Programming the .NET Framework 4.5

Module 02 – Memory Management (GC)

In This Chapter

- Overview of memory management
- Garbage collection first steps
- GC flavors
- Generations
- Interacting with the GC
- Weak references
- Finalization and Dispose
- Lab

.NET as a Managed Environment

- .NET is a **managed environment**
- Memory management is **difficult**
- The *garbage collector* (**GC**) **takes care** of memory management

Requirements

- GC design goals
- First GC implementation – **Lisp, circa 1963**

.NET Tracing Garbage Collection

- Objects are collected at **non-deterministic times**
- No promise of deterministic finalization
- **No overhead** while the GC is **idle**

Managed Heap, Next Object Pointer

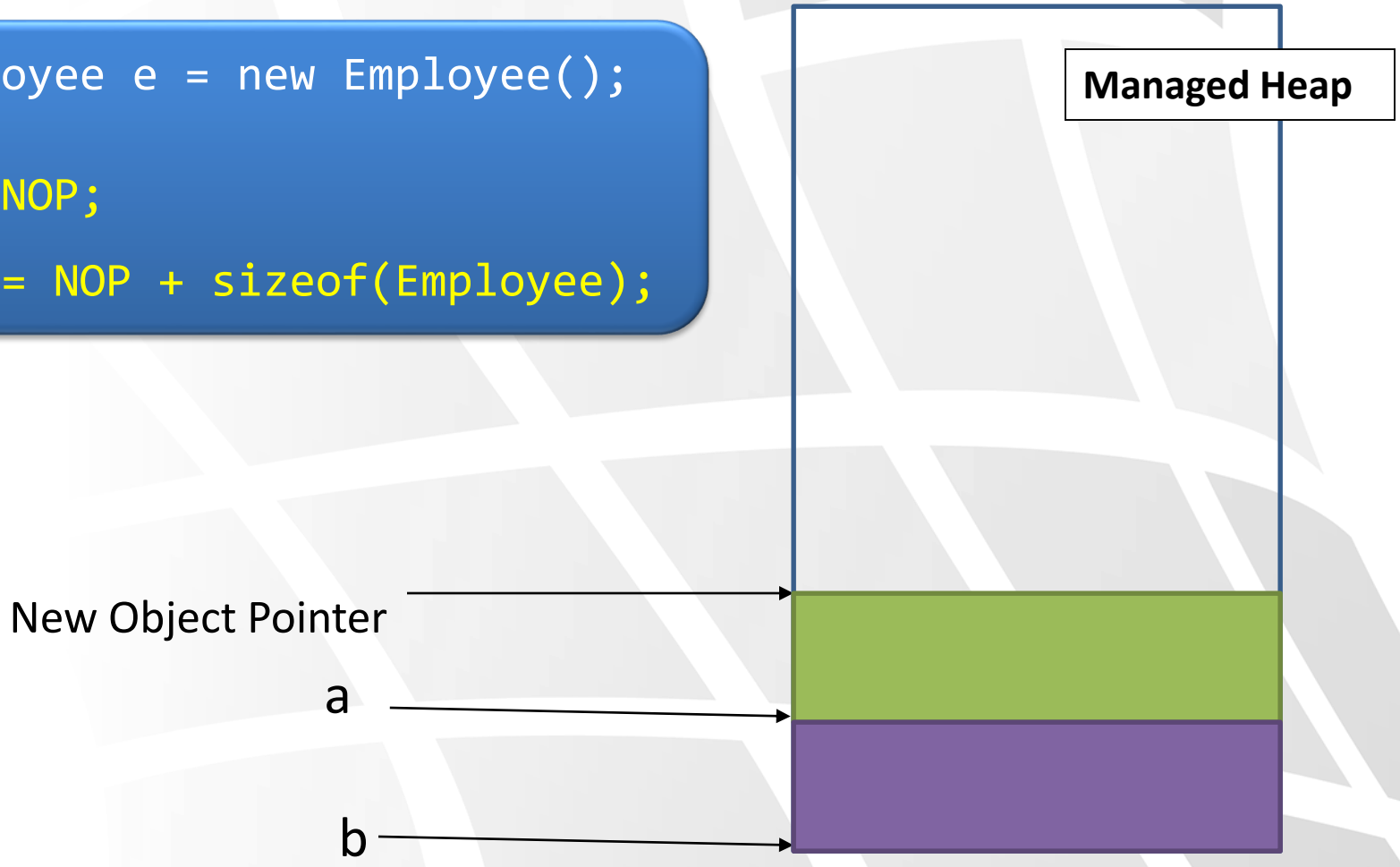
- A **managed heap** is created on initialization
- A pointer points to the next object (**NOP**)
- An object **allocation increments** the pointer and returns the previous address

Object Allocation and the Managed Heap

```
Employee e = new Employee();

e = NOP;

NOP = NOP + sizeof(Employee);
```

