Contents

Types (C# Programming Guide)

2/5/2019 • 11 minutes to read • Edit Online

Types, Variables, and Values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying Types in Variable Declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and Type Conversions.

Built-in Types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These are available for you to use in any C# program. For more information about the built-in types, see Reference Tables for Types.

Custom Types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see .NET Class Library.

The Common Type System

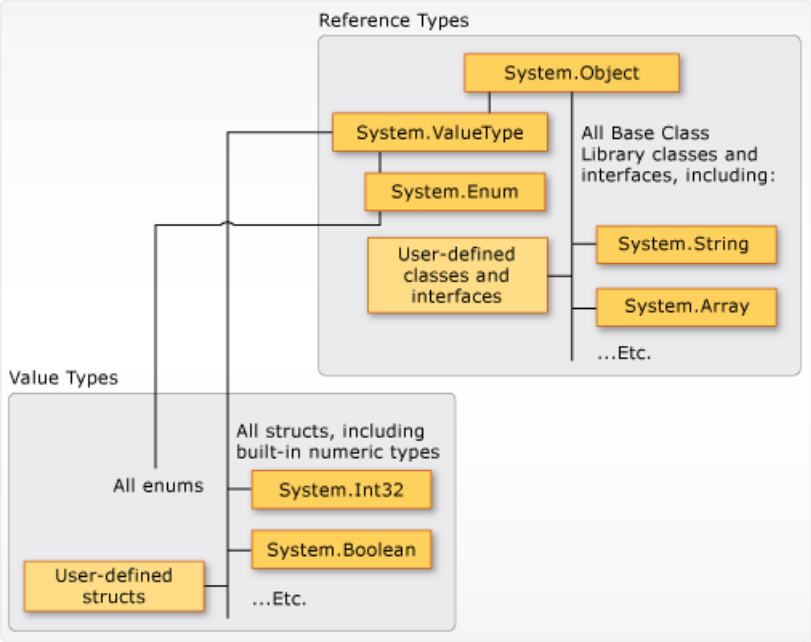
It is important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of

both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



Value types and reference types in the CTS

> **NOTE**
>
> You can see that the most commonly used types are all organized in the System namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value Types

Value types derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type byte.
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a System.Object or any interface type as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see Structs. For more information about value types in .NET, see Value Types.

The other category of value types is enum. An enum defines a set of named integral constants. For example, the System.IO.FileMode enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It is defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it is better to use enumerations instead of constant literal numbers. For more information, see System.IO.FileMode.

All enums inherit from System.Enum, which inherits from System.ValueType. All the rules that apply to structs also apply to enums. For more information about enums, see Enumeration Types.

Reference Types

A type that is defined as a class, delegate, array, or interface is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue. For more information about garbage collection, see Automatic Memory Management.

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the System.Array class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see Classes and Structs. For more information about inheritance and virtual methods, see Inheritance.

Types of Literal Values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from System.Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic Types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type System.Collections.Generic.List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see Generics.

Implicit Types, Anonymous Types, and Nullable Types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly Typed Local Variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous Types.

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a `?` after the type. For example, `int?` is an `int` type that can also have the value null. In the CTS, nullable types are instances of the generic struct type System.Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types.

Related Sections

For more information, see the following topics:

- Casting and Type Conversions
- Boxing and Unboxing
- Using Type dynamic
- Value Types
- Reference Types
- Classes and Structs
- Anonymous Types
- Generics

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Conversion of XML Data Types

- Integral Types Table

Casting and type conversions (C# Programming Guide)

2/3/2019 • 4 minutes to read • Edit Online

Because C# is statically-typed at compile time, after a variable is declared, it cannot be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` cannot be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you cannot assign the string "Hello" to it, as the following code shows:

```
int i;
i = "Hello"; // error CS0029: Cannot implicitly convert type 'string' to 'int'
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as `double`. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions**: No special syntax is required because the conversion is type safe and no data will be lost. Examples include conversions from smaller to larger integral types, and conversions from derived classes to base classes.
- **Explicit conversions (casts)**: Explicit conversions require a cast operator. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions**: User-defined conversions are performed by special methods that you can define to enable explicit and implicit conversions between custom types that do not have a base class–derived class relationship. For more information, see Conversion Operators.
- **Conversions with helper classes**: To convert between non-compatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays, you can use the System.BitConverter class, the System.Convert class, and the `Parse` methods of the built-in numeric types, such as Int32.Parse. For more information, see How to: Convert a byte Array to an int, How to: Convert a String to a Number, and How to: Convert Between Hexadecimal Strings and Numeric Types.

Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For example, a variable of type long (64-bit integer) can store any value that an int (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see Implicit Numeric Conversions Table.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

```
Derived d = new Derived();
Base b = d; // Always OK.
```

Explicit conversions

However, if a conversion cannot be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly informing the compiler that you intend to make the conversion and that you are aware that data loss might occur. To perform a cast, specify the type that you are casting to in parentheses in front of the value or variable to be converted. The following program casts a double to an int. The program will not compile without the cast.

