.NET Memory Management

Part2

Speaker: Ivan Kupriianov

Position: .NET Software Engineer

Email: ivan_kupriianov@epam.com

Agenda

- Managed & Unmanaged code. System resources
- Stack memory deallocating
- Garbage Collection in .NET

Why to use automatic memory allocation & garbage collection?

- Frees developers from having to manually release memory.
- Efficiently allocates objects in the managed heap.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future objects allocations.
- Provides memory safety by making sure that an object cannot use the content of another object.

Managed & Unmanaged Resources by respectful resources

- To put it very simply, managed code is just that: code whose execution is managed by a [Common Language] runtime. ©MSDN
- Managed resources - resources that after being abandoned CLR knows how to manage. ©StackOverflow comment

But this sounds very abstract, doesn’t it? What are the resources at all?

System resources

- System resources – any kind of computer native resources, a programmer can request from operating system. E.g.:
  - Processors
  - Memory
  - Timers
  - Network connections
  - File descriptors
  - etc.

Managed & Unmanaged resources, based on system resources

- Managed resources – kind of resources, after being handled by CLR itself, **won’t** leave any open system resources.
- Unmanaged resources – kind of resources, after being handled by CLR itself, **will** leave some open system resources.

How does CLR manages resources?

- Stack memory is allocated and deallocated “on the spot” – in the thread stack. Typical data in stack: value types for methods and references to objects allocated in heap.
- Heap memory is deallocated with Garbage Collector. Typical data in heap: instances of classes.

What is the heuristic for the GC algorithm?

- It’s faster to compact the memory for a portion of the managed heap than for the entire managed heap.
- Newer objects have shorter lifetimes and older objects have longer lifetimes.
- Newer objects tend to be related to each other and accessed by the application around the same time.

What are the triggers for GC to start?

- The memory that’s used by allocated objects on the managed heap surpasses an acceptable threshold. This threshold is continuously adjusted as the process runs.
- The system has low physical memory. This is detected by either the low memory notification from the OS or low memory as indicated by the host.
- The GC.Collect() method is called.

How to know what to clean up?

- Let’s stop the execution and mark all the objects in use. A simplified algorithm would look something like this:

```
void Collect()
{
    List gcRoots=GetAllGCRoots();
    foreach (objectRef root in gcRoots)
    {
        Mark(root);
    }
    Cleanup();
}
```

```
Void Mark(objectRef o)
{
     if (!InUseList.Exists(o))
     {
         InUseList.Add(o);
         List refs=GetAllChildReferences(o);
         foreach (objectRef childRef in refs)
         {
             Mark(childRef);
         }
     }
}
```

But what are the GC Roots?

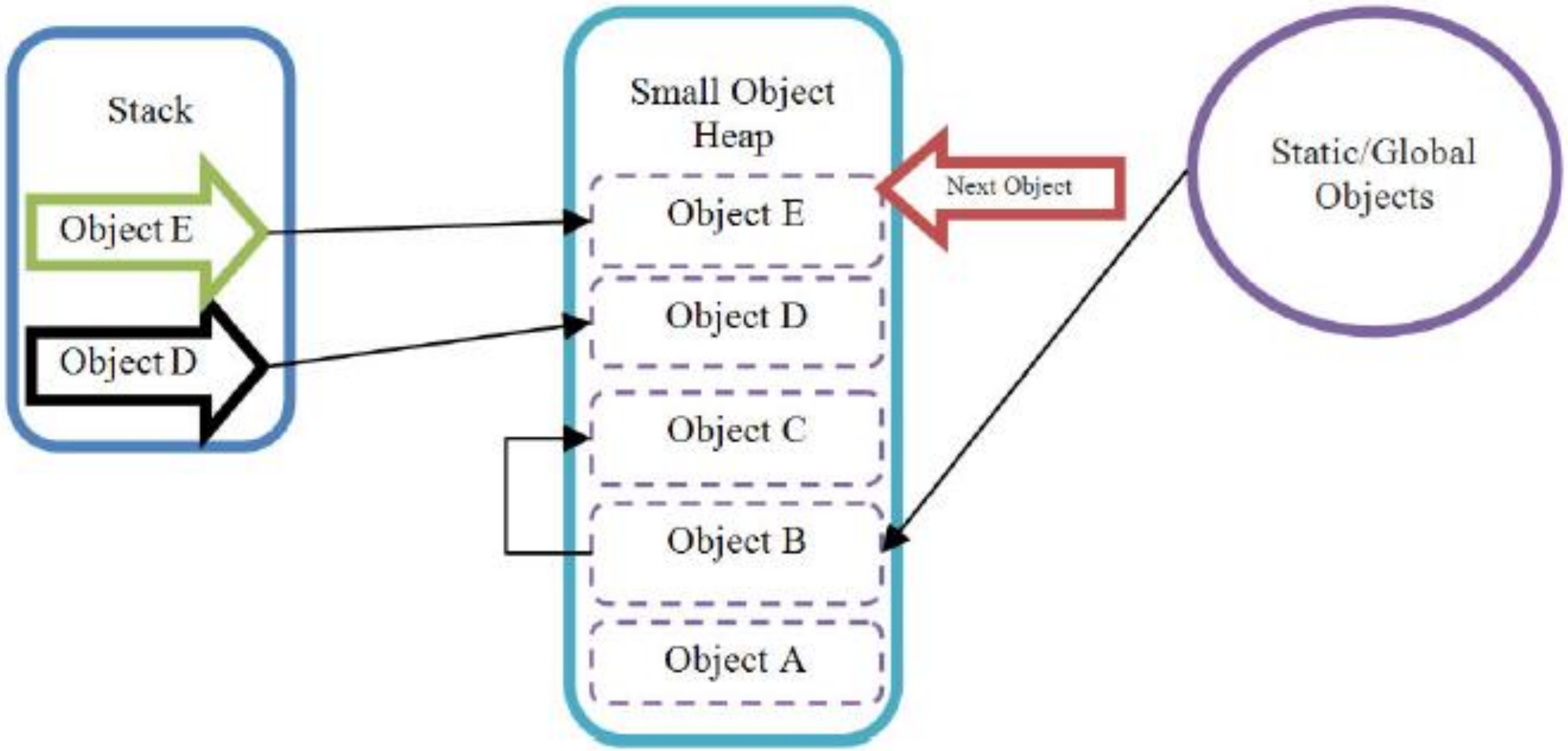
GC Roots

- Static fields
- Local variables and parameters on the threads' stack
- Object finalization references (*)
- CPU registers
- Interop references (.NET objects passed to COM/API calls)

