XAM370

Diagnosing Memory Management Issues

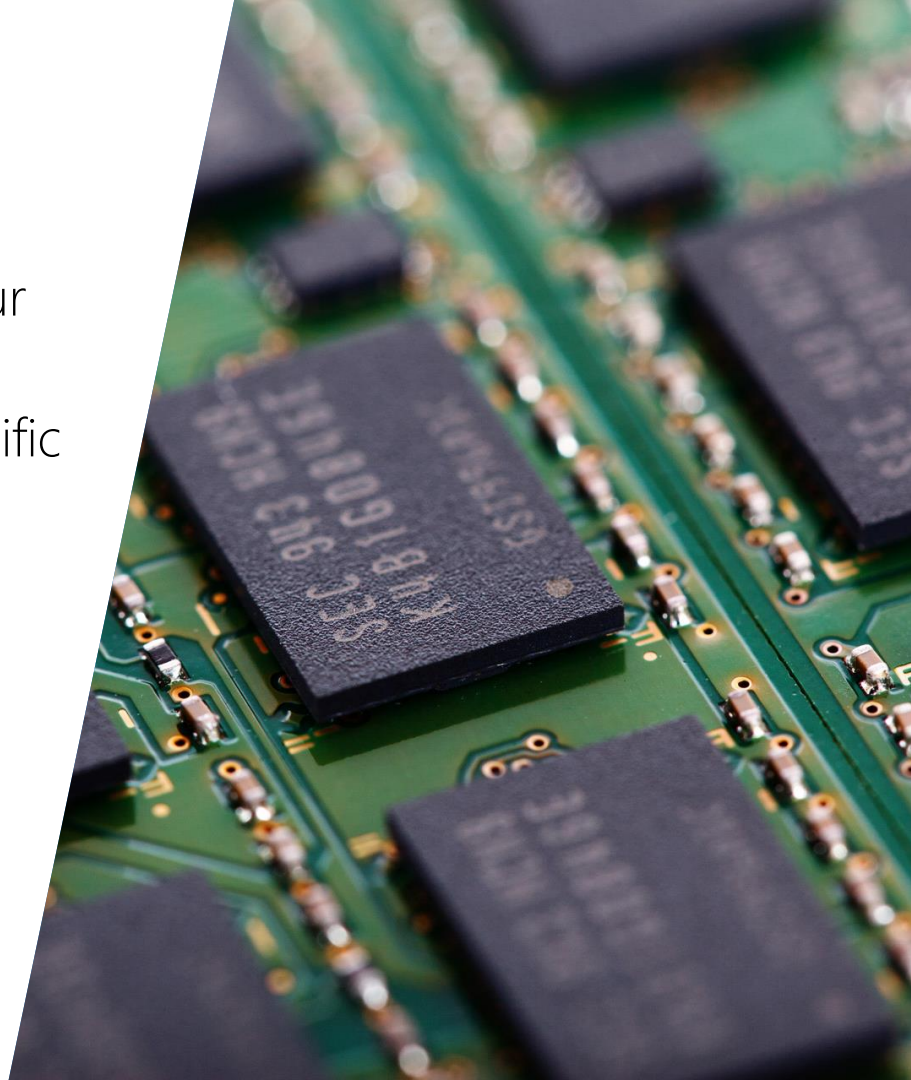
Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Identify and fix memory leaks in your code
2. Recognize and fix Xamarin.iOS specific memory problems
3. Recognize and fix Xamarin.Android specific memory problems



Identify and fix memory leaks in your code

Xamarin University

Tasks

1. Find memory leaks in your code
2. Fix common leaks in managed code

Reminder: Memory

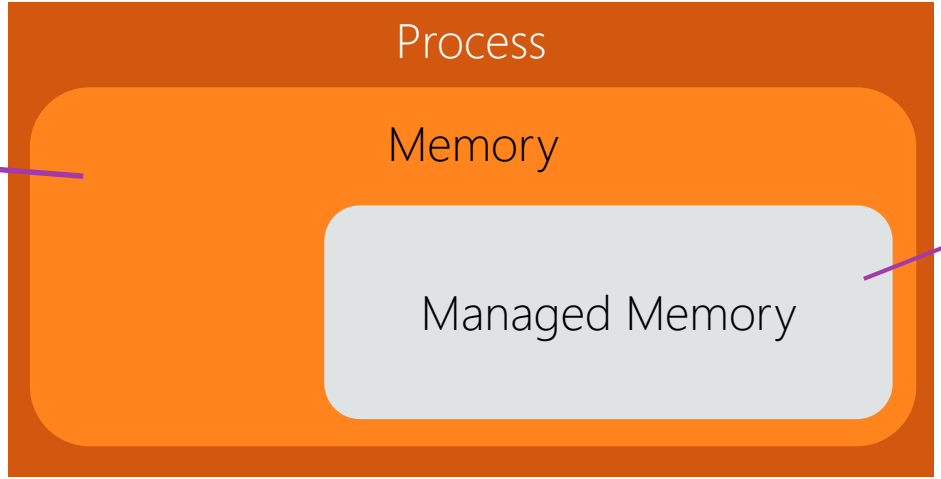
- .NET/Mono uses a **Garbage Collector** (referred to as **GC**) which periodically stops your program and frees the memory your app is no longer using

- This happens automatically as needed

Defining memory

- Your application works with **two types of memory**, both allocated from the same process space; apps must be concerned about both types

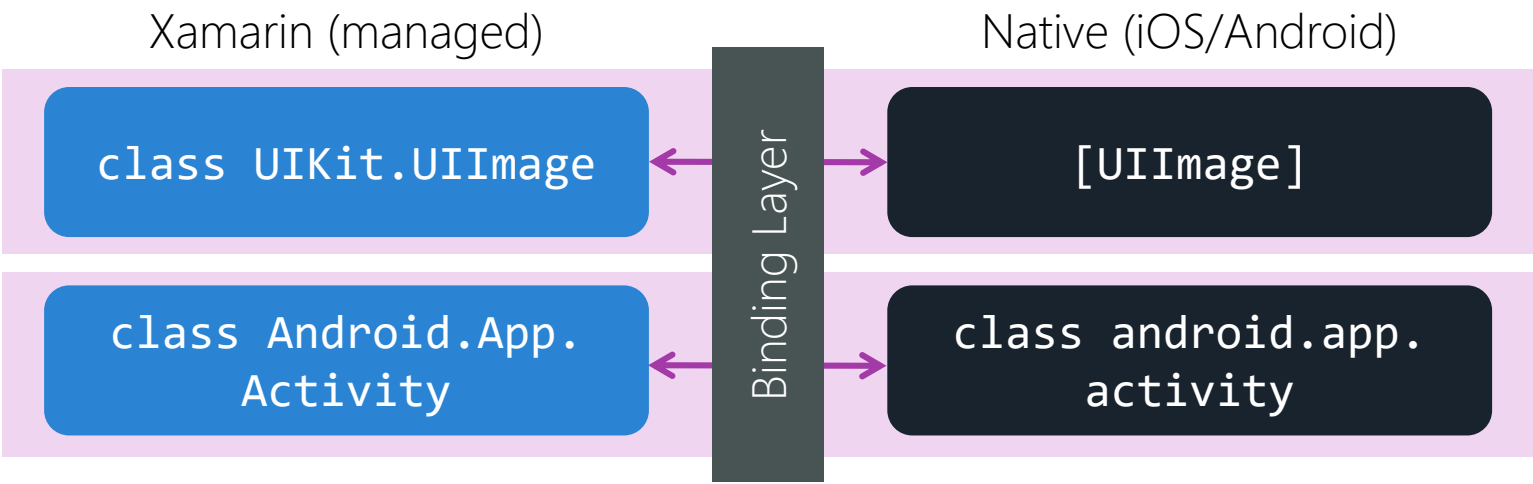
```
UIView
UIImage
MKMapView
...
Activity
Intent
TextView
...
```



C# and .NET objects are allocated here - this is all GC knows about

Objects in memory

- Some objects live in both the managed and the native world and must be treated properly to ensure that they are freed when the app is finished with them … **but not before**!

