**C# .NET**

\*\*C/C++: **Unmanaged code**. Compiles into machine code that will be used for that OS only (goes straight to memory and starts. All memory management must be done by programmer)

\*\*C#: **Managed code**

* Compiles to Intermediate Language code (independent of the OS)
* The **CLR (Common Language Runtime)** performs JIT (Just In-time Compilation) to produce machine code that corresponds to that specific OS. The CLR does memory management, security boundaries, and type safety

CLR allows passing boundaries between managed and unmanaged world 🡪 *interoperability (interop)*

These provisions would allow you to **wrap up an unmanaged library** and call into it.

**Memory Allocation**

**Stack**: **LIFO**, **allocation is done during compile time**. When a method is invoked, the CLR bookmarks the top of the stack, method pushes data onto the stack as it executes

**Heap**: allow objects to be allocated/deallocated in **random order**, **allocation is done at run time**. The heap requires the overhead of a garbage collector to keep things in order.

*Value type variables are stored in the stack and reference type variables are stored in the heap.*

A picture containing text, screenshot, diagram, line

Description automatically generatedA **value type** holds the data within its own memory location.

Value types => *bool, byte, char, decimal, double, float, int, long, uint, ulong, ushort, enum, struct*

A **reference type** contains a pointer to another memory location that holds the real data. (C++/CLI)

Reference types => *class, interface, delegate, string, object, dynamic, arrays*

\*\* Instance names are reference type (pointers) 🡪 Shallow copy:

|  |
| --- |
| Person alice = new Person { Name = "Alice", Age = 36 }; |
| Person bob = alice; |
| bob.Name = "Bob"; |
| Console.WriteLine($"Alice's name: {alice.Name} --- Bob's name: {bob.Name}"); |
| // Alice’s name: Bob --- Bob’s name: Bob |

* Each local variable (i.e. one declared in a method) is stored on the stack. That includes reference type variables — the variable itself is on the stack, but remember that the value of a reference type variable is only a reference (or null), not the object itself.
* A struct variable declared within a method will always be on the stack, whereas a struct variable which is an instance field of a class will be on the heap, which points to the struct instance on another place in the heap. (MUST USE new keyword when initializing)
* **Every static variable** is stored on the heap, regardless of whether it’s declared within a reference type or a value type. There is only one slot in total no matter how many instances are created.

**Memory Management**

.NET’s Garbage Collector (GC) manages the allocation and release of memory for your application.

However, *when we create objects that include unmanaged resources such as windows, files, network and database connections*, we must explicitly release those resources after using them in our applications.

1. **Finalizers** (Destructors) ~Class() { /\* clean up statements \*/}

Finalizers are used to perform any necessary clean-up when a class instance is being collected by GC. Finalizers cannot be called explicitly, they are called by the GC

1. **Dispose** (from IDisposable Interface)

If the application *uses an expensive external resource*, we should release the resource explicitly

|  |
| --- |
| class DatabaseConnection : IDisposable  { |
| #region IDisposable Support |
| private bool disposedValue = false; // To detect redundant calls |
|  |
| protected virtual void Dispose(bool disposing) |
| { |
| if (!disposedValue) |
| { |
| if (disposing) |
| { |
| // TODO: dispose managed state (managed objects). |
| Console.WriteLine("Explicit call: Dispose is called by the user."); |
| } else |
| { |
| Console.WriteLine("Implicit call: Dispose is called through finalization."); |
| } |
|  |
| // TODO: free unmanaged resources (unmanaged objects) and override a finalizer. |
| // TODO: set large fields to null. |
| disposedValue = true; |
| } |
| } |
|  |
| // TODO: override a finalizer only if Dispose above has code to free unmanaged resources. |
| ~DatabaseConnection() |
| { |
| // Do not change this code. Put cleanup code in Dispose(bool disposing) above. |
| Dispose(false); |
| } |
|  |
| // This code added to correctly implement the disposable pattern. |
| public void Dispose() |
| { |
| // Do not change this code. Put cleanup code in Dispose(bool disposing) above. |
| Dispose(true); |
| // TODO: uncomment the following line if the finalizer is overridden above. |
| GC.SuppressFinalize(this); |
| } |
| #endregion |
| } |

Another commonly used method to call Dispose is using ***using*** statement

|  |
| --- |
| class Program |
| { |
| static void Main(string[] args) |
| { |
| using (var connection = new DatabaseConnection()) |
| { |
| //Write your operational code here |
| } |
| } |
| } |
|  |

* Automatically calls connection.Dispose(true);