How to elegantly check if a number is within a range?

Ask Question

128

23

How can I do this elegantly with C# and .NET 3.5/4?

For example, a number can be between 1 and 100.

I know a simple if would suffice; but the keyword to this question is elegance. It's for my toy project not for production.

This questions wasn't about speed, but about code beauty. Stop talking about efficiency and such; remember you're preaching to the choir.

c# numbers int

edited May 19 '16 at 23:16

Peter Mortensen
14.1k 19 88 114

asked Jul 6 '10 at 17:30

Sergio Tapia
16k 66 162 246

21 Re: Your "edit" - **simple is elegant**. I personally find the if statement more elegant than any non-standard means of doing this check... – Reed Copsey Jul 6 '10 at 17:37

4 "Everything should be made as simple as possible, but not simpler." - Albert Einstein – corsiKa Jul 6 '10 at 17:40

3 @Sergio: I don't feel I am being pedantic. I feel that people

using anything but the more obvious is a poor choice, IMO. – Reed Copsey Jul 6 '10 at 17:45

3 @Sergio: I guess, then, I don't see the point of the question ;) – Reed Copsey Jul 6 '10 at 17:52

6 @Sergio: if `if` ain't "baroque" don't fix it. – StriplingWarrior Jul 6 '10 at 19:03

23 Answers

113

There are a lot of options:

```
int x = 30;
if (Enumerable.Range(1,100).Contains(x))
    //true

if (x >= 1 && x <= 100)
    //true
```

Also, check out this SO post for regex options.

edited May 23 '17 at 11:55

Community ♦
1 1

answered Jul 6 '10 at 17:41

Dustin Laine
32.2k 8 75 112

276 Enumerable.Range has to generate the enumerable of integers first, and then loop over each item to find it. That's a terrible idea and performance compared to checking a value is drastically different. I think we should adopt a moto, just because LINQ Extensions are cool, doesn't mean they should be used for everything. – Matthew Abbott Jul 6 '10 at

Adam Robinson Jul 6 '10 at 17:47

13 I agree this is a terrible idea performance-wise, but the OP wants something more fancy than an `if` statement. This certainly accomplishes that... ;) – Tim Coker Jul 6 '10 at 17:49

2 The point of this question wasn't to debate what way is faster during execution. It was what code looks nicer and still gets the job done (again: I don't care about performance); this one wins by far. Easy to read and simple to use. – Sergio Tapia Jul 6 '10 at 19:15

6 It's worth to note that the second parameter isn't "stop", but "count". So for instance, Enumerable.Range(150, 300).Contains(400) will return true. – Shathur Jul 1 '13 at 12:46

82

Do you mean?

```
if(number >= 1 && number <= 100)
```

or

```
bool TestRange (int numberToCheck, int bottom, int top)
{
  return (numberToCheck >= bottom && numberToCheck <= top)
}
```

edited Sep 24 '15 at 15:12

Liam Laverty

155 1 9

answered Jul 6 '10 at 17:32

kemiller2002

97.3k 23 177 234

Jul 6 '10 at 17:33

1 You don't need "is" in there... This won't compile. (Otherwise, I agree 100%) – Reed Copsey Jul 6 '10 at 17:34

Thanks Reed. I feel so stupid for not seeing that. – kemiller2002 Jul 6 '10 at 17:36

4 @Ben, just wait until I try and patent it too :) – kemiller2002 Jul 6 '10 at 17:38

I think this is the most solid solution but not that elegantly the questioner looking for, isn't ? – Kevin Simple May 27 '15 at 4:40

50

Just to add to the noise here, you could create an extension method:

```
public static bool IsWithin(this int value, int minimum, i
{
    return value >= minimum && value <= maximum;
}
```

Which would let you do something like...

```
int val = 15;

bool foo = val.IsWithin(5,20);
```

That being said, this seems like a silly thing to do when the check itself is only one line.

edited Oct 19 '15 at 23:17


151k 27 256 323

1 @Ben: I went on the subject, which says "within a range" (which I don't think is ambiguous in that regard), but you're right in that the question body says "between 1 and 100" (which is, of course, ambiguous). – Adam Robinson Jul 6 '10 at 17:44

42

As others said, use a simple if.

