

ГБПОУ г. Москвы

«Первый московский образовательный комплекс»

Объектно-ориентированное программирование

Лекция 3. Введение в язык C#

Язык объектно-ориентированного программирования C#

Общие сведения по языку C#

- Появился в 2001 году.
- Основан на языках Java и Visual Basic
- Общий прародитель C++
- В первой версии языка:
 - 80 ключевых слов
 - 12 встроенных (базовых) типов данных
- Включает все необходимое для создания объектно-ориентированных, компонентных программ.
- Одобрен в качестве международного стандарта [ECMA](#) (ECMA-334) и [ISO](#)(ISO/IEC 23270)

Андерс Хейлсберг (Anders Hejlsberg)



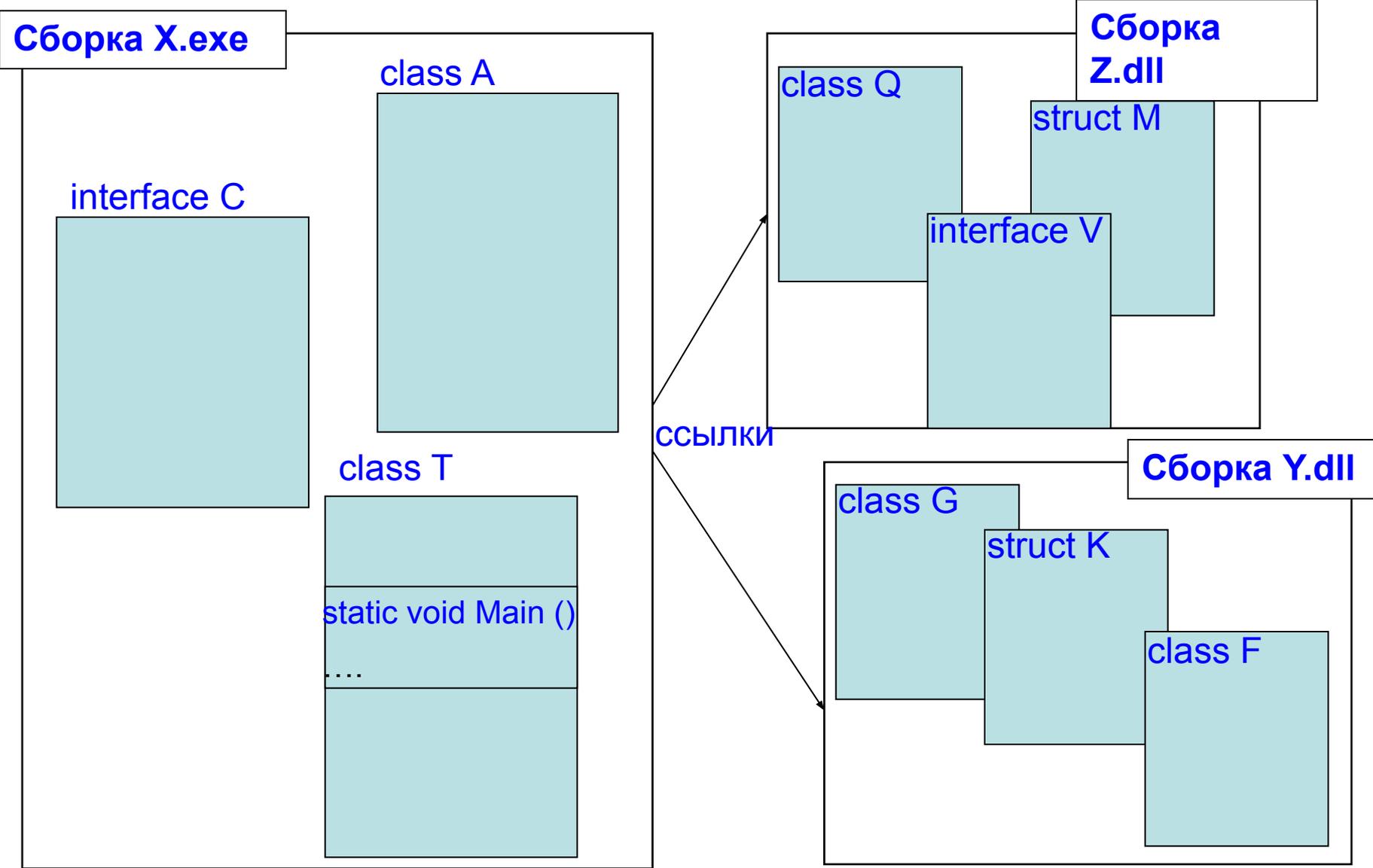
- Главный проектировщик и ведущий архитектор.
- Датский учёный в области информатики.
- В 1980 году он написал компилятор языка Паскаль, который продал фирме Borland (этот компилятор дожил до 7 версии (*Borland Pascal*)).
- До 1996 года главный проектировщик фирмы Borland, создал новое поколение компиляторов Паскаля: получился язык [Delphi](#).
- В 1996 году перешёл в Microsoft, где работал над языком J++ и библиотекой C++ - Windows Foundation Classes.
- Позже возглавил комиссию по созданию и проектированию языка C#.

Программа на C#

- Программа это набор взаимосвязанных классов.
- Класс содержит данные и функции
- В одном из классов должна быть функция **Main**, с этой функции начинается выполнение программы
- Классы могут быть в разных файлах, в разных сборках (в библиотеках классов).
- На внешние модули (библиотеки, или выполняемые модули) должны быть ссылки (references).
- Для удобства ссылок на классы программы, желательно использовать пространство имен
- Для удобства записи имен внешних классов можно использовать оператор **using**.

- Определение программы (Дейкстра)
 - Программа = Алгоритм + Данные.
- Объектно-ориентированное определение программы:
 - Программа это набор типов (классов, интерфейсов и т.д.)
 - Тип = Данные + Методы
 - Метод = Алгоритм + Данные.

Структура программы (сборки) на языке C#



```

using <имя namespace>;
...
namespace Nnnn
{
    Объявление класса
class Cccc
    ...
    Объявление структуры
struct Ssss
    ...
    Объявление интерфейса
interface Iccc
    ...
}

```

Рис. 2.3. Структура приложения.

```

class Cccc
{
    Объявление поля _a
    ...
    Объявление свойства A
    ...
    Объявление метода M( )
    ...
    Объявление события
event A
    ...
}

```

Рис. 2.4. Структура описания класса

Простая программа на C#

```
using System;
namespace ConsoleApp
{
    class Program
    {
        static void Main()
        {
            Console.Write("Введите радиус круга:");
            string s = Console.ReadLine();
            double r = Convert.ToDouble(s);
            double p = Math.PI * r * r;
            Console.WriteLine("Площадь круга = {0}", p);
            Console.ReadLine();
            return;
        }
    }
}
```

Сравнение C# и Java

```
using System;
class Program
{
public static void Main(string [] )
{
    Console.Write("Введите радиус круга:");
    string s = Console.ReadLine();
    double r = Convert.ToDouble(s);
    double p = Math.PI * r * r;
    Console.WriteLine("Площадь круга = {0}", p);
    Console.ReadLine();
}
}
```

```
import java.util.Scanner;
class Program
{
private static final double PI = 3.1416;
public static void main ( String [] args )
{
    Scanner keyboard=new Scanner ( System.in ) ;
    System.out.print ( "Введите радиус круга:" ) ;
    float r = keyboard.nextFloat();
    float p = PI * r * r;
    System.out.print ( "Площадь круга = :" ) ;
    System.out.println( 2*val*val*PI ) ;
}
}
```

Ввод с клавиатуры в Java

```
byte in[ ] = new byte[100];  
System.in.read(in);  
String vvod = new String(in);  
vvod = vvod.trim();  
int b = Integer.parseInt(vvod);
```

Классы

- **Классы это основные пользовательские типы данных.**
- Экземпляры класса – **Объекты.**
- Классы описывают
 - все элементы объекта (данные)
 - его поведение (методы),
 - устанавливают начальные значения для данных объекта, если это необходимо.
- При создании экземпляра класса в памяти создается копия данных этого класса.
 - Созданный таким образом экземпляр класса называется объектом.

Составные элементы класса

1. **Поля (field)** – обычно скрытые данные класса (внутренне состояние)
2. **Методы (methods)** – операции над данными класса (поведение) (можно называть функциями)
3. **Свойства (property)** – доступ к данным класса с помощью функций
 - **get** – получить
 - **set** – задать
4. **События (event)** – оповещение пользователей класса о том, что произошло что-то важное.

