Лабораторная работа №6. Обработка ошибок и отладка программ. Создание консольного калькулятора

Цель работы:

Освоить:

- Основы **обработки ошибок** в C# с использованием конструкции try-catch.
- Базовые навыки **отладки программ** с помощью средств IDE (Visual Studio, Visual Studio Code, JetBrains Rider).
- Разработку простых консольных приложений (на примере калькулятора).

Шаг 1. Теоретическая часть

В Visual Studio 2022 создайте новый консольный проект с именем **ErrorHandlingCalculator.**

Во время выполнения программы могут возникать ошибки, которые сложно или невозможно предугадать заранее. Такие ситуации называют **исключениями** (exceptions). Примеры:

- Деление на ноль.
- Неверный формат пользовательского ввода.
- Потеря сетевого соединения.
- Отсутствие файла и т.д.

Для обработки исключений в языке C# предусмотрена конструкция **try...catch...finally**.

Общая структура

```
try
{
    // Код, в котором может возникнуть исключение
}
catch (ExceptionType ex)
{
    // Блок обработки исключения
}
finally
{
    // Блок, выполняющийся всегда
}
```

- **try** — здесь пишется основной код, в котором потенциально может возникнуть ошибка.

- **catch** — перехватывает ошибку определённого типа и выполняет соответствующие действия.
- **finally** — выполняется в любом случае: произошла ошибка или нет. Используется, например, для освобождения ресурсов.

Пример с делением на ноль:

```
int x = 5;
int y = x / 0;
Console.WriteLine($"Результат: {y}");
Console.WriteLine("Конец программы");
```

При запуске возникнет исключение **System.DivideByZeroException**, и выполнение программы прервётся:

```
Unhandled exception. System.DivideByZeroException: Attempted
 to divide by zero.
   at Program.<Main>$(String[] args) in C:\Users\Leontev\sou
rce\repos\ErrorHandlingCalculator\Program.cs:line 2
```

И в этом случае единственное, что нам остается, это завершить выполнение программы.

Чтобы избежать подобного аварийного завершения программы, следует использовать для обработки исключений конструкцию **try...catch...finally**. Так, перепишем пример следующим образом:

```
try {
    int x = 5;
    int y = x / 0;
    Console.WriteLine(y);
} catch {
    Console.WriteLine("Возникло исключение!");
} finally {
    Console.WriteLine("Блок finally");
}
Console.WriteLine("Конец программы");
```

Поведение:

- Ошибка будет обработана.
- Программа завершится корректно.

Мы также можем обрабатывать несколько ошибок:

```
try {
    int x = Convert.ToInt32(Console.ReadLine());
    int y = x / 0;
    Console.WriteLine($"Результат: {y}");
} catch (DivideByZeroException ex) {
    // Обрабатываем ошибку деления на ноль
    Console.WriteLine($"Ошибка: {ex.Message}");
} catch (FormatException ex) {
    // Обрабатываем ошибку неверного формата ввода
    Console.WriteLine($"Ошибка: {ex.Message}");
} catch (Exception ex) {
    // Обрабатываем другие возможные ошибки
    Console.WriteLine($"Произошла ошибка: {ex.Message}");
}

Console.WriteLine("Конец программы");
```

- **DivideByZeroException** — деление на 0.
- **FormatException** — некорректный ввод (например, буквы вместо чисел).
- **Exception** — "ловушка" на все прочие ошибки.

Шаг 2. Практика: Консольный калькулятор с обработкой ошибок

Задача: Создать консольное приложение «Калькулятор», в котором:

- Пользователь вводит два числа и выбирает операцию (+, -, *, /).
- Программа выполняет операцию и выводит результат.
- Все ошибки обрабатываются с помощью конструкции try-catch.

В начале нам нужно считать два числа:

```
Console.Write("Введите первое число: ");
double number1 = Convert.ToDouble(Console.ReadLine());
```

```
Console.Write("Введите второе число: ");
double number2 = Convert.ToDouble(Console.ReadLine());
```

- **Почему не** int**?** Потому что double поддерживает десятичные дроби (например, 5.5 + 2.3).

Далее нам нужно считать операцию, которую введёт пользователь:

```
Console.Write("Введите операцию (+, -, *, /): ");
string? op = Console.ReadLine();
```

Для сохранения результат вычислений, создадим переменную **result**:

```
double result = 0;
```

Используя конструкцию **if/else** выполним необходимые математеические действия и выведем результат:

```
if (op == "+") result = number1 + number2;
else if (op == "-") result = number1 - number2;
else if (op == "*") result = number1 * number2;
else if (op == "/") result = number1 / number2;
else Console.WriteLine("Неизвестная операция!");
```

Для практики вспомним и перепишем код с использованием конструкции **switch/case**:

```
switch (op) {
    case "+": result = number1 + number2; break;
    case "-": result = number1 - number2; break;
    case "*": result = number1 * number2; break;
    case "/": result = number1 / number2; break;
    default: Console.WriteLine("Неизвестная операция!"); return;
}
```

Какие у нас сейчас есть проблемы в нашем коде?

- Если ввести буквы вместо чисел — программа упадёт.
- Если делить на ноль — будет ошибка.

