Обзор C# 6.0

.NEXT 2014 Moscow

C# 1.0

- Объектно-ориентированный язык
- Строгая типизация

C# 2.0

- Обобщенные типы
- Блоки итераторов
- Типы, допускающие значения null
- Анонимные методы
- Статические классы
- Разделяемые типы

C# 3.0

- Неявно типизированные локальные переменные
- Инициализаторы объектов и коллекций
- Автоматически реализуемые свойства
- Анонимные типы
- Методы расширения
- Лямбда-выражения
- Деревья выражений
- LINQ

C# 4.0

- Динамическая типизация (dynamic)
- Именованные аргументы
- Необязательные параметры
- Обобщенная ковариантность и контравариантность

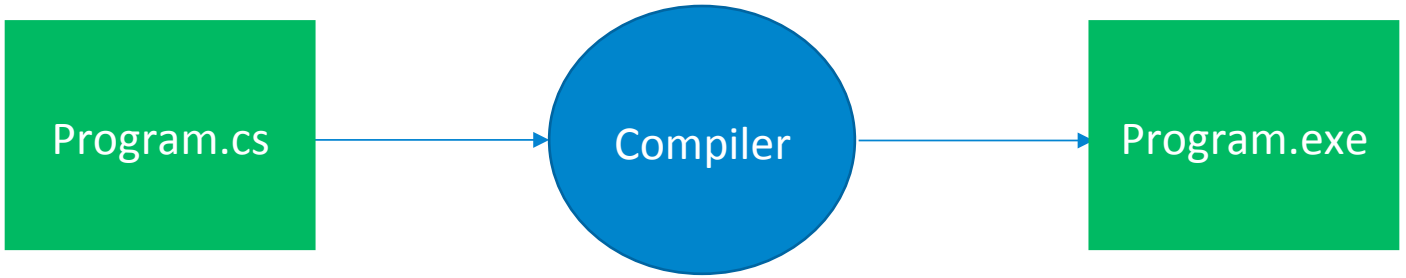
C# 5.0

- Асинхронные методы (async/await)

Эволюция C#


C# 6.0
C# 5.0
C# 4.0
C# 3.0
C# 2.0
C# 1.0
?
Асинхронное программирование
Динамическая типизация
LINQ
Обобщенные типы
Управляемый код

Проблемы компилятора на сегодняшний день



- Черный ящик
- Написан на неуправляемом C++
- Сложен в расширении и поддержке
- Бремя поддержки интегрированных в VS инструментов (редакторы кода, парсеры)

Roslyn как спасение

- Roslyn – платформа компиляторов .NET
  - Полностью переписанные компиляторы C# и VB.NET
  - Написаны на C# и VB.NET соответственно
  - Compiler as service: интерфейс API для анализа кода
  - Прост в расширении и поддержке
  - Open Source

C# 6.0

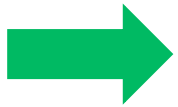
- **Roslyn**
- Getter-only auto-properties
- Auto-Property Initializers
- Using static classes
- String interpolation
- Expression-bodied members
- Index initializers
- Null-propagation operator
- Nameof operator
- Exception filters
- Await in catch and finally

Пример кода

```
public class Point
  {
     private readonly int _x;
     private readonly int _y;
     public int X { get { return _x; } }
     public int Y { get { return _y; } }
     public Point(int x, int y) { x = _x; y = _y; }
     public double Distance
     {
        get { return Math.Sqrt(X * X + Y * Y); }
     }
     public override string ToString()
     {
        return string.Format("({0}, {1})", X, Y);
     }
  }
```

Getter-only auto-properties

```
public class Point
  {
     public int X { get; }
     public int Y { get; }
     public Point(int x, int y)
     {
        X = x;
        Y = y;
     }
     ...
  }
```



```
public class Point
  {
     private readonly int <X>k__BackingField;
     private readonly int <Y>k__BackingField;
     public Point(int x, int y)
     {
        base..ctor();
        this.X = x;
        this.Y = y;
     }
     public int get_X()
     {
        return this.<X>k__BackingField;
     }
     public int get_Y()
     {
        return this.<Y>k__BackingField;
     }
}
```

- Упрощают создание неизменяемых типов
- Подобно ***readonly*** полям, инициализация возможна как при объявлении свойства, так и из конструктора

Auto-property initializers

```
public class Point
  {
     public int X { get; } = 17;
     public int Y { get; } = 17;
     ...
  }
```

```
public class Point
  {
     private readonly int <X>k__BackingField;
     private readonly int <Y>k__BackingField;

     public Point()
     {
        this.<X>k__BackingField = 17;
        this.<Y>k__BackingField = 17;
     }
     ...
  }
```

- Выполняются в том порядке в котором определены
- Инициализируют нижележащие поля
- Не могут ссылаться на this

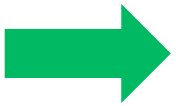
Using static classes

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

     ...

     public double Distance
     {
        get { return Math.Sqrt(X * X + Y * Y); }
     }
  }
```

