**Self Study\_Task\_1**

In .NET, code is compiled into IL (Intermediate Language), then converted to native machine code at runtime using the JIT compiler. This can cause runtime overhead, especially during application startup. .NET uses several techniques to reduce this overhead.

**Key Techniques**

1. **Ahead-of-Time Compilation**

.NET provides tools like:

* **NGen** (for .NET Framework)
* **ReadyToRun (R2R)** (for .NET Core / .NET 5+)

These precompile IL into native code before runtime.

**Benefit:** Faster startup, less JIT time.

**2. Tiered Compilation**

Used in .NET Core and later:

* Starts with fast, low-optimized code.
* Recompiles "hot" methods with better optimizations.

**Benefit:** Quick startup + improved long-term performance.

**3. Method Inlining**

Small methods are embedded directly into their callers.

**Benefit:** Reduces function call overhead.

**4. Code Caching**

Once compiled, a method stays in memory as native code.

**Benefit:** Avoids recompiling the same method.

**5. Lazy Compilation**

Only compiles methods when they’re called for the first time.

**Benefit:** Saves time by skipping unused code.

**Conclusion**

.NET reduces JIT overhead using:

* Precompilation (NGen, R2R)
* Tiered and lazy compilation
* Inlining and caching  
  These strategies improve startup speed and runtime performance.

**2. Difference Between Compiled and Interpreted Languages – and Where C# Fits**

Programming languages can be divided into **compiled** and **interpreted** languages based on how code is translated and executed:

* **Compiled Languages**:  
  These languages are translated into **machine code** by a compiler **before execution**. This machine code runs directly on the hardware, leading to better performance. Examples include **C** and **C++**.
* **Interpreted Languages**:  
  These are **executed line-by-line at runtime** by an interpreter. They are more flexible but slower in performance. Examples include **Python** and **JavaScript**.
* **C# Position**:  
  C# is **both compiled and interpreted**. It is first **compiled into Intermediate Language (IL)** using the C# compiler. Then, during execution, the **Just-In-Time (JIT) compiler** of the .NET runtime interprets this IL and converts it into native machine code.  
  Hence, C# is considered a **managed and hybrid language**.

**3. Comparison Between Implicit, Explicit, Convert, and Parse Casting**

C# supports several ways of converting data between types. Here is a structured comparison:

| **Type** | **Definition** | **Example** | **Notes** |
| --- | --- | --- | --- |
| **Implicit** | Automatic conversion between compatible types where there is no risk of data loss. | int a = 10; double b = a; | Safe and no need for cast syntax. |
| **Explicit** | Manual conversion between types, often where data loss is possible. | double d = 5.6; int x = (int)d; | Requires cast syntax (type); may lose data. |
| **Convert** | Uses System.Convert methods to convert between types safely. | Convert.ToInt32("123") | Handles null, more flexible, runtime-checked. |
| **Parse** | Parses a string to a specific data type. | int.Parse("123") | Throws exception if string is null or invalid. |

Bouns

**5. What is meant by C# is managed code?**  
C# runs under the control of the .NET runtime (CLR), which handles memory management, security, and garbage collection automatically.

**6. What is meant by "struct is considered like class"?**  
Both struct and class can have fields and methods, but struct is a value type stored in the stack, while class is a reference type stored in the heap.