```
class Test
{
    static void Main()
    {
        double x = 1234.7;
        int a;
        // Cast double to int.
        a = (int)x;
        System.Console.WriteLine(a);
    }
}
// Output: 1234
```

For a list of the explicit numeric conversions that are allowed, see Explicit Numeric Conversions Table.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

```
// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe) a;
```

A cast operation between reference types does not change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see Polymorphism.

Type conversion exceptions at run time

In some reference type conversions, the compiler cannot determine whether a cast will be valid. It is possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time will cause an InvalidCastException to be thrown.

```
using System;

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // Cause InvalidCastException at run time
        // because Mammal is not convertible to Reptile.
        Reptile r = (Reptile)a;
    }

}
```

C# provides the is and as operators to enable you to test for compatibility before actually performing a cast. For more information, see How to: Safely cast using pattern matching, as and is Operators.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Types
- () Operator
- explicit
- implicit
- Conversion Operators
- Generalized Type Conversion
- How to: Convert a String to a Number

Boxing and Unboxing (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

Boxing is the process of converting a value type to the type `object` or to any interface type implemented by this value type. When the CLR boxes a value type, it wraps the value inside a System.Object and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable `i` is *boxed* and assigned to object `o`.

```
int i = 123;
// The following line boxes i.
object o = i;
```

The object `o` can then be unboxed and assigned to integer variable `i`:

```
o = 123;
i = (int)o;  // unboxing
```

The following examples illustrate how boxing is used in C#.

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));


// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");

// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
```

```
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
// 8
// 9
// Sum: 30
```

Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see Performance.

Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type `object` or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

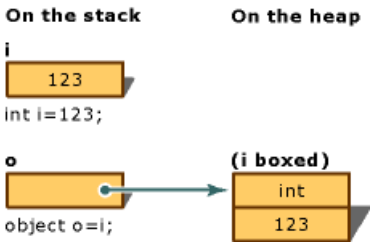
Consider the following declaration of a value-type variable:

```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

```
// Boxing copies the value of i into object o.
object o = i;
```

The result of this statement is creating an object reference `o`, on the stack, that references a value of the type `int`, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following figure.



Boxing Conversion

It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Description

This example converts an integer variable `i` to an object `o` by using boxing. Then, the value stored in the variable `i` is changed from `123` to `456`. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

Example

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
    The value-type value = 456
    The object-type value = 123
*/
```

Unboxing

Unboxing is an explicit conversion from the type `object` to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

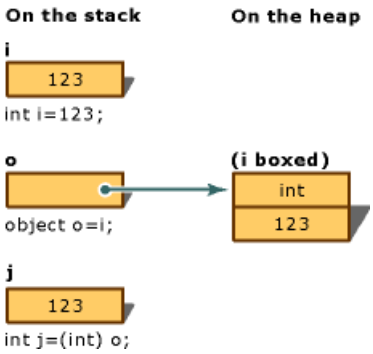
- Checking the object instance to make sure that it is a boxed value of the given value type.

- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

The following figure demonstrates the result of the previous statements.



Unboxing Conversion

For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox `null` causes a NullReferenceException. Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.

Example

The following example demonstrates a case of invalid unboxing and the resulting `InvalidCastException`. Using `try` and `catch`, an error message is displayed when the error occurs.

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            System.Console.WriteLine("Unboxing OK.");
        }
        catch (System.InvalidCastException e)
        {
            System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
        }
    }
}
```

This program outputs:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

If you change the statement:

```
int j = (short) o;
```

to:

```
int j = (int) o;
```

the conversion will be performed, and you will get the output:

```
Unboxing OK.
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Related Sections

For more information:

- Reference Types
- Value Types

See also

- C# Programming Guide

Using type dynamic (C# Programming Guide)

12/11/2018 • 5 minutes to read • Edit Online

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.

```
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();
}
```

```
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**.

```
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`:

```
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.

```
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`.

```
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.

```
// 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 a new API 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 /link (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.

