Hidden Features of C#? [closed]

Asked 16 years, 4 months ago Modified 5 years, 9 months ago Viewed 758k times

1472

votes

As it currently stands, this question is not a good fit for our Q&A format. We expect answers to be supported by facts, references, or expertise, but this question will likely solicit debate, arguments, polling, or extended discussion. If you feel that this question can be improved and possibly reopened, visit the help center for guidance.

Closed 12 years ago.

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

This came to my mind after I learned the following from this question:

```
where T : struct
```

We, C# developers, all know the basics of C#. I mean declarations, conditionals, loops, operators, etc.

Some of us even mastered the stuff like Generics, anonymous types, lambdas, LINQ, ...

But what are the most hidden features or tricks of C# that even C# fans, addicts, experts barely know?

Here are the revealed features so far:

Keywords

- `yield` by Michael Stum
- `var` by Michael Stum
- `using()` statement by kokos
- `readonly` by kokos
- `as` by Mike Stone

- `as` / `is` by Ed Swangren
- `as` / `is` (improved) by Rocketpants
- `default` by deathofrats
- `global::` by pzycoman
- `using()` blocks by AlexCuse
- `volatile` by Jakub Šturc
- `extern alias` by Jakub Šturc

Attributes

- `DefaultValueAttribute` by Michael Stum
- `ObsoleteAttribute` by DannySmurf
- `DebuggerDisplayAttribute` by Stu
- `DebuggerBrowsable` and `DebuggerStepThrough` by bdukes
- `ThreadStaticAttribute` by marxidad
- `FlagsAttribute` by Martin Clarke
- `ConditionalAttribute` by AndrewBurns

Syntax

- `??` (coalesce nulls) operator by kokos
- Number flaggings by Nick Berardi
- `where T:new` by Lars Mæhlum
- Implicit generics by Keith
- One-parameter lambdas by Keith
- Auto properties by Keith
- Namespace aliases by Keith
- Verbatim string literals with @ by Patrick
- `enum` values by lfoust
- @variablenames by marxidad
- `event` operators by marxidad
- Format string brackets by Portman
- Property accessor accessibility modifiers by xanadont

- Conditional (ternary) operator (`?:`) by JasonS
- `checked` and `unchecked` operators by Binoj Antony
- `implicit and explicit` operators by Flory

Language Features

- Nullable types by Brad Barker
- Anonymous types by Keith
- `__makeref __reftype __refvalue` by Judah Himango
- Object initializers by lomaxx
- Format strings by David in Dakota
- Extension Methods by marxidad
- `partial` methods by Jon Erickson
- Preprocessor directives by John Asbeck
- `DEBUG` pre-processor directive by Robert Durgin
- Operator overloading by SefBkn
- Type inferrence by chakrit
- Boolean operators taken to next level by Rob Gough
- Pass value-type variable as interface without boxing by Roman Boiko
- Programmatically determine declared variable type by Roman Boiko
- Static Constructors by Chris
- Easier-on-the-eyes / condensed ORM-mapping using LINQ by roosteronacid
- `__arglist` by Zac Bowling

Visual Studio Features

- Select block of text in editor by Himadri
- Snippets by DannySmurf

Framework

- `TransactionScope` by KiwiBastard
- `DependantTransaction` by KiwiBastard
- `Nullable<T>` by IainMH

- `Mutex` by Diago
- `System.IO.Path` by ageektrapped
- `WeakReference` by Juan Manuel

Methods and Properties

- `String.IsNullOrEmpty()` method by KiwiBastard
- `List.ForEach()` method by KiwiBastard
- `BeginInvoke()`, `EndInvoke()` methods by Will Dean
- `Nullable<T>.HasValue` and `Nullable<T>.Value` properties by Rismo
- `GetValueOrDefault` method by John Sheehan

Tips & Tricks

- Nice method for event handlers by Andreas H.R. Nilsson
- Uppercase comparisons by John
- Access anonymous types without reflection by dp
- A quick way to lazily instantiate collection properties by Will
- JavaScript-like anonymous inline-functions by roosteronacid

Other

- netmodules by kokos
- LINQBridge by Duncan Smart
- Parallel Extensions by Joel Coehoorn

c# hidden-features

Share

edited Sep 25, 2017 at 20:53

community wiki
56 revs, 31 users 10%
Serhat Ozgel

Comments disabled on deleted / locked posts / reviews

296 Answers

Sorted by: Highest score (default)

1 2 3 4 5 … 10 Next

751
votes

This isn't C# per se, but I haven't seen anyone who really uses `System.IO.Path.Combine()` to the extent that they should. In fact, the whole Path class is really useful, but **no one uses it!**

I'm willing to bet that every production app has the following code, even though it shouldn't:

```
string path = dir + "\\" + fileName;
```

Share

answered Aug 13, 2008 at 1:53

community wiki
ageektrapped

583
votes

lambdas and type inference are underrated. **Lambdas can have multiple statements** and they **double as a compatible delegate object** automatically (just make sure the signature match) as in:

```
Console.CancelKeyPress +=
    (sender, e) => {
        Console.WriteLine("CTRL+C detected!\n");
        e.Cancel = true;
    };
```

Note that I don't have a `new CancellationEventHandler` nor do I have to specify types of `sender` and `e`, they're inferable from the event. Which is why this is less cumbersome to writing the whole `delegate (blah blah)` which also requires you to specify types of parameters.

Lambdas don't need to return anything and type inference is extremely powerful in context like this.