- Экземпляры классов создаются с помощью оператора *new*.
- Для получения данных объекта или вызова методов объекта, используется оператор “.” (точка).

```
Student s;
```

```
s = new Student();
```

```
s.Name = “Иванов А.”;
```

- При создании экземпляра класса, копия данных, описываемых этим классом, записывается в память и присваивается переменной ссылочного типа

- В этом случае описание класса **Автомобиль** может выглядеть следующим образом:

```
class Автомобиль
{
// описание свойств
  public string Модель;
  public float Расход_топлива;
  private int Число_цилиндров;
// описание методов
  public void Повернуть_руль(){...};
  private Регулировка_датчика(){...};
// описание события
  event Перегрев_двигателя();
}
```

Описание классов программы

```
using XXX; // чужие пространства имен
namespace MMM // свое пространство имен
{
    class AAA // наш класс MMM.AAA
    {
        ...
    }
    class BBB // другой наш класс MMM.BBB
    {
        ...
    }
}
```

Пример простого класса

```
namespace TestProg // наше пространство имен
{
    class Point // наш класс MMM.Point
    {
        public int x, y; // поля класса
    }
    class Program
    {
        static void Main( )
        {
            Point a;
            a = new Point();
            a.x = 4;
            a.y = 3;
        }
    }
}
```

Пример описания и использования класса

- Самый простой класс

```
class Car
{
}
```

- Класс с полями

```
class Car
{
    // состояние Car.
    public string petName;
    public int currSpeed;
}
```

- Класс с методами

```
class Car
{
    // состояние Car.
    public string petName;
    public int currSpeed;
    // функциональность Car.
    public void PrintState()
    {
        Console.WriteLine("{0} is going {1}
MPH.",
            petName, currSpeed);
    }
    public void SpeedUp(int delta)
    {
        currSpeed += delta;
    }
}
```

```
static void Main(string[] args)
{
    Console.WriteLine
        ("***Используем Class Types***");
    // Создаем и настраиваем объект Car.
    Car myCar = new Car();
    myCar.petName = "Henry";
    myCar.currSpeed = 10;
    // Ускоряем автомобиль несколько раз
    // и выводим на печать новое состояние.
    for (int i = 0; i <= 10; i++)
    {
        myCar.SpeedUp(5);
        myCar.PrintState();
    }
    Console.ReadLine();
}
```

- Неправильное использование класса

```
static void Main(string[] args)
{
    // Ошибка! Забыли использовать операцию 'new'!
    Car myCar;
    myCar.petName = "Fred";
}
```

Инкапсуляция (*Encapsulation*)

- Максимальное закрытие доступа к состоянию объектов.
- Состояние объекта можно менять только используя свойства и методы (не используя переменные).
- Свойства и методы открытые внешним пользователям класса - интерфейс (interface).
- Это позволяет
 - Избежать неправильного использования объектов (защита от дурака)
 - Изменять и развивать класс не мешая его использовать

Описатели режимов доступа (access modifiers)

| Access Modifier | Ограничения |
|---------------------------------|---|
| <code>public</code> | Нет ограничений. Элементы отмеченные <code>public</code> видны любому методу любого класса. |
| <code>private</code> | Элементы класса А отмеченные как <code>private</code> доступны только методам класса А. |
| <code>protected</code> | Элементы класса А отмеченные как <code>protected</code> доступны методам класса А и методам <i>производным</i> от класса А. |
| <code>internal</code> | Элементы класса А отмеченные как <code>internal</code> доступны методам любого класса в сборке, в которой описан класс А. |
| <code>protected internal</code> | Элементы класса А отмеченные как <code>protected internal</code> доступны методам класса А, методам классов производных от класса А, а также любому классу в сборке, где описан класс А. Это как <code>protected</code> ИЛИ <code>internal</code> . (Нет режима <code>protected</code> И <code>internal</code> .) |

Типы данных

Основные понятия

- Программа это набор типов

$$P = \{T_1, T_2, \dots, T_n\}$$

- Тип задает:
 - Количество ячеек памяти
 - Состояние (значения данных)
 - Поведение (методы)
 - Операции в которых может участвовать
 - Преобразования к другим типам.

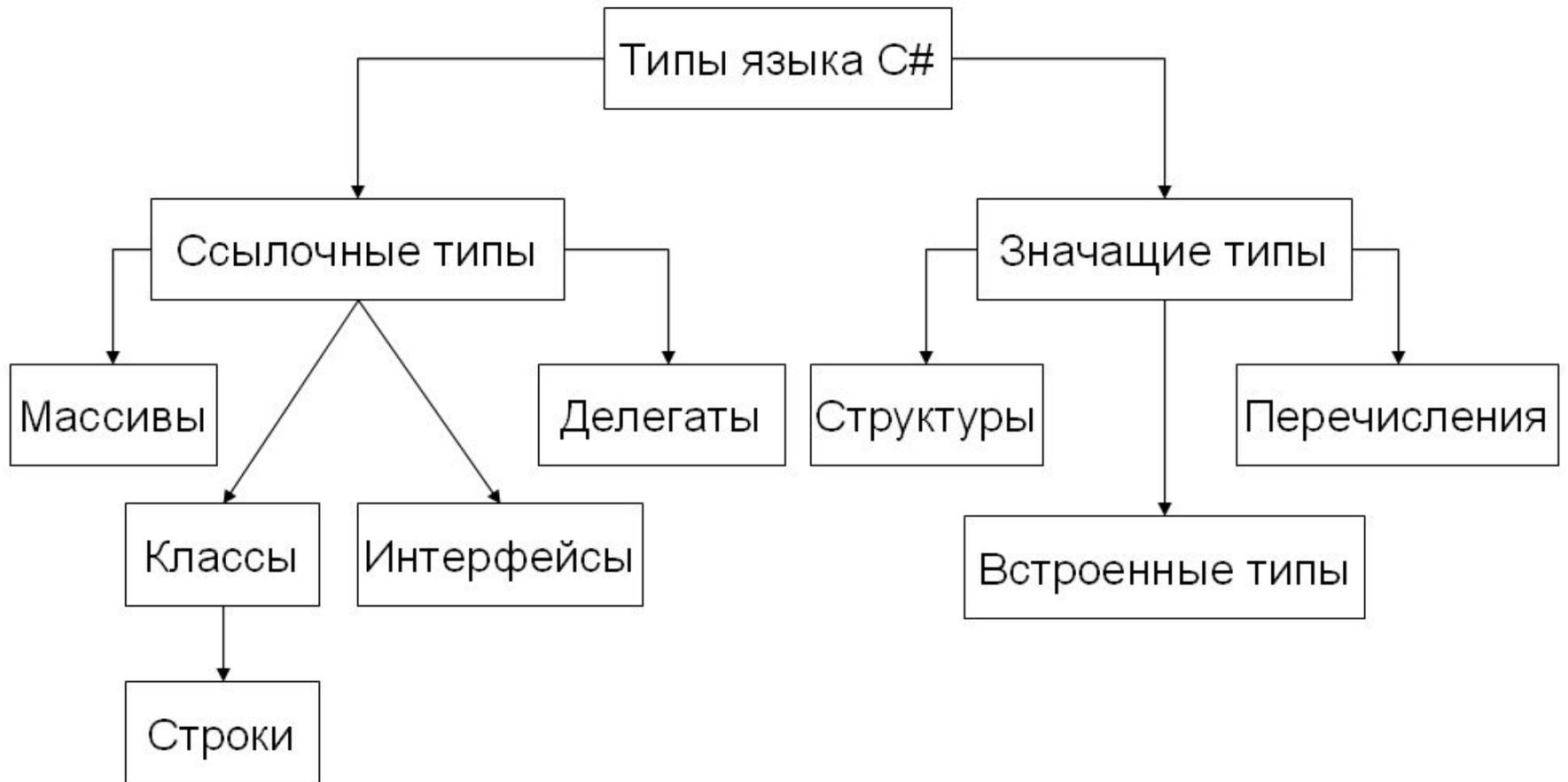
Тип данных

- Язык C# является строго типизированным языком – все данные (константы и переменные) программы имеют явно или неявно заданный *тип*.
- Тип данных определяет:
 - количество используемой памяти (в байтах);
 - набор операции, в которых может участвовать данные такого типа;
 - способы явного и неявного преобразования в другие типы.

Основные сведения о типах

- Все элементы программы имеют тип (*переменные, константы, выражения, методы, параметры методов, и т.п.*)
- Для всех переменных требуется объявлять тип.
- Результат вычисления выражения имеет определенный тип.
- **Переменные и выражения должны иметь один и тот же тип при присвоении.**
- Если типы разные, выполняется их преобразование:
 - неявное (без прямого указания программиста)
 - явное (в результате заданного преобразования, кастинг)

Виды типов языка C#



Хранение данных программы

Данные используемые программой (переменные, константы) могут храниться в в двух типах оперативной памяти:

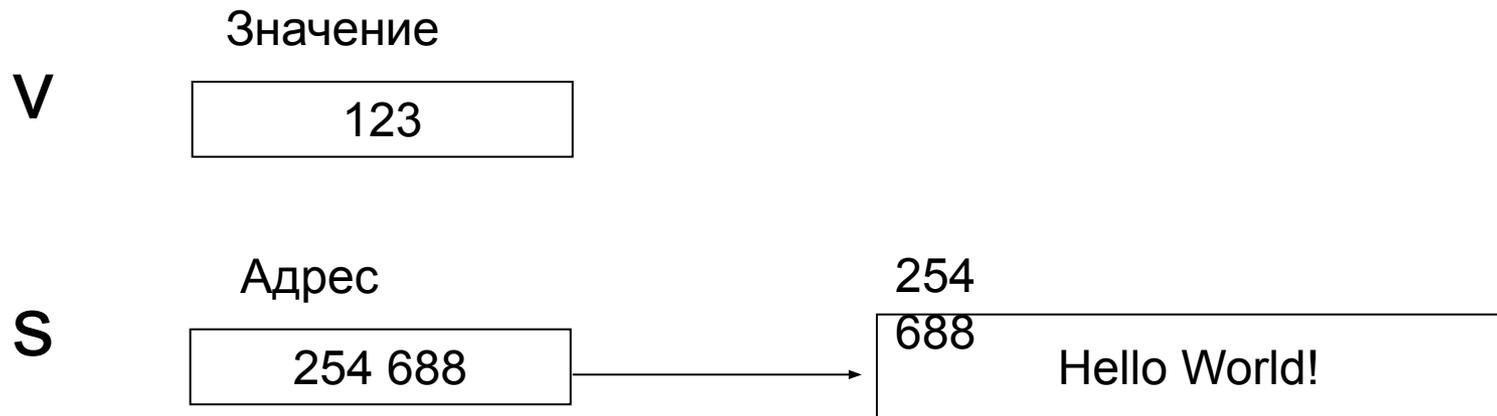
- Стек (линейная память)
- Куча (динамическая память)