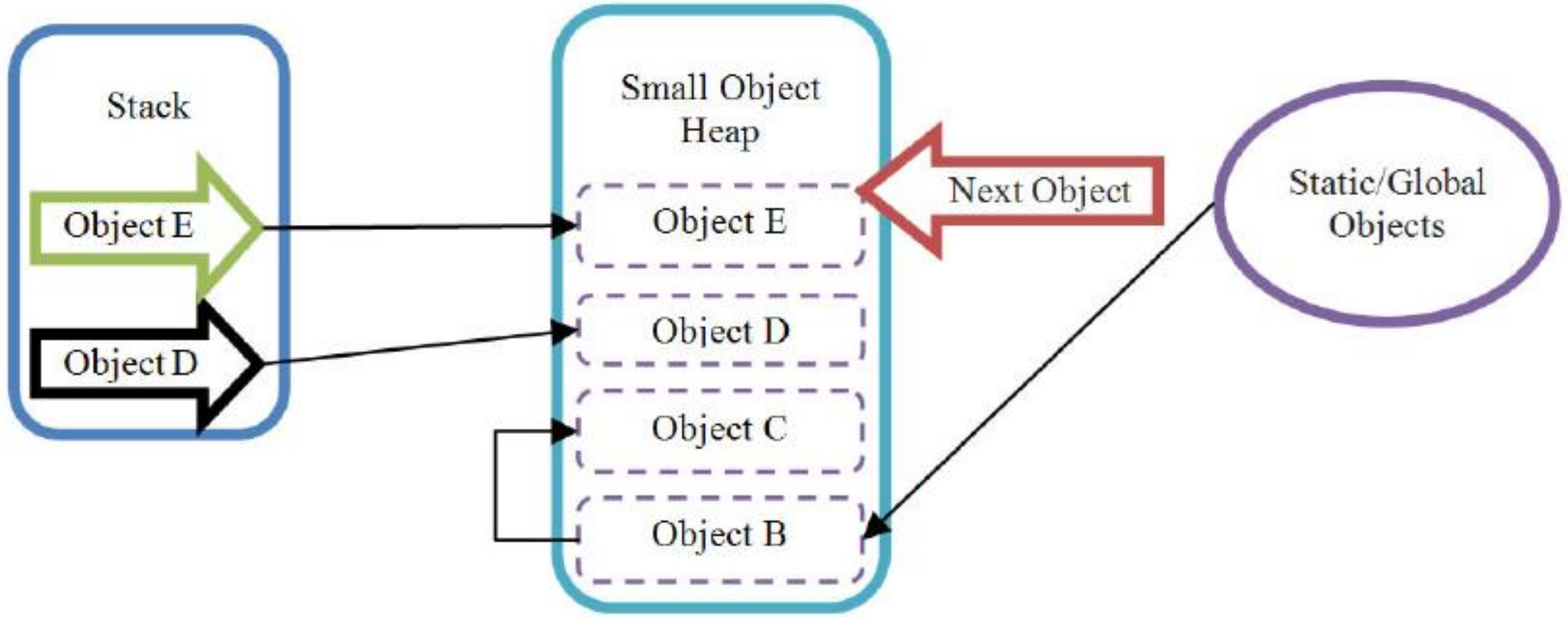
When GC knows what to clean it … just cleans

- The relocation phase – update of the existing references (in code e.g.)
- The compacting phase – compaction of objects in the heap to avoid heap segmentation

