**1. Compiled vs. Interpreted Languages**

The key difference between **compiled** and **interpreted** languages lies in how the code is executed:

* **Compiled Languages**:
  + Code is translated directly into machine code (binary code) by a compiler before execution.
  + The entire program is converted in one go, and an executable file is created. This file can be run independently of the original source code.
  + Compilation is done once, and the program runs faster because it's already in a machine-readable format.
  + Examples: C, C++, Go, Rust.
* **Interpreted Languages**:
  + Code is translated into machine code line-by-line or statement-by-statement by an interpreter at runtime.
  + There’s no intermediate executable file; the source code must be available every time you run the program.
  + This often results in slower execution because the code is being translated during execution.
  + Examples: Python, JavaScript, Ruby.

**C# and Its Compilation**

C# is **primarily a compiled language**. However, it uses a combination of both:

* **Compilation**: C# code is compiled into an intermediate language (IL) by the C# compiler (csc), which is platform-agnostic.
* **Execution**: At runtime, the **Common Language Runtime (CLR)** executes the IL code. The CLR just-in-time (JIT) compiles the IL to native machine code when the program is run.

So, while C# is compiled, it relies on runtime execution via the CLR, which is a hybrid approach compared to purely compiled languages.

**2. Implicit, Explicit, Convert, and Parse Casting in C#**

Casting in C# allows you to convert one data type to another. Here's how **implicit**, **explicit**, **Convert**, and **Parse** differ:

* **Implicit Casting**:
  + Performed automatically by the compiler when converting between compatible types.
  + Generally occurs when converting a smaller type to a larger one (e.g., int to long).
  + **Example**:
  + int i = 100;
  + long l = i; // Implicit casting from int to long
* **Explicit Casting**:
  + Requires the developer to explicitly specify the conversion, typically when converting from a larger type to a smaller one, which may lose data.
  + Uses parentheses to perform the cast.
  + **Example**:
  + long l = 100L;
  + int i = (int)l; // Explicit casting from long to int (possible loss of data)
* **Convert**:
  + A method in the System.Convert class that allows safe type conversions between different types (including incompatible types).
  + Unlike explicit casting, it handles type mismatches or overflow errors, often with default behavior like rounding or truncating.
  + **Example**:
  + string str = "123";
  + int i = Convert.ToInt32(str); // Converts string to int
* **Parse**:
  + A method that converts a string representation of a type into its respective type (usually numbers or dates).
  + Throws exceptions if the format is incorrect (e.g., parsing a non-numeric string to an integer).
  + **Example**:
  + string str = "123";
  + int i = int.Parse(str); // Converts string to int

**Summary of Differences:**

* **Implicit**: Automatically done by the compiler for compatible types (e.g., smaller to larger types).
* **Explicit**: Done by the developer when there's a risk of data loss (e.g., larger to smaller types).
* **Convert**: Used for safe conversion, handling edge cases and type mismatches (like null values).
* **Parse**: Used for converting strings into data types (e.g., parsing a number from a string), throws an exception if invalid.

Let me know if you need further clarification!

Part 3

**1. What does "Managed Code" mean in C#?**

In the context of C#, **managed code** refers to code that is executed by the **Common Language Runtime (CLR)**, which is part of the .NET Framework. Managed code runs under the supervision of the CLR, which provides several services like memory management, type safety, garbage collection, exception handling, and security.

Key points about **managed code**:

* **Memory Management**: The CLR automatically handles memory allocation and garbage collection, meaning developers don't need to manually allocate or free memory.
* **Type Safety**: The CLR ensures that types are used correctly according to their definitions, preventing type errors at runtime.
* **Security**: The CLR provides security mechanisms to prevent unsafe operations, such as unauthorized memory access.
* **Platform Independence**: Managed code is platform-independent as it runs on the CLR, which can run on any platform where the .NET runtime is available (Windows, Linux, macOS).

In contrast, **unmanaged code** is directly executed by the operating system, and the developer must handle memory allocation and deallocation themselves (e.g., C or C++).

**Example**: C# is managed code because the C# code is compiled to Intermediate Language (IL) which is executed by the CLR. In contrast, C++ code is unmanaged because it is compiled directly to native machine code that runs directly on the hardware.

**2. What does "struct is considered like class before" mean?**

In C#, **structs** and **classes** are both used to define types, but they have different behavior, particularly in terms of how they are stored and passed around in memory.

However, historically (and in some contexts), a **struct** was considered "like a class" because both can define methods, properties, fields, and events. The main distinction comes from the **value type** versus **reference type** behavior.

**Key differences between structs and classes:**

1. **Value Type vs. Reference Type**:
   * **Structs** are **value types**, meaning when a struct is assigned to a new variable or passed to a method, a **copy** of the struct is made. The struct data is stored directly in the variable or stack.
   * **Classes** are **reference types**, meaning when a class is assigned to a new variable or passed to a method, only the **reference (memory address)** to the object is passed. The object itself is stored on the heap.
2. **Memory Allocation**:
   * **Structs** are usually stored on the **stack** (except when part of a heap-allocated object), making them generally more efficient in terms of memory allocation and deallocation.
   * **Classes** are stored on the **heap** and have garbage collection for cleanup, but allocating and deallocating memory on the heap can be less efficient.
3. **Inheritance**:
   * **Structs** do not support inheritance from other structs or classes (except from System.ValueType, which is implicit).
   * **Classes** support full inheritance, meaning a class can inherit from another class and implement interfaces.
4. **Default Constructor**:
   * **Structs** cannot define a parameterless constructor (C# 9.0 allows init properties and constructor enhancements). They always have an implicit parameterless constructor, initializing all fields to their default values.
   * **Classes** can define any constructor, including parameterless ones.

**Historical Context:**

Earlier in C#'s history, **structs** and **classes** were sometimes considered more similar than they are now in terms of capabilities. But their differences in memory management and behavior are now well defined. The major distinction, however, lies in whether the type is a **value type** (struct) or a **reference type** (class).

So, when people say "struct is considered like a class before," they might be referring to the fact that both structs and classes can hold similar data members (fields) and methods, but now the distinction between value types and reference types is clearer, especially with newer C# versions.

Let me know if you'd like more details on any specific point!

Self Report

**Self-Report: Studying C#**

I've been diving into C# recently, and it's been an interesting journey so far. The language is pretty intuitive, especially with its strong type system, which really helps avoid a lot of potential issues when working with different data types. I’ve been getting the hang of how to write clean, efficient code using object-oriented principles, especially with classes and methods.

The transition from understanding value types (like structs) to reference types (like classes) was a bit tricky at first, but now I feel like I have a solid understanding of memory management and the performance differences between the two. I'm learning how the CLR manages memory through garbage collection, and that’s been a big revelation compared to languages like C, where you have to manually manage memory.

One thing I’ve been exploring more recently is how to deal with exceptions in C#—using try-catch blocks effectively and knowing when and where to catch exceptions to prevent crashes. The concept of **managed code** also stood out to me, especially the way the CLR handles so many tasks automatically.

I’ve also learned about casting, and I find the differences between **implicit**, **explicit**, **Convert**, and **Parse** casting interesting. They all serve their purposes, and I’ve been practicing using each one to better understand when to use which.

Overall, C# feels like a very powerful and versatile language, and I’m looking forward to deepening my understanding of LINQ, async/await, and diving into more advanced topics like generics and reflection. It's definitely a language that’s opening up a lot of possibilities for building apps and working with data in a structured way.