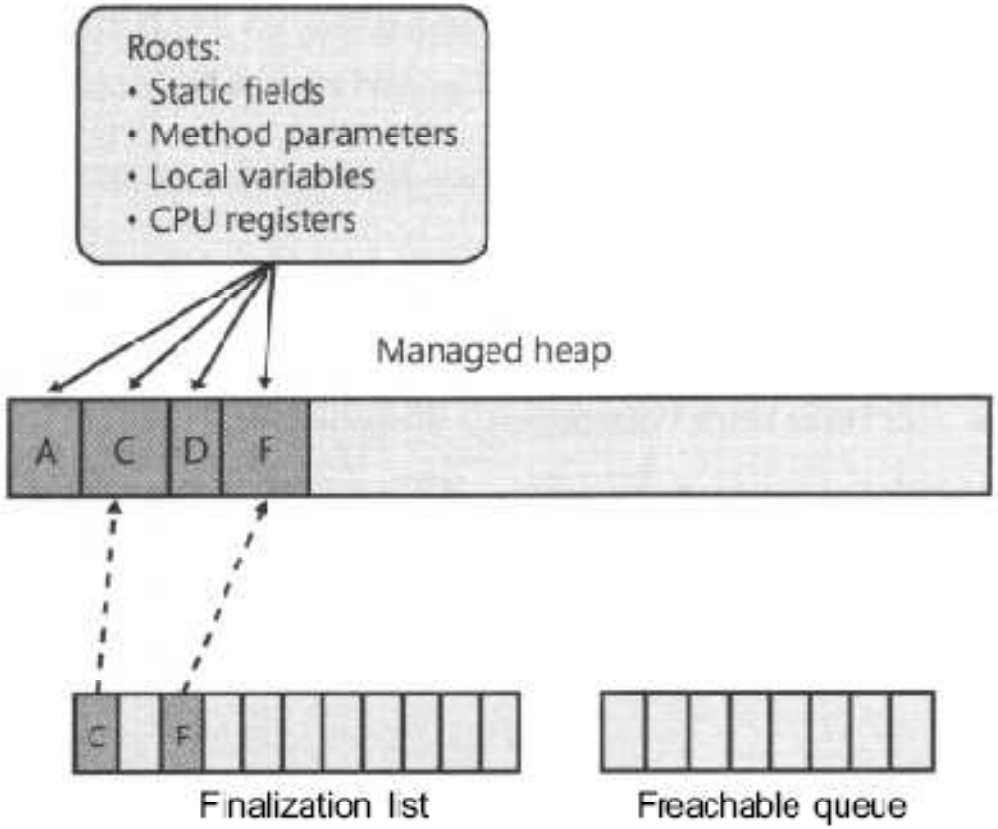


Figure 20-7 Status of managed heap after second garbage collection

Construir um tipo Disposable (1)

- Finalizadores são usados para finalizar um objecto quando ocorre uma recolha
- Se se pretender libertar os recursos imediatamente sem "esperar" pelo finalizador implementa-se alternativamente a interface IDisposable

```
public interface IDisposable
{
    void Dispose();
}
```

- Vantagem: o objecto pode libertar os recursos sem ser necessário incorrer no custo do processamento da lista de finalização e sem ter que esperar pelo *garbage collector* para activar a lógica de finalização do objecto

Construir um tipo Disposable (2)

```
// Implementing IDisposable.
public class MyResourceWrapper : IDisposable
{
    // The object user should call this method
    // when they finish with the object.
    public void Dispose()
    {
        // Clean up unmanaged resources...
        // Dispose other contained disposable objects...
        // Just for a test.
        Console.WriteLine("***** In Dispose! *****");
    }
}
```

Construir um tipo Disposable (3)

- Ao contrário do método Finalize() que não deve comunicar (nem finalizar) outros objectos membros finalizáveis, no método Dispose() deve fazer-se *dispose* dos membros deste objecto
  - O *garbage collector* não tem conhecimento do método Dispose(): é um método como outro qualquer

```
public class Program
{
    static void Main()
    {
        Console.WriteLine("***** Fun with Dispose *****\n");
        MyResourceWrapper rw = new MyResourceWrapper();
        if (rw is IDisposable)
           rw.Dispose();
        Console.ReadLine();
    }
}
```

Palavra chave using (1)

```
static void Main(string[] args)
{
      MyResourceWrapper rw = new
            MyResourceWrapper ();
      try
      {
            // Use the members of rw.
      }
      finally
      {
            // Always call Dispose(),
            //error or not.
            rw.Dispose();
      }
}
```

```
static void Main(string[] args)
{
      // Dispose() is called
      automatically when the
      // using scope exits.

      using(MyResourceWrapper rw =
            new MyResourceWrapper())
      {
            // Use rw object.
      }
}
```

Palavra chave using (2)

```
.method private hidebysig static void Main(string[] args) cil managed
{
      ...
      .try
      {
            ...
      } // end .try
      finally
      {
            ...
            IL_0012: callvirt instance void
            SimpleFinalize.MyResourceWrapper::Dispose()
      } // end handler
      ...
} // end of method Program::Main
```

Construir um tipo Finalizable e Disposable

```
// A sophisticated resource wrapper.
public class MyResourceWrapper : IDisposable  {
      // The garbage collector will call this method if the
      // object user forgets to call Dispose().
      ~ MyResourceWrapper()  {
            // Clean up any internal unmanaged resources.
            // Do **not** call Dispose() on any managed objects.
      }
      // The object user will call this method to clean up
      // resources.
      public void Dispose() {
            // Clean up unmanaged resources here.
            // Call Dispose() on other contained disposable objects.
            // No need to finalize if user called Dispose(), so suppress
            finalization.
            GC.SuppressFinalize(this);
      }
}
```

Padrão Disposable (1)

- A implementação anterior sofria de três problemas:
  - Código repetido no Finalize() e Dispose()
  - Deve usar-se um método privado para realizar o processamento
  - Garantir que o Finalize() não deverá fazer *dispose* de objectos managed enquanto o Dispose() o deverá fazer
  - Garantir que o código do Dispose() não é realizado múltiplas vezes, resultante de múltiplas invocações pelo utilizador
    - Mas apenas uma vez

Padrão Disposable (2)

```
public class MyResourceWrapper : IDisposable
{
    // Used to determine if Dispose()
    // has already been called.
    private bool disposed = false;
    public void Dispose()
    {
        // Call our helper method.
        // Specifying "true" signifies that
        // the object user triggered the cleanup.
        CleanUp(true);

        // Now suppress finalization.
        GC.SuppressFinalize(this);
        }
```

Padrão Disposable (3)

```
        private void CleanUp(bool disposing) {
            // Be sure we have not already been disposed!
            if (!this.disposed) {
                // If disposing equals true, dispose all
                // managed resources.
                if (disposing) {
                        // Dispose managed resources.
                }
                // Clean up unmanaged resources here.
            }
            disposed = true;
        }
        ~MyResourceWrapper() {
            // Call our helper method. Specifying "false" signifies
            that the GC triggered the cleanup.
                CleanUp(false);
        }
}
```