Первое что нам нужно сделать поместить наш код в блок **try**:

```
try {
    switch (op) {
        case "+": result = number1 + number2; break;
        case "-": result = number1 - number2; break;
        case "*": result = number1 * number2; break;
        case "/": result = number1 / number2; break;
        default: Console.WriteLine("Неизвестная операция!"); return;
    }
} catch {
```

На 0 мы знаем, что нельзя делить, поэтому сразу создадим это исключение:

```
case "/":
    if (number2 == 0)
        throw new DivideByZeroException("Нельзя делить на ноль!");
    result = number1 / number2; break;
```

Далее нам нужно создать блок catch, в котором нужно обработать исключения:

- **FormatException** — если ввели не число.
- **DivideByZeroException** — если делим на ноль.
- **Exception** — все остальные ошибки (например, переполнение памяти).

```
} catch (FormatException) {
    Console.WriteLine("Ошибка ввода! Введите корректные числа.");
} catch (DivideByZeroException ex) {
    Console.WriteLine($"Ошибка: {ex.Message}");
} catch (Exception ex) {
    Console.WriteLine($"Произошла непредвиденная ошибка: {ex.Message}");
}
```

И создаём блок **finally** — выполняется **всегда**, даже если была ошибка:

```
} finally {
    Console.WriteLine("Работа калькулятора завершена.");
}
```

Можно ещё добавить бесконечное выполнение нашей программы, чтобы она не закрывалась после завершения.

Для этого добавляем бесконечный цикл **while(true):**

```
while (true)
{
 // Код
}
```

Шаг 3. Преобразование значений

Наиболее распространено преобразование в тип **string**, поэтому все типы включают метод **ToString**, наследуемый ими от класса **System.Object**.

Метод **ToString** преобразует текущее значение любой переменной в текстовое представление. Некоторые типы не могут быть осмысленно представлены в виде текста, поэтому возвращают свое пространство имен и название типа.

Рассмотрим пример преобразования в тип **string**.

Введите операторы для объявления некоторых переменных, преобразуйте их в представление **string** и запишите их в консоль:

```
int number = 12;
Console.WriteLine(number.ToString());
bool boolean = true;
Console.WriteLine(boolean.ToString());
DateTime now = DateTime.Now;
Console.WriteLine(now.ToString());
object me = new();
Console.WriteLine(me.ToString());
```

Запустите код и проанализируйте результат:

```
12
True
08.07.2025 13:26:00
System.Object
```

Вторым по популярности считается преобразование из строковых переменных в числа или значения даты и времени.

Здесь вместо метода **ToString** выступает метод **Parse**. Только несколько типов имеют метод **Parse**, включая все числовые типы и тип **DateTime**.

Рассмотрим пример.

Добавьте следующие операторы, чтобы разобрать целое число и значение даты и времени из строк, а затем запишите результат в консоль:

```
int age = int.Parse("27");
DateTime birthday = DateTime.Parse("4 July 1980");
Console.WriteLine($"I was born {age} years ago.");
Console.WriteLine($"My birthday is {birthday}.");
Console.WriteLine($"My birthday is {birthday:D}.");
```

Запустите код и проанализируйте результат:

```
I was born 27 years ago.
My birthday is 04.07.1980 0:00:00.
My birthday is 4 июля 1980 г..
```

Ошибки при использовании метода Parse

Существует одна проблема с методом Parse, которая заключается в том, что он выдает ошибку, если строка не может быть преобразована.

Добавьте следующий оператор, чтобы попытаться превратить строку, содержащую буквы, в целочисленную переменную:

```
int count = int.Parse("abc");
```

Запустите код и проанализируйте результат:

```
Unhandled exception. System.FormatException: The input string 'abc'
 was not in a correct format.
   at System.Number.ThrowFormatException[TChar](ReadOnlySpan`1 valu
e)
   at System.Int32.Parse(String s)
   at Program.<Main>$(String[] args) in C:\Users\Paltos\VsualStudio
Project\ErrorHandlingCalculator\Program.cs:line 1
```

Вы увидите трассировку стека.

Шаг 4. Метода TryParse

Чтобы избежать ошибок, вместо **Parse** вы можете использовать метод **TryParse**.

Он пробует преобразовать исходную строку и, если это возможно, возвращает **true**. В противном случае возвращает **false**.

Ключевое слово **out** требуется для того, чтобы разрешить методу **TryParse** устанавливать переменную **count** при проведении преобразования.

Рассмотрим пример.

Замените объявление **int count** операторами, чтобы применить метод **TryParse**, и попросите пользователя ввести число:

```
Console.Write("Какой максимальный балл по дисциплинам? ");
string? input = Console.ReadLine();
if (int.TryParse(input, out int count)) {
    Console.WriteLine($"Это {count} баллов.");
} else {
    Console.WriteLine("Некорректный ввод.");
}
```

Запустите код, введите число 100 и проанализируйте результат:

```
Какой максимальный балл по дисциплинам? 100
Это 100 баллов.
```

Снова запустите код, введите **сто** и проанализируйте результат:

```
Какой максимальный балл по дисциплинам? сто
Некорректный ввод.
```

Помимо этого, вы можете использовать методы типа **System.Convert** для преобразования значений **string** в другие типы. Однако, как и метод **Parse**, он выдает ошибку, если преобразование невозможно.

Шаг 5. Перехват с помощью фильтров

В оператор **catch** вы можете добавить фильтры, используя ключевое слово **when**:

```
Console.Write("Введите сумму: ");
string? amount = Console.ReadLine();
try {
    decimal amountValue = decimal.Parse(amount);
} catch (FormatException) when (amount.Contains("$")) {
    Console.WriteLine("В суммах нельзя использовать знак доллара!");
} catch (FormatException) {
    Console.WriteLine("Суммы должны содержать только цифры!");
}
```

Шаг 6. Выброс исключений переполнения с помощью оператора checked

Оператор **checked** дает .**NET** команду при возникновении переполнения вызывать исключение, вместо того чтобы позволять ему происходить автоматически, что делается по умолчанию из соображений производительности.