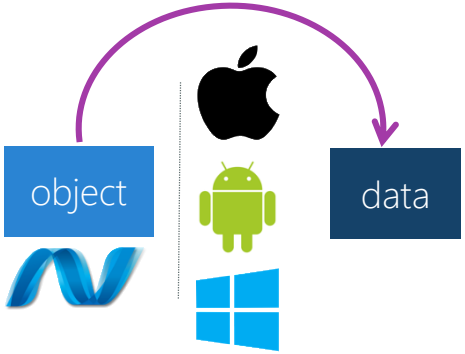


Memory Leaks in the GC world

- Memory leaks aren't as common when a GC is involved but can still happen in a few specific scenarios

Holding onto references

Thread Local values



Objects passed into the native platform ("pinned")

Monitoring memory allocations

- Several tools you can use to help check the memory allocations in your Xamarin application

Instruments

Android Profiler

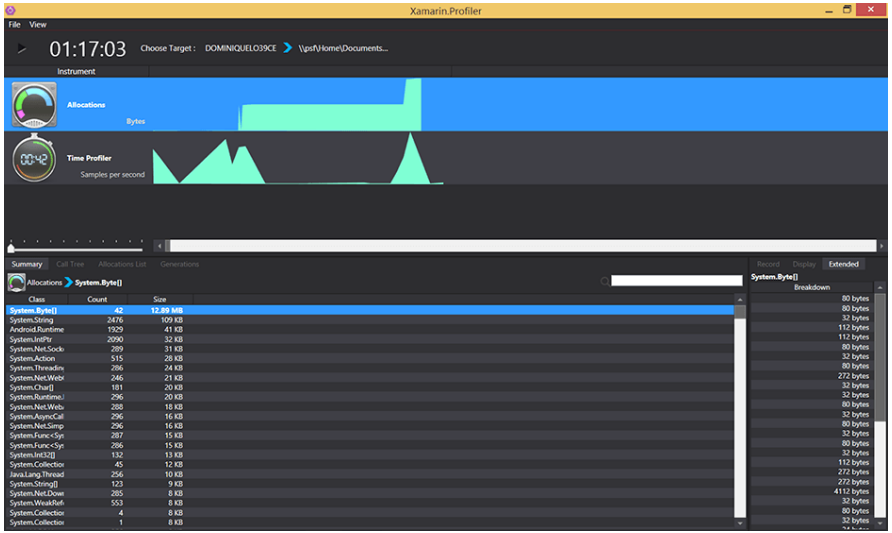
Xamarin Profiler

Visual Studio Profiling Tools

All of these tools allow you to look for strange patterns – e.g. an unexpectedly high number of some type of object, larger than normal working set, etc.

Xamarin Profiler

- Xamarin Profiler is a managed memory monitor available with an enterprise license that can help identify all the managed objects being allocated and retained in your app



Demonstration

Using the Xamarin Profiler to monitor allocations

Xamarin University

Monitoring memory growth

- **GC_MAJOR** reports current and previous memory usage for major heap and LOS; watch these values to identify potential leaks

```
GC_MAJOR: (user request) pause 6.53ms, total 6.54ms, bridge 0.00ms
major 1008K/16912K los 64K/0K
```

How much memory is currently in use by the major generation

How much memory *was* in use prior to this collect?

Checking for a leak

- Can use **`WeakReference`** as a diagnostic tool if you suspect an object is not being collected when you expect it to

```
DataLoader dl = new DataLoader ();
WeakReference wr = new WeakReference (dl);
...
dl = null;
...
GC.Collect ();
if (wr.IsAlive) {
    Debug.WriteLine ("DataLoader still alive");
}
```

What is a weak reference?

- A **`WeakReference`** is a reference to an object that *does not* protect the object from being collected by the GC; it allows your application to access the wrapped object as long as the GC has not collected it yet

I've got your back for now, but if there's any sign of trouble, I'm outta here!

`WeakReference`

Checking a weak reference?

```
public class MemoryLeakCheck<T> where T : class
{
    string filename; int lineNumber; WeakReference<T> reference;

    public MemoryLeakCheck(T theObject, [CallerFilePath]string filename = "",
                [CallerLineNumber]int lineNumber = 0) {
        this.reference = new WeakReference<T>(theObject);
        this.filename = filename;
        this.lineNumber = lineNumber;
    }

    public void Check() {
        GC.Collect();
        Debug.Assert(!reference.TryGetTarget(out T _),
           $"Object allocated at {filename}-{lineNumber} is still alive.");
    }
}
```

Another technique: Finalizers

- Can use a **finalizer** as a diagnostic to identify when an object is being collected

```
public class IThinkImAloneNow
{
    ...
#if DEBUG
    ~IThinkImAloneNow() {
        Console.WriteLine ("I'm about to be collected!");
    }
#endif
}
```

Remember that finalizers are expensive and should be avoided in production code

Watch out for hidden references!

- Delegates (and events) keep the subscriber object alive as long as the publisher is alive!

```
public override void ViewDidAppear (bool animated) {
   base.ViewDidAppear (animated);
   NSNotificationCenter.DefaultCenter.AddObserver(
       UIDevice.BatteryStateDidChangeNotification, OnChargingChanged);
}

private void OnChargingChanged (NSNotification notification) { ... }
```

Delegate references

- When you wire up a .NET delegate to an *instance method*, it must hold a **reference** to the owning instance

```
...DefaultCenter.AddObserver(..., OnChargingChanged);
```

Shorthand for:

```
...DefaultCenter.AddObserver(..., this.OnChargingChanged);
```

Which is shorthand for:

```
...DefaultCenter.AddObserver(...,
     new Action<NSNotification>(this.OnChargingChanged));
```

Delegate references

- When you wire up a .NET delegate to an *instance* method, it must hold a **reference** to the owning instance

```
...DefaultCenter.AddObserver(...,
      new Action<NSNotification>(this.OnChargingChanged));
```

```
public class Action<T> : Delegate
{
    public object Target { get; }
    public MethodInfo Method { get; }
    ...
}
```

Strong reference held by delegate

Why is this a problem?

- Delegates become a problem when the publisher of the event *outlives* the subscriber to the event

I live forever..

I'm supposed to live only while the user works with me

Fixing the delegate problem

- Follow .NET event guidelines: **always unsubscribe from delegates**

```
NSObject token;

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

public override void ViewDidDisappear (bool animated) {
    base.ViewDidDisappear (animated);
    NSNotificationCenter.DefaultCenter.RemoveObserver(token);
}
```

Individual Exercise

Finding and fixing delegate reference leaks

Xamarin University

Thread Locals

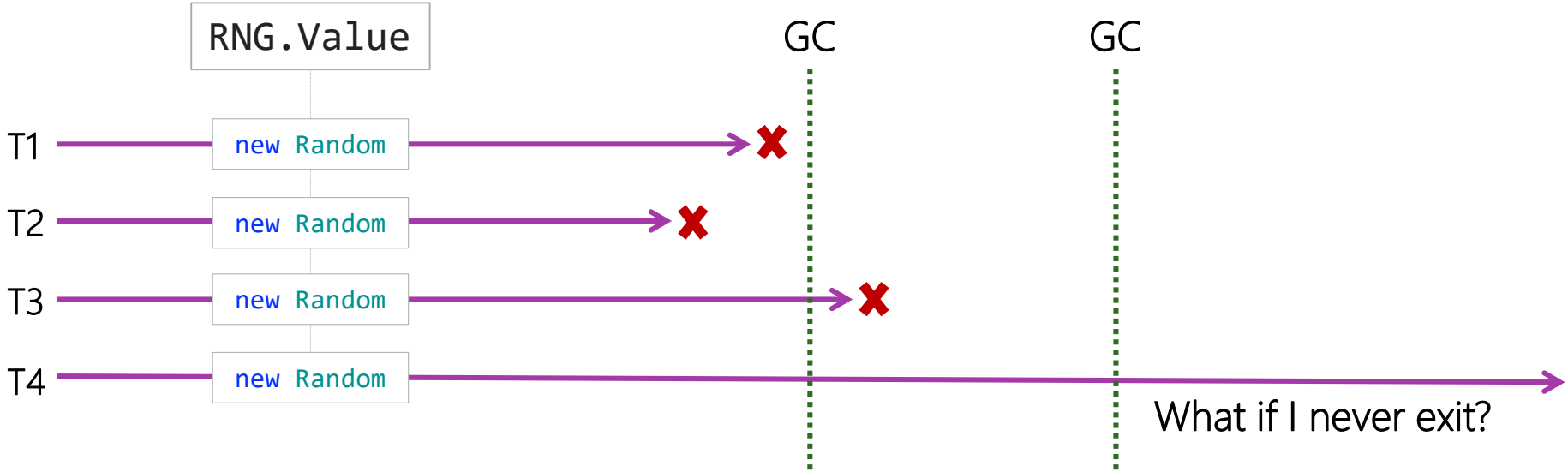
- **`ThreadLocal<T>`** is a convenient way to create strongly-typed, thread-scoped values which are unique per-thread

```
var RNG = new ThreadLocal<Random>(() =>
            new Random(new object().GetHashCode()));
Parallel.For (0, 1000, i => {
    // Need a truly random # for each thread
    int rndNumber = RNG.Value.Next (100);
    ...
});
```

