Garbage Collection

.NET CORE

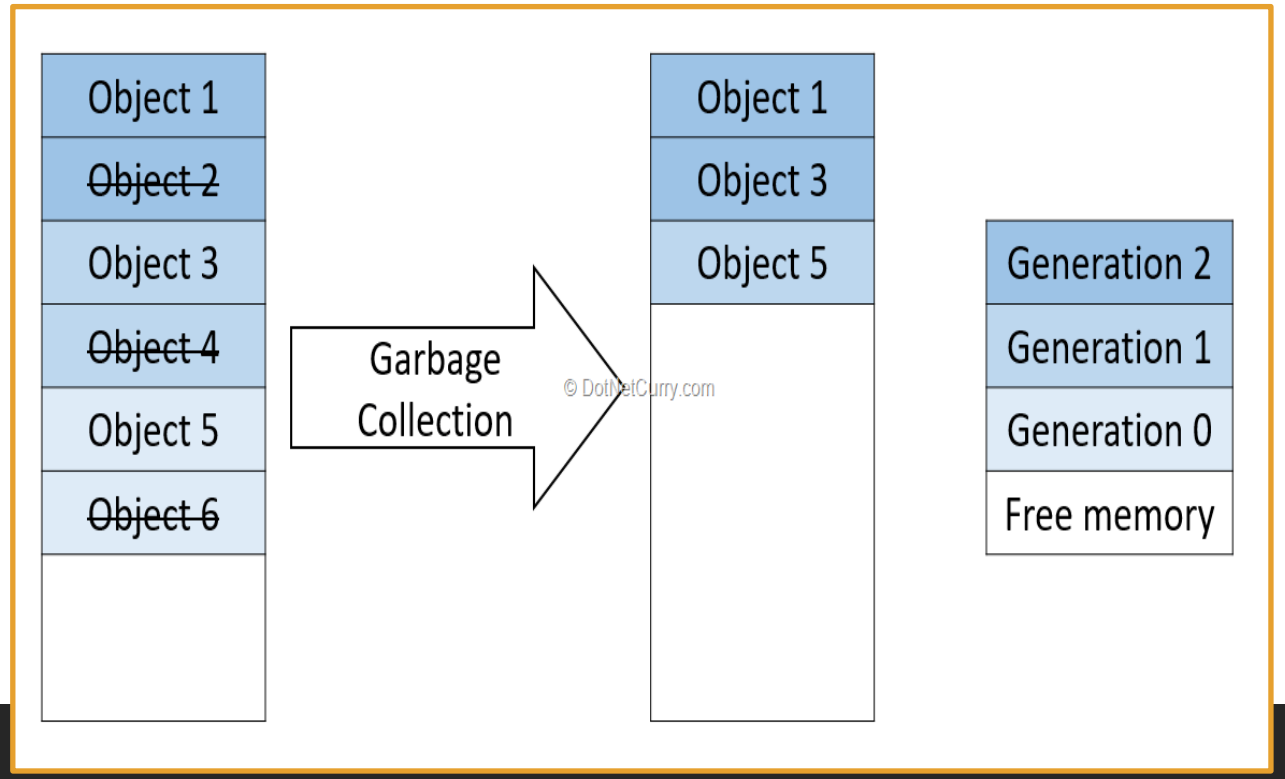
*.NET's **Garbage Collector** manages the allocation and release of memory for your application. It checks for objects in the managed **heap** that are no longer being used by the application and reclaims their memory.*

