Fields (C# Programming Guide)

Article • 07/30/2022 • 4 minutes to read

A *field* is a variable of any type that is declared directly in a class or struct. Fields are *members* of their containing type.

A class or struct may have instance fields, static fields, or both. Instance fields are specific to an instance of a type. If you have a class `T`, with an instance field `F`, you can create two objects of type `T`, and modify the value of `F` in each object without affecting the value in the other object. By contrast, a static field belongs to the type itself, and is shared among all instances of that type. You can access the static field only by using the type name. If you access the static field by an instance name, you get CS0176 compile-time error.

Generally, you should use fields only for variables that have private or protected accessibility. Data that your type exposes to client code should be provided through methods, properties, and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*.

Fields typically store the data that must be accessible to more than one type method and must be stored for longer than the lifetime of any single method. For example, a type that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that aren't used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class or struct block by specifying the access level of the field, followed by the type of the field, followed by the name of the field. For example:

C#

```csharp
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

To access a field in an instance, add a period after the instance name, followed by the name of the field, as in `instancename._fieldName`. For example:

```C#
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically assign the `Day` field to `"Monday"`, for example, you would declare `Day` as in the following example:

```C#
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it will overwrite any value given during field declaration. For more information, see Using Constructors.

Note

A field initializer cannot refer to other instance fields.

Fields can be marked as public, private, protected, internal, protected internal, or private protected. These access modifiers define how users of the type can access the fields. For more information, see Access Modifiers.

A field can optionally be declared static. Static fields are available to callers at any time, even if no instance of the type exists. For more information, see Static Classes and Static Class Members.

A field can be declared readonly. A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is similar to a constant, except that the C# compiler doesn't have access to the value of a static read-only field at compile time, only at run time. For more information, see Constants.

A field can be declared required. A required field must be initialized by the constructor, or by an object initializers when an object is created. You add the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute to any constructor declaration that initializes all required members.

The `required` modifier can't be combined with the `readonly` modifier on the same field.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- The C# type system
- Using Constructors
- Inheritance
- Access Modifiers
- Abstract and Sealed Classes and Class Members