Определим начальное значение переменной **int** как максимальное значение минус один. Затем увеличим его несколько раз, выводя каждый раз значение переменной.

Как только переменная станет больше своего максимального значения, код снова начнет работать от минимального значения, увеличивая каждый раз значение переменной.

Рассмотрим пример.

Добавим операторы для объявления и назначения целого числа, которое на единицу меньше его максимально возможного значения, а затем увеличим его и запишите его значение в консоль три раза:

```
int x = int.MaxValue - 1;
Console.WriteLine($"Initial value: {x}");
x++;
Console.WriteLine($"After incrementing: {x}");
x++;
Console.WriteLine($"After incrementing: {x}");
x++;
Console.WriteLine($"After incrementing: {x}");
```

Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
After incrementing: -2147483648
After incrementing: -2147483647
```

Теперь с помощью блока оператора **checked** научим компилятор предупреждать нас о переполнении, обернув операторы:

```
checked {
    int x = int.MaxValue - 1;
    Console.WriteLine($"Initial value: {x}");
    x++;
    Console.WriteLine($"After incrementing: {x}");
    x++;
    Console.WriteLine($"After incrementing: {x}");
    x++;
    Console.WriteLine($"After incrementing: {x}");
}
```

Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
Unhandled exception. System.OverflowException: Arithmetic operation resulted in an overflow.
   at Program.<Main>$(String[] args) in C:\Users\Leontev\source\repos\ErrorHandlingCalculator\Program.cs:line 7
```

Как и любое другое исключение, нам нужно поместить эти операторы в блок **try** и отобразить пользователю поясняющее сообщение об ошибке:

```
try {
    checked {
        int x = int.MaxValue - 1;
        Console.WriteLine($"Initial value: {x}");
        x++;
        Console.WriteLine($"After incrementing: {x}");
        x++;
        Console.WriteLine($"After incrementing: {x}");
        x++;
        Console.WriteLine($"After incrementing: {x}");
    }
} catch (OverflowException) {
    Console.WriteLine("The code overflowed but I caught the exception.");
}
```

Запустите код и проанализируйте результат:

```
Initial value: 2147483646
After incrementing: 2147483647
The code overflowed but I caught the exception.
```

Шаг 7. Отладка в процессе разработки

Рассмотрим отладку, создав консольное приложение с преднамеренной ошибкой, для отслеживания и исправления которой мы затем будем использовать специальный инструментарий.

Добавьте функцию с преднамеренной ошибкой:

```
static double Add(double a, double b) {
    return a * b; // Преднамеренная ошибка!
}
```

Под функцией **Add** добавьте операторы для объявления некоторых переменных, а затем сложите их вместе с помощью ошибочной функции:

```
double a = 4.5;
double b = 2.5;
double answer = Add(a, b);
Console.WriteLine($"{a} + {b} = {answer}");
Console.WriteLine("Нажмите ENTER чтобы продолжить.");
Console.ReadLine(); // ожидание нажатия клавиши ENTER пользователем
```

Запустите консольное приложение и проанализируйте результат:

```
4,5 + 2,5 = 11,25
Нажмите ENTER чтобы продолжить.
```

В этом коде обнаружена следующая ошибка: 4.5 плюс 2.5 должно в результате давать 7, но никак не 11.25!

Мы воспользуемся инструментами отладки, чтобы расправиться с этой ошибкой.

Установка точек останова и начало отладки

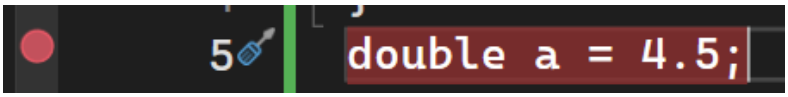
Точки останова позволяют помечать строки кода, на которых следует приостановить выполнение программы, чтобы найти ошибки.

Использование Visual Studio 2022

Установим точку останова, а затем начнем отладку с помощью Visual Studio 2022.

