CSC271

Managing Non-Memory Resources

Download class materials from university.xamarin.com

Mark Smith | mark.smith@xamarin.com

Microsoft

Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Cleaning up non-memory resources

Xamarin University

Tasks

1. Resource Types
2. Using `IDisposable`
3. Implementing `IDisposable`
4. The Dispose pattern
5. Thinking about versioning

Resource Types

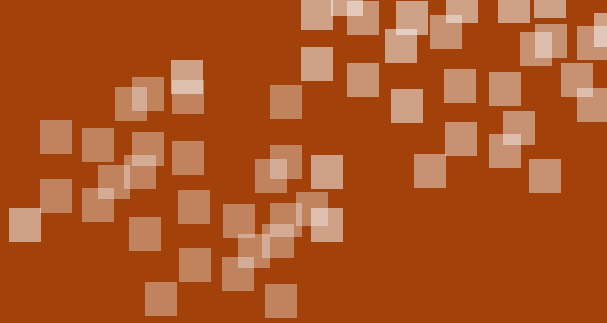
- Applications can work with a variety of **resources**
  - Memory
  - Files and local databases
  - Network connections
  - …
- … but only memory is automatically managed by the runtime

GC is unpredictable

- Our applications should not rely on the GC to release non-memory resources which need more deterministic cleanup

```
public string ReadJsonData(string filename)
{
    return new StreamReader (filename).ReadToEnd ();
}
```

When will this file be closed?



Demonstration

When does a file get closed?

Xamarin University

Introducing: IDisposable

- **`IDisposable`** interface indicates an object's desire for immediate cleanup when the owner is finished with the object

```
public class StreamReader :  IDisposable { ... }
```

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

Using IDisposable

- Your code is responsible for calling **`Dispose`** when you are done with the object – the runtime does not know anything about **`IDisposable`**!

```
public string ReadJsonData(string filename)
{
    StreamReader reader = new StreamReader (filename);
    try {
        return reader.ReadToEnd ();
    }
    finally {
        reader.Dispose ();
    }
}
```

Using IDisposable

- C# **`using`** block adds call to **`Dispose`** when scope is exited by emitting a **`try`**/**`finally`** into the generated code

```
public string ReadJsonData(string filename)
{
    using (StreamReader reader = new StreamReader (filename))
    {
        return reader.ReadToEnd ();
    } // Dispose is called here
}
```

Look for IDisposable on types

- `Dispose` is often used to provide a way to **identify** and **invoke** native resource cleanup requirements

```
[Register ("NSObject", true)]
public class NSObject : INSObjectProtocol,
    IEquatable<NSObject>, INativeObject,
    IDisposable
{
    // Base class for most iOS things...
}
```

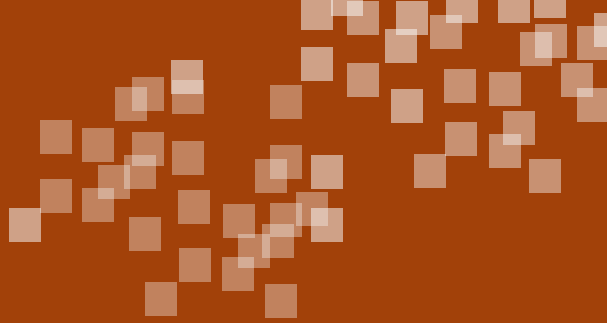
Example: disposing a native resource

```
NSObject token;

public override void ViewDidAppear (bool animated) {
    base.ViewDidAppear (animated);
    token = NSNotificationCenter.DefaultCenter.AddObserver (...);
}

public override void ViewDidDisappear (bool animated) {
    base.ViewDidDisappear (animated);
    token.Dispose();
}
```