Managed heap before GC



Managed heap after GC



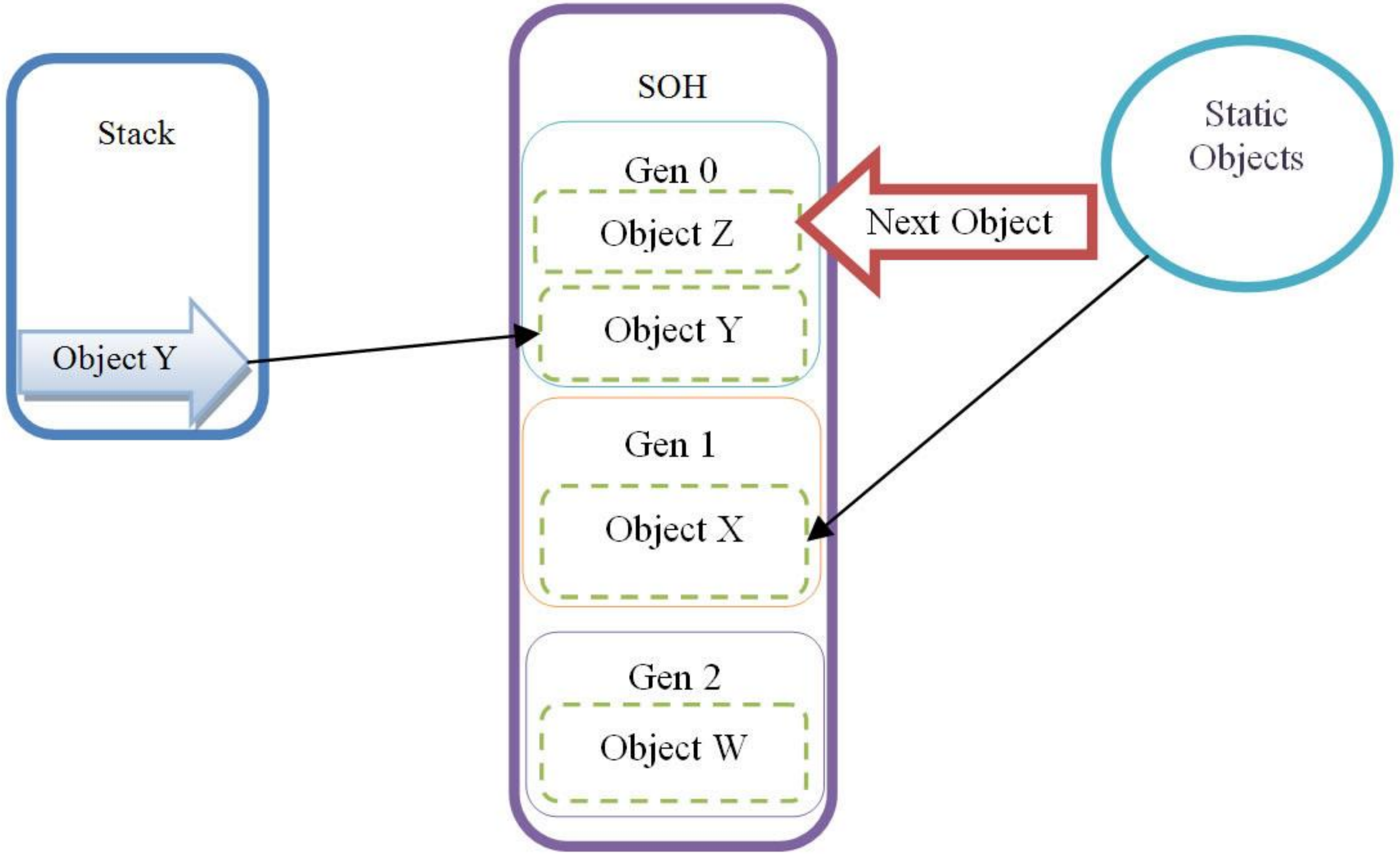
Problems of the current model

- P1: There are some objects that will live longer than others, much longer. Every collection GC needs to check all the objects if they are alive (and their internals). But processor time is precious!
- P2: What to do with big objects? Every garbage collection we need to move them. It takes lots of time also!

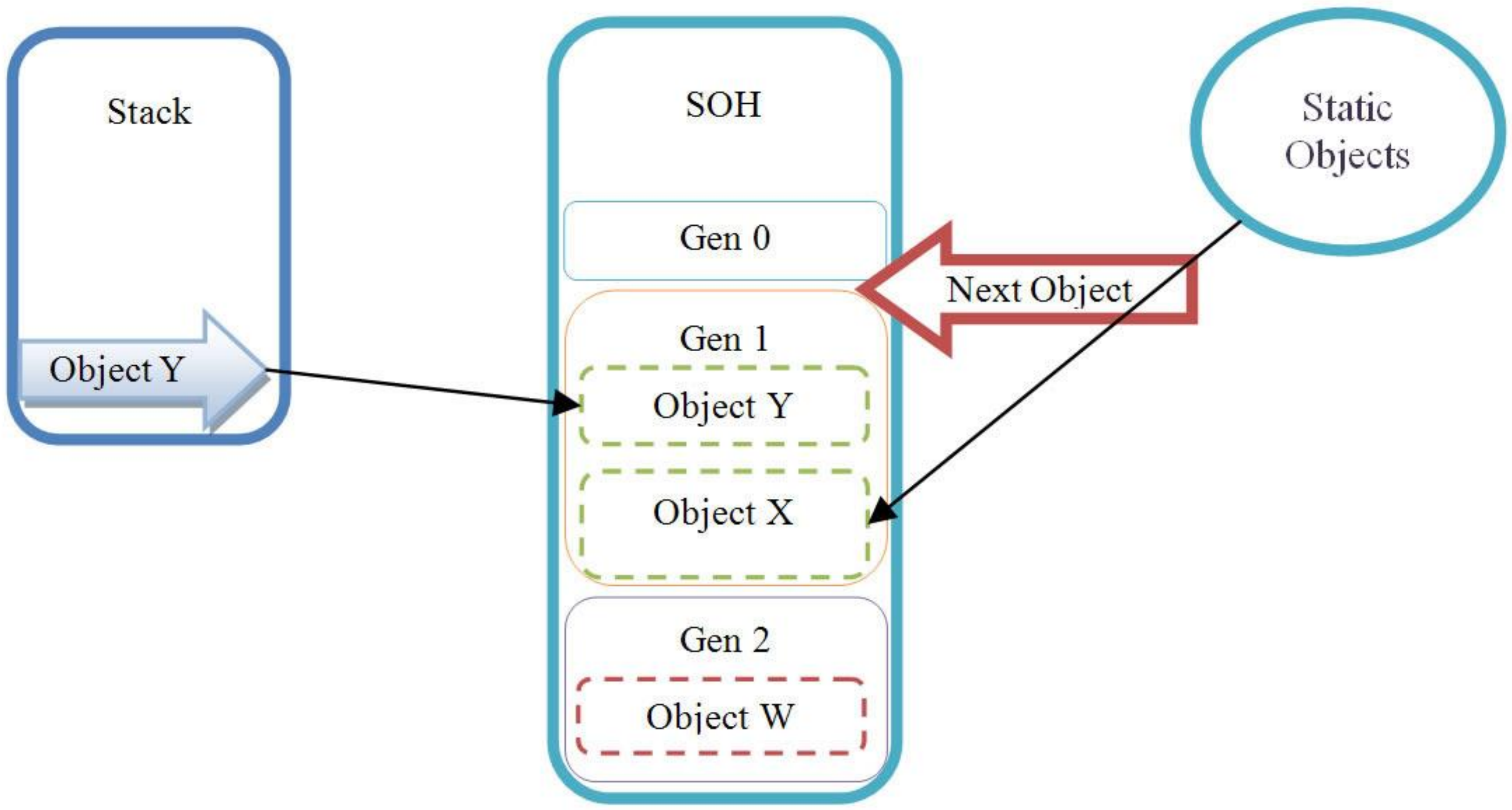
Solution for P1: Long-living objects

- Let’s introduce Generations 0, 1, 2;
- When an object has just been created, it is classified as a Gen 0 object, which just means that it's new and hasn't been inspected by the GC yet;
- Gen 1 objects have been inspected by the GC once and survived;
- Gen 2 objects have survived two or more such inspections;
- Gen 0 will be checked much more often than Gen 1, and Gen 1 will be checked much more often than Gen 2.