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

```
// 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 Visual 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.

Walkthrough: Creating and Using Dynamic Objects (C# and Visual Basic)

2/12/2019 • 11 minutes to read • Edit Online

Dynamic objects expose members such as properties and methods at run time, instead of at compile time. This enables you to create objects to work with structures that do not match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the `GetProperty` method of the element. To reference the `id` attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the `id` attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script that is interpreted at run time.

You reference a dynamic object by using late binding. In C#, you specify the type of a late-bound object as `dynamic`. In Visual Basic, you specify the type of a late-bound object as `Object`. For more information, see dynamic and Early and Late Binding.

You can create custom dynamic objects by using the classes in the System.Dynamic namespace. For example, you can create an ExpandoObject and specify the members of that object at run time. You can also create your own type that inherits the DynamicObject class. You can then override the members of the DynamicObject class to provide run-time dynamic functionality.

In this walkthrough you will perform the following tasks:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an `IronPython` library.

Prerequisites

You need IronPython for .NET to complete this walkthrough. Go to their Download page to obtain the latest version.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Creating a Custom Dynamic Object

The first project that you create in this walkthrough defines a custom dynamic object that searches the contents of a text file. Text to search for is specified by the name of a dynamic property. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the

start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies `dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and does not remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

To create a custom dynamic class

1. Start Visual Studio.
2. On the **File** menu, point to **New** and then click **Project**.
3. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `DynamicSample`, and then click **OK**. The new project is created.
4. Right-click the DynamicSample project and point to **Add**, and then click **Class**. In the **Name** box, type `ReadOnlyFile`, and then click **OK**. A new file is added that contains the ReadOnlyFile class.
5. At the top of the ReadOnlyFile.cs or ReadOnlyFile.vb file, add the following code to import the System.IO and System.Dynamic namespaces.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

6. The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. Update the class statement to inherit the `DynamicObject` class, as shown in the following code example.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

8. Add the following code to the `ReadOnlyFile` class to define a private field for the file path and a constructor for the `ReadOnlyFile` class.

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

```
' Store the path to the file and the initial line count value.
Private p_filePath As String

' Public constructor. Verify that file exists and store the path in
' the private variable.
Public Sub New(ByVal filePath As String)
    If Not File.Exists(filePath) Then
        Throw New Exception("File path does not exist.")
    End If

    p_filePath = filePath
End Sub
```

9. Add the following `GetPropertyValue` method to the `ReadOnlyFile` class. The `GetPropertyValue` method takes, as input, search criteria and returns the lines from a text file that match that search criteria. The dynamic methods provided by the `ReadOnlyFile` class call the `GetPropertyValue` method to retrieve their respective results.

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return null.
        results = null;
    }
    finally
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

```
Public Function GetPropertyValue(ByVal propertyName As String,
                                 Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                                 Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCase(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCase(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCase(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function
```

10. After the `GetPropertyValue` method, add the following code to override the TryGetMember method of the DynamicObject class. The TryGetMember method is called when a member of a dynamic class is requested and no arguments are specified. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The TryGetMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

```
// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

```
' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function
```

11. After the `TryGetMember` method, add the following code to override the TryInvokeMember method of the DynamicObject class. The TryInvokeMember method is called when a member of a dynamic class is requested with arguments. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `args` argument contains an array of the arguments that are passed to the member. The TryInvokeMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

    The custom version of the `TryInvokeMember` method expects the first argument to be a value from the `StringSearchOption` enum that you defined in a previous step. The `TryInvokeMember` method expects the second argument to be a Boolean value. If one or both arguments are valid values, they are passed to the `GetPropertyValue` method to retrieve the results.

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```
' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByVal binder As InvokeMemberBinder,
                                          ByVal args() As Object,
                                          ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function
```

12. Save and close the file.

To create a sample text file

1. Right-click the DynamicSample project and point to **Add**, and then click **New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of TextFile1.txt in the **Name** box, and then click **Add**. A new text file is added to the project.

2. Copy the following text to the TextFile1.txt file.

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

3. Save and close the file.

To create a sample application that uses the custom dynamic object

1. In **Solution Explorer**, double-click the Module1.vb file if you are using Visual Basic or the Program.cs file if you are using Visual C#.

2. Add the following code to the Main procedure to create an instance of the `ReadOnlyFile` class for the TextFile1.txt file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

```
dynamic rFile = new ReadOnlyFile(@"..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

```
Dim rFile As Object = New ReadOnlyFile("..\..\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("----------------------------")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next
```

3. Save the file and press CTRL+F5 to build and run the application.

Calling a Dynamic Language Library

The next project that you create in this walkthrough accesses a library that is written in the dynamic language IronPython.