And BTW, you can always return **Lambdas that make Lambdas** in the functional programming sense. For example, here's a lambda that makes a lambda that handles a Button.Click event:

```
Func<int, int, EventHandler> makeHandler =
    (dx, dy) => (sender, e) => {
        var btn = (Button) sender;
        btn.Top += dy;
        btn.Left += dx;
    };

btnUp.Click += makeHandler(0, -1);
btnDown.Click += makeHandler(0, 1);
btnLeft.Click += makeHandler(-1, 0);
btnRight.Click += makeHandler(1, 0);
```

Note the chaining: `(dx, dy) => (sender, e) =>`

Now that's why I'm happy to have taken the functional programming class :-)

Other than the pointers in C, I think it's the other fundamental thing you should learn :-)

Share edited Mar 4, 2019 at 18:39 community wiki
12 revs, 7 users 80%
chakrit

527
votes

From Rick Strahl:

You can chain the ?? operator so that you can do a bunch of null comparisons.

```
string result = value1 ?? value2 ?? value3 ?? String.Empty;
```

Share edited Aug 27, 2008 at 1:34 community wiki
2 revs
jfs

453
votes

Aliased generics:

```
using ASimpleName = Dictionary<string, Dictionary<string, List<string>>>;
```

It allows you to use `ASimpleName`, instead of `Dictionary<string, Dictionary<string, List<string>>>`.

Use it when you would use the same generic big long complex thing in a lot of places.

Share edited Sep 12, 2009 at 20:32 community wiki
2 revs, 2 users 75%
BlackTigerX

437
votes

> From CLR via C#:
>
> When normalizing strings, it is highly recommended that you use ToUpperInvariant instead of ToLowerInvariant because **Microsoft has optimized the code for performing uppercase comparisons**.

I remember one time my coworker always changed strings to uppercase before comparing. I've always wondered why he does that because I feel it's more "natural" to convert to lowercase first. After reading the book now I know why.

Share answered Aug 15, 2008 at 11:06 community wiki jfs

254 When you "convert a string to upper case" you create a second temporary string object. I thought that this kind of comparison was not preferred, that the best way was: String.Equals(stringA, stringB, StringComparison.CurrentCultureIgnoreCase) whcih does not create this throwaway string at all. – Anthony Sep 23, 2008 at 14:44

32 What kind of optimization can you perform on comparing upper case strings that can't be done on lower case strings? I don't understand why one would be more optimal than the other. – Parappa Oct 24, 2008 at 17:38

36 Converting to uppercase rather than lowercase can also prevent incorrect behavior in certain cultures. For example, in Turkish, two lowercase i's map to the same uppercase I. Google "turkish i" for more details. – Neil Dec 17, 2008 at 17:17

34 I tried benchmarking ToUpperInvariant vs ToLowerInvariant. I cannot find any difference in their performance under .NET 2.0 or 3.5. Certainly not anything that warrant "highly recommending" using one over the other. – Rasmus Faber Jan 21, 2009 at 21:41

19 ToUpperInvariant is preferred because it makes all characters round-trip. See msdn.microsoft.com/en-us/library/bb386042.aspx. For comparisons, write `"a".Equals("A", StringComparison.OrdinalIgnoreCase)` – SLaks Jun 4, 2009 at 19:35

407 votes

My favorite trick is using the null coalesce operator and parentheses to automagically instantiate collections for me.

```
private IList<Foo> _foo;

public IList<Foo> ListOfFoo
    { get { return _foo ?? (_foo = new List<Foo>()); } }
```

Share edited Mar 21, 2010 at 17:14 community wiki 6 revs, 3 users 56% Will

23 Don't you find it hard to read? – Riri May 19, 2009 at 18:40

72 Its slightly hard to read for the noo... er, inexperienced. But its compact and contains a couple patterns and language features that the programmer should know and understand. So, while it is hard at first, it provides the benefit of being a reason to learn. – user1228 May 20, 2009 at 13:23

38 Lazy instantiation is somewhat malpractice because it's a poor mans choice to avoid thinking about class invariants. It also has concurrency issues. – John Leidegren Aug 5, 2009 at 17:23

17 I've always been informed this is bad practise, that calling a property should not do something as such. If you had a set on there and set the value to null, it would be very strange for someone using your API. – Ian Oct 7, 2009 at 8:16

8 @Joel except that setting ListOfFoo to null is neither part of the class contract nor is it good practice. That's why there isn't a setter. Its also why ListOfFoo is guaranteed to return a collection and never a null. Its just a very odd complaint that if you do two bad things (make a setter and set a collection to null) this will result in your expectations being wrong. I wouldn't suggest doing Environment.Exit() in the getter, either. But that also has nothing to do with this answer. – user1228 Jun 21, 2010 at 17:01

315 votes

Avoid checking for null event handlers

Adding an empty delegate to events at declaration, suppressing the need to always check the event for null before calling it is awesome. Example:

```
public delegate void MyClickHandler(object sender, string myValue);
public event MyClickHandler Click = delegate {}; // add empty delegate!
```

Let you do this

```
public void DoSomething()
{
    Click(this, "foo");
}
```

Instead of this

```
public void DoSomething()
{
    // Unnecessary!
    MyClickHandler click = Click;
    if (click != null) // Unnecessary!
    {
        click(this, "foo");
    }
}
```

Please also see this related discussion and this blog post by Eric Lippert on this topic (and possible downsides).

