

# **Информационные технологии и программирование**

## **Лекция 2. Основы языка C#**

### **Содержание лекции:**

- **Виды типизаций**
- **Преобразование типов**
- **Выражения и операции**
- **Ввод и вывод на консоль**
- **Форматированный вывод**
- **Отладка программы**

# Виды типизаций

**Статическая / динамическая** типизация.

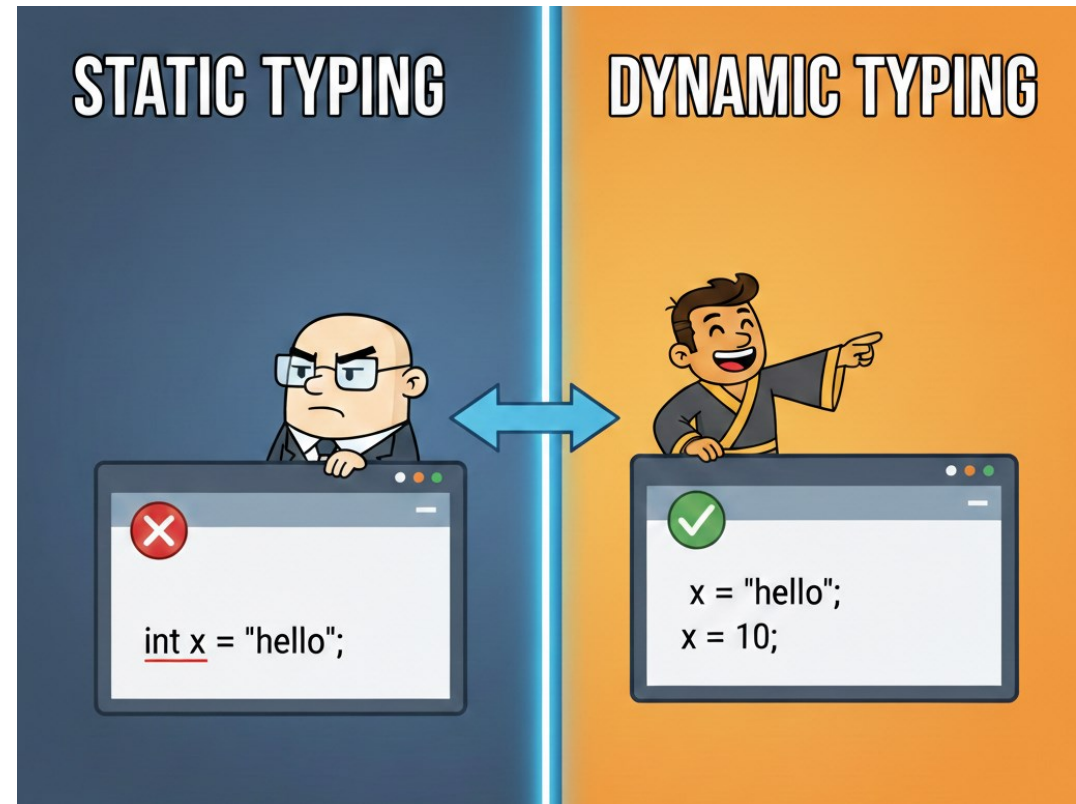
**Статическая** определяется тем, что конечные типы переменных и функций устанавливаются на этапе компиляции.

В **динамической** типизации все типы выясняются во время выполнения программы.

## Примеры:

Статическая: C, Java, C#;

Динамическая: Python, JavaScript, Ruby.



**Сильная / слабая** типизация (иногда строгая / нестрогая).

Язык обладает **сильной** типизацией, если его система типов гарантирует, что программа не может выполнить операцию над данными несовместимого типа, за исключением явно разрешённых преобразований.

Язык обладает **слабой** типизацией, если его система типов допускает выполнение операций над данными различных типов посредством неявных преобразований.

**Примеры:**



Сильная: C#, Java, Python, Haskell, Lisp;

Слабая: C, JavaScript, Visual Basic, PHP.

