SoftServe

Empowering your Business through Software Development

The Basics of Object Lifetime. Disposing objects.

01.09.2014

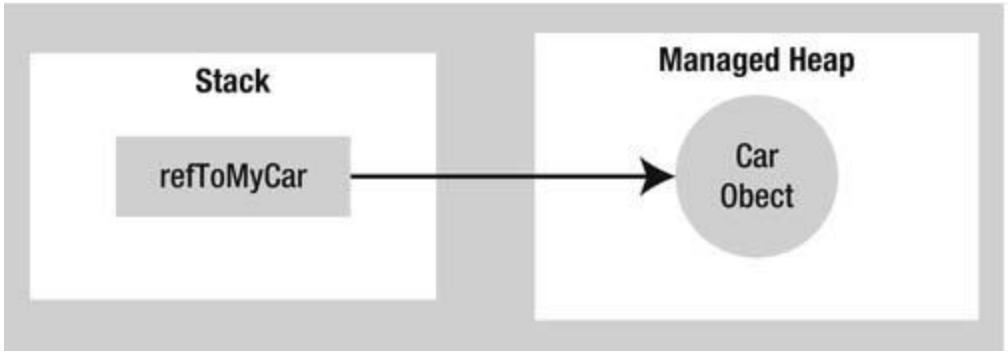
Approved by V. Bartish

AGENDA

- The Basics of Object Lifetime
- Building Finalizable Objects
- Building Disposable Objects
- Dispose pattern

The Basics of Object Lifetime

References to objects on the managed heap

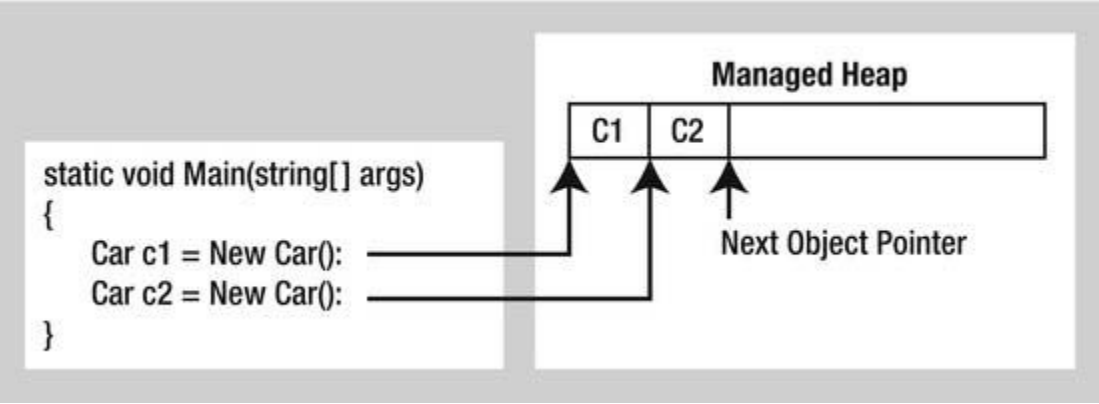



```
Car refToMyCar = new Car("Zippy", 50);
```

■ **Rule** Allocate a class instance onto the managed heap using the new keyword and forget about it.

The Basics of Object Lifetime

The details of allocating objects onto the managed heap

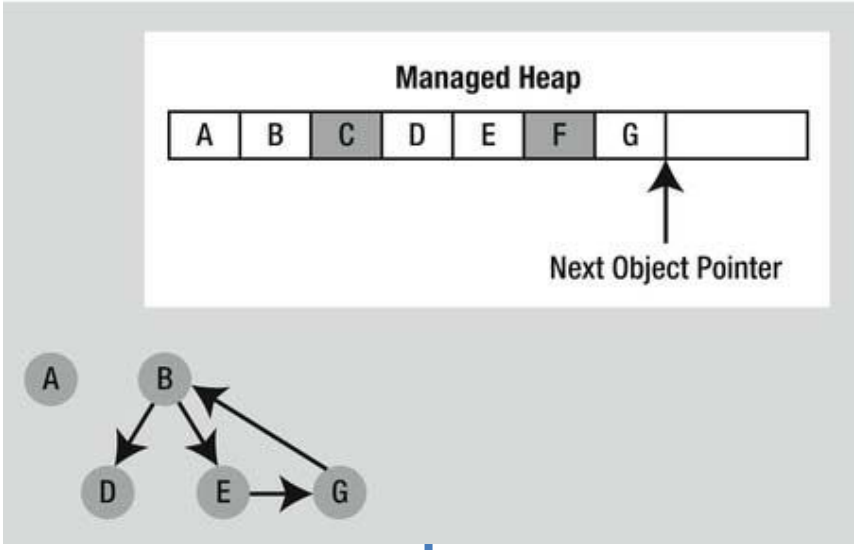


■ **Rule** If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur.

