Switch statement fallthrough in C#?

Asked 16 years, 2 months ago   Modified 1 year, 1 month ago   Viewed 243k times

429

Switch statement fallthrough is one of my personal major reasons for loving `switch` vs. `if/else if` constructs. An example is in order here:

```
static string NumberToWords(int number)
{
    string[] numbers = new string[]
        { "", "one", "two", "three", "four", "five",
          "six", "seven", "eight", "nine" };
    string[] tens = new string[]
        { "", "", "twenty", "thirty", "forty", "fifty",
          "sixty", "seventy", "eighty", "ninety" };
    string[] teens = new string[]
        { "ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen",
          "sixteen", "seventeen", "eighteen", "nineteen" };

    string ans = "";
    switch (number.ToString().Length)
    {
        case 3:
            ans += string.Format("{0} hundred and ", numbers[number / 100]);
        case 2:
            int t = (number / 10) % 10;
            if (t == 1)
            {
                ans += teens[number % 10];
                break;
            }
            else if (t > 1)
                ans += string.Format("{0}-", tens[t]);
        case 1:
            int o = number % 10;
            ans += numbers[o];

            break;
        default:
            throw new ArgumentException("number");
    }
    return ans;
}
```

The smart people are cringing because the `string[]`s should be declared outside the function: well, they are, this is just an example.

The compiler fails with the following error:

```
Control cannot fall through from one case label ('case 3:') to another
Control cannot fall through from one case label ('case 2:') to another
```

Why? And is there any way to get this sort of behaviour without having three `if`s?

c# switch-statement

Share Improve this question Follow

asked Oct 6, 2008 at 13:00
Matthew Scharley
132k ● 55 ● 196 ● 225

14 Answers

Sorted by: Highest score (default)

749

(Copy/paste of an answer I provided elsewhere)

Falling through `switch`-`case`s can be achieved by having no code in a `case` (see `case 0`), or using the special `goto case` (see `case 1`) or `goto default` (see `case 2`) forms:

```
switch (/*...*/) {
    case 0: // shares the exact same code as case 1
    case 1:
        // do something
        goto case 2;
    case 2:
        // do something else
        goto default;
    default:
        // do something entirely different
        break;
}
```

Share
Improve this answer
Follow

edited Nov 1, 2023 at 16:17
TylerH
21.2k ● 76 ● 79 ● 110

answered Oct 6, 2008 at 13:13
Alex Lyman
15.9k ● 4 ● 40 ● 42

162 I think that, in this particular instance, goto is not considered harmful. – Thomas Owens Oct 6, 2008 at 19:32

16 Would it be overkill to create a new reserved word `fallthrough` that you could use to indicate an explicit desire to allow fall through. The compiler could then check for accidental fall through, but allow purposeful usage. – Dancrumb Sep 9, 2011 at 15:16

15 @Dancrumb: At the time that the feature was written, C# had not yet added any "soft" keywords (like 'yield', 'var', 'from', and 'select'), so they had three real options: 1) make 'fallthrough' a hard keyword (you cant use it as a variable name), 2) write code necessary to support such an soft keyword, 3) use already reserved keywords. #1 was a big issue for those porting code; #2 was a fairly large engineering task, from what I understand; and the option they went with, #3 had a side benefit: other devs reading the code after the fact could

learn about the feature from the base concept of goto – Alex Lyman Oct 14, 2011 at 19:43

19 All this talk about new/special keywords for an explicit fallthrough. Couldn't they just have used the 'continue' keyword? In other words. Break out of the switch or continue onto the next case (fall through). – Tal Even-Tov Nov 12, 2013 at 11:32

9 @scott.korin Why? Would you prefer implicit fall-through? I'm pretty sure that the last 30 years of C-like language usage has told us how horrible an idea that is. The only other real option is not even supporting this, and forcing either code duplication, or (gasp) standard goto-labels. – Alex Lyman Dec 12, 2014 at 3:38

57

The "why" is to avoid accidental fall-through, for which I'm grateful. This is a not uncommon source of bugs in C and Java.