## Явная / неявная типизация.

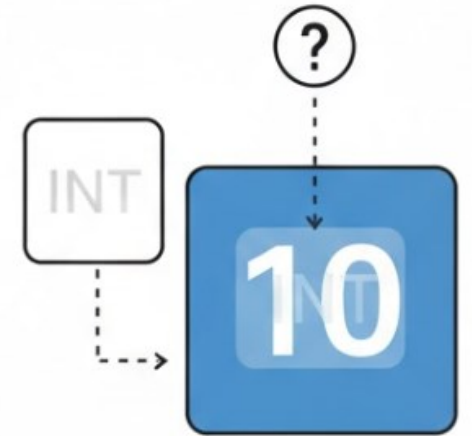
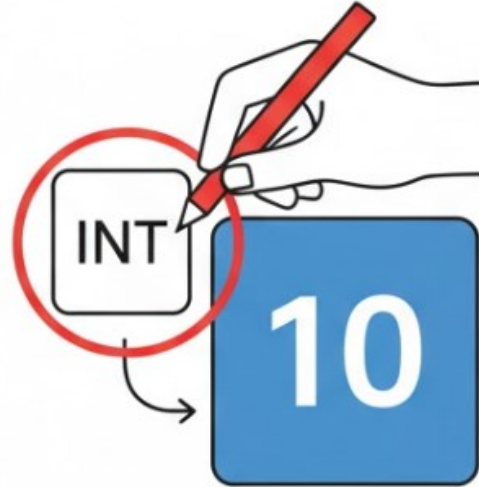
Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно.

Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

### Примеры:

Явная: C++, D, C#

Неявная: PHP, Lua, JavaScript



# Преобразование типов

**Преобразование типов** данных, это приведение одного типа к другому.

Пример **явного** преобразования:

```
double var1 = 5.9;  
int var2 = (int)var1;
```

Преобразование может привести к потере данных. В переменную **var2** будет записано **5**, дробная часть **отбрасывается**.

Чтобы избежать отбрасывания нужно использовать Convert.

```
int var3 = Convert.ToInt32(5.9); //Будет 6
```

Пример **неявного** преобразования:

```
int var4 = 5;  
float var5 = var4; В переменную var5 будет записано 5.
```

## Примечание:

Начиная с версии C# 3.0, в язык была введена поддержка неявной типизации. Ключевое слово **var** позволяет объявлять переменные без явного указания типа, при этом тип переменной автоматически выводится компилятором на основе выражения в правой части оператора присваивания.

```
var floatVar = 5.2f; //тоже самое, что и  
float floatVar = 5.2f;
```

```
var strVar = "Строка"; //тоже самое, что и  
string strVar = "Строка";
```

```
var intVar = 12; //тоже самое, что и  
int intVar = 12;
```

**var — это НЕ динамический тип!** Это просто сокращение. Переменная всё равно имеет строгий тип.

**Константами** называются объекты данных, которые не изменяют своего значения на всём времени выполнения программы.

```
const float myFloatVariable = 10.2f;
```

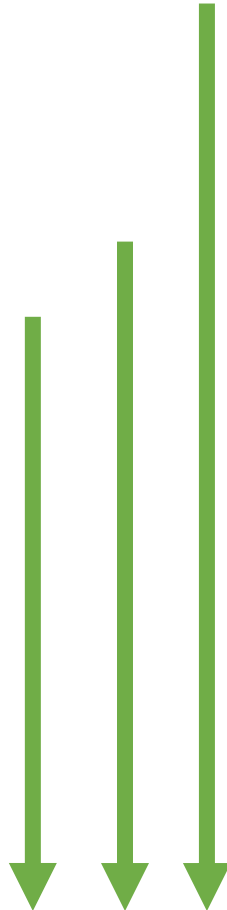
**Область видимости**, или **контекст переменной** — это часть кода, в пределах которого доступна данная переменная.

### Область видимости

myIntVariable

myFloatVariable  
myCharVariable

intMax  
dblMax



```
int myIntVariable;  
myIntVariable = 11;
```

```
const float myFloatVariable = 10.2f;  
char myCharVariable = 'q';
```

```
int intMax = int.MaxValue;  
double dblMax = double.MaxValue;
```

# Выражения и операции

**Операнд** – аргумент операции, т.е. данные, на которые действует операция.

**Операция** – действие (совокупность действий) выполняемое над данными.

**Выражение** – объединение операндов с помощью операций.

Операнд1 @ Операнд2 @ ... ОперандN

где @ - знак операции.

Операции разделяются по **арности** (количество операндов):

- унарные (или одноместные – один операнд);
- бинарные (двуместные – два операнда);
- тернарные (трехместные – три операнда).

Операции упорядочены по приоритету.

