C# Types

C# is a strongly typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method declaration specifies a name, number of parameters, and type and kind (value, reference, or output) for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types and more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library and user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following items:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The interface(s) it implements.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure all operations that are performed in your code are **type safe**. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error (in C#, bool is not convertible to int). The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

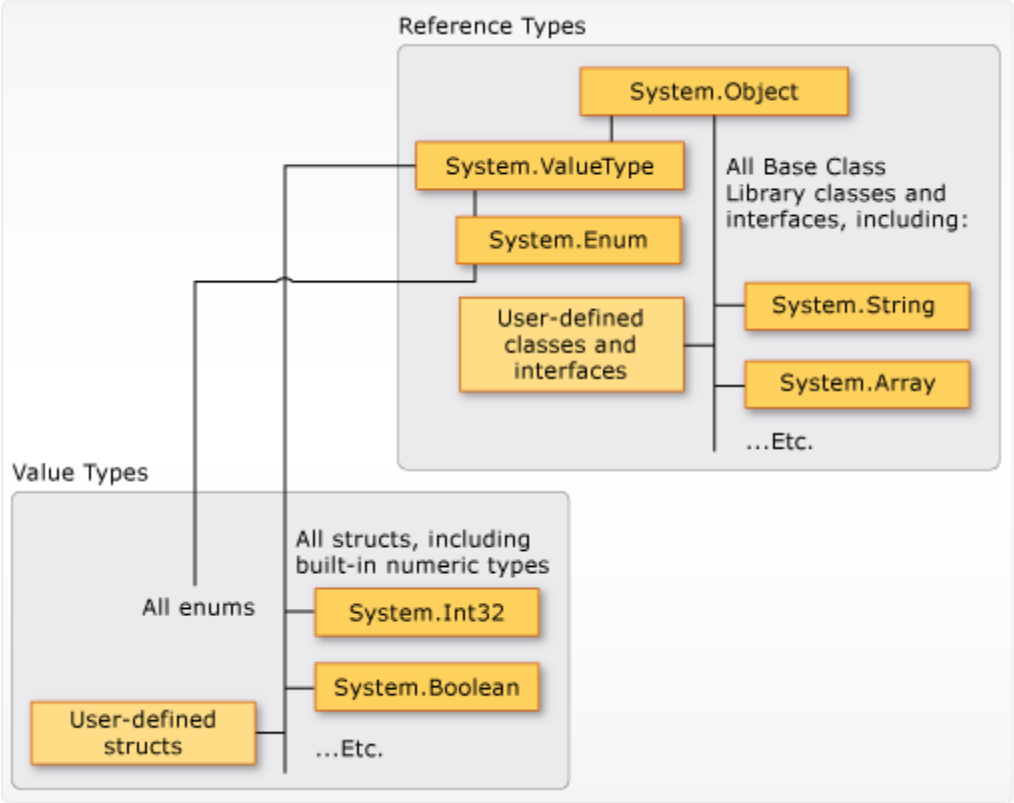
The common type system (CTS)

There are two kinds of **types** in C#: **value types** and **reference types**. Variables of value types directly contain their data whereas variables of reference types store references to their data, the latter being known as objects. With reference types, it's possible for two variables to reference the same object and possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for ref and out parameter variables).

It's important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called base types. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS).

- Each type in the CTS is defined as either a value type or a reference type. These types include all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are structs. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.



C#'s value types are further divided into simple types, enum types, struct types, nullable value types and tuple value types. C#'s reference types are further divided into class types, interface types, array types, and delegate types.

The following outline provides an overview of C#'s type system.

1. **Value types**
    a. **Simple types**
        i. **Signed integral**: `sbyte`, `short`, `int`, `long`
        ii. **Unsigned integral**: `byte`, `ushort`, `uint`, `ulong`
        iii. **Unicode characters**: `char`, which represents a UTF-16 code unit
        iv. **IEEE binary floating-point**: `float`, `double`
        v. **High-precision decimal floating-point**: `decimal`
        vi. **Boolean**: `bool`, which represents Boolean values—values that are either true or false
    b. **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.

    c. **Struct types:** User-defined types of the form `struct S {...}`
    d. **Nullable value types:** Extensions of all other value types with a null value
    e. **Tuple value types:** User-defined types of the form (T1, T2, ...)
2. **Reference types**
    a. **Class types**
        i. **Ultimate base class of all other types**: `object`
        ii. **Unicode strings**: string, which represents a sequence of UTF-16 code units
        iii. **User-defined types** of the form `class C {...}`
    b. **Interface types:** User-defined types of the form `interface I {...}`
    c. **Array types:** Single-dimensional, multi-dimensional and jagged. For example: `int[]`, `int[,]`, and `int[][]`
    d. **Delegate types:** User-defined types of the form `delegate int D(...)`

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.

C# supports single-dimensional and multi-dimensional **arrays** of any type. Unlike the types listed above, array types don't have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays, or a "jagged" array, of `int`.

Nullable types don't require a separate definition. For each non-nullable type T, there's a corresponding nullable type T?, which can hold an additional value, null. For instance, `int?` is a type that can hold any 32-bit integer or the value null, and `string?` is a type that can hold any string or the value null.

C#'s type system is unified such that a value of any type can be treated as an **`object`**. Every type in C# directly or indirectly derives from the object class type, and object is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type object. Values of value types are treated as objects by performing **boxing** and **unboxing** operations. In the following example, an int value is converted to object and back again to int.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

When a value of a value type is assigned to an object reference, a "box" is allocated to hold the value. That box is an instance of a reference type, and the value is copied into that box. Conversely, when an object reference is cast to a value type, a check is made that the referenced object is a box of the correct value type. If the check succeeds, the value in the box is copied to the value type.