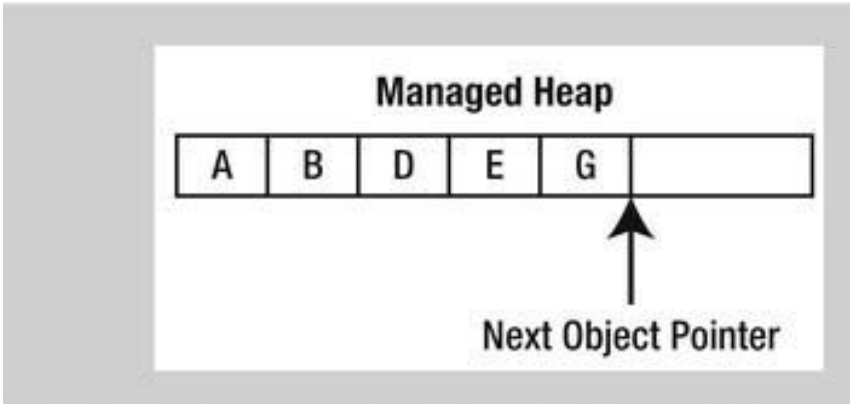
c2-=null;

! assigning a reference to null does not force the garbage collector to remove the object from the heap.

The Role of Application Roots

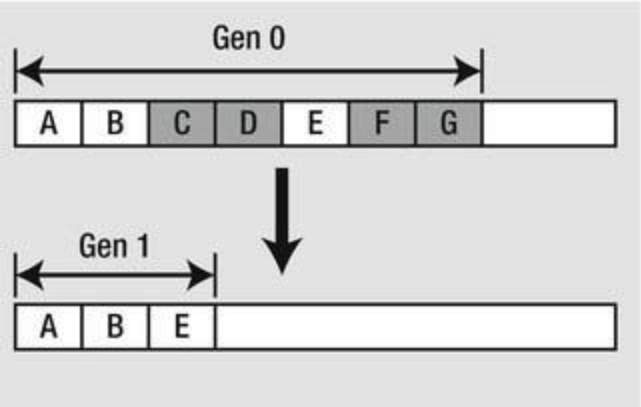


A clean and compacted heap



- ***Root* is a storage location containing a reference to an object on the managed heap**:
  - References to global objects
  - References to any static objects/static fields
  - References to local objects within an application's code base
  - References to object parameters passed into a method
  - References to objects waiting to be *finalized*
  - Any CPU register that references an object

Object Generations



- Each object on the heap belongs to one of the following generations:
  - *Generation 0*: Identifies a newly allocated object that has never been marked for collection.
  - *Generation 1*: Identifies an object that has survived a garbage collection (i.e., it was marked for collection but was not removed due to the fact that the sufficient heap space was acquired).
  - *Generation 2*: Identifies an object that has survived more than one sweep of the garbage collector.

■ **Note** Generations 0 and 1 are termed *ephemeral generations*.

Building Finalizable Objects

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

- it is not possible to directly call an object's **Finalize**() method from a class instance .
- the ***garbage collector*** will call an object's Finalize() method before removing the object from memory.

Rule The reason to override Finalize() is if your C# class is making use of unmanaged resources via PInvoke or complex COM interoperability tasks. The reason is that you are manipulating memory that the CLR cannot manage.

Building Finalizable Objects

```
// Override System.Object.Finalize() via finalizer syntax.
class MyResourceWrapper
{
  ~MyResourceWrapper()
  {
    // Clean up unmanaged resources here.

    // Beep when destroyed (testing purposes only!)
    Console.Beep();
  }
}
```

- You can't override the Finalize() method directly in your class, but you may use of a **destructor** syntax to achieve the same effect.
- **Destructor** never takes an access modifier (implicitly protected), never takes parameters, and can't be overloaded (only one finalizer per class).