Операции одинакового приоритета выполняются в очередности слева направо, кроме операций присваивания, которые выполняются справа налево. Изменить очередность операции в выражении можно с помощью круглых скобок.

Арифметические.....	+ - * / % ++ --
Логические (булевы) .....	! ~ &&    true false
Битовые.....	&   ^
Слияния строки.....	+
Сдвиговые .....	<< >>
Отношения .....	== != < > <= >=
Замещения.....	= += -= *= /= %= &=  = ^= <<= >>= ??
Доступа к элементу .....	.
Индексации .....	[]
Приведения типа.....	()
Выбор по условию.....	? :
Конкатенация и удаление делегата .....	+ -
Создания объекта.....	new
Типа информации.....	as is sizeof typeof
Управление исключениями по переполнению.....	checked unchecked
Адресации и разадресации .....	* -> [] &

Знак операции	Приоритет	Аргументность	Действие	Примечание
- , +	0	1	Смена знака (унарные минус, плюс)	—
!	0	1	Логическая НЕ	Для булевого операнда
~	0	1	Инверсия	Для целочисленных операндов
++	0	1	Инкремент	Префиксная операция выполняется до использования переменной, постфиксная – после
--	0	1	Декремент	
*	1	2	Умножение	—
/	1	2	Деление	
%	1	2	Остаток от деления	Первый операнд может быть вещественным

```
var var1 = 5;
Console.WriteLine(++var1); //6
Console.WriteLine(var1++); //6, var1 = var1 + 1;
но после выполнения метода 7
```

+, -	2	2	Сложение, вычитание	—
<<	3	2	Сдвиг влево	Для целочисленных операндов
>>	3	2	Сдвиг вправо	—
<	4	2	Отношение меньше	Результат – значение булевого типа (true или false)
>	4	2	Отношение больше	
<=	4	2	Отношение меньше или равно	
>=	4	2	Отношение больше или равно	
==	5	2	Отношение равенства	

```

uint value = 15; // 00001111
uint doubled = value << 1; // 00011110 = 30
uint shiftFour = value << 4; // 11110000 = 240
var var1 = 5;
bool b1 = var1 == 5; // true

```

Знак операции	Приоритет	Арность	Действие	Примечание
<code>!=</code>	5	2	Отношение неравенства	Результат – значение булевого типа ( <code>true</code> или <code>false</code> )
<code>&amp;</code>	6	2	Битовая И	Для целочисленных операндов
<code>^</code>	7	2	Битовая исключающая ИЛИ	
<code> </code>	8	2	Битовая ИЛИ	
<code>&amp;&amp;</code>	9	2	Логическая И	Для булевого операнда
<code>  </code>	10	2	Логическая ИЛИ	Для булевого операнда

```

var var1 = 5;
var var2 = 7;
bool b = var1 >= 3 && var2 < 11;
//true

```

```

int c = var1 & var2;
//    0000 0101
// & 0000 0111
// = 0000 0101 = 5

```

НЕ ПУТАТЬ **&** и **&&** - это РАЗНЫЕ операторы!

??	11	2	Логического замещения	Первый операнд проверяется на null, и если не равен, его значение возвращается, в противном случае возвращается значение второго операнда
? :	12	3	Проверка или выбор по условию	
=	13	2	Присвоение	
*=		2	Умножение с замещением	<pre> var var1 = 5; var1 += 7; //var1 = var1 + 7; // будет 12 </pre>
/=		2	Деление с замещением	
+=		2	Сложение с замещением	
-=		2	Вычитание с замещением	

<<=		2	Сдвиг влево с за- мещением	—
>>=		2	Сдвиг вправо с за- мещением	
&=		2	Битовая И с заме- щением	
^=		2	Битовая исключа- ющая ИЛИ с заме- щением	
=		2	Битовая ИЛИ с за- мещением	

Ошибка студентов №1.

**Пример:**

```
int a = 5;
```

```
int b = 2;
```

```
var c = a / b;
```

Студенты ждут 2.5, а получают 2, т.к. при для целых типов дробной части не бывает и она отбрасывается.

Для 2.5 нужно писать (**double**)a / b. Или нужно, чтобы хотя бы один из операндов был типа **double**.

# Математические операции, класс Math

Для выполнения различных математических операций в .NET существует класс **Math**.