Disposing the notification token *unregisters our delegate handler*

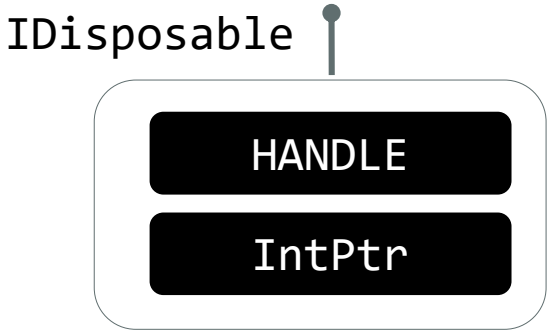


Demonstration

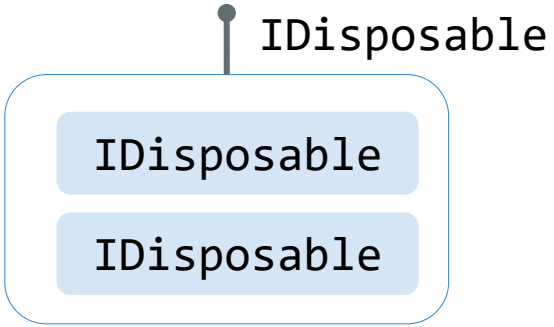
Using IDisposable

Implementing IDisposable

- Custom classes should *always* implement **`IDisposable`** if they:



Create + own native (OS) resources



Hold references to objects which implement **`IDisposable`**

Implementing IDisposable

- **`Dispose`** method should always completely release the held resource
  - Invoke the native API to free handle
  - Call the inner **`Dispose`** method of held references

```
public class DataLoader : IDisposable
{
    StreamReader reader;

    public DataLoader (string filename) {
        reader = new StreamReader (filename);
    }

    public void Dispose () {
        reader.Dispose ();
    }

    public string ReadJsonData () {
        reader.BaseStream.Position = 0;
        return reader.ReadToEnd ();
    }
}
```

Rules for IDisposable

1. **`Dispose`** should **never** throw an exception *unless* a critical error occurs attempting to free the resource

2. Once **`Dispose`** has been called, the object should be considered *invalid*. Objects can throw **`ObjectDisposedException`** if a client continues to use the instance

3. **`Dispose`** should be **idempotent** – multiple calls should just return

Implementing IDisposable (#2)

```
public class DataLoader : IDisposable
{
   StreamReader reader;
   bool disposed;
   ...

   public void Dispose () {
      if (!disposed) {
         reader?.Dispose ();
         disposed = true;
      }
   }
   public string ReadJsonData () {
      if (disposed) throw new ObjectDisposedException("DataLoader");
      ...
   }
}
```

Close vs. Dispose

- Consider providing a **`Close`** method in addition to implementing **`IDisposable`** if close is standard terminology for the object type you are modeling
- In most cases, **`Close`** and **`Dispose`** will be identical and do the same thing
- However, reuse is not supported in the **`Dispose`** case, but can be supported in the case of **`Close`**

Thinking about inheritance

- **`IDisposable`** gets tricky when inheritance is involved – can use explicit interface inheritance in the derived class and pass onto the base

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

```
public class DerivedClass
     : BaseClass, IDisposable
{
   void IDisposable.Dispose() {
      base.Dispose ();
      ...
   }
}
```

```
BaseClass bc = new DerivedClass();
...
// Calls BaseClass.Dispose
bc.Dispose(); // PROBLEM!

// Calls DerivedClass.Dispose
IDisposable d = bc;
d.Dispose();  // OK!
```

The Dispose pattern

- Framework guidelines promote the use of a virtual **`Dispose`** implementation to simplify versioning and potential inheritance

```
public class DataLoader : IDisposable
{
    protected virtual void Dispose(bool isDisposing) {
        if (!disposed) {
            if (isDisposing) {
                reader.Dispose ();
                disposed = true;
            }
        }
    }
    public void Dispose () { Dispose (true); }
}
```

All cleanup is done in the virtual method

Interface forwards call to virtual method

Inheritance and the Dispose pattern

- Virtual method allows derived classes to participate in the disposal and ensures that the entire hierarchy sees the **`Dispose`** request

```
public class SecondDataLoader : DataLoader
{
    protected override void Dispose (bool isDisposing)
    {
        base.Dispose (isDisposing);
        if (isDisposing) {
            ...
        }
    }
}
```

Must call base implementation to pass down the chain

Thinking about versioning

- Adding an interface contract to a base class *is a breaking change*
- If you think your object will need `Dispose` in the future, and your class is not sealed, then consider implementing it now to save yourself some future pain

Problem with IDisposable

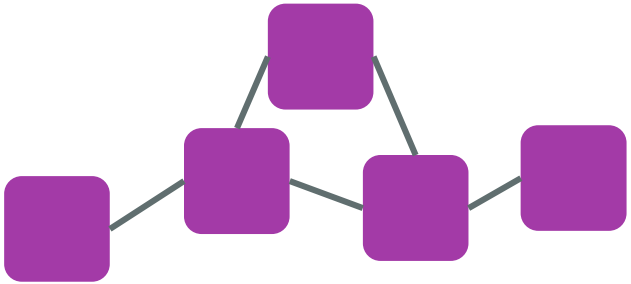
- `IDisposable` was created to solve a specific problem: **How to deterministically release expensive non-memory resources** ..

