

Coding conventions serve the following purposes:

- They create a consistent look to the code, so that readers can focus on content, not layout.
- They enable readers to understand the code more quickly by making assumptions based on previous experience.
- They facilitate copying, changing, and maintaining the code.
- They demonstrate C# best practices.

Naming Conventions

- In short examples that do not include [using directives](#), use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.
- C#Copy
- ```
var currentPerformanceCounterCategory = new
System.Diagnostics.
PerformanceCounterCategory();
```
- You do not have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

## Layout Conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](#).
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines are not indented automatically, indent them one tab stop (four spaces).

- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.
- C#Copy
- ```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

Commenting Conventions

- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.
- C#Copy
- ```
// The following declaration creates a query. It does
not run
// the query.
```
- Do not create formatted blocks of asterisks around comments.

## Language Guidelines

The following sections describe practices that the C# team follows to prepare code examples and samples.

### String Data Type

- Use the `+` operator to concatenate short strings, as shown in the following code.
- C#Copy

- `string` `displayName = nameList[n].LastName + ", " + nameList[n].FirstName;`
- To append strings in loops, especially when you are working with large amounts of text, use a [StringBuilder](#) object.
- C#Copy
- ```
var phrase =
    "lalalalalalalalalalalalalalalalalalalalalalalalalalalala
    lalala";

var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}

//Console.WriteLine("tra" + manyPhrases);
```

Implicitly Typed Local Variables

- Use [implicit typing](#) for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.
- C#Copy
- ```
// When the type of a variable is clear from the
context, use var
// in the declaration.
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
```
- Do not use [var](#) when the type is not apparent from the right side of the assignment.
- C#Copy
- ```
// When the type of a variable is not clear from the
context, use an
// explicit type.
```

```
int var4 = ExampleClass.ResultSoFar();
```

- Do not rely on the variable name to specify the type of the variable. It might not be correct.

- C#Copy

- `// Naming the following variable inputInt is misleading.`

```
// It is a string.
```

```
var inputInt = Console.ReadLine();
```

```
Console.WriteLine(inputInt);
```

- Avoid the use of `var` in place of [dynamic](#).
- Use implicit typing to determine the type of the loop variable in [for](#) and [foreach](#) loops.
- The following example uses implicit typing in a `for` statement.

- C#Copy

- `var syllable = "ha";`

```
var laugh = "";
```

```
for (var i = 0; i < 10; i++)
```

```
{
```

```
    laugh += syllable;
```

```
    Console.WriteLine(laugh);
```

```
}
```

- The following example uses implicit typing in a `foreach` statement.

- C#Copy

- `foreach (var ch in laugh)`

```
{
```

```
    if (ch == 'h')
```

```
        Console.Write("H");
```

```
    else
```

```
        Console.Write(ch);
```

```
}
```

```
Console.WriteLine();
```

Unsigned Data Type

- In general, use `int` rather than unsigned types. The use of `int` is common throughout C#, and it is easier to interact with other libraries when you use `int`.

Arrays

- Use the concise syntax when you initialize arrays on the declaration line.
- C#Copy
- ```
// Preferred syntax. Note that you cannot use var here
instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize
the elements one at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

## Delegates

- Use the concise syntax to create instances of a delegate type.
- C#Copy
- ```
// First, in class Program, define the delegate type
and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);
```

```
// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- C#Copy
- // In the Main method, create an instance of Del.

```
// Preferred: Create an instance of Del by using
condensed syntax.
```

```
Del exampleDel2 = DelMethod;
```

```
// The following declaration uses the full syntax.
```

```
Del exampleDel1 = new Del(DelMethod);
```

try-catch and using Statements in Exception Handling

- Use a [try-catch](#) statement for most exception handling.
- C#Copy
- ```
static string GetValueFromArray(string[] array, int
index)
{
 try
 {
 return array[index];
 }
 catch (System.IndexOutOfRangeException ex)
 {
 Console.WriteLine("Index is out of range: {0}",
index);
 throw;
 }
}
```

- Simplify your code by using the C# [using statement](#). If you have a [try-finally](#) statement in which the only code in the `finally` block is a call to the [Dispose](#) method, use a `using` statement instead.

- C#Copy

- `// This try-finally statement only calls Dispose in the finally block.`