The workaround is to use goto, e.g.

```
switch (number.ToString().Length)
{
    case 3:
        ans += string.Format("{0} hundred and ", numbers[number / 100]);
        goto case 2;
    case 2:
    // Etc
}
```

The general design of switch/case is a little bit unfortunate in my view. It stuck too close to C - there are some useful changes which could be made in terms of scoping etc. Arguably a smarter switch which could do pattern matching etc would be helpful, but that's really changing from switch to "check a sequence of conditions" - at which point a different name would perhaps be called for.

Share
Improve this answer
Follow

edited Apr 11, 2018 at 21:24
Jens Bannmann
5,055 ● 7 ● 52 ● 81

answered Oct 6, 2008 at 13:13
Jon Skeet
1.5m ● 889 ● 9.3k ● 9.3k

1 This is the difference between switch and if/elseif in my mind. Switch is for checking various states of a single variable, whereas if/elseif can be used to check any number of things that connected, but not necessarily a single, or the same variable. – Matthew Scharley Oct 6, 2008 at 13:19

2 If it was to prevent accidental fall through then I feel a compiler warning would have been better. Just like you have one if your if statement has an assignment: `if (result =  true) { }` – Tal Even-Tov Nov 12, 2013 at 11:38

6 @TalEven-Tov: Compiler warnings should really be for cases where you can almost always fix the code to be better. Personally I'd prefer implicit breaking, so it wouldn't be a problem to start with, but that's a different matter. – Jon Skeet Nov 12, 2013 at 11:52

1 The annoying thing is that they still make you end the cases with `break`. – Derf Skren Nov 2, 2020 at 2:57

1 Newer versions of C# have mercifully added pattern matching switch statements. – sonicbhoc Aug 11, 2021 at 16:56

32

To add to the answers here, I think it's worth considering the opposite question in conjunction with this, viz. why did C allow fall-through in the first place?

Any programming language of course serves two goals:

1. Provide instructions to the computer.
2. Leave a record of the intentions of the programmer.

The creation of any programming language is therefore a balance between how to best serve these two goals. On the one hand, the easier it is to turn into computer instructions (whether those are machine code, bytecode like IL, or the instructions are interpreted on execution) then more able that process of compilation or interpretation will be to be efficient, reliable and compact in output. Taken to its extreme, this goal results in our just writing in assembly, IL, or even raw op-codes, because the easiest compilation is where there is no compilation at all.

Conversely, the more the language expresses the intention of the programmer, rather than the means taken to that end, the more understandable the program both when writing and during maintenance.

Now, `switch` could always have been compiled by converting it into the equivalent chain of `if-else` blocks or similar, but it was designed as allowing compilation into a particular common assembly pattern where one takes a value, computes an offset from it (whether by looking up a table indexed by a perfect hash of the value, or by actual arithmetic on the value*). It's worth noting at this point that today, C# compilation will sometimes turn `switch` into the equivalent `if-else`, and sometimes use a hash-based jump approach (and likewise with C, C++, and other languages with comparable syntax).

In this case there are two good reasons for allowing fall-through:

1. It just happens naturally anyway: if you build a jump table into a set of instructions, and one of the earlier batches of instructions doesn't contain some sort of jump or return, then execution will just naturally progress into the next batch. Allowing fall-through was what would "just happen" if you turned the `switch`-using C into jump-table–using machine code.
2. Coders who wrote in assembly were already used to the equivalent: when writing a jump table by hand in assembly, they would have to consider whether a given

block of code would end with a return, a jump outside of the table, or just continue on to the next block. As such, having the coder add an explicit `break` when necessary was "natural" for the coder too.

At the time therefore, it was a reasonable attempt to balance the two goals of a computer language as it relates to both the produced machine code, and the expressiveness of the source code.