- But the pattern requires *your* code to call `Dispose` in the correct places

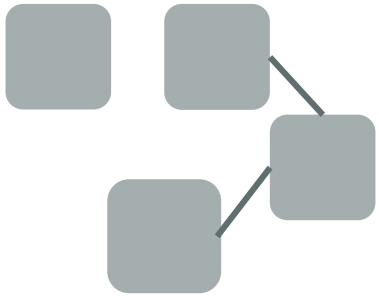
What if you don't??

What is a finalizer?

- Finalizers were created as a *last chance* opportunity for an object to release native resources (e.g. non-managed things) *just before the owner is collected by the GC*



Reachable ("live") objects

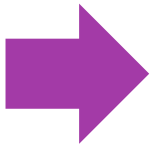


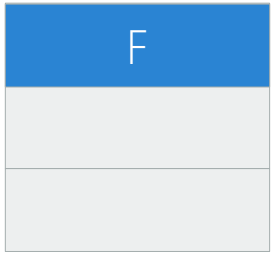
Unreachable ("dead") objects

The cost of a finalizer

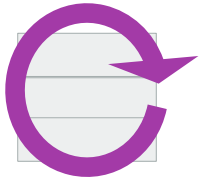
- Be very careful when declaring a finalizer – it adds significant expense to the allocation and destruction of your object

`new F()`
(**F** is **finalizable**)





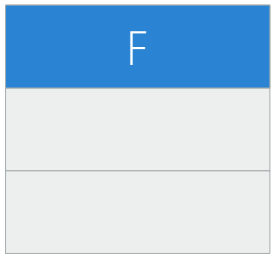
RegisteredForFinalization queue



Every GC must scan **RFF queue** looking for objects ready to finalize

The cost of a finalizer

- Once an object is not reachable by user code, but is in the registered for finalization queue, it is ready to finalize



RegisteredForFinalization queue

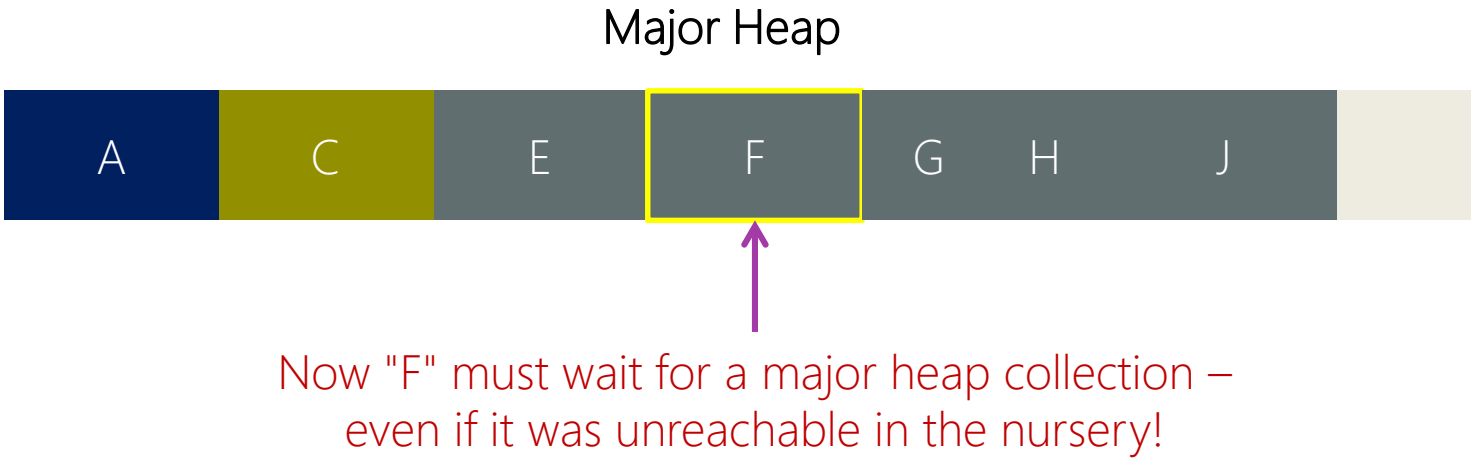
GC promotes the object and moves it to the **Ready to Finalize** queue

`F.Finalize()`

Dedicated *finalizer* thread wakes up and calls **`Finalize`** method on each object in queue

The cost of a finalizer

- Once the object has been finalized, it is then ready for the GC to collect and deallocate the memory … that means the object is *always* promoted!