Here we allocate a *unique* **`Random`** object for each thread – the passed delegate to **`ThreadLocal<T>`** is executed once (1st time) on each thread that accesses **`Value`**

Problem with Thread Locals

- Thread local values are stored in a *static list* and are not cleaned up by default until sometime after the thread exits



Cleaning up thread locals

- Should make sure to dispose **`ThreadLocal<T>`** when all your threads are done using it – this will release all the underlying values; be aware that it *does not* call **`Dispose`** on the values!

```
var RNG = new ThreadLocal<Random>(() =>
            new Random(new object().GetHashCode()));
Parallel.For (0, 1000, i => {
    // Need a truly random # for each thread
    int rndNumber = RNG.Value.Next (100);
    ...
});
RNG.Dispose();
```

Summary

1. Find memory leaks in your code
2. Fix common leaks in managed code

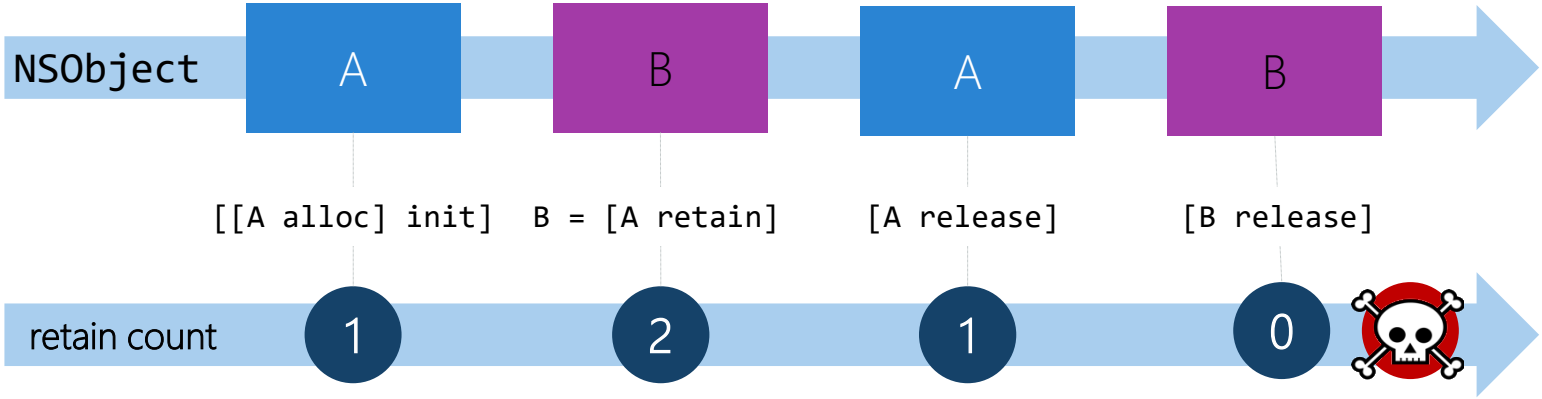
Xamarin.iOS

Tasks

- Identify strong reference cycles
- Dispose native resources
- Manage event handler lifecycles

Memory Management in iOS

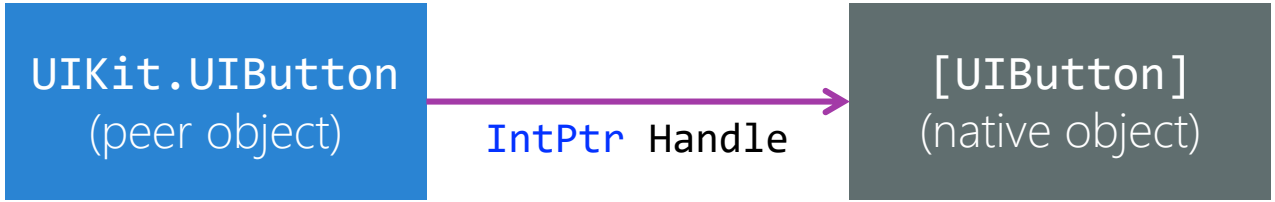
- iOS uses *reference counting* to manage memory (manual or automatic)



ARC (Automatic Reference Counting) makes this easier to work with in Obj-C and Swift, but memory leaks and dangling pointers are still major pain points in native iOS dev

Peer wrapper objects

- Xamarin.iOS creates a managed wrapper, called a **peer object**, for every native object accessed by the Xamarin.iOS runtime



Allocated by Xamarin.iOS and collected by the GC

Allocated by the iOS runtime and reference counted

Peers: retain count

- Peer object increment the retain count; it is released when the managed peer is **disposed** or **finalized**

```
var b = new UIButton(...)
```

```
[[UIButton alloc] init]
```

1

```
...
b = null;
```

GC Collect

```
[b release]
```

0

or

```
b.Dispose();
```

```
[b release]
```

0

Peer object types

- Xamarin.iOS supports *two* types of peer objects:

Framework Peers

User Peers

What is a Framework Peer?

- **Framework peers** are built-in, stateless types that wrap known iOS objects

Xamarin.iOS.dll

```
UIViewController
UIView
UIButton
CNContact
CGPDFDocument
WKWebView
MKMapView
...
```

What is a Framework Peer?

- Peers always call the native object to get or set the state

```
UIButton button = new UIButton();
button.SetTitle("Click Me", UIControlState.Normal);
if (button.CurrentTitle == "Click Me") { ... }
```

C#


```
[button setTitle:@"Click Me" forState:UIControlStateNormal]
button.currentTitle
```

Framework Peers: creation

- Framework peers are created to represent a native object when your code first accesses the object (e.g. when it is created, or accessed through a property)

```
UIButton button = (UIButton) View.Subviews[0];
```

Runtime will construct a new wrapper to represent the button if one did not exist yet

Framework Peers + GC

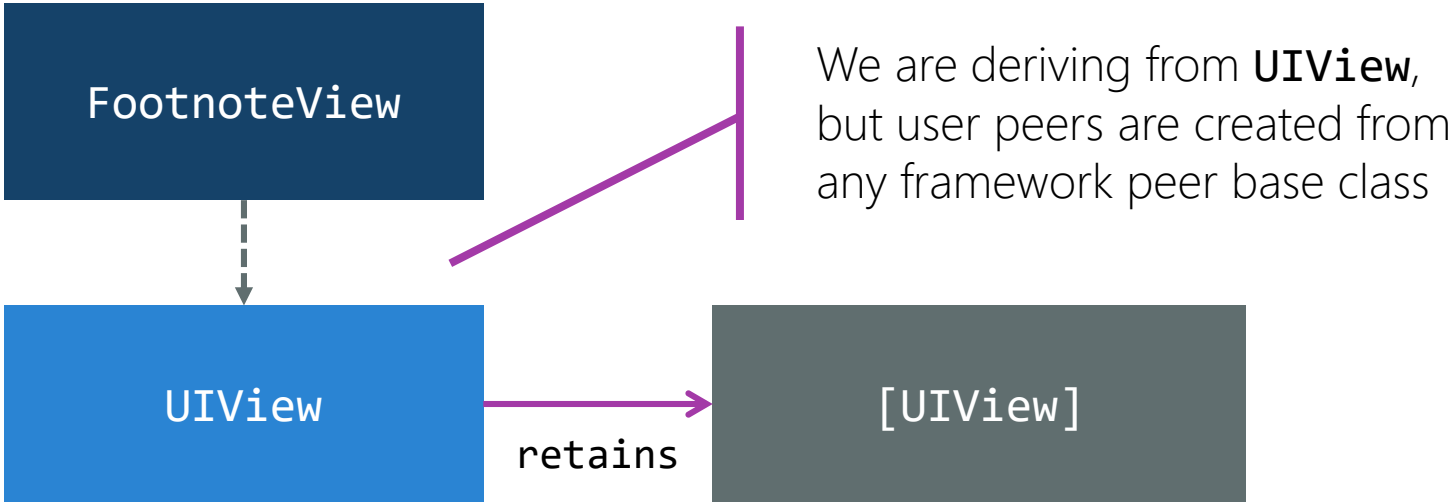
- Framework peers can be **collected** when not referenced by managed code; runtime will re-create a new wrapper if necessary

```
partial void OnShowDetails(UIButton sender)
{
    ... // Use button
}
```

Passed peer instance may, or may not be the same instance seen previously – however it will *always* refer to the same native object handle

What is a User Peer?

