Basic Details I

C# is a versatile and powerful programming language that is defined by many important underlying details. Many of these details are unique to C#, while others take inspiration from popular languages. C# uses variable scoping and the “scope” keyword for its scope handling, has an intricate type system, and has an effective and efficient memory management structure. All of these details come together to make C# an effective programming language.

To start, C# uses scoping to determine where variables and objects are accessible within a certain program. A variable or object has three types of scopes, which include class level, method level, and block level. The only difference between each level is where the variable or object can be accessed. Class level scope refers to a variable or object that is declared outside of any method or constructor. These are called fields, and they can be accessed from anywhere inside a specific class. This means their scope is limited to the class it’s declared in. Method level scope, also called local scope, is used when a variable or object is declared within a certain method. These are called local variables and can only be accessed inside a specific method. This means its scope is limited to the method it’s declared in. A variable or an object declared in an if statement or loop have block level scoping. They are accessible from anywhere inside the block it’s declared in. This means its scope is limited to the if statement or loop it’s declared in.

Along with variable scoping, C# also has the contextual keyword “scoped” to handle scoping. In Microsoft Learn, Declaration Statements, Bill Wagner describes how the “scoped” keyword works. Wagner says, “The contextual keyword ‘scoped’ restricts the lifetime of a value. Adding the ‘scoped’ modifier asserts that your code won’t extend the lifetime of the variable”. The “scoped” modifier can be applied to parameters and local variables only when the type is a ref struct. Without the keyword “scoped”, passing locally declared ref structs could lead to issues

related to lifetime. It is also useful to use this keyword to safely work with stack-allocated data without violating lifetime expectations.

C# has an effective type system as well. C# is a strongly and statically typed programming language. This means that every variable declared must have a data type assigned to it by compile time. It does have a few exceptions that stray away from the static typing norms. First, there is the “var” keyword. This is used to infer the type of a variable during initialization. Another oddity is the “dynamic” keyword. This keyword allows for a variable to be checked at runtime instead of compile time. Along with the “var” and “dynamic” types, C# has other built-in types that represent integers, characters, decimals, strings, and objects (to name a few). These built-in types of C# are not unique to itself, and they can be found across many other programming languages.

C# also allows for custom types. In Microsoft Learn, The C# Type System, Bill Wagner says, “You can use the struct, class, interface, enum, and record constructs to create your own custom types”. These custom types are more unique since it provides “record class” and “record struct”. These types are primarily used for encapsulating data. The “record” or “record class” syntax is used as a synonym to clarify a reference type. An example of “record class” could look like this: public record Student(string firstName, string lastName, int gpa);. On the other hand, “record struct” is used to define a value type with similar functionality. An example of “record struct” could look like this: public record struct Data(double y, int z);. The primary constructor parameters to a record are referred to as positional parameters. There is a difference between positional parameters in both keywords. Positional parameters are immutable in a “record class”, but in a “record struct” positional parameters are mutable. Bill Wagner goes into more depth with the two syntaxes in Microsoft Learn, Records (C# reference).

Another notable feature about types in C# is it supports pointer related operators and reference types. C# allows for the usage of pointer operators that can be found in languages such as C. Although C# has automated memory management, it still allows for the utilization of pointers. Reference types are defined by using the “class” or “record” keywords. C# allows for the usage of reference types with classes and records, as seen in the above paragraph. Reference types have different compile-time rules and run-time behavior than value types.

With memory management in C#, allocation and deallocation are handled automatically. Automatic memory management in C# is provided and managed by Common Language Runtime (CLR). In Microsoft Learn, Automatic Memory Management, Bill Wagner explains how memory is handled in C#. Wagner says, “When you initialize a new process, the runtime serves as a contiguous region of address space for the process”. This is also called the managed heap. All reference types, which include objects, are allocated on the managed heap. Memory is allocated contiguously on the heap for each object created by the application. Value types, such as int and char, are allocated on the stack. Both the stack and the managed heap are considered to be effective and efficient. Automatic memory allocation helps improve the efficiency of C#.

With C#, the garbage collector determines the best time to release and collect memory based on an optimizing engine. It bases the best time on the allocations (in the managed heap) being made in an application. Once it determines the best time, the garbage collector performs the appropriate tasks at the optimized time. Wagner says, “To optimize the performance of the garbage collector, the managed heap is divided into three generations: 0, 1, and 2”. The managed heap is divided into these generations for a few reasons. One reason is it’s faster to compact the memory together instead of dealing with the full heap. Another reason is newer objects tend to be accessed at the same time and related to one another. Newer objects are stored in generation 0

while objects that have survived garbage collections are promoted to generations 1 and 2. After performing collections on generation 0, the optimizing engine will determine if generations 1 and 2 need to go through a collection. This way of garbage collection and releasing memory is very effective and efficient. Dividing the heap into three generations helps the optimizing engine find the best time to perform a collection on each generation of the heap.

All these different details and aspects come together to create the versatile programming language C#. There are a handful of these details that are unique to C#, and there are some that are taken from other languages. They all come together to create an efficient and effective programming language. The scope handling, type system, and memory management are three prominent details that help define C# as an effective programming language.