You should think about the ordering.

e.g

```
1 <= x && x <= 100
```

is easier to read than

```
x >= 1 && x <= 100
```

edited Mar 25 '16 at 16:04


Marshal
4,114 6 45 78

answered Jul 6 '10 at 17:53


Esben Skov Pedersen
3,376 2 23 39

17 "Easier" is in the eye of the beholder. I personally prefer to have the variable in question on the left and the constant or variable *not* in question on the right. – Adam Robinson Jul 6 '10 at 18:25

2 Number line order is the clearest initially - but you can train your eyes/mind for other orders. Specifically - I like the trick of placing the *constant* on the left, always. If you do that the compiler will tell you when you've typed `=` instead of `==`. It doesn't help with non-equality relational operators - but it is easy to get used to using it consistently. – davidbak Mar 22 '16 at 21:31

1 I just want to add that this solution is not useful in any case. Consider `x` is a complex function call or time-consuming Linq-expression. In this case you would do this twice which isn't a good thing. Sure you should store the value into a temporary local variable but there are some cases (e.g. in else-if-statements) where you only want to call the functions after other if's or else-if's failed. With temporary variables you have to call them anyway before. An extension method (mentioned in other answers) is the best solution imho in those cases. – Robert S. Jun 17 '16 at 21:39

3 I like number line order too, and also for the complement test, e.g. x < 10 || 20 < x. To me it shouts "x is outside the range 10 - 20". – William T. Mallard Jul 27 '16 at 8:12

35

You can reduce the number of comparisons from two to one by using some math. The idea is that one of the two factors becomes negative if the number lies outside of the range and zero if the number is equal to one of the bounds:

If the bounds are inclusive:

```
(x - 1) * (100 - x) >= 0
```

or

```
(x - min) * (max - x) >= 0
```

If the bounds are exclusive:

```
(x - 1) * (100 - x) > 0
```

or

```
(x - min) * (max - x) > 0
```

However, in production code I would simply write `1 < x && x < 100`, it is easier to understand.

edited Mar 6 '18 at 15:00

answered Sep 20 '13 at 17:03

Olivier Jacot-Descombes

71.2k 10 93 144

3 By my standards this by far the most elegant solution, interesting is that for me it also seems to run somewhat faster than checking the both of the expressions, that said it also seems more inconsistent (speed seems to vary more) would be interesting to see if there's any research done on which one is the faster. – Thomas Lindvall Mar 18 '15 at 19:34

3 Tested your solution on javascript and its accurate with floating-point numbers up to 14 decimals. It's a very good code snippet. It'd upvote you thrice if I could – rubbyrubber Dec 29 '15 at 23:32

3 Though, there is a minor issue if large positive numbers are involved, it can overflow! XD You might want to keep that in mind when writing your code. – BrainStorm.exe Feb 8 '16 at 19:56

1 The question asks for elegance and is therefore more of academic than of practical value. Personally I would just use a

1 For those concerned about performance, `1 < x & x < 100` (no && short circuit) instructs the compiler that it can always evaluate `x < 100` no matter the result of `1 < x`. Strangely (due to branch prediction) it is faster to always do this simple operation than it is to sometimes skip it. – Tom Leys Dec 7 '17 at 2:23

16

With a bit of extension method abuse, we can get the following "elegant" solution:

```
using System;

namespace Elegant {
    public class Range {
        public int Lower { get; set; }
        public int Upper { get; set; }
    }

    public static class Ext {
        public static Range To(this int lower, int upper)
            return new Range { Lower = lower, Upper = upper
        }

        public static bool In(this int n, Range r) {
            return n >= r.Lower && n <= r.Upper;
        }
    }

    class Program {
        static void Main() {
            int x = 55;
            if (x.In(1.To(100)))
                Console.WriteLine("it's in range! elegantly
        }
    }
}
```

answered Jul 6 '10 at 18:12

14

I propose this:

```
public static bool IsWithin<T>(this T value, T minimum, T
IComparable<T> {
    if (value.CompareTo(minimum) < 0)
       return false;
    if (value.CompareTo(maximum) > 0)
       return false;
    return true;
}
```

Examples:

```
45.IsWithin(32, 89)
true
87.2.IsWithin(87.1, 87.15)
false
87.2.IsWithin(87.1, 87.25)
true
```

and of course with variables:

```
myvalue.IsWithin(min, max)
```

It's easy to read (close to human language) and works with any comparable type (integer, double, custom types...).