- User peers (sometimes called derived objects) are *custom* managed types which derive from a built-in iOS wrapper



How are user peers different?

- User peers can hold **managed state** – e.g. things which are *not* part of the native control state

```
public class FootnoteView : UIView
{
    public int FootnoteId { get; set; }
    public string FootnoteText { get; set; }
    ...
}
```

These fields are part of the *managed object only*, iOS knows nothing about them

User peers + GC

- Having managed state changes how the runtime must treat the object

```
public override void ViewDidLoad()
{
    base.ViewDidLoad();
    this.Add(new FootnoteView {FootnoteId = 1,
                               FootnoteText = "☹ nobody reads me" });
}
```

We are not holding onto the managed object here – it is kept alive because it has been added into the view hierarchy

User peers: staying alive!

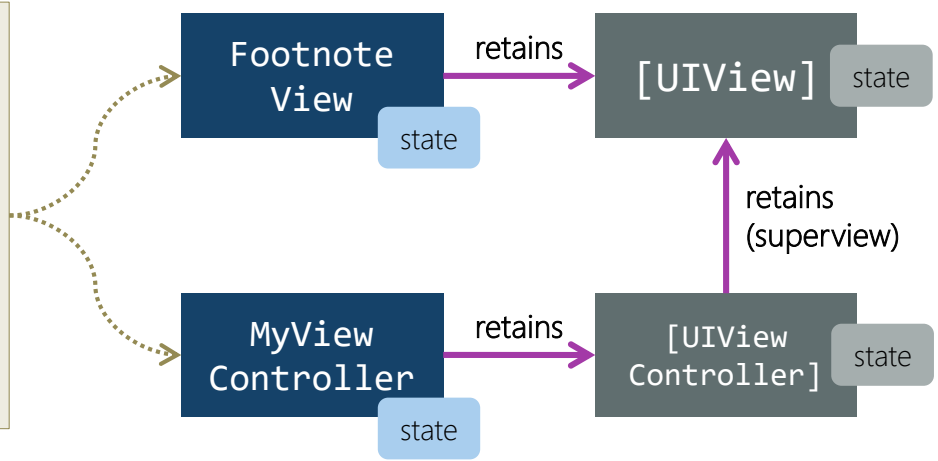
- Xamarin.iOS **keeps user peers alive** even if there are no references in your managed code; this ensures that **state is preserved**

```
public class MyViewController : UIViewController
{
   public override void ViewDidDisappear() {
      // Get the footnote we displayed
      FootnoteView fn = (FootnoteView) View.Subviews[0];
      int id = fn.FootnoteId;
      ... // state is there because it's the same wrapper
   }
   ...
}
```

User Peer: preserving state

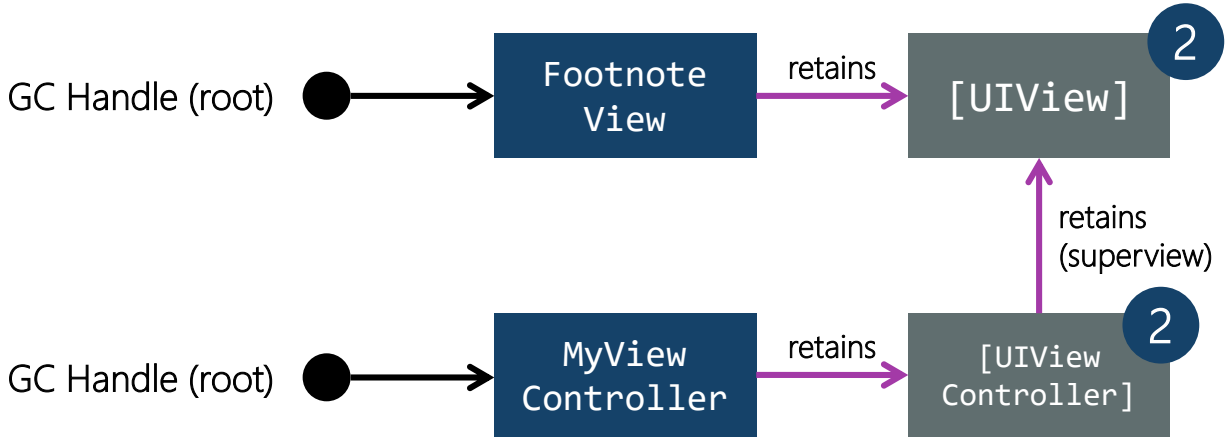
- Xamarin.iOS ensures state is preserved by *rooting* any user peer that has no managed references; this keeps it from being collected

Problem:
No user code has a reference to either of the peer objects; normally this would allow the GC to collect them …



User Peer: preserving state

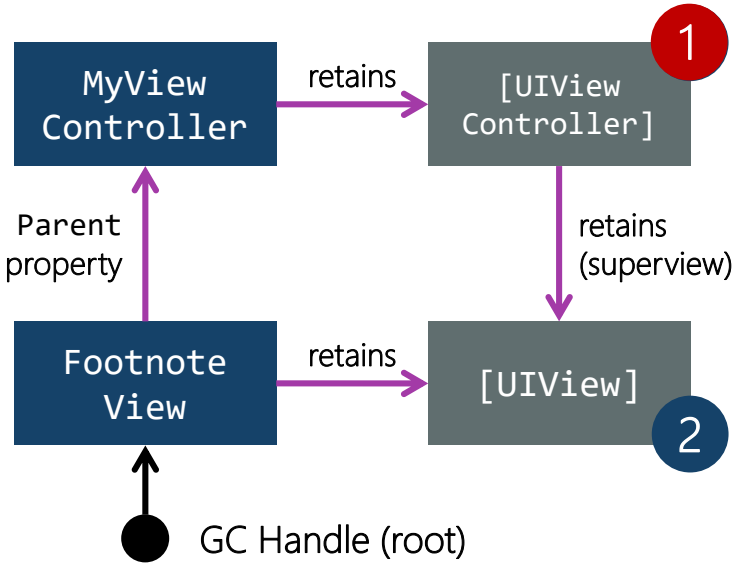
- Xamarin.iOS ensures state is preserved by *rooting* any user peer that has no managed references; this keeps it from being collected



User Peers + reference cycles

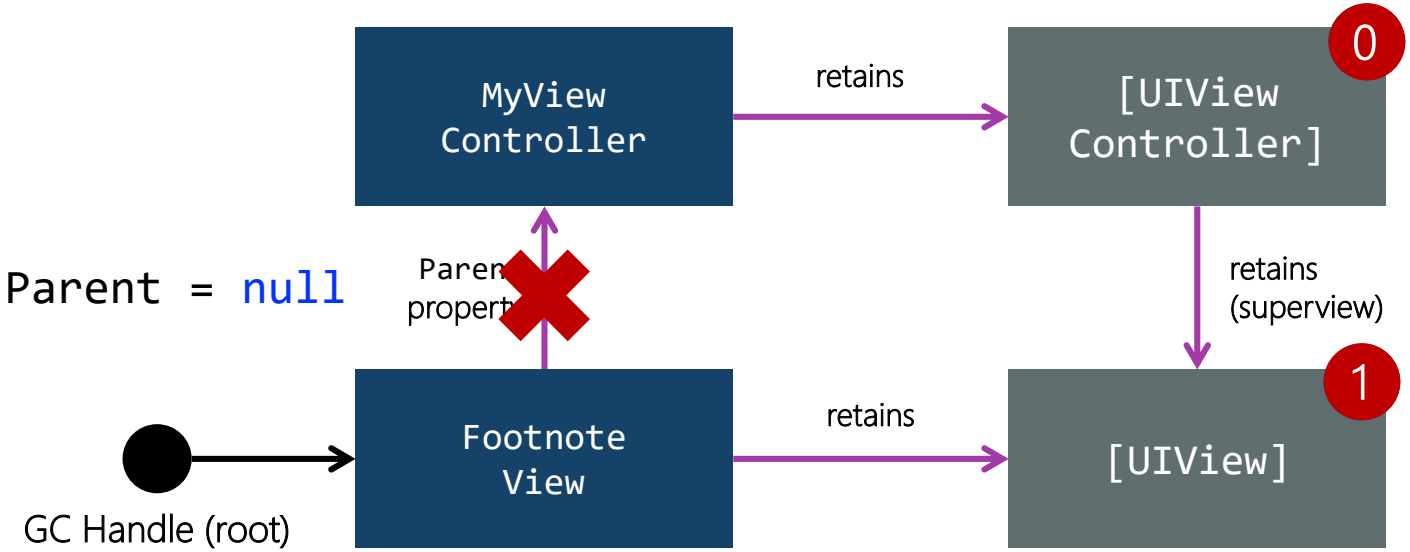
- Holding a managed reference to a peer from a user peer creates a reference cycle that cannot be broken automatically by GC

```
class MyViewController {}

class FootNoteview : UIView {
    public MyViewController Parent;
}
...
myVC.Add(new FootnoteView {
    Parent = this
});
```



