INHERITANCE

1. C# and .NET support *single inheritance* only.
2. But can implement multiple interfaces at the same time.
3. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types.

4. Not all members of a base class are inherited by derived classes. The following members are not inherited:

    - Static constructors, which initialize the static data of a class.
    - Instance constructors, which you call to create a new instance of the class. Each class must define its own constructors.
    - Finalizers, which are called by the runtime's garbage collector to destroy instances of a class.

5. A member's accessibility affects its visibility for derived classes as follows:

    Private members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes.

    - Protected members are visible only in derived classes.
    - Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
    - Public members are visible in derived classes and are part of the derived class's public interface. Public inherited members can be called just as if they are defined in the derived class.
    - Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the virtual keyword. By default, base class members are not marked as `virtual` and cannot be overridden.
    - In some cases, a derived class *must* override the base class implementation. Base class members marked with the abstract keyword require that derived classes override them.
    - Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance.

ENUMERATION

An *enumeration type* (or *enum type*) is a value type defined by a set of named constants of the underlying integral numeric type.

- By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type.
- You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an extension method.
- The default value of an enumeration type `E` is the value produced by expression `(E)0`, even if zero doesn't have the corresponding enum member.
- You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices.

Structure types

A *structure type* (or *struct type*) is a value type that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type.

- Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. For structure-type variables, an instance of the type is copied.

- Typically, you use structure types to design small data-centric types that provide little or no behavior.

- Because structure types have value semantics, we recommend you define *immutable* structure types.

readonly struct

You use the `readonly` modifier to declare that a structure type is immutable. All data members of a `readonly` struct must be read-only as follows:

- Any field declaration must have the readonly modifier
- Any property, including auto-implemented ones, must be read-only. In C# 9.0 and later, a property may have an init accessor.

That guarantees that no member of a `readonly` struct modifies the state of the struct. That means that other instance members except constructors are implicitly readonly.

In a `readonly` struct, a data member of a mutable reference type still can mutate its own state. For example, you can't replace a **List<T>** instance, but you can add new elements to it.

`readonly` instance members

You can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct. If you can't declare the whole structure type as `readonly`, use the `readonly` modifier to mark the instance members that don't modify the state of the struct.

Within a `readonly` instance member, you can't assign to structure's instance fields. However, a `readonly` member can call a non-`readonly` member. In that case, the compiler creates a copy of the structure instance and calls the non-`readonly` member on that copy. As a result, the original structure instance isn't modified.

Typically, you apply the `readonly` modifier to the following kinds of instance members:

- methods:

```
public readonly double Sum()
{
    return X + Y;
}
```

You can also apply the `readonly` modifier to methods that override methods declared in System.Object:

```
public readonly override string ToString() => $"({X}, {Y})";
```

- properties and indexers:

```
private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}
```

If you need to apply the `readonly` modifier to both accessors of a property or indexer, apply it in the declaration of the property or indexer.

The compiler declares a `get` accessor of an **auto-implemented property** as `readonly`, regardless of presence of the `readonly` modifier in a property declaration.

In C# 9.0 and later, you may apply the `readonly` modifier to a property or indexer with an `init` accessor:

```
public readonly double X { get; init; }
```

You can apply the `readonly` modifier to static fields of a structure type, but not any other static members, such as properties or methods.

The compiler may make use of the `readonly` modifier for performance optimizations.

Nondestructive mutation

Beginning with C# 10, you can use the with expression to produce a copy of a structure-type instance with the specified properties and fields modified.

record struct

Beginning with C# 10, you can define record structure types. Record types provide built-in functionality for encapsulating data. You can define both `record struct` and `readonly record struct` types. A record struct can't be a ref struct.

Interface

An interface defines a contract. Any class or struct that implements that contract must provide an implementation of the members defined in the interface.

- An interface may define a default implementation for members. It may also define static members in order to provide a single implementation for common functionality.
- Beginning with C# 11, an interface may define `static abstract` or `static virtual` members to declare that an implementing type must provide the declared members.

An interface can be a member of a namespace or a class. An interface declaration can contain declarations (signatures without any implementation) of the following members:

- Methods
- Properties
- Indexers
- Events

Default interface members

These preceding member declarations typically don't contain a body. An interface member may declare a body. Member bodies in an interface are the *default implementation*. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation. An interface may include:

- Constants
- Operators
- Static constructor.
- Nested types

- Static fields, methods, properties, indexers, and events
- Member declarations using the explicit interface implementation syntax.
- Explicit access modifiers (the default access is public).

Static abstract and virtual members

Beginning with C# 11, an interface may declare `static abstract` and `static virtual` members for all member types except fields