Having code easy to read is important because the developer will not waste "brain cycles" to understand it. In long coding sessions wasted brain cycles make developer tired earlier and prone to bug.

answered Apr 5 '17 at 8:51

Anton M

396 3 11

having a boolean flag to determine if inclusive or not – Ben Apr 13 '17 at 16:58

Good. It is easy to understand. I changed the name `IsInRange`. `I'm not that keen on Ben's inclusive boolean as that requires a few more brain cycles. It has the advantage that it can be used in any class that that implements IComparer. This is in my Extensions now along with` LiesWithin / `LiesInside. Just can't decide which.` NotOutside would work but I don't like negative conditions – Paulustrious Oct 2 '17 at 19:03

7

If this is incidental, a simple `if` is all you need. If this happens in many places, you might want to consider these two:

- PostSharp. Decorate methods with attributes that 'inject' code into the method after compilation. I don't know for sure, but I can imagine it can be used for this.

Something like:

```
[Between("parameter", 0, 100)]
public void Foo(int parameter)
{
}
```

- Code contracts. Has the advantage that the constraints can be checked at compile time, by static verification of your code and the places that use your code.

answered Jul 6 '10 at 17:47

JulianR

+1 for code contracts; it's specific to validating a parameter, but it's a frequent use case and static verification has the potential to be extremely useful. – Dan Bryant Jul 6 '10 at 18:32

5

```
if (value > 1 && value < 100)
{
    // do work
}
else
{
    // handle outside of range logic
}
```

answered Jul 6 '10 at 17:33

Nick Larsen ♦

14.7k 6 55 85

5

Using an `&&` expression to join two comparisons is simply the most elegant way to do this. If you try using fancy extension methods and such, you run into the question of whether to include the upper bound, the lower bound, or both. Once you start adding additional variables or changing the extension names to indicate what is included, your code becomes longer and harder to read (for the vast majority of programmers). Furthermore, tools like Resharper will warn you if your comparison doesn't make sense (`number > 100 && number < 1`), which they won't do if you use a method ('i.IsBetween(100, 1)').

The only other comment I'd make is that if you're checking

```
Contract.Requires(number > 1 && number < 100)
```

This is more elegant than `if(...) throw new Exception(...)`, and you could even get compile-time warnings if someone tries to call your method without ensuring that the number is in bounds first.

answered Jul 6 '10 at 19:16

StriplingWarrior
110k 20 187 237

2 FYI, the contracts static analyzer is happier when the lower bound and upper bound constraints are split into separate Requires statements. – Dan Bryant Jul 6 '10 at 20:12

Thanks Dan Bryant, that is precisely what I was here looking for. Cannot find much material on suggestions on style of conditions for the Requires and other related Code Contract methods. – jpierson Sep 30 '13 at 1:11

2

If you want to write more code than a simple if, maybe you can: Create a Extension Method called IsBetween

```
public static class NumberExtensionMethods
{
    public static bool IsBetween(this long value, long Min
    {
        // return (value >= Min && value <= Max);
        if (value >= Min && value <= Max) return true;
        else return false;
    }
}
```

...

```
MessageBox.Show(MyNumber.IsBetween(1, 100).ToString());
```

Addendum: it's worth noting that in practice you very rarely "just check for equality" (or <, >) in a codebase. (Other than in the most trivial situations.) Purely as an example, any game programmer would use categories something like the following in every project, as a basic matter. Note that in this example it (happens to be) using a function (Mathf.Approximately) which is built in to that environment; in practice you typically have to carefully develop your own concepts of what comparisons means for computer representations of real numbers, for the type of situation you are engineering. (Don't even mention that if you're doing something like, perhaps a controller, a PID controller or the like, the whole issue becomes central and very difficult, it becomes the nature of the project.) BY no means is the OP question here a trivial or unimportant question.

```
private bool FloatLessThan(float a, float b)
    {
    if ( Mathf.Approximately(a,b) ) return false;
    if (a<b) return true;
    return false;
    }

private bool FloatLessThanZero(float a)
    {
    if ( Mathf.Approximately(a,0f) ) return false;
    if (a<0f) return true;
    return false;
    }

private bool FloatLessThanOrEqualToZero(float a)
    {
    if ( Mathf.Approximately(a,0f) ) return true;
    if (a<0f) return true;
    return false;
    }
```