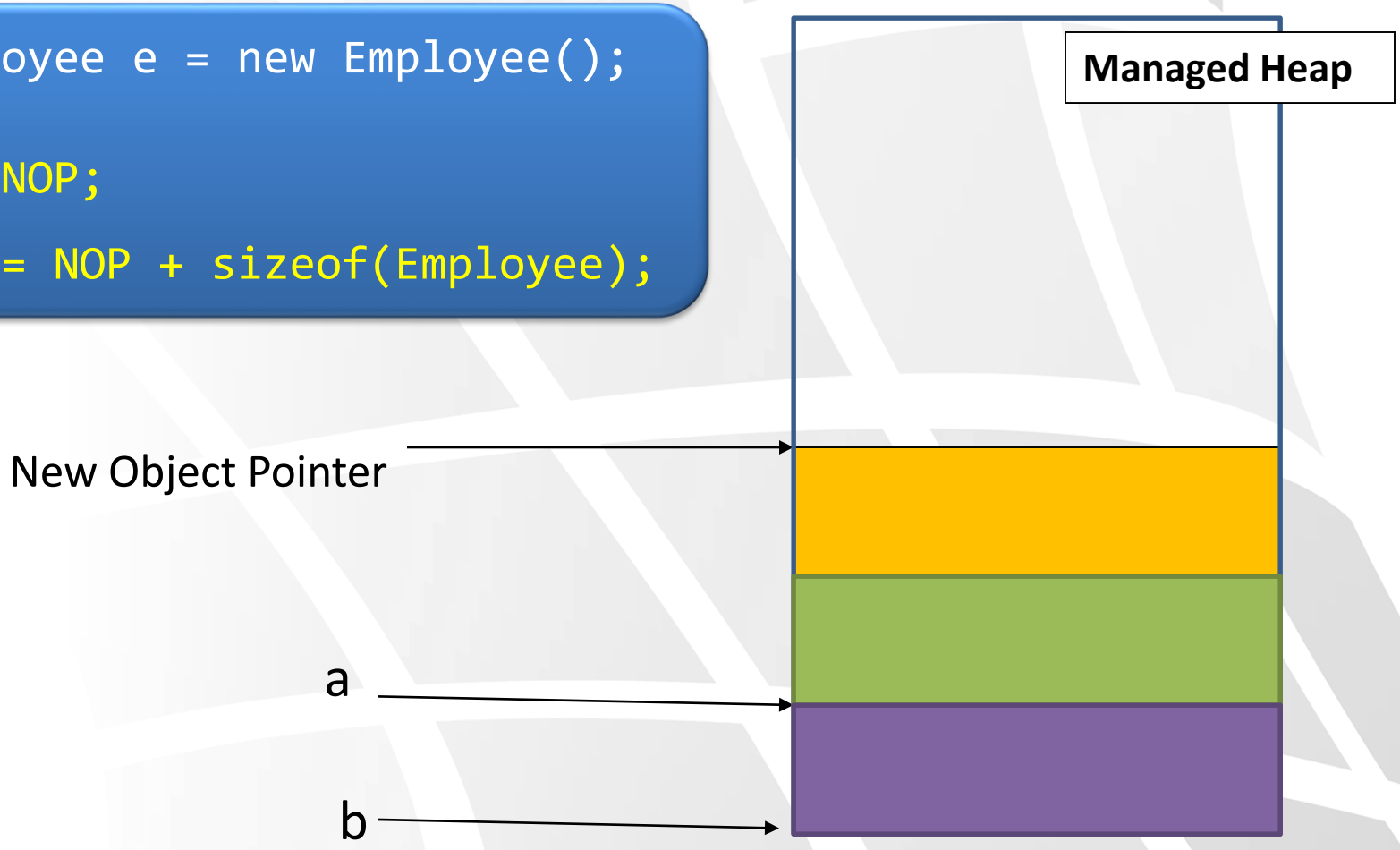


Object Allocation and the Managed Heap (contd.)

```
Employee e = new Employee();

e = NOP;

NOP = NOP + sizeof(Employee);
```



When Does a GC Occur?

- In *this* model, a GC occurs when memory runs out

- The GC has to **free memory** (**Sweep**) by **detecting unreferenced** objects (**Mark**)

How does it detect that an object is unreferenced?

Mark Phase

- The GC builds a **graph** of all **reachable objects**
- The **rest** is considered "**garbage**"
- To extend an object's lifetime, use the **`GC.KeepAlive`** method

Sweep Phase

- The GC **compacts** the **heap** by moving live objects close together
- The GC **updates** the next object pointer (**NOP**)

Mark Phase – Roots

- An application has a set of **roots**
  - **Static** object references
  - **Local** objects in currently active methods
  - Other types (GC handles, finalization queue, …)
- Where does **objRef** stop being an active root?

In Release Mode

In Debug Mode

```
static public void Main() {
        object objRef = ...;
        int i = objRef.GetHashCode();
        PerformLengthyCalculation(i);
}
```

Local Roots: System.Threading.Timer

Demo

Generations

- A **full GC** is extremely **expensive**!
  - Linear in # of referenced objects
- The **heap** is **divided** into **generations**
  - This enables a partial collection process

Generations: Assumptions

- Collecting a portion of the heap is **faster** than collecting the whole heap
- The **newer an object** is, the **shorter** will be its **lifetime**
- The **older an object** is, the **longer** will be its **lifetime**

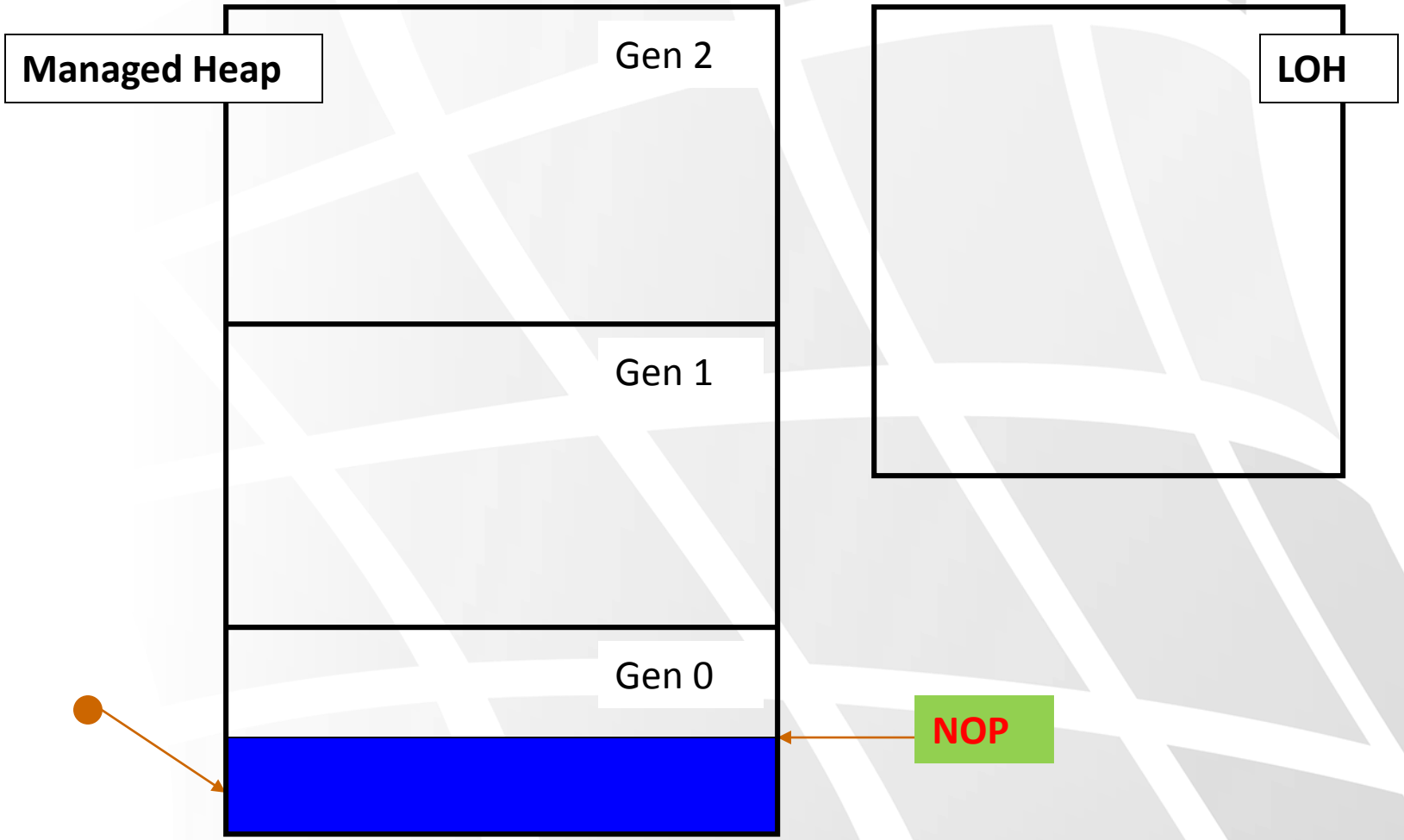
Allocation and Promotion

- New objects are allocated in **generation 0**
- When generation 0 fills, a GC occurs **in generation 0**
- Survivors are promoted to **generation 1**
- And so on – up to **generation 2**