Share

edited May 23, 2017 at 11:55

community wiki
6 revs, 4 users 79%
andnil

87 I believe a problem will appear if you rely on this technique and then you have to serialize the class. You will eliminate the event, and then on deserialization you will get a NullRefference.... .So one can just stick to the "old way" of doing things. It's safer. – sirrocco Oct 13, 2008 at 9:58

16 you can still set your event handler to null, so you can still get a null reference, and you still have a race condition. – Robert Paulson Oct 19, 2008 at 23:00

64 A quick profile test shows that dummy-subscribed event handler without null test takes roughly 2x the time of unsubscribed event handler with null test. Multicast event handler without null test takes about 3.5x the time of singlecast event handler with null test. – P Daddy Nov 17, 2008 at 6:53

54 This avoids the need for a null check by just always having a self-subscriber. Even as an empty event this carries an overhead that you don't want. If there are no subscribers you don't want to fire the event at all, not always fire an empty dummy event first. I would consider this bad code. – Keith Dec 4, 2008 at 14:49

56 This is a terrible suggestion, for the reasons in the above comments. If you must make your code look "clean", use an extension method to check for null then call the event. Someone with modify privileges should definitely add the cons to this answer. – Greg Mar 31, 2009 at 4:29

304 votes

Everything else, plus

1) implicit generics (why only on methods and not on classes?)

```
void GenericMethod<T>( T input ) { ... }

//Infer type, so
GenericMethod<int>(23); //You don't need the <>.
GenericMethod(23);      //Is enough.
```

2) simple lambdas with one parameter:

```
x => x.ToString() //simplify so many calls
```

3) anonymous types and initialisers:

```
//Duck-typed: works with any .Add method.
var colours = new Dictionary<string, string> {
    { "red", "#ff0000" },
    { "green", "#00ff00" },
    { "blue", "#0000ff" }
};

int[] arrayOfInt = { 1, 2, 3, 4, 5 };
```

Another one:

4) Auto properties can have different scopes:

```
public int MyId { get; private set; }
```

Thanks @pzycoman for reminding me:

5) Namespace aliases (not that you're likely to need this particular distinction):

```
using web = System.Web.UI.WebControls;
using win = System.Windows.Forms;

web::Control aWebControl = new web::Control();
win::Control aFormControl = new win::Control();
```

Share   edited Feb 13, 2012 at 21:56   community wiki
8 revs, 4 users 88%
Keith

14 in #3 you can do Enumerable.Range(1,5) – Echostorm Oct 7, 2008 at 14:08

18 i think you've been able to initialize arrays with int[] nums = {1,2,3}; since 1.0 :) doesn't even need the "new" keyword – Lucas Oct 7, 2008 at 23:26

13 also lambda without parameters ()=> DoSomething(); – Pablo Retyk Jan 12, 2009 at 8:38

5 I've used both { get; internal set; } and { get; protected set; }, so this pattern is consistent. – Keith May 4, 2009 at 21:23

7 @Kirk Broadhurst - you're right - `new web.Control()` would also work in this example. The `::` syntax forces it to treat the prefix as a namespace alias, so you could have a class called `web` and the `web::Control` syntax would still work, while the `web.Control` syntax wouldn't know whether to check the class or the namespace. Because of that I tend to always use `::` when doing namespace aliases. – Keith Mar 8, 2010 at 11:32

285
votes

I didn't know the "as" keyword for quite a while.

```
MyClass myObject = (MyClass) obj;
```

vs

```
MyClass myObject = obj as MyClass;
```

The second will return null if obj isn't a MyClass, rather than throw a class cast exception.

Share answered Aug 12, 2008 at 16:42 community wiki Mike Stone

42 Don't over-do it though. Lots of people seem to use as because the prefer the syntax even though they want the semantics of a (ToType)x. – Scott Langham Sep 19, 2008 at 18:07

4 I don't believe it offers better performance. Have you profiled it? (Obviously though it does when the cast fails... but when you use (MyClass) cast, failures are exceptional.. and extremely rare (if they happen at all), so it makes no difference. – Scott Langham Jan 21, 2009 at 13:01

7 This is only more performant if the usual case is the cast failing. Otherwise the direct cast (type)object is faster. It takes longer for a direct cast to throw an exception than to return null though. – Spence Jan 26, 2009 at 14:54

15 Right along the same lines of the "as" keyword... the "is" keyword is just as useful. – dkpatt May 7, 2009 at 19:09

28 You can abuse it and have a NullReferenceException down your road later when you could have had a InvalidCastException earlier. – Andrei Rînea Sep 2, 2009 at 7:55

261 votes

Two things I like are Automatic properties so you can collapse your code down even further:

```
private string _name;
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}
```

becomes

```
public string Name { get; set;}
```

Also object initializers:

```
Employee emp = new Employee();
emp.Name = "John Smith";
emp.StartDate = DateTime.Now();
```

becomes

```
Employee emp = new Employee {Name="John Smith", StartDate=DateTime.Now()}
```

Share

answered Aug 13, 2008 at 7:39

community wiki
lomaxx

6 Should it be noted that Automatic Properties are a C# 3.0 only feature? – Jared Updike Sep 18, 2008 at 22:53

21 Automatic Properties were introduced with the 3.0 compiler. But since the compiler can be set to output 2.0 code, they work just fine. Just don't try to compile 2.0 code with automatic properties in an older compiler! – Joshua Shannon Mar 10, 2009 at 14:59

74 Something many people don't realise is that get and set can have different accessibility, eg: public string Name { get; private set;} – Nader Shirazie Jun 13, 2009 at 20:27

7 Only problem with Automatic Properties is that it doesn't seem to be possible to provide a default initialization value. – Stein Åsmul Sep 12, 2009 at 20:57

7 @ANeves : no it's not. From that page: **A DefaultValueAttribute will not cause a member to be automatically initialized with the attribute's value. You must set the initial value in your code.** `[DefaultValue]` is used for the designer so it knows whether to show a property in bold (meaning non-default). – Roger Lipscombe Jun 14, 2010 at 8:24

254 votes

The 'default' keyword in generic types:

```
T t = default(T);
```

results in a 'null' if T is a reference type, and 0 if it is an int, false if it is a boolean, etcetera.