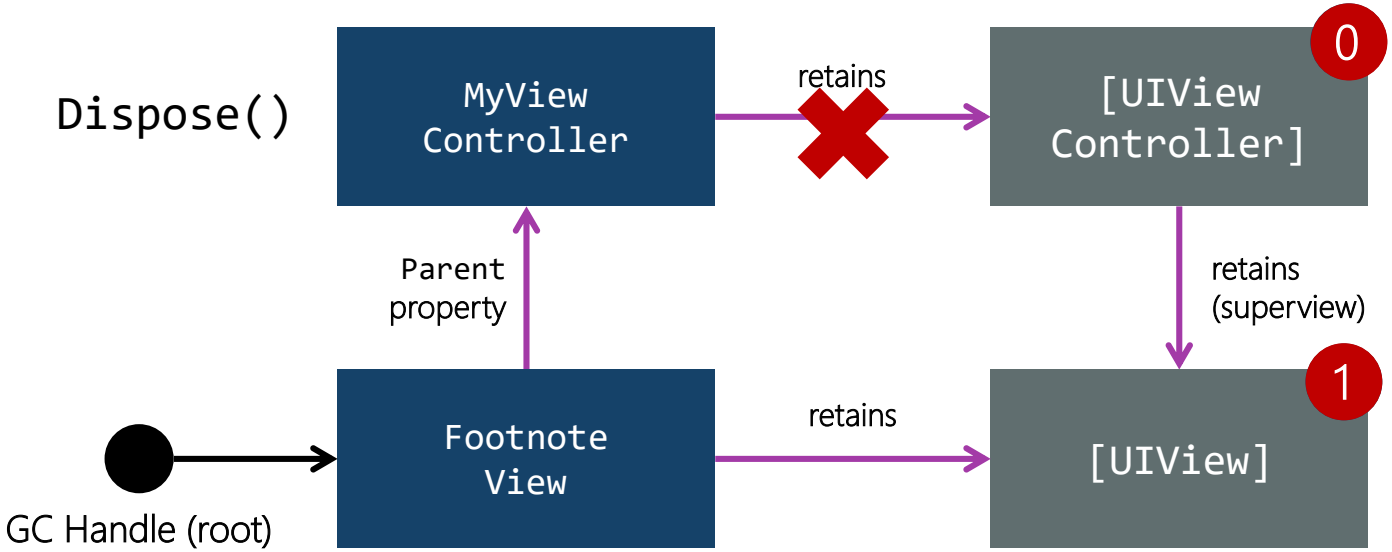
User Peers: breaking the reference

- Must manually break the reference cycle on one side or the other



User Peers: breaking the reference

- Must manually break the reference cycle on one side or the other



Group Exercise

Identifying and breaking strong reference cycles

Xamarin University

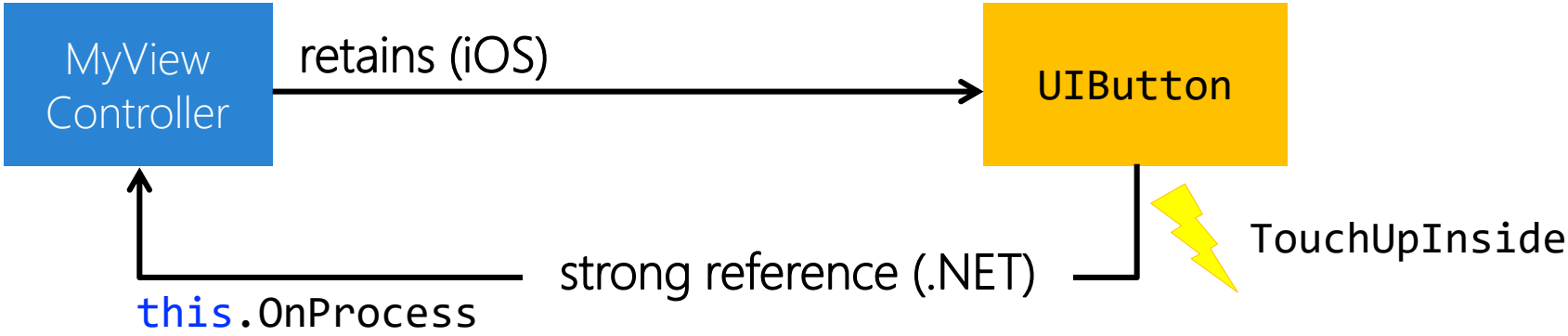
Framework peers > User peers

- When you wire up an event handler on a framework peer, this adds *state* to the peer object, so Xamarin.iOS **promotes** the object to a user peer

```
public class MyViewController : UIViewController
{
   public override void ViewDidLoad() {
      ...
      ProcessButton.TouchUpInside += OnProcess;
   }

   void OnProcess(object sender, EventArgs e) { ... }
}
```

What happens?

- The two user peers keep each other alive – this time due to the event handler (vs. a **Parent** property), but the end result is the same.. the graph will not be cleaned up properly!



Fixing the event problem

- Never forget: **always unsubscribe from event handlers**

```
public override void ViewDidAppear (bool animated) {
    base.ViewDidAppear (animated);
    ProcessButton.TouchUpInside += OnProcess;
}

public override void ViewDidDisappear (bool animated) {
    base.ViewDidDisappear (animated);
    ProcessButton.TouchUpInside -= OnProcess;
}
```

This is only necessary for *manually* wired events – if you use the designer to subscribe to UI actions, it is handled by the iOS runtime and doesn't use the event directly

Group Exercise

Watch out for peer promotions

Xamarin University

Xamarin.iOS Tips

- ✓ Prefer full delegate methods over lambdas – it makes it easier to see and understand strong references
- ✓ Call `Dispose()` to release native resources immediately (vs. waiting on a GC) when you are finished with a peer wrapper
- ✓ Always unsubscribe from events you manually wire up; alternatively, use the Storyboard to connect events which can then be cleaned up automatically

