Using type dynamic (C# Programming Guide)

Article • 09/15/2021 • 5 minutes to read • 12 contributors

In this article

C# 4 introduces a new type, `dynamic`. The type is a static type, but an object of type `dynamic` bypasses static type checking. In most cases, it functions like it has type `object`. At compile time, an element that is typed as `dynamic` is assumed to support any operation. Therefore, you do not have to be concerned about whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program. However, if the code is not valid, errors are caught at run time.

For example, if instance method `exampleMethod1` in the following code has only one parameter, the compiler recognizes that the first call to the method, `ec.exampleMethod1(10, 4)`, is not valid because it contains two arguments. The call causes a compiler error. The second call to the method, `dynamic_ec.exampleMethod1(10, 4)`, is not checked by the compiler because the type of `dynamic_ec` is `dynamic`. Therefore, no compiler error is reported. However, the error does not escape notice indefinitely. It is caught at run time and causes a run-time exception.

C#

```csharp
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);
```

```
        dynamic dynamic_ec = new ExampleClass();
        // The following line is not identified as an error by the
        // compiler, but it causes a run-time exception.
        dynamic_ec.exampleMethod1(10, 4);

        // The following calls also do not cause compiler errors, whether
        // appropriate methods exist or not.
        dynamic_ec.someMethod("some argument", 7, null);
        dynamic_ec.nonexistentMethod();
    }
```

C#

Copy

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

The role of the compiler in these examples is to package together information about what each statement is proposing to do to the object or expression that is typed as `dynamic`. At run time, the stored information is examined, and any statement that is not valid causes a run-time exception.

The result of most dynamic operations is itself `dynamic`. For example, if you rest the mouse pointer over the use of `testSum` in the following example, IntelliSense displays the type **(local variable) dynamic testSum**.

C#

Copy

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Operations in which the result is not `dynamic` include:

- Conversions from `dynamic` to another type.
- Constructor calls that include arguments of type `dynamic`.

For example, the type of `testInstance` in the following declaration is `ExampleClass`, not `dynamic`:

C#

Copy

```
var testInstance = new ExampleClass(d);
```

Conversion examples are shown in the following section, "Conversions."

Conversions

Conversions between dynamic objects and other types are easy. This enables the developer to switch between dynamic and non-dynamic behavior.

Any object can be converted to dynamic type implicitly, as shown in the following examples.

C#

Copy

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Conversely, an implicit conversion can be dynamically applied to any expression of type `dynamic`.

C#

Copy

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Overload resolution with arguments of type dynamic

Overload resolution occurs at run time instead of at compile time if one or more of the arguments in a method call have the type `dynamic`, or if the receiver of the method call is of type `dynamic`. In the following example, if the only accessible `exampleMethod2` method is defined to take a string argument, sending `d1` as the argument does not cause a compiler error, but it does cause a run-time exception. Overload resolution fails at run time because the run-time type of `d1` is `int`, and `exampleMethod2` requires a string.

C#

Copy

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic language runtime

The dynamic language runtime (DLR) is an API that was introduced in .NET Framework 4. It provides the infrastructure that supports the `dynamic` type in C#, and also the implementation of dynamic programming languages such as IronPython and IronRuby. For more information about the DLR, see Dynamic Language Runtime Overview.

COM interop

C# 4 includes several features that improve the experience of interoperating with COM APIs such as the Office Automation APIs. Among the improvements are the use of the `dynamic` type, and of named and optional arguments.

Many COM methods allow for variation in argument types and return type by designating the types as `object`. This has necessitated explicit casting of the values to coordinate with strongly typed variables in C#. If you compile by using the **EmbedInteropTypes** (C# Compiler Options) option, the introduction of the `dynamic` type enables you to treat the occurrences of `object` in COM signatures as if they were of type `dynamic`, and thereby to avoid much of the casting. For example, the following statements contrast how you access a cell in a Microsoft Office Excel spreadsheet with the `dynamic` type and without the `dynamic` type.

C#

Copy

```
// Before the introduction of dynamic.
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

C#

Copy

```
// After the introduction of dynamic, the access to the Value property and
// the conversion to Excel.Range are handled by the run-time COM binder.
excelApp.Cells[1, 1].Value = "Name";
Excel.Range range2010 = excelApp.Cells[1, 1];
```

Related topics

Title	Description
dynamic	Describes the usage of the `dynamic` keyword.
Dynamic Language Runtime Overview	Provides an overview of the DLR, which is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).
Walkthrough: Creating and Using Dynamic Objects	Provides step-by-step instructions for creating a custom dynamic object and for creating a project that accesses an `IronPython` library.
How to access Office interop objects by using C# features	Demonstrates how to create a project that uses named and optional arguments, the `dynamic` type, and other enhancements that simplify access to Office API objects.

Recommended content

?? and ??= operators - C# reference

Learn about ?? and ??= which are the C# null-coalescing operators.

Tuple types - C# reference

Learn about C# tuples: lightweight data structures that you can use to group loosely related data elements

params keyword for parameter arrays - C# reference

params keyword for parameter arrays - C# reference

$ - string interpolation - C# reference

String interpolation provides a more readable and convenient syntax to format string output than traditional string composite formatting.

Nullable value types - C# reference

Learn about C# nullable value types and how to use them

Selection statements - C# reference

Learn about C# selection statements: if and switch.

Static Constructors - C# Programming Guide

A static constructor in C# initializes static data or performs an action done only once. It runs before the first instance is created or static members are referenced.

Extension Methods - C# Programming Guide

Extension methods in C# enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

Show more