## Примеры использования:

// Вводим с консоли число

```
double value = double.Parse(Console.ReadLine());
```

// Далее операции с Math

```
double sqrt = Math.Sqrt(value); // Квадратный корень
```

```
double pow = Math.Pow(sqrt, 5); // Возведение в 5-ю степень
```

```
double abs = Math.Abs(value); // Модуль числа
```

```
double cos = Math.Cos(value); // Косинус
```

```
double asin = Math.Asin(value); // Арксинус
```

```
double floor = Math.Floor(value); // Округление до ближайшего  
целого в меньшую сторону
```

```
double ceiling = Math.Ceiling(value); // Округление до  
ближайшего целого в большую сторону
```

```
double pi = Math.PI; // Число Пи
double exp = Math.Exp(value); // e^value Возведение числа e в
заданную степень
double min = Math.Min(0, value); // минимум двух заданных чисел
double max = Math.Max(0, value); // максимум двух заданных
чисел
double round = Math.Round(value, 3); // Округление с заданной
точностью (3)
```

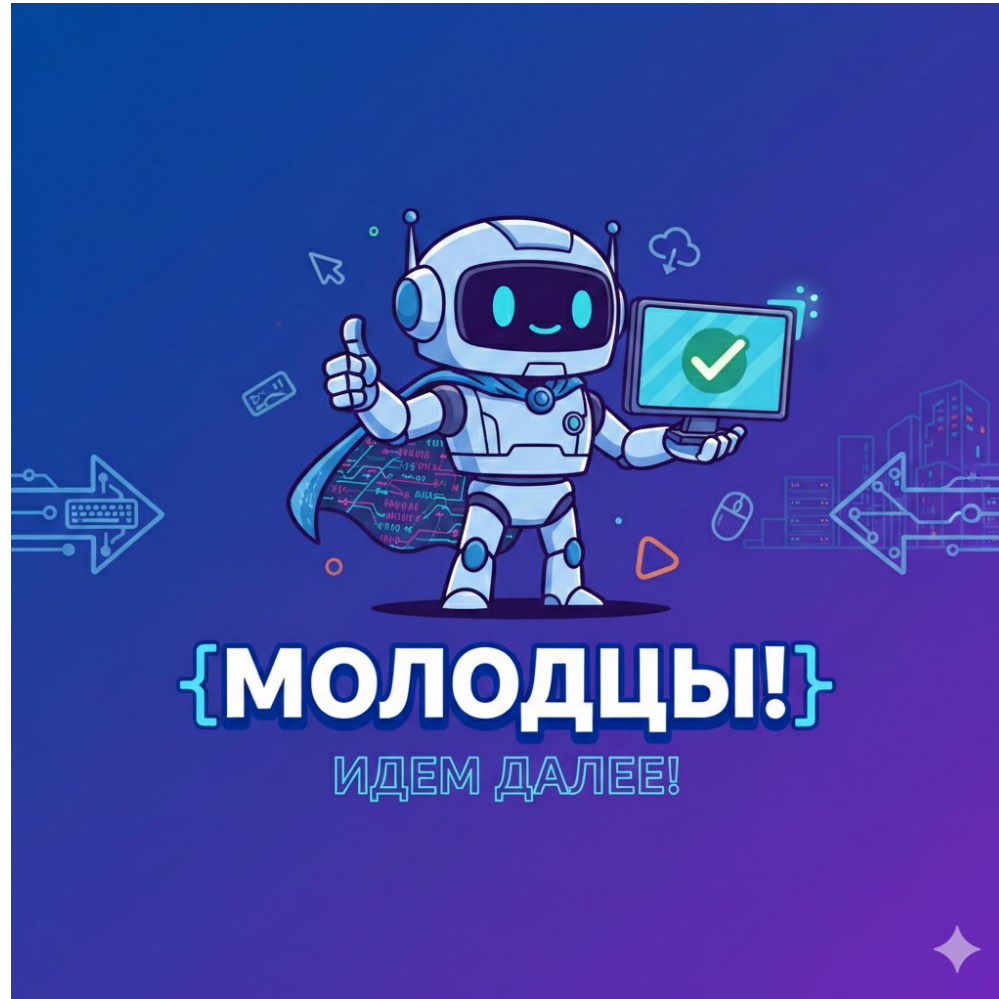
**Округление:** Math.Round в C# по умолчанию использует "Банковское округление" (до ближайшего четного). 2.5 -> 2, 3.5 -> 4. Это шокирует новичков, но об этом надо знать.

```
double clamp = Math.Clamp(value, 1, 5); // Ограничивание
значения диапазоном
double sign = Math.Sign(value); // 1 если value > 0, -1 если <
0, 0 если = 0
```

Ошибка студентов №2.

