

# Dispose Pattern in C#

# Agenda

- Destructor and Finalizer in C#
- IDisposable and RAII
- Dispose Pattern for Managed and Unmanaged Resources
- Objects with Critical Finalization
- Simplified Dispose Pattern
- Recommended Links

# Destructor and Finalizer in C#

# Destructor in C#

- Destructor in C# language created with tilde (“~”) is syntax sugar for Finalize method, to which it is converted on compilation stage of the application
- That is why it is correct to say that destructor and finalizer is the same in C#
- While “destructor” term has special meaning in programming, it is better to say that C# does not have destructor at all, we will use term “finalizer” instead

# Finalizer Problem

- Time of finalizer call is not defined in .NET, that is why finalizers do not guarantee:
  1. Time of resource release
  2. Fact of resource release

**IDisposable and RAII**

# Interface IDisposable


- Provides a mechanism for releasing unmanaged resources.

## ▲ Syntax

```
C# C++ F# VB
[ComVisibleAttribute(true)]
public interface IDisposable
```

The IDisposable type exposes the following members.

## ▲ Methods

	Name	Description
	Dispose	Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.

# RAII Idiom

- RAII – Resource Acquisition Is Initialization
- RAII means that resource should be allocated in constructor and released in destructor
- OO languages with direct resource management completely corresponds to RAII



# Keyword using

- Keyword “using” not completely implements RAII

```
// Opening file inside using block
using (FileStream file = File.OpenRead("foo.txt"))
{
    // Leaving method on condition
    if (someCondition) return;
    // File closes automatically
}
```

```
// What if file opens outside using block?
FileStream file2 = File.OpenRead("foo.txt");
```

# Method Dispose

- Dispose method differs from destructor in that way that it not destroys the object but destroys the resource
- Danger Consequences of dispose call: object is not destroyed but resource is not available and any further method call or access to property is potentially dangerous

# Dispose Pattern for Managed and Unmanaged Resources

# Dispose Pattern

- Taking into account all previously mentioned, we have to implement special dispose pattern in .NET to ensure that resources are released in proper way

# Managed and Unmanaged Resources

- Unmanaged resources – IntPtr, socket descriptors, any OS objects obtained with WinAPI etc.
- If **unmanaged** resource is wrapped into class with RAII it becomes **managed** resource
- Any of two types of resources implies different approaches to work with them

# Sample Resource Wrapper

```
class NativeResourceWrapper : IDisposable
{
    // IntPtr - unmanaged resource descriptor
    private IntPtr nativeResourceHandle;
    public NativeResourceWrapper()
    {
        //Acquiring unmanaged resource
        nativeResourceHandle = AcquireNativeResource();
    }
    public void Dispose()
    {
        // Releasing unmanaged resource
        ReleaseNativeResource(nativeResourceHandle);
    }
    // Finalizer will be explained later
    ~NativeResourceWrapper() {...}
}
```

# Main Idea of Dispose Pattern

The main idea of Dispose Pattern is:

1. Place all logic of resource release into separate method;
2. Call it from Dispose method;
3. Also call it from finalizer;
4. Add special flag that helps to distinguish who exactly (Dispose or Finalizer) called the method.

# 1. Interface Implementation

- Class that has both managed and unmanaged resources implements **IDisposable** interface

```
class Boo : IDisposable { ... }
```



# 2. Method Dispose(bool disposing)

- Class contains method **Dispose(bool disposing)** that does all job to release resources;
- disposing parameter tells if method is called from **Dispose** method or from **Finalize**. This method should be protected virtual for non-sealed classes and private for sealed classes

```
// For not-sealed classes  
protected virtual void Dispose(bool disposing) {}
```

```
// For sealed classes  
private void Dispose(bool disposing) {}
```

# 3. Method Dispose()

- Dispose method implementation: first we call **Dispose(true)**, then we may call **GC.SuppressFinalize()** method that suppresses finalizer call:

```
public void Dispose()  
{  
    Dispose(true /*called by user directly*/);  
    GC.SuppressFinalize(this);  
}
```

# Notes to GC.SuppressFinalize() Call

- **GC.SuppressFinalize()** should be called **after** `Dispose(true)` but **not before** because if method **Dispose(true)** fails with exception the execution of finalizer should not be cancelled and it will give another chance to free resources
- **GC.SuppressFinalize()** should be called for classes that do not have finalizers because finalizers may be created for child classes. The only exception is sealed classes.