Различие между значащими и ССЫЛОЧНЫМИ ТИПАМИ

```
int v = 123;
```

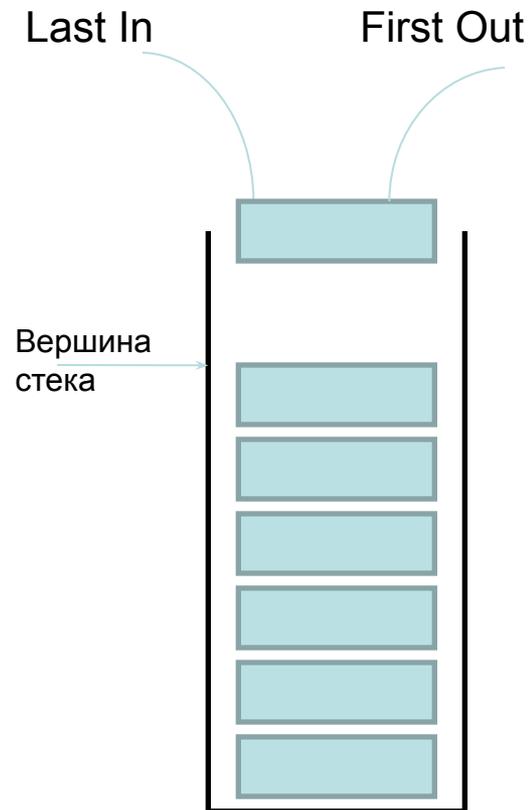
```
string s;
```

```
s = "Hello World!";
```



Стек (stack)

- Стек – это линейный участок памяти (массив), который действует как структура данных типа «Последним пришел – первым ушел» (last-in, first-out – LIFO).
- Данные могут добавляться только к вершине стека и удаляться из вершины.
- Добавление и удаление данных из произвольного места стека невозможно.
- Операции по добавлению и удалению элементов из стека выполняются очень быстро.
- Размеры стека, как правило, ограничены, и время хранения данных зависит от времени жизни переменной.
- Для всех локальных переменных методов и передаваемых методам параметров память выделяется в вершине стека.
- После того, как методы заканчивают работу вся выделенная память в стеке для их переменных автоматически освобождается.



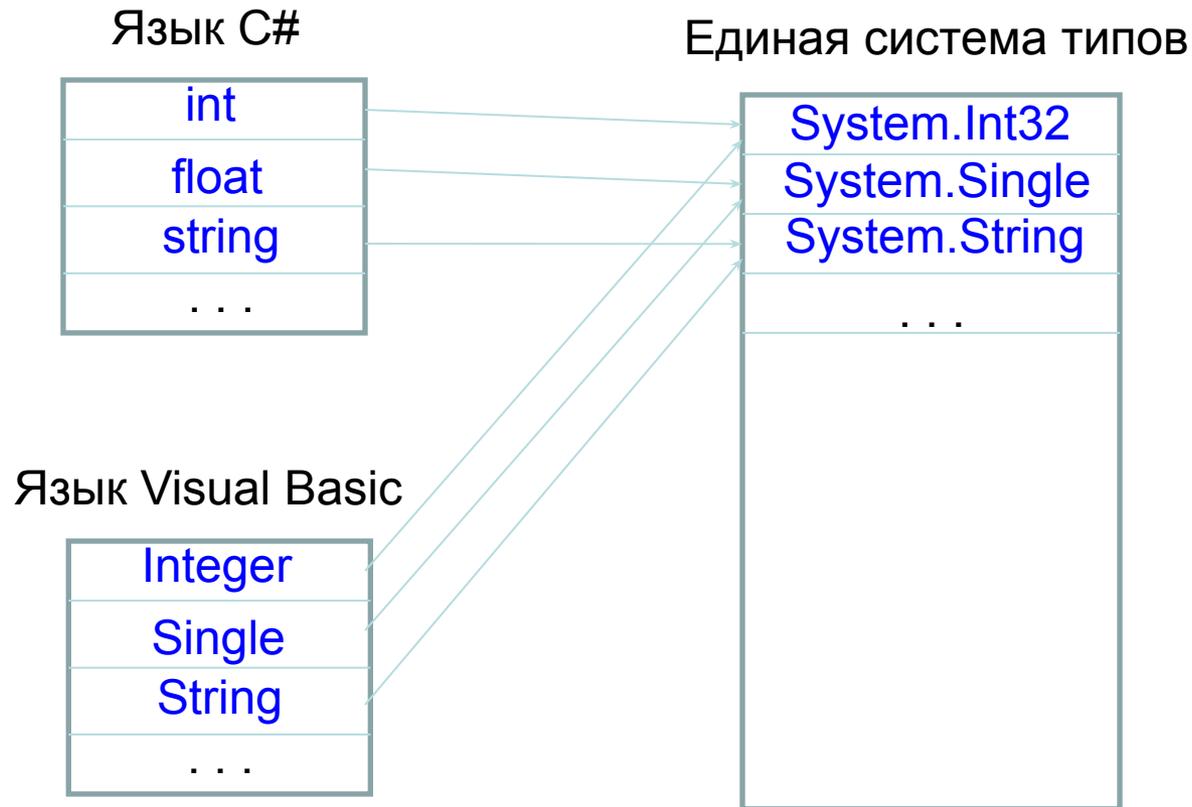
Куча (heap)

- **Куча (heap)** – это область оперативной памяти, в разных частях которой по запросу программы, операционная система может выделять участки требуемого размера для хранения объектов классов.
- Память в куче выделяется с помощью операции `new`.
- В отличие от стека, участки памяти в "куче" могут выделяться и освобождаться в любом порядке.
- Программа может хранить элементы данных в "куче", но она не может явно удалять их из нее.
- Вместо этого компонент среды CLR, называемый Garbage Collector (GC), автоматически удаляет не используемые участки "кучи", когда он определит, что код программы уже не имеет доступа к ним (не хранит их адреса).
- Автоматическая сборка мусора освобождает программиста от необходимости освобождать не используемую память вручную, что часто приводит к ошибкам работы программы.

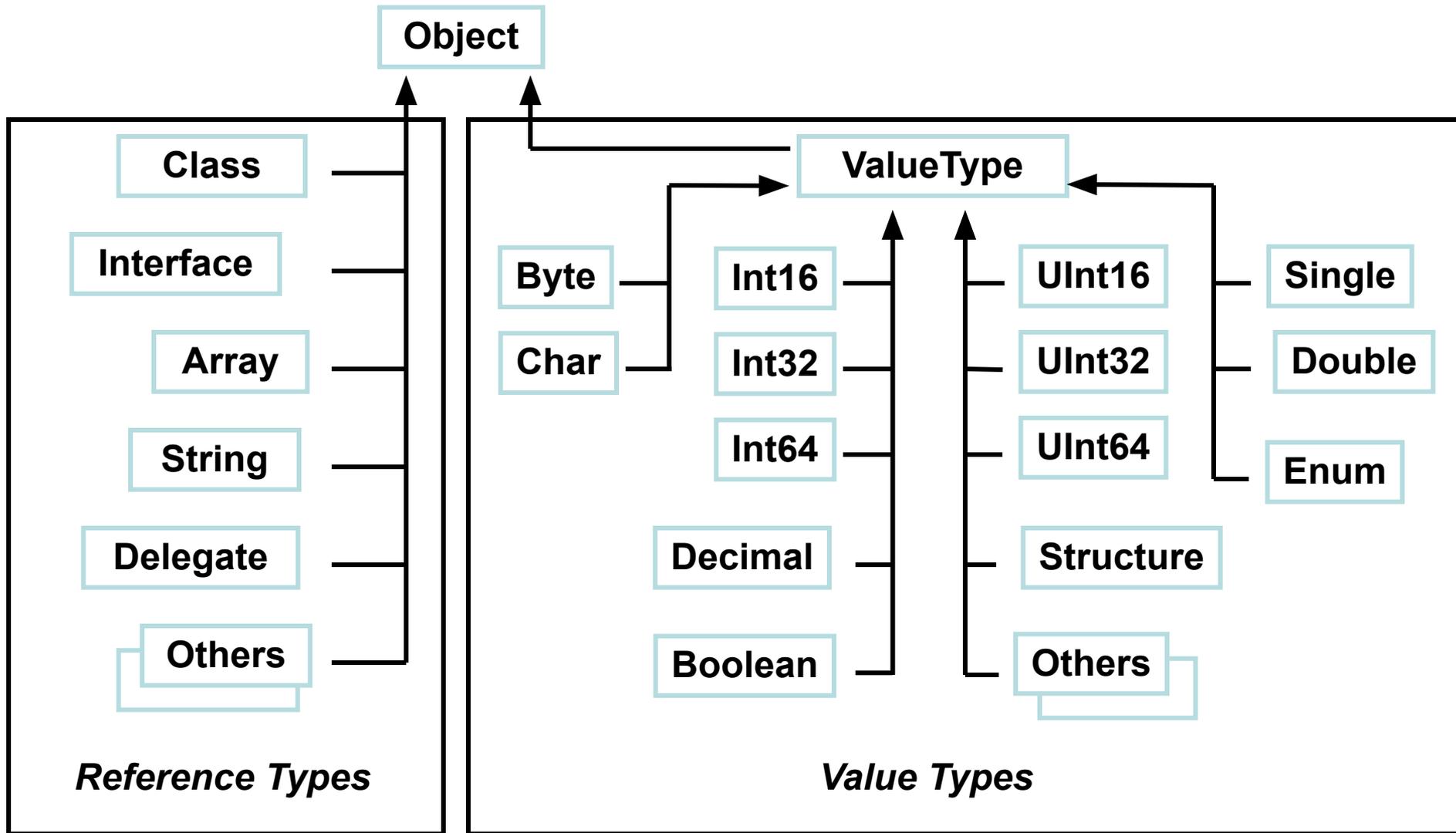
Системные типы данных CLR

- В .Net Framework есть общие для всех языков, системные типы данных.
- *Общая для всех языков система типов* (Common Type System, CTS).
- Эти типы разработаны специалистами компании Microsoft.
 - Например:
 - `System.Int32`
 - `System.Single`
 - `System.String`
 -