Declaring a finalizer

- Finalizers are declared on managed objects through a language-specific syntax; in C# we use **`~ClassName`**

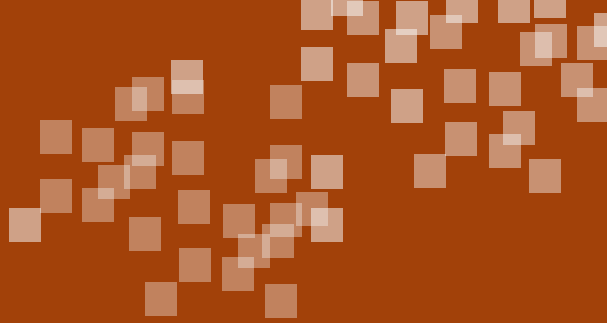
```
public class F
{
    ~F()
    {
        // Code executed by finalizer thread
        Debug.WriteLine ("Object F is being finalized!");
    }
}
```

Finalizers should only be called by the runtime

Finalizers according to .NET

- C#'s **~ClassName** syntax generates an override of **Object.Finalize**

```
public class F
{
    public F() { /* Generated by compiler */ }
    protected override void Finalize()
    {
        // Code executed by finalizer thread
        Debug.WriteLine ("Object F is being finalized!");
    }
}
```



Demonstration

The cost of finalizers

Back to the Dispose pattern

- If an object declares a finalizer, it should also be disposable … the Dispose pattern has support to deal with finalizers through the passed boolean

```
public class F : IDisposable
{
    protected virtual void Dispose(bool isDisposing)
    {
        // TODO: cleanup resources
    }

    ~F() { Dispose(false); }

    public void Dispose () { Dispose (true); }
}
```

Pass **false** to **Dispose** when call is coming from finalizer

Back to the Dispose pattern

- **`Dispose`** should cleanup *both* native and managed resources if called from **`IDisposable`**

```
protected virtual void Dispose(bool isDisposing) {
    if (!disposed) {
        if (isDisposing)
        {
            ... // Cleanup MANAGED resources here
        }
        disposed = true;
        // Cleanup NATIVE resources here
        ...
    }
}
```

Cleanup all the managed and native resources only when called from **`IDisposable`**

Back to the Dispose pattern

- `Dispose` should only release native resources if called from finalizer – managed references should not be used

```
protected virtual void Dispose(bool isDisposing) {
    if (!disposed) {
        if (isDisposing)
        {
            ... // Cleanup MANAGED resources here
        }
        disposed = true;
        // Cleanup NATIVE resources here
        ...
    }
}
```

Ignore managed references when called from the finalizer and only cleanup native resources

Finalization ordering

- Child references are not in a *stable* state during finalization

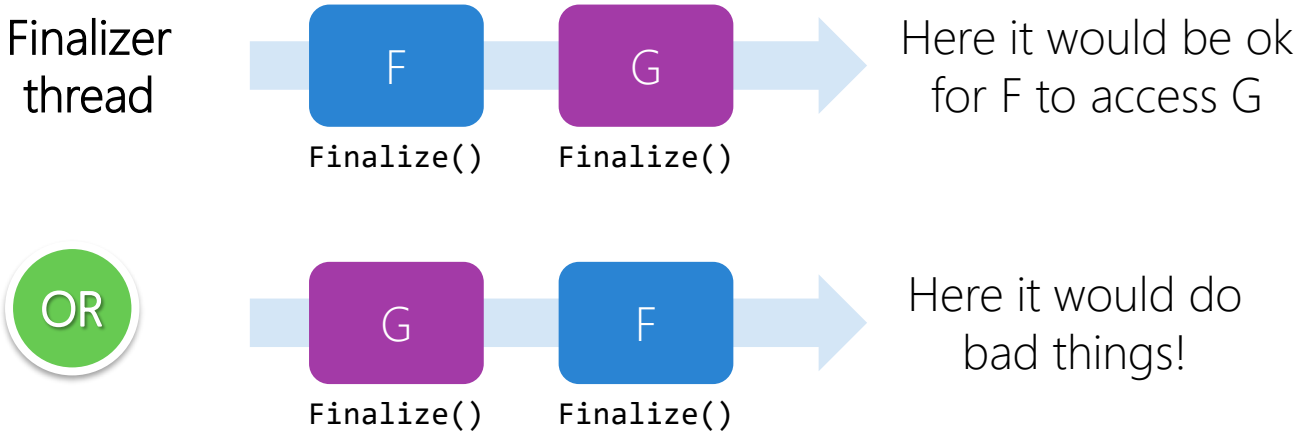
```
class F : IDisposable
{
   private G g = new G();
   ~F() {
      string val = g.ToString();
      ...
   }
}
```

```
class G : IDisposable
{
   ~G() { ... }
}
```

