NAMESPACES

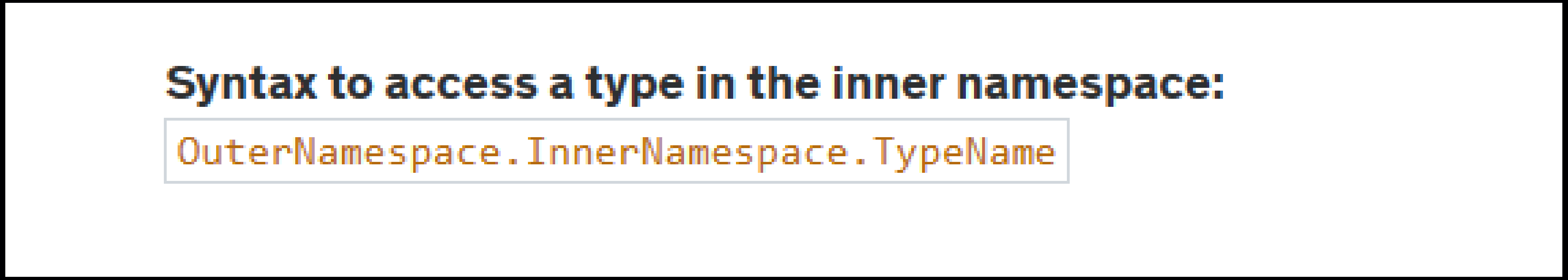
Namespaces is a collection of classes and "other types such as interfaces, structures, delegate types, enumerations).

Namespaces goal is to group-up classes and other types that are related to a particular project-module, into an unit.

Syntax to access a type that is present inside the namespace: NamespaceName.TypeName

Nested Namespaces

The namespace which is declared inside another namespace is called as "Nested namespace" or "Inner Namespace".
Use nested namespaces, in order to divide the classes of a larger namespace, into smaller groups.



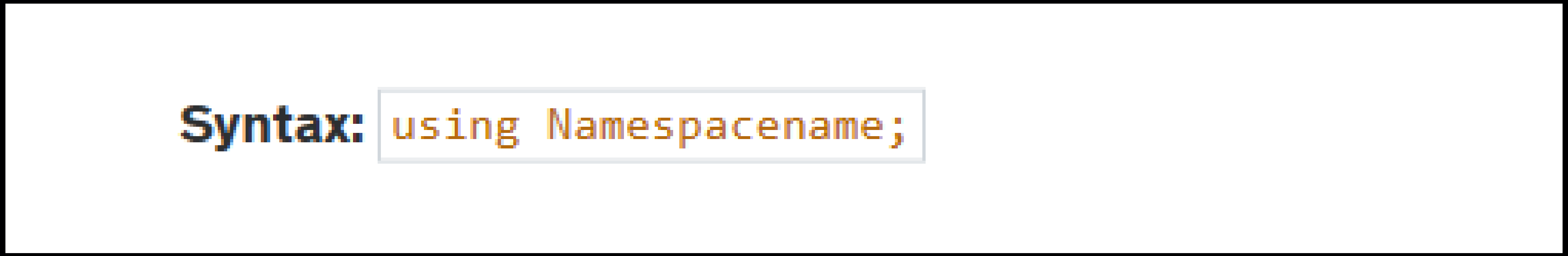
Syntax to access a type in the inner namespace:

```
OuterNamespace.InnerNamespace.TypeName
```

NAMESPACES

Importing Namespaces ('using' Directive)

The "using" is a directive statement (top-level statement) that should be placed at the top of the file, which specifies the namespace, from which you want to import all the classes and other types.



