

Module 27: "Dispose"



Agenda

- ▶ Introductory Example: Handling Open Files
- ▶ Challenges
- ▶ Pattern: Dispose
- ▶ Background: Garbage Collection and Finalizers
- ▶ Implementing the Very Basic Dispose Pattern
- ▶ Implementing the Basic Dispose Pattern
- ▶ Implementing the Dispose Pattern
- ▶ C# Syntax Support
- ▶ Overview of Dispose



Introductory Example: Handling Open Files

```
class FileWriter
{
    private readonly FileStream _fs;

    public FileWriter() => _fs = File.Create(@"FileWriter.txt");
    public void Log()
    {
        string s = $"{DateTime.Now}{Environment.NewLine}";
        _fs.Write(Encoding.ASCII.GetBytes(s), 0, s.Length);
    }
}
```

```
FileWriter fileWriter = new FileWriter();
fileWriter.Log();
fileWriter.Log();
fileWriter.Log();

fileWriter = null; // FileWriter is no longer needed?
```



Challenges

- ▶ How and when will the FileStream be closed?
- ▶ Will it ever be?
- ▶ How do we signal that the object is "no longer used"?



Pattern: Dispose

- ▶ *Provide a deterministic resource management mechanism for your objects, i.e. make it "disposable" if it contains a resource that needs manual handling.*
- ▶ Outline
 - Provide a method with clean-up logic
 - Implement the **IDisposable** interface on your class
 - Implement a finalizer on your class, if needed
- ▶ Origin: Folklore



Background: 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



Background: 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



Background: Defining Destructors

- ▶ Put cleanup logic in the destructor
 - As 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 FileWriter
{
    private readonly FileStream _fs;
    ...
    public FileWriter() => _fs = File.Create(@"FileWriter.txt");
    ~FileWriter() => _fs.Close();
}
```



Be Careful Out There!

- ▶ The finalization process takes place after “ordinary” garbage collection
- ▶ Avoid destructors whenever possible
 - Costs time
 - Hard to debug
 - Prolongs object life and memory usage
- ▶ Cannot know exactly when finalization takes place...!



Two Approaches to Resource Management

- ▶ 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**



IDisposable

- ▶ .NET has **IDisposable** interface built-in for implementing Dispose Pattern

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



Implementing the Very Basic Dispose Pattern

- ▶ Create a **Dispose()** method cleaning up managed resources

```
class FileWriter : IDisposable
{
    private readonly FileStream _fs;

    public FileWriter() => _fs = File.Create(@"FileWriter.txt");

    public void Dispose() => _fs?.Dispose();


    ...
}
```



Implementing the Basic Dispose Pattern

```
private bool _isDisposed = false;
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
protected virtual void Dispose(bool disposing)
{
    if (_isDisposed == false)
    {
        if( disposing )
        {
            _fs?.Dispose();
        }
    }
    _isDisposed = true;
}
```

Technically
not needed,
but...



Implementing the Basic Dispose Pattern

- ▶ But... Also remember disposed check in all public methods

```
class FileWriter : IDisposable
{
    ...
    public void Log()
    {
        if( _isDisposed )
        {
            throw new ObjectDisposedException(nameof(FileWriter));
        }

