



C# Programming in Depth

Prof. Dr. Bertrand Meyer

March 2007 - May 2007

Lecture 4: Garbage Collection & Exception Handling

Lisa (Ling) Liu

Overview



- Scope and lifetime
- Garbage collection mechanism
- Exception handling

Scope and lifetime



- Scope of a variable is portion of program text within which it is declared
 - Need not be contiguous
 - In C#, is static: independent of data
- Lifetime or extent of storage is portion of program execution during which it exists
 - Always contiguous
 - Generally dynamic: dependent on data
- Class of lifetime
 - Static: entire duration of program
 - Local or automatic: duration of call or block execution (local variable)
 - Dynamic: From time of allocation statement (new) to deallocation, if any.

Object lifetime in C#



- Memory allocation for an object should be made using the "new" keyword
- Objects are allocated onto the managed heap, where they are automatically deallocated by the runtime at "some time in the future"
- Garbage collection is automated in C#

Rule: Allocate an object onto the managed heap using the **new** keyword and forget about it

Object creation



- When a call to `new` is made, it creates a CIL "newobj" instruction to the code module

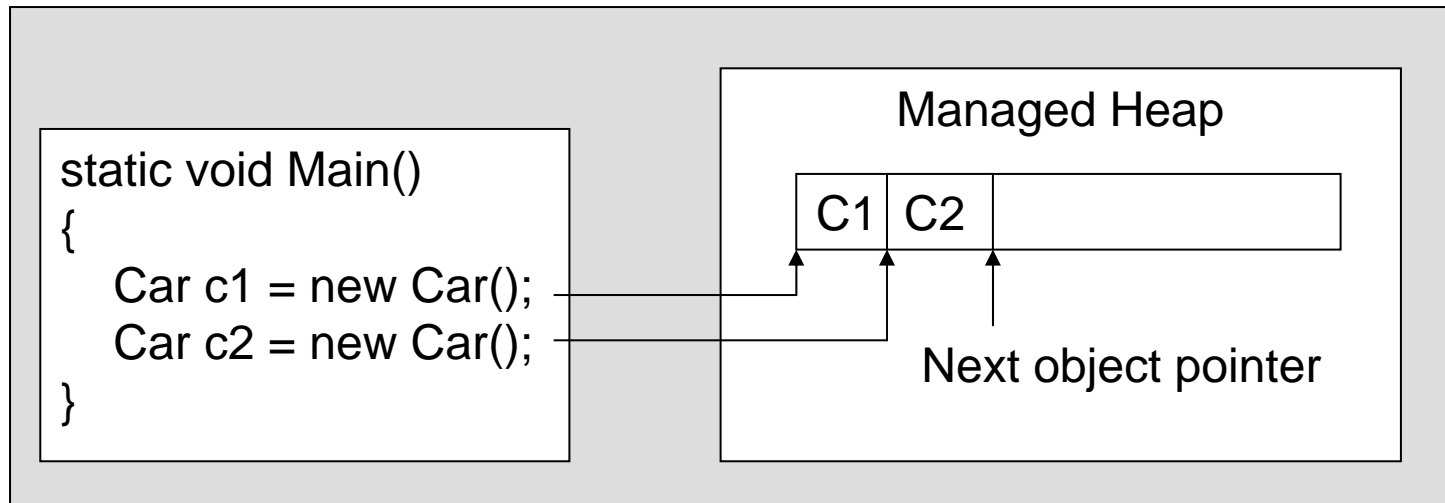
```
public static int Main (string[] args)
{
    Car c = new Car("Viper", 200, 100);
}
```

IL_000c: newobj instance void CilNew.Car::.ctor (string, int32, int32)

Tasks taken by CIL newobj instruction



- Calculate the total amount of memory required for the object
- Examine the managed heap to ensure enough room for the object
- Return the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap



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

Garbage collection steps

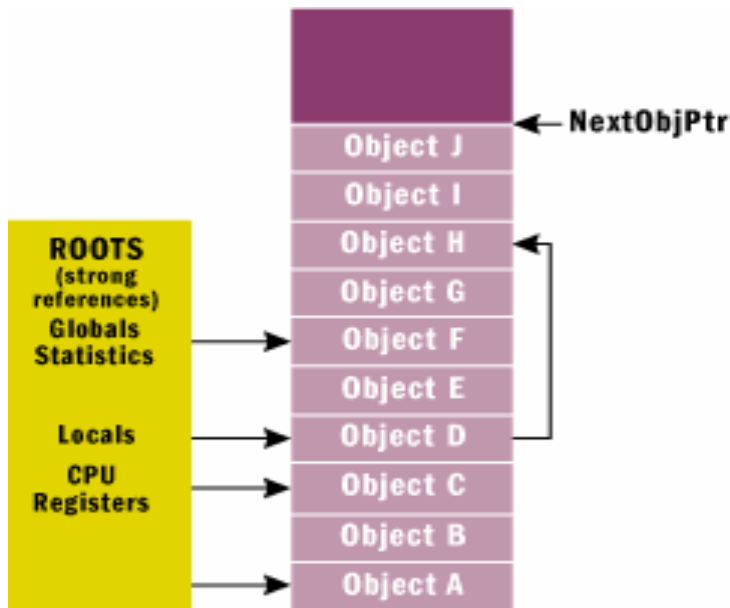


1. The garbage collector searches for managed objects that are referenced in managed code — mark
 2. The garbage collector attempts to finalize objects that are unreachable
 3. The garbage collector frees objects that are unmarked and reclaims their memory
- } sweep

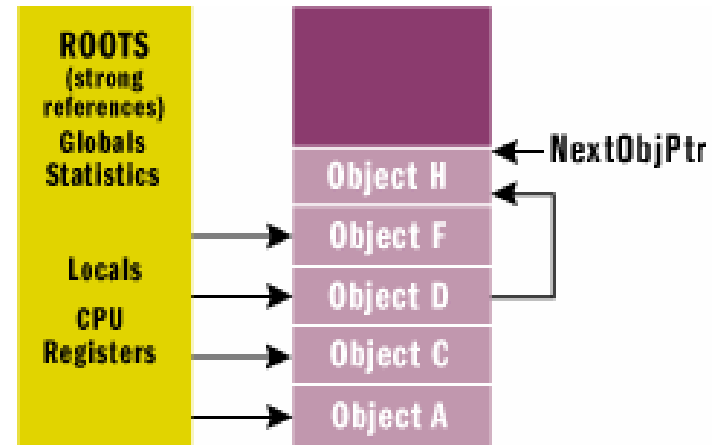
How to decide an object is unreachable



- Object graph represents each reachable object on the heap



Allocated objects on the heap



Managed heap after collection

Optimize the decision process



- Object generations
 - Each object on the heap is assigned to a specific "generation" (0 ~ 2)
 - Generation 1: newly allocated objects
 - Generation 2: objects that survived a garbage collection
 - Generation 3: objects that survived more than one garbage collection
 - The garbage collector first investigate generation 0 objects. If marking and sweeping these objects can result in required amount of free memory, any surviving objects' generation are promoted by 1

The System.GC type



- Provide a set of static method for interacting with garbage collection
- Use this type when you are creating types that make use of unmanaged resource



Building finalizable objects

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

- Override Finalize() to perform any necessary memory cleanup for your type
- A call to Finalize () occurs:
 - natural garbage collection
 - GC.Collect()
 - Application domain is unloaded from the memory

Override System.Object.Finalize()



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

        ...
    }
}
```

When to override `System.Object.Finalize()`



Rule: The only reason to override `Finalize()` is if your C# class is making use of unmanaged resources via `PInvoke` or complex COM interoperability tasks (typically via the `System.Runtime.InteropServices.Marshal` type).

It is illegal to override `Finalize()` on structure types.

Building Disposable Objects



- Another approach to handle an object's cleanup.
- Implement the `IDisposable` interface
- Object users should manually call `Dispose()` before allowing the object reference to drop out of scope
- Structures and classes can both support `IDisposable` (unlike overriding `Finalize()`)

```
// Implementing IDisposable
public class MyResourceWrapper : IDisposable
{
    // The object user should call this method
    // when they finished with the object.
    public void Dispose()
    {
        // Clean up unmanaged resources here.
        // Dispose other contained disposable objects.