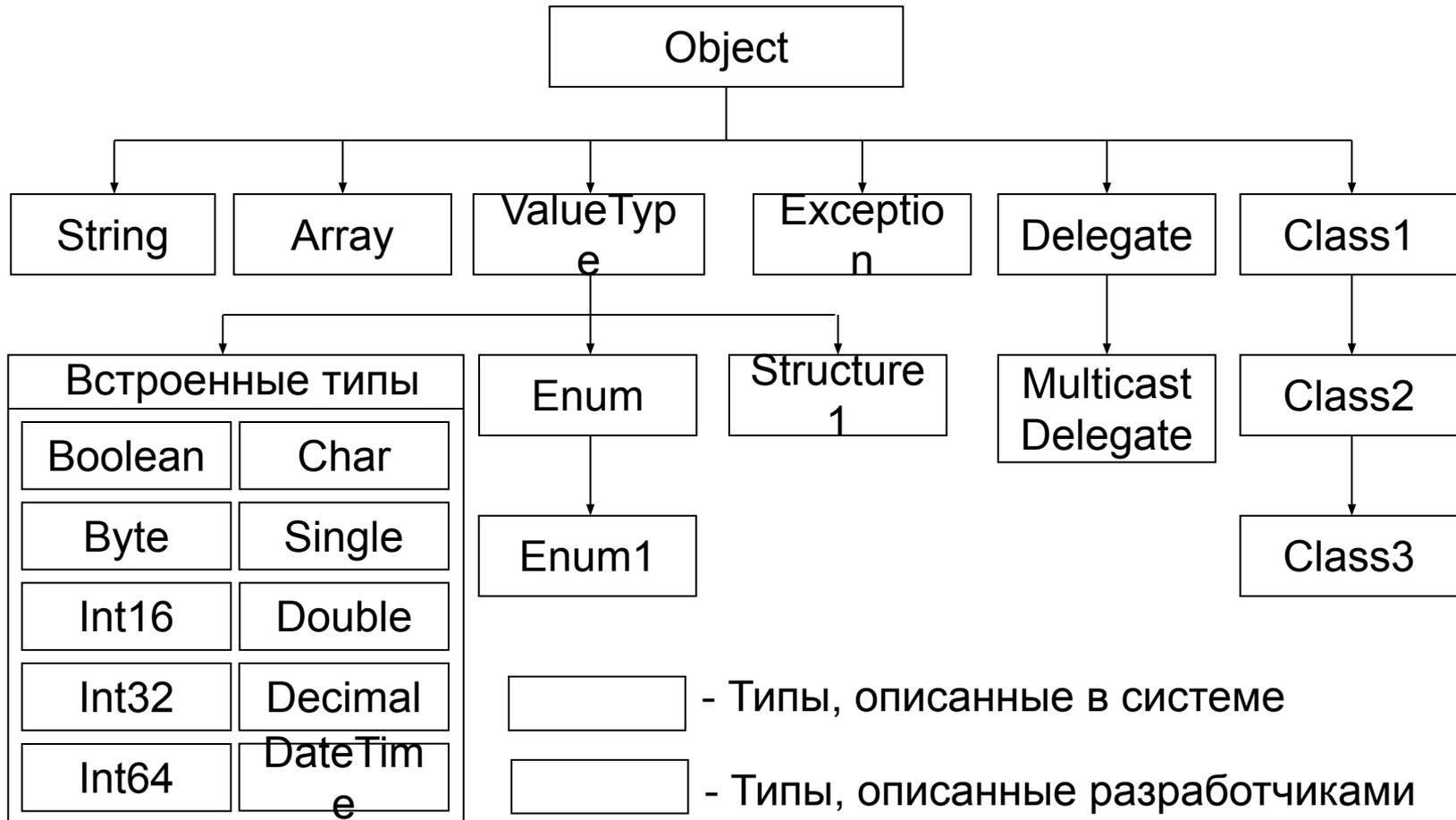
Соответствие встроенных типов и системных типов



Типы определенные в CLR



Наследование типов в CLR



Класс Object

- Все классы при создании наследуют все свойства и методы родительского класса **Object**.
 - Все *встроенные типы* нужным образом переопределяют *методы* родителя и добавляют *собственные поля, свойства и методы*.
- Типы, создаваемые пользователем, также являются *потомками класса Object*,
- Для них необходимо переопределить *методы* родителя, если предполагается использование *этих методов*;
 - реализация родителя, предоставляемая по умолчанию, не обеспечивает нужного эффекта.

Методы класса `System.Object`

- `Equals(object)`
 - виртуальный метод,
 - возвращает `true` если значения объектов совпадают (по умолчанию, если два объекта расположены в одном месте памяти).
- `GetHashCode()`
 - виртуальный метод,
 - возвращает целое число (*хэш-код*), однозначно идентифицирующее экземпляр класса.
- `GetType()`
 - возвращает объект типа `Type`, описывающий тип объекта.
- `ToString()`
 - виртуальный метод,
 - возвращает символьное представление значения переменной
 - по умолчанию возвращает строку, представляющую полное имя типа объекта.

Пример объявления переменных

- Пример объявления переменных и присваивания им значений в языке C# показан ниже:

```
int x=11;
```

```
int v = new Int32();
```

```
v = 007;
```

```
string s1 = "Agent";
```

```
s1 = s1 + v.ToString() + x.ToString();
```

Встроенные типы языка C#

| Имя типа | Системный тип CLR | Значения - диапазон | Размер – точность |
|--|-----------------------------|--|---------------------|
| Логический тип | | | |
| <code>bool</code> | <code>System.Boolean</code> | <code>true, false</code> | 8 бит |
| Арифметические целочисленные типы | | | |
| <code>sbyte</code> | <code>System.SByte</code> | -128 ÷ +127 | Знаковое, 8 бит |
| <code>byte</code> | <code>System.Byte</code> | 0 ÷ 255 | Беззнаковое, 8 бит |
| <code>short</code> | <code>System.Short</code> | -32768 ÷ +32767 | Знаковое, 16 бит |
| <code>ushort</code> | <code>System.UShort</code> | 0 ÷ 65535 | Беззнаковое, 16 бит |
| <code>int</code> | <code>System.Int32</code> | -2,147,483,648 ÷ +2,147,483,647 | Знаковое, 32 бит |
| <code>uint</code> | <code>System.UInt32</code> | ≈(0 ÷ 4*10 ⁹) | Беззнаковое, 32 бит |
| <code>long</code> | <code>System.Int64</code> | ≈(-9*10 ¹⁸ ÷ 9*10 ¹⁸) | Знаковое, 64 бит |
| <code>ulong</code> | <code>System.UInt64</code> | ≈(0 ÷ 18*10 ¹⁸) | Беззнаковое, 64 бит |

Встроенные типы (продолжение)

| Имя типа | Системный тип CLR | Значения - диапазон | Размер - точность |
|--|-----------------------------|---|-------------------------------|
| Арифметический тип с плавающей точкой | | | |
| <code>float</code> | <code>System.Single</code> | $\pm(1.5 \cdot 10^{-45} \div 3.4 \cdot 10^{+38})$ | 32 бита (точность 7 цифр) |
| <code>double</code> | <code>System.Double</code> | $\pm(5.0 \cdot 10^{-324} \div 1.7 \cdot 10^{+308})$ | 64 бита (точность 15–16 цифр) |
| Арифметический тип с фиксированной точкой | | | |
| <code>decimal</code> | <code>System.Decimal</code> | $\pm(1.0 \cdot 10^{-28} \div 7.9 \cdot 10^{+28})$ | 28–29 значащих цифр |
| Символьные типы | | | |
| <code>char</code> | <code>System.Char</code> | U+0000 ÷ U+ffff | 16 бит Unicode символ |
| <code>string</code> | <code>System.String</code> | Строка из символов Unicode | |
| Объектный тип | | | |
| Имя типа | Системный тип | Примечание | |
| <code>object</code> | <code>System.Object</code> | Базовый тип всех <i>встроенных</i> и пользовательских типов | |
| <code>void</code> | | Отсутствие какого-либо значения | |
| <code>var</code> | | Отложенное определение типа | |

Логический тип данных `bool`

- Соответствует системному типу `System.Boolean`.
- Может хранить только значения констант `true` и `false` (булевы константы)
- Можно назначать только булевы значения или константы, а также значений логического выражения:
`bool bc = (c > 64 && c < 123);`

Тип данных `decimal`

- Занимает 128-бит памяти.
- Имеет большую точность и меньший диапазон значений чем типы с плавающей точкой (floating-point);
 - Диапазон $\pm 1.0 \times 10^{-28} \div \pm 7.9 \times 10^{28}$
 - Точность 28 значащих цифр
- Больше подходит для финансовых и денежных вычислений;
- Чтобы константа обрабатывалась как `decimal` нужно добавить суффикс `m` или `M`:
`decimal myMoney = 300.5m;`

Не определенный тип - `var`

- Для переменной можно задать неопределенный тип (`var`) и присвоит некоторое значение.
- В этом случае компилятор автоматически определит тип присваиваемого значения и назначит его переменной.
- Например, объявление переменной:
`var name = "Петров А.В.";`
– аналогично следующему объявлению:
`string name = "Петров А.В.";`
- В этом случае обязательно нужно инициализировать переменную при ее объявлении.