Fattie
20.3k 31 212 460

answered Sep 20 '13 at 16:35

Tony
8,468 6 46 82

1 Replace the if and else with `return (value >= Min && value <= Max);` – AeroX Nov 27 '14 at 14:20

the elegant way to write the comparison is "in logical order..." if (Min <= value && value <= Max). That is much prettier. – Fattie Apr 1 '15 at 3:35

this is of course the correct solution. – Fattie Apr 1 '15 at 3:35

2 Further on this question, it's so surprising that nobody has mentioned the central issue in any real-world project (particularly if you're a game engineer) **is that you have to deal with the approximation issue**. In any real-world software you essentially never "just do a comparison" (whether equality or <, >) you have to consider and deal with the error issue, depending on the situation at hand. I've edited in an addendum to this answer (the only correct answer here!) since no more answers are allowed. – Fattie Apr 2 '15 at 4:32

Thank you for this observation and the addendum. – Tony Apr 2 '15 at 12:43

2

Cause all the other answer are not invented by me, here just my implementation:

```
public enum Range
{
    /// <summary>
    /// A range that contains all values greater than star
    /// </summary>
```

```
equal to end.
    /// </summary>
    Closed,
    /// <summary>
    /// A range that contains all values greater than or e
end.
    /// </summary>
    OpenClosed,
    /// <summary>
    /// A range that contains all values greater than star
end.
    /// </summary>
    ClosedOpen
}

public static class RangeExtensions
{
    /// <summary>
    /// Checks if a value is within a range that contains
and less than or equal to end.
    /// </summary>
    /// <param name="value">The value that should be check
    /// <param name="start">The first value of the range t
    /// <param name="end">The last value of the range to b
    /// <returns><c>True</c> if the value is greater than
to end, otherwise <c>false</c>.</returns>
    public static bool IsWithin<T>(this T value, T start,
IComparable<T>
    {
        return IsWithin(value, start, end, Range.ClosedOpe
    }

    /// <summary>
    /// Checks if a value is within the given range.
    /// </summary>
    /// <param name="value">The value that should be check
    /// <param name="start">The first value of the range t
    /// <param name="end">The last value of the range to b
    /// <param name="range">The kind of range that should
given kind of range the start end end value are either inc
    /// <returns><c>True</c> if the value is within the gi
<c>false</c>.</returns>
    public static bool IsWithin<T>(this T value, T start,
IComparable<T>
    {
        if (value == null)
```

```
            throw new ArgumentNullException(nameof(start))

        if (end == null)
            throw new ArgumentNullException(nameof(end));

        switch (range)
        {
            case Range.Open:
                return value.CompareTo(start) > 0
                       && value.CompareTo(end) < 0;
            case Range.Closed:
                return value.CompareTo(start) >= 0
                       && value.CompareTo(end) <= 0;
            case Range.OpenClosed:
                return value.CompareTo(start) > 0
                       && value.CompareTo(end) <= 0;
            case Range.ClosedOpen:
                return value.CompareTo(start) >= 0
                       && value.CompareTo(end) < 0;
            default:
                throw new ArgumentException($"Unknown para
nameof(range));
        }
    }
}
```

You can then use it like this:

```
var value = 5;
var start = 1;
var end = 10;

var result = value.IsWithin(start, end, Range.Closed);
```

answered Feb 7 '17 at 8:42

Oliver

33.8k 7 74 120

1

```
public bool IsWithinRange(int number, int topOfRange, int
includeBoundaries) {
    if (includeBoundaries)
        return number <= topOfRange && number >= bottomOfR
    return number < topOfRange && number > bottomOfRange;
}
```

edited Jul 6 '10 at 17:42

answered Jul 6 '10 at 17:33

Ben Hoffstein

91.1k 8 94 114

3 There are actually four cases, inclusive/inclusive, inclusive/exclusive, exclusive/inclusive and exclusive/exclusive. – William T. Mallard Mar 28 '14 at 16:44

1

In C, if time efficiency is crucial and integer overflows will wrap, one could do `if ((unsigned)(value-min) <= (max-min)) ...` . If 'max' and 'min' are independent variables, the extra subtraction for (max-min) will waste time, but if that expression can be precomputed at compile time, or if it can be computed once at run-time to test many numbers against the same range, the above expression may be computed efficiently even in the case where the value is within range (if a large fraction of values will be below the valid range, it may be faster to use `if ((value >= min) && (value <= max)) ...` because it will **exit early** if value is less than min).