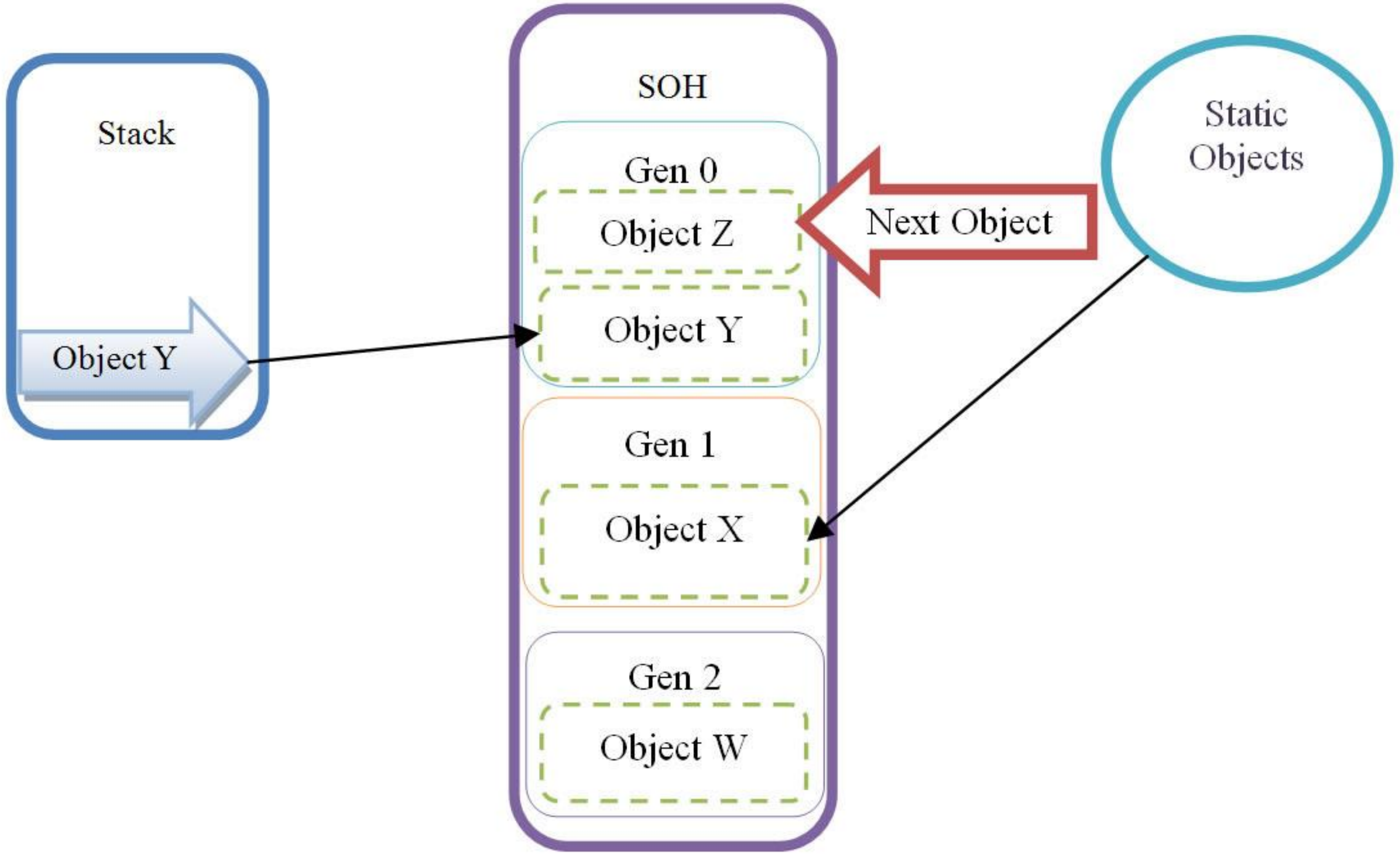
Managed heap before GC (revisited)