- Можно также использовать неявное задание типа массива.
- Например, следующие операторы объявляют массив типа **Point**:

```
var points = new[ ] { new Point(1, 2), new  
Point(3, 4) };
```

Nullable типы данных

- **Nullable** типы данных это такие значащие типы данных, которые кроме обычных значений могут хранить и значение **null**.
 - Например, nullable **System.Boolean** может хранить значения из набора **{true, false, null}**.
- Очень удобны при работе с базами данных, которые кроме значений колонок могут указывать на отсутствие значения.
- Для описания **nullable** типа переменной нужно добавит символ вопроса (?) в конец названия типа.
- Это можно делать только для значащих типов.

Примеры nullable типов данных

```
static void LocalNullableVariables()  
{  
    // Описываем некоторые локальные значащие nullable типы.  
    int? nullableInt = 10;  
    double? nullableDouble = 3.14;  
    bool? nullableBool = null;  
    char? nullableChar = 'a';  
    int?[] arrayOfNullableInts = new int?[10];  
  
    // Error! String является ссылочным типом!  
    // string? s = "oops";  
}
```

Свойства **Nullable** типов

- Свойство `bool HasValue` – есть ли значение у переменной.
- `<тип> Value` – значение переменной (если переменная используется в не допустимой операции, то формируется исключение `System.InvalidOperationException`)
- Пример 1:
`int? n = null;`
`int m = 5 + n.Value; //формируется исключение`
- Пример 2:
`int? n = null;`
`// можно выполнить явное преобразование`
`int m = 5 + (int)n; //формируется исключение`
- Пример 3:
`int? n = null;`
`int? m = 5 + n; // m = null`

Использование nullable типов данных

```
static void Main(string[] args)
{
    Console.WriteLine("***** Работа с Nullable Data *****\n");
    DatabaseReader dr = new DatabaseReader();
    // Получаем int из “базы данных”.
    int? i = dr.GetIntFromDatabase();
    if (i.HasValue) // проверка на наличие значения переменной
        Console.WriteLine("Value of 'i' is: {0}", i.Value);
    else
        Console.WriteLine("Value of 'i' is undefined.");
    // Получаем bool из “базы данных”.
    bool? b = dr.GetBoolFromDatabase();
    if (b != null)
        Console.WriteLine("Значение 'b' равно: {0}", b.Value);
    else
        Console.WriteLine("Значение 'b' не определено.");
    Console.ReadLine();
}
```

Данные

- Программа работает с данными.
- Данные могут храниться в виде:
 - переменных
 - констант

Переменные программы

- **Переменные** – это именованные участки оперативной памяти, которые могут хранить:
 - значения некоторого типа (для значащих типов, в стеке),
 - ссылки на экземпляры некоторых классов (для ссылочных типов, ссылки на объекты, расположенные в "куче").

Объявление переменных

- Прежде, чем переменная может быть использована, она должна быть объявлена.
- Объявление переменных можно делать в любом месте программы.
- При объявлении переменных задается:
 - имя (идентификаторы)
 - Должно начинаться с буквы или подчеркика (_).
 - Буква может быть из любого алфавита (unicode)
 - Количество символов не ограничено.
 - тип (встроенный или пользовательский)
 - могут быть заданы модификаторы
 - режим доступа,
 - возможность изменения,
 - сохранность значений.

Объявление переменных

- Формат объявления переменных:

<тип> <имя>;

<тип> <имя> [= <значение>]

[<модификаторы>] <тип> <имя> [= <значение>];

где [<модификаторы>] = {<режим доступа>, static, const}.

- Например:

```
public int x = 5;
```

```
public static const int n=10;
```

Время жизни переменных

- Переменные появляются (рождаются)
 - статические переменные создаются при запуске программы (описатель `static`).
 - не статические
 - переменные методов появляются в результате их объявления.
 - переменные классов (поля) появляются в результате создания объекта класса.
- Переменные удаляются (умирают)
 - не статические
 - Переменные методов после закрытия блока в котором они объявлены (`}`).
 - Переменные класса после уничтожения объекта
 - статические уничтожаются после завершения программы.

Области видимости переменных

- **Область видимости** переменной (variable scope) это участок программы, в котором переменную можно использовать.
- В общем случае областью видимости локальной переменной является участок программы, от строки, в которой она объявляется, до первой фигурной скобки, завершающей блок или метод, в котором переменная объявлена.
- Областью видимости локальных переменных, которые объявляются в операторах цикла (например, **for** или **while**), является содержание (тело) данного цикла.

Пример областей видимости переменных

```
public void ScopeTest() {  
    int n = 0;  
    for (int i = 0; i < 10; i++)  
    {  
        Console.WriteLine(i);  
    } // i выходит из области видимости и удаляется  
    // можно объявить другую переменную с именем i  
    { // начало блока  
        var i = "другой блок"; // строка  
        Console.WriteLine(i);  
    } // i опять выходит из области видимости  
} // переменная n тоже выходит из области видимости
```

Константы

- В С# *константы* могут задаваться в виде
 - литералов (набора символов) или
 - именованных *констант*.
- Например:
 $y = 7.7;$
 - значение *константы* "7.7" является одновременно ее именем, она имеет и тип.
- Константы с дробной частью по умолчанию имеют тип `double`.

Задание типа констант

- Тип целой константы определяется ее значением (количеством цифр).
- Константы с дробной частью имеют тип **double**.
- Для изменения типа констант используются приставки:
 - **F - float** – например: **0.23F**
 - **D - double** – например: **2.7D**
 - **M - decimal** – например: **12.34M**

- Для точного указания некоторых типов можно задавать символ, стоящий после литерала (в верхнем или нижнем регистре).
- Например:
 - `f` – тип `float` (например: `10.2f`);
 - `d` – тип `double` (например: `10.2d`);
 - `m` – тип `decimal` (например: `10.2m`).

Именованные константы

- Для объявления именованной *константы* перед типом переменной добавляется модификатор `const`,
- Именованную константу обязательна инициализация констант и не может быть отложена.
- Например:
`const float c = 0.1f;`

Строковые константы

- Под строковыми константами понимается последовательность символов заключенных в двойные кавычки.
 - Например: “Петров С.А.”
- В C# существуют два вида строковых констант:
 - **обычные константы**, которые представляют строку символов, заключенную в двойные кавычки – "SSSS";
 - **@-константы**, заданные обычной константой с предшествующим знаком @.

Строковые константы

- В обычных константах некоторые символы интерпретируются особым образом.
 - Это требуется, для задания в строке специальных управляющих символов, в виде `escape`-последовательностей.
- Например:
 - `\n` - символ перехода на новую строку;
 - `\t` - символ табуляции (отступ на заданное количество символов);
 - `\\` - символ обратной косой черты;
 - `\"` - символ кавычки, вставляемый в строку, но не сигнализирующий о ее окончании.

@-константы

- Часто при задании констант, определяющих путь к файлу, приходится каждый раз удваивать символ обратной косой черты: “C:\\test.txt”, что не совсем удобно.
- В этом случае и используются @-константы, в которых все символы понимаются в полном соответствии с их изображением.
- Например, две следующие строки будут аналогичными:

```
s1 = "c:\\c#book\\ch5\\chapter5.doc";
```

```
s2 = @"c:\c#book\ch5\chapter5.doc";
```

2. Операции

Операции

- Переменные и константы могут участвовать (объединяться) с помощью операций.
- **Операция** – это термин или символ, получающий на вход одно или несколько операндов (переменных или констант) или выражений (переменных или констант, связанных между собой знаками операций), и возвращающий значение некоторого типа.
- Например: $a + b$ или $++a * pi$.

Виды операций

- Операции, получающие на вход один операнд, например операция приращения (**++**) или **new**, называются **унарными операциями**.
- Операции, получающие на вход два операнда, например, арифметические операции (**+**, **-**, *****, **/**) называются **бинарными операциями**.
- Одна операция – условная (**?:**), получает на вход три операнда и является единственной **тринарной операцией** в C#.

Базовые операции

| Основные операции | |
|-----------------------------------|---|
| <i>Выражение</i> | <i>Описание</i> |
| <code>x.y</code> | доступ к элементам типа |
| <code>f(x)</code> | вызов метода и делегата |
| <code>a[x]</code> | доступ к массиву и индексатору |
| <code>x++</code> | постфиксное приращение |
| <code>x--</code> | постфиксное уменьшение |
| <code>new T(...)</code> | создание объекта класса или делегата |
| <code>new T(...) {...}</code> | создание объекта с инициализацией |
| <code>new T[...]</code> | создание массива (см. раздел 3.5). |
| <code>typeof(T)</code> | получение объекта <code>System.Type</code> для <code>T</code> |
| <code>delegate { }</code> | анонимная функция (анонимный метод) |

Унарные операции

| Унарные операции | |
|------------------|--|
| <i>Выражение</i> | <i>Описание</i> |
| $-x$ | отрицательное значение |
| $!x$ | логическое отрицание |
| $\sim x$ | поразрядное отрицание |
| $++x$ | префиксное приращение |
| $--x$ | префиксное уменьшение |
| $(T)x$ | явное преобразование x в тип T (кастинг) |

Бинарные операции

| Мультипликативные операции | | Аддитивные операции | |
|----------------------------|-----------------|---------------------|--------------------------------|
| <i>Выражение</i> | <i>Описание</i> | <i>Выражение</i> | <i>Описание</i> |
| * | умножение | $x + y$ | сложение, объединение строк |
| / | деление | $x - y$ | вычитание |
| % | остаток | | |
| Операции сдвига | | Операции равенства | |
| <i>Выражение</i> | <i>Описание</i> | <i>Выражение</i> | <i>Описание</i> |
| $x \ll y$ | сдвиг влево | $x == y$ | равно |
| $x \gg y$ | сдвиг вправо | $x != y$ | не равно |