What happens if g has been collected?

Finalization ordering

- The finalizer thread calls each object in the queue one at a time, but the order is **unknown** and **unpredictable**



Suppressing finalization

- If an object is disposed by the owner, then it should always suppress finalization by calling **`GC.SuppressFinalize`**

```
protected virtual void Dispose(bool isDisposing) {
   if (!disposed) {
      if (isDisposing) {
         ... // Cleanup MANAGED resources
        GC.SupressFinalize(this);
      }
      disposed = true;
      ... // Cleanup NATIVE resources
  }
}
```

Diagnosing finalizers

- If your finalizer gets invoked, it means the client *did not* call **Dispose** and should be considered an error – can add **Debug.Assert** to catch the mistake in your **debug code**

```
protected virtual void Dispose(bool isDisposing) {
   if (!disposed) {
      // Oops .. Client forgot to dispose me!
      Debug.Assert (isDisposing, nameof(F) + " was not disposed!");
      ...
   }
}
```

The ugly side of finalization

- When the finalizer calls into your finalizable object, it executes *user code* which can do whatever it wants

```
~F() {
    while (true)
        Debug.WriteLine("BwaHaHaha")
}
```

Hocus Pocus – every finalizable object is now a memory leak

The ugly side of finalization

- When the finalizer calls into your finalizable object, it executes *user code* which can do whatever it wants

```
public class F
{
    public static F LiveObject;
    ~F() {
        LiveObject = this;
    }
}
```

Abracadabra – our object has been *resurrected* from the dead!

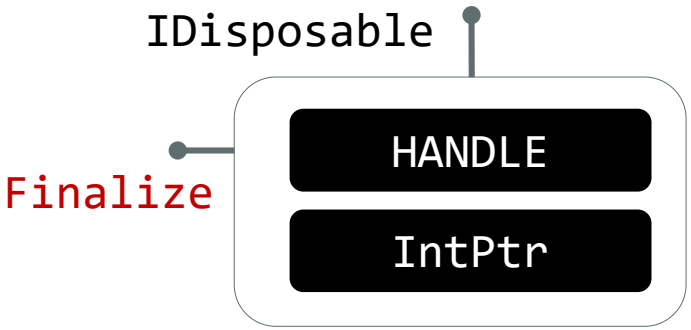
Resurrecting objects

- Legal to resurrect objects during finalization – but since the finalizer has already been called, it won't be finalized a second time without help

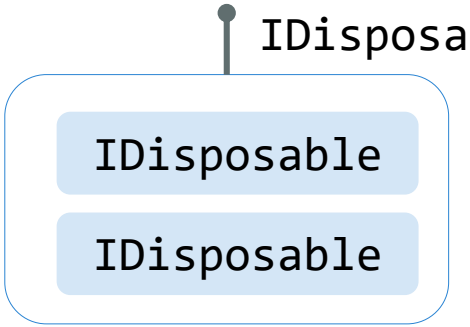
```
public class F
{
   public static List<F> pool = ...;
   public static F Create() { /* Get from pool or alloc */ }
   ~F() {
      pool.Add(this); // Put into pool – now it's alive!
      GC.ReRegisterForFinalize(this);
   }
}
```

Finalizers vs. IDisposable

- There is often confusion about when to use finalizers; here's the rule:



Create + own native (OS) resources



No finalizer necessary!

