CSC270

Garbage Collection Fundamentals

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Explain the Garbage Collector

Explain the Garbage Collector

Xamarin University

Tasks

1. Advantages to using a GC
2. What triggers a GC?
3. Monitoring GCs in your app
4. Generational garbage collectors
5. Helping out the GC

Memory Management

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

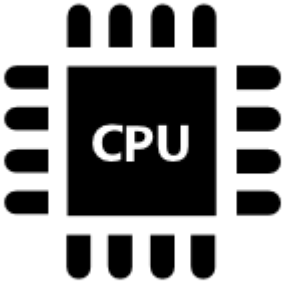
- GC in Mono and .NET are similar, but the implementation is quite different, so some strategies you use in .NET may need adjustment

Advantages to using a GC

- Most modern languages/runtimes include some form of GC because of the benefits associated with it

Simple to work with

Reduces Memory Leaks



Cache Locality

Demonstration

Show allocations and collections

Xamarin University

Working with the GC

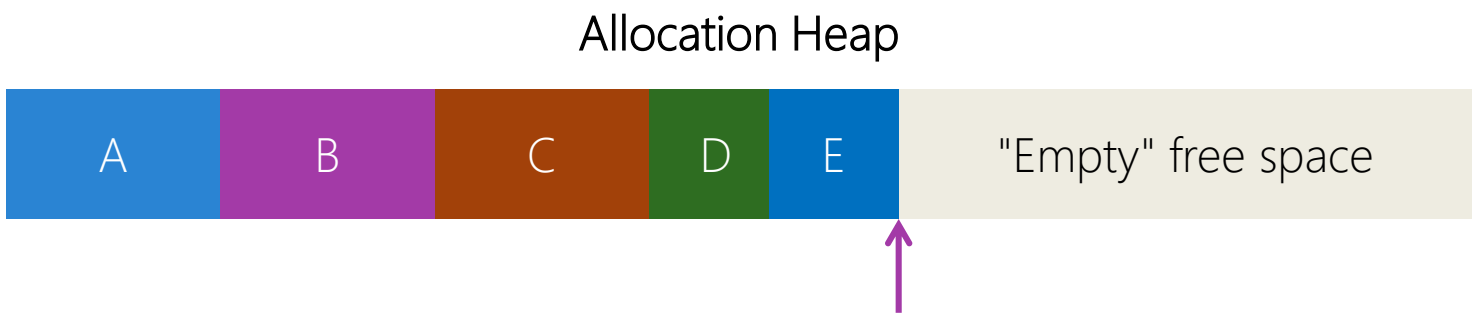
- Cannot *disable* GC, but understanding how it works can help you avoid performance issues in your application, there are two things you will be concerned with

What triggers a GC?

Minimizing the pause time introduced by GC

What triggers a GC?

- GC runs as part of the memory allocation process

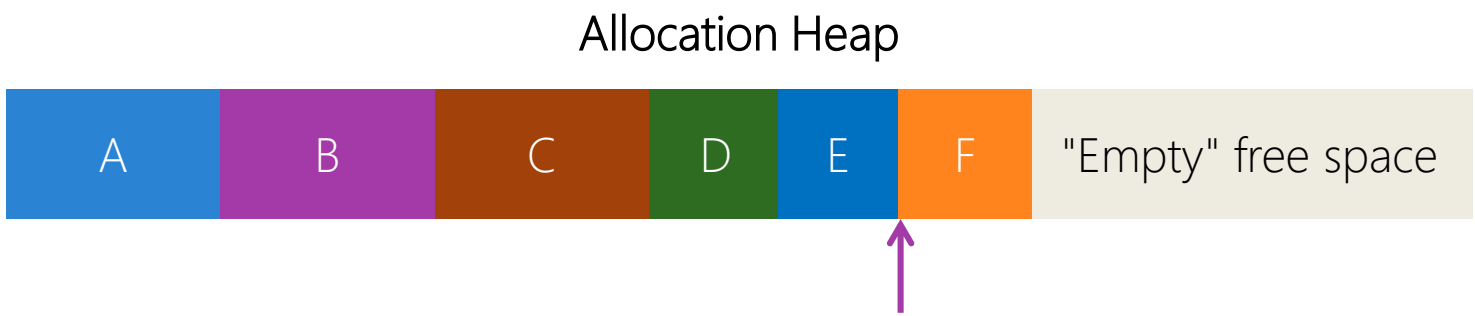



```
var f = new F();
```

Current allocation location
(pointer to free space)

What triggers a GC?

- Runtime uses a lock-free "pointer bump" allocation algorithm

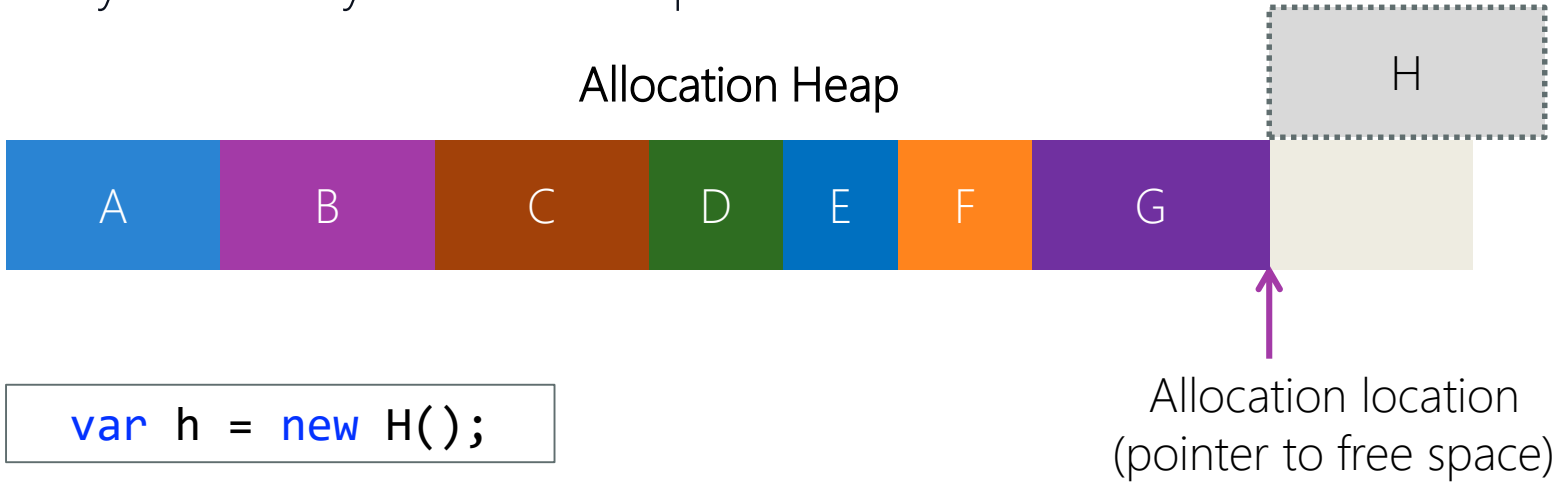


Allocator returns a memory block large enough to fit an "F" instance with the block zero'd out and moves "free" pointer

New allocation location (pointer to free space)

What triggers a GC?