**Значок ^.** В математике это степень, в C# — XOR.

Хотите степень? Используйте `Math.Pow()`. Символ ^ делает другое!



## Ввод и вывод на консоль

Для вывода на консоль используется конструкция:

```
Console.WriteLine("Что то");
```

Значение, которое стоит внутри скобок, автоматически приводится к строковому виду:

```
Console.WriteLine(123); Вывод: «123»
```

Чтобы запросить ввод данных с консоли используется конструкция:

```
string str = Console.ReadLine();
```

Данное выражение всегда возвращает тип **string**.

Но, что если мы вводим число и хотим получить число?

```
int number = int.Parse(Console.ReadLine());
```

**int.Parse** преобразует строку в целое число, но если строку преобразовать нельзя, то программа завершится с ошибкой!

Но, что если ошибка нам не нужна, а пользователь может ввести не число и надо это как то обработать?

```
bool isNumber = int.TryParse(Console.ReadLine(), out int number);
```

Подробнее выражение **out** мы изучим позже, сейчас нужно просто запомнить.

В данном случае в переменной **isNumber** будет лежать **True**, если строку удалось преобразовать в число и **False** – если преобразовать не удалось. А в переменной **number** будет лежать полученное число в первом случае и значение по умолчанию т.е. 0 во втором.

Аналогично существуют методы:

```
double.Parse(); double.TryParse();  
float.Parse(); float.TryParse(); и т.д.
```

# Форматированный вывод

**Рассмотрим на примерах.**

У нас есть некоторая переменная `int value = 5;`

и мы хотим получить строку вида: «**Васе value лет**» («**Васе 5 лет**»),

**вопрос:** как подставить значение из переменной `value` в строку?

Для этого есть ряд способов:

`string str = «Васе » + value + « лет»;` **Конкатенация строк (УСТАРЕЛ!)**

**Недостатки:** при каждом вызове оператора конкатенации «+», в памяти выделяется место под новую строку, новая строка записывается в новую ячейку памяти, а старая строка уничтожается.

Много «+» => очень медленная работа программы.

## Продолжение примера

мы хотим получить строку вида: «**Васе value лет**»,

**вопрос:** как подставить значение из переменной `value` в строку?

```
int value = 5;  
string str = string.Format(«Васе {0} лет», value); (ТОЖЕ УСТАРЕЛ!)
```

Примечание:

Если строку нужно вывести в консоль, то `string.Format` можно не писать:

```
Console.WriteLine(«Васе {0} лет», value);
```

**И самый удобный способ:**

```
string str = $«Васе {value} лет»; Интерполяция строк (знак доллара)
```

## Пример 2:

Мы хотим получить строку вида: «**Васе age лет, у него money рублей**»

```
int age = 25;
```

```
int money = 20000;
```

```
string str = string.Format(«Васе {0} лет, у него {1} рублей»,age,money);
```

```
string str = $«Васе {age} лет, у него {money} рублей»;
```

**Вопрос к аудитории.** Что будет, если написать:

```
string.Format(«Васе {0} лет, у него {0} рублей», age, money);
```

```
string.Format(«Васе {1} лет, у него {1} рублей», age, money);
```

```
string.Format(«Васе {1} лет, у него {0} рублей», money, age);
```

### Пример 3:

Мы хотим получить строку вида: «**Vase age лет, у него money рублей**»,  
но тип **money** вещественный и нам нужно только 2 знака.

```
int age = 25;
```

```
double money = 2.15367521E4; //21536.7521
```

```
string str = string.Format(«Vase {0} лет, у него {1 : f2} рублей»,age,money);
```

```
string str = $«Vase {age} лет, у него {money : f2} рублей»;
```

d – целые числа; f – дробные; c – денежный формат;

e – экспоненциальная форма записи;

g – выбирает наиболее короткий из f и e.