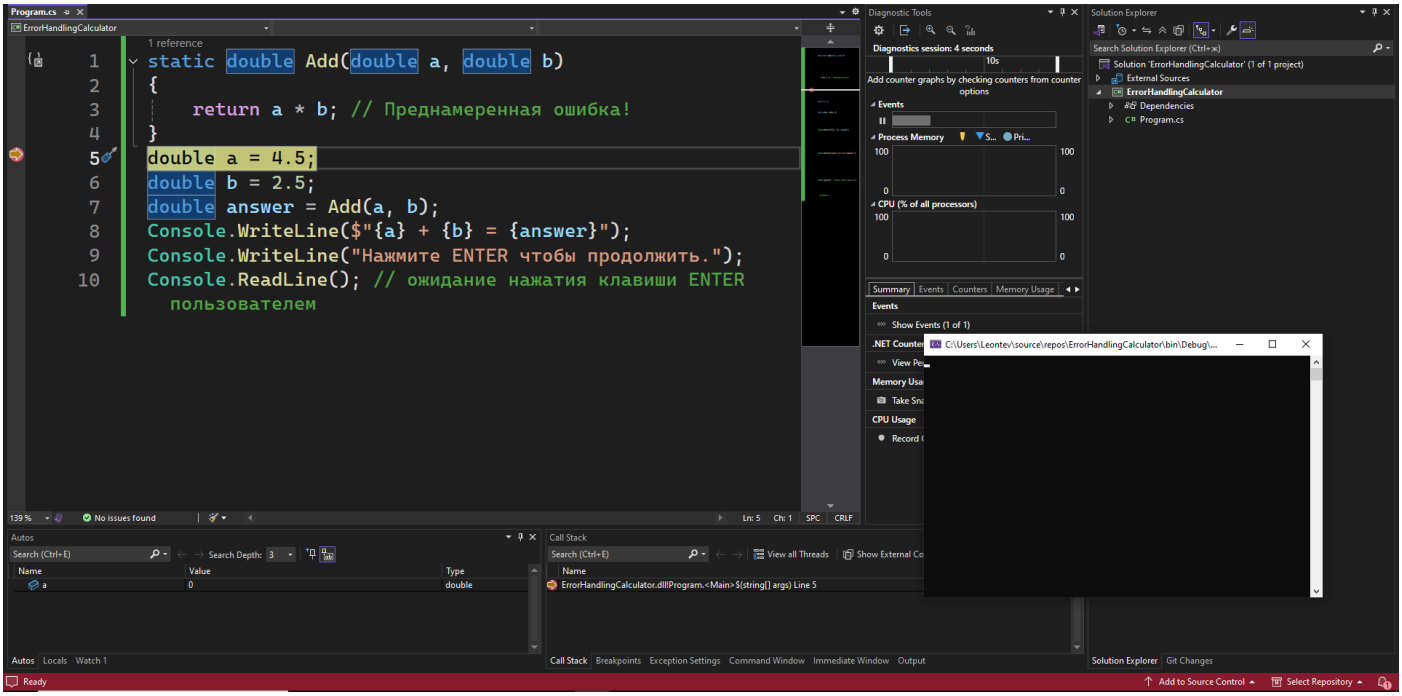
На строке с переменной a, наведите мышь слева от строк, чтобы у вас появилась белая точка и нажмите на неё (или нажмите клавишу **F9** на строке**)**:

Слева на полях появится красный кружок, указывающий на то, что точка останова была установлена:

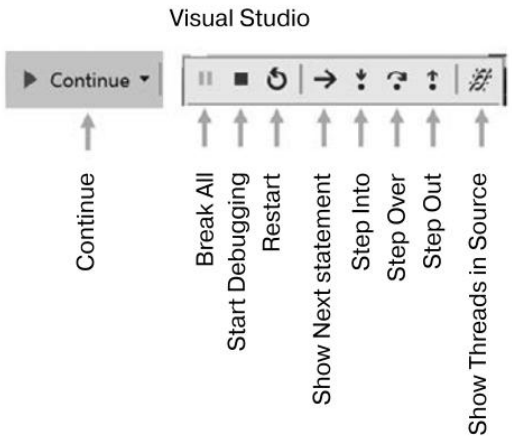


Если на точке останова щелкнуть правой кнопкой мыши, то вы увидите дополнительные команды, например, выключение или изменение параметров точки останова.

Выберите команду меню **Debug** -> **Start Debugging** или нажмите клавишу **F5**. **Visual Studio** запустит консольное приложение, а затем приостановит выполнение при обнаружении точки останова. Это называется режимом приостановки выполнения. Затем появятся следующие дополнительные окна: Locals (Локальные переменные) (отображающие текущие значения локальных переменных), Watch 1 (Отслеживаемые выражения 1) (отображающие любые определеные вами отслеживаемые выражения), Call Stack (Вызов стека), Exception Settings (Настройки исключений) и Immediate Window (Окно интерпретации). Появится также панель инструментов Debugging (Отладка). Строка кода, которая будет выполнена следующей, подсвечивается желтым, и на нее указывает стрелка такого же цвета, размещенная на поле:



Программа Visual Studio 2022 содержит одну кнопку на стандартной панели инструментов, позволяющую запускать или продолжать отладку, и отдельную панель инструментов отладки для остальных средств.



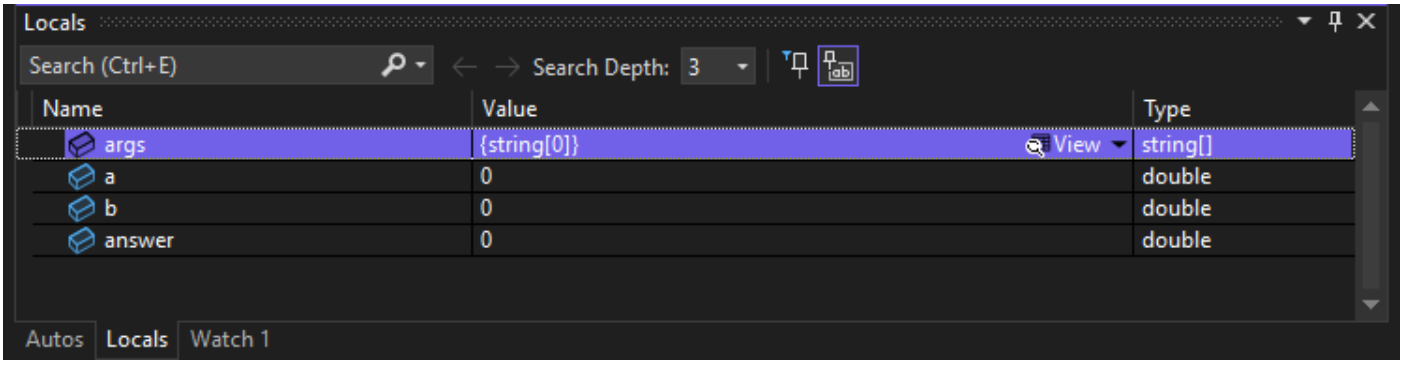
- **Continue / F5** — запускает код с текущей позиции до достижения следующей точки останова.
- **Step Over / F10, Step Into / F11 и Step Out / Shift+F11** (синие стрелки над точками) — различными способами пошагово выполняют код (будет рассмотрено далее).
- **Restart / Ctrl+Shift+F5** или **Cmd+Shift+F5** (круглая стрелка) — завершает выполнение и сразу вновь запускает программу.
- **Stop / Shift+F5** (красный квадрат) — прекращает выполнение сеанса отладки.