```
using System.Math;

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

     ...

     public double Distance
     {
        get { return Sqrt(X * X + Y * Y); }
     }
  }
```

String interpolation

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

     public override string ToString()
     {
        return string.Format("({0}, {1})", X,
Y);
     }
  }
```



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

     public override string ToString()
     {
        return $"({X}, {Y})";
     }
  }
```

- Задание формата

```
$"Name = {myName}, hours = {DateTime.Now:hh}"
```

- Текущий синтаксис немного отличается от финальной версии

```
"Name = \{myName}, hours = \{DateTime.Now:hh}"
```

Expression-bodied members

- Лямбда-выражения имеют поддержку коротких (однострочных) методов

```
(string text) => { return
text.Length; };
```

```
(string text) => text.Length;
```

```
(text) => text.Length;
```

```
text => text.Length;
```

Почему бы тоже самое не сделать для обычных методов?

Expression-bodied members

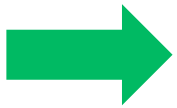
```
public double Distance
{
   get { return Sqrt(X * X + Y * Y); }
}
public override string ToString()
{
   return string.Format("({0}, {1})", X, Y);
}
```

```
public double Distance => Sqrt(X * X + Y * Y);

public override string ToString() => $"({X}, {Y})";
```

- Особенно удобны при объявлении getter-only свойств

Index initializers

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

     public JObject ToJson()
     {
        var json = new JObject();
        json["X"] = X;
        json["Y"] = Y;
        return json;
     }
  }
```



```
public JObject ToJson()
{
    var json = new JObject() { ["X"] = X, ["Y"] = Y };
    return json;
}
```

```
public JObject ToJson() => new JObject()
                                { ["X"] = X, ["Y"] = Y };
```

- 2 способа инициализации словаря

```
var dict = new Dictionary<string, int>()
               {
                  {"C#",  6},
                  {"C++",  14}
               };
```

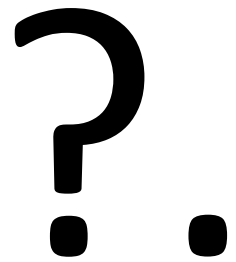
```
var dict = new Dictionary<string, int>()
               {
                  ["C#"] = 6,
                  ["C++"] = 14,
               };
```

Null-propagation operator

```
var cust = GetCustomer();
if (cust != null &&
    cust.Address != null &&
    cust.Address.PostCode != null)
return cust.Address.PostCode.ToString();
return "unknown";
```

Основная идея нового оператора – избавиться от множества проверок на равенство null

- Тернарный оператор ? : (C# 1.0)
- Объединяющий null оператор ?? (C# 2.0)



Null-propagation operator

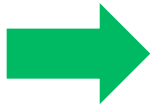
- Семантика оператора, такая же как у оператора ? : за исключением того, что выражение вычисляется только один раз

```
e?.x       =>   ((e == null) ? null : e.x)
e?.m(…)    =>   ((e == null) ? null : e.m(…))
e?[…]      =>   ((e == null) ? null : e[…])
```

- Оператор **?.** использует временные переменные для потокобезопасности
- Оператор **?.** является право ассоциативным
- Оператор **?.** имеет оптимизации при работе со значимыми типами

Null-propagation operator

```
var cust = GetCustomer();
if (cust != null &&
    cust.Address != null &&
    cust.Address.PostCode != null)
return cust.Address.PostCode.ToString();
return "unknown";
```



```
var postCode = cust?.Address?.PostCode ?? "unknown";
```

```
var cust = GetCustomer();
string arg_1F_0;
if (cust == null)
{
  arg_1F_0 = null;
}
else
{
  Address expr_13 = cust.Address;
  arg_1F_0 = ((expr_13 != null) ?
                  expr_13.PostCode : null);
}
string arg_28_0 = arg_1F_0 ?? "unknown";
```

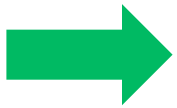
- Оператор **?.** vs значимые типы

```
int? postCodeLength = cust?.Address?.PostCode?.Length;
int? postCodeLength = cust?.Address?.PostCode.Length;
```

Null-propagation operator при вызове событий

Потокобезопасный вызов события до появления оператора

```
var onChanged = onChanged;
if (onChanged != null)
{
     onChanged(this, args);
}
```

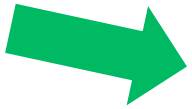


Потокобезопасный вызов события с использованием оператора ?.

```
onChanged?.Invoke(this, args);
```

Null-propagation operator в действии

```
public Point FromJson(JObject json)
  {
    if (json != null &&
        json["X"] != null &&
        json["X"].Type == JTokenType.Integer &&
        json["Y"] != null &&
        json["Y"].Type == JTokenType.Integer)
        {
          return new Point((int)json["X"], (int)json["Y"]);
        }
    return null;
 }
```