ИЛИ

```
string str = $«Vase {age} лет, у него {money : 0.00} рублей»;
```

Получим строку вида: «**Vase 25 лет, у него 21536.75 рублей**»

# Вопросы

Что получится в результате операций:

```
int i = 0;
```

```
i++;
```

```
Console.WriteLine(i);
```

```
++i;
```

```
Console.WriteLine(i);
```

```
Console.WriteLine(++i);
```

```
int a = 10;
```

```
var b = 3;
```

```
var c = a % b;
```

```
Console.WriteLine(c);
```

```
bool isBusy = false;
```

```
bool e = !isBusy;
```

```
Console.WriteLine(e);
```

Что получится в результате операций:

```
int i = 0;  
i += 10;  
i *= 3;  
bool c = i < 30;  
bool d = i == 30;  
bool e = (i > 0 && i < 20) || (i > 25 && i < 40);  
Console.WriteLine(e);  
i /= 5;  
Console.WriteLine(i);
```

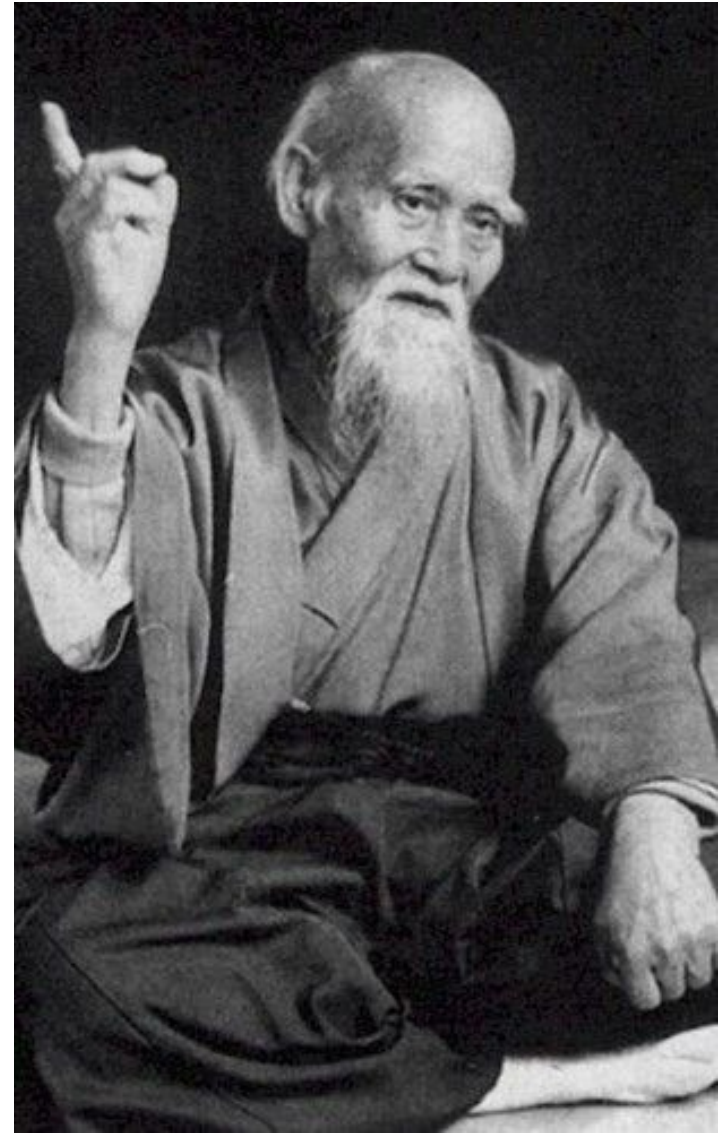
# Отладка программы

**Разработка сложной программы невозможна без отладки.**

**Отла́дка** — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки.

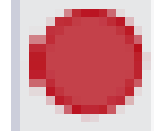
**Зачем это нужно прямо сейчас?**

- Вы видите, как меняются значения переменных на каждом шаге.
- Вы можете остановить время в любой момент.
- Вы понимаете, почему программа выдает 0 вместо 2.5.

