Module 07

”Object Lifetime”

TEKNOLOGISK INSTITUT

Agenda

- Introducing Lifetime
- Enter Garbage Collection
- Class Destructors
- The Disposable Pattern

Lifespan of an Object

- An object is created
  - Memory is allocated
  - Memory is initialized into an object by running the constructor
- Object is alive and kicking
  - It is passed in and out of methods and operations are invoked
- The object is destroyed
  - The object is de-initialized into unused memory
  - Memory is then deallocated

Objects, Values, and Scope

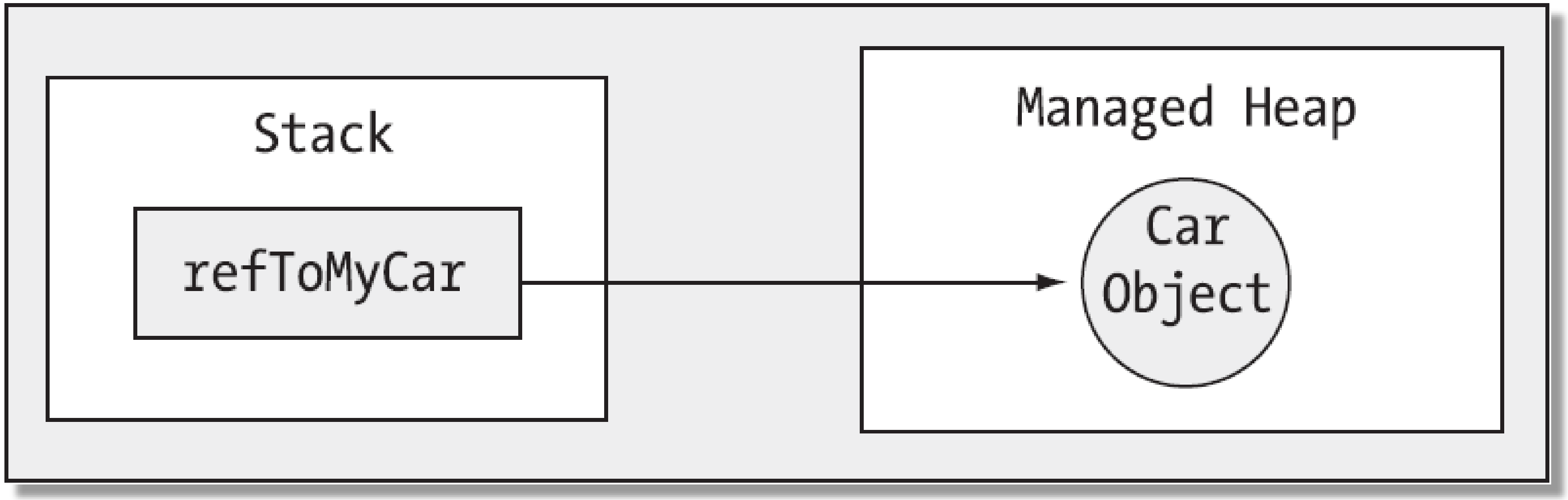
- Local variables live only throughout the scope in which they are declared
  - Fixed lifetime
  - Scheduled destruction

- Objects can outlive the scope in which the were allocated
  - Unbounded lifetime
  - Undetermined destruction

```
static void Main()
{
   bool b = true;
   A longLivingVariable;
   if( b )
   {
      int i = 0;
      while( true )
      {
         A a = new A( i );
         if( ++i % 100 == 0 )
         {
           longLivingVariable = a;
         }
      }
   }
}
```