        string s = ...;
    }
}
```

Implementing the Dispose Pattern

```
~FileWriter() => Dispose(false);
```

```
protected virtual void Dispose(bool disposing)
```

```
{
```

```
    if (_isDisposed == false)
```

```
    {
```

```
        if (disposing)
```

```
        {
```

```
            // Dispose managed resources here
```

```
            _fs?.Dispose();
```

```
        }
```

```
        // Clean up unmanaged resources here
```

```
        ...
```

```
    }
```

```
    _isDisposed = true;
```

```
}
```

This is the
only new
parts

Remembering to Dispose

- ▶ Many .NET Framework classes implement **IDisposable**
- ▶ Do try to dispose objects if they implement **IDisposable**
- ▶ If your class holds on to **IDisposable** resources, you should implement **IDisposable** on your class as well..!
- ▶ But how do we make sure to remember to invoke **Dispose()** ?
 - Even in the presence of exceptions etc.?



C# Syntax Support

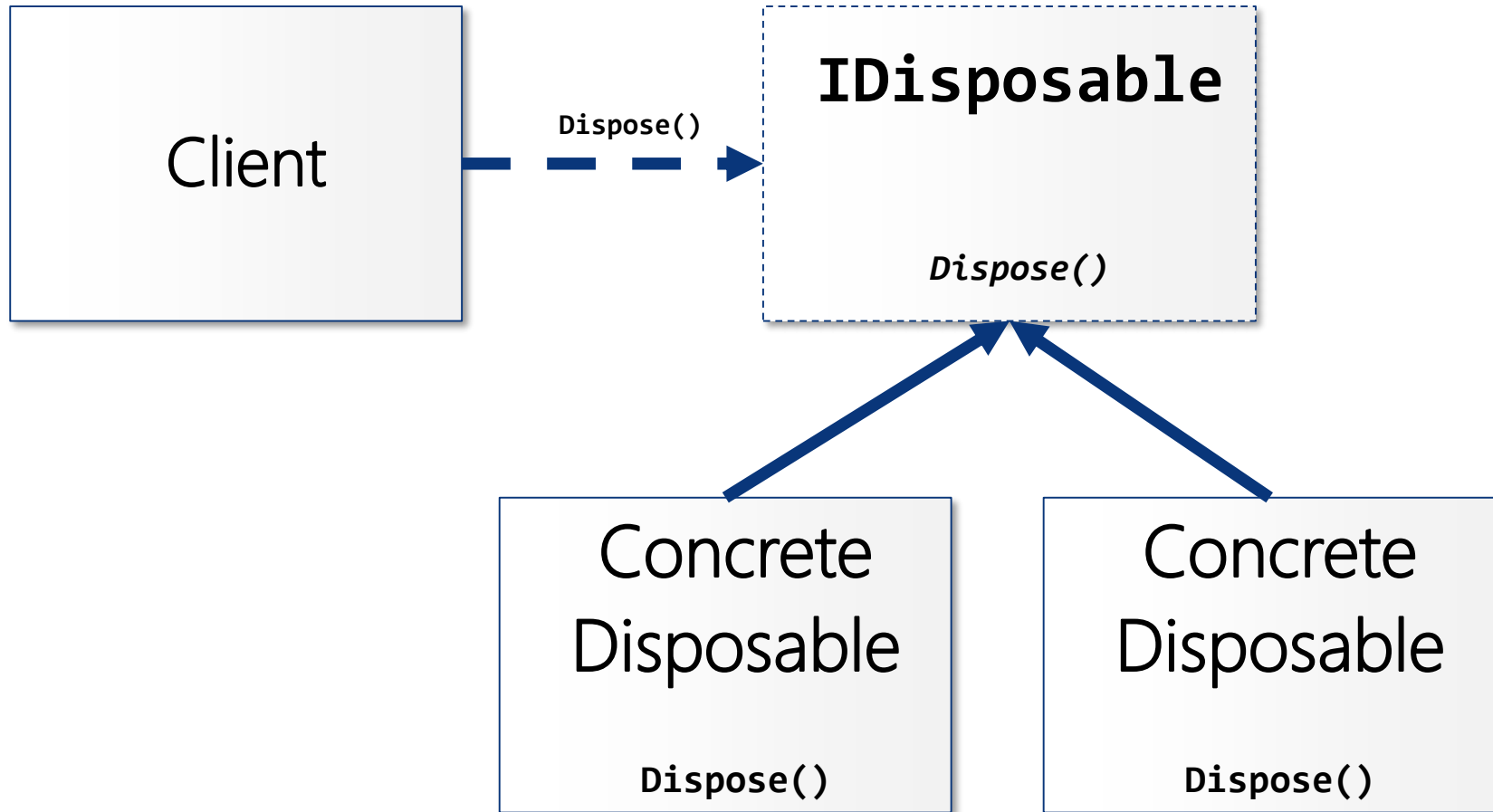
- ▶ The **using** statement is a convenient shorthand for calling **Dispose()**

```
using (FileWriter fileWriter = new FileWriter())  
{  
    fileWriter.Log();  
    fileWriter.Log();  
    ...  
} // <-- Invokes Dispose()  
  
// FileWriter is no longer needed
```

- ▶ **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()**



Overview of Dispose Pattern



Overview of Dispose Pattern

▶ **IDisposable**

- Interface pre-built into .NET
- Provides a **Dispose()** method for deterministic clean-up logic

▶ Concrete Disposable

- Implements **IDisposable** interface and supplies appropriate deterministic clean-up logic for disposable resources it is holding
 - Distinguishes between managed and unmanaged resources

▶ Client

- Invokes **Dispose()** on Concrete Disposable to perform deterministic clean-up
- Preferably uses **using** when using resources whenever possible





WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Hasselvangel 243

8355 Solbjerg

Denmark