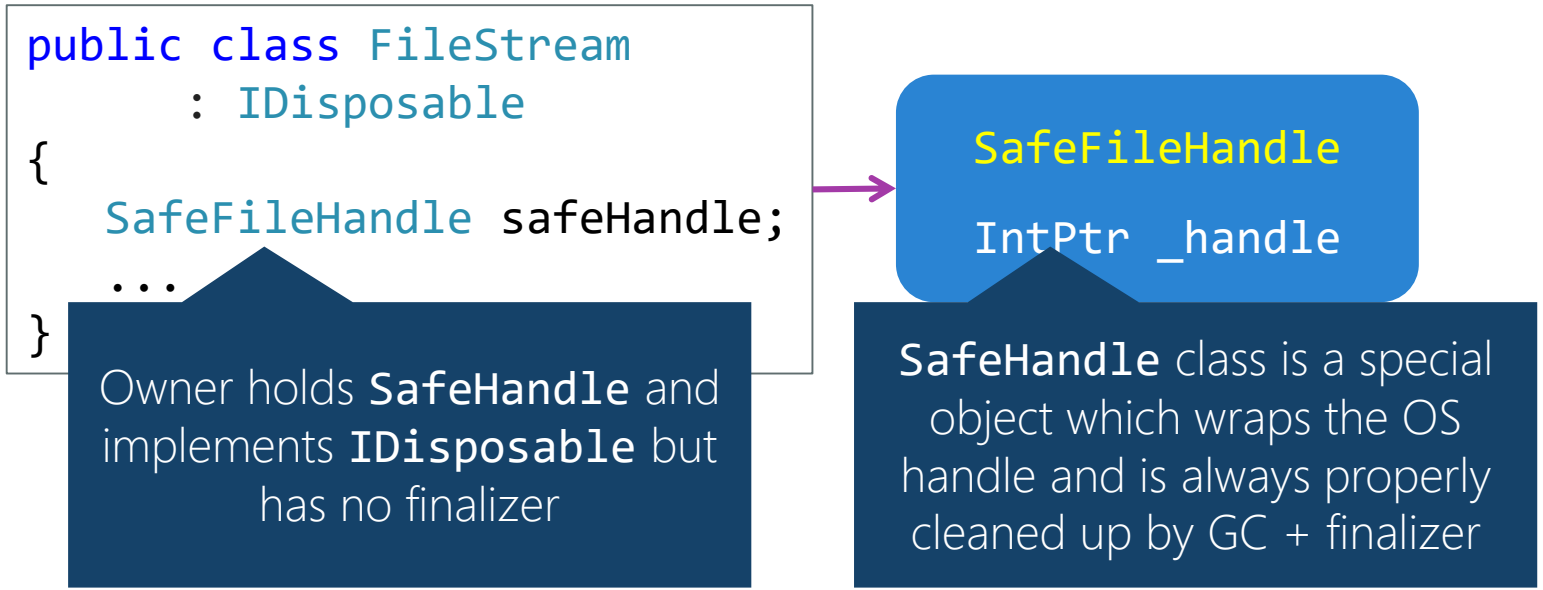
Hold references to objects which implement **IDisposable**

The purpose of finalizers

- Microsoft created finalizers specifically to cleanup native Win32 handles (e.g. HWND, HFILE, etc.)
- However finalizers are often written incorrectly or are misused and cause issues in our apps
- In .NET 2.0, Microsoft decided to fix this problem

What is a SafeHandle?

- **SafeHandle** was introduced to always release native OS handles without the use of a client/owner finalizer



Types of SafeHandles

- Several variations of **`SafeHandle`** in the framework – derive or use the one most appropriate to your native resource

Class	Purpose
`SafeHandleMinusOneIsInvalid`	Represents an OS handle where (-1) is considered invalid
`SafeHandleZeroOrMinusOneIsInvalid`	Represents an OS handle where (0) or (-1) are invalid values
`SafeFileHandle`	Represents a file-based handle
`SafeRegistryHandle`	Represents a REGISTRY handle (Win32)
`SafeWaitHandle`	Represents a sync **`WaitHandle`**

Example: creating a SafeHandle

- Derive from the best base class and then override **`ReleaseHandle`**

```
public sealed class MySafeHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    internal MySafeHandle () : base(true) {}
    public MySafeHandle (IntPtr existingHandle, bool ownsHandle) : base(ownsHandle)
    {
        SetHandle (existingHandle);
    }

    [System.Security.SecurityCritical]
    override protected bool ReleaseHandle ()
    {
        return UnsafeNativeMethods.CloseHandle (handle);
    }
}
```

Demonstration

Using SafeHandles

Summary

1. Other resource types
2. Using `IDisposable`
3. Implementing `IDisposable`
4. The Dispose pattern
5. Thinking about versioning

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

Microsoft