Панели отладки

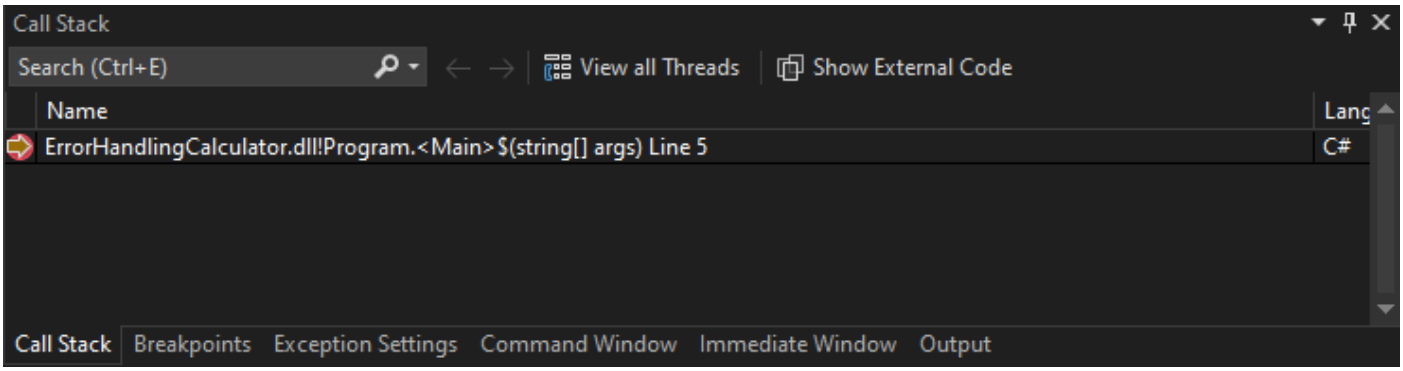
Во время отладки в Visual Studio отображаются дополнительные окна, позволяющие по мере выполнения кода отслеживать полезную информацию, например, переменные.

Ниже перечислены наиболее полезные окна.

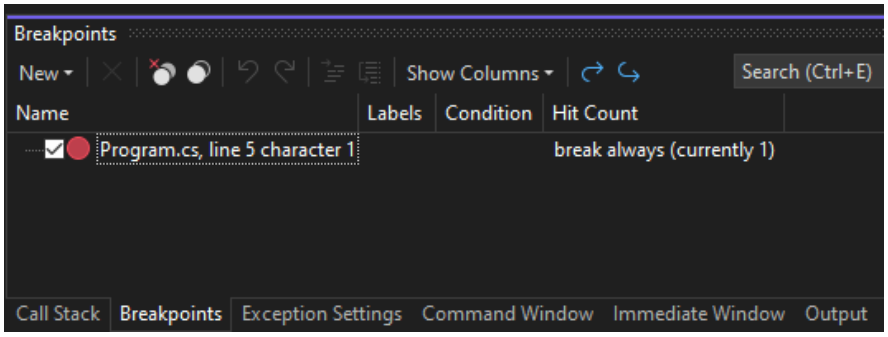
- VARIABLES (Переменные), включая Locals (Локальные), отображает имена, значение и типы всех используемых локальных переменных. Отслеживайте содержимое этого окна, когда отлаживаете свой код.



- WATCH или WATCH 1 (Наблюдение) отображает значения переменных и выражений, которые вы вводите вручную.
- CALL STACK (Стек вызовов) отображает стек вызовов функций.

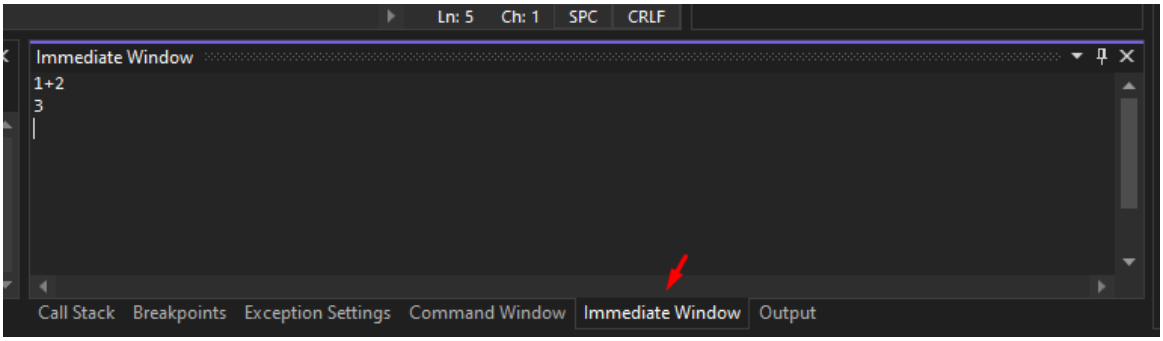


- BREAKPOINTS (Точки останова) отображает все ваши точки останова, что позволяет лучше контролировать их.