Share

answered Aug 13, 2008 at 10:20

community wiki
Eric Minkes

4 Plus: type? as a shortcut for Nullable<type>. default(int) == 0, but default(int?) == null. – sunside Jul 24, 2010 at 17:22

225 votes

Attributes in general, but most of all DebuggerDisplay. Saves you years.

Share

edited Apr 26, 2009 at 16:34

community wiki
2 revs, 2 users 67%
boj

9 Using DebuggerDisplay Attribute (MSDN): msdn.microsoft.com/en-us/library/x810d419.aspx – Jon Adams Sep 26, 2008 at 17:14

220 votes

> The @ tells the compiler to ignore any escape characters in a string.

Just wanted to clarify this one... it doesn't tell it to ignore the escape characters, it actually tells the compiler to interpret the string as a literal.

If you have

```
string s = @"cat
             dog
             fish"
```

it will actually print out as (note that it even includes the whitespace used for indentation):

```
cat
             dog
             fish
```

Share edited Sep 12, 2009 at 20:33 community wiki 4 revs, 4 users 77% lomaxx

wouldn't the string include all the spaces that you used for indentation? – andy Oct 13, 2008 at 22:39

18 Yes it's called verbatim string. – Joan Venge Mar 3, 2009 at 20:57

4 It would be clearer if the output showed the spaces that would be printed out as well. Right now it seems as if the new lines characters are printed but spaces are ignored. – aleemb Apr 28, 2009 at 11:15

Very useful for escaping regular expressions and long SQL queries. – ashes999 Dec 21, 2010 at 16:07

It also maps `{{` to `{` and `}}` to `}` making it useful for `string.Format()`. – Ferruccio Jan 8, 2012 at 16:12

219 votes

I think one of the most under-appreciated and lesser-known features of C# (.NET 3.5) are Expression Trees, **especially** when combined with Generics and Lambdas. This is an approach to API creation that newer libraries like NInject and Moq are using.

For example, let's say that I want to register a method with an API and that API needs to get the method name

Given this class:

```
public class MyClass
{
     public void SomeMethod() { /* Do Something */ }
}
```

Before, it was very common to see developers do this with strings and types (or something else largely string-based):

```
RegisterMethod(typeof(MyClass), "SomeMethod");
```

Well, that sucks because of the lack of strong-typing. What if I rename "SomeMethod"? Now, in 3.5 however, I can do this in a strongly-typed fashion:

```
RegisterMethod<MyClass>(cl => cl.SomeMethod());
```

In which the RegisterMethod class uses `Expression<Action<T>>` like this:

```
void RegisterMethod<T>(Expression<Action<T>> action) where T : class
{
    var expression = (action.Body as MethodCallExpression);

    if (expression != null)
    {
        // TODO: Register method
        Console.WriteLine(expression.Method.Name);
    }
}
```

This is one big reason that I'm in love with Lambdas and Expression Trees right now.

Share   edited Jun 6, 2010 at 20:22   community wiki
4 revs, 4 users 91%
Jason Olson

3 I have a reflection utility class that does the same with FieldInfo, PropertyInfo, etc... – Olmo Nov 24, 2008 at 22:09

Yes, this is great. I was able to use methods like this to write code like `EditValue(someEmployee, e => e.FirstName);` in my business logic and have it automatically generate all the plumbing logic for a ViewModel and View to edit that property (so, a label with the text "First Name" and a TextBox with a binding that calls the setter of the FirstName property when the user edits the name, and updates the View using the getter). This seems to be the basis for most of the new internal DSLs in C#. – Scott Whitlock Dec 1, 2010 at 4:11

4 I think these aren't so much lesser-known as lesser-understood. – Justin Morgan Mar 17, 2011 at 17:20

Why do you need to register a method? I have never used this before - where and when would this be used? – MoonKnight Jan 26, 2012 at 9:46

208 votes

"yield" would come to my mind. Some of the attributes like [DefaultValue()] are also among my favorites.

The "var" keyword is a bit more known, but that you can use it in .NET 2.0 applications as well (as long as you use the .NET 3.5 compiler and set it to output 2.0 code) does not seem to be known very well.