Syntax:
```
using Namespacename;
```

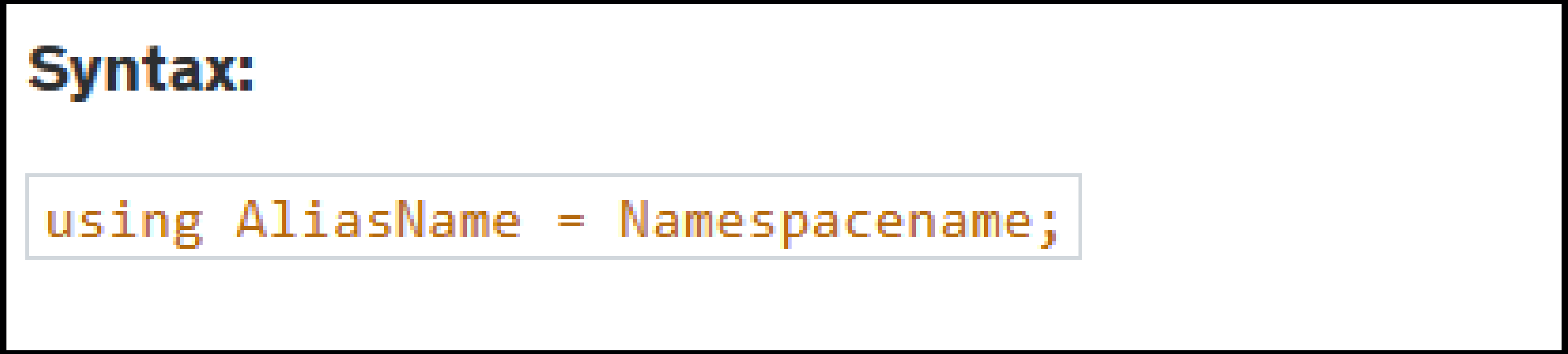
When you import a namespace, you can directly access all of its classes and other types (but not inner namespaces).

The "using directives" are written independently for every file.
"One using directive" can import "one namespace" only.

NAMESPACES

'using' Alias Name

The "using alias" directive allows you to create "alias name" for the namespace.



Syntax:

```
using AliasName = Namespacename;
```

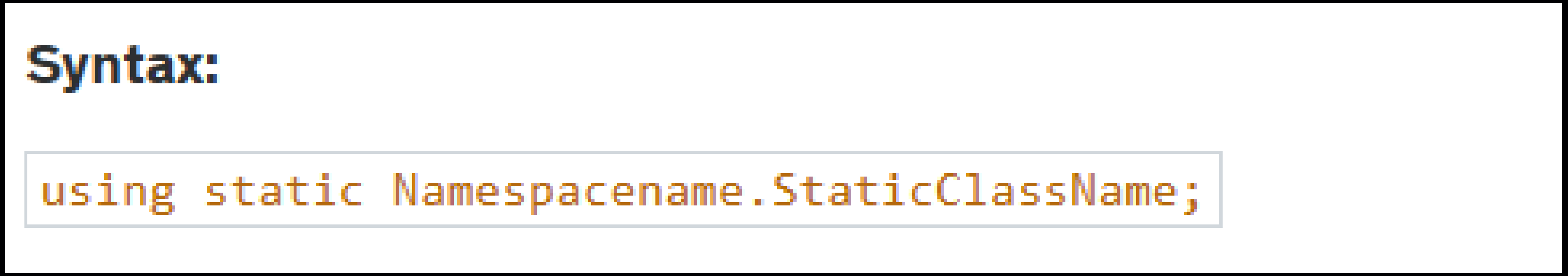
Use "using alias" directive, if you want to access long namespaces with shortcut name.

It is much useful to access specific namespace, when there is namespace name ambiguity (two classes with same name in two different namespaces and both namespaces are imported in the same file).

NAMESPACES

'using' static

The "using static" directive allows you import a static class directly from a namespace; so that you can directly access any of its methods anywhere in the current file.



