

C# identifier naming rules and conventions

Article • 08/01/2023

An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace.

Naming rules

Valid identifiers must follow these rules:

- Identifiers must start with a letter or underscore (`_`).
- Identifiers may contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters. For more information on Unicode categories, see the [Unicode Category Database](#).

You can declare identifiers that match C# keywords by using the `@` prefix on the identifier. The `@` isn't part of the identifier name. For example, `@if` declares an identifier named `if`. These [verbatim identifiers](#) are primarily for interoperability with identifiers declared in other languages.

For a complete definition of valid identifiers, see the [Identifiers article in the C# Language Specification](#).

Naming conventions

In addition to the rules, there are many identifier [naming conventions](#) used throughout the .NET APIs. By convention, C# programs use `PascalCase` for type names, namespaces, and all public members. In addition, the `dotnet/docs` team uses the following conventions, adopted from the [.NET Runtime team coding style](#):

- Interface names start with a capital `I`.
- Attribute types end with the word `Attribute`.
- Enum types use a singular noun for nonflags, and a plural noun for flags.
- Identifiers shouldn't contain two consecutive underscore (`_`) characters. Those names are reserved for compiler-generated identifiers.
- Use meaningful and descriptive names for variables, methods, and classes.

- Avoid using single-letter names, except for simple loop counters. See exceptions for syntax examples noted in the following section.
- Prefer clarity over brevity.
- Use PascalCase for class names and method names.
- Use camelCase for method arguments, local variables, and private fields.
- Use PascalCase for constant names, both fields and local constants.
- Private instance fields start with an underscore (`_`).
- Static fields start with `s_`. Note that this isn't the default Visual Studio behavior, nor part of the [Framework design guidelines](#), but is configurable in `editorconfig`.
- Avoid using abbreviations or acronyms in names, except for widely known and accepted abbreviations.
- Use meaningful and descriptive namespaces that follow the reverse domain name notation.
- Choose assembly names that represent the primary purpose of the assembly.

The examples that describe the syntax of C# constructs often use single letter names that match the convention used in the [C# language specification](#):

- Use `s` for structs, `c` for classes.
- Use `m` for methods.
- Use `v` for variables, `p` for parameters.
- Use `r` for `ref` parameters.

The preceding single-letter names are allowed only in the language reference section.

In the following examples, guidance pertaining to elements marked `public` is applicable when working with `protected` and `protected internal` elements, all of which are intended to be visible to external callers.

Pascal case

Use pascal casing ("PascalCasing") when naming a `class`, `Interface`, `struct`, or `delegate` type.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(  
    string Street,  
    string City,  
    string StateOrProvince,  
    string ZipCode);
```

C#

```
public struct ValueCoordinate  
{  
}
```

C#

```
public delegate void DelegateType(string message);
```

When naming an `interface`, use pascal casing in addition to prefixing the name with an `I`. This prefix clearly indicates to consumers that it's an `interface`.

C#

```
public interface IWorkerQueue  
{  
}
```

When naming `public` members of types, such as fields, properties, events, use pascal casing. Also, use pascal casing for all methods and local functions.

C#

```
public class ExampleEvents  
{  
    // A public field, these should be used sparingly  
    public bool IsValid;  
  
    // An init-only property  
    public IWorkerQueue WorkerQueue { get; init; }  
  
    // An event  
    public event Action EventProcessing;  
  
    // Method  
    public void StartEventProcessing()  
    {  
        // Local function  
        static int CountQueueItems() => WorkerQueue.Count;  
        // ...  
    }  
}
```

```
}  
}
```

When writing positional records, use pascal casing for parameters as they're the public properties of the record.

C#

```
public record PhysicalAddress(  
    string Street,  
    string City,  
    string StateOrProvince,  
    string ZipCode);
```

For more information on positional records, see [Positional syntax for property definition](#).

Camel case

Use camel casing ("camelCasing") when naming `private` or `internal` fields and prefix them with `_`. Use camel casing when naming local variables, including instances of a delegate type.

C#

```
public class DataService  
{  
    private IWorkerQueue _workerQueue;  
}
```

Tip

When editing C# code that follows these naming conventions in an IDE that supports statement completion, typing `_` will show all of the object-scoped members.

When working with `static` fields that are `private` or `internal`, use the `s_` prefix and for thread static use `t_`.

C#

```
public class DataService  
{
```

```
private static IWorkerQueue s_workerQueue;

[ThreadStatic]
private static TimeSpan t_timeSpan;
}
```

When writing method parameters, use camel casing.

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

For more information on C# naming conventions, see [C# Coding Style](#) .

Type parameter naming guidelines

The following guidelines apply to type parameters on generic type parameters. These are the placeholders for arguments in a generic type or a generic method. You can read more about [generic type parameters](#) in the C# programming guide.

- **Do** name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name would not add value.

./snippets/coding-conventions

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Consider** using `T` as the type parameter name for types with one single letter type parameter.

./snippets/coding-conventions

```
public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- **Do** prefix descriptive type parameter names with "T".

./snippets/coding-conventions

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- **Consider** indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` may be called `TSession`.

The code analysis rule [CA1715](#) can be used to ensure that type parameters are named appropriately.

Extra naming conventions

- Examples that don't include [using directives](#), use namespace qualifications. If you know that a namespace is imported by default in a project, you don't have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they're too long for a single line, as shown in the following example.

C#

```
var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- You don't have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.