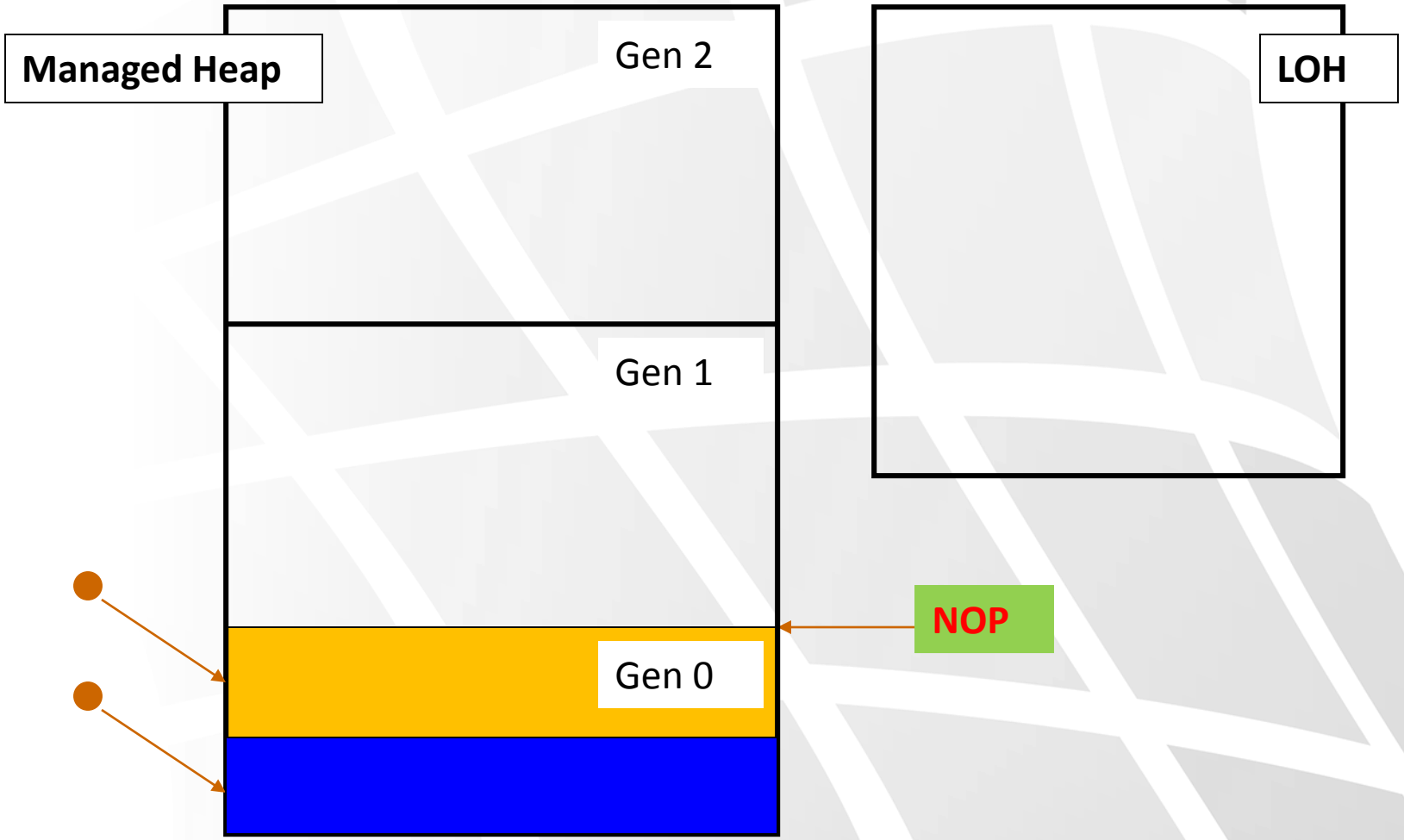
Generations – Collection

- **Gen 0** and **gen 1** collections are **frequent** but **fast**
- **Gen 2** collections are **slow** but **rare**
- **Large objects** (>85KB) are managed in a separate Large Object Heap (**LOH**)

