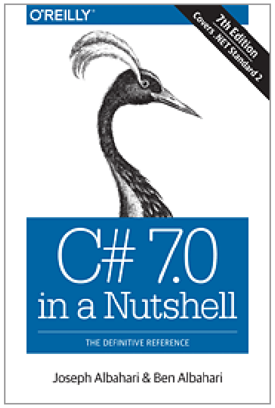


Beginning Level **C#**

Resource Management

Yong Zhang, Ph.D

Weber State University

2019

Garbage Collection and Disposal

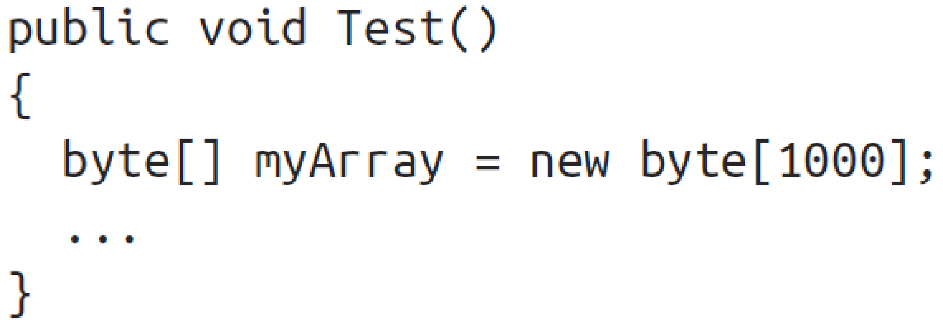
- Manual resource management: **disposal**
  - Open/close files
  - Locks
  - Operating system handles
  - Unmanaged objects
- Automatic resource management: **garbage collection**
  - Managed objects

Garbage Collection in .NET Framework

- Every type/variable requires memory in your application
- Garbage Collection deals with reference types
- Reference types remain in scope until no longer needed
- Garbage Collection does not require manual release of resources
    - It checks for objects on the heap no longer referenced;
    - First start, it assumes all objects are garbage;
    - Doesn’t release at startup though;
    - It walks the root of all objects, checking references;
    - Generates a graph of objects that can be collected.

Garbage Collection in .NET Framework

- Garbage collection does not happen immediately after an object is orphaned.
- Garbage collection happens periodically, although not to a fixed schedule.



```
public void Test()
{
  byte[] myArray = new byte[1000];
  ...
}
```

Performance

- GC exacts a performance hit
- Does not run all the time
- GC is invoked automatically when heap is full
- Should never receive OutOfMemoryExcpetion with GC

How to Use It

- Let the framework handle the GC execution
- You can call GC.Collect(), but it is nonderministic
- GC will run when system resources permit
- Supports a finalization phase for your objects
- Use Finalize() method in your objects for cleanup

Finalizer / Destructors

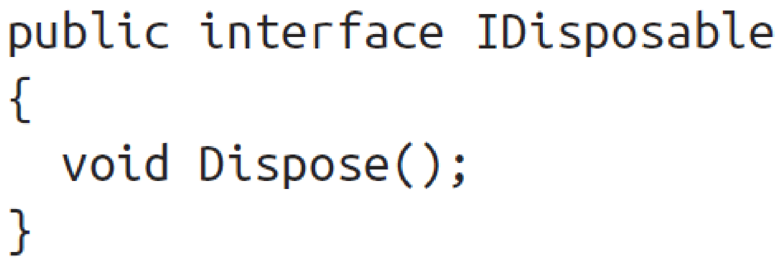
- A destructor is to destroy an object
- Cannot be used with structures, only classes
- Can only have one destructor
- Cannot be overloaded or inherited
- Take no modifiers or parameters
- Example: 04-Resource Management /DestructorExample

How Destructor Works

- Cannot be called, invoked automatically by the GC
- If present, destructors will implicitly call the **Finalize()** method
- Not commonly required
- Finalizers are possible because garbage collection works in distinct phases.
  - First, the GC identifies the unused objects ripe for deletion. Those without finalizers are deleted right away.
  - Those with pending (unrun) finalizers are kept alive (for now) and are put onto a special queue. At that point, garbage collection is complete, and your program continues executing.
  - The *finalizer thread* then kicks in and starts running in parallel to your program, picking objects off that special queue and running their finalization methods.

Introducing Idisposable, Dispose, and Close

- The .NET Framework defines a special interface for types requiring a tear-down method:

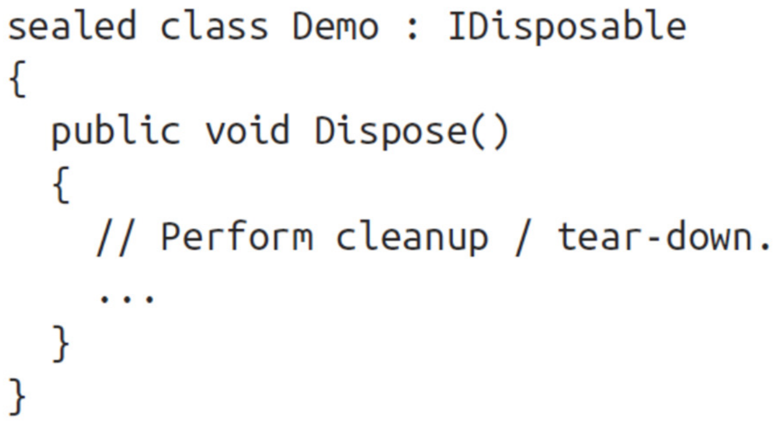


```
public interface IDisposable
{
  void Dispose();
}
```

- IDisposable is an interface
- Use it to release unmanaged resources

Introducing Idisposable, Dispose, and Close

- In simple scenarios, writing your own disposable type is just a matter of implementing IDisposable and writing the Dispose method:



```
sealed class Demo : IDisposable
{
  public void Dispose()
  {
    // Perform cleanup / tear-down.
    ...
  }
}
```

Introducing Idisposable, Dispose, and Close

- Once disposed, an object is beyond redemption. It cannot be reactivated, and calling its methods or properties (other than Dispose) throws an ObjectDisposedException.
- Calling an object's Dispose method repeatedly causes no error.
- If disposable object *x* "owns" disposable object *y*, *x*'s Dispose method automatically calls *y*'s Dispose method—unless instructed otherwise.

Introducing Idisposable, Dispose, and Close

- Some types define a method called **Close** in addition to Dispose.
- The Framework is not completely consistent on the semantics of a Close method, although in nearly all cases it’s either:
  - Functionally identical to Dispose
  - A functional *subset* of Dispose
- An example of the latter is IDbConnection:
  - a Closed connection can be re-Opened;
  - a Disposed connection cannot.
- Another example is a Windows Form activated with ShowDialog:
  - Close hides it;
  - Dispose releases its resources.

Introducing Idisposable, Dispose, and Close

- When to Dispose: **if in doubt, dispose**
- Objects wrapping an unmanaged resource handle will nearly always require disposal, in order to free the handle.
  - Examples include Windows Forms controls, file or network streams, network sockets, GDI+ pens, brushes, and bitmaps.
- Conversely, if a type is disposable, it will often (but not always) reference an unmanaged handle, directly or indirectly.