Бинарные операции (продолжение)

| Операции отношения и типа | |
|---|---|
| <i>Выражение</i> | <i>Описание</i> |
| <code>x < y</code> | меньше |
| <code>x > y</code> | больше |
| <code>x <= y</code> | меньше или равно |
| <code>x >= y</code> | больше или равно |
| <code>x is T</code> | возвращает значение true, если x относится к типу T, в противном случае возвращает значение false |
| <code>x as T</code> | возвращает x типа T или нулевое значение, если x не относится к типу T |
| Операции назначения и анонимные операции | |
| <i>Выражение</i> | <i>Описание</i> |
| <code>=</code> | присваивание |
| <code>x op= y</code> | составные операции присвоения: <code>+=, -=, *=, /=, %=, &=, =, !=, <<=, >>=</code> |

Логические и условные операции

| Логические, условные операции и Null-операции | | |
|---|------------------|---|
| Категория | Выражение | Описание |
| Логическое AND | $x \ \& \ y$ | целочисленное поразрядное AND, логическое AND |
| Логическое исключающее XOR | $x \ \wedge \ y$ | целочисленное поразрядное исключающее XOR, логическое исключающее XOR |
| Логическое OR | $x \ \ y$ | целочисленное поразрядное OR, логическое OR |
| Условное AND | $x \ \&\& \ y$ | вычисляет y только если x имеет значение true |
| Условное OR | $x \ \ y$ | вычисляет y только если x имеет значение false |
| Объединение нулей | $x \ \?\? \ y$ | равно y , если $x = \text{null}$, в противном случае равно x |
| Условное | $x \ ? \ y : z$ | равно y , если x имеет значение true, z если x имеет значение false |

Приоритеты операций языка C#

| Приоритет | Категория | Операции | Ассоциативность |
|-----------|-------------------------------------|---|-----------------|
| 0 | Первичные | (expr) x.y f(x) a[x] x++ x-- new sizeof(t) typeof(t) | Слева направо |
| 1 | Унарные | + - ! ~ ++x --x (T)x | Слева направо |
| 2 | Мультипликативные (Умножение) | * / % | Слева направо |
| 3 | Аддитивные (Сложение) | + - | Слева направо |
| 4 | Сдвиг | << >> | Слева направо |
| 5 | Отношения, проверка типов | < > <= >= is as | Слева направо |
| 6 | Эквивалентность | == != | Слева направо |
| 7 | Логическое | & | Слева направо |
| 8 | Логическое исключаящее ИЛИ (XOR) | ^ | Слева направо |
| 9 | Логическое ИЛИ (OR) | | Слева направо |
| 10 | Условное И | && | Слева направо |
| 11 | Условное ИЛИ | | Слева направо |
| 12 | Условное выражение | ? : | Справа налево |
| 13 | Присваивание | = *= /= %= += -= <<= >>= &= ^= = | Справа налево |

Пояснение приоритета операций

- Вычисление *выражений* начинается с выполнения операций высшего *приоритета*.
 - Например: первым делом вычисляются *выражения* в круглых скобках – (expr), определяются значения полей объекта – *x.y*, вычисляются функции – *f(x)*, переменные с индексами – *a[i]*.
- Можно заключать выражения в скобки для принудительного вычисления некоторых частей выражения раньше других.

Пример

- Выражение “ $2 + 3 * 2$ ” в обычном случае будет иметь значение 8, поскольку операции умножения выполняются раньше операций сложения.
- Результатом вычисления выражения “ $(2 + 3) * 2$ ” будет число 10, поскольку компилятор C# получит данные о том, что операцию сложения (+) нужно вычислить до выполнения операции умножения (*).

Ассоциативность операций

- Если есть несколько операций с одинаковым приоритетом, то они вычисляются в соответствии с их ассоциативностью.
- Операции с *левой ассоциативностью* вычисляются слева направо.
 - Например, $x * y / z$ вычисляется как $(x * y) / z$.
- Операции с *правой ассоциативностью* вычисляются справа налево.
 - Операции присваивания и третичная операция ($?:$) имеют правую ассоциативность.
- Операции с *левой ассоциативностью* вычисляются слева направо.
 - Все другие двоичные операции имеют левую ассоциативность.

Перегрузкой операций

- Порядок выполнения операций с объектами пользовательских классов и структур можно изменить.
- Такой процесс называется *перегрузкой операций*.

Тип результата операции

- Тип результата операции зависит от типов участвующих в операции операндов.
- Типом арифметической операции является наиболее сложный тип операнда. Значение другого операнда преобразуется к более сложному типу.
- Наименее сложный тип **byte**, наиболее сложный **decimal**.

```
int a=5;
```

```
double d=2.6;
```

```
a * d // тип результата double
```

```
a / 2 // тип результата int
```

Тип результата операции (2)

- Типом результата операции присваивания является тип левого операнда (переменной, которой присваивается значение).

```
int n;
```

```
n = a * d // тип результата int
```

- Тип операций отношения является `bool`.

```
a > 5 // тип результата bool
```

- Тип логических операций является `bool`.

```
bool b = true, c = false;
```

```
b && c // тип результата bool
```

Зачем нужны типы данных?

Чтобы гарантировать осмысленность выполняемых операций:

$$2 \text{ 🥬} + 3 \text{ 🥬} = 5 \text{ 🥬}$$

$$3 \text{ 🥕} + 4 \text{ 🥕} = 7 \text{ 🥕}$$

$$5 \text{ 🥬} + 2 \text{ 🥕} = \text{???}$$

Преобразование типов

- **Неявное преобразование** (implicit conversion) – выполняется автоматически.
- **Явное преобразование** (explicit conversion) – выполняется по заданию программиста.

Неявное преобразование типов (implicit conversion)

- К **неявным** относятся преобразования, результат выполнения которых всегда успешен и не приводит к потере точности данных.
- *Неявные преобразования* выполняются автоматически.
 - Если на диаграмме есть переход из типа А в тип В то, выполняется неявное преобразование типов
 - Если нет неявного преобразования то выдается исключение “Cannot implicitly convert type 'int?' to 'int'. An explicit conversion exists (are you missing a cast?)”

Явное преобразование типов (explicit conversion)

- К **явным** относятся разрешенные преобразования, выполнение которых не гарантируется или может привести к потере точности.
- Два способа явного преобразования:
 - Использование операции приведения типов (cast)
`int i = (int) f; // с обрезанием дробной части`
 - Использование стандартного класса `Convert`
`int i = Convert.ToInt32(f); // с округлением до ближайшего целого`

Неявное и явное преобразование

```
// Error: no conversion from int to short
```

```
int x=5, y=6;
```

```
short z = x + y;
```

```
int a = 5;
```

```
float b = 1.5F;
```

```
b = a;
```

```
// нужно явное преобразование (кастинг)
```

```
a = (int)b;
```

Неявное преобразование типов на языке Java

- `char c='X';`
- `int code=c;`
- `System.out.println(code);`
- Ответ: **88** (ASCII code of X)

