A tour of the C# language

C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in .NET. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language in C# 11 and earlier. If you want to explore the language through interactive examples, try the introduction to C# tutorials.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;
class Hello
{
   static void Main()
   {
      Console.WriteLine("Hello, World");
   }
}
```

The "Hello, World" program starts with a using directive that references the System namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries.
A using directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the using directive, the program can use Console.WriteLine as shorthand for System.Console.WriteLine.
The Hello class declared by the "Hello, World" program has a single member, the method named Main. The Main method is declared with the static modifier. By convention, a static method named Main serves as the entry point of a C# program.
The output of the program is produced by the WriteLine method of the Console class in the System namespace. This class is provided by the standard class libraries, which, by default, are automatically referenced by the compiler.

Types and variables

A *type* defines the structure and behavior of any data in C#. The declaration of a type may include its members, base type, interfaces it implements, and operations permitted for that type.
A *variable* is a label that refers to an instance of a specific type.
There are two kinds of types in C#:

1. Value types
2. Reference types.

Value types

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, *nullable value types*, and *tuple value types*.

Value types	**Details**

Simple types	Signed integral, unsigned integral, unicode characters, IEEE binary floating-point, High-precision decimal floating-point, and boolean.
Enum types	User-defined types of the form enum E {...}. An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.
Struct types	User-defined types of the form struct S {...}
Nullable value types	Extensions of all other value types with a null value
Tuple value types	User-defined types of the form (T1, T2, ...)

Reference types

Reference types	Details
Class types	Ultimate base class of all other types: object. Unicode strings: string, which represents a sequence of UTF-16 code units. User-defined types of the form class C {...}
Interface types	User-defined types of the form interface I {...}
Array types	Single-dimensional, multi-dimensional, and jagged. For example: int[], int[,], and int[][]
Delegate types	User-defined types of the form delegate int D(...)

C# programs use *type declarations* to create new types. A type declaration specifies the name and the members of the new type. Six of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, delegate types, and tuple value types. You can also declare record types, either record struct, or record class. Record types have compiler-synthesized members. You use records primarily for storing values, with minimal associated behavior.

- A class type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.
- A struct type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and don't typically require heap allocation. Struct types don't support user-specified inheritance, and all struct types implicitly inherit from type object.
- An interface type defines a contract as a named set of public members.

A class or struct that implements an interface must provide implementations of the interface's members. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

- A delegate type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They're also similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.
You can explore more about C# in this tutorials.