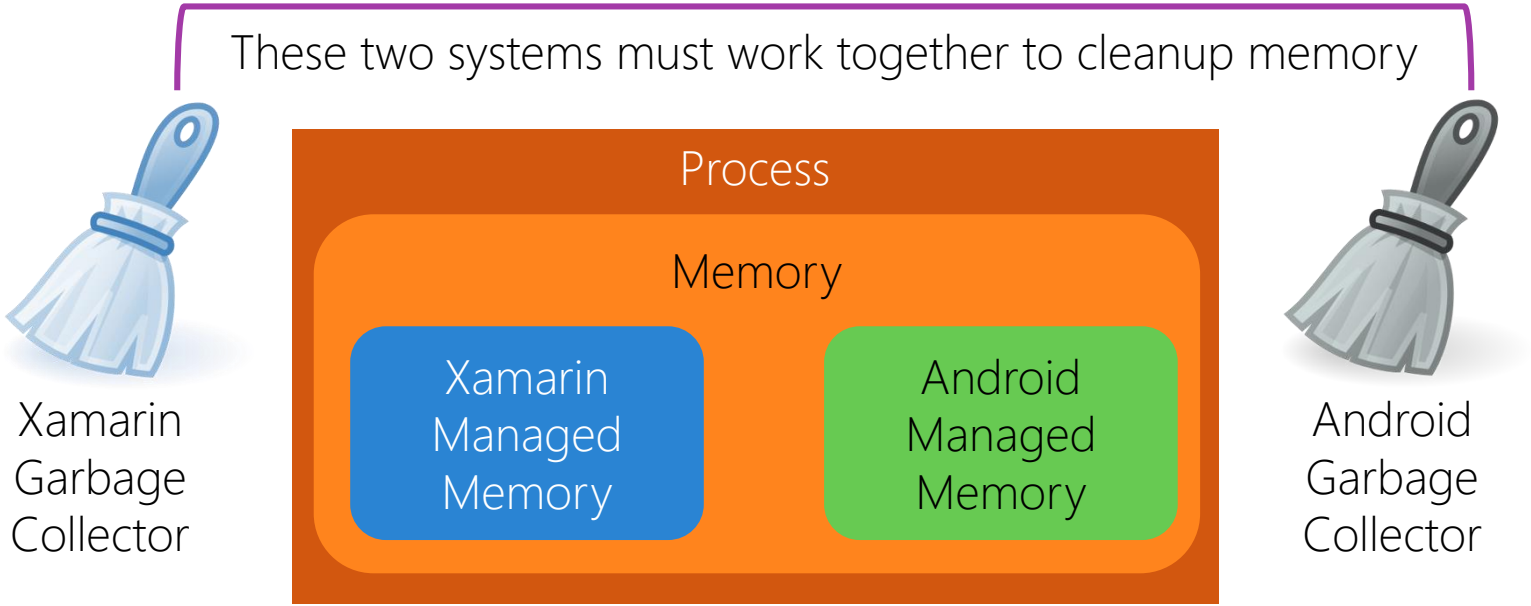
Xamarin.Android

Tasks

- Improve garbage collector performance
- Free native resources

Memory Management in Android

- Android also uses Garbage Collection to clean up resources



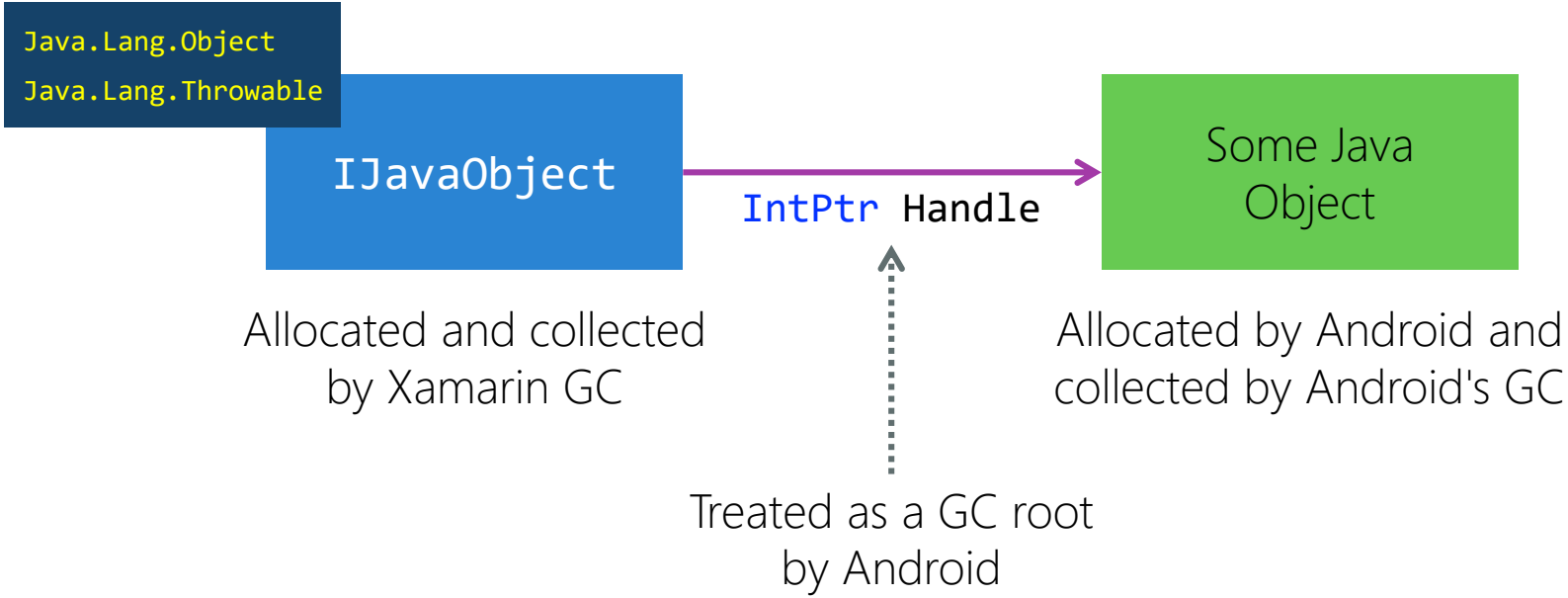
Android + Xamarin

- Xamarin.Android also has *peer objects* used to reference the native Java objects known to the Android JVM

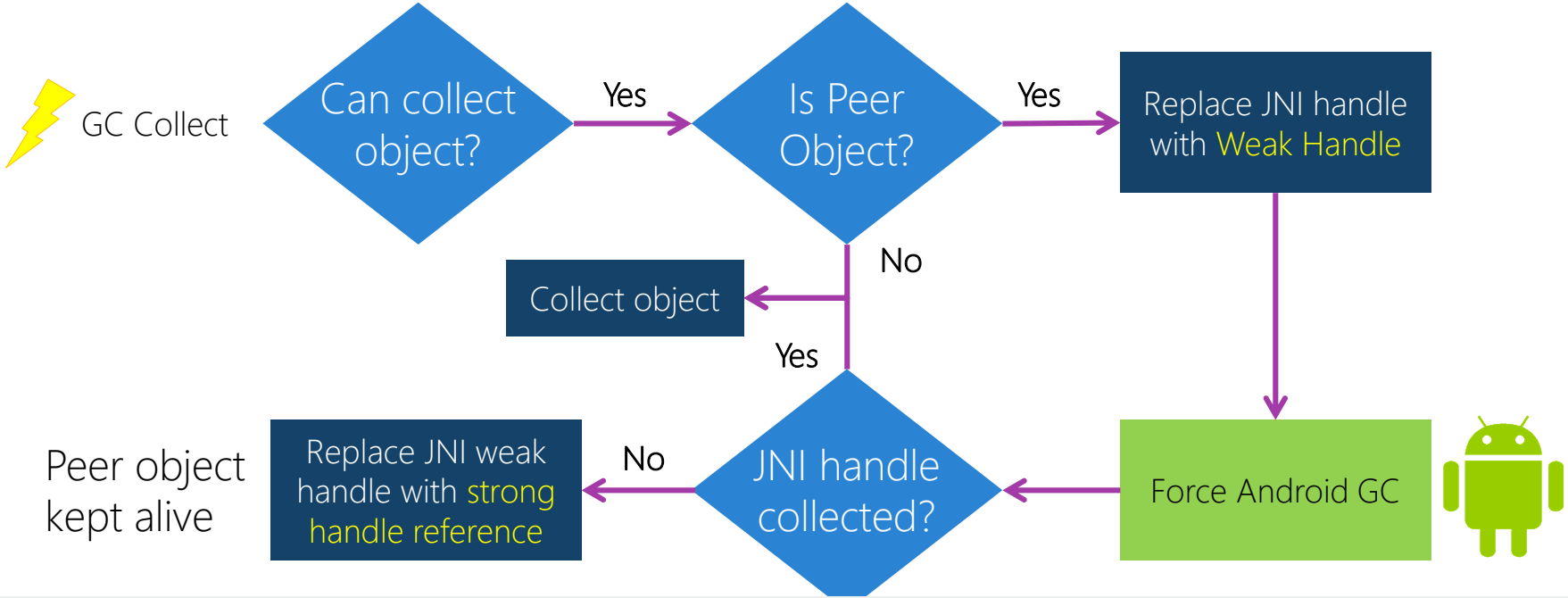
```
namespace Android.Runtime
{
    public interface IJavaObject : IDisposable
    {
        // JNI reference to Java object this is wrapping
        public IntPtr Handle { get; set; }
        ...
    }
}
```

GC process [Android]

- `IJavaObject` keeps a strong reference (JNI handle) to the platform Java object to keep it alive while the managed object is alive



GC process [Xamarin]

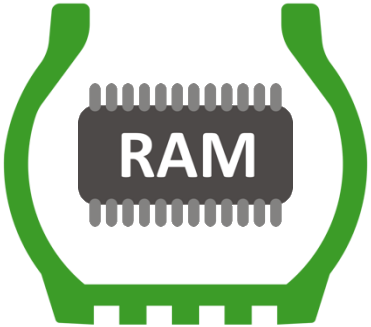


Note: this is a simplified view of what happens during a collection, the implementation has quite a few more details and special cases to deal with!