Syntax:

```
using static Namespacename.StaticClassName;
```

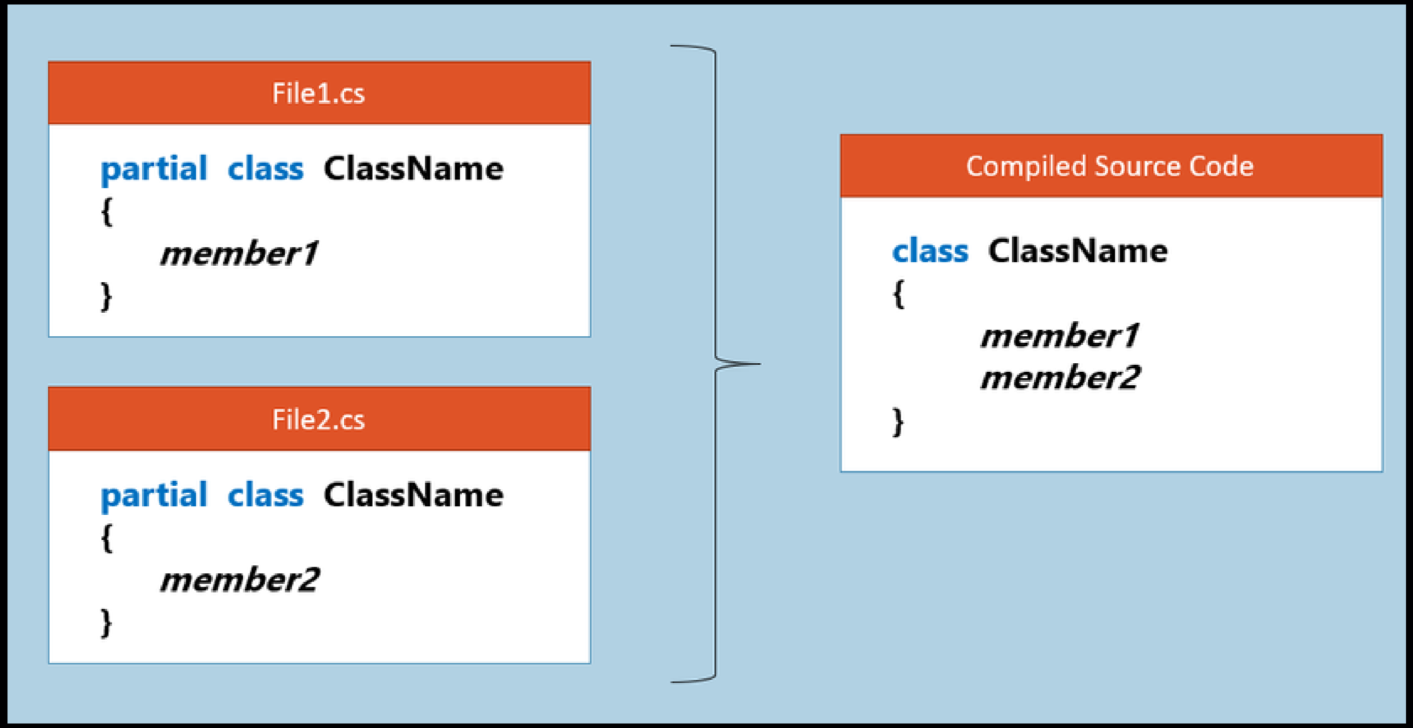
Use the "using static" directive to access methods of static class easily, without repeating the class name each time.

PARTIAL CLASSES

Partial Class is a class that splits into multiple files. Each file is treated as a "part of the class".

Rules:

- At compilation time, all partial classes that have the same name, become a "single class".
- All the partial classes (that want to be a part of a class) should have the same name should be in the same namespace and same assembly & and should have the same access-modifier (such as 'internal' or 'public').
- Duplicate members are not allowed in partial classes.
- Any attributes/modifiers (such as abstract, sealed) applied on one partial class, will be applied to all partial classes that have the same name.
- The 'partial' keyword can be used only before the keywords 'class', 'struct', 'interface', and 'void'.
- Each partial class can be developed individually, by different developers/teams.