Before using an implementation like that, though, benchmark one on the target machine. On some

independently whereas in the subtract-and-compare method the subtraction has to complete before the compare can execute.

edited Jul 23 '13 at 11:55

Sachin Chavan
3,682 5 42 71

answered Jul 6 '10 at 18:05

supercat
58.4k 4 117 156

1

How about something like this?

```
if (theNumber.isBetween(low, high, IntEx.Bounds.INCLUSIVE_
{
}
```

with the extension method as follows (tested):

```
public static class IntEx
{
    public enum Bounds
    {
        INCLUSIVE_INCLUSIVE,
        INCLUSIVE_EXCLUSIVE,
        EXCLUSIVE_INCLUSIVE,
        EXCLUSIVE_EXCLUSIVE
    }

    public static bool isBetween(this int theNumber, int l
    {
        bool result;
        switch (boundDef)
        {
            case Bounds.INCLUSIVE_INCLUSIVE:
                result = ((low <= theNumber) && (theNumber
                break;
```

```
            case Bounds.EXCLUSIVE_INCLUSIVE:
                result = ((low < theNumber) && (theNumber
                break;
            case Bounds.EXCLUSIVE_EXCLUSIVE:
                result = ((low < theNumber) && (theNumber
                break;
            default:
                throw new System.ArgumentException("Invali
argument");
        }
        return result;
    }
}
```

answered Mar 28 '14 at 16:52

William T. Mallard

789 2 16 23

I would do a Range object, something like this:

1

```
public class Range<T> where T : IComparable
{
    public T InferiorBoundary{get;private set;}
    public T SuperiorBoundary{get;private set;}

    public Range(T inferiorBoundary, T superiorBoundary)
    {
        InferiorBoundary = inferiorBoundary;
        SuperiorBoundary = superiorBoundary;
    }

    public bool IsWithinBoundaries(T value){
        return InferiorBoundary.CompareTo(value) > 0 &&
SuperiorBoundary.CompareTo(value) < 0;
    }
}
```

Then you use it this way:

```
Range<int> myRange = new Range<int>(1,999);
bool isWithinRange = myRange.IsWithinBoundaries(3);
```

That way you can reuse it for another type.

edited May 19 '16 at 23:17

Peter Mortensen
14.1k 19 88 114

answered Sep 20 '13 at 15:57

IEatBagels
725 7 18

Your `Range` object needs to use the `CompareTo` method to compare items, not the `<` operator. – Servy Sep 20 '13 at 16:06

You're right, though if implementing IComparable you should also override operators (at least that's what my VS code analysis is saying), meaning < would work. Although I might be wrong, I don't have much experience and this is my first answer on SO – IEatBagels Sep 20 '13 at 17:27

No, your compiler *won't* say that this works. This won't compile. It is entirely reasonable for an object to implement `IComparable` and not overload the `<` operator. – Servy Sep 20 '13 at 17:28

I edited, sorry about the mistake indeed it doesn't compile – IEatBagels Sep 20 '13 at 17:34

1

```
static class ExtensionMethods
{
    internal static bool IsBetween(this double number,doub
    {
```

```
    internal static bool IsBetween(this int number, double
    {
        return Math.Min(bound1, bound2) <= number && numbe
    }
}
```

Usage

double numberToBeChecked = 7;

var result = numberToBeChecked.IsBetween(100,122);

var result = 5.IsBetween(100,120);

var result = 8.0.IsBetween(1.2,9.6);

edited Mar 30 at 13:00

answered Mar 30 at 9:58

İBRAHİM GAZALOĞLU
11 1 3

I'd go with the more simple version:

0

```
if(Enumerable.Range(1,100).Contains(intInQuestion)) { ...D
```

edited Nov 8 '13 at 8:43

answered Nov 8 '13 at 8:33

cseder
144 8

wrong answer to the question, but with this method you can use the static Enumerable.Range(x,y) to check for matches in an elegant way. – cseder Nov 8 '13 at 8:45

Terrible answer. Surprising behavior. Enumerable.Range(2,100) gives range of (2, 101) instead of range (2, 100). It requires the allocation and looping of potentially 100 ints. Additionally the Contains method on Enumberable.Range will ensure that they are actually looped over. – Ryan The Leach Apr 24 '18 at 7:43

0

I was looking for an elegant way to do it where the bounds might be switched (ie. not sure which order the values are in).