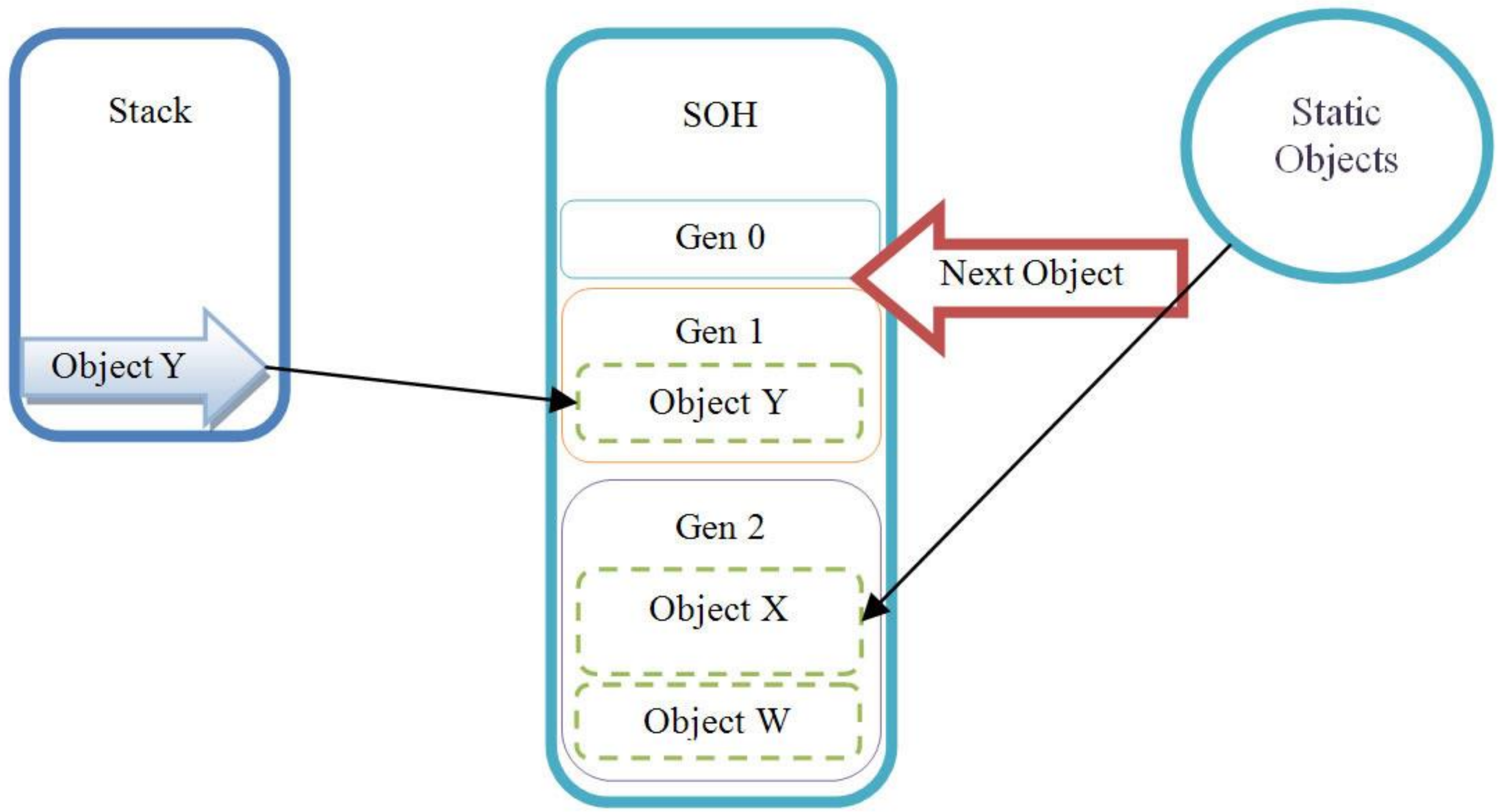
Managed heap after GC Gen 0



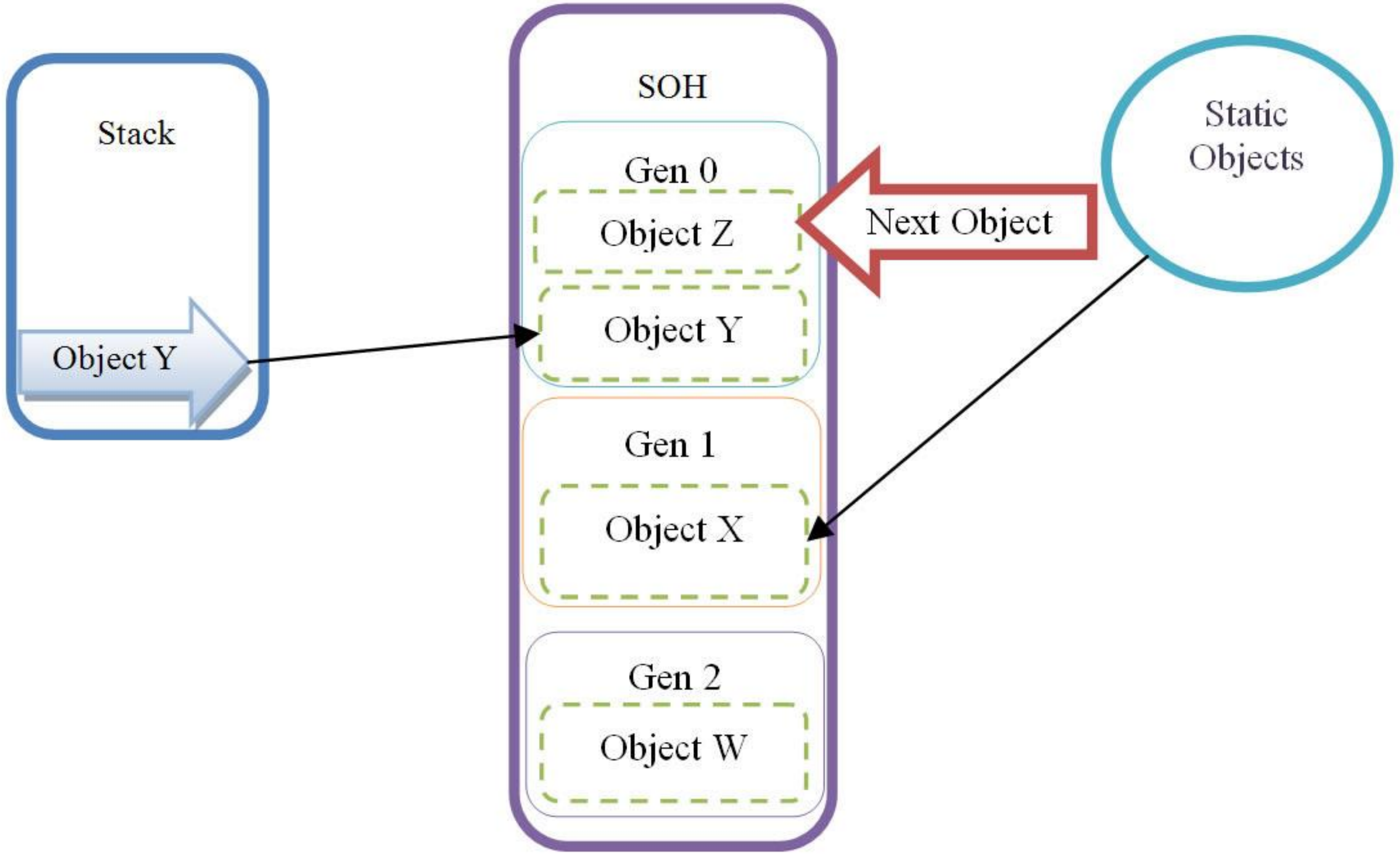
Managed heap before GC (revisited 2)