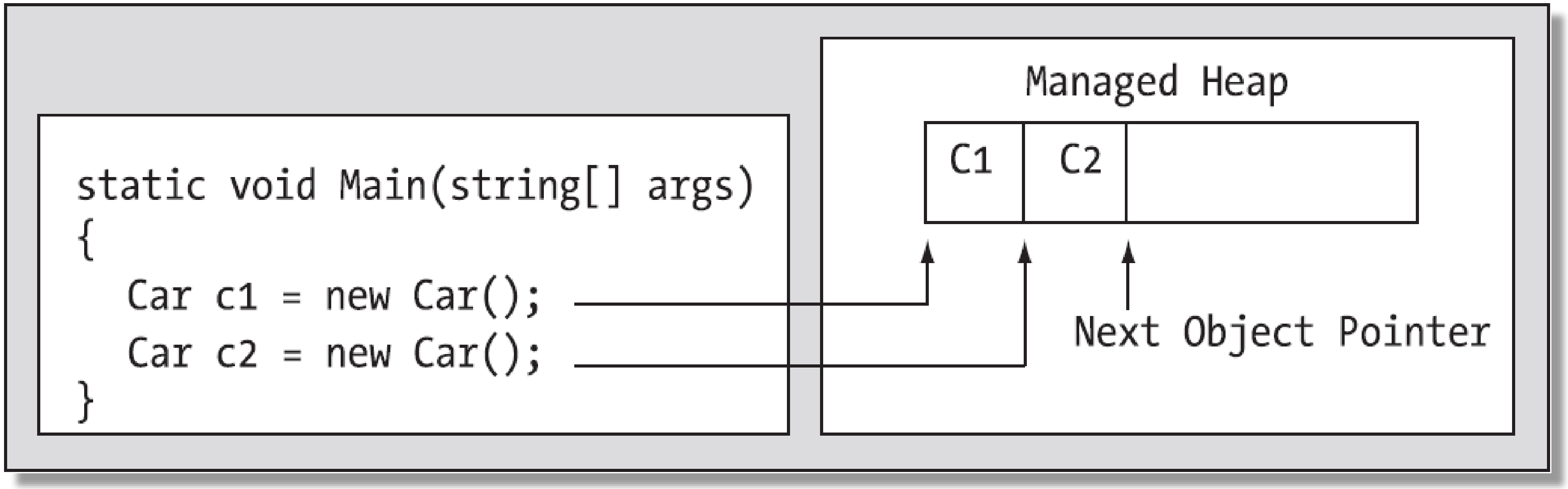
The Stack and The Heap

- Local variables are allocated on the *Stack*
- Objects are allocated on the *Heap*



Allocating Objects

- A new object is always allocated at The Next Object Pointer
  - This pointer is then advanced to the next block



```
static void Main(string[] args)
{
   Car c1 = new Car();
   Car c2 = new Car();
}
```

- Can this go on forever?

Deallocating Objects

- There is no construct in C# to explicitly destroy objects
  - This is to avoid
    - Forgetting to destroy objects
    - Destroying more than once
    - Dangling references
    - …

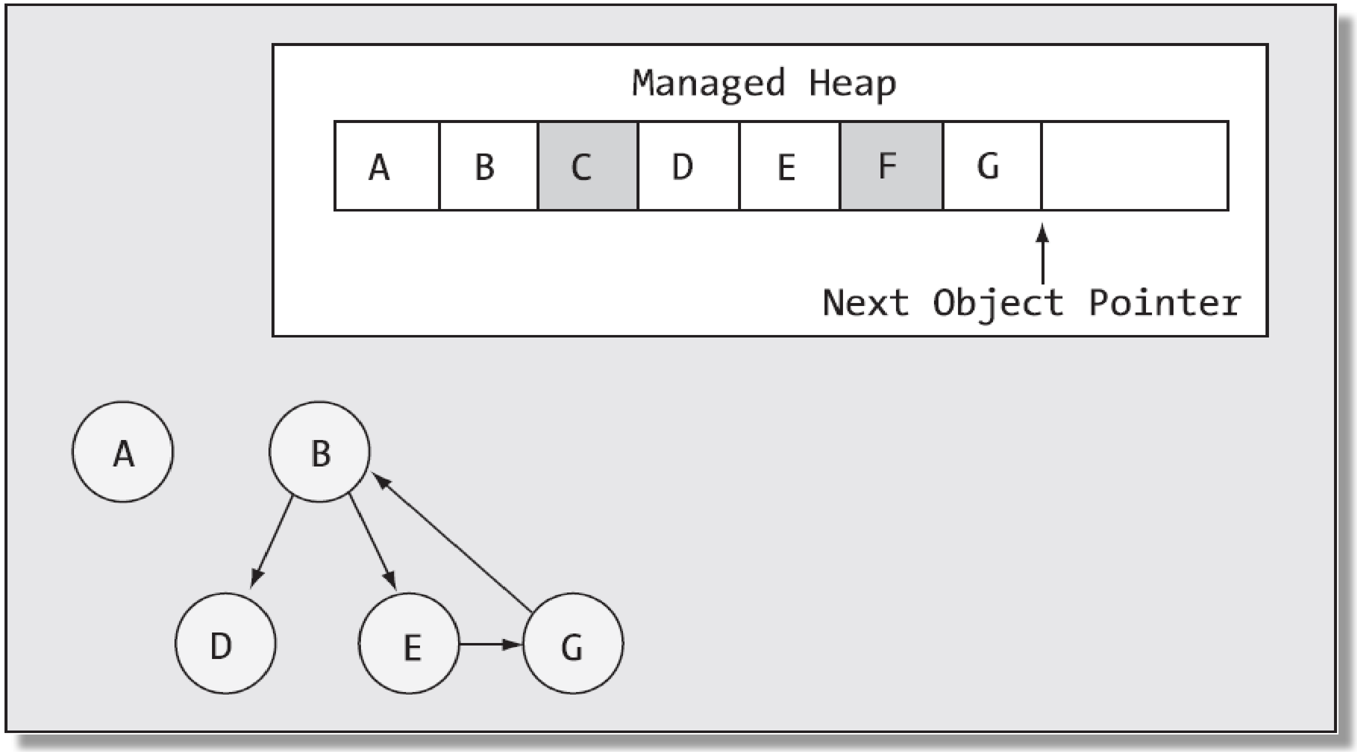
- The garbage collector *finalizes* the objects back into unused memory

