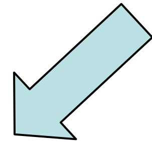
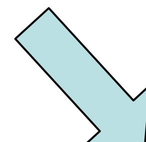
Основные конструкции языка C#

.NET Framework

CLR + библиотка классов .NET

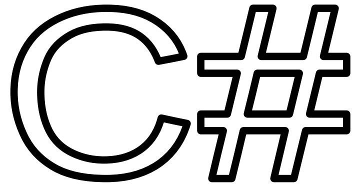
CLR – Common language runtime - общеязыковая исполняющая среда

CLR – виртуальная машина для *промежуточного языка* (IL – intermediate language), в который транслируются все .NET программы

Наиболее близок к промежуточному языку C#, специально разработанный вместе с платформой .NET

.NET Framework

- в отличие от Java, код на промежуточном языке не интерпретируется, а всегда выполняется в режиме динамической компиляции (JIT – just-in-time compilation)
- основной разработчик и поставщик платформы .NET – компания Microsoft
- наличие стандартов делает возможным независимую реализацию .NET (например, такая реализация разработана в рамках проекта Mono), но реализации не от Microsoft используются достаточно редко



объектно-ориентированный язык, поддерживающий автоматическое управление памятью и работу в многопоточном режиме

План рассмотрения

1. Лексика
2. Общая структура программ
3. Базовые типы и операции над ними
4. Инструкции и выражения
5. Пользовательские типы
6. …

C#: лексика

- программы пишутся с использованием символов Unicode (16 бит на символ)
- комментарии
  - однострочный: **`// и до конца строки`**
  - выделительный: **`/* комментарий */`**
- идентификаторы: должны начинаться с буквы (в понятии Unicode или _) и продолжаться буквами или цифрами: **`myIdentifier123`**, **`αρετη_μυσ`**, **`идентификатор123`**
- ключевые слова – как идентификаторы, но только из латинских букв

C#: литералы

- **`null`** – пустая ссылка
- булевские литералы: **`false`** и **`true`**
- символьные литералы: в одинарных кавычках, ESC-последовательности: **`'a'`**, **`'#'`**, **`'Ы'`**, **`'\''`** (одинарная кавычка), **`'\\'`** (обратный слеш), **`'\n'`** (перевод строки), **`'\r'`** (возврат каретки), **`'\t'`** (табуляция)
- строковые литералы – последовательность символов в двойных кавычках: "Hello", "Привет". Могут быть разбиты на несколько частей с помощью знака **+**: **`"Hello," + " world"`** (по значению совпадает с **`"Hello, world"`**)

C#: литералы

- буквальный строковый литерал –

  ESC-последовательности не преобразуются. Для этого перед открывающейся кавычкой ставят @:

  ```
  @"Hello \t world"
  @"C:\Windows\calc.exe"
  ```

  Допускаются любые символы, кроме ". Чтобы поместить туда и кавычку, надо повторить её два раза: `@""""`

C#: литералы

- целочисленные литералы: 123, -123, 0x12A, 0X1AB – по умолчанию относятся к типу **`int`**. Целочисленные литералы типа **`long`** заканчиваются на букву **`l`** или **`L`**: 123L.
- Беззнаковые целочисленные типы
  - **`uint`**: 123u, 123U
  - **`ulong`**: 123ul, 123 uL, 123Ul, 123UL

C#: литералы

- числа с плавающей точкой
  - обычная запись: 1.23
  - экспоненциальная запись: 1.23e1

  по умолчанию такие литералы относятся к типу **`double`** и могут иметь на конце символ **`d`** или **`D`**.