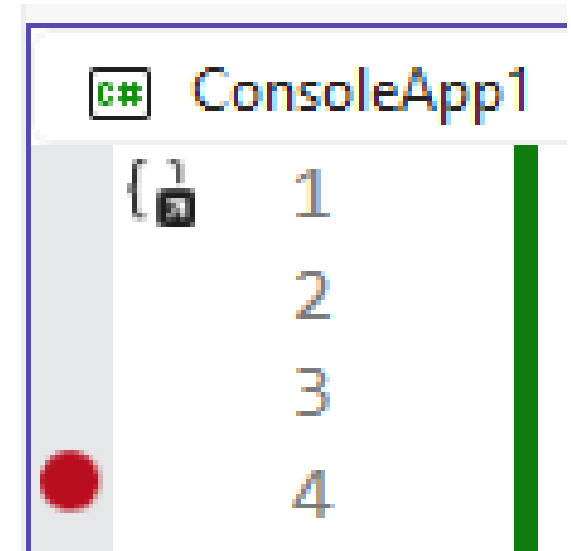


## Инструменты отладчика в Visual Studio:

**Точка останова (Breakpoint):** Красный круг слева от строки.



Здесь программа «замрет».



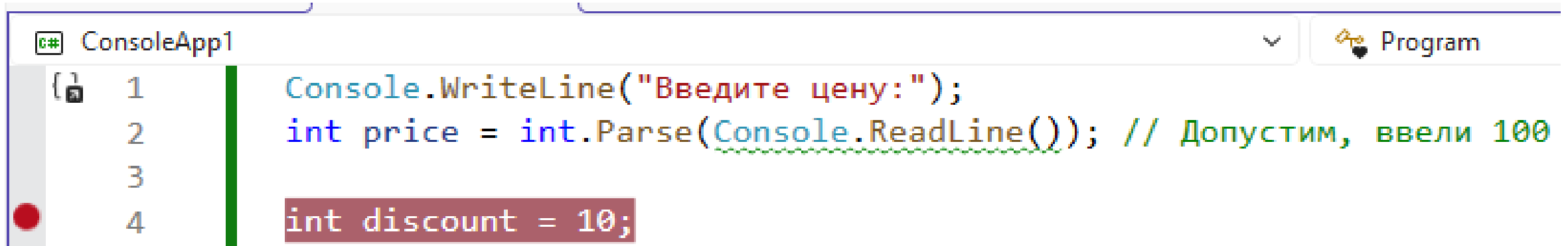
**F5:** Запустить программу до первой точки останова.

**F10:** Сделать один шаг вперед (перейти на следующую строку).

Чтобы поставить точку останова, кликните мышкой на серую полосу слева от номера строки (или нажмите **F9**).

## Что происходит в этот момент?

Программа работает на полной скорости, пока не дойдет до этой строки. Как только «стрелка» выполнения доходит до красного круга, программа встает на паузу.



The screenshot shows the Visual Studio IDE with a C# console application named 'ConsoleApp1'. The code is as follows:

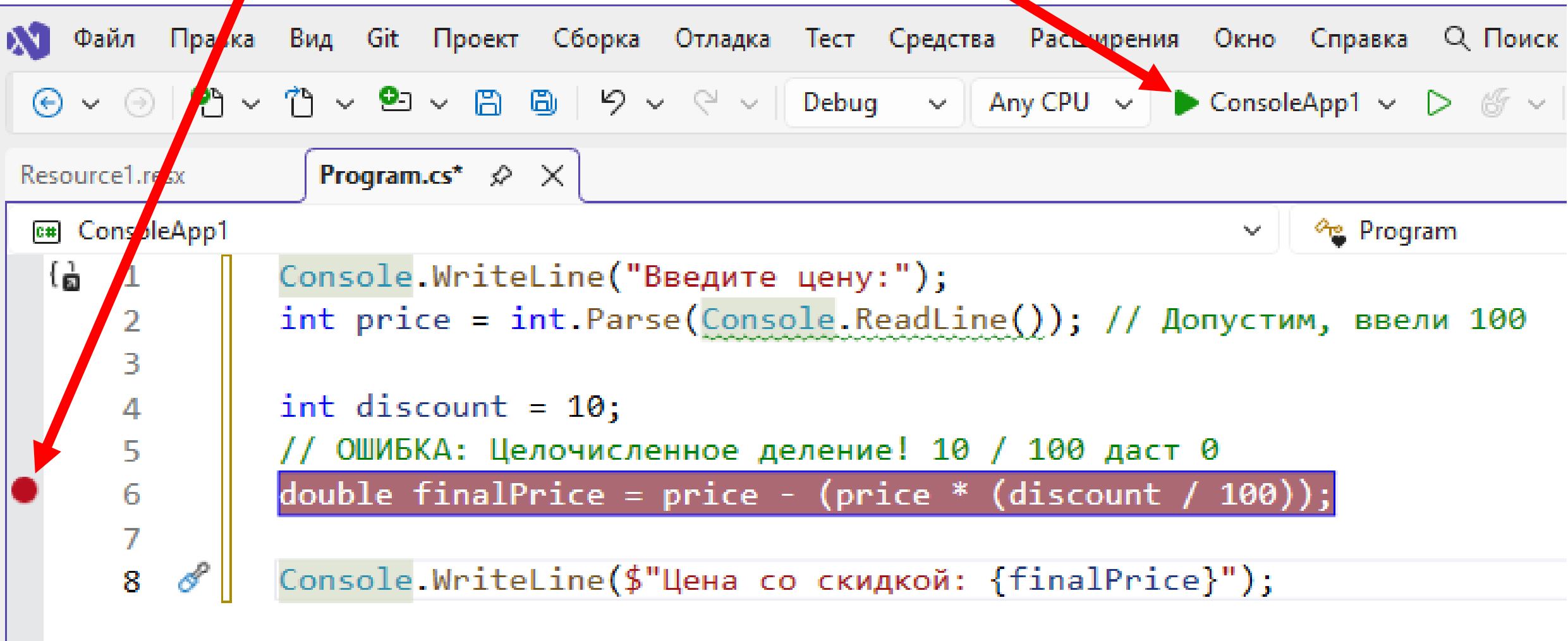
```
1 Console.WriteLine("Введите цену:");  
2 int price = int.Parse(Console.ReadLine()); // Допустим, ввели 100  
3  
4 int discount = 10;
```