Four decades later though, things are not quite the same, for a few reasons:

1. Coders in C today may have little or no assembly experience. Coders in many other C-style languages are even less likely to (especially Javascript!). Any concept of "what people are used to from assembly" is no longer relevant.
2. Improvements in optimisations mean that the likelihood of `switch` either being turned into `if-else` because it was deemed the approach likely to be most efficient, or else turned into a particularly esoteric variant of the jump-table approach are higher. The mapping between the higher- and lower-level approaches is not as strong as it once was.
3. Experience has shown that fall-through tends to be the minority case rather than the norm (a study of Sun's compiler found 3% of `switch` blocks used a fall-through other than multiple labels on the same block, and it was thought that the use-case here meant that this 3% was in fact much higher than normal). So the language as studied make the unusual more readily catered-to than the common.
4. Experience has shown that fall-through tends to be the source of problems both in cases where it is accidentally done, and also in cases where correct fall-through is missed by someone maintaining the code. This latter is a subtle addition to the bugs associated with fall-through, because even if your code is perfectly bug-free, your fall-through can still cause problems.

Related to those last two points, consider the following quote from the current edition of K&R:

> Falling through from one case to another is not robust, being prone to disintegration when the program is modified. With the exception of multiple labels for a single computation, fall-throughs should be used sparingly, and commented.
>
> As a matter of good form, put a break after the last case (the default here) even though it's logically unnecessary. Some day when another case gets added at the end, this bit of defensive programming will save you.

So, from the horse's mouth, fall-through in C is problematic. It's considered good practice to always document fall-throughs with comments, which is an application of the general principle that one should document where one does something unusual, because that's what will trip later examination of the code and/or make your code look like it has a novice's bug in it when it is in fact correct.

And when you think about it, code like this:

```
switch(x)
{
  case 1:
   foo();
   /* FALLTHRU */
  case 2:
    bar();
    break;
}
```

Is adding something to make the fall-through explicit in the code, it's just not something that can be detected (or whose absence can be detected) by the compiler.

As such, the fact that on has to be explicit with fall-through in C# doesn't add any penalty to people who wrote well in other C-style languages anyway, since they would already be explicit in their fall-throughs.†

Finally, the use of `goto` here is already a norm from C and other such languages:

```
switch(x)
{
  case 0:
  case 1:
  case 2:
    foo();
    goto below_six;
  case 3:
    bar();
    goto below_six;
  case 4:
    baz();
    /* FALLTHRU */
  case 5:
  below_six:
    qux();
    break;
  default:
    quux();
}
```

In this sort of case where we want a block to be included in the code executed for a value other than just that which brings one to the preceding block, then we're already having to use `goto`. (Of course, there are means and ways of avoiding this with different conditionals but that's true of just about everything relating to this question). As such C# built on the already normal way to deal with one situation where we want to hit more than one block of code in a `switch`, and just generalised it to cover fall-through as well. It also made both cases more convenient and self-documenting, since we have to add a new label in C but can use the `case` as a label in C#. In C# we can get rid of the `below_six` label and use `goto case 5` which is clearer as to what we are doing. (We'd also have to add `break` for the `default`, which I left out just to make the above C code clearly not C# code).

In summary therefore:

1. C# no longer relates to unoptimised compiler output as directly as C code did 40 years ago (nor does C these days), which makes one of the inspirations of fall-through irrelevant.
2. C# remains compatible with C in not just having implicit `break`, for easier learning of the language by those familiar with similar languages, and easier porting.
3. C# removes a possible source of bugs or misunderstood code that has been well-documented as causing problems for the last four decades.
4. C# makes existing best-practice with C (document fall through) enforceable by the compiler.
5. C# makes the unusual case the one with more explicit code, the usual case the one with the code one just writes automatically.
6. C# uses the same `goto`-based approach for hitting the same block from different `case` labels as is used in C. It just generalises it to some other cases.
7. C# makes that `goto`-based approach more convenient, and clearer, than it is in C, by allowing `case` statements to act as labels.

All in all, a pretty reasonable design decision

---

*Some forms of BASIC would allow one to do the likes of `GOTO (x AND 7) * 50 + 240` which while brittle and hence a particularly persuasive case for banning `goto`, does serve to show a higher-language equivalent of the sort of way that lower-level code can make a jump based on arithmetic upon a value, which is much more reasonable when it's the result of compilation rather than something that has to be maintained manually. Implementations of Duff's Device in particular lend themselves well to the equivalent machine code or IL because each block of instructions will often be the same length without needing the addition of `nop` fillers.

†Duff's Device comes up here again, as a reasonable exception. The fact that with that and similar patterns there's a repetition of operations serves to make the use of

fall-through relatively clear even without an explicit comment to that effect.

Share

Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Dec 9, 2013 at 12:15

Jon Hanna

113k ● 10 ● 149 ● 257

I'm fairly sure BASICs vary in whether they allow computed `GOTO`. The fragility of this - even more so if the BASIC in question has `RENUM` - is perhaps one motivation for allowing you to `GOTO` only a hardcoded line number. I suspect some BASICs optimise the code in such a way that computed `GOTO` can't be easily implemented. (If you dare, check out Come Here: wwwep.stewartsplace.org.uk/languages/comehere) – Stewart Jan 17 at 10:41

27

Switch fallthrough is historically one of the major source of bugs in modern softwares. The language designer decided to make it mandatory to jump at the end of the case, unless you are defaulting to the next case directly without processing.

```
switch(value)
{
    case 1:// this is still legal
    case 2:
}
```

Share Improve this answer Follow

answered Oct 6, 2008 at 13:06

Coincoin

28.5k ● 7 ● 56 ● 76

24 I well never understand why that isn't "case 1, 2:" – BCS Feb 9, 2009 at 23:02

11 @David Pfeffer: Yup it is, and so is `case 1, 2:` in languages that allow that. What I will never understand is why any modern language would't choose to allow that. – BCS Oct 18, 2010 at 13:54

@BCS with the goto statement, multiple comma separated options could be tricky to handle? – penguat Sep 13, 2011 at 12:44

1 @pengut: it might be more accurate to say that `case 1, 2:` is a single label but with multiple names. -- FWIW, I think most languages that forbid fall through don't special case the "consecutive case labels" indium but rather treat the case labels as an annotation on the next statement and require the last statement before a statement labeled with (one or more) case labels to be a jump. – BCS Sep 13, 2011 at 16:23

@BCS in C and other C-Like lenguages (not sure about C#) comma (",") is an operator, so it is not good idea to redefine its behaviour just for this "case". – Will Nov 7, 2020 at 5:40

You can 'goto case label' http://www.blackwasp.co.uk/CSharpGoto.aspx

18

> The goto statement is a simple command that unconditionally transfers the control of the program to another statement. The command is often criticised with some developers advocating its removal from all high-level programming languages because it can lead to *spaghetti code*. This occurs when there are so many goto statements or similar jump statements that the code becomes difficult to read and maintain. However, there are programmers who point out that the goto statement, when used carefully, provides an elegant solution to some problems...

Share
Improve this answer
Follow

edited Oct 1, 2016 at 20:29
gnat
6,218 ● 113 ● 55 ● 75

answered Oct 6, 2008 at 13:08
kenny
22.3k ● 8 ● 51 ● 87

7

They left out this behaviour by design to avoid when it was not used by will but caused problems.

It can be used only if there is no statement in the case part, like:

```
switch (whatever)
{
    case 1:
    case 2:
    case 3: boo; break;
}
```

Share Improve this answer Follow

answered Oct 6, 2008 at 13:04
Biri
7,171 ● 7 ● 39 ● 52

5

They changed the switch statement (from C/Java/C++) behavior for c#. I guess the reasoning was that people forgot about the fall through and errors were caused. One book I read said to use goto to simulate, but this doesn't sound like a good solution to me.

Share Improve this answer Follow

answered Oct 6, 2008 at 13:03
Ken
2,092 ● 1 ● 19 ● 16

2 C# supports goto, but not fallthrough? Wow. And it's not just those. C# is the only language that I know of that behaves this way. – Matthew Scharley Oct 6, 2008 at 13:06

1 I didn't exactly like it at first, but "fall-thru" is really a recipe for disaster (especially amongst junior programmers.) As many have pointed out, C# still allows fall-thru for empty lines (which

is the majority of cases.) "Kenny" posted a link that highlights elegant Goto use with switch-case. – Pretzel Oct 6, 2008 at 13:39

It's not a huge deal in my opinion. 99% of the time I don't want a fall through and I have been burned by bugs in the past. – Ken Oct 6, 2008 at 14:08

2 "this doesn't sound like a good solution to me" -- sorry to hear that about you, because that's what `goto case` is for. Its advantage over fallthrough is that it is explicit. That some people here object to `goto case` just shows that they were indoctrinated against "goto" without any understanding of the issue and are incapable of thinking on their own. When Dijkstra wrote "GOTO Considered Harmful", he was addressing languages that didn't have any other means of changing control flow. – Jim Balter Oct 24, 2013 at 9:03

@JimBalter and then how many people who quote Dijkstra on that will quote Knuth that "premature optimisation is the root of all evil" though that quote was when Knuth was explicitly writing about how useful `goto` can be when optimising code? – Jon Hanna Dec 8, 2013 at 15:00

0

switch (C# Reference) says

> C# requires the end of switch sections, including the final one,

So you also need to add a `break;` to your `default` section, otherwise there will still will be a compiler error.

Share
Improve this answer
Follow

edited Oct 7, 2014 at 22:53

answered Oct 7, 2014 at 9:33
gm2008
4,325 ● 1 ● 37 ● 38

It's one thing for a language to make a design decision to avoid programming pitfalls, but there is no way you can honestly say that permitting a fallthrough on the final statement could in any way be harmful. – Paul Childs Aug 26, 2021 at 22:44

-1

> A jump statement such as a break is required after each case block, including the last block whether it is a case statement or a default statement. With one exception, (unlike the C++ switch statement), C# does not support an implicit fall through from one case label to another. The one exception is if a case statement has no code.

-- C# switch() documentation

Share Improve this answer Follow

answered Oct 6, 2008 at 13:03
Powerlord
88.7k ● 17 ● 129 ● 177

1 I realise this behaviour is documented, I want to know WHY it is the way it is, and any alternatives to get the old behaviour. – Matthew Scharley Oct 6, 2008 at 13:05

-1

After each case statement require **break** or **goto** statement even if it is a default case.

Share Improve this answer Follow

answered Jun 4, 2010 at 9:02
Shilpa gulati
31 ● 1

4 If only someone had posted that two years earlier! – user146043 Mar 13, 2012 at 15:25

1 @Poldie it was funny the first time... Shilpa you don't require a break or goto for every case, just for every case with it's own code. You can have multiple cases that share code just fine. – Maverick Oct 14, 2013 at 22:05

-1

You can achieve fall through like c++ by the goto keyword.

EX:

```
switch(num)
{
   case 1:
      goto case 3;
   case 2:
      goto case 3;
   case 3:
      //do something
      break;
   case 4:
      //do something else
      break;
   case default:
      break;
}
```

Share Improve this answer Follow

answered Jul 28, 2010 at 20:30
Dai Tran
39 ● 1

11 If only someone had posted that two years earlier! – user146043 Mar 13, 2012 at 15:25

I'm quite sure C# allows fallthrough in the specific case of an empty case body. So you can do `case 1: case 2: doSomething(); break;` but as soon as you add a statement between `case 1:` and `case 2:`, it will no longer allow fall through. – Stewart May 21 at 22:14

-1

Just a quick note to add that the compiler for Xamarin actually got this wrong and it allows fallthrough. It has supposedly been fixed, but has not been released. Discovered this in some code that actually was falling through and the compiler did not complain.