  Литералы типа **`float`** оканчиваются буквами **`f`** или **`F`**: 12.34f, -12.34e+5F

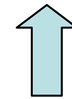
C#: общая структура программы

- программа – набор пользовательских типов данных – классов
- во избежании конфликтов по именам используются *пространства имён* (namespaces), имена пространств имён могут состоять из нескольких идентификаторов, разделённых точками: **`System.Drawing`**
- Из любого места можно сослаться на некоторый тип, используя его длинное имя, состоящее из имени содержащего его пространства имён, точки и имени самого типа

C#: общая структура программы

- код пользовательских типов размещается в файлах с расширением .cs
- декларация пространства имён:

  namespace mynamespace { … }

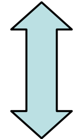


- все типы, описанные в фигурных скобках, попадают в это пространство имён
- типы, описанные вне декларации пространства имён, попадают в пространство имён с пустым именем

C#: общая структура программы

- пространства имён могут быть вложены

```
namespace A.B { … }
```



результаты эквивалентные

```
namespace A {
 namespace B {
  …
 }
}
```

- в одном файле можно декларировать много типов, относящихся к разным пространствам имён
- элементы одних и тех же пространств имён могут описываться в разных файлах

C#: общая структура программы

- пользовательский тип описывается целиком в одном файле, за исключением *частичных типов*, помеченных модификатором **`partial`** – их элементы можно описывать в разных файлах, и эти описания объединяются, если не противоречат друг другу

C#: общая структура программы

- чтобы ссылаться на типы, декларированные в других пространствах имён, по их коротким именам, можно воспользоваться директивами использования:

  `using System.Collections;`

  делает возможным ссылки с помощью короткого имени на любой тип (или вложенное пространство имён) пространства имён **`System.Collections`**

  `using System;`

  ~~**`using Collections;`**~~

  `using System.Collections;`

C#: общая структура программы

- результат компиляции C# программы – динамически загружаемая библиотека (DLL) или исполняемый файл (.exe), имеющие особую структуру. Такие библиотеки называются *сборками* (*assembly*)
- входной точкой программы является метод

  ```
  public static void Main()
  ```

  одного из классов. Такой метод может также иметь параметр типа **`string[]`** (параметры командной строки) и/или возвращать значение типа **`int`**

C#: базовые типы

- в C# имеются
  - ссылочные типы
    - имеют собственную идентичность
    - на такой объект можно иметь ссылку из другого объекта
    - передаются по ссылке если являются аргументами или результатами методов
  - типы значений
    - представляют собой значения, не имеющие собственной идентичности – все равные между собой значения неотличимы друг от друга
    - нельзя сослаться только на одно из них

C#: базовые типы

- примитивные типы данных
  - логический
  - числовой
  - символьный
- есть возможность декларировать пользовательские типы значений — *структурные типы* (**`struct`**) и *перечисления* (**`enum`**)
- ссылочные типы называются *классами* (**`class`**)
- и ссылочные типы, и типы значений наследуются от **`System.Object`**, который также можно использовать под именем **`object`**
- для каждого примитивного типа есть структурный тип-обёртка (преобразования происходят неявно)

C#: базовые типы

Поэтому все элементы класса **`object`** имеются во всех примитивных типах — у их значений можно, как у обычных объектов, вызывать методы:

```
2.Equals(3)
(-123).ToString()
```

C#: логический тип — `bool`

- его обёртка — System.Boolean
- значения — логические значения, их всего два: **true** и **false**
- нет никаких неявных преобразований между логическими и целочисленными значениями

C#: логический тип

- операции
  - == и != — сравнение на равенство и неравенство
  - ! — отрицание
  - && и || — условные (короткие) конъюнкция и дизъюнкция
  - & и | — длинные конъюнкция и дизъюнкция
  - ^ — исключающее «или» или сумма по модулю 2
  - для операций &, |, ^ имеются соответствующие операции присваивания: &=, |=, ^=

C#: целочисленные типы

размер	знаковый	беззнаковый
1 байт	**`sbyte`** (System.SByte) (-128..127)	**`byte`** (System.Byte) (0..255)
2 байта	**`short`** (System.Int16) (-32768..32767)	**`ushort`** (System.UInt16) (0..65535)
4 байта	**`int`** (System.Int32) (-2^{31}..[2^{31}-1])	**`uint`** (System.UInt32) (0..[2^{32}-1])
8 байт	**`long`** (System.Int64) (-2^{63}..[2^{63}-1])	**`ulong`** (System.UInt64) (0..[2^{64}-1])