# 4. Parameter “disposing”

- Method **Dispose(bool disposing)** has two parts:
  1. If this method called from **Dispose** (**disposing** parameter is **true**) we should release both managed and unmanaged resources;
  2. If this method is called from finalizer (that is possible under normal circumstances only during garbage collection process when **disposing** parameter is **false**), we release only unmanaged resources.

```
void Dispose(bool disposing)
{
    if (disposing)
    {
        // Releasing managed resources only
    }

    // Releasing unmanaged resources
}
```

# 5. Finalizer

- [OPTIONAL] Class may have finalizer and call **Dispose(bool disposing)** from it passing **false** as parameter.

```
~Boo()  
{  
    Dispose(false /*not called by user directly*/);  
}
```

- Also we should take into account that finalizer may be called even for partially constructed classes, if constructor for such class raises an exception. That is why resource releasing code should handle situation when resources are not allocated yet

# 6. Field “disposed”

- The good practice is to create special Boolean field **disposed** which indicates that object’s resources are released.
- Disposable objects should allow **any** number of **Dispose()** method calls and generate an exception when any public member of the object is accessed after first call to the method (when **dispose flag** is set to true).

```
void Dispose(bool disposing)
{
    if (disposed)
        return; // Resources are already released
    // Releasing resources
    disposed = true;
}
```

```
public void SomeMethod()
{
    if (disposed)
        throw new ObjectDisposedException();
}
```

# Objects with Critical Finalization

# 7. Object with Critical Finalization

Class may be inherited from **CriticalFinalizerObject**:

- Finalizer for such classes compiled with JIT-compiler immediately when the instance is constructed (apart to default on demand compilation). This allows finalizer to complete successfully even if the memory is full
- CLR does not guarantee order of finalizer calls that makes impossible to access other objects from finalizer that contain unmanaged resources. But CLR guarantees that finalizers for usual objects will be called before childs of **CriticalFinalizerObject**. This allows from “usual” objects to access field `SafeHandle` that is guaranteed to be released later
- Finalizers for such classes will be called even in case of abnormal termination of application domain.

```
// Use with caution  
class Foo : CriticalFinalizerObject {}
```



# Simplified Dispose Pattern

# Simplifying Dispose Pattern

- Most difficulties with Dispose pattern implementation based on assumption that same class (or class hierarchy) may contain managed and unmanaged resources at the same time
- But Single Responsibility Principle (SRP) suggest us that we do not mix resources of different kinds
- RAll idiom suggests a solution: **if you have unmanaged resource, do not use it directly, wrap it into managed wrapper and work with it**

# Simplified Dispose Pattern

- Used only for managed resources

```
class SomethingWithManagedResources : IDisposable
{
    public void Dispose()
    {
        // No Dispose(true) и and no calls to GC.SuppressFinalize()
        DisposeManagedResources();
    }

    // No parameters, this method releases unmanaged resources only
    protected virtual void DisposeManagedResources() {}
}
```

# Recommended Links

# Recommended Links

- Dispose pattern <http://habrahabr.ru/post/129283/>
- IDisposable: What Your Mother Never Told You About Resource Deallocation  
<http://www.codeproject.com/Articles/29534/IDisposable-What-Your-Mother-Never-Told-You-About>
- Implementing Finalize and Dispose for cleaning unmanaged resources  
<http://msdn.microsoft.com/ru-ru/library/b1yfk5e.aspx>
- Does C# have destructor? <http://habrahabr.ru/post/122639/>