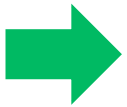


```
public Point FromJson(JObject json)
{
   if (json?["X"]?.Type == JTokenType.Integer &&
       json?["Y"]?.Type == JTokenType.Integer)
       {
         return new Point((int)json["X"], (int)json["Y"]);
       }
   return null;
}
```

NameOf operator

- Валидация параметров метода

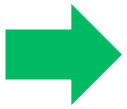
```
if (json == null)
    throw new
ArgumentNullException("json");
```



```
if (json == null)
    throw new ArgumentNullException(nameof(json));
```

- Реализация интерфейса INotifyPropertyChanged

```
private string currentTime;
public string CurrentTime
{
   get { return this.currentTime; }
   set
   {
     this.currentTime = value;
     this.OnPropertyChanged("CurrentTime");
   }
}
```



```
private string currentTime;
public string CurrentTime
{
   get { return this.currentTime; }
   set
   {
     this.currentTime = value;
     this.OnPropertyChanged(nameof(CurrentTime));
   }
}
```

Exception filters

```
try
  {
    throw new CustomException { Severity = 100 };
  }
catch (CustomException ex) if (ex.Severity > 70)
  {
    Console.WriteLine("Error");
  }
catch (CustomException ex)
  {
    Console.WriteLine("Warning");
  }
```

- Фильтры исключений можно использовать для совершения побочного действия, например, логирования

- VB.NET и F# их имеет
- Теперь и C# их имеет

```
private static bool Log(Exception e)
{
  /* log it */ ;
  return false;
}
```

```
try { }
catch (Exception e) if (Log(e)) { }
```

Await in catch and finally

```
try
 {
  ...
 }
catch (ConfigurationException ex) if (ex.Severity > 70)
 {
  await LogAsync(ex);
 }
finally
 {
  await CloseAsync();
 }
```

```
try
{
  yield return 1;
  yield return 2;
}
finally
{
  yield return 3;
}
```

In C# 5.0 we don’t allow the await keyword in catch and finally blocks, because we'd somehow convinced ourselves that it wasn’t possible to implement. Now we've figured it out, so apparently it wasn't impossible after all.

Небольшие изменения C# 6.0

- Конструкторы по умолчанию у структур

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

   public Point()
    {
     X = 1;
     Y = 2;
    }
 }
```

Будет ли вызван конструктор при создании массива? списка?

```
var array = new Point[100];
var list = new List<Point>(100);
```

Небольшие изменения C# 6.0

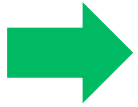
- Метод расширения Add для инициализации коллекций

```
//Invalid
LinkedList<int> list = new LinkedList<int>();
list.Add(1);
```

```
//Valid
ICollection<int> list = new LinkedList<int>();
list.Add(1);
```

```
//Invalid
LinkedList<int> list = new LinkedList<int> { 10, 20 };
```

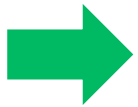
```
public static class Extensions
{
  public static void Add<T>(this ICollection<T> source, T item)
   {
     source.Add(item);
   }
}
```



```
//Valid
LinkedList<int> list = new LinkedList<int> { 10, 20 };
```

C# 5.0 VS C# 6.0

```
public class Point
    {
        private readonly int _x;
        private readonly int _y;

        public int X { get { return _x; } }
        public int Y { get { return _y; } }

        public Point(int x, int y) { x = _x; y = _y; }

        public double Distance
        {
            get { return Math.Sqrt(X * X + Y * Y); }
        }

        public override string ToString()
        {
            return string.Format("({0}, {1})", X, Y);
        }
    }
```



```
using System.Math;

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

        public Point(int x, int y) { X = x; Y = y; }

        public double Distance => Sqrt(X * X + Y * Y);

        public override string ToString() => $"({X}, {Y})";
    }
```

Что не попало в C# 6.0

- Primary constructors
- Declaration expression
- Param for enumerable
- Constructor type parameter inference
- Binary literals and digit separators
- Semicolon operator

```
public class Point(int x, int y)
{
  public int X { get; } = x;
  public int Y { get; } = y;
}

public int Sum(params IEnumerable<int> numbers)
{
  return numbers.Sum();
}

var tuple = new Tuple<string, int>("Tom",
22);
var tuple = 0b0021_1010;
var tuple = 0xFF_AF_02_AB;
var y = (var x = 4; WriteLine(x); x*x + 1); // 17
```

```
if (int.TryParse(input, out var result))
{
  return result;
}
return 0; // result is out of scope
var tuple = Tuple.Create("Tom", 22);
var tuple = new Tuple("Tom", 22);
```

В заключение

- Roslyn – революция в мире компиляторов
- Что с фичами не попавшими в C# 6.0?
- C# 7.0 - Pattern matching?

C# 6.0 = Roslyn + many small features

C# 6.0

- **Roslyn**

Syntactic sugar

- String interpolation
- Expression-bodied members
- Nameof operator
- Null-propagation operator
- Using static classes

Unfinished work

- Getter-only auto-properties
- Auto-Property Initializers
- Await in catch and finally
- Index initializers

IL C#

- Exception filters
- Parameterless constructors in structs