A red circle breakpoint is set on line 4. The 'Program' button in the top right corner is highlighted, indicating the current state of the debugger.

*Выполнение замирает ДО того, как строка будет исполнена.*



**Точка остановки**







**Запуск отладки (F5)**



Текущие значения переменных можно посмотреть на вкладке «Локальные»

Локальные

Поиск (Ctrl+E)  

Имя	Значение	Тип
 args	 Просмотр 	string[]
 price	100	int
 discount	10	int
 finalPrice	0	double

## Практический пример (Логическая ошибка)

**Задача:** Посчитать итоговую цену со скидкой 10%.

**Код студента с ошибкой:**

```
Console.WriteLine("Введите цену:");  
int price = int.Parse(Console.ReadLine()); // Допустим, ввели  
100  
  
int discount = 10;  
// ОШИБКА: Целочисленное деление! 10 / 100 даст 0  
double finalPrice = price - (price * (discount / 100));  
  
Console.WriteLine($"Цена со скидкой: {finalPrice}");
```

## Как отлаживать (инструкция):

Поставьте Breakpoint на строке с finalPrice.

Нажмите **F5**. Введите цену 100.

Когда программа замрет, посмотрите в окно **Locals**.

Вы увидите: price = 100, discount = 10.

Нажмите **F10**, чтобы выполнить строку.

Опа! finalPrice стало равно 100.0. Почему не 90?

Вывод: Вы сразу видите, что часть формулы (10 / 100) превратилась в 0.

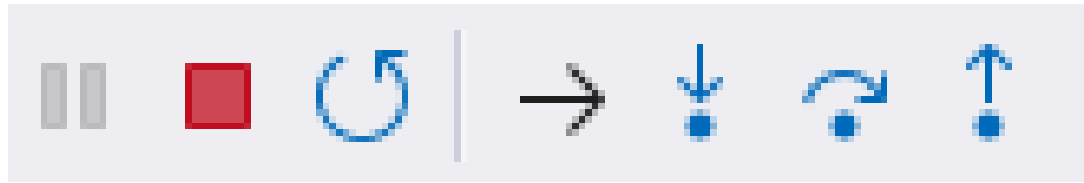
Энное деление! 10 / 100 даст 0

```
price - (price * (discount / 100)); ▶
```

discount / 100 | 0

```
"Цена со скидкой: {finalPrice}");
```

# Управление шагами



## Кнопка

**F5 (Продолжить)**

**F10 (Шаг с обходом)**

**Shift + F5**

## Действие

Бежать до следующей точки останова.

Перейти на ровно одну строку вниз.

Полностью остановить программу.

## Зачем нужно

Пропустить кусок кода, в котором мы уверены.

Медленно идти по алгоритму и смотреть изменения.

Если мы уже поняли, в чем ошибка, и хотим её исправить.