This will only work on newer versions of C# where the ?: exists

```
bool ValueWithinBounds(float val, float bounds1, float bou
{
    return bounds1 >= bounds2 ?
      val <= bounds1 && val >= bounds2 :
      val <= bounds2 && val >= bounds1;
}
```

Obviously you could change the = signs in there for your purposes. Could get fancy with type casting too. I just needed a float return within bounds (or equal to)

answered May 13 '17 at 13:17

Kalikovision
31 5

0

equal mean? In general you should wrap all floating point numbers in what is called a 'epsilon ball' this is done by picking some small value and saying if two values are this close they are the same thing.

```
    private double _epsilon = 10E-9;
    /// <summary>
    /// Checks if the distance between two doubles is with
    /// In general this should be used for determining equ
    /// </summary>
    /// <param name="x0">The orgin of intrest</param>
    /// <param name="x"> The point of intrest</param>
    /// <param name="epsilon">The minimum distance between
    /// <returns>Returns true iff x  in (x0-epsilon, x0+eps
    public static bool IsInNeghborhood(double x0, double x
x) < epsilon;

    public static bool AreEqual(double v0, double v1) => Is
_epsilon);
```

With these two helpers in place and assuming that if any number can be cast as a double without the required accuracy. All you will need now is an enum and another method

```
    public enum BoundType
    {
        Open,
        Closed,
        OpenClosed,
        ClosedOpen
    }
```

The other method follows:

```
    public static bool InRange(double value, double upperB
BoundType bound = BoundType.Open)
    {
        bool inside = value < upperBound && value > lowerB
        switch (bound)
```

```
            case BoundType.Closed:
                return inside || AreEqual(value, upperBoun
lowerBound);
            case BoundType.OpenClosed:
                return inside || AreEqual(value, upperBoun
            case BoundType.ClosedOpen:
                return inside || AreEqual(value, lowerBoun
            default:
                throw new System.NotImplementedException("
        }
    }
```

Now this may be far more than what you wanted, but it keeps you from dealing with rounding all the time and trying to remember if a value has been rounded and to what place. If you need to you can easily extend this to work with any epsilon and to allow your epsilon to change.

answered Sep 22 '17 at 15:21

rahicks
196 1 11

0

Elegant because it doesn't require you to determine which of the two boundary values is greater first. It also contains no branches.

```
public static bool InRange(float val, float a, float b)
{
    // Determine if val lies between a and b without first
b)
    return ( a <= val & val < b ) | ( b <= val & val < a )
}
```

answered Dec 7 '17 at 2:27

Tom Leys
15.4k 6 35 59

0

I don't know but i use this method:

```
    public static Boolean isInRange(this Decimal dec, Deci
includesMin = true, bool includesMax = true ) {

    return (includesMin ? (dec >= min) : (dec > min)) && (
(dec < max));
}
```

And this is the way I can use it:

```
    [TestMethod]
    public void IsIntoTheRange()
    {
        decimal dec = 54;

        Boolean result = false;

        result = dec.isInRange(50, 60); //result = True
        Assert.IsTrue(result);

        result = dec.isInRange(55, 60); //result = False
        Assert.IsFalse(result);

        result = dec.isInRange(54, 60); //result = True
        Assert.IsTrue(result);

        result = dec.isInRange(54, 60, false); //result =
        Assert.IsFalse(result);

        result = dec.isInRange(32, 54, false, false);//res
        Assert.IsFalse(result);

        result = dec.isInRange(32, 54, false);//result = T
        Assert.IsTrue(result);
    }
```

answered Oct 8 '18 at 16:55

user8790965
1 1

Please provide an example of usage below the code block, this will help OP know if it fits his purpose – Gabriel Balsa Cantú Oct 8 '18 at 17:10

-1

You are looking for `in [1..100]` ? That's only Pascal.

answered Mar 7 at 16:46

Polluks
124 1 10

1 Not true, its not only Pascal. Many modern languages have features like this. In Kotlin, for example it is called "Pattern Matching". Example `when (number) { in 0..9 -> println("1 digit") in 10..99 -> println("2 digits") in 100..999 -> println("3 digits") }` – this.myself Mar 13 at 13:16