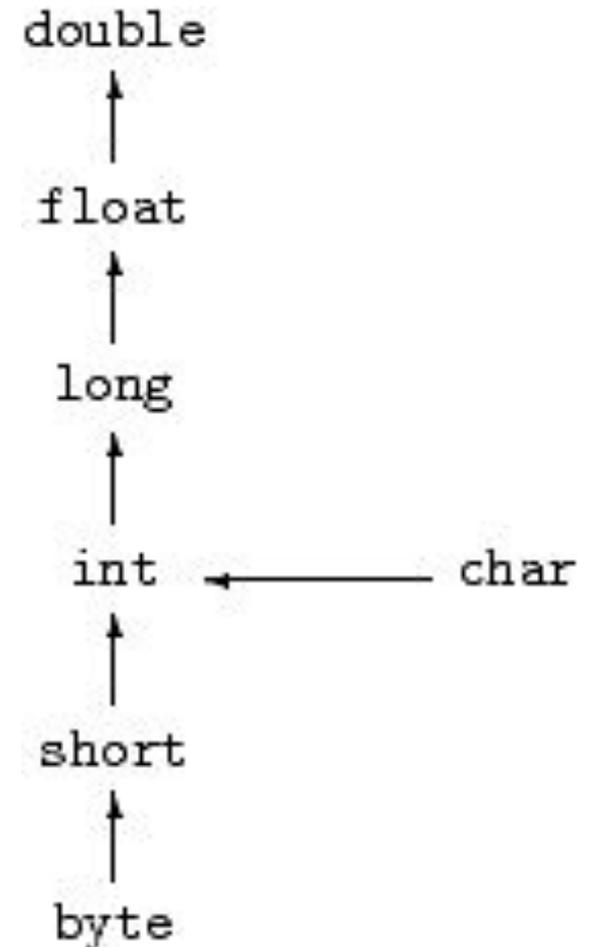


Схема неявного приведение встроенных типов

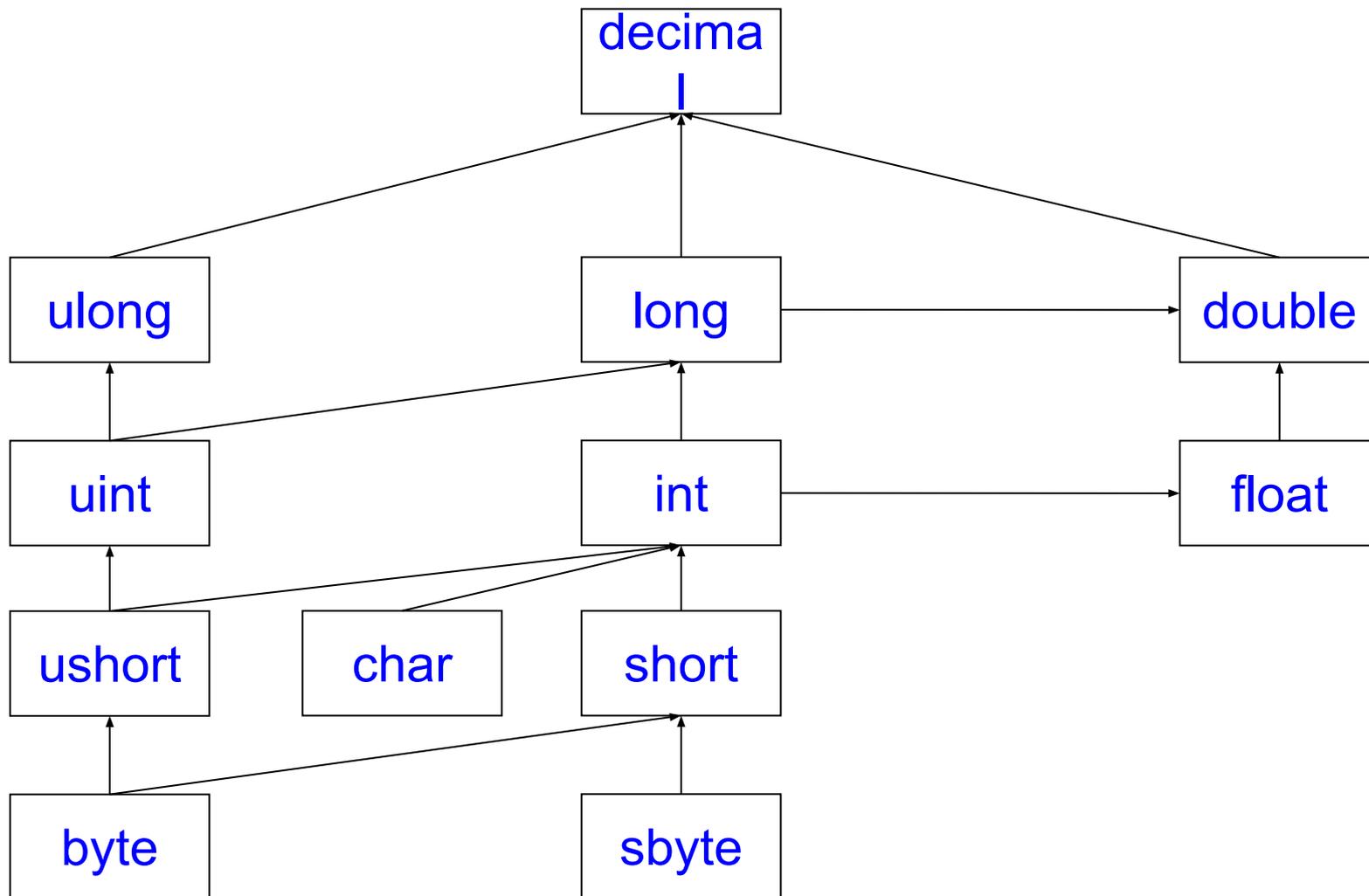
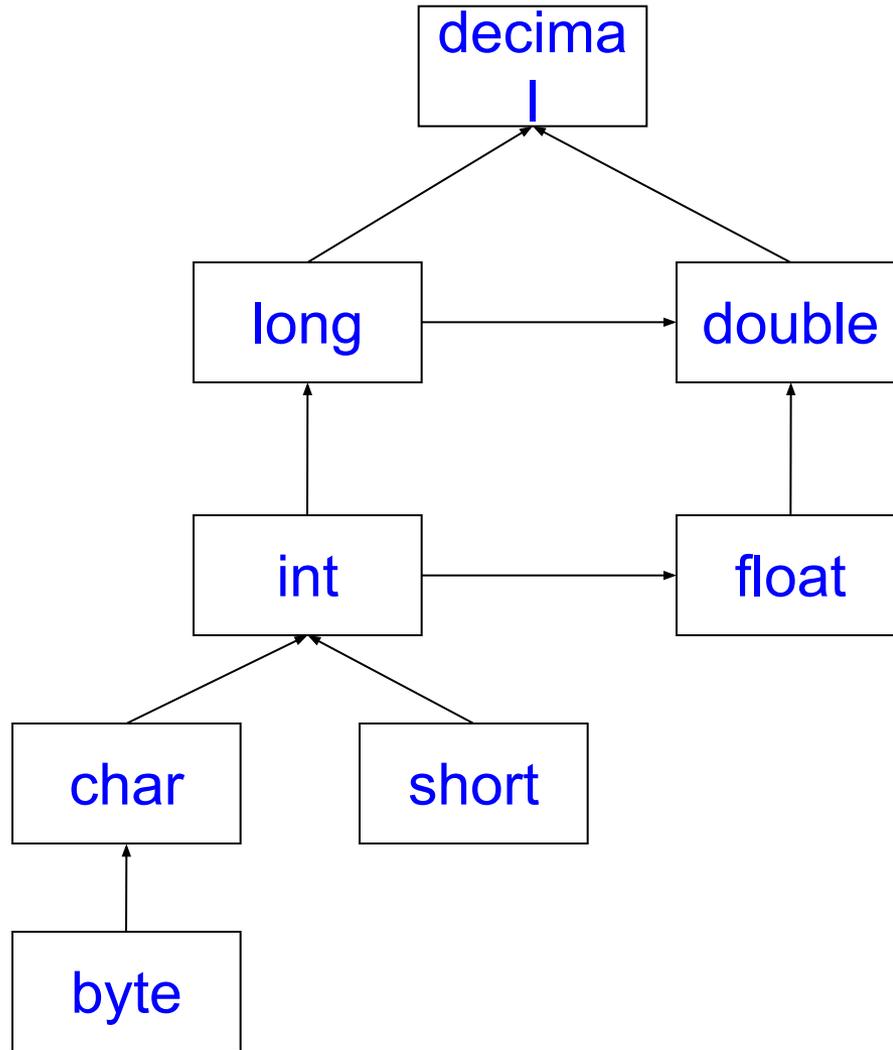


Схема неявного приведение встроенных типов (упрощенная)



Применение диаграммы

- Если на диаграмме задан путь (стрелками) от типа А к типу В, то это означает, что возможно *неявно преобразовать* из типа А в тип В.
 - Например из `short` в `float`
- Все остальные преобразования между подтипами арифметического типа существуют, но являются *явными*.
 - Например из `float` в `int`

Пример приведения встроенных ТИПОВ

```
bool c1 = true;  
int d = c1; // Error! Cannot implicitly convert type 'bool' to 'int'  
d = (int) c1; // Error! Cannot convert type 'bool' to 'int'
```

```
int a = 5;  
float f = 1.9; // Error! Literal of type double cannot be implicitly  
              // converted to type 'float'; use an 'F' //suffix to create a literal  
              // of this type
```

```
float b = 1.9;  
a = (int)b; // a = 1 – отбрасывается дробная часть  
b = a;     // b = 1.0
```

```
decimal d = 2;  
d = (decimal)b;  
d = a;
```

Явное преобразование типа

- Для указания явного преобразования типов используется операция **приведения к типу (кастинг)**, которая имеет высший *приоритет* и следующий вид:
`(type) <выражение>`
- Она задает явное преобразование типа, определенного *выражением*, к типу, указанному в скобках. Например: `int i = (int) 2.99; // i = 2;`
- Если описаны пользовательские типы T и P, и для типа T описано явное преобразование в тип P, то возможна следующая запись:
`T y;
P x = new P();
y = (T) x;`
- Следует отметить, что существуют явные преобразования внутри арифметического типа, но не существует, например, явного преобразования арифметического типа в тип `bool`.
- Например:
`double a = 5.0;
int p = (int)a;
//bool b = (bool)a; // ошибка!!!`
- В данном примере явное преобразование из типа `double` в тип `int` выполняется, а преобразование `double` в тип `bool` приводит к ошибке, потому и закомментировано.

Преобразование типов с помощью класса `Convert`

- Можно задать явным образом требуемое преобразование, используя специальные методы преобразования, определенные в классе `System.Convert`, которые обеспечивают преобразование значения одного типа к значению другого типа (в том числе значения строкового типа к значениям встроенных типов).
- Класс `Convert` содержит 15 статических методов вида `To<Type>` (`ToBoolean()`,...`ToUInt64()`);
`string s1 = Console.ReadLine();`
`int ux = Convert.ToUInt32(s1);`
- Все методы `To<Type>` класса `Convert` перегружены и каждый из них имеет, как правило, более десятка реализаций с аргументами разного типа.
- Преобразование вещественного к целому типу выполняется с округлением
`float b = 1.5;`
`a = Convert.ToInt32(b); // a=2`
- Есть преобразование логического к целому типу
`bool b = true;`
`a = Convert.ToInt32(b); // a=1`

Пример преобразования типов

```
System.Single f = 0.5F;
```

```
float b = f;
```

```
int a;
```

```
a = (int)f; // с обрезанием дробной части
```

```
a = Convert.ToInt32(f); // с округлением
```

```
string s = "123";
```

```
// a = (int)s;
```

```
a = Convert.ToInt32(s);
```

Преобразование типов из строк с помощью метода **Parse()**

- У всех типов есть статический метод **Parse()**, который выполняет преобразование строки текста в соответствующий формат.
- Для проверки возможности преобразования использовать метод **bool TryParse(x)**, он возвращает **true**, если можно преобразовать иначе **false**

```
static void ParseFromStrings()
{
    Console.WriteLine("=> Data type parsing:");
    bool b = bool.Parse("True");
    Console.WriteLine("Value of b: {0}", b);
    double d = double.Parse("99.884");
    Console.WriteLine("Value of d: {0}", d);
    int i = int.Parse("8");
    Console.WriteLine("Value of i: {0}", i);
    char c = Char.Parse("w");
    Console.WriteLine("Value of c: {0}", c);
    Console.WriteLine();
}
```

Операция присваивания

- В C# присваивание является операцией, которая может использоваться в выражениях. В выражении, называемом множественным *присваиванием*, списку переменных присваивается одно и то же значение.
 - Например: $x = y = z = w = (u+v+w)/(u-v-w);$
- При присвоении переменных разного типа выполняется преобразование типа правого операнда к типу левого операнда.
- Т.е. компилятор пытается выполнить преобразование типа переменной стоящей справа в тип переменной, стоящей слева.