Disadvantages to having two GCs

- Xamarin.Android does not suffer from the cyclic reference problem encountered in iOS, but has unique issues of it's own

GCs take longer



App memory pressure is increased

Improving GC performance

- Should **Dispose** peer objects (**using** is your friend) and set references to **null** so GC can clean things up more quickly

```
static byte[] buf = new byte[1024];
...
using (Bitmap smallPic = BitmapFactory.DecodeByteArray(buf, ...))
using (Drawable dr = new BitmapDrawable(smallPic))
{
   layout.Background = dr;
   buf = null;
}
```

Thinking about big objects

- Some objects are much larger than the peer object – for example images often take up a significant block of native memory but look like a small object to the runtime

Peer object is ~20 bytes

479k on disk
1,562k in memory

Initiating a GC

- When you have released/disposed a large object, it can be helpful to call **`GC.Collect`** or **`JavaSystem.Gc()`** to reduce the working set

```
async void LoadBitmap(string url)
{
   using (HttpClient client = new HttpClient())
   using (var bitmap = await BitmapFactory.DecodeStreamAsync(
                 await client.GetStreamAsync(url)))
   {
      ... // Use bitmap
   }
   GC.Collect();
}
```

GC and Android types

- GC cost for walking a peer object graph is significantly higher because it must look for *both* managed *and* Java relationships between objects

```
class Tweet { ... }

class FeedActivity : ListActivity {
    List<Tweet> tweets = new List<Tweet>(1000);

    protected override void OnCreate (Bundle bundle) {
        base.OnCreate(bundle);
        ListAdapter = new ArrayAdapter<Tweet>(this,
            Android.Resource.Layout.SimpleListItem1, tweets);
   }
}
```

Here the GC will be forced to check **all 1000 `Tweet`** objects to see if any reference another peer

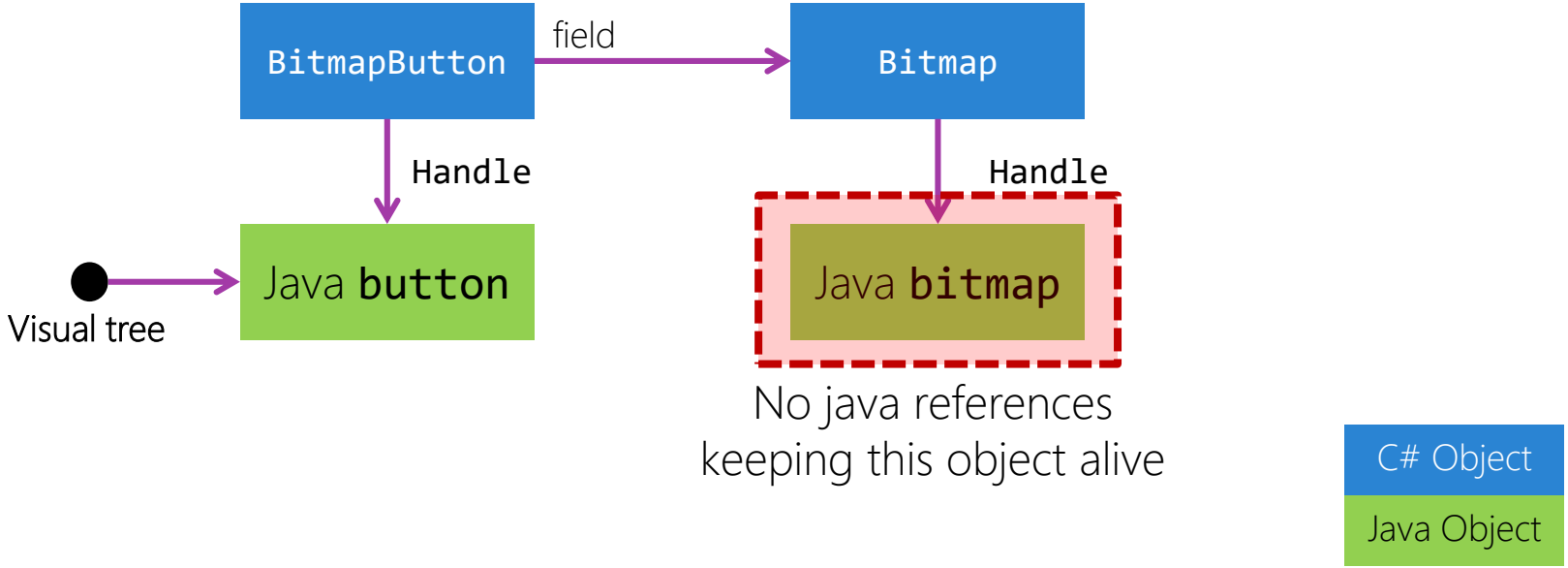
GC and Android types

Each tweet object has at least 6 additional references which need to be walked..

```
class Tweet
{
    public string Id { get; set; }
    public string Text { get; set; }
    public List<Tweet> Retweets { get; set; }
    public string CreatedAt { get; set; }
    public List<string> Hashtags { get; set; }
    public int FavoritedCount { get; set; }
    public string InReplyTo { get; set; }
    public string Language { get; set; }
    public Place Location { get; set; }
    ...
}
```

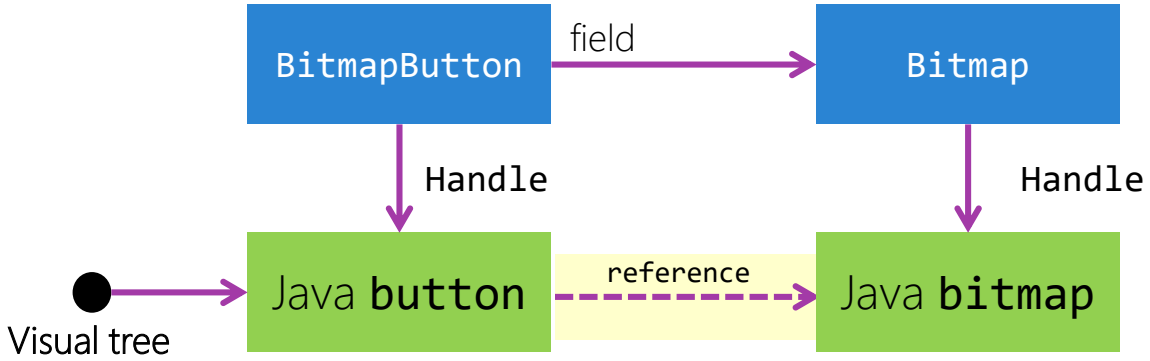
Why scan for relationships?

- When Xamarin GC runs, it will replace the strong JNI handle with a weak reference and invoke Android GC, which would then collect the Java bitmap



Why scan for relationships?

- Peers are scanned for **relationships** to ensure that each one is mirrored in the JVM – this keeps objects from being collected prematurely



Avoiding the peer walk

- Instead, prefer to split the data away from your peer objects to a non-peer object that holds the data **and is rooted**

```
class Tweet { ... }
static class TweetData { ... }
```

Can now be collected during a normal GC pass without involving peer scan

No direct reference used which needs to be examined by GC

```
class FeedActivity : ListActivity {
    ... // no instance reference
    ListAdapter = new ArrayAdapter<Tweet>(
      this, ..., TweetData.All);
}
```

Note: this sort of split is only necessary for larger object graphs – a handful of references is fine and should not impact your performance by a significant margin

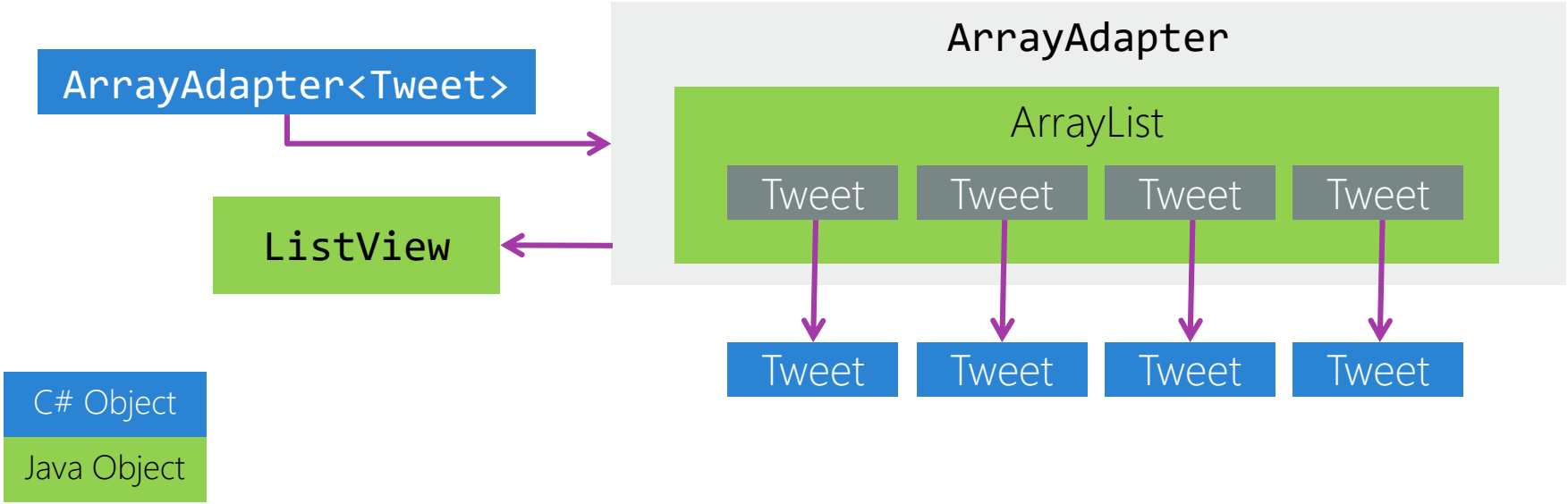
Boxing

- If possible, try to avoid passing **non-peer objects** into Java methods

```
class Tweet { ... }

class FeedActivity : ListActivity {

    protected override void OnCreate (Bundle bundle) {
        base.OnCreate(bundle);
        ListAdapter = new ArrayAdapter<Tweet>(this,
            Android.Resource.Layout.SimpleListItem1, TweetData.All);
    }
}
```