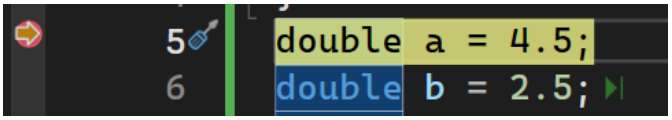
В режиме приостановки в нижней части области редактирования также имеется полезная панель.

DEBUG CONSOLE (Консоль отладки) или Immediate Window (Окно непосредственной отладки) обеспечивает интерактивное взаимодействие с вашим кодом. Вы можете запросить состояние программы, например, введя имя переменной. Здесь вы можете задать вопрос и получить на него ответ. К примеру, чтобы спросить «Сколько будет 1 + 2?», надо набрать 1+2 и нажать клавишу Enter:

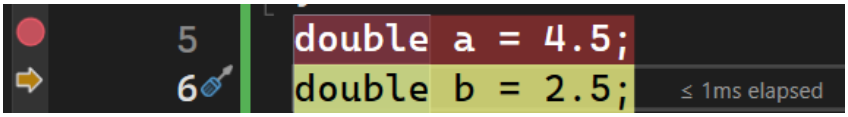


Рассмотрим несколько способов пошагового выполнения кода.

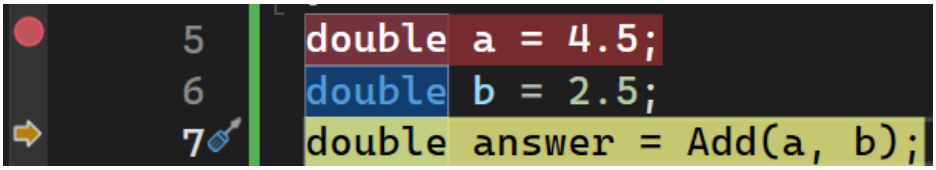
1. Нажмите кнопку **Step Into** на панели инструментов, или нажмите клавишу **F11**.



Желтая подсветка переместится на одну строку кода:



2. Нажмите кнопку **Step Over** на панели инструментов, или нажмите клавишу **F10**, желтая подсветка переместится на одну строку кода:



На данный момент вы можете видеть, что нет никакой разницы между использованием команды **Step Into** или **Step Over**.

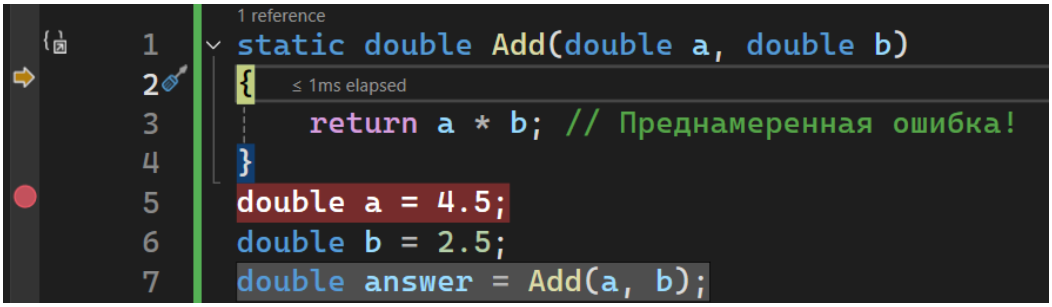
3. Теперь вы окажетесь на строке вызова метода Add.

Разницу между командами **Step Into** (Шагнуть внутрь) и **Step Over** (Перешагнуть) можно увидеть, когда следующим оператором является вызов метода.

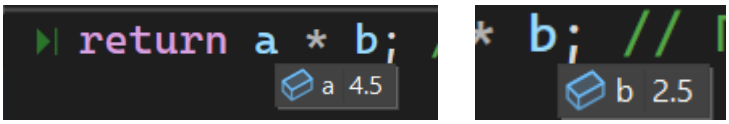
- Если вы нажмете кнопку **Step Into** (Шагнуть внутрь), то отладчик войдет в метод, так что вы сможете пройти по каждой его строке.

- Если вы нажмете кнопку **Step Over** (Перешагнуть), то отладчик выполнит метод целиком. Отладчик не пропустит метод, не выполнив его.