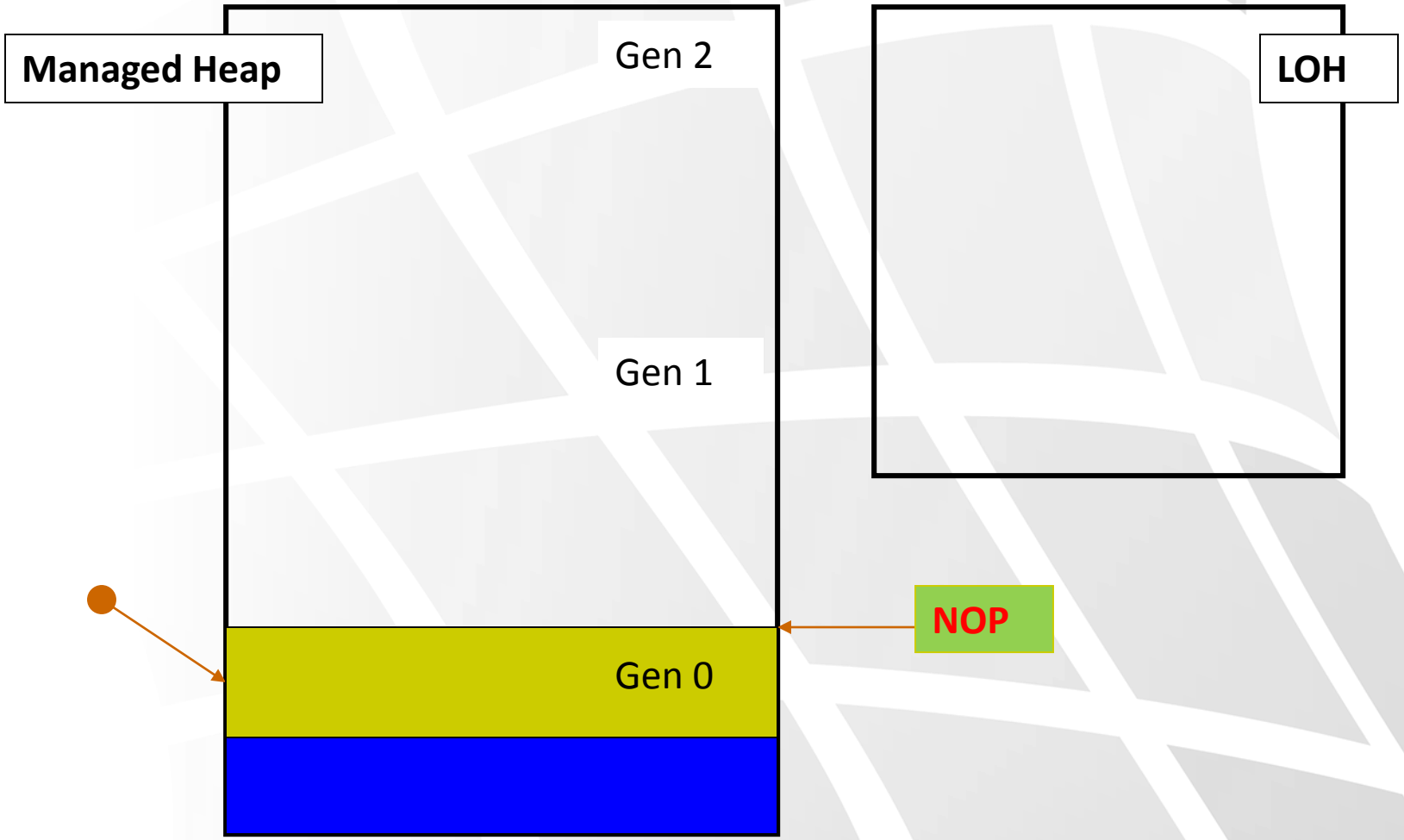
Generations Illustrated



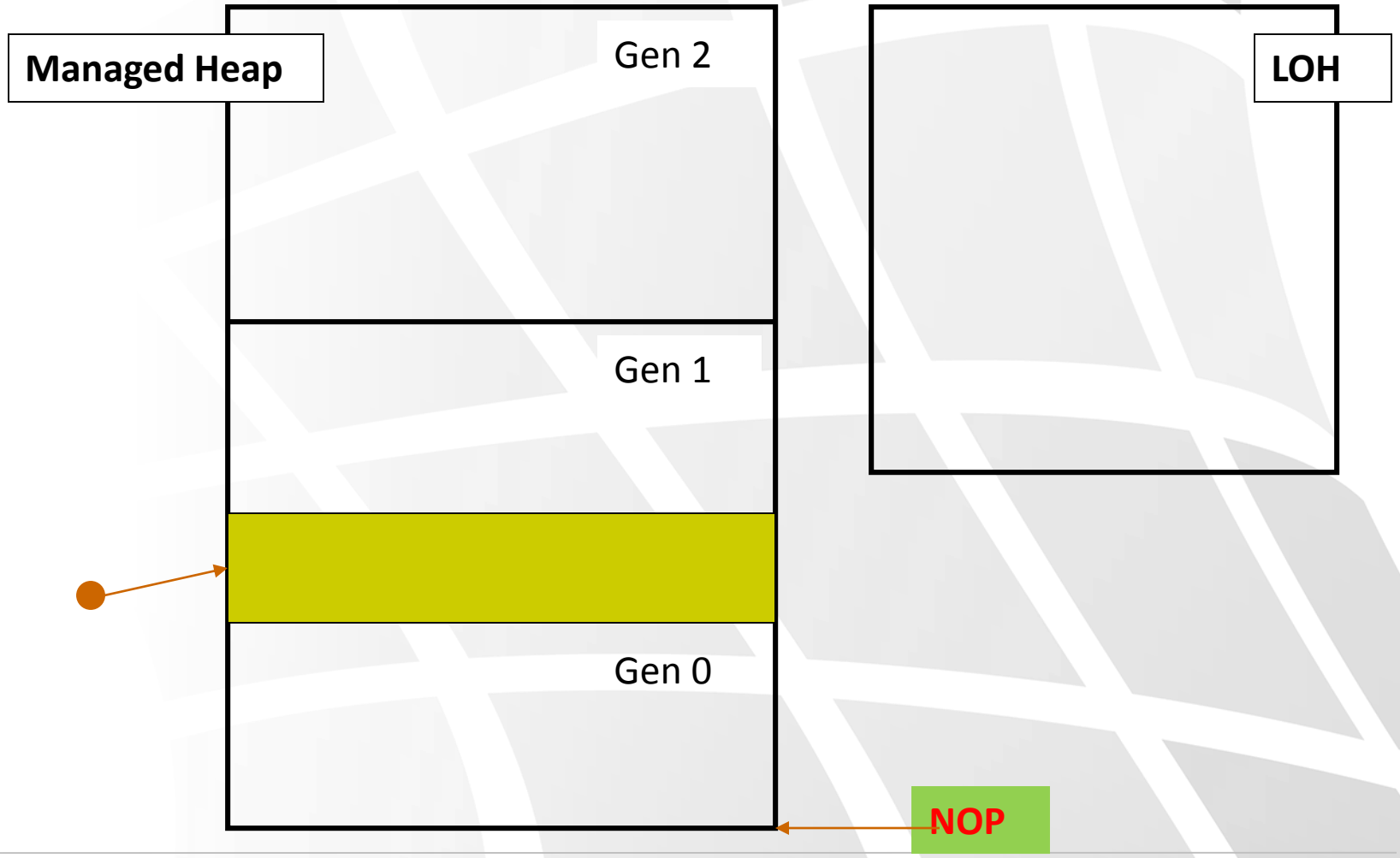
Allocations Are Made



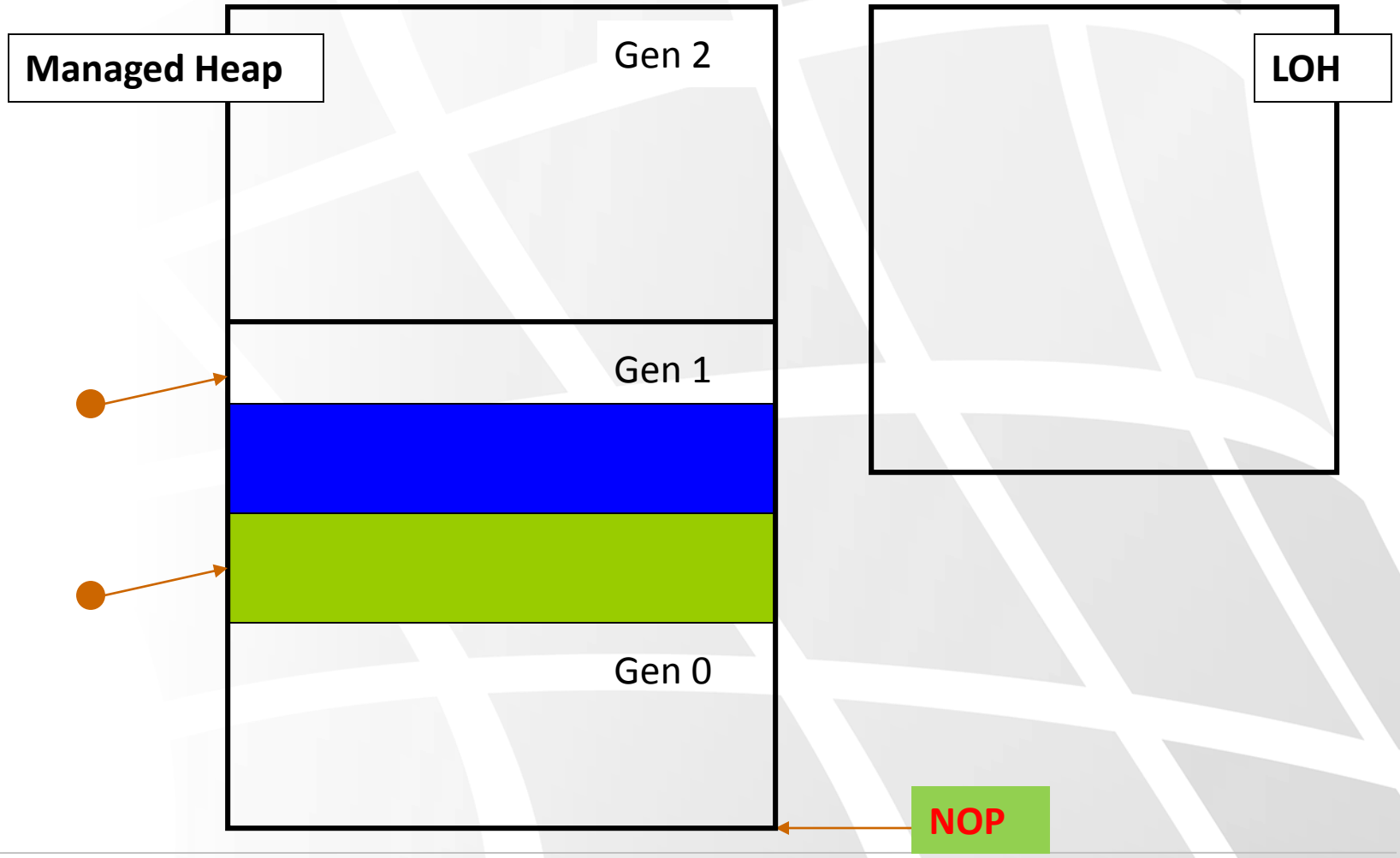
Generation 0 Fills



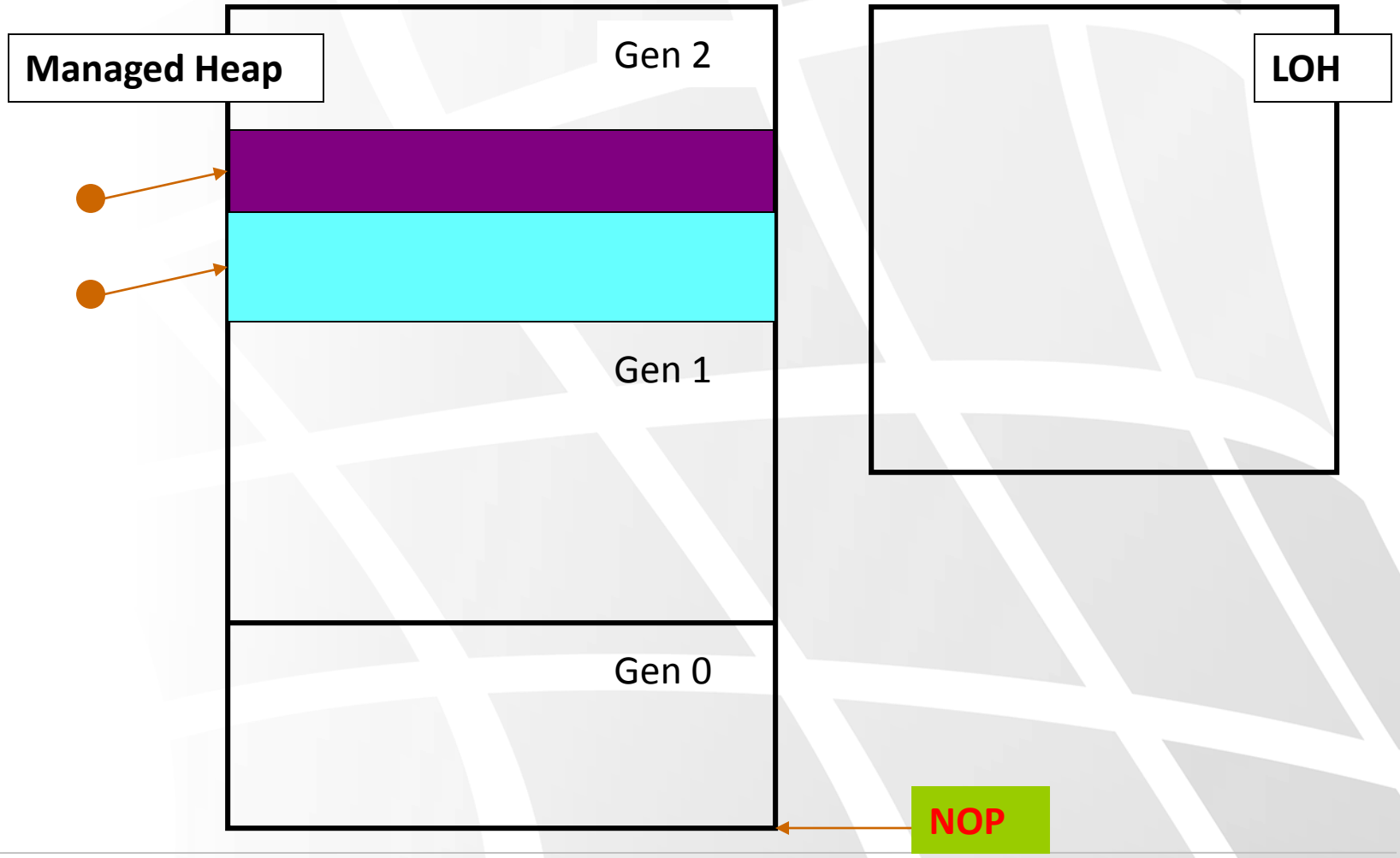
GC Occurs in Generation 0



Generation 1 Fills



GC Occurs in Generation 1



GC and Thread Suspension

- During GC, all **managed threads** may be **suspended**
- Threads running **unmanaged** code can **still execute**

GC Flavors

- GC Flavors optimize GC for specific application types
- Client GC (a.k.a. **Workstation GC**) is optimized for low latency
  - Concurrent GC, background GC improve latency
- **Server GC** is optimized for high throughput and scalability
  - Concurrent GC, background GC improve latency (.NET 4.5)

Choosing the Right Flavor

- Choosing between the two models is usually automatic
- Can be controlled through app.config

What are the pros and cons?

- The current state can be read using **GCSettings**

Latency Mode and Collection Mode

- GCSettings.LatencyMode
  - GCLatencyMode.Interactive(WKS GC)
  - GCLatencyMode.Batch (SVR GC)
  - GCLatencyMode.LowLatency
    - Try to avoid Gen 2 GCs