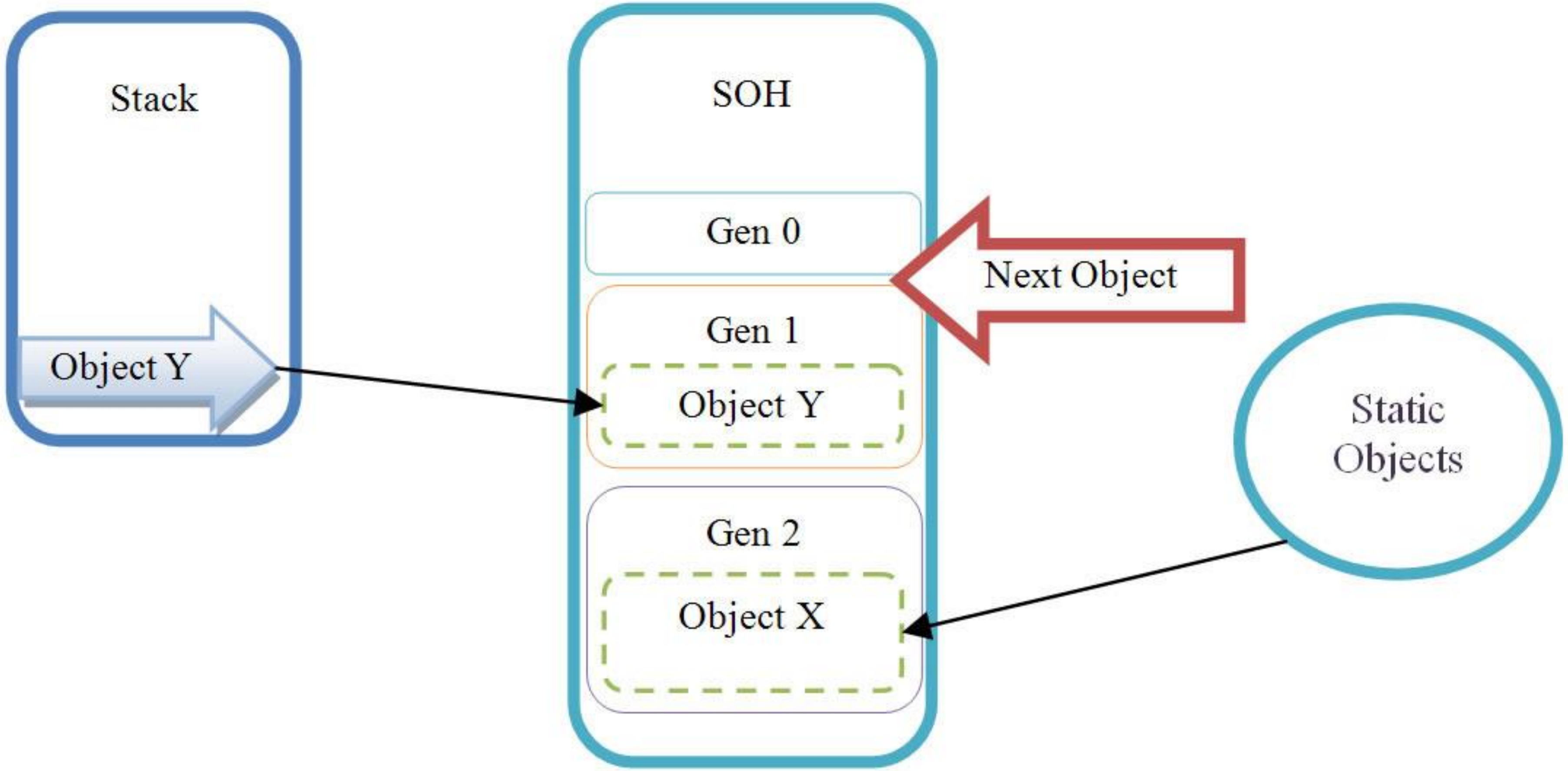
Managed heap after GC Gen 1



Managed heap before GC (revisited 2)



Managed heap after GC Gen 2



One of the triggers for GC - thresholds

- Gen 0 hits ~256 KB (~2^18 B)*
- Gen 1 hits ~2 MB (~2^21 B)*
- Gen 2 hits ~10 MB (~2^23 B)*

*****_These are thresholds for the newly started app_

Solution for P2: Big ones (85 000 B+)

- Let’s introduce a separate heap – Large Object Heap, that will contain big objects only, will mark them as Gen 2 objects and will not fragment this heap.
- Moving is hard. In this heap we will track free space fragments, and when a new big object is created, we will try to find the smallest fragment that we can fit in. Otherwise – we’ll take fragment “on the top” of the heap.

Great! No? Unmanaged resources!...

- Do you remember? Not everything is managed by the CLR. Imagine you need to write you own data connection provider to your new state-of-the-art database server, which is, obviously, not a CLR managed object.
- And, basically, all CLR managed resources are system resources under the hood. So, how do we make our own?

