SAR
DOT
NET

Syntactic sugar of C#: language improvements in latest versions

by Dmitry Vereskun

SEP 30, 2017

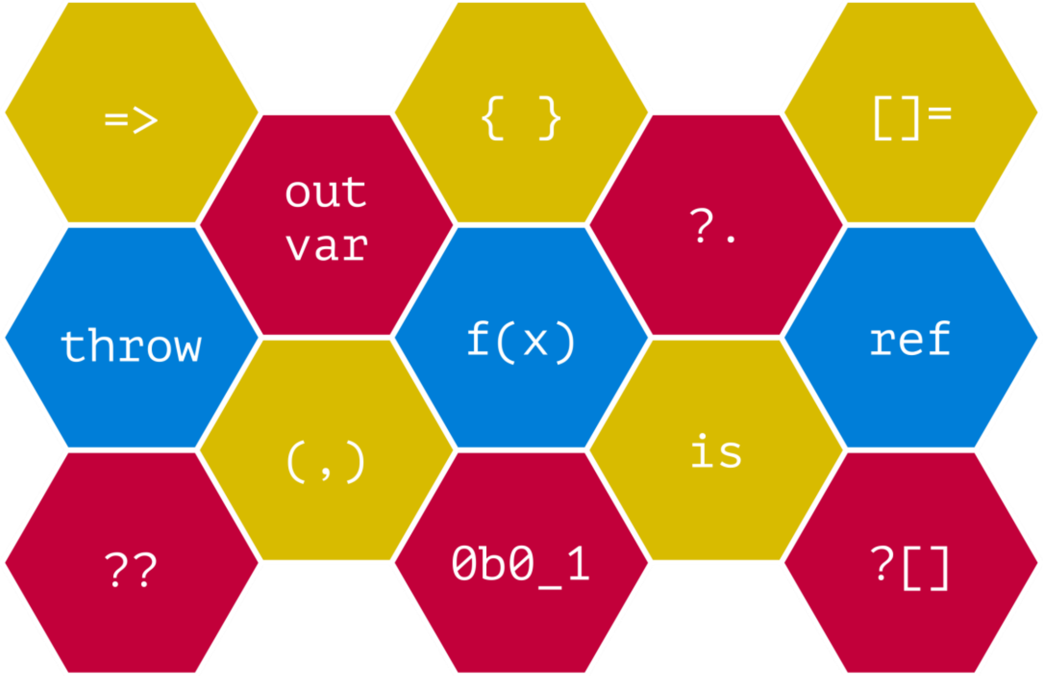
AGENDA

- Properties enhancements
- Lambda expressions
- Initializers
- Number literals
- Inline variables
- Null expressions
- Throw expression
- Local functions
- Ref locals and ref returns
- Value tuples
- Pattern matching
- Async improvements
- Default keyword

6 C# 6.0 (VS 2015)

7 C# 7.0 (VS 2017)

7.1 C# 7.1 (VS 2017.3)



PROPERTIES

```
private string _value;

public string GetValue()
{
    return _value;
}

public void SetValue(string value)
{
    _value = value;
}
```

```
private string _value;

public string Value
{
    get
    {
        return _value;
    }
    set
    {
        _value = value;
    }
}
```

AUTO-PROPERTIES

```
private string _value;

public string Value
{
    get
    {
        return _value;
    }
    set
    {
        _value = value;
    }
}
```

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

or

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

GETTER-ONLY PROPERTIES AND METHODS

```
private string _value;

public string DoubleValue
{
    get
    {
        return _value * 2;
    }
}
```

```
private string _value;

public string DoubleValue => _value * 2;
```

```
public int GetResult()
{
    return evaluate_result();
}
```

```
public int GetResult()
    => evaluate_result();
```

LAMBDAS EVERYWHERE

```
private string _value;

public string Value
{
    get
    {
        return _value;
    }
    set
    {
        _value = value;
    }
}
```

```
private string _value;

public string Value
{
    get => _value;
    set => _value = value;
}
```

```
public MyClass(int n)
{
    _n = n;
}
```

```
public MyClass(int n) => _n = n;
```

AUTO-PROPERTY INITIALIZERS AND READ-ONLY PROPERTIES

6

```
private string _value = "N/A";

public string Value
{
    get
    {
        return _value;
    }
    set
    {
         _value = value;
    }
}
```

```
public string Value { get; set; } = "N/A";
```

```
private readonly int _onlyFive = 5;

public int OnlyFive
{
    get
    {
        return _onlyFive;
    }
}
```

```
public int OnlyFive { get; } = 5;
```

