Built-in reference types (C# reference)

Article • 02/25/2023

C# has many built-in reference types. They have keywords or operators that are synonyms for a type in the .NET library.

The object type

The `object` type is an alias for System.Object in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from System.Object. You can assign values of any type to variables of type `object`. Any `object` variable can be assigned to its default value using the literal `null`. When a variable of a value type is converted to object, it's said to be *boxed*. When a variable of type `object` is converted to a value type, it's said to be *unboxed*. For more information, see Boxing and Unboxing.

The string type

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for System.String in .NET.

Although `string` is a reference type, the equality operators == and != are defined to compare the values of `string` objects, not references. Value based equality makes testing for string equality more intuitive. For example:

C#

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

The previous example displays "True" and then "False" because the content of the strings is equivalent, but `a` and `b` don't refer to the same string instance.

The + operator concatenates strings:

C#

```
string a = "good " + "morning";
```

The preceding code creates a string object that contains "good morning".

Strings are *immutable*--the contents of a string object can't be changed after the object is created. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to `b`. The memory that had been allocated for `b` (when it contained the string "h") is then eligible for garbage collection.

C#

```
string b = "h";
b += "ello";
```

The `[]` operator can be used for readonly access to individual characters of a string. Valid index values start at `0` and must be less than the length of the string:

C#

```
string str = "test";
char x = str[2];  // x = 's';
```

In similar fashion, the `[]` operator can also be used for iterating over each character in a string:

C#

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

String literals

String literals are of type `string` and can be written in three forms, raw, quoted, and verbatim.

Raw string literals are available beginning in C# 11. Raw string literals can contain arbitrary text without requiring escape sequences. Raw string literals can include

whitespace and new lines, embedded quotes, and other special characters. Raw string literals are enclosed in a minimum of three double quotation marks (""""):

C#

```
"""
This is a multi-line
    string literal with the second line indented.
"""
```

You can even include a sequence of three (or more) double quote characters. If your text requires an embedded sequence of quotes, you start and end the raw string literal with more quote marks, as needed:

C#

```
"""""""
This raw string literal has four """", count them: """" four!
embedded quote characters in a sequence. That's why it starts and
ends
with five double quotes.

You could extend this example with as many embedded quotes as needed
for your text.
"""""""
```

Raw string literals typically have the starting and ending quote sequences on separate lines from the embedded text. Multiline raw string literals support strings that are themselves quoted strings:

C#

```
var message = """
"This is a very important message."
""";
Console.WriteLine(message);
// output: "This is a very important message."
```

When the starting and ending quotes are on separate lines, the newlines following the opening quote and preceding the ending quote aren't included in the final content. The closing quote sequence dictates the leftmost column for the string literal. You can indent a raw string literal to match the overall code format:

C#

```
var message = """
    "This is a very important message."
```

```
    """;
Console.WriteLine(message);
// output: "This is a very important message."
// The leftmost whitespace is not part of the raw string literal
```

Columns to the right of the ending quote sequence are preserved. This behavior enables raw strings for data formats such as JSON, YAML, or XML, as shown in the following example:

C#

```
var json= """
    {
        "prop": 0
    }
    """;
```

The compiler issues an error if any of the text lines extend to the left of the closing quote sequence. The opening and closing quote sequences can be on the same line, providing the string literal neither starts nor ends with a quote character:

C#

```
var shortText = """He said "hello!" this morning.""";
```

You can combine raw string literals with string interpolation to include quote characters and braces in the output string.

Quoted string literals are enclosed in double quotation marks ("):

C#

```
"good morning"  // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\` for backslash, `\u0066` for the letter f, and `\n` for newline.

C#

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

> **Note**
>
> The escape code `\udddd` (where `dddd` is a four-digit number) represents the Unicode character U+`dddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddddd`.

Verbatim string literals start with `@` and are also enclosed in double quotation marks. For example:

```
C#

@"good morning"  // a string literal
```

The advantage of verbatim strings is that escape sequences *aren't* processed, which makes it easy to write. For example, the following text matches a fully qualified Windows file name:

```
C#

@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

```
C#

@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

UTF-8 string literals

Strings in .NET are stored using UTF-16 encoding. UTF-8 is the standard for Web protocols and other important libraries. Beginning in C# 11, you can add the `u8` suffix to a string literal to specify UTF-8 encoding. UTF-8 literals are stored as `ReadOnlySpan<byte>` objects. The natural type of a UTF-8 string literal is `ReadOnlySpan<byte>`. Using a UTF-8 string literal creates a more clear declaration than declaring the equivalent System.ReadOnlySpan<T>, as shown in the following code:

```
C#

ReadOnlySpan<byte> AuthWithTrailingSpace = new byte[] { 0x41, 0x55,
0x54, 0x48, 0x20 };
ReadOnlySpan<byte> AuthStringLiteral = "AUTH "u8;
```

To store a UTF-8 string literal as an array requires the use of ReadOnlySpan<T>.ToArray() to copy the bytes containing the literal to the mutable array:

```
C#

byte[] AuthStringLiteral = "AUTH "u8.ToArray();
```

UTF-8 string literals aren't compile time constants; they're runtime constants. Therefore, they can't be used as the default value for an optional parameter. UTF-8 string literals can't be combined with string interpolation. You can't use the `$` token and the `u8` suffix on the same string expression.

The delegate type

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

```
C#

public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

In .NET, `System.Action` and `System.Func` types provide generic definitions for many common delegates. You likely don't need to define new custom delegate types. Instead, you can create instantiations of the provided generic types.

A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see Delegates and Generic Delegates. Delegates are the basis for Events. A delegate can be instantiated by associating it either with a named or anonymous method.

The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters. For more information on the degree of variance that is allowed in the method signature, see Variance in Delegates. For use with anonymous methods, the delegate and the code to be associated with it are declared together.

Delegate combination or removal fails with a runtime exception when the delegate types involved at run time are different due to variant conversion. The following example demonstrates a situation that fails:

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but may fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

You can create a delegate with the correct runtime type by creating a new delegate object. The following example demonstrates how this workaround may be applied to the preceding example.

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of
Action<string>.
Action<string> wrappedObjectAction = new Action<string>
(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

Beginning with C# 9, you can declare *function pointers*, which use similar syntax. A function pointer uses the `calli` instruction instead of instantiating a delegate type and calling the virtual `Invoke` method.

The dynamic type

The `dynamic` type indicates that use of the variable and references to its members bypass compile-time type checking. Instead, these operations are resolved at run time. The `dynamic` type simplifies access to COM APIs such as the Office Automation APIs, to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type `dynamic` behaves like type `object` in most circumstances. In particular, any non-null expression can be converted to the `dynamic` type. The `dynamic` type differs from `object` in that operations that contain expressions of type `dynamic` aren't resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As

part of the process, variables of type `dynamic` are compiled into variables of type `object`. Therefore, type `dynamic` exists only at compile time, not at run time.

The following example contrasts a variable of type `dynamic` to a variable of type `object`. To verify the type of each variable at compile time, place the mouse pointer over `dyn` or `obj` in the `WriteLine` statements. Copy the following code into an editor where IntelliSense is available. IntelliSense shows **dynamic** for `dyn` and **object** for `obj`.

C#

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

The WriteLine statements display the run-time types of `dyn` and `obj`. At that point, both have the same type, integer. The following output is produced:

Console

```
System.Int32
System.Int32
```

To see the difference between `dyn` and `obj` at compile time, add the following two lines between the declarations and the `WriteLine` statements in the previous example.

C#

```
dyn = dyn + 3;
obj = obj + 3;
```

A compiler error is reported for the attempted addition of an integer and an object in expression `obj + 3`. However, no error is reported for `dyn + 3`. The expression that contains `dyn` isn't checked at compile time because the type of `dyn` is `dynamic`.

The following example uses `dynamic` in several declarations. The `Main` method also contrasts compile-time type checking with run-time type checking.

C#

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.ExampleMethod(10));
            Console.WriteLine(ec.ExampleMethod("value"));

            // The following line causes a compiler error because
ExampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.ExampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.ExampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to
ExampleMethod
            // with two arguments does not produce an error at com-
pile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.ExampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic _field;
        dynamic Prop { get; set; }

        public dynamic ExampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
```

```
}
// Results:
// Local variable
// 2
// Local variable
```

C# language specification

For more information, see the following sections of the C# language specification:

- §8.2.3 The object type
- §8.2.4 The dynamic type
- §8.2.5 The string type
- §8.2.8 Delegate types
- C# 11 - Raw string literals
- C# 11 - Raw string literals

See also

- C# Reference
- C# Keywords
- Events
- Using Type dynamic
- Best Practices for Using Strings
- Basic String Operations
- Creating New Strings
- Type-testing and cast operators
- How to safely cast using pattern matching and the as and is operators
- Walkthrough: creating and using dynamic objects
- System.Object
- System.String
- System.Dynamic.DynamicObject