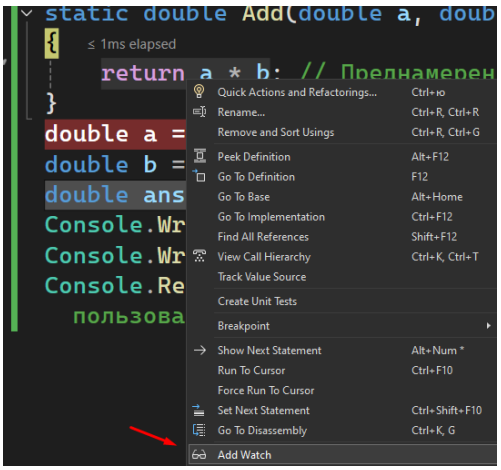
4. Нажмите кнопку **Step Into:**



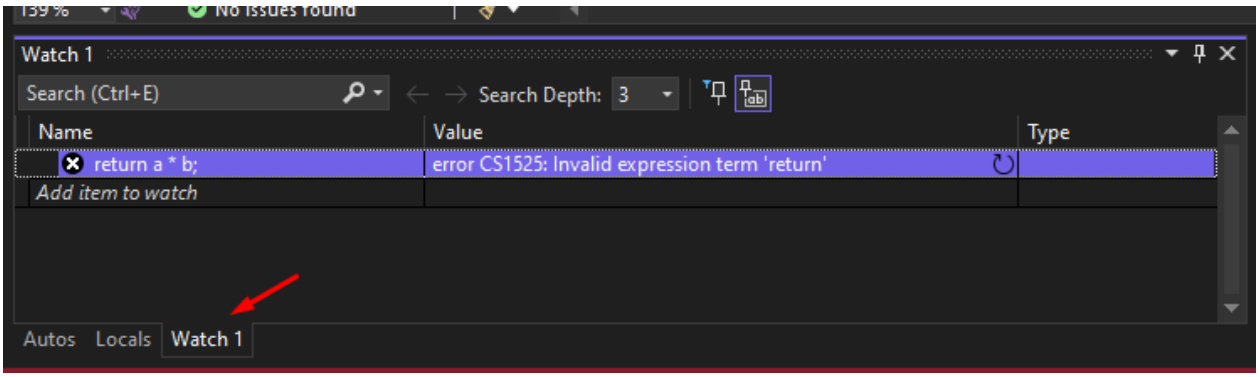
5. Наведите указатель мыши на параметры **a** или **b** в окне редактирования кода и обратите внимание на всплывающую подсказку, показывающую их текущее значение:



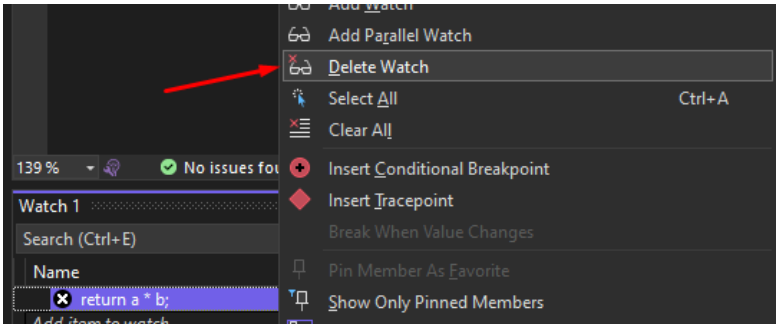
6. Выберите выражение **a * b**, щелкните на нем правой кнопкой мыши и выберите **Add Watch** (Добавить контрольное значение):



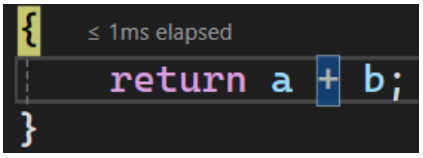
Выражение добавляется в окно **WATCH 1** (Контрольное значение 1), показывая, что эта операция умножает **a** на **b** и выдает результат:



7. В окне **Watch 1 (Контрольное значение 1)** щелкните правой кнопкой мыши на выражении и выберите **Delete Watch** (Удалить контрольное значение):



8. Исправьте ошибку, сменив операцию * на + в функции **Add**:



9. Остановите, перекомпилируйте и перезапустите отладку. Для этого нажмите кнопку с круговой стрелкой **Restart** (Перезагрузить) либо сочетание клавиш **Ctrl+Shift+F5**.

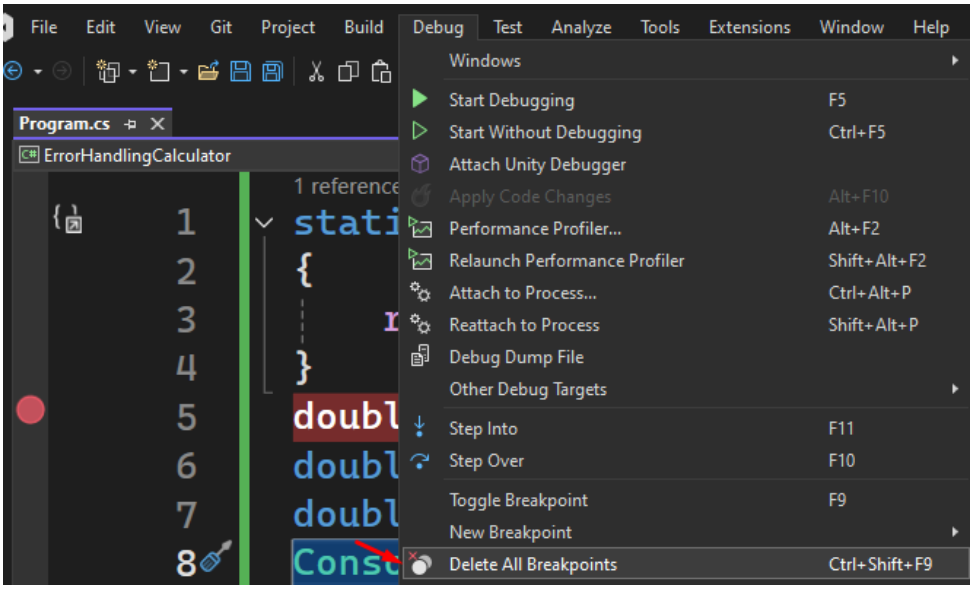
10. Перешагните функцию **(Step Over)**, при этом заметив, что она теперь правильно рассчитывается, затем нажмите кнопку **Continue** (Продолжить) или клавишу **F5**.