SINGLETON IN 2 (OR 3) LINES OF CODE

6

```
public class Singleton
{
    private Singleton() { }

    public static Singleton Instance { get; } = new Singleton();
}
```

```
public class Singleton
{
    private Singleton() { }

    static Singleton() => Instance = new Singleton();

    public static Singleton Instance { get; }
}
```

DICTIONARY INITIALIZERS

6

- Classic

```
var capitals = new Dictionary<string, string>
{
    { "Russia", "Moscow" }, // call Add
    { "Belarus", "Minsk" },
    { "Ukraine", "Kyiv" },
    { "USA", "Washington, D.C." },
};
```

- New

```
var capitals = new Dictionary<string, string>
{
    ["Russia"]  = "Moscow", // use indexer
    ["Belarus"] = "Minsk",
    ["Ukraine"] = "Kyiv",
    ["USA"]     = "Washington, D.C.",
};
```

NUMBER LITERALS

- Type literals:
  - float (**2.3f**)
  - double (**4d**)
  - decimal (**12.6m**)
  - long (**25L**)
  - uint (1000**u**)
  - ulong (13**ul**)

- Integer literals:
  - Decimal (**123**)
  - Hexadecimal (**0x7b**)

- What's new?
  - Binary integer literal (**0b01111011**)
  - Digit group separator: _

    Works with any number literals (**100_000**)

```
int maxValue  = 2_147_483_647;
byte mask     = 0b0111_10011;
```

INLINE VARIABLES

- Before:

```
int intValue;
if (int.TryParse(str, out intValue))
{
    // some actions with intValue
}
```

- After:

```
if (int.TryParse(str, out int intValue))
{
    // some actions with intValue
}
```

or

```
if (int.TryParse(str, out var intValue))
{
    // some actions with intValue
}
```

NULL EXPRESSIONS » NULL COALESCING OPERATOR ??

- Null coalescing operator can be applied for any nullable types

```
string s = GetValue();
if (s == null) s = "Default value";
```

```
string s = GetValue() ?? "Default value";
```

```
int? parentId = reader["parentId"] as int?;

int pId = parentId.HasValue ? parentId.Value : -1;
```

```
int pId = (reader["parentId"] as int?) ?? -1;
```

NULL EXPRESSIONS » NULL-CONDITIONAL OPERATOR ?.

- So called Safe navigation operator or Elvis operator
- If left operand is not null, the operator acts as a regular dot operator
- If left operand is null, it returns a default value for expected type

Type of foo.bar	Type of foo?.bar
void	void
Nullable type (incl. reference types)	Same nullable type
Non-nullable type (structs, enums)	Nullable<T> wrapper

SAFE DELEGATE INVOCATION WITH NULL-CONDITIONAL OPERATOR

```
protected virtual void OnComplete()
{
    EventHandler complete = Complete;
    if (complete != null)
    {
        complete(this, EventArgs.Empty);
    }
}
```

```
protected virtual void OnComplete()
    => Complete?.Invoke(this, EventArgs.Empty);
```

NULL EXPRESSIONS » NULL-CONDITIONAL OPERATOR ?.

```
if (user != null)
{
    user.Show();
}

int? age = (user != null)
    ? user.Age
    : default(int?);

string name = (user != null && user.Name != null)
    ? user.Name
    : "Guest";
```

```
user?.Show();
int? age = user?.Age;
string name = user?.Name ?? "Guest";
```

NULL EXPRESSIONS » NULL-CONDITIONAL OPERATOR ?[]

6

```
User[] users = GetUsers();
if (users != null)
{
    Show(users[0].Name);
}
else
{
    Show("no users");
}
```

```
User[] users = GetUsers();
Show(users?[0].Name ?? "no users");
```

```
User[] users = GetUsers();
if (users != null && users.Length > 0)
{
    Show(users[0].Name);
}
else
{
    Show("no users");
}
```

```
User[] users = GetUsers();
Show(users?.FirstOrDefault()?.Name);
```

THROW EXPRESSION

- Throw can be placed in any place instead of ordinary expression
- Best to union it with ?: or ?? operators

```
if (name == null)
{
    throw new ArgumentNullException(nameof(name));
}

this.name = name;
```