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/

Fundamentals of memory

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals#fundamentals-of-memory

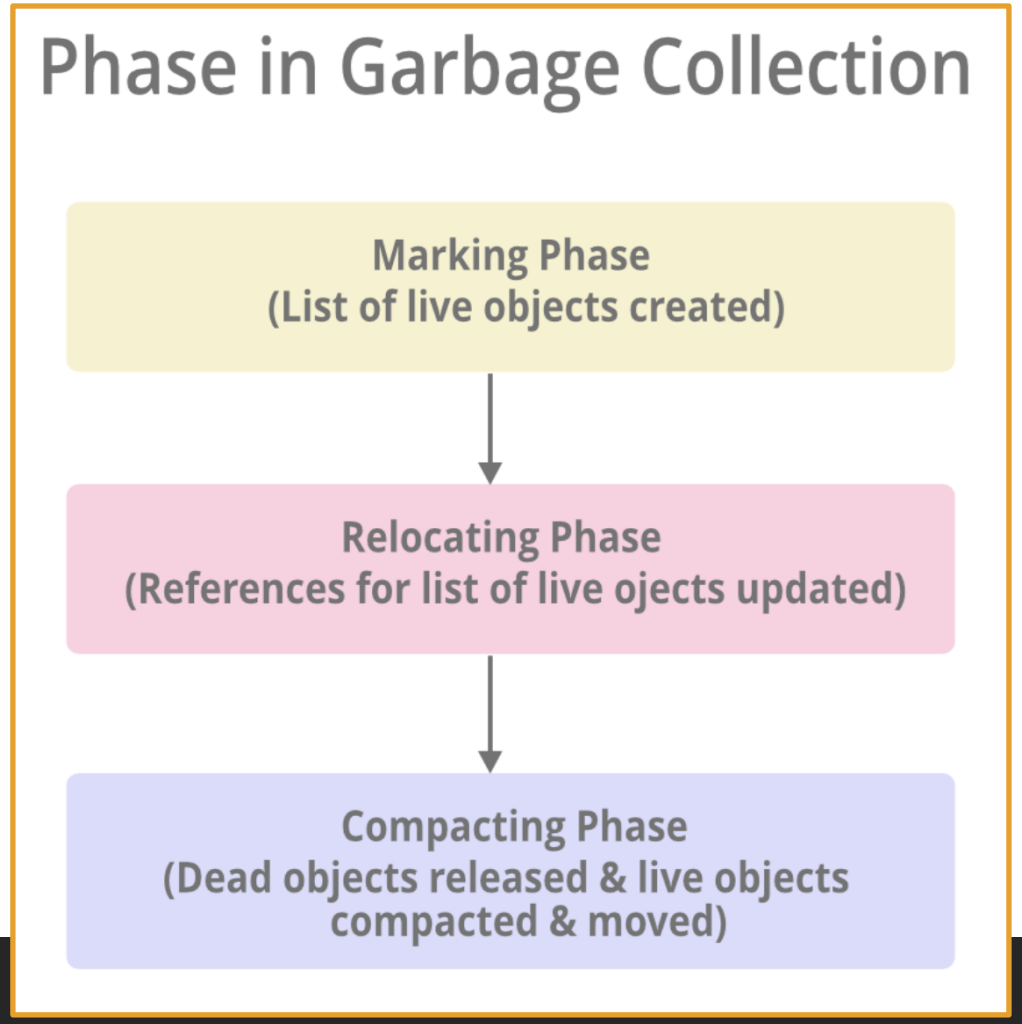
- Each ***process*** (program) has 2GB of virtual memory allocated.
- In C#, you cannot decide where or how memory is allocated during the process.
- The ***Garbage Collector (GC)*** allocates and frees memory.
- Virtual memory has three states:
  - Free
    - unallocated
    - available
  - Reserved
    - available
    - unusable for other processes
    - must be committed in order to store data
  - Committed
    - assigned to physical storage
- The frequency of ***garbage collection*** depends on the volume of allocations and the amount of survived memory on the managed heap.