C#: целочисленные типы

- минимальные и максимальные значения можно найти в их типах-обёртках в виде констант **`MinValue`** и **`MaxValue`**
- операции
  - == и != — сравнение на равно и неравно
  - <, <=, >, >= — порядок
  - +, -, *, /, % — арифметика
  - ++, -- — увеличение/уменьшение на единицу, могут быть префиксные и постфиксные

    ```
    x=1; y=++x; z=x++; // x=? y=? z=?
    ```
  - ~, &, |, ^ — побитовые логические операции
  - <<, >> — сдвиг (>> дополняет для знаковых типов значением знакового бита, для беззнаковых — нулём)
  - +=, -=, *=, /=, %=, ~=, &=, |=, ^=, <<=, >>=

C#: типы с плавающей точкой

- представление (**float**, **double**) и операции над ними соответствуют стандарту IEEE 754
  - – float: 32бита=1+23+8 (7-8 значащих цифр)
  - – double: 64=1+52+11 (15-16 значащих цифр)
- помимо обычных чисел есть значения:
  - -0.0
  - +∞, -∞
  - NaN – Not-a-Number

C#: типы с плавающей точкой

Операции:

- ==, != — равно/неравно (-0.0==0.0, NaN не равно ни одному числу и самому себе)
- <,<=,>,>= — порядок (не верно: -0.0<0.0), все операции с NaN возвращают false
- +,-,*,/,% — арифметика, a%b==a-b*n, где n — самое большое по модулю целое число, не превосходящее |a/b|, знак которого совпадает со знаком a/b
- 1.0/0.0=+∞   -1.0/0.0=-∞   0.0/0.0=NaN
- ++, -- — увеличение/уменьшение на единицу

C#: типы с плавающей точкой

- классы-обёртки
  - **float** <=> System.Single
  - **double** <=> System.Double
- специальные значения (**const**):
  - **`MaxValue`** (макс конечное значение)
  - **`Epsilon`** (мин положительное значение)
  - **`PositiveInfinity`** (положительная бесконечность)
  - **`NegativeInfinity`** (отрицательная бесконечность)
  - **`NaN`**
- **decimal** (System.Decimal) — 28 знач. цифр

C#: строковые преобразования

у всех базовых типов есть методы

- Parse(string) — преобразует строковое представление к базовому типу:

```
int x = int.Parse("123");
bool b = Boolean.Parse("True");
```

- ToString() — возвращает строковое представление объекта; метод класса System.Object, поэтому имеется у всех типов данных

```
123.ToString();  // => "123"
 b.ToString();     // => "True" или "False"
```

C#: выражения

- выражения строятся при помощи применения операторов к именам и литералам
- операторы:
  - x.y — уточнение имени
  - f(x) — вызов метода *f* с параметрами *x*
  - a[x] — вычисление элемента массива
  - new — оператор создания нового объекта
  - ++, -- — унарные операторы инкремента/декремента
  - (T)x — явное преобразование типа
  - арифметика, сравнение
  - ?: — тернарный условный оператор (a ? x : y)
  - =, =+, =-, … — присваивание

C#: выражения

- контролируемые преобразования типов
  – оператор **is**:

    (x **is** T) — тип результата логический

  – оператор **as**:

    (x **as** T) — тип результата T, если x не приводится к типу T, результат — **null**

```
class A { … }

class B: A { … }

B x = new B();
```

```
(x is B) == true

(x is A) == true
```

C#: инструкции

- большинство инструкций заимствованы из языка Си
- *блок* — набор инструкций в { }
- пустая инструкция — ;
- декларация локальных переменных:

  <тип> <имя>[=<значение>];

  int[ ][ ] array = **new int**[ ][ ]{{1,2},{3,4,5}};
- инструкция может быть помечена с помощью метки (<метка>: <инструкция>)

C#: инструкции

- условная инструкция:

  if (*expression*) *statement*;

  или

  if (*expression*) *statement*1;
  else *statement*2;

expression — выражение логического типа

C#: инструкции

- инструкция выбора:

  switch (*expression*) { … }

  где *expression* может быть
    - целочисленным
    - перечислением
    - строковым (**string**)
  - группа инструкций обязательно должна оканчиваться
    - **break**
    - **goto default**
    - **goto case** *value*

C#: инструкции

- циклы **while** и **do**

  while (*expression*) *statement*
  do *statement* **while** (*expression*);