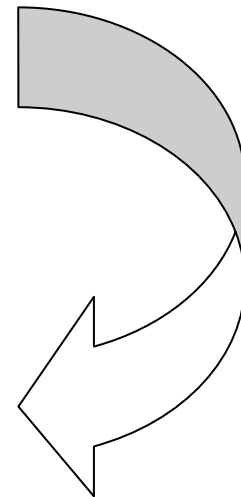
    }
}
```

Rule: Always call `Dispose()` on any object you directly create if the object supports `IDisposable`.

Reusing the C# using Keyword

```
MyResourceWrapper rw = new MyResourceWrapper();  
try  
{  
    // Use the member of rw  
}  
finally  
{  
    rw.Dispose()  
}
```

```
// Dispose() is called automatically when the  
// using scope exits.  
using (MyResourceWrapper rw2 = new  
MyResourceWrapper())  
{  
    // Use rw2 object.  
}
```



Questions



- What's the difference between Finalize and Dispose?
- What's the difference between Dispose and Using?

.Net exception handling



- When your application encounters an exceptional circumstance, such as a division by zero or low memory warning, an exception is generated.
- Once an exception occurs, the flow of control immediately jumps to an associated exception handler, if one is present.
- If no exception handler for a given exception is present, the program stops executing with an error message.
- Actions that may result in an exception are executed with the `try` keyword.
- An exception handler is a block of code that is executed when an exception occurs. In `C#`, the `catch` keyword is used to define an exception handler.
- Exceptions can be explicitly generated by a program using the `throw` keyword.
- Exception objects contain detailed information about the error, including the state of the call stack and a text description of the error.
- Code in a finally block is executed even if an exception is thrown, thus allowing a program to release resources.

C# exception handling structure



```
try
{
    // Code to try here.
}
catch (System.Exception ex)
{
    // Code to handle exception here.
}
finally
{
    // Code to execute after try (and possibly catch) here.
}
```

System.Exception base class



```
public class Exception: ISerializable, _Exception
{
    public virtual IDictionary Data { get; }
    protected Exception (SerializationInfo info, StreamingContext context);
    public Exception (string message);
    public Exception ()
    public virtual Exception GetBaseException ();
    public virtual void GetObjectData (SerializationInfo info, StreamingContext
context);
    public System.Type GetType ();
    protected int HRESULT {get; set;}
    public virtual string HelpLink {get; set;}
    public System.Exception InnerException {get;}
    public virtual string Message {get;}
    public virtual string Source {get; set;}
    public virtual string StackTrace {get;}
    public MethodBase TargetSite {get;}
    public override string ToString ();
}
```

```
static void Main (string[] args)
{
    string[] strFiles;
    try
    {
        strFiles = Directory.GetFiles (args[0]);
    }
    catch (Exception e)
    {
        Console.WriteLine ("Method: {0}", e.TargetSite);
        Console.WriteLine ("Message: {0}", e.Message);
        Console.WriteLine ("Source: {0}", e.Source);
        Console.WriteLine ("StackTrace: {0}", e.StackTrace);
    }
    Console.WriteLine ("Remaining part");
}
```

Unhandled exceptions





Exception categories

- System-level exceptions (`System.SystemException`)
 - Exceptions thrown by the CLR and are regarded as nonrecoverable, fatal errors (derive from `System.SystemException`)
- Application-level exceptions (`System.ApplicationException`)
 - Exceptions thrown by your application (derive from `System.ApplicationException`)

Building custom Exception



- A strongly typed exception that represents the unique details of the problem regarding the type
- Best practice
 - Derive from Exception / ApplicationException
 - Is marked with the [Serializable] attribute
 - Defines a default constructor
 - Defines a constructor that sets the inherited Message property
 - Defines a constructor to handle "inner exceptions"
 - Defines a constructor to handle the serialization of your type

Questions?