Edit: kokos, thanks for pointing out the ?? operator, that's indeed really useful. Since it's a bit hard to google for it (as ?? is just ignored), here is the MSDN documentation page for that operator: *?? Operator (C# Reference)*

Share edited May 23, 2011 at 11:18 community wiki 4 revs, 2 users 80% Michael Stum

14 The default value's documentation says it is not really setting the value of the property. It's only a helper for visualisers and code generators. – Boris Callens Nov 8, 2008 at 1:18

2 As for DefaultValue: In the meantime, some libraries use it. ASP.net MVC uses DefaultValue on Parameters of a Controller Action (Which is very useful for non-nullable types). Strictly speaking of course, this is a code generator as the value is not set by the compiler but by MVC's code. – Michael Stum Jan 10, 2010 at 1:32

6 The name for the ?? operator is the "Null Coalescing" operator – Armstrongest Mar 29, 2010 at 18:23

yield is my favorite, the coalesce operator is up there though. I don't see anything about CAS, or assembly signing, strong names, the GAC... I suppose only CAS is C#... but so many developers have no clue about security. – BrainSlugs83 Feb 28, 2012 at 21:05

197 votes

I tend to find that most C# developers don't know about 'nullable' types. Basically, primitives that can have a null value.

```
double? num1 = null;
double num2 = num1 ?? -100;
```

Set a nullable double, *num1*, to null, then set a regular double, *num2*, to *num1* or *-100* if *num1* was null.

http://msdn.microsoft.com/en-us/library/1t3y8s4s(VS.80).aspx

one more thing about Nullable type:

```
DateTime? tmp = new DateTime();
tmp = null;
return tmp.ToString();
```

it is return String.Empty. Check this link for more details

Share edited May 23, 2017 at 10:31

community wiki
4 revs, 3 users 62%
Brad Barker

1 DateTime cannot be set to null. – Jason Jackson Sep 30, 2008 at 1:03

2 So then is "int" just C# syntactic sugar for System.Int32 ? There actually is compiler support built around the Nullable types, enabling setting them to null, for instance (which can't be done using generic structs alone), and boxing them as their underlying type. – P Daddy Nov 17, 2008 at 7:00

4 @P Daddy - yes, int is syntactic sugar for System.Int32. They are completely interchangeable, just like int? === Int32? === Nullable<Int32> === Nullable<int> – cjk Jan 8, 2009 at 18:22

6 @ck: Yes, int is an alias for System.Int32, as T? is an alias for Nullable<T>, but it isn't *just* syntactic sugar. It's a defined part of the language. The aliases not only make the code easier to read, but also fit our model better. Expressing Nullable<Double> as double? underscores the perspective that the value thus contained is (or may be) a double, not just some generic structure instantiated on type Double. (continued) – P Daddy Jun 15, 2009 at 4:09

3 ... In any case, the argument wasn't about aliases (that was just a poor analogy, I suppose), but that nullable types—using whatever syntax you prefer—are indeed a language feature, not just a product of generics. You can't reproduce all the functionality of nullables (comparison to/assignment of null, operator forwarding, boxing of underlying type or null, compatibility with null coalescing and `as` operators) with generics alone. Nullable<T> alone is rudimentary and far from stellar, but the concept of nullable types as part of the language is kick ass. – P Daddy Jun 15, 2009 at 4:15

192 votes

Here are some interesting hidden C# features, in the form of undocumented C# keywords:

```
__makeref

__reftype

__refvalue

__arglist
```

These are undocumented C# keywords (even Visual Studio recognizes them!) that were added to for a more efficient boxing/unboxing prior to generics. They work in coordination with the System.TypedReference struct.

There's also __arglist, which is used for variable length parameter lists.

One thing folks don't know much about is System.WeakReference -- a very useful class that keeps track of an object but still allows the garbage collector to collect it.

The most useful "hidden" feature would be the yield return keyword. It's not really hidden, but a lot of folks don't know about it. LINQ is built atop this; it allows for delay-executed queries by generating a state machine under the hood. Raymond Chen recently posted about the internal, gritty details.

Share   edited Nov 10, 2011 at 17:14   community wiki
4 revs, 3 users 78%
Judah Himango

2 More details on the undocumented keywords by Peter Bromberg. I still don't get if there are reasons to ever use them. – HuBeZa Jan 31, 2011 at 11:53

@HuBeZa with the advent of generics, there aren't many (any?) good reasons to used __refType, __makeref, and __refvalue. These were used primarily to avoid boxing prior to generics in .NET 2. – Judah Gabriel Himango Jan 31, 2011 at 19:14

Matt, with the introduction of generics in .NET 2, there's little reason to use these keywords, as their purpose was to avoid boxing when dealing with value types. See the link by HuBeZa, and also see codeproject.com/Articles/38695/UnCommon-C-keywords-A-Look#ud
– Judah Gabriel Himango Nov 10, 2011 at 17:13

183
votes

Unions (the C++ shared memory kind) in pure, safe C#

Without resorting to unsafe mode and pointers, you can have class members share memory space in a class/struct. Given the following class:

```
[StructLayout(LayoutKind.Explicit)]
public class A
{
    [FieldOffset(0)]
    public byte One;

    [FieldOffset(1)]
    public byte Two;

    [FieldOffset(2)]
    public byte Three;

    [FieldOffset(3)]
    public byte Four;
    [FieldOffset(0)]
    public int Int32;
}
```

You can modify the values of the byte fields by manipulating the Int32 field and vice-versa. For example, this program:

```
    static void Main(string[] args)
    {
        A a = new A { Int32 = int.MaxValue };

        Console.WriteLine(a.Int32);
        Console.WriteLine("{0:X} {1:X} {2:X} {3:X}", a.One, a.Two, a.Three,
a.Four);

        a.Four = 0;
        a.Three = 0;
        Console.WriteLine(a.Int32);
    }
```

Outputs this:

```
2147483647
FF FF FF 7F
65535
```

just add using System.Runtime.InteropServices;