  где *expression* — логическое выражение,
  statement — тело цикла

  семантика та же, что и в Си

C#: инструкции

- цикл **for** заимствован из Си:

  for (*A*; *B*; *C*) *statement*

  выполняется практически как

  A; **while** (*B*) { *statement* *C*; }

  - любой из элементов *A*, *B*, *C* может отсутствовать
  - *B* — выражение логического типа (при отсутствии заменяется на **true**)

C#: инструкции

- цикл перебора элементов коллекции

  foreach (*type id* **in** *expression*)

  statement

  выражение *expression* должно быть массивом или коллекцией

C#: инструкции

- инструкции **break** и **continue**
  - **break** — прерывает самый вложенный, содержащий его цикл
  - **continue** — прерывает выполнение текущей итерации и переходит к выполнению следующей (если она есть)
- инструкция **goto** — передаёт управление на инструкцию, помеченную меткой, которая следует за **goto**

C#: инструкции

- инструкция возврата управления

  return

  используется для возврата управления из операции

  если операция должна вернуть значение некоторого типа, после **return** должно стоять выражение этого же типа

  return *expression*

C#: инструкции

- инструкция создания исключительной ситуации:

  throw *expression*

  где *expression* должно иметь тип исключения

  Исключение (***exception***) — объект, с информацией о какой-то особой (исключительной) ситуации, в которой операция не может вернуть обычный результат

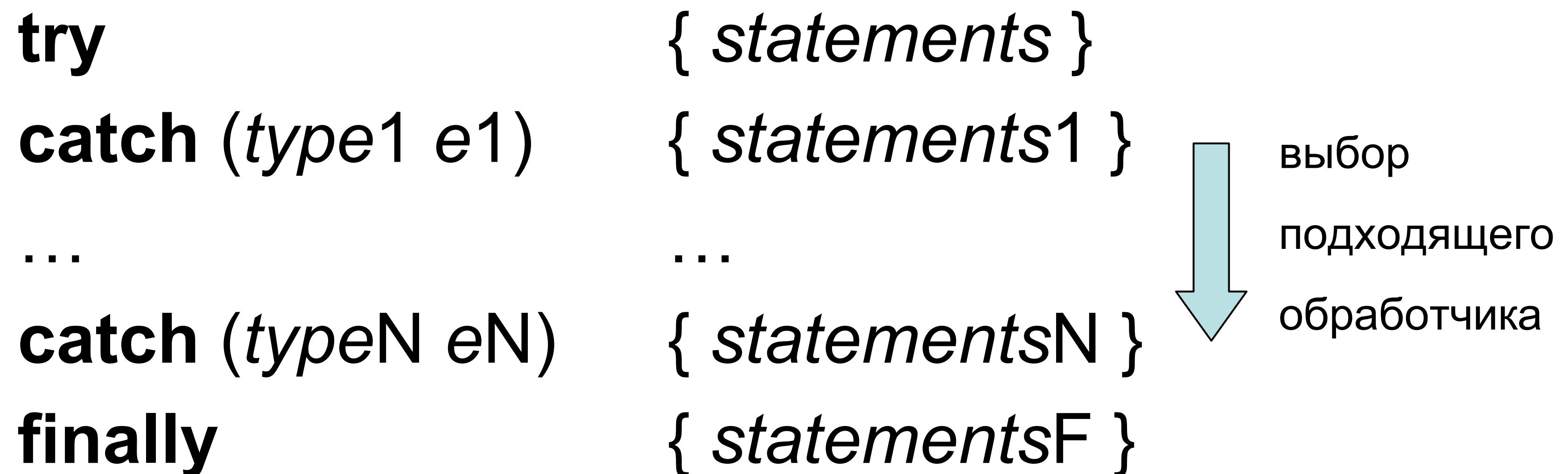
C#: инструкции

- исключения C# — все наследники System.Exception
- объекты-исключения как минимум содержат:
  - сообщение о возникшей ситуации (свойство **string** Message)
  - состояние стека

    (свойство **string** StackTrace)

C#: инструкции

- блок обработки исключительных ситуаций



хотя бы один **catch** или **finally** должен быть