  - GCLatencyMode.SustainedLowLatency (.NET 4.5)
    - try to avoids full blocking GCs

Interacting with the GC

- The **System.GC** is the framework's static class which represents the GC

Informational Methods	Control Methods
GC.GetTotalMemory	**GC.Collect - do not use it!!!!**
GC.GetGeneration GC.MaxGeneration	GC.AddMemoryPressure GC.RemoveMemoryPressure
GC.CollectionCount	GC.WaitForPendingFinalizers GC.SuppressFinalize GC.ReRegisterForFinalize

Collect

- `GC.Collect(int, GCCollectionMode)`
  - `GCCollectionMode.Default, Forced`
  - `GCCollectionMode.Optimized`

- **Do not do it at home!!!**

Notification That a GC Occurs

- In .NET 3.5 SP1, a GC Notification API was added to the framework
- **GC.RegisterForFullGCNotification** and friends
  - Operational only in **non- concurrent** mode

GC Notifications

Demo

Weak References

- A large object is rarely used
- Plan of action:
  - Allocate and keep alive
  - Allocate, use and destroy (repeat)

Weak References (contd.)

- Storing a *weak reference* to an object enables the GC to collect it
- A weak reference can be converted to a *strong* one if the object is alive
- Useful for any *service* that shouldn't keep the object alive

Weak References: Cache

Demo

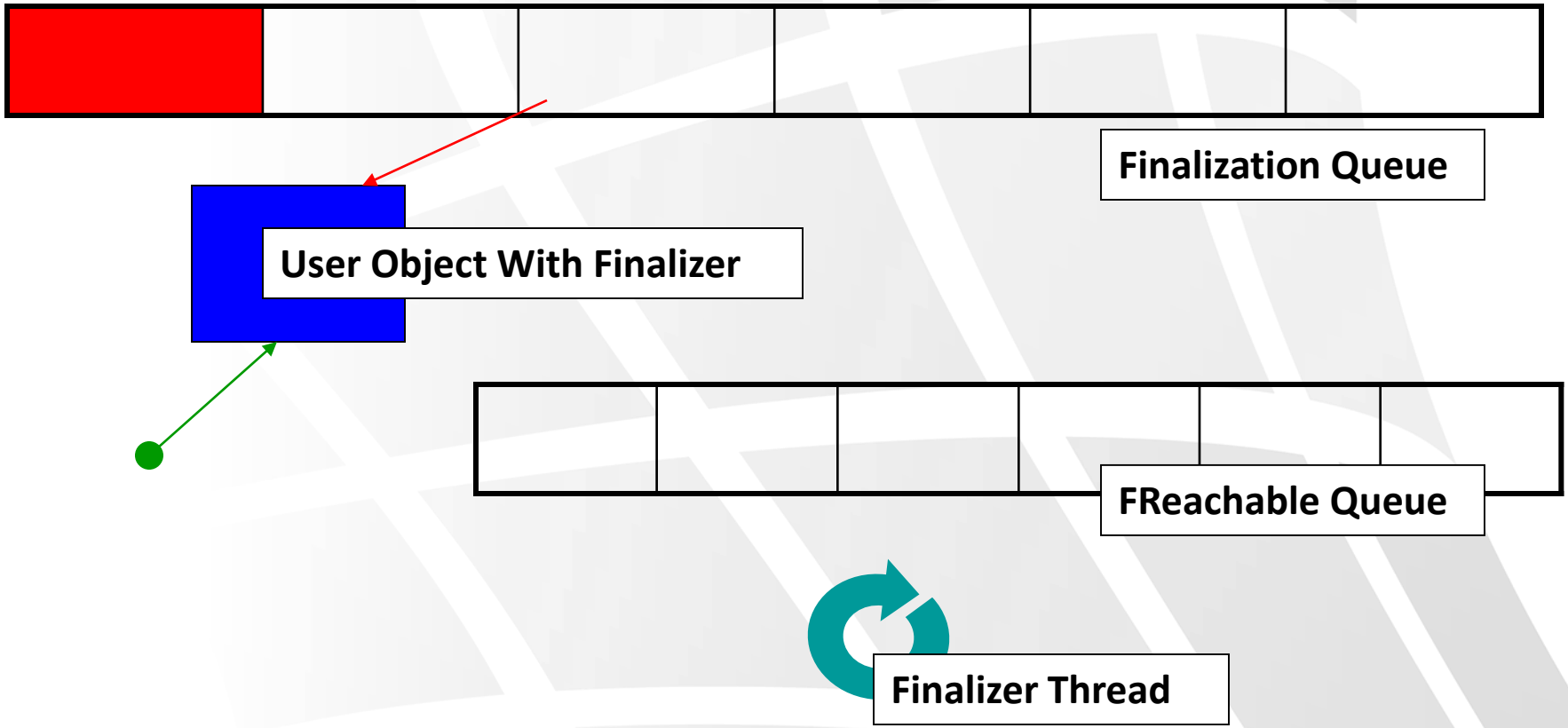
Finalization

- Unmanaged resources require finalization code
- A finalizer (**~ClassName**) is mapped to a method overriding **Object.Finalize**
- Finalization code is executed *at some point* after the object becomes unreachable

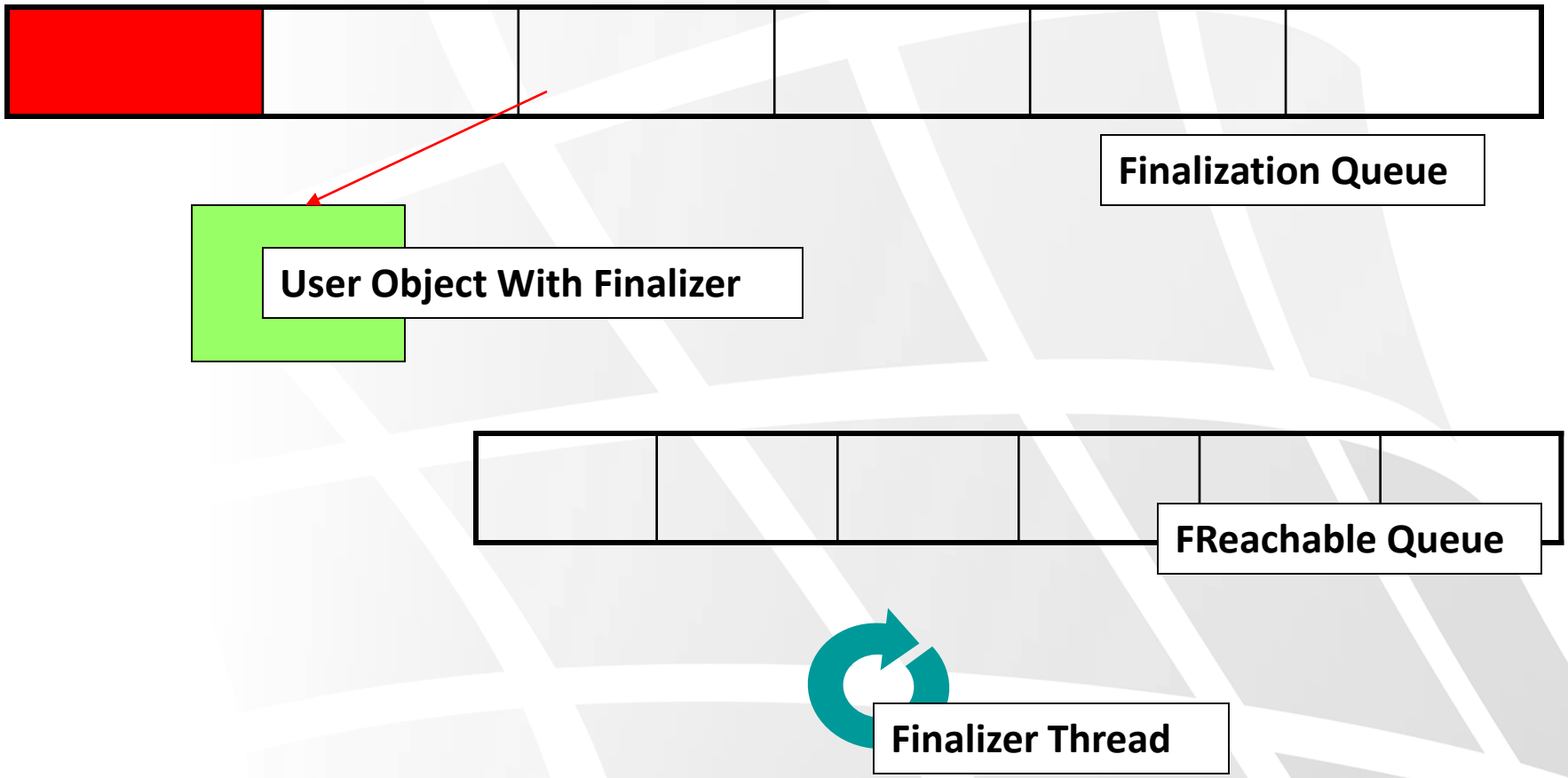
Finalization Internals

- Finalizable objects start in the **finalization queue**
- When GC finds the object unreachable it is moved to the **freachable queue**
- The **finalizer thread** executes its finalization method
- At the next GC, the object is reclaimed

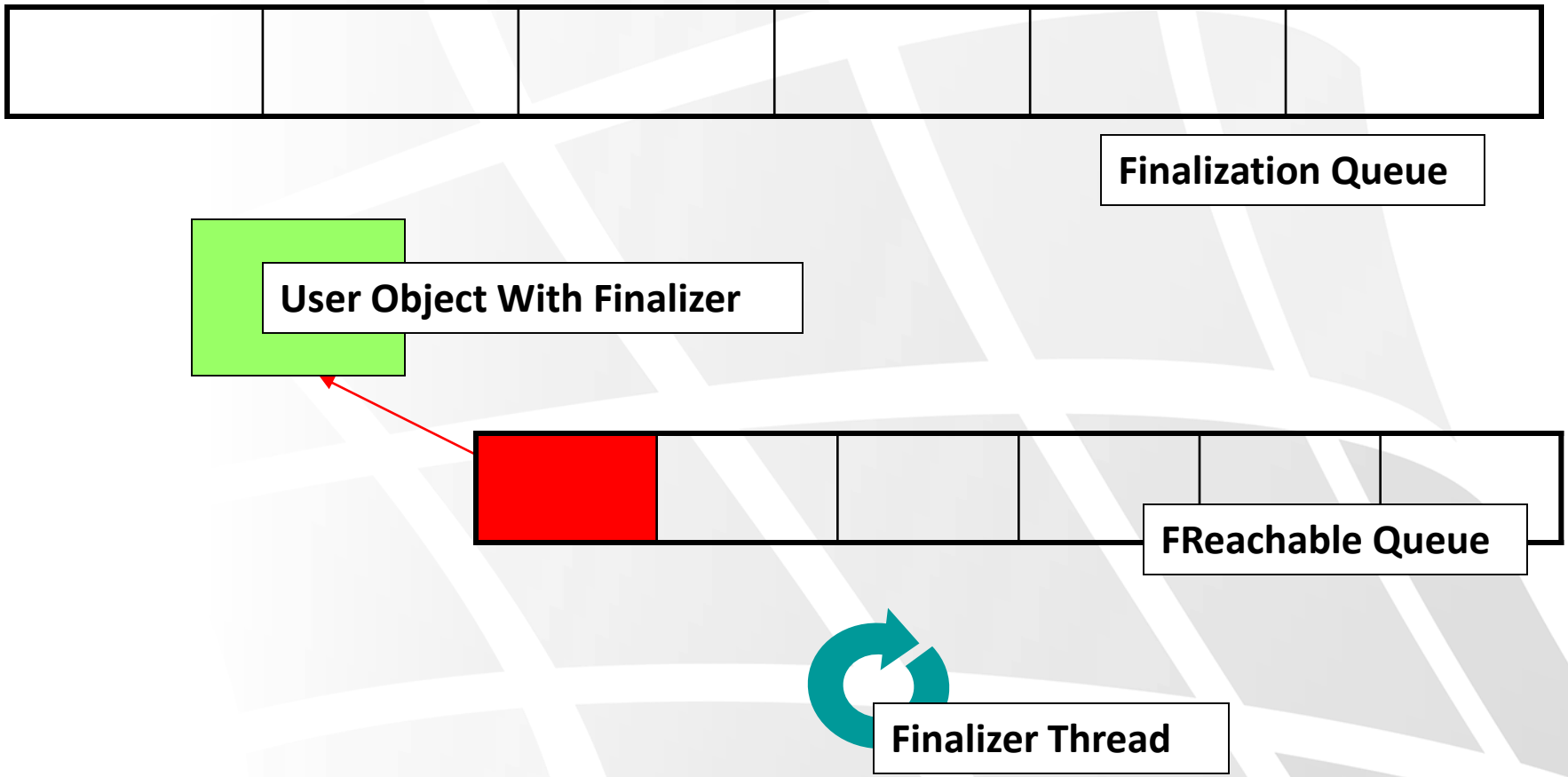
Finalization Illustrated



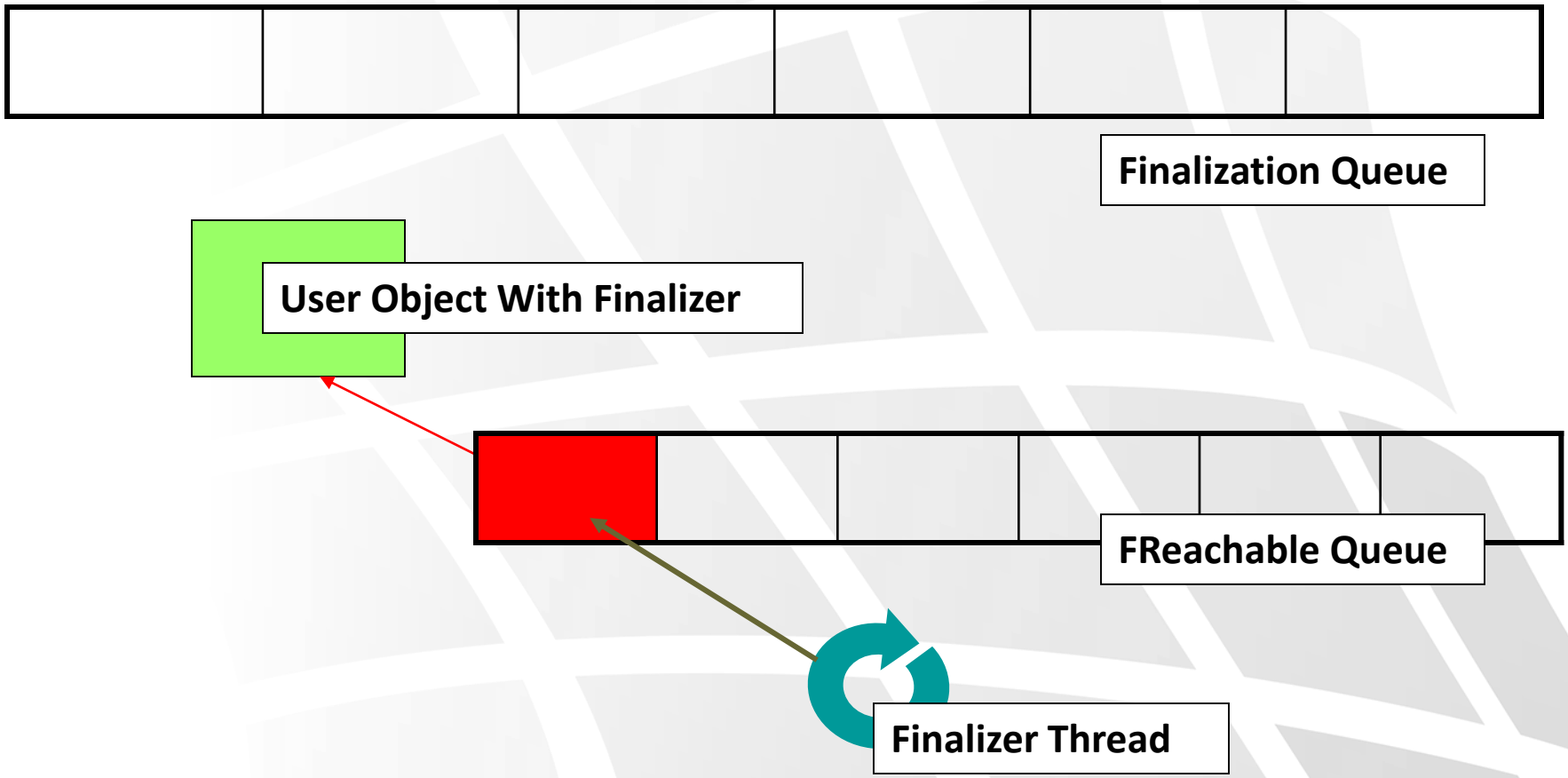
Object is No Longer in Use



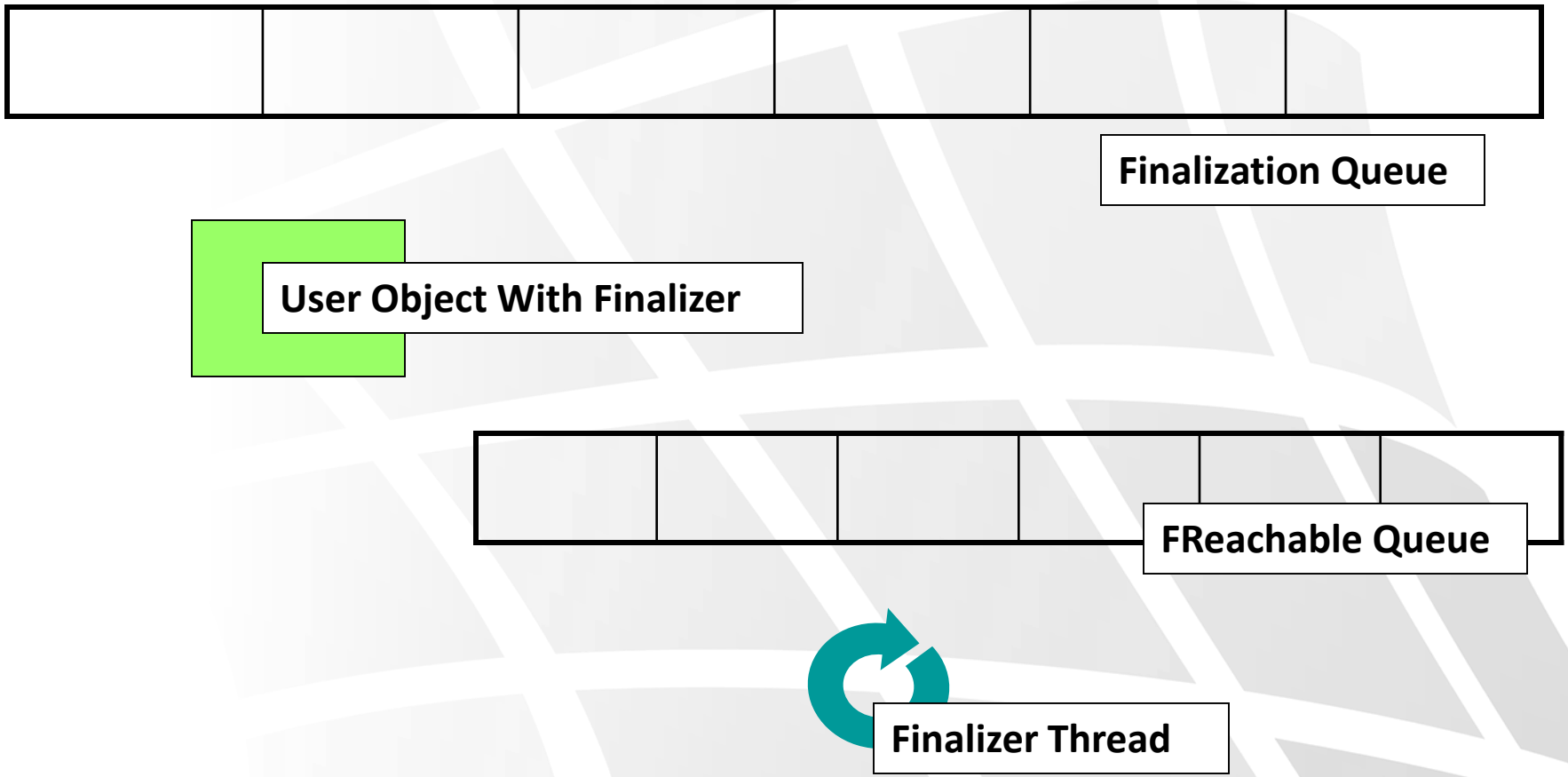
GC Occurs


Finalization Queue
User Object With Finalizer
FReachable Queue
Finalizer Thread

Finalizer Thread Wakes Up



Finalizer Is Done



Another GC Occurs

Finalization Queue

FReachable Queue

Finalizer Thread

Finalization Pitfalls: Memory Leaks, Race Conditions, Deadlocks

Demo

Avoid Finalization If Possible

- Finalization has terrible performance
- Finalization can cause correctness problems

How can we obtain **deterministic finalization**?

The Dispose Pattern

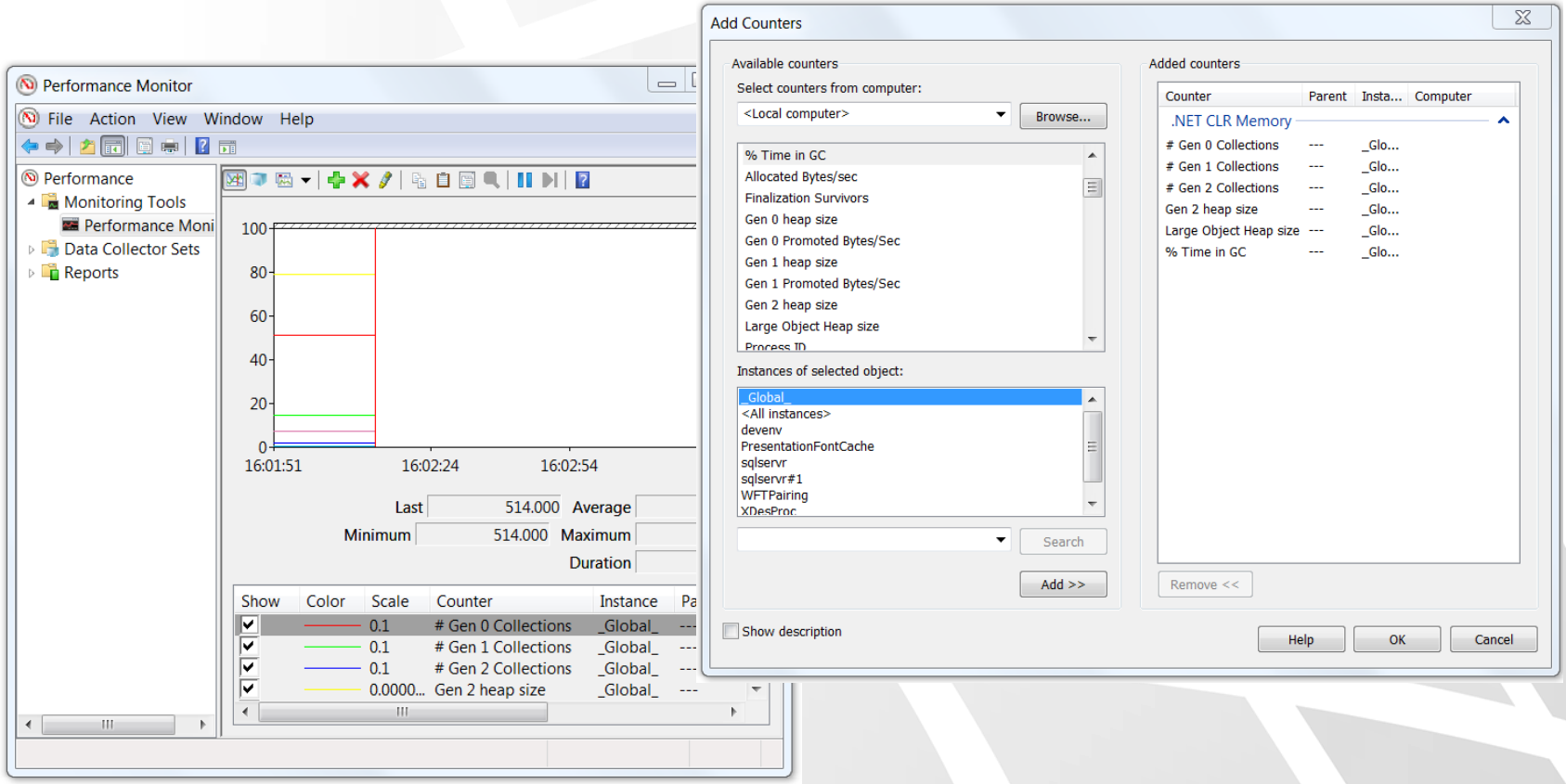
- Implement **IDisposable**
- A single method called **Dispose**
- Perform finalization work in that method
- To prevent double deletion (and effect on the finalization performance), call **GC.SuppressFinalize**

Weak Timer

Lab

Diagnostic

- Use PerfMon counter to trace the GC behavior



Summary

- Overview of memory management
- Garbage collection first steps
- GC flavors
- Generations
- Interacting with the GC
- Weak references
- Finalization and Dispose
- Lab

Questions