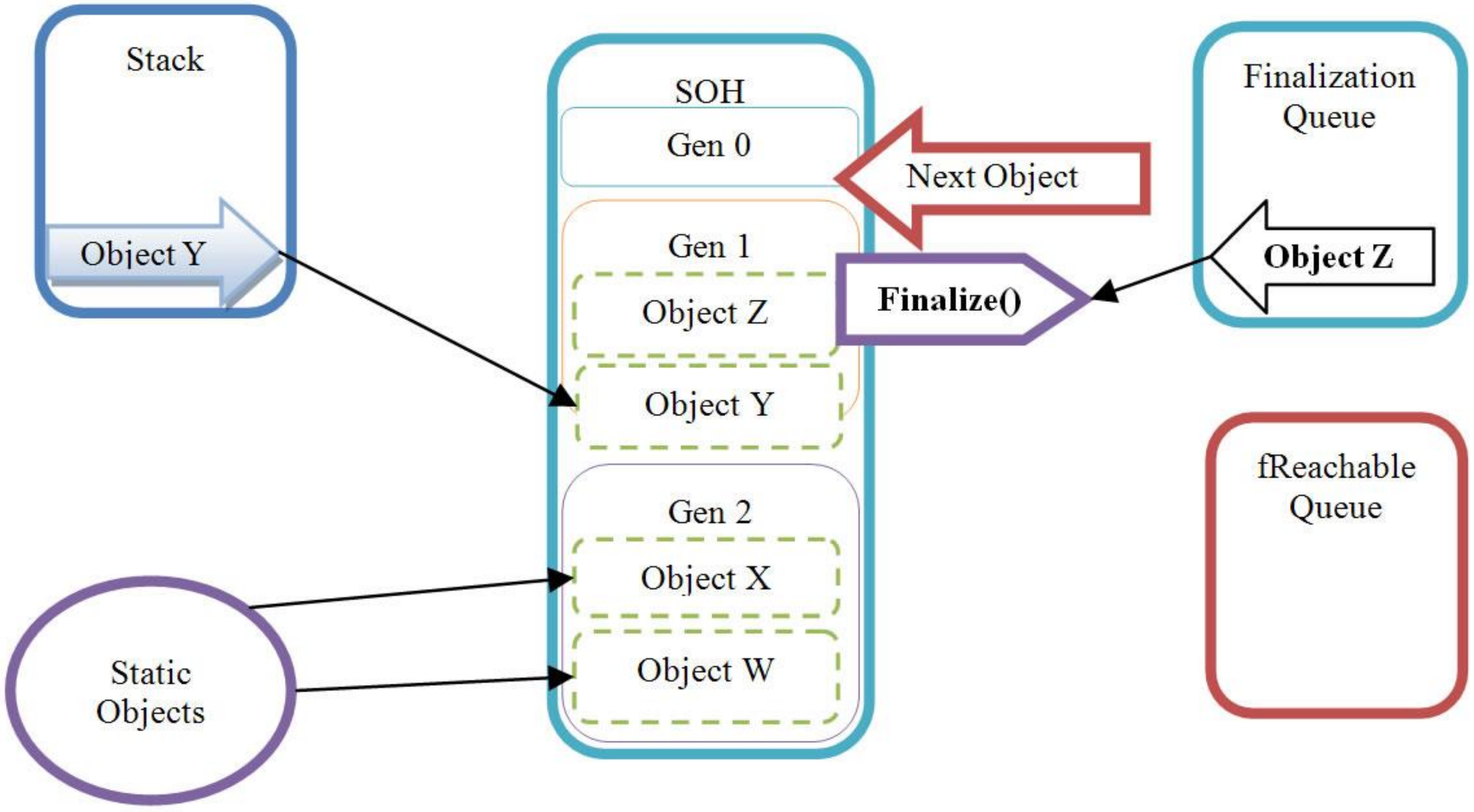
Let’s finalize it!

- Let’s introduce new class members - finalizers. They are protected virtual methods dedicated to freeing the unmanaged resources used by a class. And the programmers’ responsibility to write the cleaning code right by themselves.
- It’s not a destructor from C++. It will not be called at the moment when the object is not referenced anymore, it is different time.
- You are not able to call a finalizer by yourself. It is GC’s responsibility.