⇩

```
this.name = name ?? throw new ArgumentNullException(nameof(name));
```

```
string first = args?.Length > 0
    ? args[0]
    : throw new ArgumentException("Array is null or empty", nameof(args));
```

LOCAL FUNCTIONS

```
public static void QuickSort(int[] arr)
{
    if (arr == null) throw new ArgumentNullException(nameof(arr));
    if (arr.Length == 0) return;

    SortSegment(0, arr.Length - 1);

    void SortSegment(int from, int to)
    { … }

    void SwapAndMove(ref int i, ref int j)
    { … }
}
```

```
void SortSegment(int from, int to)
{
    int mid = arr[from + (to - from) / 2],
        i = from, j = to;

    while (i <= j)
    {
        while (arr[i] < mid) i++;
        while (arr[j] > mid) j--;
        if (i <= j) SwapAndMove(ref i, ref j);
    }

    if (i < to) SortSegment(i, to);
    if (from < j) SortSegment(from, j);
}
```

```
void SwapAndMove(ref int i, ref int j)
{
     int temp = arr[i];
     arr[i++] = arr[j];
     arr[j--] = temp;
}
```

Complete QuickSort with local functions

```csharp
public static void QuickSort(int[] arr)
{
    if (arr == null) throw new ArgumentNullException(nameof(arr));
    if (arr.Length == 0) return;

    SortSegment(0, arr.Length - 1);

    void SortSegment(int from, int to)
    {
        int mid = arr[from + (to - from) / 2], i = from, j = to;

        while (i <= j)
        {
            while (arr[i] < mid) i++;
            while (arr[j] > mid) j--;

            if (i <= j) SwapAndMove(ref i, ref j);
        }

        if (i < to) SortSegment(i, to);
        if (from < j) SortSegment(from, j);
    }

    void SwapAndMove(ref int i, ref int j)
    {
        int temp = arr[i];
        arr[i++] = arr[j];
        arr[j--] = temp;
    }
}
```

REF LOCALS AND REF RETURNS

```
private static ref int Max(ref int x, ref int y)
{
    if (x > y)
    {
        return ref x;
    }
    else
    {
        return ref y;
    }
}


int a = 123;
int b = 456;

Max(ref a, ref b) += 100;
Console.WriteLine(b); // 556!
```

REF LOCALS AND REF RETURNS

7

```
private static ref int Max(int[] array)
{
    int max = 0;
    for (int i = 1; i < array.Length; i++)
    {
        if (array[i] > array[max]) max = i;
    }

    return ref array[max];
}



int[] arr = { 3, 1, 4, 1, 5, 9, 2, 6, 5 };
Max(arr) = 0;
```

VALUE TUPLES

```
private static (int min, int max) MinAndMax(int[] array)
{
    int min = array[0], max = min;

    for (int i = 1; i < array.Length; i++)
    {
        if (array[i] < min) min = array[i];
        if (array[i] > max) max = array[i];
    }

    return (min, max);
}

int[] arr = { 1, 4, 3, 6, 5, 8, 9 };
var minMax                 = MinAndMax(arr); // minMax.min and minMax.max
(int min, int max) minMax  = MinAndMax(arr); // same as previous
(int, int) minMax          = MinAndMax(arr); // minMax.Item1 and minMax.Item2
(int min, int max)         = MinAndMax(arr); // min and max – deconstruction
(int min, _)               = MinAndMax(arr); // min only
```

TUPLE ELEMENT NAMES INFERENCE

7.1

```
int count = 5;
string label = "Colors used in the map";

var pair = (count: count, label: label);
// old good C# 7.0. You have to define properties’ names
```

```
int count = 5;
string label = "Colors used in the map";

var pair = (count, label);
// element names are "count" and "label“. Inferred in C# 7.1
```

VALUE TUPLE DECONSTRUCTION

7

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public void Deconstruct(out int x, out int y)
    {
        x = X;
        y = Y;
    }
}
```

```
Point p = GetPoint();
(int x, int y) = p; // deconstruction
```

ORDINARY CLASS, NOTHING STRANGE... OH, WAIT~

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public static bool operator ==(Point p1, Point p2)
        => p1.X == p2.X && p1.Y == p2.Y;

    public static bool operator !=(Point p1, Point p2)
        => !(p1 == p2);

    public override bool Equals(object obj)
    {
        Point p = obj as Point;
        return (p != null && this == p);
    }

    public override int GetHashCode() => X ^ Y;
}
```