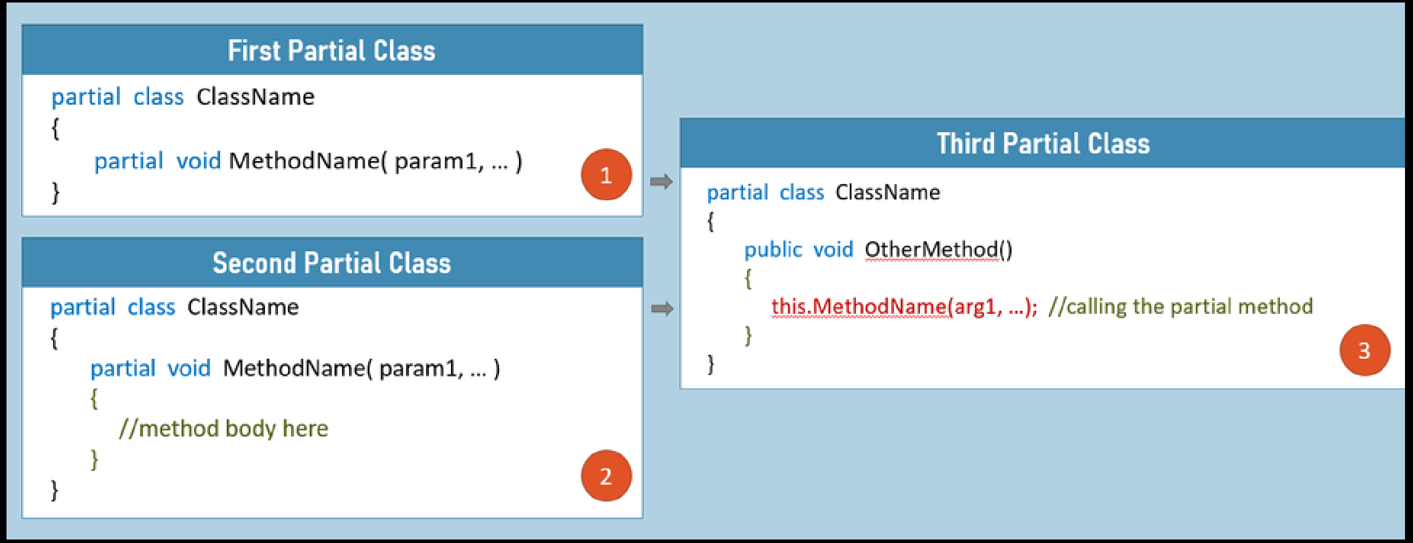
```
partial class ClassName
{
    member1
}
```

```
partial class ClassName
{
    member2
}
```

```
class ClassName
{
    member1
    member2
}
```

PARTIAL METHODS

Partial Methods are "declared in one partial class" (just like abstract method), and "implemented in another partial class", that have same name.



```
partial class ClassName
{
    partial void MethodName( param1, ... )
}
```

```
partial class ClassName
{
    partial void MethodName( param1, ... )
    {
        //method body here
    }
}
```

```
partial class ClassName
{
    public void OtherMethod()
    {
        this.MethodName(arg1, ...); //calling the partial method
    }
}
```

Rules:

- Partial Methods can only be created in partial class or partial structs.
- Partial Methods are implicitly private. It can't have any other access modifier.
- Partial Methods can have only "void" return type.
- Implementation of partial methods is optional. If there is no implementation of partial methods in any parts of the partial class, the method calls are removed by the compiler, at compilation time.
- If you are building large class libraries and decide extension of methods to other developers, partial methods can be used.

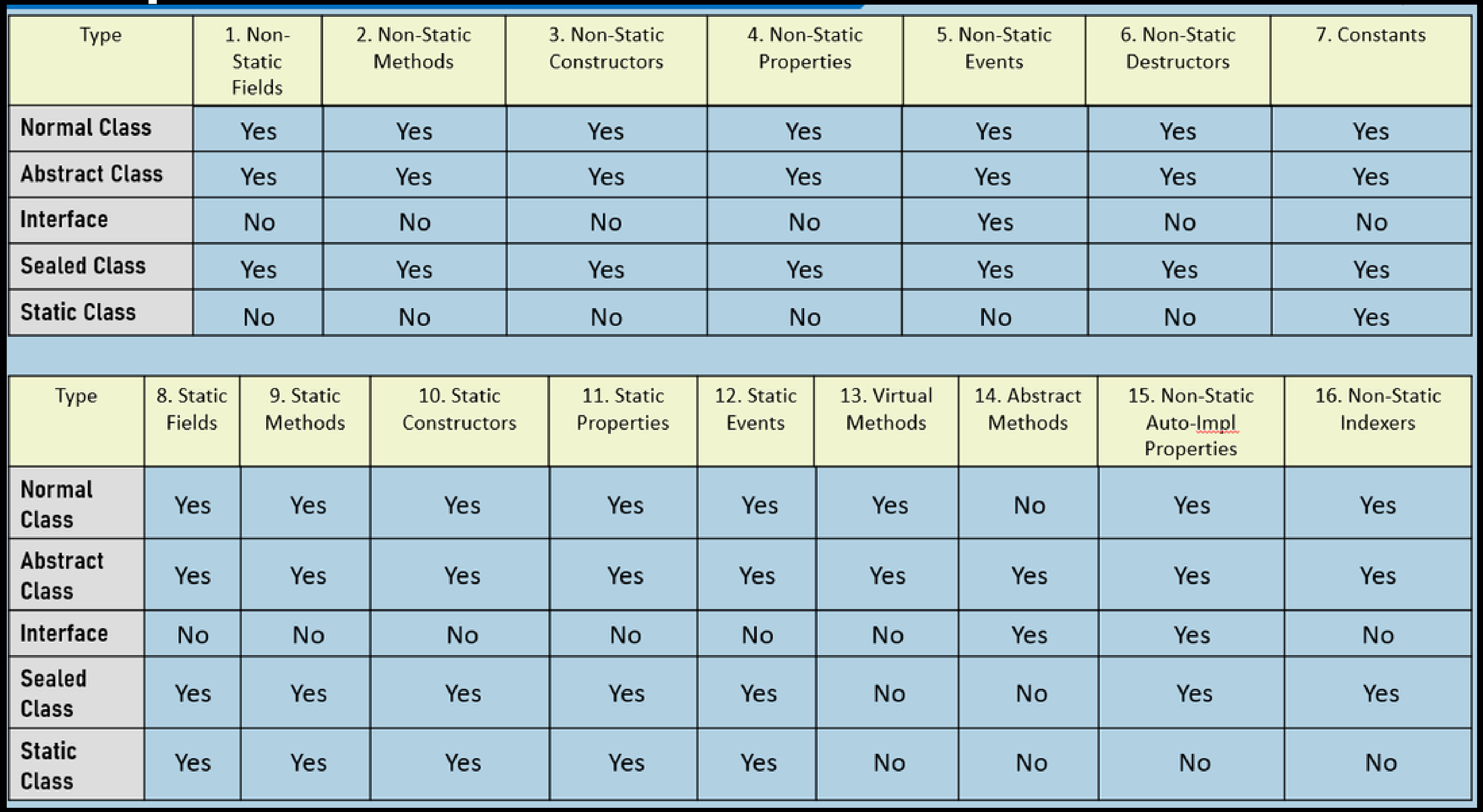
STATIC CLASSES

A static class is a class that can contain only "static members". If you don't want even a single 'instance member' (non-static member), then use 'static class'.

Advantage:

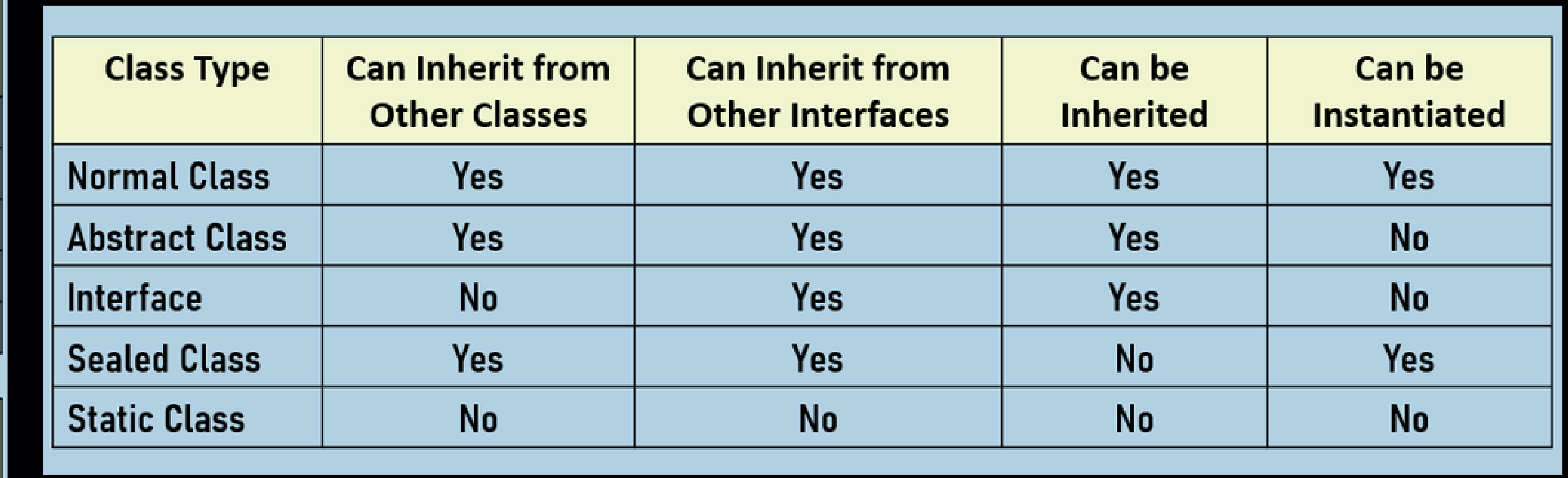
We can avoid accidental creation of object for the class, by making it as "static class".

Comparison Table: Class [vs] Static Class



Type	1. Non-Static Fields	2. Non-Static Methods	3. Non-Static Constructors	4. Non-Static Properties	5. Non-Static Events	6. Non-Static Destructors	7. Constants
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	Yes	No	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Static Class	No	No	No	No	No	No	Yes

Type	8. Static Fields	9. Static Methods	10. Static Constructors	11. Static Properties	12. Static Events	13. Virtual Methods	14. Abstract Methods	15. Non-Static Auto-Impl Properties	16. Non-Static Indexers
Normal Class	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Abstract Class	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface	No	No	No	No	No	No	Yes	Yes	No
Sealed Class	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Static Class	Yes	Yes	Yes	Yes	Yes	No	No	No	No

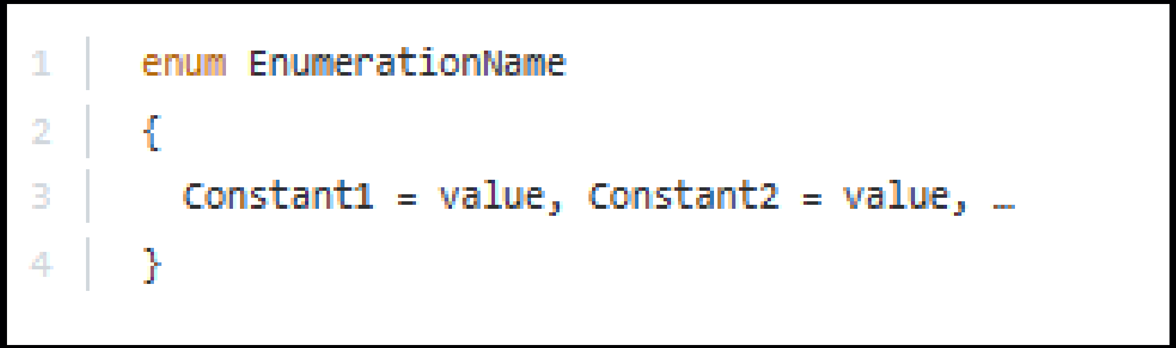


Class Type	Can Inherit from Other Classes	Can Inherit from Other Interfaces	Can be Inherited	Can be Instantiated
Normal Class	Yes	Yes	Yes	Yes
Abstract Class	Yes	Yes	Yes	No
Interface	No	Yes	Yes	No
Sealed Class	Yes	Yes	No	Yes
Static Class	No	No	No	No

ENUMERATIONS

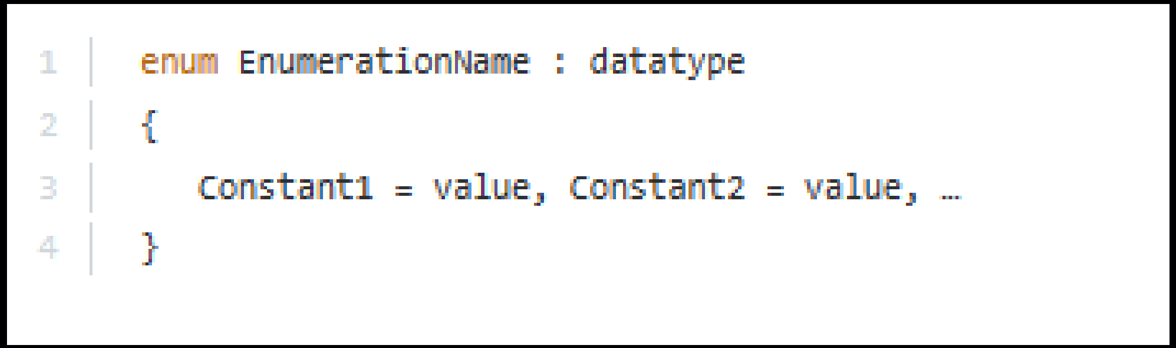
Enumeration is a collection of constants. Enumeration is used to specify the list of options allowed to be stored in a field / variable. Use enumeration if you don't want to allow other developers to assign other value into a field / variable, other than the list of values specified in the enumeration

By default, each constant will be assigned to a number, starts from zero; however you can change the number (integer only).

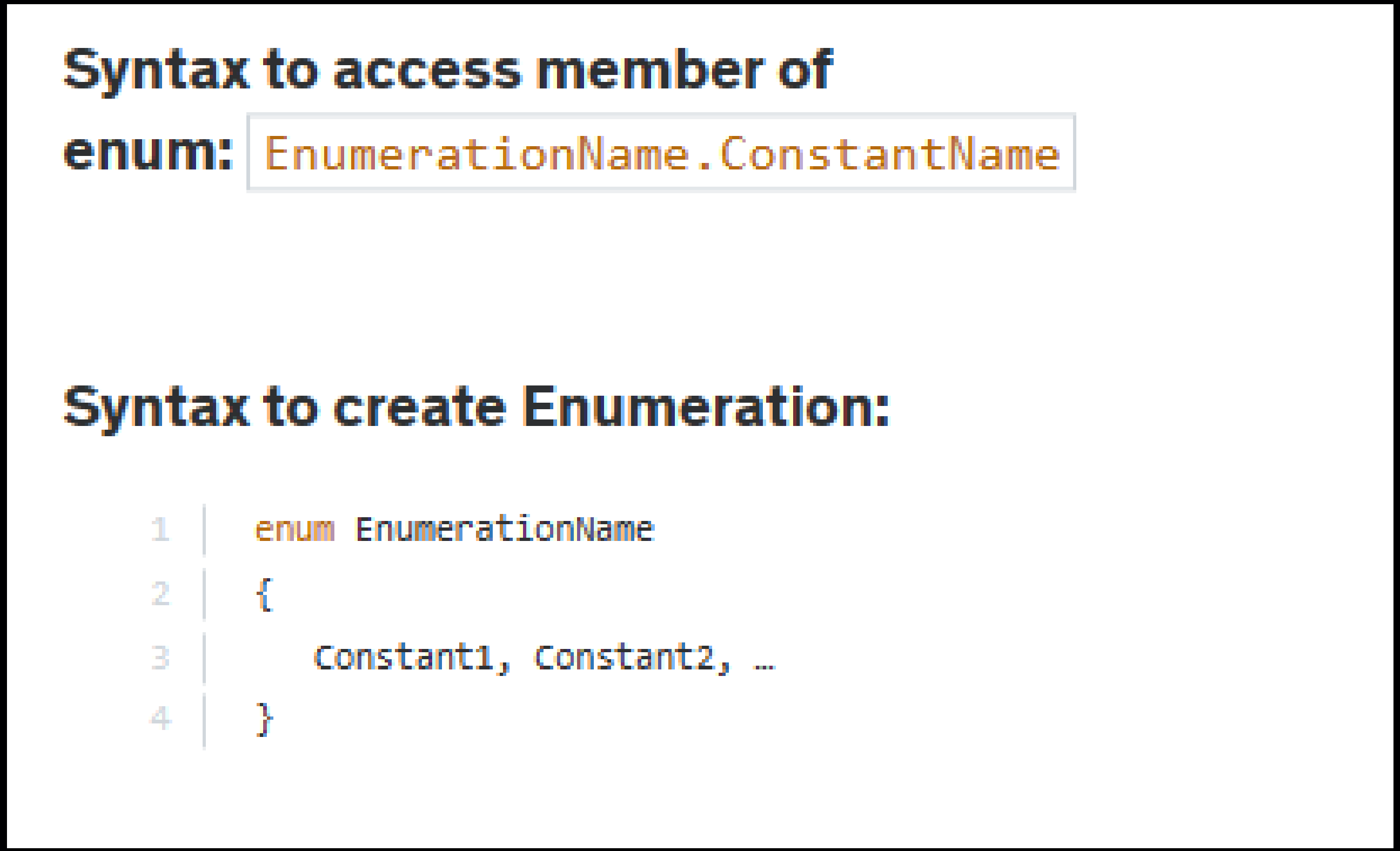


```
enum EnumerationName
{
  Constant1 = value, Constant2 = value, …
}
```

The default data type of enum member is "int". However, you can change its data type as follows.



```
enum EnumerationName : datatype
{
   Constant1 = value, Constant2 = value, …
}
```



Syntax to access member of enum: `EnumerationName.ConstantName`

Syntax to create Enumeration:

```
enum EnumerationName
{
   Constant1, Constant2, …
}
```