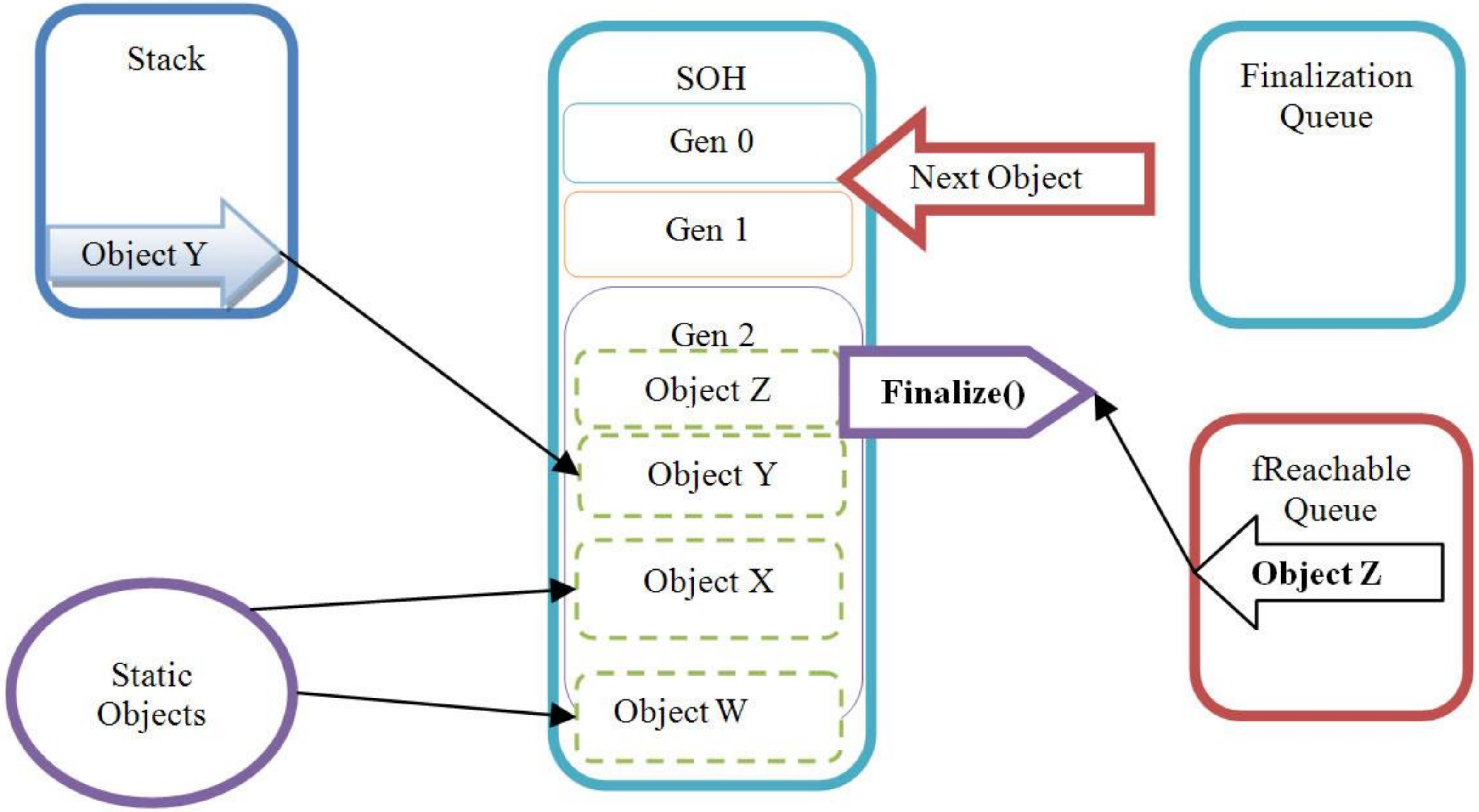
What it brings

- To call finalizer after the GC has ended, we need to keep the object and all its references in the memory (no one prevents you from referencing a field in finalizer, calling a method or access a field).
- The GC needs to know which objects have a finalizer – we need to keep track of them. On creation of the object, we will add them to Finalization Queue. When all references to the object are lost, and in normal situation it should have been killed, it will be promoted to the next Gen. It’s references will be dequeued from Finalization Queue and enqueued to fReachable Queue.
- The dedicated thread will run periodically and will call finalizers of the objects from the fReachable Queue.

GC pipeline with Finalizer (before collection)



GC pipeline with Finalizer (after collection)

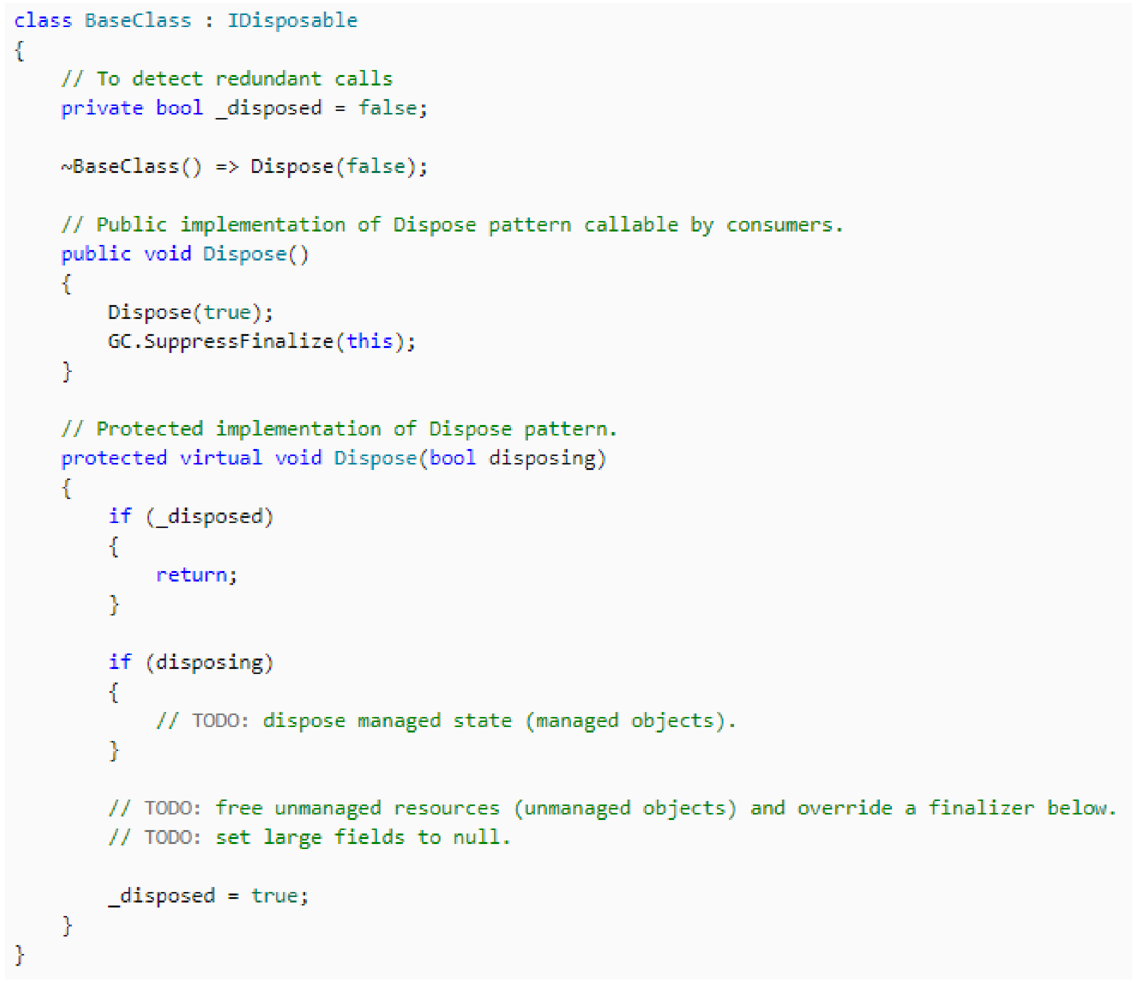


Problem

- If we take it wide, there are lots of unmanaged resources under the hood. Almost everything, basically. Then, if there any use of the GC if it clears things so slowly? Again, there is a solution!

Solution: Let’s dispose it! Disposable pattern

- Disposable pattern comes in hand. IDisposbale is only an interface. It should be used not only when implementing Disposing pattern.
- GC.SuppressFinalize(this); removes the object from the Finalization Queue and the object will not be promoted. Yay!
- Notice: protected virtual Dispose(bool) is for derived classes to override it correctly.



```
class BaseClass : IDisposable
{
    // To detect redundant calls
    private bool _disposed = false;

    ~BaseClass() => Dispose(false);

    // Public implementation of Dispose pattern callable by consumers.
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // Protected implementation of Dispose pattern.
    protected virtual void Dispose(bool disposing)
    {
        if (_disposed)
        {
            return;
        }

        if (disposing)
        {
            // TODO: dispose managed state (managed objects).
        }

        // TODO: free unmanaged resources (unmanaged objects) and override a finalizer below.
        // TODO: set large fields to null.

        _disposed = true;
    }
}
```

References

- Under the Hood of .NET Management by Chriss Farrell and Nick Harrisson
- https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/
- https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose