Fachhochschule Nordwestschweiz
Hochschule für Technik

CLR RESOURCE MANAGEMENT

Martin Kropp,Yves Senn
University of Applied Sciences Northwestern Switzerland

Learning Targets

- You
  - can explain how memory management under .NET works
  - know the .NET concepts for actively control resource management
  - can apply correctly the resource management concepts in an .NET

Content

- Resource Management
  - Automatic Garbage Collection (GC) in .NET
  - Explicit Resource Management
  - How GC works?

Automatic memory management

C# programmers don’t have to release allocated memory, the CLR takes care of it:
(better known as “Garbage Collection”)

→ `new Car(); new Car(); new Car(); ...`

Other languages (C99, for example) require explicit memory management:

```
Car* myCar = (Car*)malloc(sizeof(Car));
//…do something with myCar…
free(myCar);
```

Object creation

- C#

```
var c = new Car("Viper", 200, 100);
```

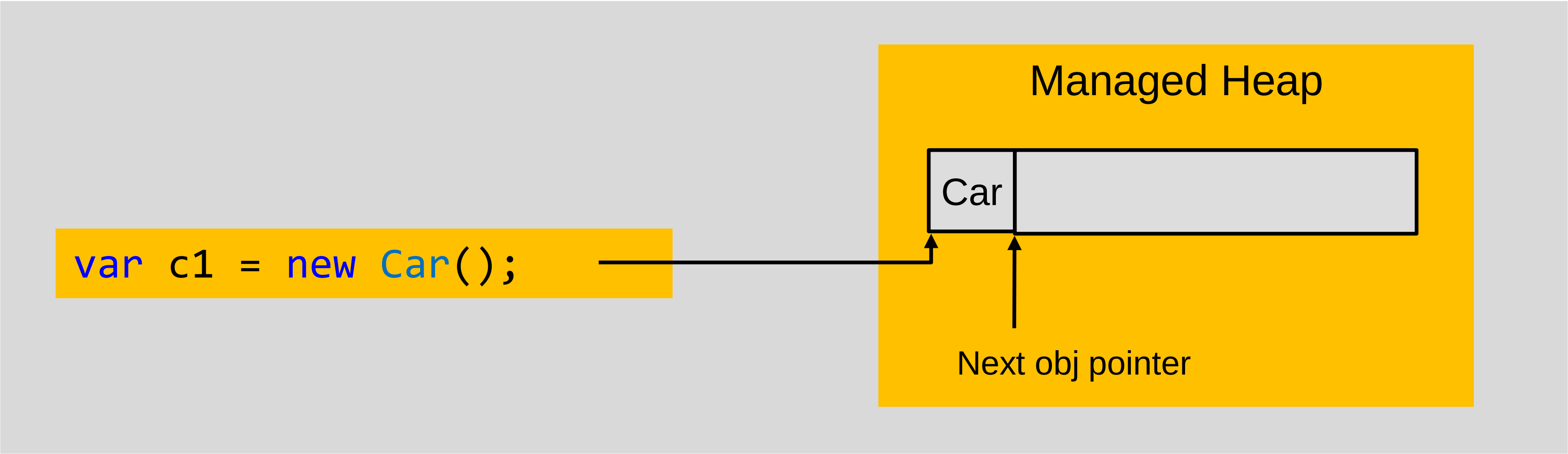
- IL

```
IL_000c: newobj instance void CilNew.Car::.ctor (string, int32, int32)
```

This `newobj` instruction:

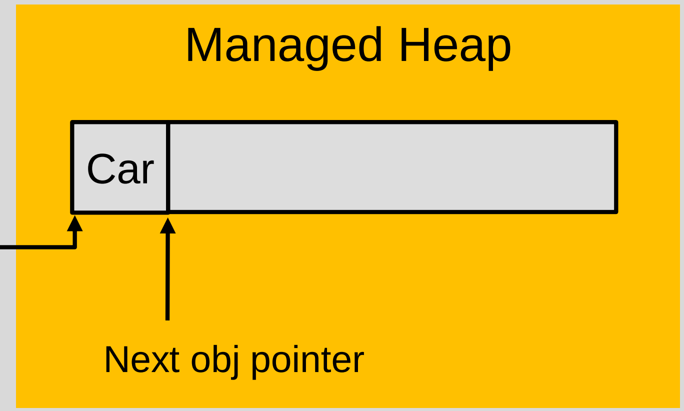
1. Calculates the total amount of memory required for the object
2. Ensures there's enough free room on the heap
3. Finds a suitable location for this new object on the heap
4. Returns a reference to the caller

The heap

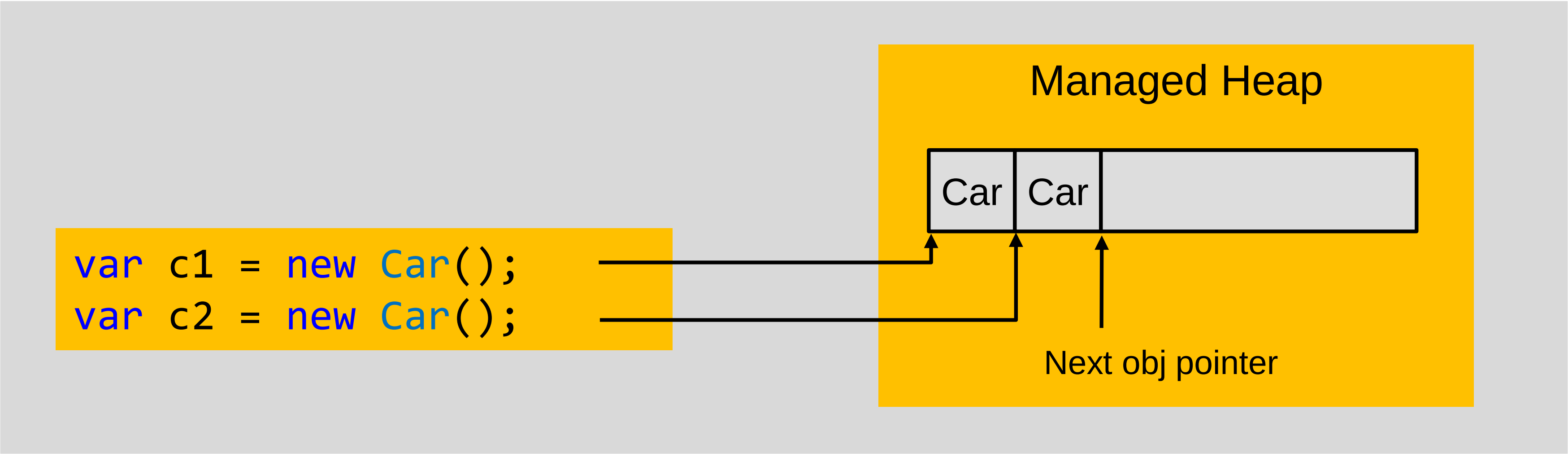


The heap

```
var c1 = new Car();
var c2 = new Car();
```



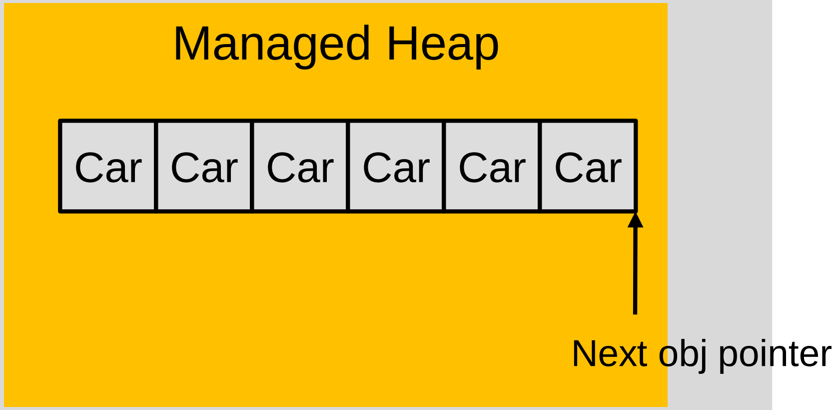
The heap



→ Very efficient allocation

The heap

```
var c1 = new Car();
var c2 = new Car();

{
  var c3 = new Car();
  var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```



How to free memory?
→ Garbage Collection

Simple garbage collection

Detection

1. The garbage collector searches for managed objects that are referenced in managed code

mark

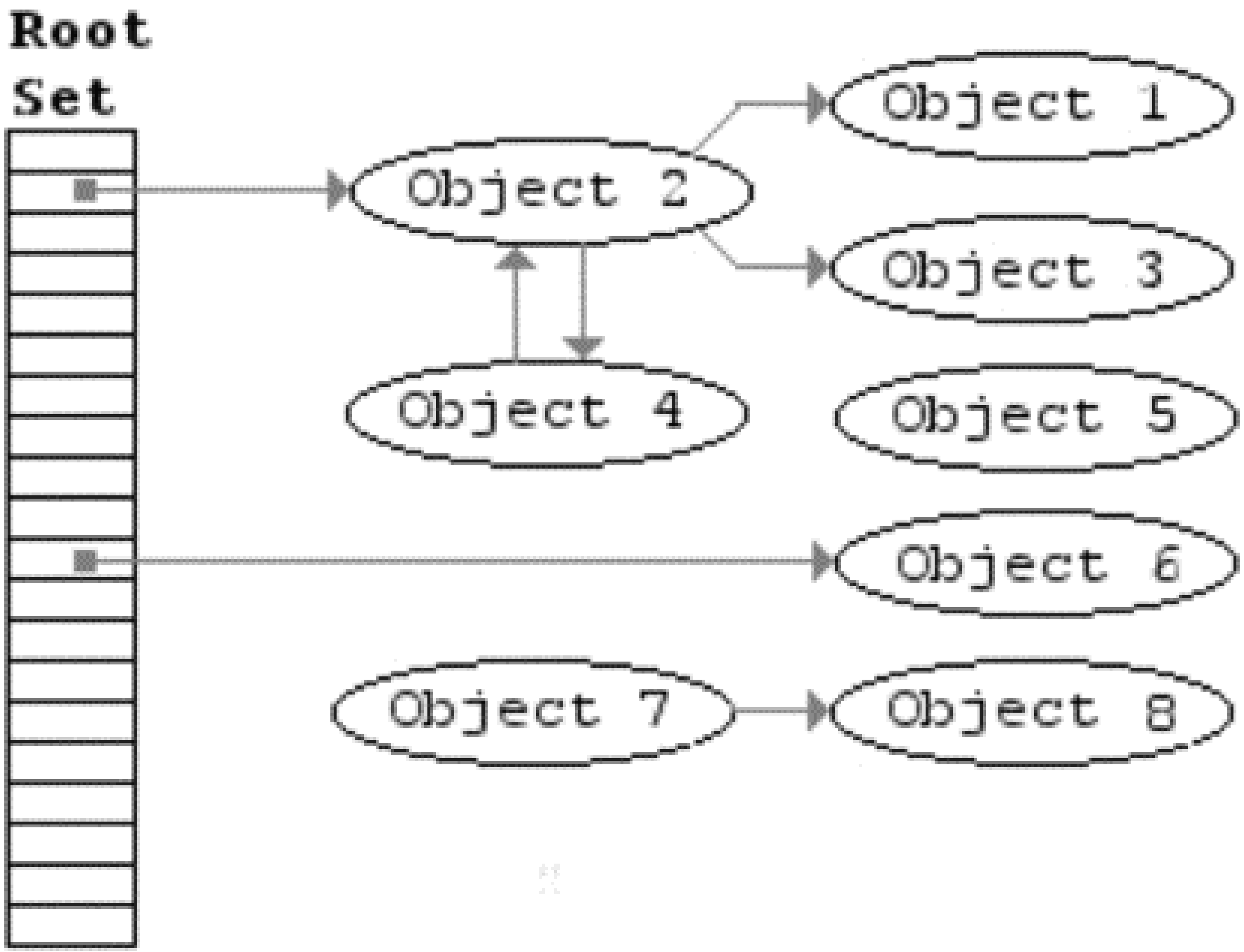
Reclamation

2. The garbage collector attempts to finalize objects that are unreachable

3. The garbage collector frees objects that are unmarked and reclaims their memory

sweep

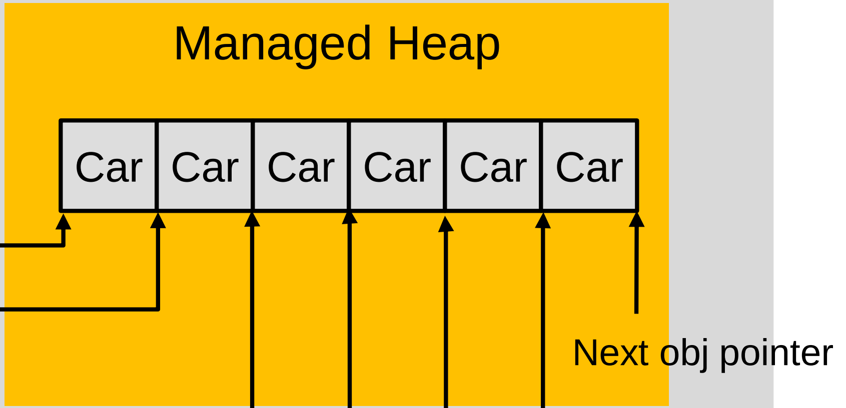
Simple garbage collection



Source: Wikipedia

The heap


```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```

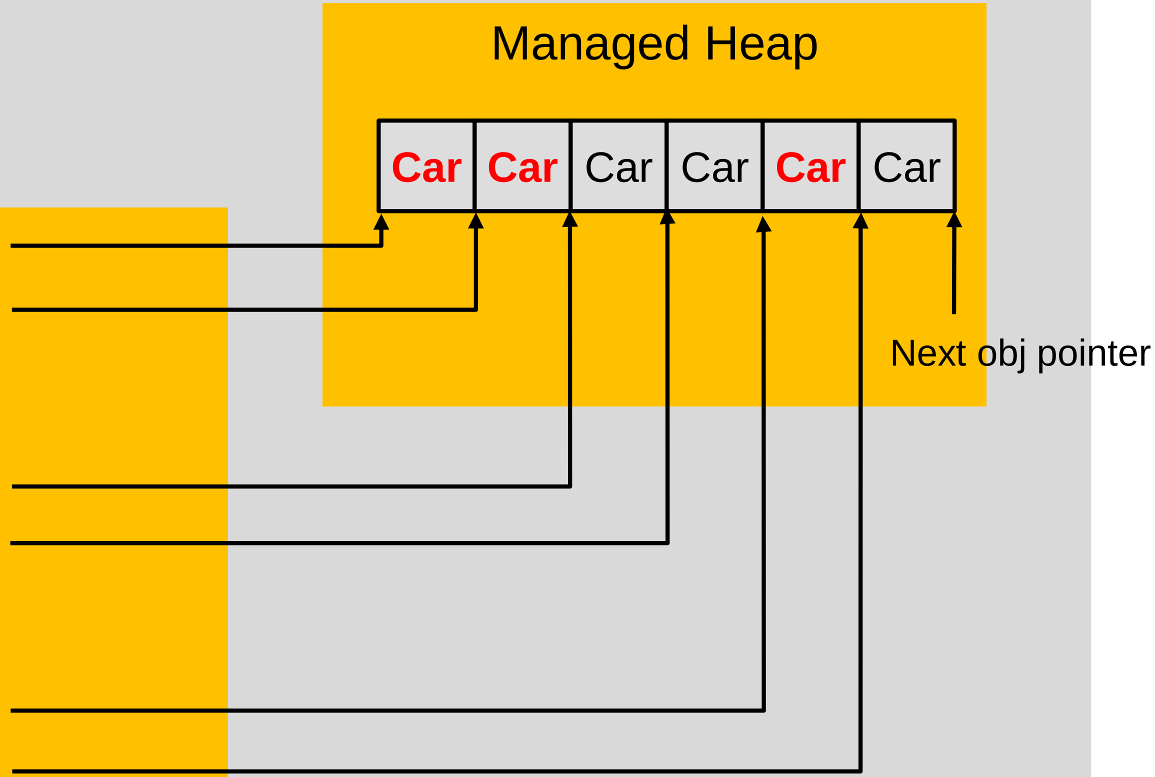


How to free memory?

→ Garbage Collection

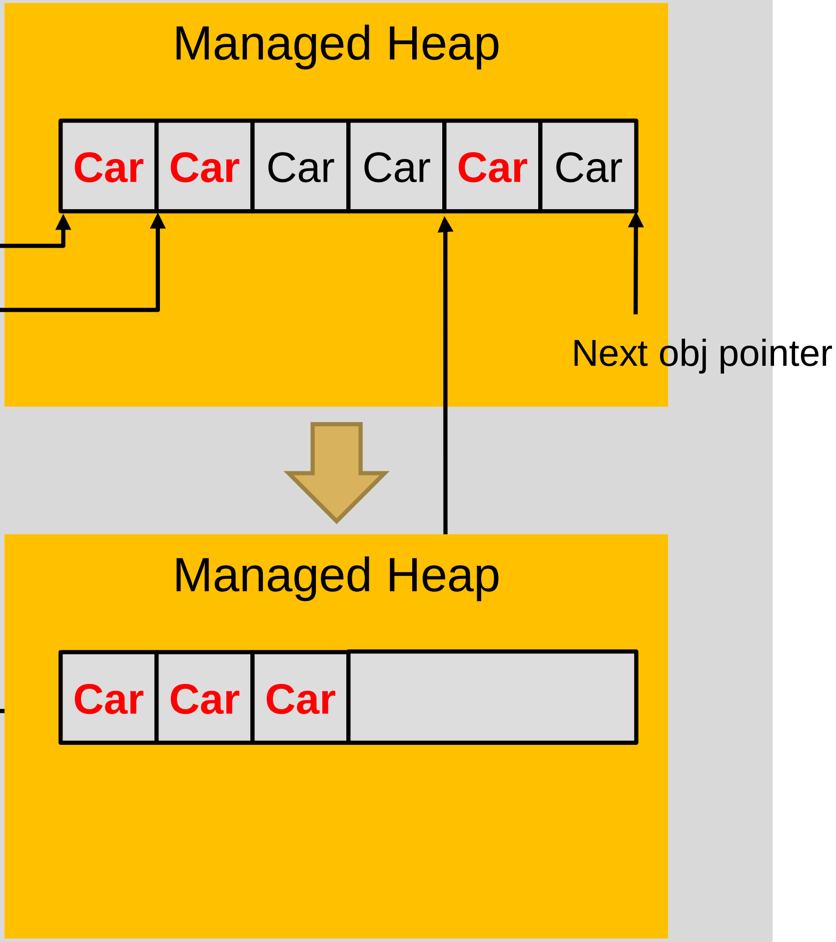
The heap


```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```



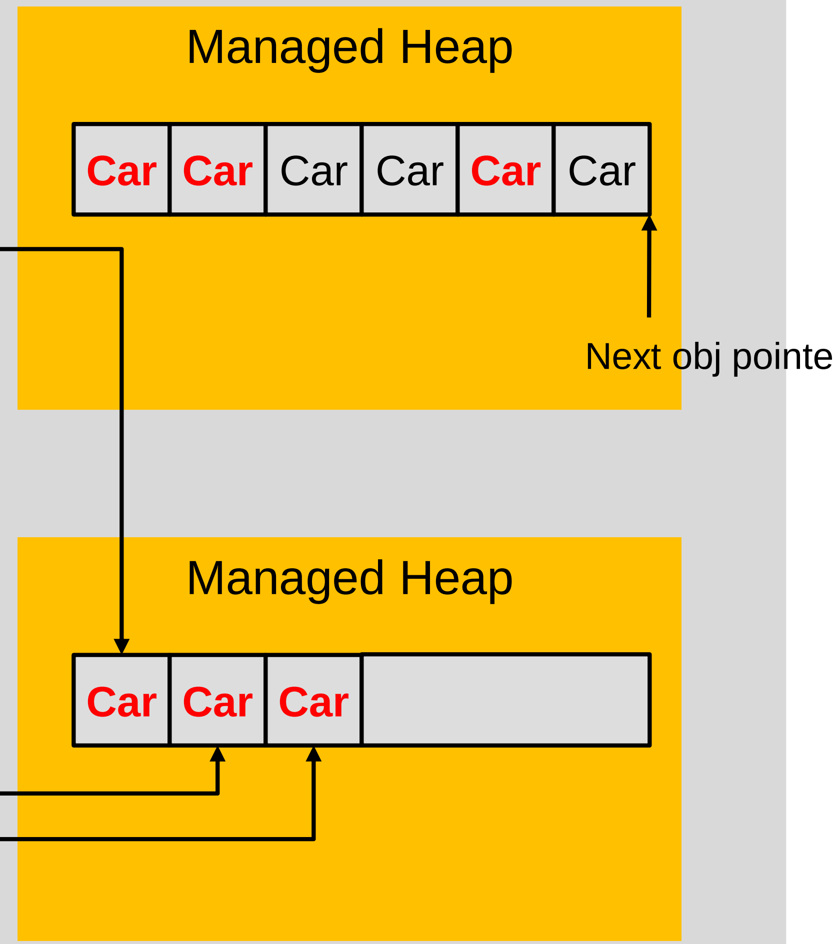
The heap


```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```



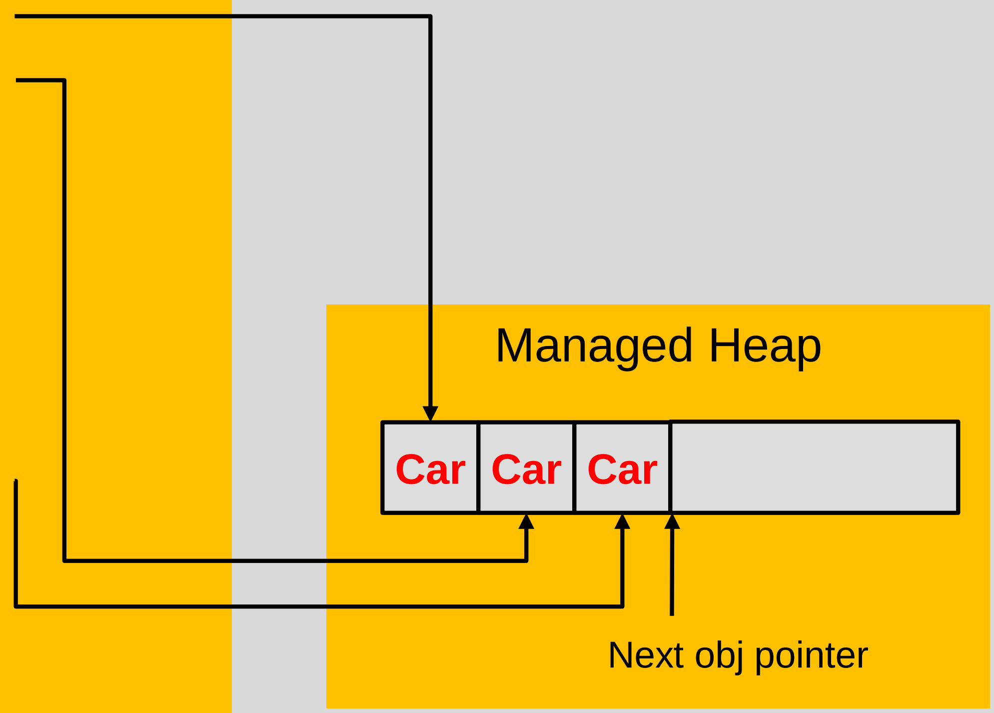
The heap


```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```



The heap


```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```

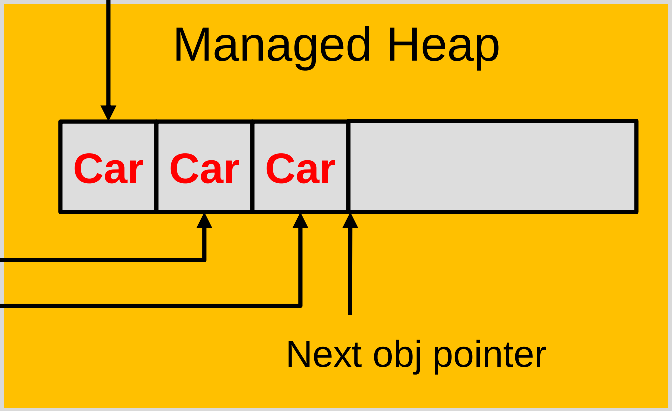


Fachhochschule Nordwestschweiz
Hochschule für Technik

The heap

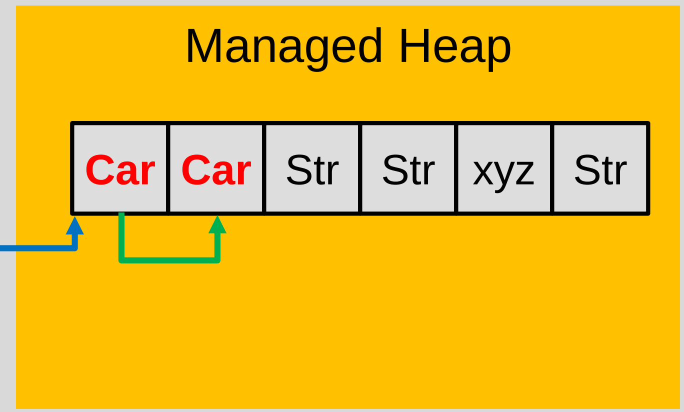
→ GC keeps *reachable* objects

```
var c1 = new Car();
var c2 = new Car();

{
   var c3 = new Car();
   var c4 = new Car();
}

var c5 = new Car();
var c6 = new Car();

c6 = null;
//...
```

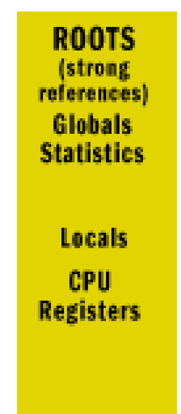


Reachability

```
var c = new Car();
c.NextCar = new Car();
```



- An object is reachable, if it is reachable from any of the **roots** via references, otherwise it is garbage
- Root set
  - Global variables
  - Stack (Arguments, local Variables), CPU registers
- Via other objects
- ~~WeakReference~~



WeakReference


```
var wr = new WeakReference<Car>(someCar);
//...

Car c;
if (wr.TryGetTarget(out c))
    Console.WriteLine($"Car {c} is still alive!",);
else
    Console.WriteLine("Car was Garbage Collected");
```

A `WeakReference` points to an object, without making it reachable.
→ Useful for GC-aware caching.

Garbage collection

This example is a *tracing*, *compacting*, *stop the world*, mark & sweep garbage collector

- tracing = Follow references to decide reachability
- compacting = Free memory by compacting heap
- stop the world = Stop all threads during GC
- mark & sweep = GC in two phases

Other approaches

- *Generational (Example: .NET)*
- Background (Example: .NET)
- Reference counting (Example: COM)
- Concurrent (Example: Oracle CMS)
- Deterministic/real-time (Example: Azul Zing)
- Region-based (Example: Oracle G1)
- Incremental (Example: Oracle CLP)
- ...

GC is an *implementation detail* in .NET

Garbage collection

Fachhochschule Nordwestschweiz
Hochschule für Technik

Garbage Collection in .NET is non-deterministic

Runtime performs GC whenever it “feels like it”:

- …nothing else to do
- …ran out of free memory
- …every [x] seconds
- Etc.

Cleanup of objects

GC cleans memory. What about files, network connections, native memory, locks, …?

→ Finalization

→ IDisposable

Finalization/destructors

- Implement a destructor (`~ClassName`) to perform cleanup
  Destructors (`~ClassName`) are syntactic sugar to override the `Finalize()` method

- Prior to an object being released, the GC calls its *finalizer/destructor, but*
  → GC is non-deterministic
  → Calling of finalizers is non-deterministic

- Destructor may be called…
  - …during “natural” garbage collection
  - …when calling `GC.Collect()`
  - …Application domain is unloaded from memory
  - …when CLR is shutting down

Finalization/destructors

- TRY NOT TO USE!

- ONLY ever for unmanaged resources
  - P/Invoke, COM, Native memory, …

- Make finalized objects as small as possible

- Never access referenced objects from the destructor

Worksheet – Part 1

Debug vs. release

Debugging can be difficult, because the scope of variables may differ in debug/release mode:

```
void Method()
{
  var c = new Car();

  //…some code that
  //doesn't use "c"
  //anymore…
}
```

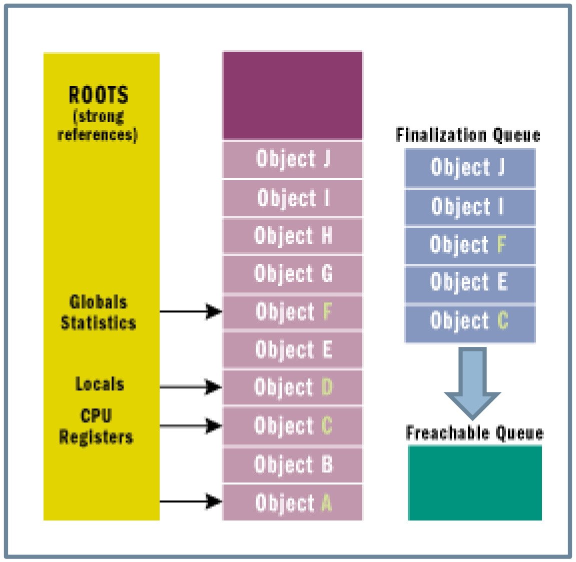
Is "c" still alive here?

Debug: yes
Release: no

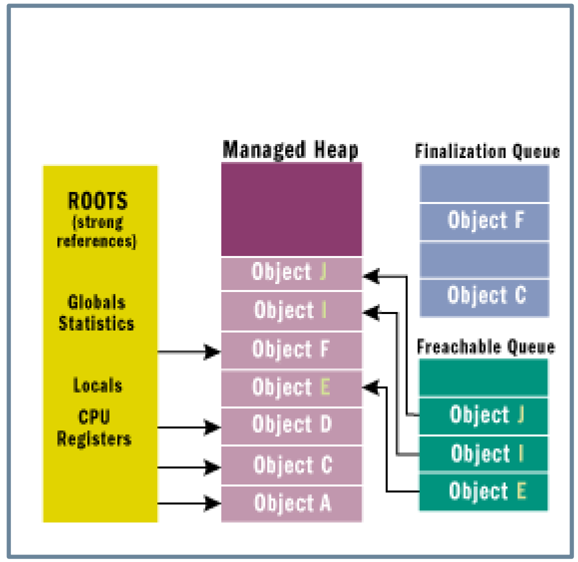
→ Finalization also depends on compiler mode

Finalization behind the scenes

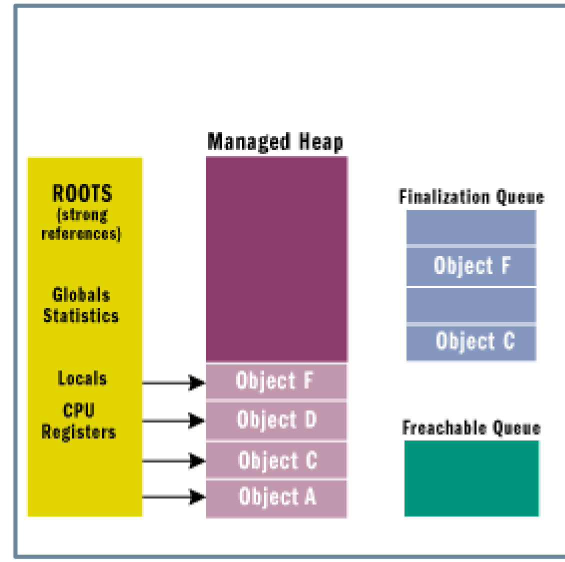
Finalization takes two Garbage Collection cycles:



Allocated objects on the heap



Unreachable objects moved to Freachable Queue during first GC



Objects removed from heap after second GC

Finalization problems

- Runs on the finalizer thread (concurrent to the rest of the application)
- Finalizer may not be called at all
- Non-deterministic
- Difficult to use reliably
- Time limits on shutdown
- Finalization order unspecified
- Impacts GC performance
  → Slows down your code
- Structs cannot have destructors

Explicit resource management

- Some objects require explicit tear-down:
  Open files, OS handles, unmanaged objects
- .NET provides the `IDisposable` interface
- Users call `Dispose()` explicitly
  → Deterministic, unlike destructors
- Structs can implement `IDisposable`
- Can be used in addition to destructors
  - If you don’t want to wait until destructor is eventually called
  - Users may want to clean up the object explicitly

IDisposable

```
// Implementing IDisposable
public class MyResourceWrapper : IDisposable
{
    // The user should call this method
    // when they no longer need this object
    public void Dispose()
    {
        // Clean up unmanaged resources & dispose
        // other contained disposable objects
    }
}
```

→ Always call `Dispose()` on objects you create

Inside using

using calls Dispose() automatically:

```
using(var l = File.AppendText("dates.txt")))
{
    l.WriteLine(DateTime.Now.ToString("yyyy-MM-ddTHH:mm:ss"));
}
```

```
File f = null;
try
{
    f = File.AppendText("dates.txt");
    f.WriteLine(DateTime.Now.ToString("yyyy-MM-ddTHH:mm:ss"));
}
finally
{
    if (f != null) try { f.Dispose(); } catch { }
}
```

`IDisposable` semantics

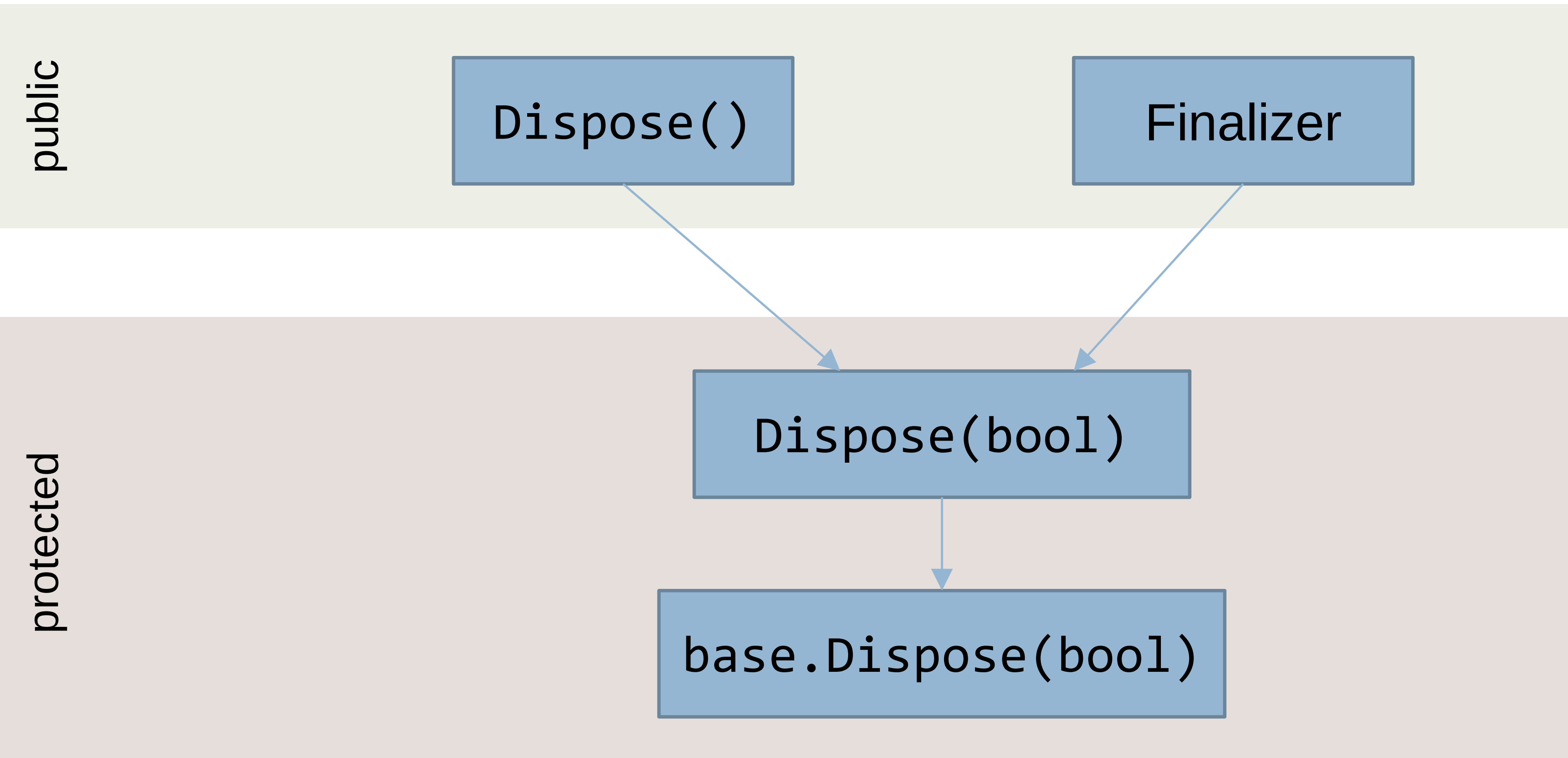
1. Once disposed, an object is beyond redemption
   - No reactivation
   - Calling its methods may cause `ObjectDisposedException`
2. Repeated `Dispose()` calls allowed
3. Objects call `Dispose()` on their child objects

Dispose pattern

- Excellent way to combine IDisposable and Finalizers to provide a backup for sloppy users, which may forget calling `Dispose()`

- Advantages
  - Ensures reliable, predictable cleanup
  - Prevents temporary resource leaks
  - Provides a standard, unambiguous pattern
  - Subclasses correctly release base class resources

Dispose pattern



Dispose pattern (1)

```
// thread-safe wrapper of an unmanaged handle
public sealed class OSHandle : IDisposable
{
    private bool disposed;

    public OSHandle(IntPtr h) { handle = h; disposed = false; }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    ~OSHandle()
    {
       Dispose(false);
    }

    ...
```

Dispose pattern (2)

```
    ...

    protected void Dispose(bool disposing)
    {
       if (!disposed)
       {
            // clean-up
            if (disposing)
            {
                /* safe to access references here */
            }
            disposed = true;

            // dispose unmanaged resources here
       }

       base.Dispose(disposing);
    }
}
```

`Dispose()` vs. Finalizers

- `Dispose()`
  - Deterministic
  - Explicitly called by user
  - Free resources (File handlers, locks, OS resources, …)

- Finalizers/destructors
  - Non-deterministic
  - Automatically called by GC
  - Free memory or as safety net

→ Dispose pattern combines best of both worlds!

System.GC

- Static methods to interact with GC

- Use this
  - Rarely, if ever
  - For micro-benchmarks
  - Responsiveness
  - When working with unmanaged resources

Worksheet – Part 2

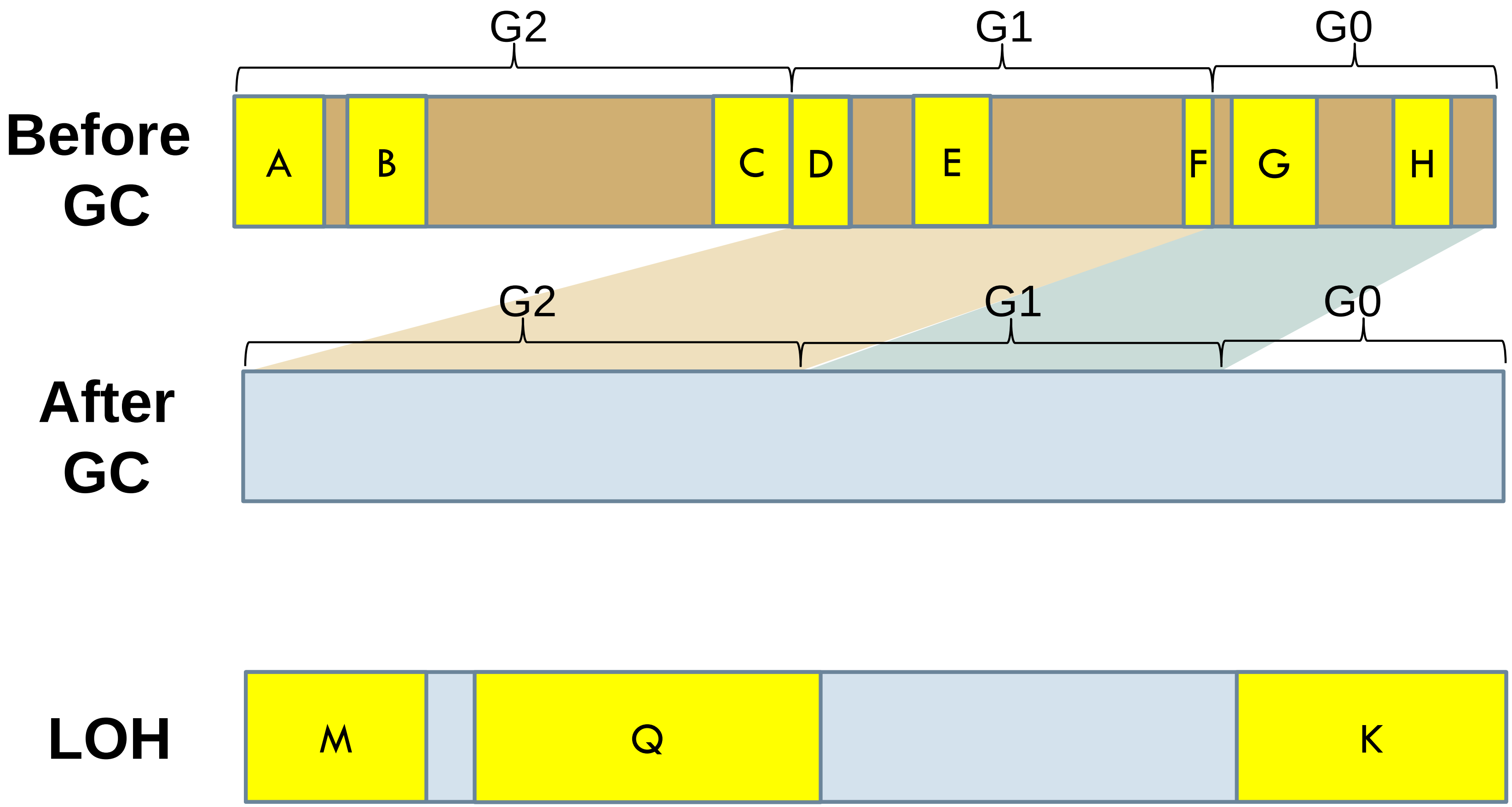
Generational GC

- Most objects are short-lived
  = most young objects die during GC

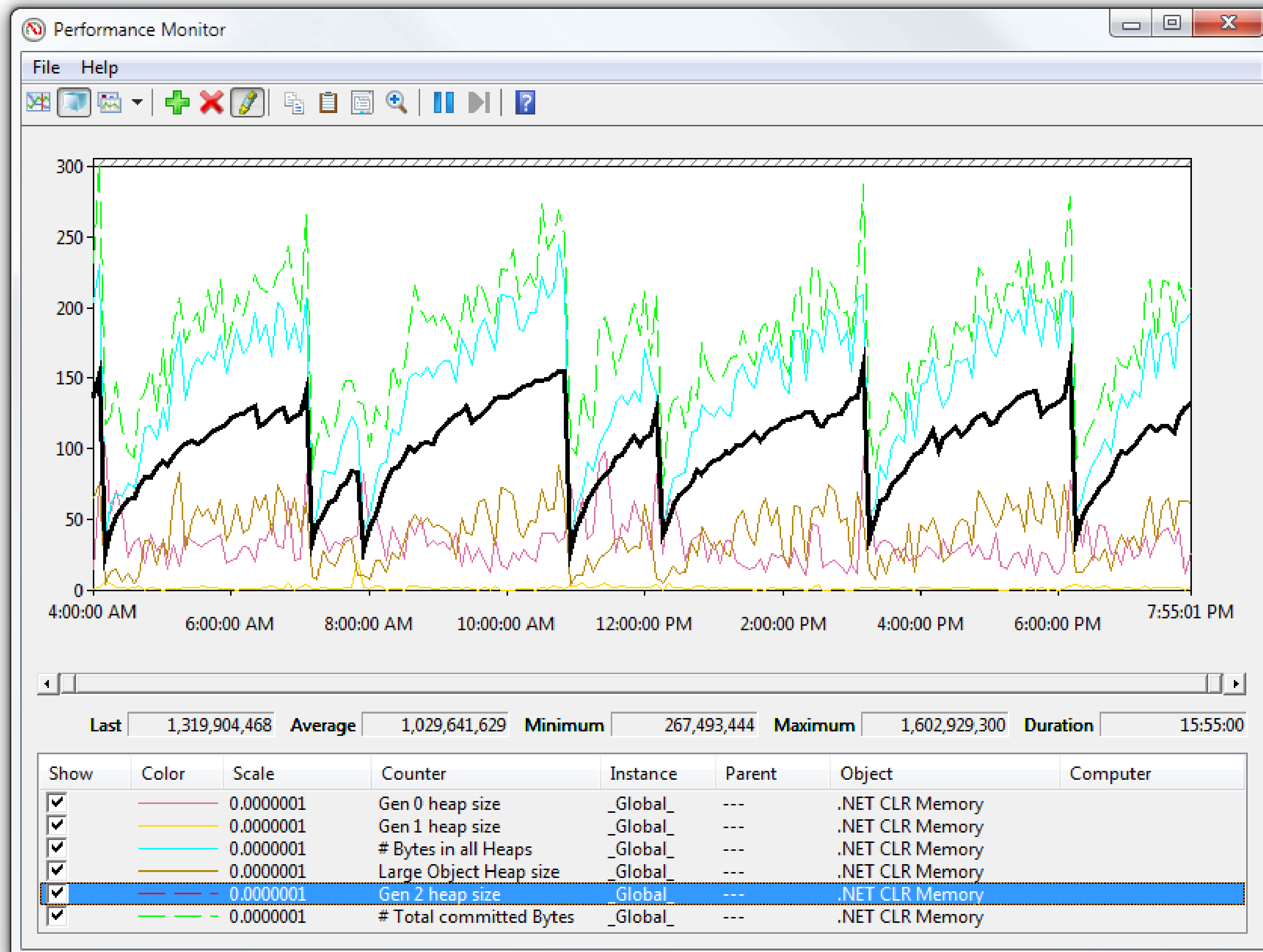
- Few objects are long-lived
  = most old objects stay alive during GC

→ Special handling of “young” objects

Generational GC



Generational GC



Source: David Hogue