- GC kicks in on the **current thread** when there's not enough space to satisfy a memory allocation request

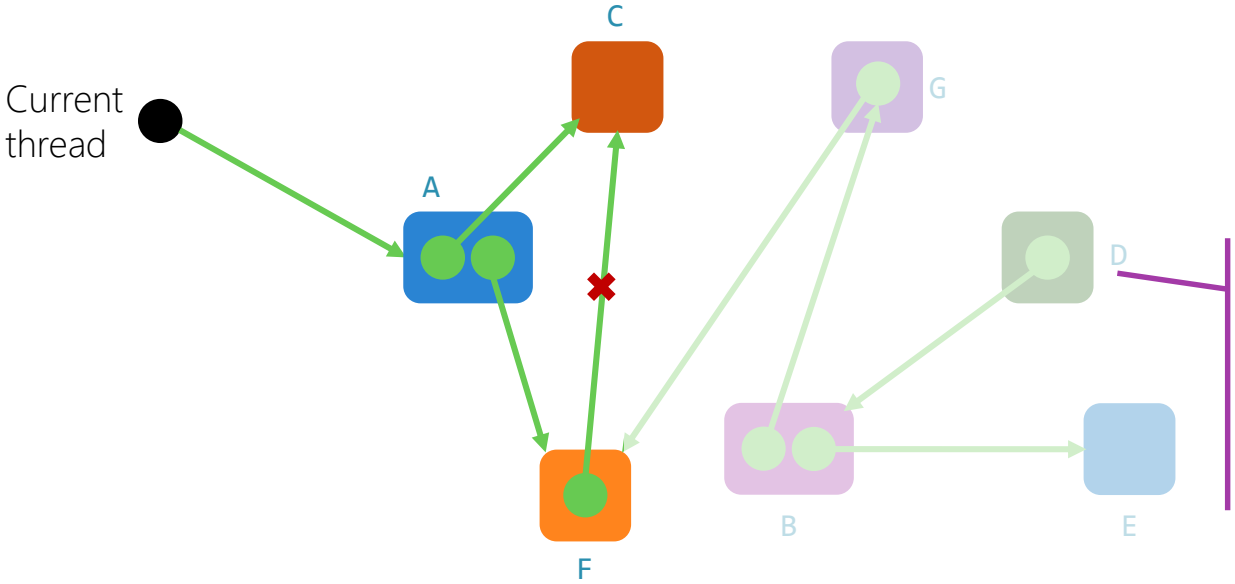



```
var h = new H();
```

There is not enough memory in allocation heap for **H** object – system triggers a GC on this thread to try to free up enough space

What happens during a GC?

- When a collection occurs, GC locates the **roots** and *traverses* the live reference graph from each one to determine what is reachable



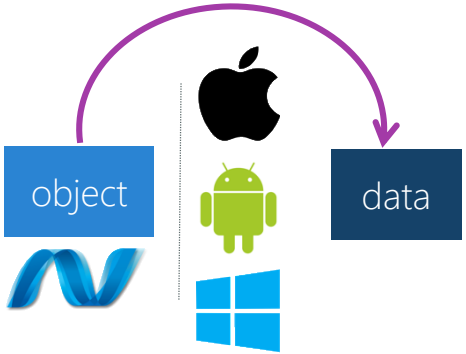
Objects left untouched are not reachable and thrown away (collected)

What is a "root"

- Live references are object instances that can be accessed from a **root object** coming from one of three places:

Static property or field

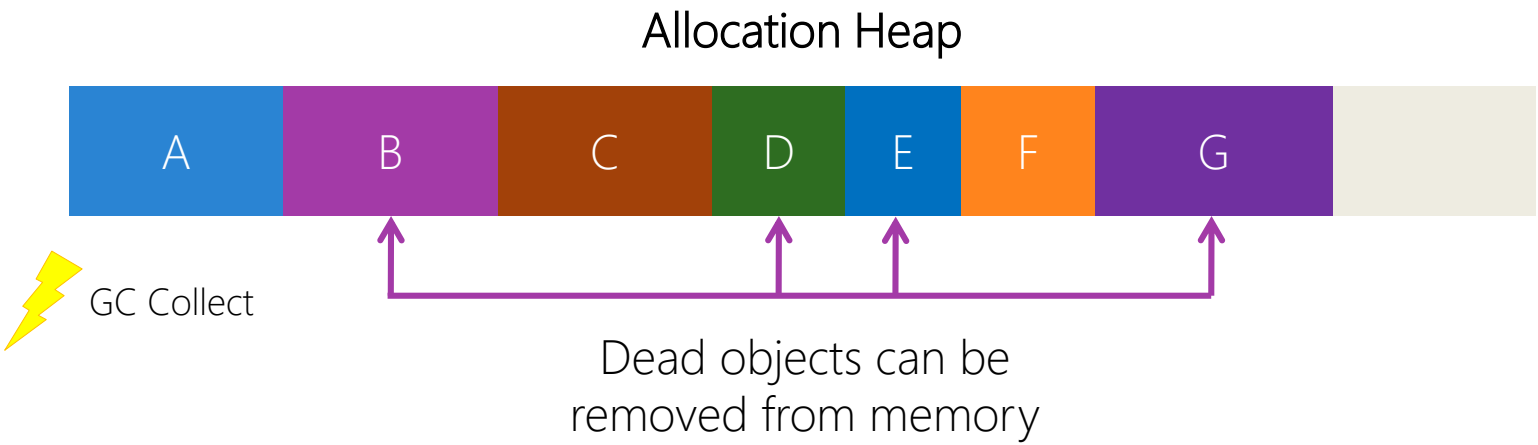
Reference on the stack of a managed thread