To create a custom dynamic class

1. In Visual Studio, on the **File** menu, point to **New** and then click **Project**.
2. In the **New Project** dialog box, in the **Project Types** pane, make sure that **Windows** is selected. Select **Console Application** in the **Templates** pane. In the **Name** box, type `DynamicIronPythonSample`, and then click **OK**. The new project is created.
3. If you are using Visual Basic, right-click the DynamicIronPythonSample project and then click **Properties**. Click the **References** tab. Click the **Add** button. If you are using Visual C#, in **Solution Explorer**, right-click the **References** folder and then click **Add Reference**.
4. On the **Browse** tab, browse to the folder where the IronPython libraries are installed. For example, C:\Program Files\IronPython 2.6 for .NET 4.0. Select the **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Scripting.dll**, and **Microsoft.Dynamic.dll** libraries. Click **OK**.
5. If you are using Visual Basic, edit the Module1.vb file. If you are using Visual C#, edit the Program.cs file.
6. At the top of the file, add the following code to import the `Microsoft.Scripting.Hosting` and `IronPython.Hosting` namespaces from the IronPython libraries.

```
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

```
Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting
```

7. In the Main method, add the following code to create a new `Microsoft.Scripting.Hosting.ScriptRuntime` object to host the IronPython libraries. The `ScriptRuntime` object loads the IronPython library module random.py.

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.6 for .NET 4.0\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

```
' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
   My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
   "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")
```

8. After the code to load the random.py module, add the following code to create an array of integers. The array is passed to the `shuffle` method of the random.py module, which randomly sorts the values in the array.

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("--------------------");
}
```

```
' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("--------------------")
Next
```

9. Save the file and press CTRL+F5 to build and run the application.

See also

- System.Dynamic
- System.Dynamic.DynamicObject

- Using Type dynamic
- Early and Late Binding
- dynamic
- Implementing Dynamic Interfaces (downloadable PDF from Microsoft TechNet)

How to: Convert a byte Array to an int (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example shows you how to use the BitConverter class to convert an array of bytes to an int and back to an array of bytes. You may have to convert from bytes to a built-in data type after you read bytes off the network, for example. In addition to the ToInt32(Byte[], Int32) method in the example, the following table lists methods in the BitConverter class that convert bytes (from an array of bytes) to other built-in types.

TYPE RETURNED	METHOD
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Example

This example initializes an array of bytes, reverses the array if the computer architecture is little-endian (that is, the least significant byte is stored first), and then calls the ToInt32(Byte[], Int32) method to convert four bytes in the array to an `int`. The second argument to ToInt32(Byte[], Int32) specifies the start index of the array of bytes.

> **NOTE**
>
> The output may differ depending on the endianess of your computer's architecture.

```
byte[] bytes = { 0, 0, 0, 25 };

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

Example

In this example, the GetBytes(Int32) method of the BitConverter class is called to convert an `int` to an array of bytes.

> **NOTE**
> The output may differ depending on the endianess of your computer's architecture.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

See also

- BitConverter
- IsLittleEndian
- Types

How to: Convert a String to a Number (C# Programming Guide)

2/13/2019 • 4 minutes to read • Edit Online

You can convert a string to a number by calling the `Parse` or `TryParse` method found on the various numeric types (`int` , `long` , `double` , etc.), or by using methods in the System.Convert class.

If you have a string, it is slightly more efficient and straightforward to call a `TryParse` method (for example, `int.TryParse("11", out number)`) or `Parse` method (for example, `var number = int.Parse("11")`). Using a Convert method is more useful for general objects that implement IConvertible.

You can use `Parse` or `TryParse` methods on the numeric type you expect the string contains, such as the System.Int32 type. The Convert.ToInt32 method uses Parse internally. The `Parse` method returns the converted number; the `TryParse` method returns a Boolean value that indicates whether the conversion succeeded, and returns the converted number in an `out` parameter. If the string is not in a valid format, `Parse` throws an exception, whereas `TryParse` returns false. When calling a `Parse` method, you should always use exception handling to catch a FormatException in the event that the parse operation fails.

Calling the Parse and TryParse methods

The `Parse` and `TryParse` methods ignore white space at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (`int` , `long` , `ulong` , `float` , `decimal` , etc.). Any white space within the string that forms the number causes an error. For example, you can use `decimal.TryParse` to parse "10", "10.3", or " 10 ", but you cannot use this method to parse 10 from "10X", "1 0" (note the embedded space), "10 .3" (note the embedded space), "10e1" (`float.TryParse` works here), and so on. In addition, a string whose value is `null` or String.Empty fails to parse successfully. You can check for a null or empty string before attempting to parse it by calling the String.IsNullOrEmpty method.

The following example demonstrates both successful and unsuccessful calls to `Parse` and `TryParse` .