- Присваивание переменной стоящей слева (тип **T**) значения переменной или результата вычисления выражения (типа **T1**) возможно только в следующих случаях:
 1. типы **T** и **T1** совпадают;
 2. тип **T** является базовым (родительским) типом для типа **T1** (в соответствии с наследованием типов);
 3. в определении типа **T1** описано явное или неявное преобразование в тип **T**.
- Так как все классы в языке **C#** – встроенные и определенные пользователем – по определению являются потомками класса **Object**, то отсюда и следует, что переменным класса **Object** можно присваивать выражения любого типа.

Специальные варианты присваивания

В языке C# для двух частных случаев *присваивания* предложен специальный синтаксис:

1. для *присваиваний* вида $x=x+1$; (переменная увеличивается или уменьшается на единицу), используются специальные **префиксные** и **постфиксные** операции "++" и "--".

2. для присваивания вида:

$X = X <operator> (\text{выражение})$; например: $x = x * 2$;

– Для таких присваиваний используется краткая форма записи: $X <operator>= \text{expression}$; например: $x *= 2$;

Арифметические операции

- В языке C# имеются обычные для всех языков *арифметические операции* – "+, -, *, /, %". Все они перегружены.
- Операции "+" и "-" могут быть унарными и бинарными.
- Операция деления "/" над целыми типами осуществляет деление нацело, для типов с плавающей и фиксированной точкой – обычное деление.
- Операция "%" определена над всеми арифметическими типами и возвращает остаток от деления нацело.

```
int a = 10;
```

```
int e = 4;
```

```
a %= e; // или a = a % e; - результат 2
```

- ***Результат выполнения выражения всегда имеет тип.***
- ***Тип результата зависит от типов операндов выражения: самый сложный тип задает тип результата выражения.***

Пример вычислений с различными арифметическими типами

```
public void Sample()
{
    int n = 7, m =3, p, q;
    p= n/m; q= p*m + n%m;
    if (q == n) Console.WriteLine("q = n");
    else Console.WriteLine("q!=n");
    double x=7.2 , y =3, u,v,w;
    u = x/y;    // 2.4
    v= u*y;    // 7.199999999999999993
    w= x % y; // 1.2
    if (v == x) // false
        Console.WriteLine("v = x");
    else
        Console.WriteLine("v!=x");
    decimal d1=7M, d2 =3, d3,d4,d5;
    d3 = d1/d2; // 2.3333333333333333333333333333
    d4= d3*d2; // 6.999999999999999999999999999
    d5= d1%d2; // 1.0
    if (d4 == d1) // false
        Console.WriteLine("d4 = d1");
    else
        Console.WriteLine("d4!=d1");
}
```

```
// небольшое изменение
decimal d1 = 7.2M, d2 = 3, d3, d4, d5;
d3 = d1 / d2;    // 2.4
d4 = d3 * d2;    // 7.2
d5 = d1 % d2;    // 1.2
if (d4 == d1)    // true
    Console.WriteLine("d4=d1");
else
    Console.WriteLine("d4!=d1");
```

Операции инкрементации и декрементации

- Операции *инкрементации* (увеличение на единицу) и *декрементации* (уменьшение на единицу) могут быть
 - префиксными (стоять перед переменной) и
 - постфиксными (стоять после переменной).
- К высшему *приоритету* относятся постфиксные операции $x++$ и $x--$. Префиксные операции имеют на единицу меньший *приоритет*.
- Результатом выполнения, как префиксных, так и постфиксных операций, является увеличение ($++$) или уменьшение ($--$) значения переменной на единицу.
- Для префиксных ($++x$, $--x$) операций результатом их выполнения является измененное значение x , постфиксные операции возвращают в качестве результата значение x до изменения.
Например:

```
int n1, n2, n = 5;  
n1 = n++; // n1 = 5; n = 6;  
n2 = ++n; // n2 = 7; n = 7;
```

Операции отношения

- *Операции отношения* используются для сравнения значений переменных и констант.
- Всего имеется 6 операций отношения: `==`, `!=`, `<`, `>`, `<=`, `>=`.
- Следует обратить внимание на запись операции
 - "равно" – `'=='` (два знака присвоить =) и
 - "не равно" – `'!='`.
- При сравнении ссылочных переменных сравниваются не сами объекты, а ссылки на объекты, если операция сравнения не переопределена.

Логические операции

- В языке *C#* логические операции делятся на две категории:
 - над логическими значениями операндов,
 - над битами операндов.
- По этой причине в *C#* существуют две унарные операции отрицания
 - логическое отрицание, заданное операцией "!", определена над операндом типа `bool`,
 - побитовое отрицание, заданное операцией "~", определена над операндом целочисленного типа, начиная с типа `int` и выше (`int`, `uint`, `long`, `ulong`).
- Результатом операции "~" является операнд, в котором каждый бит заменен его дополнением (0 на 1 и 1 на 0).

Пример логических операций

- Рассмотрим пример:

```
//операции отрицания ~,!
bool b1,b2;
b1 = 2*2==4; b2 !=b1; // b1 = true; b2 = false;
//b2= ~b1; // ошибка !
uint j1 =7, j2;
j2= ~j1; // j2 = 4294967288
//j2 = !j1; // ошибка !
int j4 = 7, j5;
j5 = ~j4; // j5 = -8
```
- В этом фрагменте закомментированы операторы, приводящие к ошибкам.
- В первом случае была сделана попытка применения операции побитового отрицания к *выражению* типа bool, во втором – логическое отрицание применялось к целочисленным данным.
- Обратите внимание на разную интерпретацию побитового отрицания для беззнаковых и знаковых целочисленных типов. Для переменных j5 и j2 строка битов, задающая значение – одна и та же, но интерпретируется по-разному.

Бинарные логические операции

- Операции `&&` и `||` определены только над данными типа `bool`:
 - `&&` – условное **И** (результат `true`, если оба операнда имеют значение `true`);
 - `||` – условное **ИЛИ** (результат `true`, если хотя бы один операнд имеет значение `true`);
- Операции `&&` и `||` называются условными (или краткими), поскольку, будет ли вычисляться второй операнд, зависит от уже вычисленного значения первого операнда.
 - в операции `&&`, если первый операнд равен значению `false`, то второй операнд не вычисляется и результат операции равен `false`.
 - в операции `||`, если первый операнд равен значению `true`, то второй операнд не вычисляется и результат операции равен `true`.
- Такое свойство *логических операций* позволяет вычислить *логическое выражение*, имеющее смысл, но в котором второй операнд не определен.

Пример логических операций

- Например, рассмотрим задачу поиска элемента массива. Заданный элемент в массиве может быть, а может и не быть.
`//Условное And – &&`
`int[] ar= {1,2,3};`
`int search = 7, i = 0;`
`// search – заданное значение`
`while ((i < ar.Length) && (ar[i]!= search)) i++;`
`if(i < ar.Length)`
`Console.WriteLine("Значение найдено");`
`else`
`Console.WriteLine("Значение не найдено");`
- Второй операнд не определен в последней проверке, поскольку индекс элемента массива выходит за допустимые пределы (в C# индексация элементов начинается с нуля). Отметим, что "нормальная" конъюнкция требует вычисления обеих операндов, поэтому ее применение в данной программе приводило бы к формированию исключения в случае, когда образца нет в массиве.

Побитовые операции

- Три бинарные побитовые операции:
 - & – AND (если значения двух бит = 1, то и результирующий бит =1);
 - | – OR (если значения хотя бы одного бита = 1, то и результирующий бит =1);
 - ^ – XOR (исключающее ИЛИ) могут использоваться как с целыми типами выше int, так и с булевыми типами. В первом случае они используются как побитовые операции, во втором – как обычные *логические операции*.

$$a = 01100101_2$$

$$b = 00101001_2$$

тогда

$$a \wedge b = 01001100_2$$

- Иногда необходимо, чтобы оба операнда вычислялись в любом случае, тогда без этих операций не обойтись. Вот пример первого их использования:

```
//Логические побитовые операции And, Or, XOR (&,|,^)
```

```
int k2 = 7, k3 = 5, k4, k5, k6;
```

```
k4 = k2 & k3; k5 = k2 | k3; k6 = k2^k3;
```

Таблицы истинности

- а и b типа bool:

| a | b | a && b |
|-------|-------|--------|
| false | false | false |
| false | true | false |
| true | false | false |
| true | true | true |

| a | b | a b |
|-------|-------|--------|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

| a | b | a ^ b |
|-------|-------|-------|
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | false |

- а и b типа int:

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| a | b | a b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| a | b | a ^ b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Условная операция

- В C# имеется *условный операция*:
(условие) ? <выражение1>:<выражение1>;
- Условием является *выражение* типа `bool`.
- Если условие истинно, то выбирается значение *<выражение1>*, в противном случае результатом является значение *<выражение2>*.
- Например:
`int a = 7, b = 9, max;`
`max = (a>b) ? a:b; // max получит значение 9.`