Benefits of Garbage Collection

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

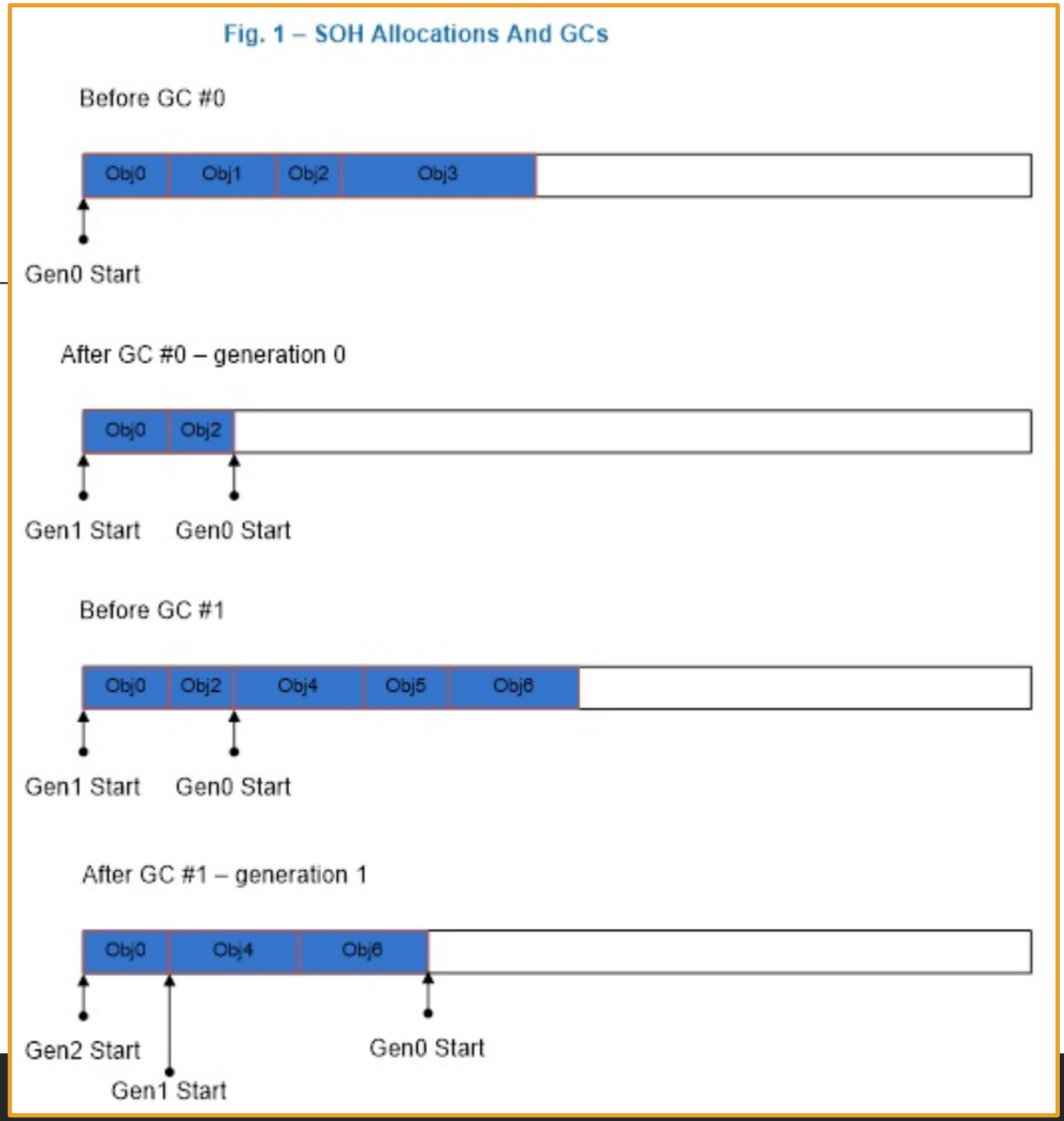
- No memory leaks.
- Efficient memory allocation.
- GC automatically reclaims unused objects, clears memory, and makes memory available.
- Constructors do not have to initialize every data field.
- GC makes sure that one object cannot use the contents of another object.



Managed Heap

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals
https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap

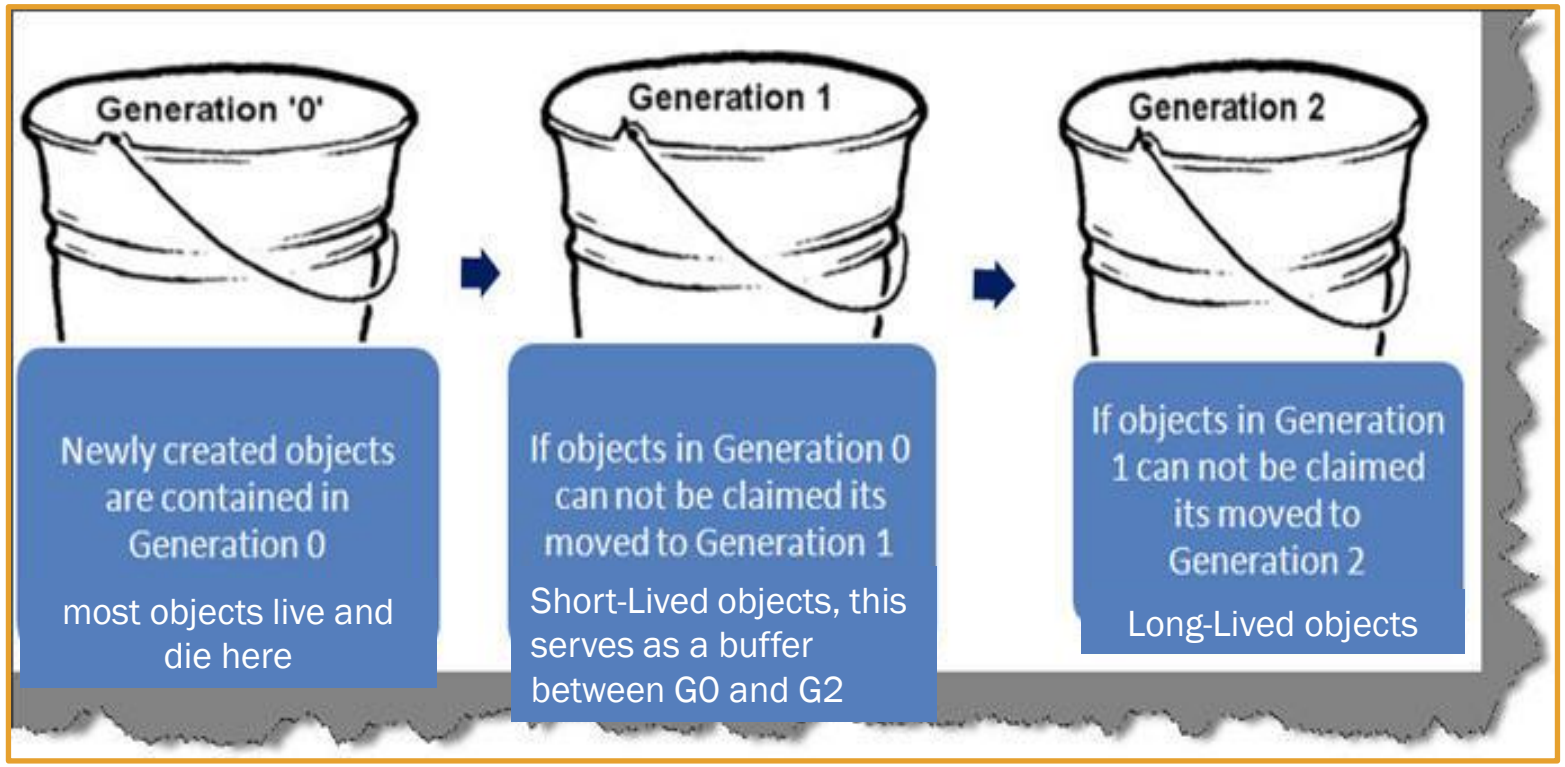
- The ***GC*** allocates a segment of memory, called the ***Managed Heap,*** to store and manage objects.
- There is one ***Managed Heap*** for each managed process.
- The ***GC*** calls the Windows ***VirtualAlloc()*** to reserve memory and ***VirtualFree()*** to release memory.
- The ***GC*** divides objects into small and large objects. Large Objects (arrays) go on the ***Large Object Heap (LOH)***, Small objects(instances) go on the ***Small Object Heap (SOH)***.