ORDINARY CLASS, NOTHING STRANGE… OH, WAIT~

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public static bool operator ==(Point p1, Point p2)
        => p1.X == p2.X && p1.Y == p2.Y;

    public static bool operator !=(Point p1, Point p2)
        => !(p1 == p2);

    public override bool Equals(object obj)
    {
        Point p = obj as Point;
        return (p != null && this == p);
    }

    public override int GetHashCode() => X ^ Y;
}
```

PATTERN MATCHING » NULL TEMPLATE

7

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public static bool operator ==(Point p1, Point p2)
        => p1.X == p2.X && p1.Y == p2.Y;

    public static bool operator !=(Point p1, Point p2)
        => !(p1 == p2);

    public override bool Equals(object obj)
    {
        Point p = obj as Point;
        return !(p is null) && (this == p);
    }

    public override int GetHashCode() => X ^ Y;
}
```

PATTERN MATCHING » TYPE TEMPLATE

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public static bool operator ==(Point p1, Point p2)
        => p1.X == p2.X && p1.Y == p2.Y;

    public static bool operator !=(Point p1, Point p2)
        => !(p1 == p2);

    public override bool Equals(object obj)
    {
        return (obj is Point p) && (this == p);
    }

    public override int GetHashCode() => X ^ Y;
}
```

PATTERN MATCHING

```
if (shape is null)
{
    // show NULL error
}
else if (shape is Rectangle r)
{
    // work with Rectangle r
}
else if (shape is Circle c)
{
    // work with Circle c
}
```

PATTERN MATCHING

```
public static void SwitchPattern(object obj)
{
    switch (obj)
    {
        case null:
            Console.WriteLine("Constant pattern"); break;

        case Person p when p.FirstName == "Dmitry":
            Console.WriteLine("Person Dmitry"); break;

        case Person p:
            Console.WriteLine($"Other person {p.FirstName}, not Dmitry"); break;

        case var x when x.GetType().IsGeneric:
            Console.WriteLine($"Var pattern with generic type {x.GetType().Name}"); break;

        case var x:
            Console.WriteLine($"Var pattern with the type {x.GetType().Name}"); break;
    }
}
```

GENERALIZED ASYNC RETURN

```
public async ValueTask<int> TakeFiveSlowly()
{
    await Task.Delay(100);
    return 5;
}
```

Now async methods can return ANY type with “async pattern” — GetAwaiter()

ASYNC MAIN

7.1

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

⬇

```
static async Task<int> Main()
{
    return await DoAsyncWork();
}

static async Task Main()
{
    await DoAsyncWork();
}
```

DEFAULT KEYWORD

7.1

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

⬇

```
Func<string, bool> whereClause = default;
```

```
void SomeMethod(int? arg = default)
{
    // ...
}
```

WHAT ARE WE WAITING FOR?

- C# 7.2
  - Read-only references and structs
  - Blittable types
  - Ref-like types (stack only)
  - Non-trailing named arguments
  - Private protected access modifier
- C# 8
  - Nullable reference types
  - Default interface methods (who says Java?)
  - Async streams
  - Extension everything

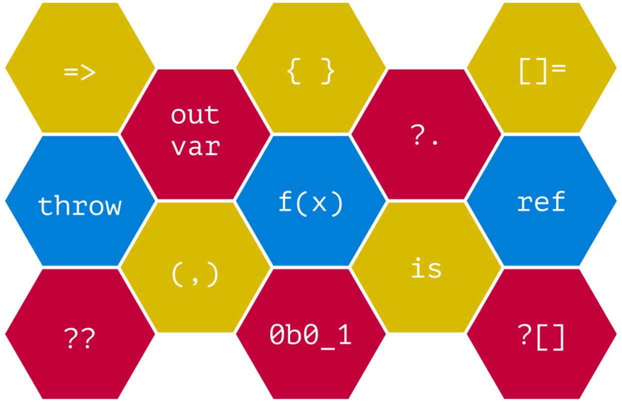
https://channel9.msdn.com/Blogs/Seth-Juarez/A-Preview-of-C-8-with-Mads-Torgersen

Thanks for attention!

Questions? Suggestions?

Author: Dmitry Vereskun
ROKO Labs, Saratov, Russia

Telegram: d_vereskun



=>
{ }
[]=
out var
?.
throw
f(x)
ref
(,)
is
??
0b0_1
?[]