```
Font font1 = new Font("Arial", 10.0f);
try
{
 byte charset = font1.GdiCharSet;
}
finally
{
 if (font1 != null)
 {
 ((IDisposable) font1).Dispose();
 }
}
```

```
// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
 byte charset = font2.GdiCharSet;
}
```

## && and || Operators

- To avoid exceptions and increase performance by skipping unnecessary comparisons, use [&&](#) instead of [&](#) and [||](#) instead of [|](#) when you perform comparisons, as shown in the following example.

- C#Copy

- `Console.Write("Enter a dividend: ");`  
`var dividend = Convert.ToInt32(Console.ReadLine());`

```
Console.Write("Enter a divisor: ");
```

```

var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the
// following condition
// causes a run-time error. The && operator short
// circuits when the
// first expression is false. That is, it does not
// evaluate the
// second expression. The & operator evaluates both,
// and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
 Console.WriteLine("Quotient: {0}", dividend /
divisor);
}
else
{
 Console.WriteLine("Attempted division by 0 ends up
here.");
}

```

## New Operator

- Use the concise form of object instantiation, with implicit typing, as shown in the following declaration.
- C#Copy
- `var instance1 = new ExampleClass();`
- The previous line is equivalent to the following declaration.
- C#Copy
- `ExampleClass instance2 = new ExampleClass();`
- Use object initializers to simplify object creation.
- C#Copy



- `// Object initializer.`  
`var instance3 = new ExampleClass { Name = "Desktop", ID`  
`= 37414,`  
`Location = "Redmond", Age = 2.3 };`  
  
`// Default constructor and assignment statements.`  
`var instance4 = new ExampleClass();`  
`instance4.Name = "Desktop";`  
`instance4.ID = 37414;`  
`instance4.Location = "Redmond";`  
`instance4.Age = 2.3;`

## Event Handling

- If you are defining an event handler that you do not need to remove later, use a lambda expression.

- C#Copy
- `public Form2()`  
`{`  
`// You can use a lambda expression to define an`  
`event handler.`  
`this.Click += (s, e) =>`  
`{`  
`MessageBox.Show(`  
  
`((MouseEventArgs)e).Location.ToString());`  
`};`  
`}`

- C#Copy
- `// Using a lambda expression shortens the following`  
`traditional definition.`  
`public Form1()`  
`{`  
`this.Click += new EventHandler(Form1_Click);`  
`}`

```

void Form1_Click(object sender, EventArgs e)
{

 MessageBox.Show(((MouseEventArgs)e).Location.ToString()
);
}

```

## Static Members

- Call [static](#) members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Do not qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

## LINQ Queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers who are located in Seattle.
- C#Copy
- ```
var seattleCustomers = from cust in customers
                        where cust.City == "Seattle"
                        select cust.Name;
```
- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.
- C#Copy
- ```
var localDistributors =
 from customer in customers
 join distributor in distributors on customer.City
 equals distributor.City
 select new { Customer = customer, Distributor =
distributor };
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as `Name` and `ID` in the result, rename them to clarify that `Name` is the name of a customer, and `ID` is the ID of a distributor.

- C#Copy

```
var localDistributors2 =
 from cust in customers
 join dist in distributors on cust.City equals
dist.City
 select new { CustomerName = cust.Name,
DistributorID = dist.ID };
```

- Use implicit typing in the declaration of query variables and range variables.

- C#Copy

```
var seattleCustomers = from cust in customers
 where cust.City == "Seattle"
 select cust.Name;
```

- Align query clauses under the `from` clause, as shown in the previous examples.
- Use `where` clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

- C#Copy

```
var seattleCustomers2 = from cust in customers
 where cust.City == "Seattle"
 orderby cust.Name
 select cust;
```

- Use multiple `from` clauses instead of a `join` clause to access inner collections. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

- C#Copy

```
// Use a compound from to access the inner sequence
within each element.
var scoreQuery = from student in students
```

```
from score in student.Scores
where score > 90
select new { Last = student.LastName,
score };
```