

Введение в C#

Обработка ошибок. Работа с процессами



На этом уроке

1. Рассмотрим способы обработки исключительных ситуаций.
2. Научимся использовать собственные исключения.
3. Рассмотрим работу с процессами.

Оглавление

[Обработка исключительных ситуаций](#)

[Блок try-catch](#)

[Оператор finally](#)

[Тип исключительных ситуаций](#)

[Стандартные исключения .NET](#)

[Создание собственных исключений](#)

[Конструкция catch-when](#)

[Процессы](#)

[Определение процесса](#)

[Входные аргументы процесса](#)

[Передача аргументов с помощью Visual Studio](#)

[Коды возврата](#)

[Создание и запуск процесса](#)

[Передача аргументов командной строки](#)

[Чтение стандартного потока вывода](#)

[Запуск файла в ассоциированной программе](#)

[Получение списка процессов](#)

[Практическое задание](#)

[Используемые источники](#)

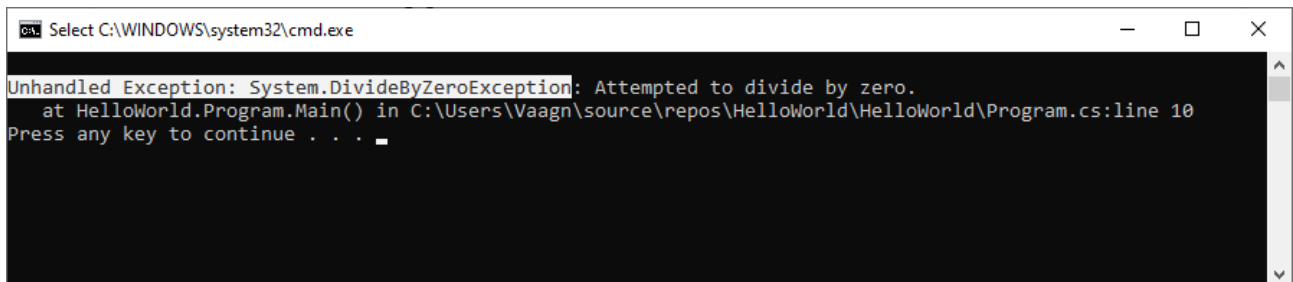
Обработка исключительных ситуаций

Бывают ситуации, когда при выполнении программы возникают ошибки. Например, при передаче файла по сети может неожиданно оборваться сетевое подключение, у пользователя может быть недостаточно прав для записи в указанную директорию, необходимый файл может отсутствовать и так далее. Такие ситуации называются исключительными ([Exception](#)). В языке C# существует возможность обработки таких ситуаций путём использования определённых операторов, конструкций и специальных классов, описывающих исключения.

Если исключительная ситуация не будет обработана в коде программы, при её наступлении программа экстренно завершится. Рассмотрим пример, вызывающий исключительную ситуацию:

```
int x = 10;  
int y = x / 0;
```

Этот код пытается совершить деление на ноль, что приводит к возникновению исключительной ситуации `System.DivideByZeroException`:



The screenshot shows a Windows command prompt window titled "Select C:\WINDOWS\system32\cmd.exe". The window contains the following text: "Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero. at HelloWorld.Program.Main() in C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:line 10 Press any key to continue . . .".

Исключения, приводящие к экстренному завершению программы, называются необработанными ([Unhandled](#)). Далее мы рассмотрим способы обработки исключений.

Блок try-catch

Для обработки исключений используется блок `try-catch`:

```
try  
{  
    int x = 10;  
    int y = x / 0;  
}  
catch  
{  
    Console.WriteLine("Во время выполнения деления произошла ошибка");  
}  
Console.WriteLine("Продолжение выполнения программы");
```

Когда какой-либо код приводит к исключительной ситуации, выполняется ближайший расположенный в стеке блок `catch`, подходящий для исключений соответствующего типа (о типах исключений мы поговорим позднее). Если подходящий блок `catch` не будет обнаружен во всём стеке вызовов, процесс завершается и выводится сообщение для пользователя. После выполнения блока `catch` продолжается выполнение программы.