Heap Object Generations

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals#generations

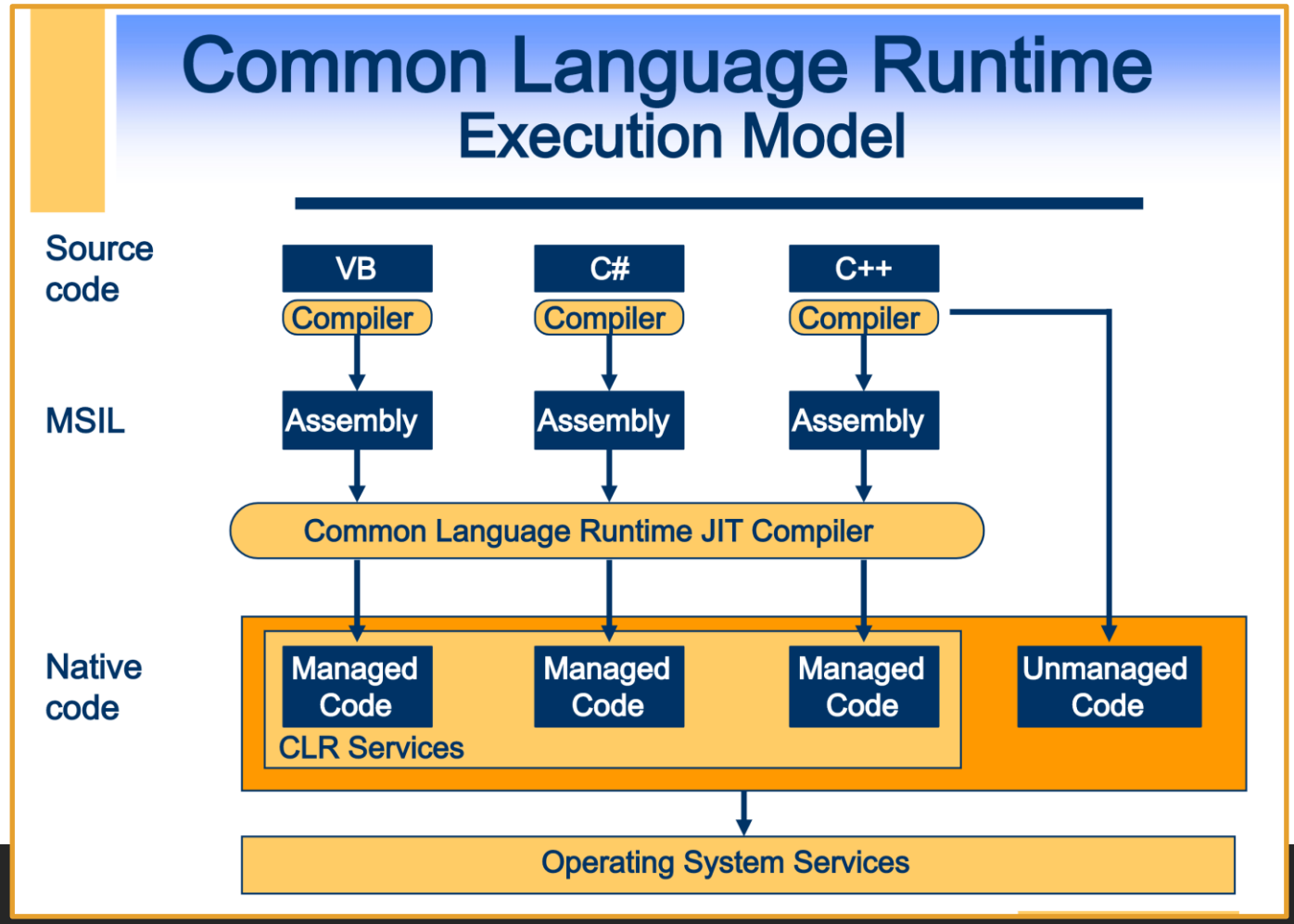
- ***Garbage collection*** happens on a whole generation at once.
- Objects that survive a ***garbage collection*** ('survivors') are promoted to the next generation.
- When ***GC*** sees that survival rate is high, it allocates more memory to that generation.
- After ***Garbage Collection***, survivors are 'compacted' (defragmentation) to the older end of the memory segment.



Managed Code

https://docs.microsoft.com/en-us/dotnet/standard/managed-code

- ***Managed code*** is code managed by the **Common Language Runtime** (***CLR***) at runtime.
- The ***CLR*** provides memory management (***GC***), security boundaries, and ***type*** safety.
- ***Managed code*** is written in a high-level language that can be run on top of .NET.
- Code is compiled into **Intermediate Language** (***IL, MSIL***, ***CIL***) code, which the ***CLR*** compiles and executes.
- The ***CLR*** manages the **Just-In-Time** compiling code from ***IL*** to machine code that can be run on any CPU.
- The ***CLR*** knows what your code is doing and can *manage* it.



Unmanaged Code

https://docs.microsoft.com/en-us/dotnet/framework/interop/

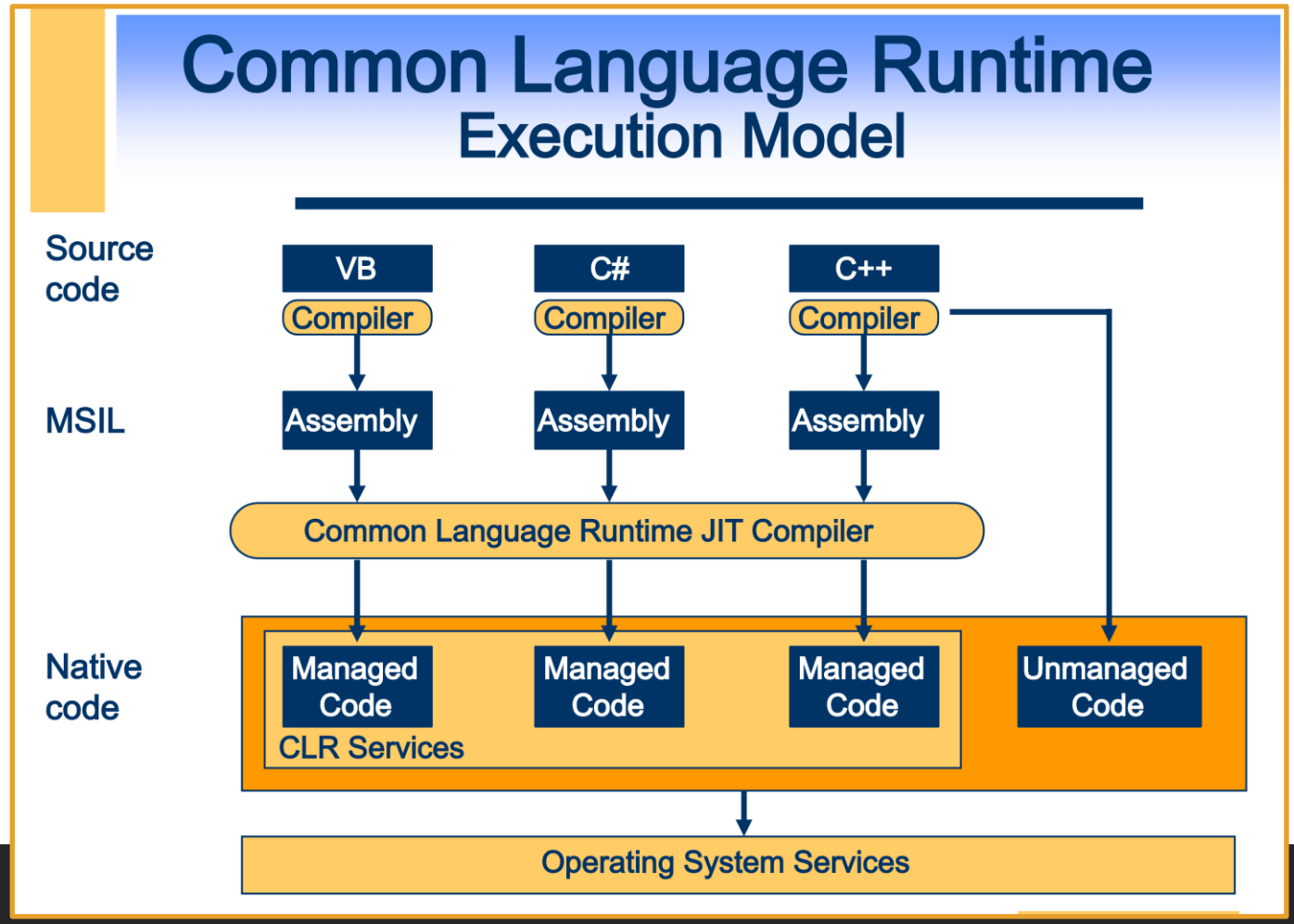
Code that runs outside the ***CLR*** is called ***Unmanaged Code***.