Agenda

- Introducing Lifetime
- **Enter Garbage Collection**
- Class Destructors
- The Disposable Pattern

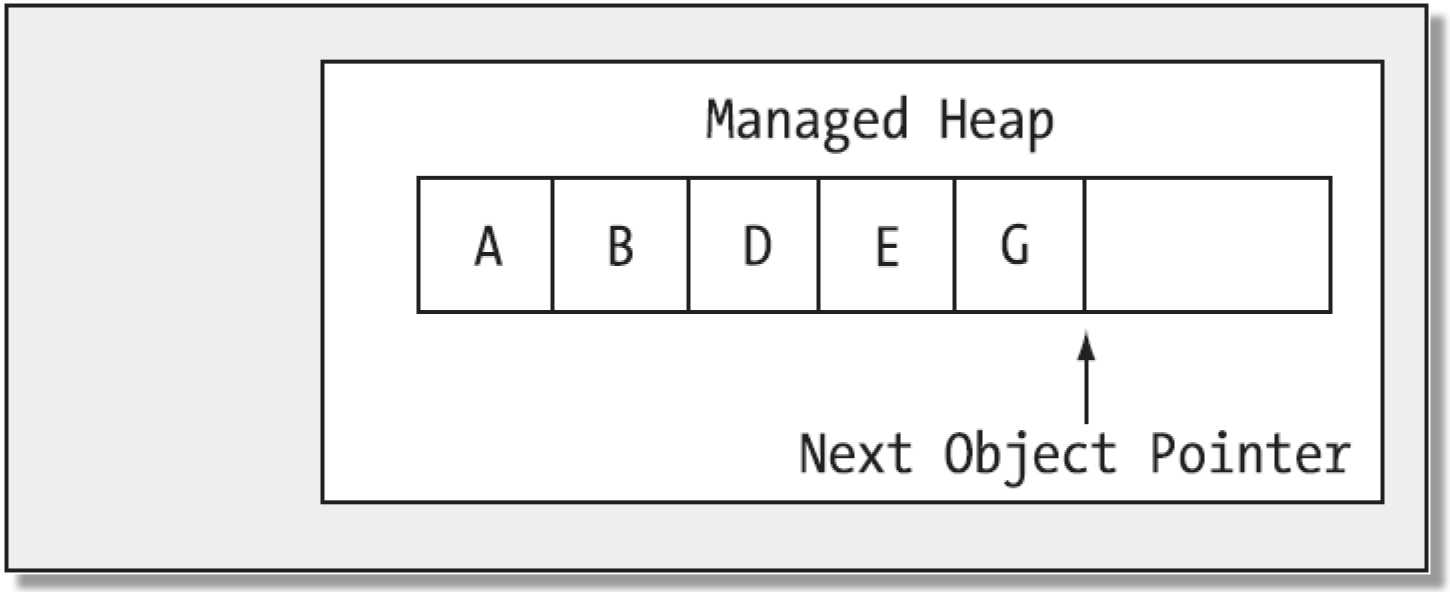
Object Graphs

- 1. Allocation is paused for a short while
- 2. Object graphs are created



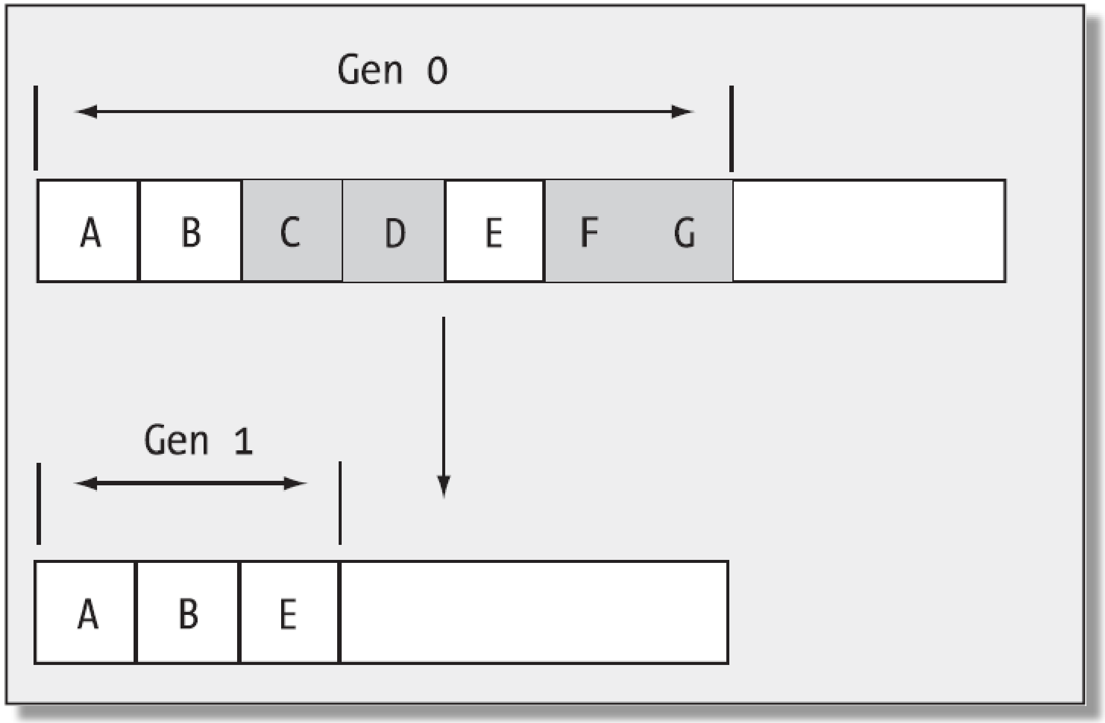
Heap Compaction

- 3. Those objects which are not referenced from *Application Roots* are deallocated
- 4. The Heap is then compacted



Generations

- 5. Generations are updated
- 6. New object is allocated as first new Generation 0 object



The System.GC Type

Name	Characteristics
`Collect()`	Forces a garbage collection given • a generation • a mode
`SuppressFinalize()`	Instructs that the object should not have its **`Finalize()`** method invoked
`WaitForPendingFinalizers()`	Suspends thread until all pending finalizable objects have been finalized
`ReRegisterForFinalize()`	Requests that the system calls the finalizer for the specified object
`AddMemoryPressure()`	Informs the CLR of a large allocation of unmanaged memory
`RemoveMemoryPressure()`	Informs the CLR of the deallocation of a large amount of unmanaged memory
`KeepAlive()`	Forces the object alive

Agenda

- Introducing Lifetime
- Enter Garbage Collection
- **Class Destructors**
- The Disposable Pattern

The `Finalize()` Method

- The garbage collector needs to know how to destroy objects
- The cleanup logic for objects is performed in the **`Finalize()`** method inherited from **`System.Object`**
- This virtual method cannot be overridden or called directly
- Implement a *class destructor* to override **`Finalize()`**
- If present, the garbage collector will invoke destructor just before turning object back into unused memory

Defining Destructors

- Put cleanup logic in the destructor

- Similar to constructors, the destructor is named after the class (but with **~**)
- Similar to constructors, destructors have no return type

- No access modifier is allowed
- Just a single destructor (with no parameters!) is allowed

```
class DataHandler
{   ...
    FileStream fs;

    ~DataHandler()
    {
        fs.Close();
    }
}
```

Destructors and Inheritance

- Destructors are invoked by following the inheritance chain from most specialized first to most general last

```
class A
{
   public A()
   {
      Console.WriteLine( "A()" );
   }
   ~A()
   {
      Console.WriteLine( "~A()" );
   }
}
```

```
class B : A
{
   public B()
   {
      Console.WriteLine( "B()" );
   }
   ~B()
   {
      Console.WriteLine( "~B()" );
   }
}
```

```
B b = new B();
b = null;
GC.Collect();
GC.WaitForPendingFinalizers(); // ???
```

Be Careful Out There!