Xamarin and Java VM

- C# objects must be *boxed* to create a JVM representation of the object; intrinsic types (strings, numeric values and dates) are all special cased



Stay in your yard

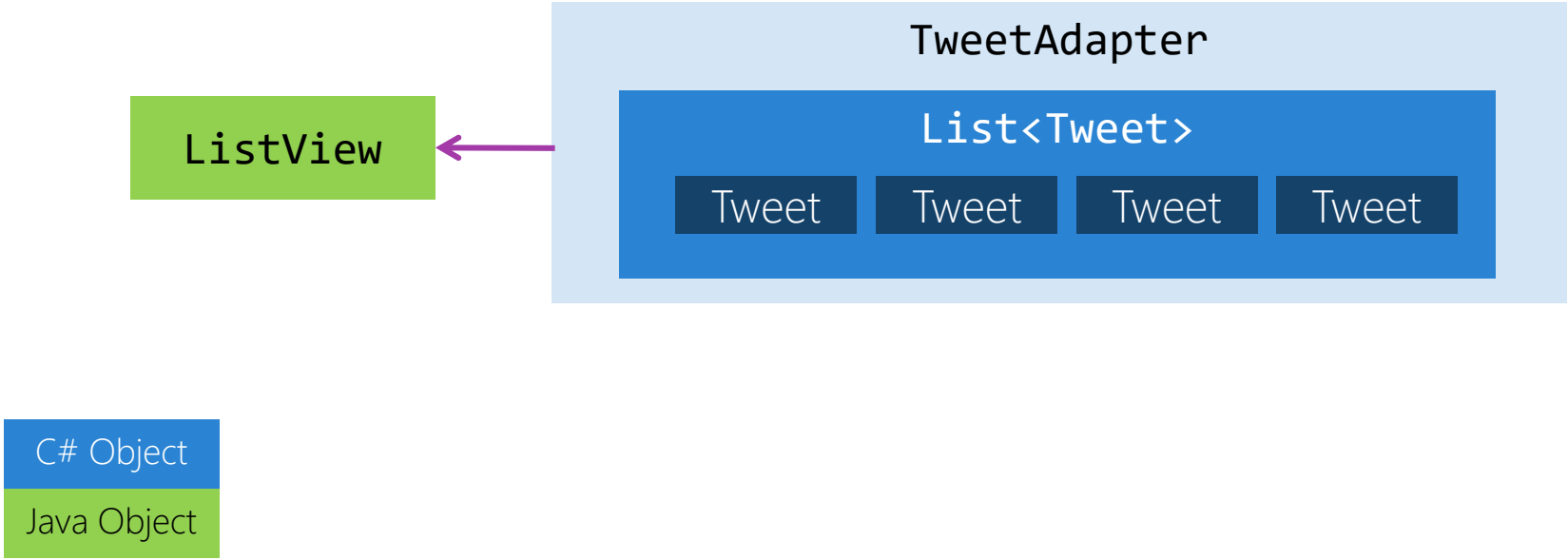
- Instead, do as much as possible in either C# or Java: interop is expensive

```
class Tweet { ... }
class TweetAdapter : BaseAdapter<Tweet> { ... }

class FeedActivity : ListActivity {

    protected override void OnCreate (Bundle bundle) {
        base.OnCreate(bundle);
        ListAdapter = new TweetAdapter(TweetData.All);
    }
}
```

Creating an adapter which conforms to an interface keeps all the data on the Xamarin side – the JVM simply invokes methods to retrieve data

Stay in your yard

- Instead, do as much as possible in either C# or Java, interop is expensive



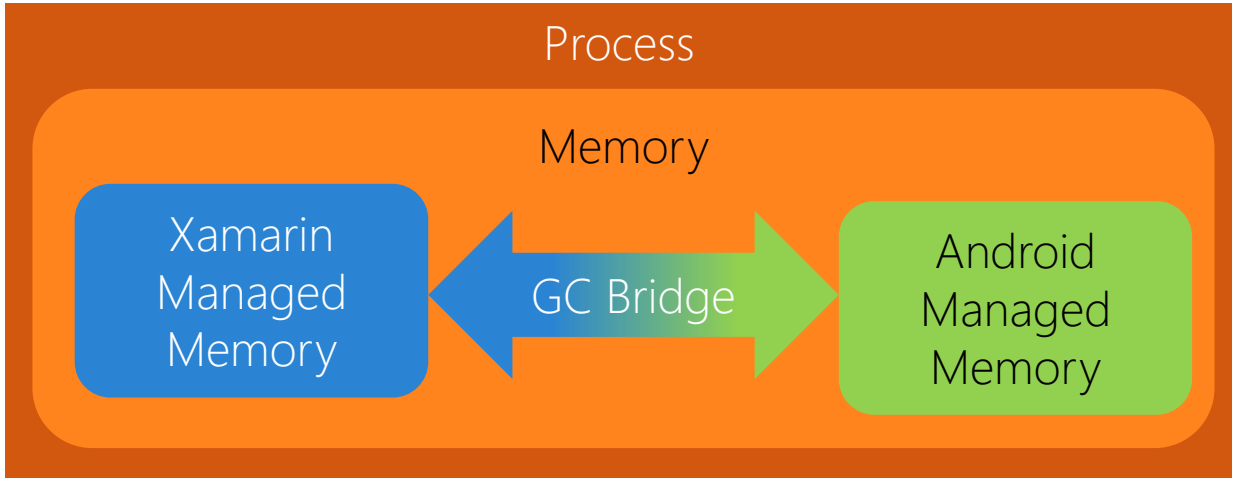
Demonstration

Show ListView memory and performance with a custom adapter

Xamarin University

GC Bridge

- Integration between Xamarin GC and JVM GC is performed through a native extension called the **GC Bridge**



Monitoring Bridge performance

- GC_ messages provide details about bridge and collection times

```
GC_MAJOR: (Minor allowance) pause 29.11ms, total 29.35ms, bridge 0.15ms major
4048K/0K los 2766K/0K
```

Lower **pause**, **bridge** and **total** times are preferred

GC Bridge choices

- Xamarin includes three different GC Bridge implementations

Old

New

Tarjan
(Default)

Selecting a different GC bridge

Solution
- MonkeyIsland
  - **MonkeyIsland**
    - References
    - Components
    - Packages (3 updates)
    - Assets
    - Properties
    - Resources
    - environment.txt
    - MainActivity.cs
    - packages.config

environment.txt

```
1 MONO_GC_PARAMS=bridge-implementation=tarjan
2
```

bridge-implementation=old
bridge-implementation=new
bridge-implementation=tarjan

- Open With ▶
- Version Control ▶
- Reveal in Finder
- Copy ⌘C
- Cut ⌘X
- Remove
- Rename ⌘R
- Build Action ▶
- Quick Properties ▶
- Properties

- Compile
- AndroidAsset
- ✓ AndroidEnvironment

Xamarin.Android Tips

- ✓ Should call `Dispose()` to release the native resources immediately (vs. waiting on a GC) when you are finished with a peer wrapper
- ✓ Avoid placing a large numbers of references in peer objects, remember GC is more expensive for these special types
- ✓ Avoid passing pure C# custom types into Java APIs if possible
- ✓ Experiment with the GC Bridge options

One more thing.. Xamarin.Forms

- ✓ Xamarin.Forms internally understands the tips we've covered – platform visual things are always disconnected and disposed
- ✓ Can still leak memory through traditional .NET *techniques*
- ✓ *DO* need to obey all the rules when creating custom renderers or effects which utilize the native platform

Thank You!

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

Microsoft