Share answered Sep 23, 2008 at 20:39 community wiki ZeroBugBounce

7 @George, works wonders when you are communicating with legacy apps over sockets using c union declarations. – scottm Jul 23, 2009 at 19:07

2 There's also meaningful to say int and float at offset 0. It's what you need if you want to manipulate floating point numbers as bit masks, which you sometimes want to. Especially if you wanna learn new things about floating point numbers. – John Leidegren Aug 5, 2009 at 17:20

2 The annoying thing about this is if you're going to use this is a struct the compiler will force you to set ALL the variables in the init function. So if you have: public A(int int32) { Int32 = int32; } it will throw "Field 'One' must be fully assigned before control is returned to the caller", so you have to set One = Two = Three = Four = 0; as well. – manixrock Oct 1, 2009 at 16:27

2 This has its uses, mostly with binary data. I use a "Pixel" struct with an int32 @ 0, and four bytes for the four components @ 0, 1, 2, and 3. Great for manipulating image data quickly and easily. – snarf Oct 16, 2009 at 23:01

57 **Warning: This approach does not take into account endianness.** That means your C# code will not run the same way on all machines. On little-endian CPUs (which store the least significant byte first), the behavior shown will be used. But on big-endian CPUs, the bytes will come out reversed from what you expected. Beware how you use this in production code - your

code may not be portable to certain mobile devices and other hardware, and may break in non-obvious ways (eg two files ostensibly in the same format but actually with byte order reversed).
– Ray Burns Jan 25, 2010 at 19:14

175 votes

Using @ for variable names that are keywords.

```
var @object = new object();
var @string = "";
var @if = IpsoFacto();
```

Share answered Aug 18, 2008 at 1:45 community wiki Mark Cidade

38 Why would you want to use a keyword as a variable name? Seems to me that this would make code less readable and obfuscated. – Jon Sep 13, 2008 at 4:18

41 Well, the reason that it's there is that CLI requires it for interoperability with other languages that might use C# keywords as member names – Mark Cidade Sep 18, 2008 at 22:04

69 If you ever wanted to use the asp.net MVC HTML helpers and define a HTML class you will be happy to know that you can use @class so it won't be recognised as the class keyword
– Boris Callens Sep 23, 2008 at 7:18

31 Great for extension methods. public static void DoSomething(this Bar @this, string foo) { ... }
– Jonathan C Dickinson Jan 7, 2009 at 11:06

45 @zihotki: Wrong. var a = 5; Console.WriteLine(@a); Prints 5 – SLaks Jun 4, 2009 at 17:40

166 votes

If you want to exit your program without calling any finally blocks or finalizers use FailFast:

```
Environment.FailFast()
```

Share edited Dec 19, 2009 at 3:06 community wiki 2 revs, 2 users 50% Jan Bannister

12 Note that this method also creates a memory dump and writes a message to the Windows error log. – RickNZ Dec 19, 2009 at 3:08

1 It should be noted no finalizers or app domain unload events will be called when this method is used... – AwkwardCoder Jan 6, 2010 at 11:43

1 +1 for the hint, but this is not C#, it is BCL of .NET. – Abel Mar 10, 2011 at 22:59

System.Diagnostics.Process.GetCurrentProcess().Kill() is faster – Tolgahan Albayrak Oct 30, 2011 at 0:05

153 votes

Returning anonymous types from a method and accessing members without reflection.

```
// Useful? probably not.
private void foo()
{
    var user = AnonCast(GetUserTuple(), new { Name = default(string), Badges =
default(int) });
    Console.WriteLine("Name: {0} Badges: {1}", user.Name, user.Badges);
}

object GetUserTuple()
{
    return new { Name = "dp", Badges = 5 };
}

// Using the magic of Type Inference...
static T AnonCast<T>(object obj, T t)
{
   return (T) obj;
}
```

Share

edited Apr 6, 2011 at 13:39

community wiki
3 revs, 3 users 95%
denis phillips

42 That really doesn't get you anything. It is actually dangerous. What if GetUserTuple is modified to return multiple types? The cast will fail at run time. One of the great things about C#/.Net is the compile time checking. It would be much better to just create a new type. – Jason Jackson Sep 30, 2008 at 1:10

9 @Jason I did say it's probably not useful but it is surprising (and I thought hidden). – denis phillips Sep 30, 2008 at 14:41

31 While cool, this seems like a rather poor design choice. You've basically defined the anonymous type in two places. At that point, just declare a real struct and use it directly. – Paul Alexander May 7, 2009 at 9:05

6 @George: such a convention would be called a... struct? – R. Martinho Fernandes Nov 13, 2009 at 11:58

2 this trick is named 'cast by sample' and it will not work if method that returns anonymous type is located in another assembly. – desco Jul 3, 2010 at 0:58

146 votes

Here's a useful one for regular expressions and file paths:

```
"c:\\program files\\oldway"
@"c:\program file\newway"
```

The @ tells the compiler to ignore any escape characters in a string.