The .NET Framework promotes interaction with COM components, COM+ services, external type libraries, and many operating system services.

Data types, method signatures, and error-handling mechanisms vary between managed and unmanaged object models.

Examples of Unmanaged Code:

- COM components,
- ActiveX interfaces,
- Windows API functions.



Idisposable Interface

https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8

- The ***Garbage Collector (GC)*** has no knowledge of unmanaged resources (window handles, open files, streams).
- ***Idisposable*** provides a method for releasing unmanaged resources.
- If your app uses an object that implements the ***Idisposable*** interface, call the object's ***IDisposable.Dispose()*** implementation when finished using it.

```
// A base class that implements IDisposable.
// By implementing IDisposable, you are announcing that
// instances of this type allocate scarce resources.
public class MyResource: IDisposable
{
    // Pointer to an external unmanaged resource
```

```
// Dispose managed resources.
component.Dispose();
```

Using Block

https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=netframework-4.8

You can use ***using*** instead of calling ***IDisposable.Dispose()*** yourself.

The ***using*** statement is a syntactic convenience. At compile time, the language compiler implements the ***intermediate language (IL)*** for a **try/finally** block.

```
public WordCount(string filename)
{
   if (! File.Exists(filename))
      throw new FileNotFoundException("The file does not exist.");

   this.filename = filename;
   string txt = String.Empty;
   using (StreamReader sr = new StreamReader(filename)) {
      txt = sr.ReadToEnd();
   }
   nWords = Regex.Matches(txt, pattern).Count;
}
```

Using Block

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement

---

The ***using*** block provides a convenient syntax that ensures the correct use of ***IDisposable*** objects.

```
using (var font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

When the lifetime of an ***IDisposable*** object is limited to a single method, you should declare and instantiate it inside the ***using*** statement. The ***using*** statement correctly calls the ***.Dispose()*** method on the object and causes the object itself to go out of scope as soon as ***.Dispose()*** is called.

Within the ***using*** block scope, the object is read-only and cannot be modified or reassigned.

Using Block

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement

```
using (var font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

Is the same as

```
{
  var font1 = new Font("Arial", 10.0f);
  try
  {
    byte charset = font1.GdiCharSet;
  }
  finally
  {
    if (font1 != null)
      ((IDisposable)font1).Dispose();
  }
}
```

The ***using*** statement ensures that ***.Dispose()*** is called even if an ***exception*** occurs within the ***using*** block scope. You can achieve the same result by putting the object inside a ***try*** block and then calling ***.Dispose()*** in a finally block.

A ***using*** block is expanded to a ***try/catch*** block at compile time (note the extra curly braces to create the limited scope for the object).