Objects passed into the native platform ("pinned")

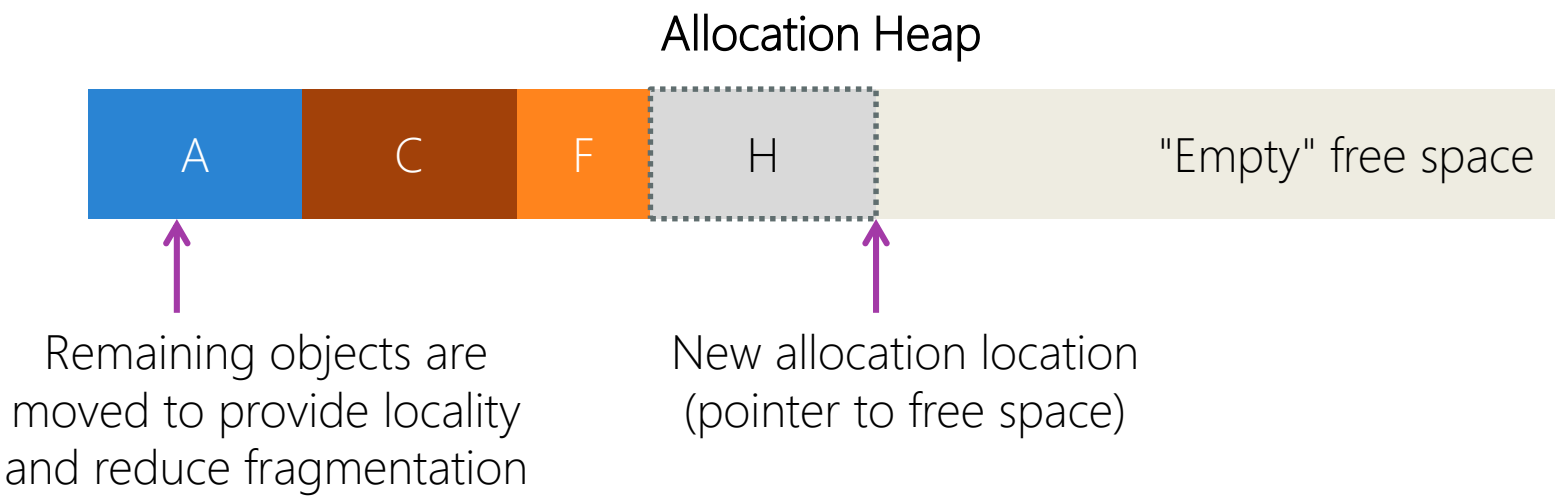
What triggers a GC?

- GC runs and identifies **live objects**, anything not referenced can be thrown away



What triggers a GC?

- Space is allocated for new object (H) and free space pointer is moved



What triggers a GC?

- Application code can also *request* that a garbage collection occur

```
Image bigPicture = ...;
...
bigPicture = null; // Image is no longer referenced

// Ask GC to run now .. to get rid of the image
GC.Collect();
```

Mono != .NET

- Mono implementation of **GC** class isn't as full featured as .NET

Does nothing	Partially implemented	Throws exception
AddMemoryPressure	Collect	WaitForFullGCApproach
RemoveMemoryPressure		WaitForFullGCComplete
		RegisterForFullGC Notification
		CancelFullGCNotification

Flash Quiz

Xamarin University

Flash Quiz

① The runtime triggers a garbage collection _____

a) Every two seconds on a timer

b) When you call `GC.Collect`

c) When you use the `Dispose` method on an object

d) When a memory allocation cannot be satisfied

Flash Quiz

① The runtime triggers a garbage collection when _____

a) Every two seconds on a timer

b) When you call **`GC.Collect`**

c) When you use the **`Dispose`** method on an object

d) When a memory allocation cannot be satisfied

Flash Quiz

② GC always runs in parallel to your program code on a separate thread

a) True

b) False

Flash Quiz

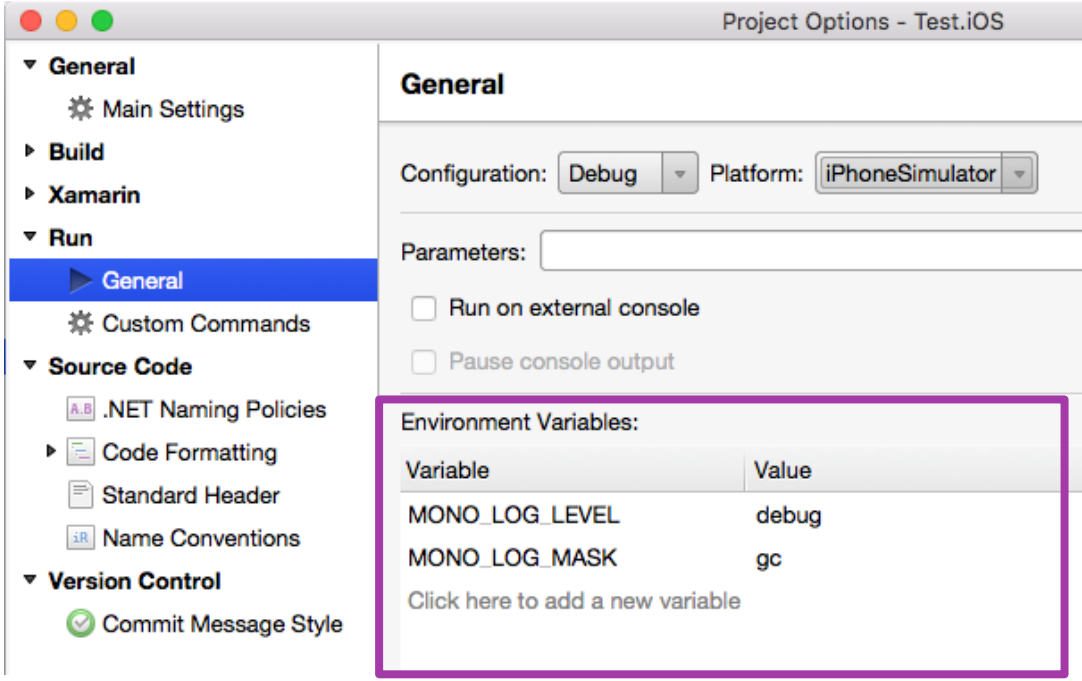
② GC always runs in parallel to your program code on a separate thread

a) True

b) False

Monitoring GCs in your app

- Xamarin.Android outputs GC traces to the debug console for all debug builds automatically
- Xamarin.iOS requires you add two environment variables to get the output in the debug console