Оператор `finally`

Оператор `finally` используется вместе с блоком `try-catch`, когда необходимо выполнить определённый набор операций в любом случае, независимо от того, произошла исключительная ситуация или нет. Чаще всего это требуется при работе с какими-либо системными ресурсами, файлами, внешними ресурсами:

```
string log = string.Empty; // пустая строка - ""
log += $"Начало работы приложения";
try
{
    int x = 10;
    int y = x / 0;
    log += $"{Environment.NewLine}Результат деления: {y}";
}
catch
{
    log += $"{Environment.NewLine}Во время выполнения деления произошла ошибка";
}
finally
{
    File.WriteAllText("log.txt", log);
}
```

Этот код записывает происходящие события в строку `log`, которая в результате сохраняется в файл. Независимо от того, произошла ошибка или нет, содержимое строки `log` будет сохранено в файл `log.txt`. `Environment.NewLine` добавляет перенос строки.

Подключение к базе данных — ещё один подходящий кандидат для заключения в блок `finally`. Количество подключений к серверу БД иногда ограничено, закрывать подключения к базе данных рекомендуется как можно быстрее. Если перед закрытием подключения возникает исключение, это ещё один случай, когда предпочтительно использовать блок `finally`.

При обработке исключений можно использовать ключевое слово `when`. Мы рассмотрим его применение позднее в этом уроке.

Тип исключительных ситуаций

Блок `try-catch` обрабатывает только исключения подходящего типа. Если тип исключения не указан, блок будет обрабатывать все исключения. Исключение в .NET — это любой класс, унаследованный от класса `System.Exception`, а также сам `System.Exception`.

Следующий код содержит блок `try-catch`, обрабатывающий только ошибки деления на ноль, но во время выполнения программы возникает необработанное исключение `NullReferenceException` и приложение завершает работу:

```
try
{
    object x = null;
    int y = (int)x / 0;
}
catch (DivideByZeroException)
{
    Console.WriteLine("Деление на ноль");
}
```

Мы можем указывать несколько блоков `catch` подряд, а при возникновении исключения оно будет обработано ближайшим блоком:

```
try
{
    object x = null;
    int y = (int)x / 0;
}
catch (DivideByZeroException)
{
    Console.WriteLine("Деление на ноль");
}
catch (NullReferenceException)
{
    Console.WriteLine("Обращение к null");
}
catch
{
    Console.WriteLine("Что-то еще пошло не так");
}
```

Последний блок выполнится для всех исключений, отличных от `DivideByZeroException` и `NullReferenceException`, так как он не содержит никакой конкретики по типу исключения. Так как все исключения наследуются от класса `Exception`, следующие два блока равнозначны:

```
catch (Exception)
{
    Console.WriteLine("Что-то ещё пошло не так");
}
```

```
catch
{
    Console.WriteLine("Что-то ещё пошло не так");
}
```

Когда возникает исключительная ситуация, в блок `catch` передаётся экземпляр класса исключения, созданный кодом, в котором произошла ошибка. В этом объекте может передаваться дополнительная информация о том, что привело к ошибке. Пример использования такого объекта:

```
try
{
    object x = 10;
    int y = (int)x / 0;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine($"Произошла ошибка: {ex.Message}");
    Console.WriteLine(ex.StackTrace);
}
```

Этот пример перехватывает исключение и выводит текст ошибки и трассировку стека. Данные свойства есть в любом типе исключений.

Стандартные исключения .NET

Рассмотрим наиболее распространённые стандартные исключения .NET.

`ArgumentException` — исключение, которое возникает, когда среди передаваемых методу аргументов есть недопустимые.

`ArithmeticException` — исключение, которое возникает при ошибках операций арифметического приведения или преобразования.

`AccessViolationException` — исключение, которое возникает при попытке чтения или записи в защищённую область памяти.

`UnauthorizedAccessException` — исключение, которое возникает в случае запрета доступа операционной системой из-за ошибки ввода-вывода или особого типа ошибки безопасности.

`IndexOutOfRangeException` — исключение, возникающее при попытке обращения к элементу массива с индексом, который находится за пределами массива.