- The finalization process takes place after "ordinary" garbage collection

- If your class has only managed resources, you should use a destructor!
- Avoid destructors whenever possible
  - Costs time
  - Hard to debug
  - Prolongs object life and memory usage

- Cannot know exactly when finalization takes place...!

Agenda

TEKNOLOGISK INSTITUT

- Introducing Lifetime
- Enter Garbage Collection
- Class Destructors
- **The Disposable Pattern**

Two Approaches to Cleaning Up

- Solution 1: Implement a destructor with cleanup logic
- Solution 2: Implement an explicit **`Dispose()`** method and remember to invoke it!
- Both solutions have shortcomings…

- Best solution is to *combine* 1 + 2:
  - Try to remember to invoke **`Dispose()`** for deterministic cleanup
  - If you don't, the garbage collector will eventually clean it up

- This is the philosophy behind implementing **`IDisposable`**

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

Implementing IDisposable

TEKNOLOGISK INSTITUT

```
public class A : IDisposable
{
   private bool _disposed = false;
   public void Dispose()
   {
      CleanUp( true );
      GC.SuppressFinalize( this );
   }
   private void CleanUp(bool disposing)
   {
      if( _disposed == false )
      {
         if( disposing )
         {
            // Dispose managed here
         }
         // Clean up unmanaged here.
      }
      _disposed = true;
   }
```

```
   ~A()
   {
      CleanUp( false );
   }
}
```

Disposing Classes

- Many .NET Framework classes implement **`IDisposable`**
- You should always invoke **`Dispose()`** on objects if they implement **`IDisposable`**

- In order to make the built-in classes more "natural", there is often a **`Close()`** method which does the same as **`Dispose()`**
  - This of course makes it even more confusing... ☹

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

   // These method both closes!
   fs.Close();    // WTF???
   fs.Dispose();
}
```

The using Statement

- The **using** statement is a convenient shorthand to help you to remember to **Dispose()**

```
using( MyResourceWrapper rw = new MyResourceWrapper() )
{
   rw.DoStuff();
   ...
}
```

- **Dispose()** is always invoked at the end of the using block – even in the presence of exceptions!

- Strive to use **using** whenever possible instead of manually invoking Dispose()

Quiz: Object Lifetime – Right or Wrong?

TEKNOLOGISK INSTITUT

✗
```
class A
{
   ...
   ~A( int i )
   {
      Console.WriteLine( i );
   }

   public void DoStuff() { ... }
}
```

✓
```
class B : IDisposable
{
   public void Dispose() { ... }

   public DoStuff() { ... }
}
```

✗
```
A a = new A();
~A();
```

✓
```
A a = new A();
a.DoStuff();
a = null;
```

✗
```
A a = new A();
a.DoStuff();
a.Dispose();
```

✗
```
B b = new B();
b.DoStuff();
b.Dispose();
```

✓
```
using( B b = new B() )
{
   b.DoStuff();
}
```

Summary

TEKNOLOGISK INSTITUT

- Introducing Lifetime
- Enter Garbage Collection
- Class Destructors
- The Disposable Pattern

Question

- You are creating a class referencing unmanaged resources. Also it maintains references to managed resources on other objects. You must ensure that the class can be explicitly cleaned up. Which three actions should you perform?

  (Each correct answer presents part of the solution. Choose three.)

TEKNOLOGISK
INSTITUT

Question (Cont'd)

a) Make the class derive from the System.GC.CleanUp class.

[✓ b) Make the class implement the IDisposable interface.]

[✓ c) Create a Dispose method which cleans up unmanaged resources and calls methods to release the referenced managed resources.]

d) Create a Dispose method that calls System.GC.Collect to force garbage collection.

[✓ e) Create a class destructor that releases the unmanaged resources.]

f) Create a class destructor that calls methods to release the referenced managed resources.

TEKNOLOGISK
INSTITUT