Building Disposable Objects

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

- **Structures** and **class** types can both implement **IDisposable** (unlike overriding Finalize(), which is reserved for class types), as the object user (not the garbage collector) invokes the Dispose() method.
- When the ***object user*** is finished using the object, the **object user manually calls Dispose()** before allowing the object reference to drop out of scope.

Building Disposable Objects

■ **Rule** It is a good idea to call Dispose() on any object you directly create if the object supports IDisposable. The assumption you should make is that if the class designer chose to support the Dispose() method, the type has some cleanup to perform. If you forget, memory will eventually be cleaned up (so don’tpanic), but it could take longer than necessary.

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

Building Disposable Objects

- A number of types in the base class libraries that do implement the **Idisposable** interface provide a (somewhat confusing) alias to the **Dispose**() method, in an attempt to make the disposal-centric method sound more natural for the defining type.
- The **System.IO.FileStream** class implements **IDisposable** (and therefore supports a **Dispose**() method), it also defines the following **Close**() method that is used for the same purpose:

```
static void DisposeFileStream()
{
  FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);

  // Confusing, to say the least!
  // These method calls do the same thing!
  fs.Close();
  fs.Dispose();
}
```

using

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  MyResourceWrapper rw = new MyResourceWrapper ();
  try
  {
  }
}
```

- **Note** If you attempt to “use” an object that does not implement **IDisposable**, you will receive a **compiler error**.

```
void Main(string[] args)
{
  Con    eLine("***** Fun with Dispose *****\n");
  // Di      is called automatically when the
  // usin    pe exits.
  using(MyResourceWrapper rw = new MyResourceWrapper())
  {
    // Use rw object.
  }
}
```

?

```
// 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 ASAP.
  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);
  }
}
```

GC.SuppressFinalize() informs the CLR that it is no longer necessary to call the destructor when this object is garbage-collected

Dispose pattern

- The **Dispose** Pattern is intended to standardize the usage and implementation of finalizers and the IDisposable interface.

√ **DO** implement the Basic Dispose Pattern on types containing instances of disposable types.

√ **DO** implement the Basic Dispose Pattern and provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers.

√ **CONSIDER** implementing the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do.

Dispose pattern

1) Involves implementing the **System.IDisposable** interface
2) Declare the **Dispose(bool)** method that implements all resource cleanup logic to be shared between the Dispose method and the optional finalizer.

```
public class DisposableResourceHolder : IDisposable
{
   private SafeHandle resource; // handle to a resource
   public DisposableResourceHolder()
    {
      this.resource = ... // allocates the resource
    }
   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing){
      if (disposing){
         if (resource!= null) resource.Dispose();
      }
   }
}
```

Dispose pattern

- **DO NOT** make the parameterless Dispose method virtual.
- The Dispose(bool) method is the one that should be overridden by subclasses.

```
// bad design
public class DisposableResourceHolder : IDisposable
{
   public virtual void Dispose(){ ... }
   protected virtual void Dispose(bool disposing){ ... }
}

// good design
public class DisposableResourceHolder : Idisposable
 {
   public void Dispose(){ ... }
   protected virtual void Dispose(bool disposing){ ... }
}
```

Dispose pattern

- √ **DO** allow the Dispose(bool) method to be called more than once. The method might choose to do nothing after the first call.

```
public class DisposableResourceHolder : IDisposable
{
    bool disposed = false;

   protected virtual void Dispose(bool disposing)
{
       if(disposed) return;
       // cleanup
       ...
       disposed = true;
   }
}
```

Dispose pattern

- √ **DO** throw an ObjectDisposedException from any member that cannot be used after the object has been disposed of.
-

```
public class DisposableResourceHolder : IDisposable
{
   bool disposed = false;
   SafeHandle resource; // handle to a resource

   public void DoSomething()
{
        if(disposed) throw new ObjectDisposedException(...);
      // now call some native methods using the resource
         ...
   }
   protected virtual void Dispose(bool disposing)
{
      if(disposed) return;
      // cleanup
      ...
      disposed = true;
   }
}
```

Dispose pattern

- √ **CONSIDER** providing method Close(), in addition to the Dispose(), if close is standard terminology in the area.
-

```
public class Stream : IDisposable
{
   IDisposable.Dispose(){
      Close();
   }
   public void Close()
{
      Dispose(true);
      GC.SuppressFinalize(this);
   }
}
```

Questions ?