Share

edited Aug 12, 2008 at 19:01

community wiki
2 revs
Patrick

27 Also, a @ constant accepts newlines inside. Perfect when assigning a multiline script to a string. – Tor Haugen Nov 19, 2008 at 16:17

11 Don't forget also to escape a quotation mark just double them up, in other words. [code]var candy = @"I like ""red"" candy canes.";[/code] – David Jan 10, 2009 at 14:39

5 I tend to build paths with Path.Combine. I definitely use the @ for regex! – Dan McClain Jun 16, 2009 at 12:45

6 @new is also a variable instead of a keyword: @this, @int, @return, @interface...so on :) – IAbstract Jan 28, 2010 at 5:59

This one doesn't come anywhere near: *But what are the most hidden features or tricks of C# that even C# fans, addicts, experts barely know?* – publicgk Aug 16, 2011 at 11:15

141
votes

Mixins. Basically, if you want to add a feature to several classes, but cannot use one base class for all of them, get each class to implement an interface (with no members). Then, write an extension method **for the interface**, i.e.

```
public static DeepCopy(this IPrototype p) { ... }
```

Of course, some clarity is sacrificed. But it works!

Share

edited Apr 22, 2011 at 19:36

community wiki
2 revs
Dmitri Nesteruk

4 Yeah I think this is the real power of extension methods. They basically allow for implementations of interface methods. – John B ♦ Jun 8, 2009 at 16:23

This is also handy if you're using NHibernate (or Castle ActiveRecord) and you have to use interfaces for your collections. This way you can give behavior to the collection interfaces. – Ryan Lundy Mar 5, 2010 at 22:28

That's basically how all of the LINQ methods are implemented, for the record. – Isabelle Wedin Oct 4, 2010 at 20:13

Here's a link that talks about what I referred to above, using extension methods with collection interfaces in NHibernate or Castle ActiveRecord: devlicio.us/blogs/billy_mccafferty/archive/2008/09/03/… – Ryan Lundy Mar 23, 2011 at 22:34

7 If only they allowed extension properties!!!! I hated having to write an extension method that begged to be a read-only property.... – John Gibb Apr 18, 2011 at 17:40

---

130 votes

> Not sure why anyone would ever want to use Nullable<bool> though. :-)

True, False, FileNotFound?

Share edited Nov 22, 2008 at 7:39 community wiki 2 revs, 2 users 91% Michael Stum

87 if expect a user to answer a yes no question then null would be appropriate if the question has not been answered – Omar Kooheji Oct 24, 2008 at 12:38

22 Nullable types are handy for interaction with a database where table columns are often nullable. – tuinstoel Jan 1, 2009 at 16:40

11 Yes, No, Maybee? – Dan Blair May 22, 2009 at 20:19

21 Store values of a ThreeState CheckBox – Shimmy Weitzhandler Jul 13, 2009 at 15:04

8 As in SQL: Yes/no/unknown. – erikkallen Sep 9, 2009 at 11:30

---

116 votes

This one is not "hidden" so much as it is misnamed.

A lot of attention is paid to the algorithms "map", "reduce", and "filter". What most people don't realize is that .NET 3.5 added all three of these algorithms, but it gave them very SQL-ish names, based on the fact that they're part of LINQ.

> "map" => Select
> Transforms data from one form into another

> "reduce" => Aggregate
> Aggregates values into a single result
>
> "filter" => Where
> Filters data based on a criteria

The ability to use LINQ to do inline work on collections that used to take iteration and conditionals can be incredibly valuable. It's worth learning how all the LINQ extension methods can help make your code much more compact and maintainable.

Share answered Aug 16, 2008 at 23:55 community wiki Brad Wilson

1 Select also acts like the "return" function in monads. See stackoverflow.com/questions/9033/hidden-features-of-c#405088 – Mark Cidade Jan 1, 2009 at 16:15

1 Using "select" is only required if you use the SQLish syntax. If you use the extension method syntax -- someCollection.Where(item => item.Price > 5.99M) -- the use of select statements isn't required. – Brad Wilson Apr 1, 2010 at 18:41

7 @Brad, that (where) is a filter operation. Try doing a map operation without select... – Eloff May 10, 2010 at 22:31

2 LINQ is the big thing that happened to C# in my opinion: stackoverflow.com/questions/2398818/… – Leniel Maccaferri May 29, 2010 at 20:53

1 Aggregate's smallest signature is the "reduce" function, Aggregate's middle signature is the much more powerful "fold" function! – Brett Widmeier Mar 14, 2011 at 22:50

115
votes

```
Environment.NewLine
```

for system independent newlines.

Share edited Sep 27, 2009 at 19:25 community wiki
2 revs, 2 users 67%
nt.

10 The annoying thing about this one, is that it isn't included into the compact framework. – Stormenet Dec 8, 2008 at 7:21

14 Its worth pointing out that this is specific to the application's host platform - so if you are creating data intended for another system, you should use \n or \r\n appropriately. – Mesh Jun 2, 2009 at 12:04

This is part of the BCL of .NET, not a feature of C# per se. – Abel Mar 10, 2011 at 23:01

111
votes

If you're trying to use curly brackets inside a String.Format expression...

```
int foo = 3;
string bar = "blind mice";
String.Format("{{I am in brackets!}} {0} {1}", foo, bar);
//Outputs "{I am in brackets!} 3 blind mice"
```

Share edited May 19, 2010 at 14:43 community wiki
3 revs, 2 users 85%
Portman

19 @Kyralessa: Actually, yes, they are braces, but "curly brackets" is an alternate name for them. `[` and `]` are square brackets, and `<` and `>` are angle brackets. See en.wikipedia.org/wiki/Bracket. – icktoofay Mar 26, 2010 at 1:22

4 You are correct. But when I commented, it didn't say "curly" brackets. – Ryan Lundy Mar 26, 2010 at 2:04

2 Very nice one to point out. Remember my first String.Format looked like: String.Format("{0}I am in curly brackets{1} {2} {3}", "{","}", 3, "blind mice"); Once I found out that escaping these is done by using {{ and }} I was so happy :) – Gertjan Apr 6, 2010 at 14:04