```
using System;

public class StringConversion
{
    public static void Main()
    {
       string input = String.Empty;
       try
       {
           int result = Int32.Parse(input);
           Console.WriteLine(result);
       }
       catch (FormatException)
       {
           Console.WriteLine($"Unable to parse '{input}'");
       }
       // Output: Unable to parse ''

       try
       {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
       }
       catch (FormatException e)
       {
           Console.WriteLine(e.Message);
       }
       // Output: -105

        if (Int32.TryParse("-105", out int j))
            Console.WriteLine(j);
        else
            Console.WriteLine("String could not be parsed.");
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
            Console.WriteLine(inputString);
        else
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an int.");
        // Output: Int32.TryParse could not parse 'abc' to an int.
     }
}
```

The following example illustrates one a approach to parsing a string that is expected to include leading numeric characters (including hexadecimal characters) and trailing non-numeric characters. It assigns valid characters from the beginning of a string to a new string before calling the TryParse method. Because the strings to be parsed contain a small number of characters, the example calls the String.Concat method to assign valid characters to a new string. For a larger string, the StringBuilder class can be used instead.

```
using System;

public class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = String.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || (Char.ToUpperInvariant(c) >= 'A' && Char.ToUpperInvariant(c) <= 'F')
|| c == ' ') {
                numericString = String.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }
        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber, null, out int i))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
            // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str) {
            // Check for numeric characters (0-9), a negative sign, or leading or trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
            {
                numericString = String.Concat(numericString, c);
            } else
                break;
        }
        if (int.TryParse(numericString, out int j))
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
            // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Calling the Convert methods

The following table lists some of the methods from the Convert class that you can use to convert a string to a number.

NUMERIC TYPE	METHOD
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)

NUMERIC TYPE	METHOD
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

The following example calls the Convert.ToInt32(String) method to convert an input string to an int. The example catches the two most common exceptions that can be thrown by this method, FormatException and OverflowException. If the resulting number can be incremented without exceeding Int32.MaxValue, the example adds 1 to the result and displays the output.

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between –2,147,483,648 and +2,147,483,647 (inclusive): ");

            string input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its current value");
                }
           }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string go = Console.ReadLine();
            if (go.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

See also

- Types
- How to: Determine Whether a String Represents a Numeric Value
- .NET Framework 4 Formatting Utility

How to: Convert Between Hexadecimal Strings and Numeric Types (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

These examples show you how to perform the following tasks:

- Obtain the hexadecimal value of each character in a string.
- Obtain the char that corresponds to each value in a hexadecimal string.
- Convert a hexadecimal `string` to an int.
- Convert a hexadecimal `string` to a float.
- Convert a byte array to a hexadecimal `string`.

Example

This example outputs the hexadecimal value of each character in a `string`. First it parses the `string` to an array of characters. Then it calls ToInt32(Char) on each character to obtain its numeric value. Finally, it formats the number as its hexadecimal representation in a `string`.

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

Example

This example parses a `string` of hexadecimal values and outputs the character corresponding to each hexadecimal value. First it calls the Split(Char[]) method to obtain each hexadecimal value as an individual `string` in an array. Then it calls ToInt32(String, Int32) to convert the hexadecimal value to a decimal value represented as an int. It shows two different ways to obtain the character corresponding to that character code. The first technique uses ConvertFromUtf32(Int32), which returns the character corresponding to the integer argument as a `string`. The second technique explicitly casts the `int` to a char.

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value = {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

Example

This example shows another way to convert a hexadecimal `string` to an integer, by calling the Parse(String, NumberStyles) method.

```
string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

Example

The following example shows how to convert a hexadecimal `string` to a float by using the System.BitConverter class and the UInt32.Parse method.

```
string hexString = "43480170";
uint num = uint.Parse(hexString, System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

Example

The following example shows how to convert a byte array to a hexadecimal string by using the System.BitConverter class.

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

See also

- Standard Numeric Format Strings
- Types
- How to: Determine Whether a String Represents a Numeric Value