Debug output

- GC messages are pre-pended with **GC_** tags

What triggered this GC

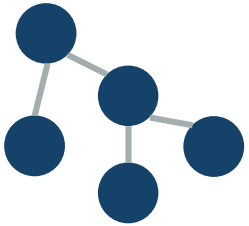
```
GC_MINOR: (Nursery full) pause 0.42ms, total 0.43ms, bridge 0.00ms promoted 5K major 5728K los 37K
GC_MAJOR: (user request) pause 6.07ms, total 6.07ms, bridge 0.00ms major 5728K/5728K los 29K/29K
```

How long did the GC take

How long does a GC take?

- The duration (pause time) it takes to do a collection depends on two primary factors:

What type, or **generation** of collection is occurring



How many objects are reachable and have to be examined

Generational garbage collectors

- GC works off the premise that *most objects die young*; Mono and Android separate objects into two *generations* (.NET uses three)

Nursery: where objects are allocated

Objects that get collected here tend to be *transient* or *temporary* objects such as strings, return values, and intermediate calculated objects used in methods

Generational garbage collectors

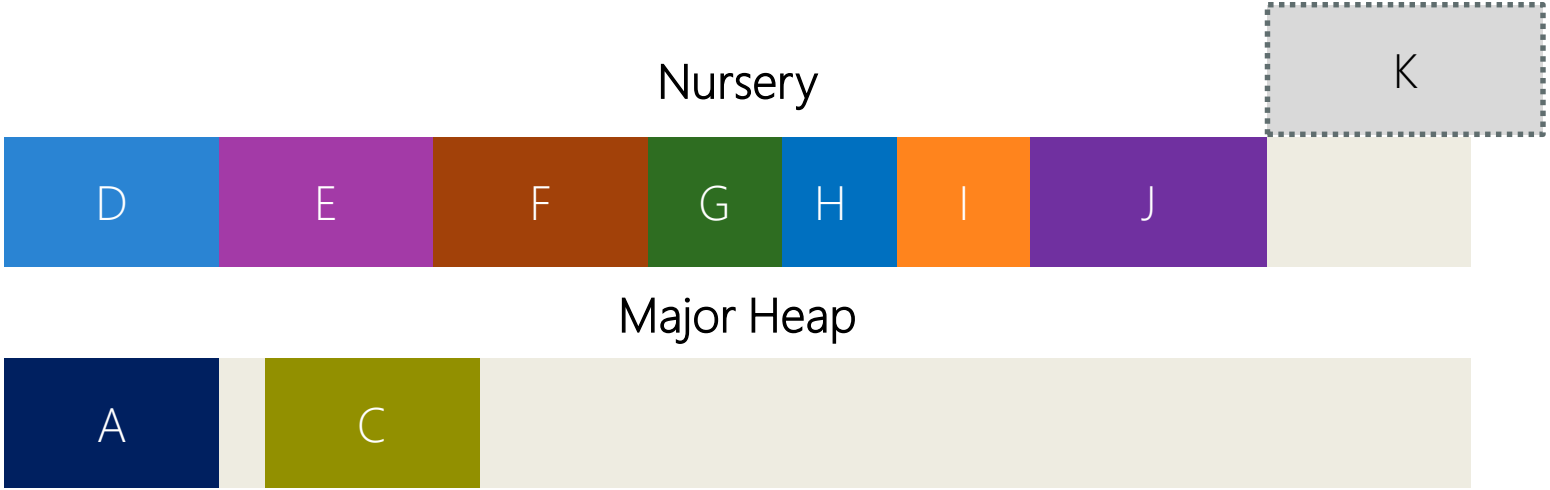
- GC works off the premise that *most objects die young*; Mono and Android separate objects into two *generations* (.NET uses three)

Objects moved here have survived a collection (matured) and statistically will likely be reachable for a while (e.g. sometimes for the entire life of the application)

Major:
objects which have survived a collection

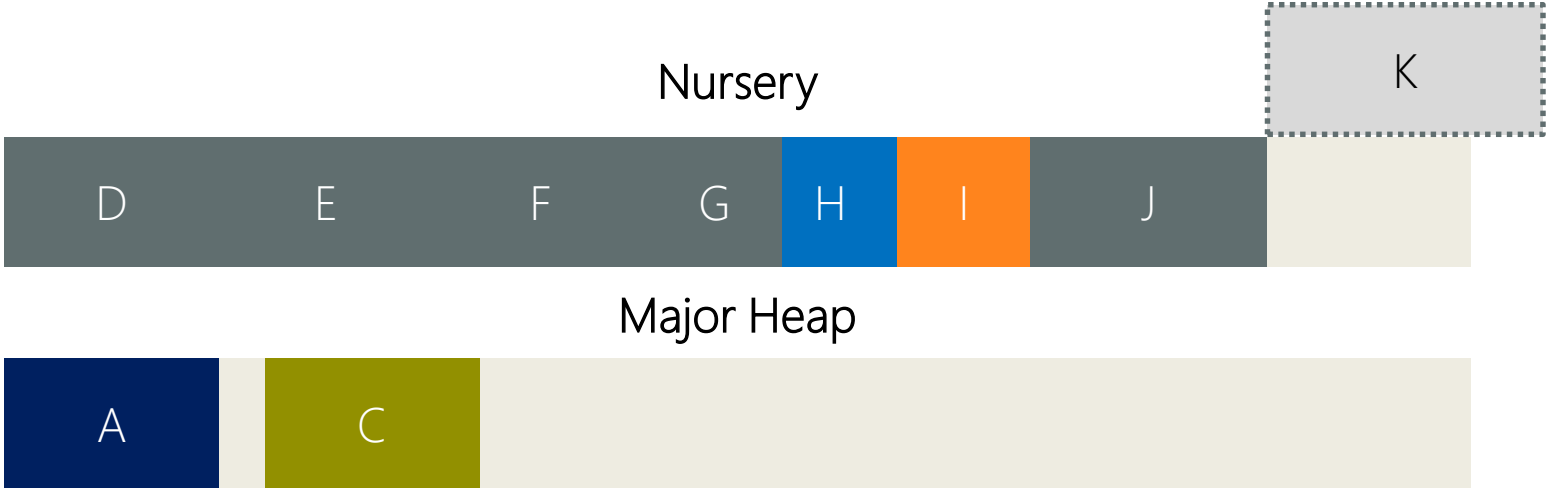
Collecting the nursery

- Nursery collections happen anytime an allocation fails; since the nursery is small, this happens frequently and pauses tend to be short



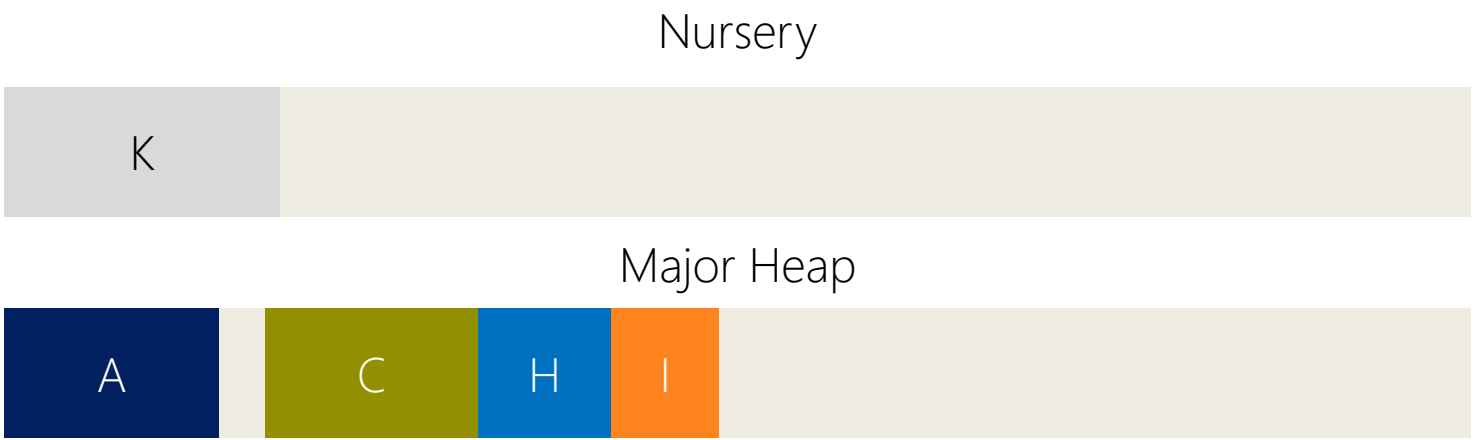
Collecting the nursery

- Nursery collections happen anytime an allocation fails; since the nursery is small, this happens frequently and pauses tend to be short



Adding generations

- Live objects are moved into **major heap** and pointer references are changed to point to the new locations



Debug log: promoting memory

- **GC_MINOR** reports promotion size and current major generation size

```
GC_MINOR: (Nursery full) pause 0.42ms, total 0.43ms, bridge 0.00ms promoted 5K major 5728K los 37K
```

How much memory survived nursery collection and was copied

How much is currently in the major generation

Debug log: 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?

Collecting the major heap

- Major heap is collected less frequently; typically due to promotion growth or when there's not enough space in the heap to complete a nursery collection

Remember: each time the GC runs, it **stops the world** and our app is not executing user code, major collections could potentially stop your app for up to a second

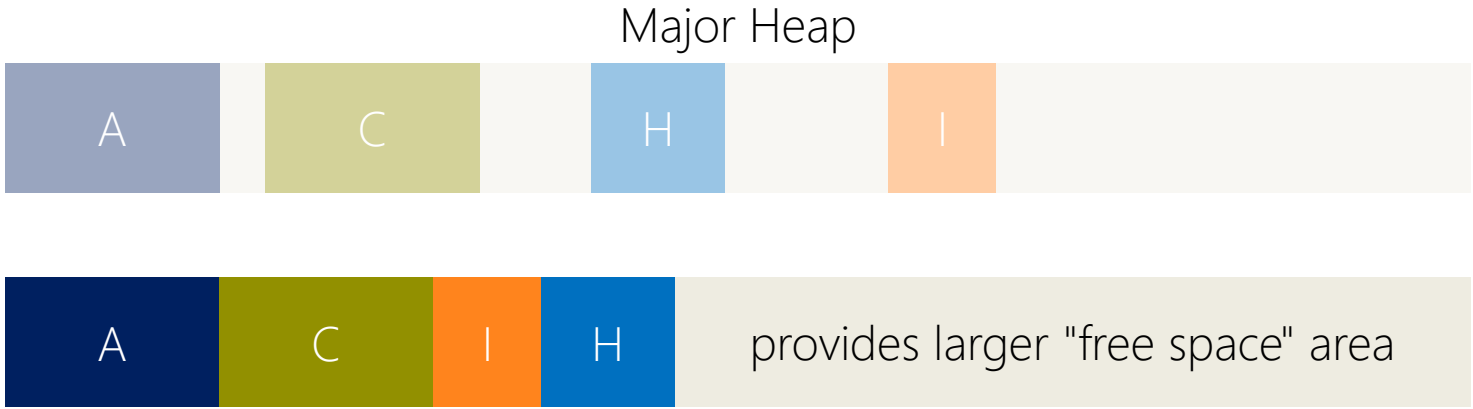
Nursery vs. Major collection ratio

- System allows the major heap to **grow by ~1/3** before initiating a full collection
- Healthy GC behavior is around **1:10** for major to minor collections, depending on what the app does
- Check debug log or use **`GC.CollectionCount`** to determine the ratio for your app

```
45.571 GC_MAJOR: ...
45.588 GC_MINOR: major 4256K
45.606 GC_MINOR: major 6320K
45.625 GC_MINOR: major 7888K
45.641 GC_MINOR: major 8336K
45.660 GC_MINOR: major 10416K
45.682 GC_MINOR: major 11840K
45.699 GC_MINOR: major 12160K
45.719 GC_MINOR: major 14240K
45.738 GC_MINOR: major 15536K
45.754 GC_MINOR: major 15728K
45.772 GC_MINOR: major 17808K
45.790 GC_MINOR: major 18976K
45.806 GC_MINOR: major 19040K
45.829 GC_MINOR: major 21104K
45.851 GC_MAJOR: ...
```

Compacting the major heap

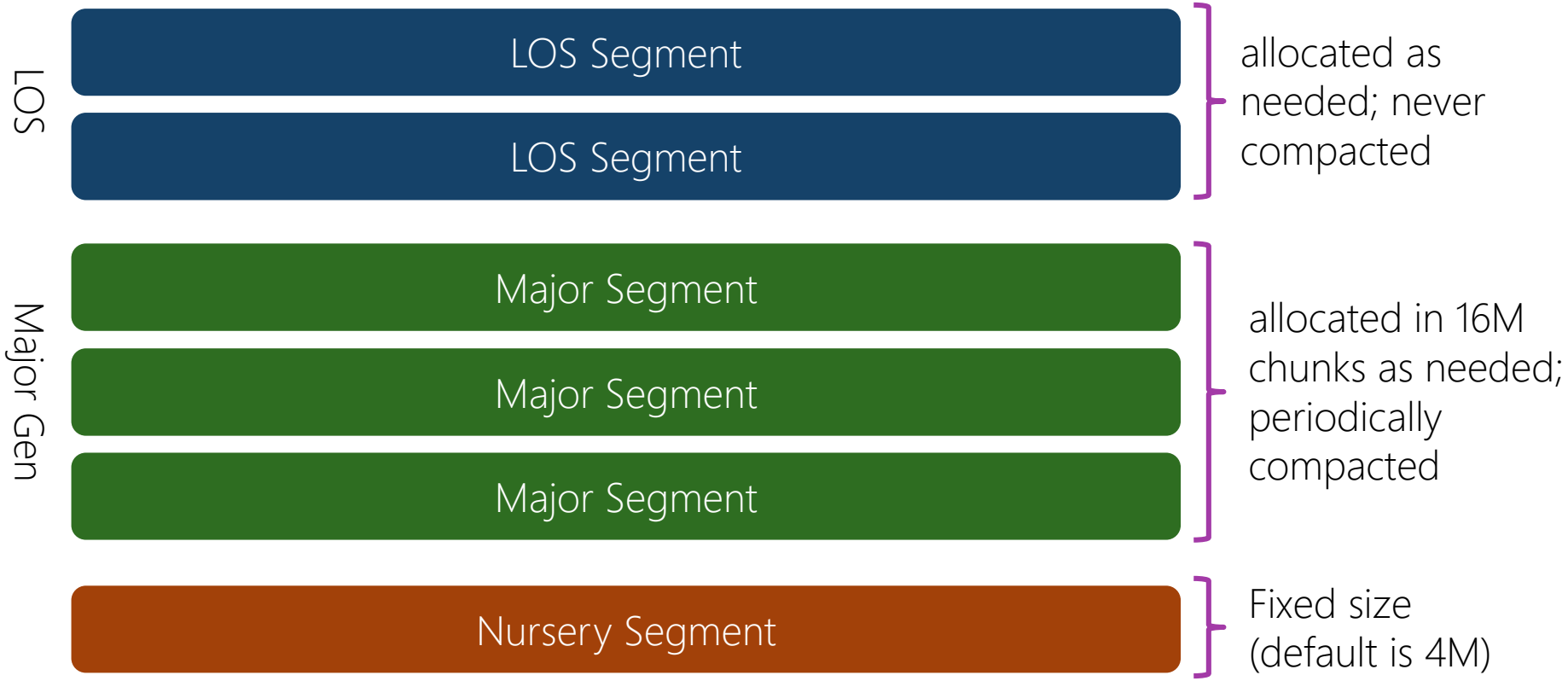
- Major heap is organized in *buckets* and objects are moved from the nursery into a specific bucket based on size; GC will *periodically compact* these buckets to reduce fragmentation



Big objects are different

- Some objects are more expensive to move/compact due to their size (i.e. large arrays)
- These *large objects* are placed into a special heap referred to as the **Large Object Space (LOS)** (or Large Object Heap in .NET)
- Collected on full collections and not compacted

SGen memory management



Flash Quiz

Xamarin University

Flash Quiz

① The **nursery generation** will grow and shrink based on usage

a) True

b) False

Flash Quiz

① The **nursery generation** will grow and shrink based on usage

a) True

b) False

Flash Quiz

② To turn on GC diagnostic tracing in iOS you need to _____

a) Do nothing. It's on by default.

b) Add an entry to your **info.plist**

c) Add two environment variables to the project settings

Flash Quiz

② To turn on GC diagnostic tracing in iOS you need to _____

a) Do nothing. It's on by default.

b) Add an entry to your info.plist

c) Add two environment variables to the project settings

Setting variables to null

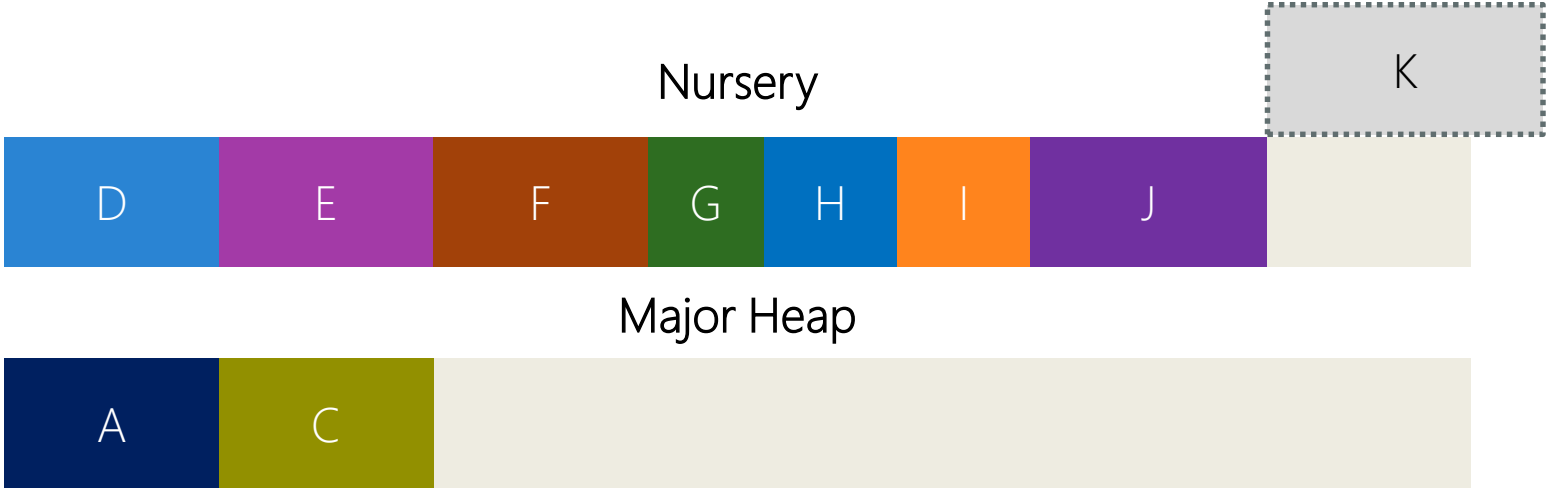
- Can set **static** or class instance fields and properties to **null** to let the GC collect the associated references

```
public class App : Application
{
    public IDBFactory DbFactory { get; set; }

    protected override void OnStart () {
        IDBRepository repo = DbFactory.Create ();
        ...
        // Don't need factory anymore.
        DbFactory = null;
    }
}
```

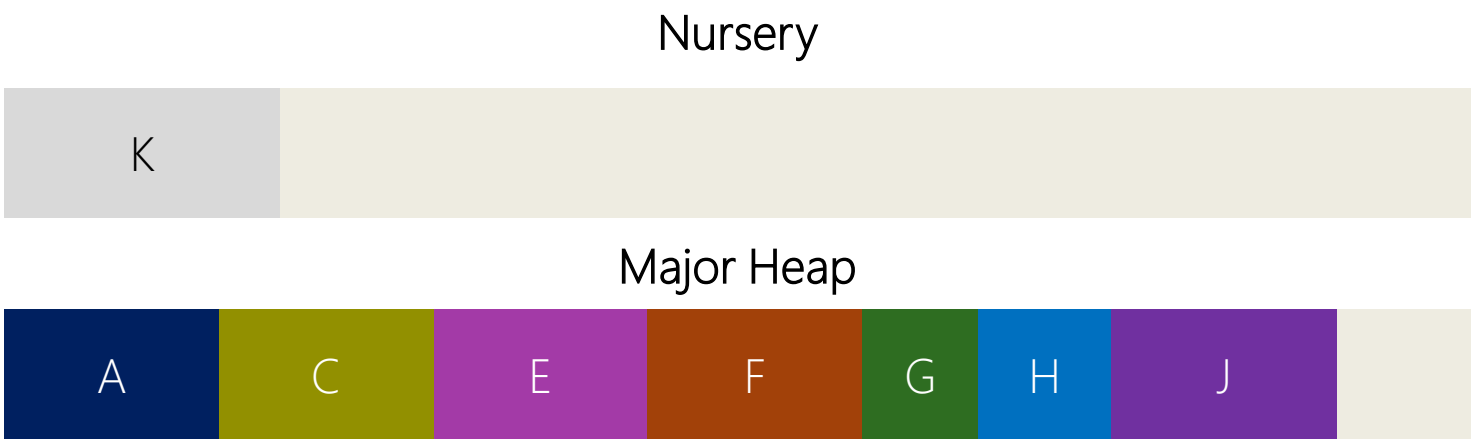
Best case scenario

- We want our objects to *either* have very short lifetimes and be removed in the nursery collection, or to live long lifetimes



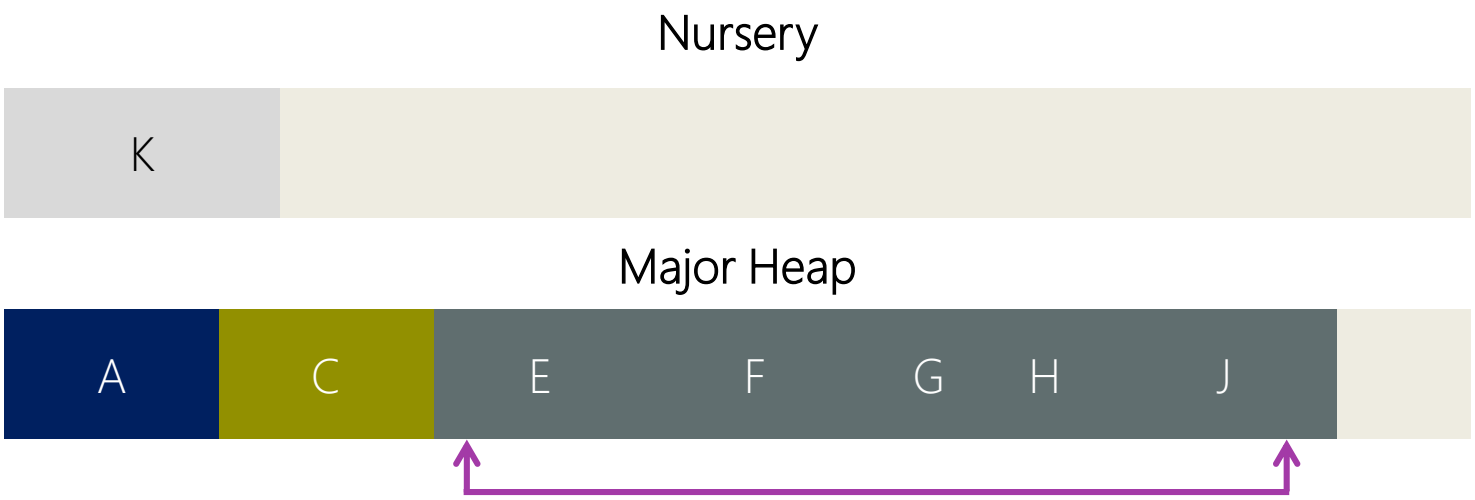
Worst case scenarios

- App is executing code which still had references to a bunch of temporary objects when a nursery collection occurs, which moves *all of these objects to the major generation!*



Worst case scenarios

- The worst thing scenario is when object(s) are promoted to the major generation *and then expires shortly thereafter*



Now these objects require a major generation collection to go away …

Minimizing the mid-life collection

- Can call **`GC.Collect`** before starting memory intensive data processing to try to avoid mid-processing collections

```
void DoSomeExpensiveThing()
{
    // Force a collect before we start so nursery is empty
    GC.Collect ();
    ... // Do work that allocates objects
}
```

Tweaking the GC

- **MONO_GC_PARAMS** environment variable can be set to tweak the GC settings based on your app needs

nursery-size=#bytes
(defaults to 4M)

Can change the fixed segment size used for the nursery to minimize the # of collections at the expensive of a larger working set

```
MONO_GC_PARAMS=nursery-size=8M
```

Tweaking the GC

- **MONO_GC_PARAMS** environment variable can be set to tweak the GC settings based on your app needs

nursery-size=#bytes
(defaults to 4M)

soft-heap-limit=#bytes
(defaults to -1)

Can set a max working set limit to reduce how often the major heap is collected

```
MONO_GC_PARAMS=soft-heap-limit=512M
```

Tweaking the GC

- **MONO_GC_PARAMS** environment variable can be set to tweak the GC settings based on your app needs

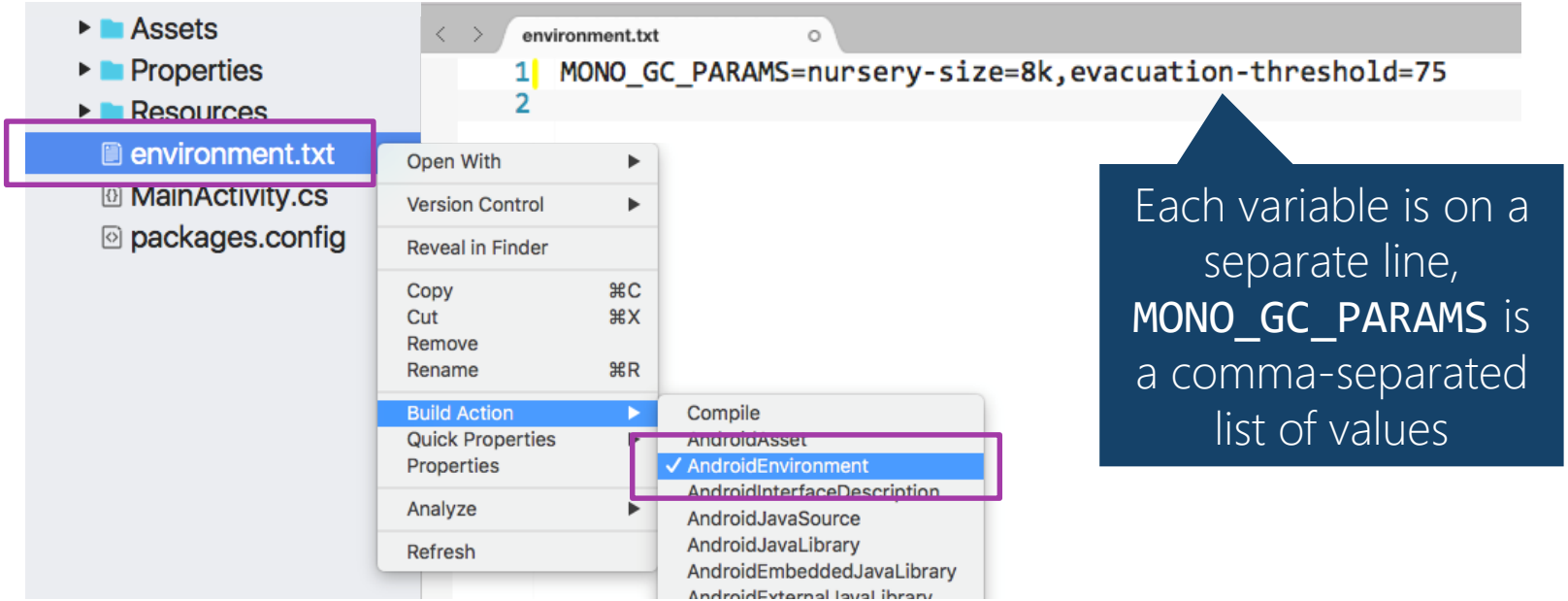
nursery-size=#bytes
(defaults to 4M)

Can adjust how often the major heap is compacted based on the fragmentation percentage

evacuation-threshold=%
(defaults to 66)

```
MONO_GC_PARAMS=evacuation-threshold=50
```

Setting environment variables in Android



Each variable is on a separate line, **MONO_GC_PARAMS** is a comma-separated list of values

Demonstration

Adjust the nursery size

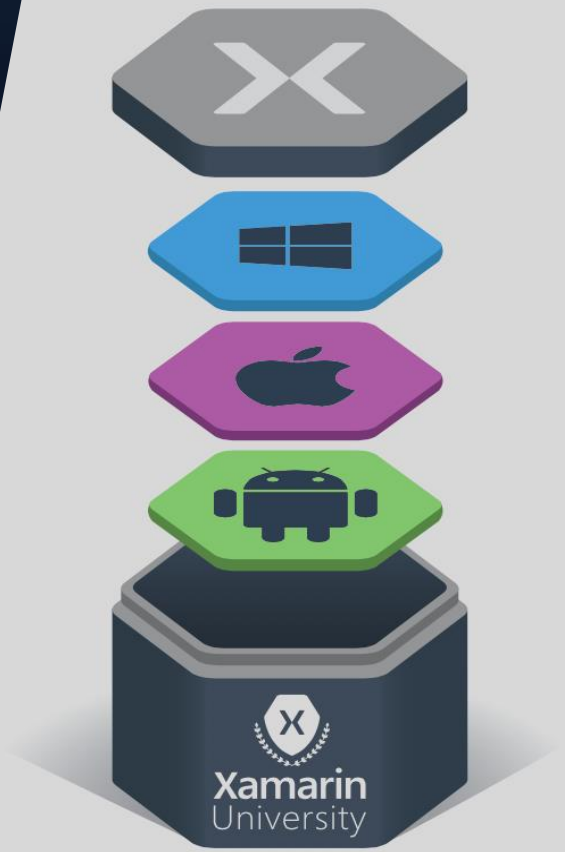
Xamarin University

Summary

1. Advantages to using a GC
2. What triggers a GC?
3. Monitoring GCs in your app
4. Generational garbage collectors
5. Helping out the GC

Thank You!

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



Microsoft