Lab 02: GC and Memory Management

Prerequisites

This lab is based on an iOS project so you will need an OSX based development environment, or the proper Windows setup with an OSX based build host.

Downloads

Download the starting solution and a completed version from http://university.xamarin.com.

Lab Goals

The goal of this lab will be to demonstrate how you can accidentally interfere with the garbage collection process by holding strong references to objects.

Steps

Open the Starting Solution

1. Launch Xamarin Studio and open the starting solution in the begin folder – **Lab2.GCAndMe.sln**.
2. This project has a simple `UIImageView`-derived class in it named **MyImageView**. Open this source file to examine the class – it overrides the `Dispose` method and the finalizer so we can see when this object is collected and disposed (if ever).
3. Go ahead and run the application in the iOS Simulator or on a physical device.
4. It will present a very simple UI with a button and image. The button handler will remove the image from the view by calling `RemoveFromSuperview` – go ahead and click on it to see the button disappear.
5. Our desire is that the GC collect the image view. It's especially important in this case because the image view is holding onto an image – which could potentially be rather large, so getting rid of it ASAP is important. However, it does *not* get collected – no text shows up to that effect in the output window.
6. Stop the app – let's try to help the GC out by forcing it to run a collection. Maybe, it just never actually collected anything. Find the comment `TODO: Step 1 - Force GC` and uncomment the code that follows it.
7. Run the app again, remove the image and now force a GC and watch the output window – does the image get collected now? Hit the button a few times to see if that helps.

It won't be collected. The problem is that we are still holding a reference to the image view – can you find it?

It's the local imageView field we declared when we created the image view! The lambda we are using to remove it from the superview is actually a delegate, and because it accesses that field, the compiler reorganized our code and moved the field declared in our `ViewDidLoad` method into a class field so both methods could access it. This inadvertently promoted the field and made it reachable while our View Controller is alive. It would have been obvious if we had used a method delegate instead of a lambda here, but the effect is the same.

8. Add the following code to the ViewDidLoad method after your Force GC button:

```
AddButton(ref y, "Set ImageView reference to null",
    (sender, e) => {
        Console.WriteLine ("Setting reference to null.");
        imageView = null;
    });
```

9. Run the application again, remove the button from the UI and then click your new button and watch the output window. You should see it get collected *and* disposed. Notice that the disposable happens *after* the finalizer – this is because the finalizer is calling the `Dispose` method as a last chance effort to cleanup unmanaged resources the image view is holding onto. This works, but isn't as efficient as it could be. Instead, we should dispose the image view as soon as we are done with it.

10. Add a call to dispose the image view after it is removed from the super view:

```
AddButton(ref y, "Remove Image from Screen",
    (sender, e) => {
        Console.WriteLine ("Removing image from UI");
        imageView.RemoveFromSuperview();
        imageView.Dispose();
    });
```

11. Run the app again and remove the image from the screen. Notice we now see our call for dispose with a boolean of True to indicate it's being called by our user code vs. the finalizer. You will not, however, see the object being finalized. This is because the dispose call removed the finalizer – it was not necessary anymore since we directly disposed the object. It is being collected now, we just don't see a notification about it.

12. An alternative approach to calling Dispose directly is to manage this object with a `WeakReference`. In this way, we won't keep a strong hold on the object – so if nothing else references it, the GC can collect it. You can use weak references to track objects for debugging as well, which can help to determine if it's being collected when you think it should be.

13. Change your code as shown below with the highlighted lines:

```
WeakReference wr = new WeakReference(imageView);
imageView = null;

// Create the button to remove our image
AddButton(ref y, "Remove Image from Screen",
    (sender, e) => {
        Console.WriteLine ("Removing image from UI");
        var iv = wr.Target as UIImageView;
        if (iv != null)
            iv.RemoveFromSuperview();

    });
```

14. Run the app a final time and remove the image – you should see the image view get collected and disposed in the normal GC cycle – this time because we aren't holding a live, strong reference to the object. If you examine the IsLive property on the weak reference, you will see it changes to false after this collection occurs.

Summary

In this lab, we looked at how the garbage collection process cleans up our memory and how event handlers can interfere with that process. You also saw how you can dispose objects in your code to clean them up after you are finished with them, and also how to use weak references to remove our strong holds on objects which can also help the GC work better in this mobile environment.