Настройка точек останова

Вы можете легко настроить более сложные точки останова.

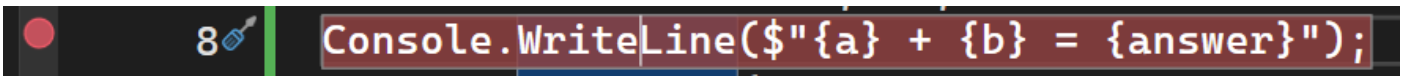
1. Если вы все еще выполняете отладку кода, то нажмите кнопку **Stop** (Стоп) или нажмите комбинацию клавиш **Shift+F5**.

2. Выберите команду меню **Debug –> Delete All Breakpoints**

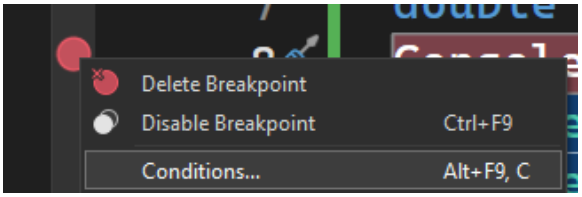


3. Щелкните кнопкой мыши на операторе **Console.WriteLine**, который выводит ответ.

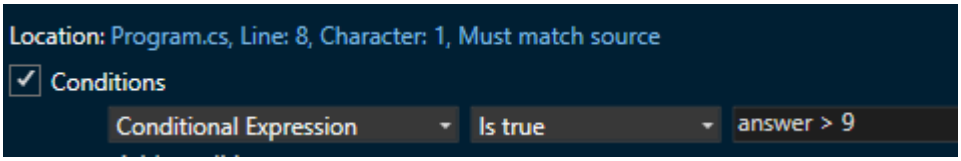
4. Установите точку останова, нажав клавишу **F9**:



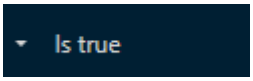
5. Щёлкните правой кнопкой мыши на точке останова и выберите Conditions (Условия).



Введите выражение, например, «переменная answer должна быть больше 9», нажмите **Enter**:



Обратите внимание, что для активации точки останова выражение должно иметь значение true.



6. Начните отладку и обратите внимание, что точка останова не достигнута.

7. Остановите отладку.

8. Измените точку останова или ее условия и установите ее выражение в значение меньше 9.

9. Начните отладку и обратите внимание, что точка останова достигнута.

10. Остановите отладку.

Самостоятельные задания

Задание 1. Расширение функциональности калькулятора

Модифицируйте консольный калькулятор следующим образом:

1. **Добавьте поддержку операции возведения в степень (^)**
   Реализуйте возможность вычисления степени одного числа по основанию другого. Например: 2 ^ 3 = 8.
2. **Организуйте непрерывную работу программы в цикле**
   Калькулятор должен работать до тех пор, пока пользователь не введёт команду exit. После этого программа завершает выполнение.
3. **Выделите каждую арифметическую операцию в отдельный метод**
   Например:

static double Add(double a, double b) { return a + b; }

static double Subtract(double a, double b) { return a - b; }

Это повысит читаемость кода и улучшит его структуру.

Задание 2. Отладка в Visual Studio Code и JetBrains Rider

Практически выполните отладку калькулятора в **двух IDE** — Visual Studio Code и JetBrains Rider.

Ваша задача:

1. Открыть и запустить проект в каждой из IDE.
2. Установить точки останова, выполнить пошаговое выполнение, просмотреть значения переменных.
3. Сделать **скриншоты** с примером отладки:
   - точка останова;
   - текущее значение переменной;
   - окно Output или Debug;
   - стек вызовов (если доступен).
4. Кратко опишите **отличия в процессе отладки** между Visual Studio Code и Rider. Обратите внимание на удобство интерфейса, подсказки, настройку конфигурации и скорость запуска.

Задание 3. Проверка деления с защитой от ошибок

Создайте новую консольную программу, которая:

- Запрашивает у пользователя два числа.
- Выполняет деление, но **предварительно проверяет**, что делитель не равен нулю.
- Если пользователь всё же ввёл ноль, программа должна **не выбрасывать исключение**, а вежливо попросить ввести другое число.

Пример вывода:

Введите делимое: 12

Введите делитель: 0

На ноль делить нельзя. Повторите ввод делителя:

Задание 4. Проверка корректности ввода

Напишите консольную программу, которая:

- Принимает на вход только целые числа.
- Если пользователь вводит текст, буквы, пустую строку — программа **не завершается**, а просит ввести корректное значение.
- Все ошибки должны обрабатываться через try-catch.

  Цикл повторяется, пока не получено допустимое число. После этого программа завершает работу.

Задание 5. Отладка калькулятора с ошибками

1. Возьмите исходный калькулятор из лабораторной и **искусственно добавьте несколько ошибок**:
   - Присваивание неверной переменной.
   - Отсутствие проверки на деление на ноль.
   - Пропущенная обработка формата (например, Convert.ToInt32 без проверки).
2. Затем:
   - Сначала **запустите** его и убедитесь, что он не работает корректно.
   - Проведите **пошаговую отладку** с использованием точек останова.
   - Найдите и исправьте все ошибки.
   - Сделайте скриншоты до и после исправлений.

   Добавьте комментарии в код, указывающие, где была ошибка и как вы её устранили.