Muahahahaa... Programming already 3 years in .Net and I didn't know this one. :D – Arnis Lapsa Apr 23, 2010 at 10:54

"curly" brackets acting similar to escape sequences ?? – Pratik Dec 14, 2010 at 10:48

104
votes

1. ?? - coalescing operator

2. using (statement / directive) - great keyword that can be used for more than just calling Dispose
3. readonly - should be used more
4. netmodules - too bad there's no support in Visual Studio

Share edited Apr 30, 2010 at 10:31 community wiki
4 revs, 3 users 53%
Doctor Jones

6 using can also be used to alias a long namespace to a more convenient string, i.e.: using ZipEncode = MyCompany.UtilityCode.Compression.Zip.Encoding; There's more here: msdn.microsoft.com/en-us/library/sf0df423.aspx – Dave R. Dec 9, 2008 at 15:41

2 It really isn't the same thing. When calling Dispose, you may use the using statement, when aliasing types you are using a using directive. – Øyvind Skaar Jan 11, 2009 at 19:29

2 Just in case you'd like a name to #1 (as you did with the ternary operator), ?? is called the null coalescing operator. – J. Steen Apr 23, 2009 at 12:47

4 @LucasAardvark: As J Steen mentioned it's called the null coalescing operator. Search for that! – kokos Jun 21, 2009 at 1:54

1 To search for ?? operator at google try: google.com/search?q=c%23+%3F%3F+operator – backslash17 Dec 25, 2009 at 22:06

103 votes

@Ed, I'm a bit reticent about posting this as it's little more than nitpicking. However, I would point out that in your code sample:

```
MyClass c;
  if (obj is MyClass)
    c = obj as MyClass
```

If you're going to use 'is', why follow it up with a safe cast using 'as'? If you've ascertained that obj is indeed MyClass, a bog-standard cast:

```
c = (MyClass)obj
```

...is never going to fail.

Similarly, you could just say:

```
MyClass c = obj as MyClass;
if(c != null)
{
   ...
}
```

I don't know enough about .NET's innards to be sure, but my instincts tell me that this would cut a maximum of two type casts operations down to a maximum of one. It's hardly likely to break the processing bank either way; personally, I think the latter form looks cleaner too.

Share answered Aug 12, 2008 at 18:03 community wiki
Dogmang

16 If the cast is to the exact type (cast to "A" when object is "A", not derived from it), the straight cast is ~3x FASTER than "as". When casting a derived type (cast to "A" when object is "B", which derives from "A"), the straight cast is ~0.1x slower than "as". "is", then "as" is just silly. – P Daddy Nov 17, 2008 at 7:15

2 No, but you could write "if ((c = obj as MyClass) != null)". – Dave Van den Eynde Mar 25, 2009 at 12:14

10 `is` and `as` won't do user casts. So, the above code is asking with the `is` operator if obj is derived from MyClass (or has an implicit system defined cast). Also, `is` fails on `null`. Both of these edge cases may be important to your code. For instance, you may want to write: `if( obj == null || obj is MyClass ) c = (MyClass)obj;` But this is strictly different from: `try { c = (MyClass)obj; } catch { }` since the former will not perform any user defined conversions, but the latter will. Without the `null` check, the former will also not set `c` when `obj` is `null`. – Adam Luter Sep 9, 2009 at 12:02

2 In IL, a cast goes to a CASTCLASS but an as/is go to an ISINST instruction. – John Gibb Apr 18, 2011 at 17:39

5 I ran a test for this, casting `IEnumerable<int>` to `List<int>`, and casting `object` (`new object()`) to `IEnumerable<int>`, to make sure there are no mistakes: direct cast: 5.43ns, is->as cast: 6.75ns, as cast: 5.69ns. Then testing invalid casts: direct cast: 3125000ns, as cast: 5.41ns. Conclusion: stop worrying about the 1% factor, and just make sure you use is/as when the cast might be invalid, cause exceptions (also if handled) are VERY slow compared to casting, we're talking about a factor 578000 slower. Remember that last part, the rest doesn't matter (.Net FW 4.0, release build) – Aidiakapi Sep 13, 2011 at 11:06

98 votes

Maybe not an advanced technique, but one I see all the time that drives me crazy:

```
if (x == 1)
{
   x = 2;
}
else
{
   x = 3;
}
```

can be condensed to:

```
x = (x==1) ? 2 : 3;
```

Share

answered Aug 19, 2008 at 15:54

community wiki
JasonS

10 The ternary operator has been banned in my group. Some people just don't like it, though I've never understood why. – Justin R. Mar 26, 2009 at 19:06

13 I guess because they're not the same things, being an operator not a statement. I prefer the first approach myself. It's more consistent and easily expandable. Operators can't contain statements, so the minute you have to expand the body, you'll have to convert that ternary operator to an if statement – kervin Apr 18, 2009 at 16:34

16 can be condensed to x++; Ok it's pointless ^^ – François Jun 4, 2009 at 13:14

5 @Guillaume: To account for all values of x: x = 2 + System.Math.Min(1,System.Math.Abs(x-1)); – mbeckish Jun 6, 2009 at 20:17

38 It's actually called the "conditional operator" - it is *a* ternary operator because it takes three arguments. en.wikipedia.org/wiki/Conditional_operator – Blorgbeard Jun 30, 2009 at 9:24