Share Improve this answer Follow

answered Feb 14, 2014 at 22:19

user486646

-1

This post is about trying to justify why the disallowing of fall-through cases in a switch is not a nice thing. I've also provided an example of where cascading fall-through of cases actually works really well and is particularly difficult to achieve with such good results any other way. Sure, It's a niche example, but it's still a worthy use-case.

The problem here with the disallowing of fall-through on cases within a switch, is that it prevents you being able to do certain loop-unrolls. Whilst I can appreciate some of the other viewpoints here, I don't think that fall-through is intrinsically bad or requires a deeper level of understanding.

I ran into an interesting case where I really needed fall-through and having to use goto, almost seemed like a bit of a cludge. It was where I doing some bit manipulation and depending on how many bits that were left, I needed to cascade through cases. This resulted in the following switch:

```
switch (bitsLeft % 11)
{
    case 10:
       store.Append((byte)data[3] << 3);
       goto case 9;
    case 9:
       store.Append((byte)(data[2] << 7 | data[4] >> 1));
       goto case 8;
    case 8:
       store.Append((byte)(data[0] >> 2 | data[1] >> 4));
       goto case 7;
    case 7:
       store.Append((byte)(data[4] << 6 | data[5] << 4));
       goto case 6;
    case 6:
       store.Append((byte)(data[6] << 7 | data[0] >> 5));
       goto case 5;
    case 5:
       store.Append((byte)data[7] << 1);
```

```
        goto case 4;
    case 4:
        store.Append((byte)(data[7] >> 2 | data[6] << 2));
        goto case 3;
    case 3:
        store.Append((byte)(data[8] << 3 | data[3] >> 5));
        goto case 2;
    case 2:
        store.Append((byte)(data[4] >> 1 | data[6] << 1));
        goto case 1;
    case 1:
        store.Append((byte)data[7] << 2 | data[2] << 1));
        break;
    default: break;
}
```

I just don't see how disallowing fall-through and being forced to use goto really adds anything, especially in situations where you want to cascade cases. Moreover, writing the above with if/else-if would amount to huge amounts of code duplication.

It's also hard to accept the reasoning that the lack of fall-through is due to the large amount of bugs caused by it. I put this reasoning into the same sort of category as using braces for single-line if statements. Using the practise of **always** adding braces **BEFORE** altering code below an if condition really prevents these kinds of bugs.

The same strategy can equally be applied to switch statements. When writing a switch, include all of the cases and add a *break;* for each. Only then that start implementing the code for each case. There's never any risk that a case could fall-through because all cases have break statements right from the outset. Okay, it is possible that adding cases to a switch after implementation could lead to a forgotten break; but starting a case implementation with a break statement and writing code above it is always going to prevent accidental fall-through. Moreover, many switches have cases that are dependent on an enumeration, so providing empty cases with just a break before starting implementation is trivial.

My case is quite niche and disallowing of fall-through cases probably suits most people's needs, but it does seem a little harsh that it's not a configurable option. Maybe using annotations would have been a better way to go?

Share

Improve this answer

Follow

edited Nov 1, 2023 at 3:55

answered Oct 31, 2023 at 20:41

The Welder

1,052 ● 10 ● 25

-12

You forgot to add the "break;" statement into case 3. In case 2 you wrote it into the if block. Therefore try this:

```
case 3:
{
    ans += string.Format("{0} hundred and ", numbers[number / 100]);
    break;
}


case 2:
{
    int t = (number / 10) % 10;
    if (t == 1)
    {
        ans += teens[number % 10];
    }
    else if (t > 1)
    {
        ans += string.Format("{0}-", tens[t]);
    }
    break;
}

case 1:
{
    int o = number % 10;
    ans += numbers[o];
    break;
}

default:
{
    throw new ArgumentException("number");
}
```

Share Improve this answer Follow

answered Oct 6, 2008 at 13:10

Marcus
1,125 ● 2 ● 21 ● 36

2 This produces vastly wrong output. I left the switch statements out by design. The question is why the C# compiler sees this as an error when almost no other language has this restriction.
– Matthew Scharley Oct 6, 2008 at 13:13

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.