`InvalidCastException` — исключение, которое возникает в случае недопустимого приведения или явного преобразования.

`NullReferenceException` — исключение, которое возникает при операциях с объектом, равным `null`, например при попытке получить доступ к его свойствам.

`OutOfMemoryException` — исключение, которое возникает при недостаточном объёме памяти для продолжения выполнения программы.

`StackOverflowException` — исключение, которое возникает при переполнении стека выполнения из-за чрезмерного количества вложенных вызовов методов.

Создание собственных исключений

При разработке программного обеспечения в некоторых ситуациях возникает необходимость в создании собственных типов исключений, например, чтобы обобщённо обрабатывать распространённые ошибки. Для создания собственных типов исключений нужно создать класс, родителем которого будет `System.Exception`. Также хорошей практикой считается помечать собственные исключения атрибутом `[Serializable]`, это поможет избежать ошибок, связанных с потерей данных при передаче таких исключений по сети или в прочих случаях, подразумевающих сериализацию:

```
using System;
namespace HelloWorld
{
    [Serializable]
    public class AddressException : Exception { }
}
```

Напишем код, который приведёт к исключительной ситуации:

```
static void Main()
{
    try
```

```

    {
        Building building = new Building(3, 1);
        string address = GetAddress(building);
    }
    catch (AddressException)
    {
        Console.WriteLine("Здание не содержит адреса");
    }
}

static string GetAddress(Building building)
{
    if (string.IsNullOrEmpty(building.Address))
    {
        throw new AddressException();
    }

    return building.Address;
}

```

Когда необходимо создать исключительную ситуацию, используется ключевое слово `throw`, за которым должен следовать объект с типом, унаследованным от `Exception` (или сам `Exception`). Теперь такое исключение будет перехвачено блоком `catch` с подходящим типом. Благодаря конкретизации типа исключения в блоке `catch` мы знаем точно, какая ошибка возникла в ходе выполнения программы и можем обработать её подходящим образом.

Конструкция `catch-when`

Мы можем обрабатывать разные типы исключений в разных блоках `try-catch`. С помощью ключевого слова мы можем конкретизировать блоки `catch` ещё детальнее, передавая исключение в тот или иной блок в зависимости от условия. Например, нам необходимо расширить исключение с типом `AddressException`:

```

using System;
namespace HelloWorld
{
    public enum ErrorCodes
    {
        NoAddress,
        NoBuilding
    }

    public class AddressException : Exception
    {

```



```

    public AddressException(ErrorCodes code)
    {
        Code = code;
    }

    public ErrorCodes Code { get; }
}

```

Мы добавили исключению конструктор, принимающий код из перечисления. Само перечисление описано выше. Для простоты мы описали и перечисление, и исключение в одном .cs-файле, но на практике желательно создавать отдельный файл под каждую сущность в коде.

Перепишем код в классе Program.cs:

```

static string GetAddress(Building building)
{
    if (string.IsNullOrEmpty(building.Address))
    {
        throw new AddressException(ErrorCodes.NoAddress);
    }

    if (building.Address.Split(' ').Length < 3)
    {
        throw new AddressException(ErrorCodes.NoBuilding);
    }

    return building.Address;
}

```

Как и ранее, мы выбрасываем исключение с типом AddressException, но теперь оно содержит разные значения кода ошибки. Если строка адреса содержит меньше трёх частей, считаем, что номер дома не указан (формат адреса: "ул. {улица}, {дом}").

Мы всё ещё можем обработать обе исключительные ситуации одним блоком catch. Но вместо этого мы воспользуемся ключевым словом when:

```

try
{
    Building building = new Building(3, 1);
    building.Address = "ул. Победы";
    // Для задания корректного адреса раскомментируйте следующую строку:
    // building.Address = "ул. Победы, 23";
    string address = GetAddress(building);
    Console.WriteLine($"Адрес здания: {address}");
}
catch (AddressException ex) when (ex.Code == ErrorCodes.NoAddress)

```

```
{  
  
    Console.WriteLine("Здание не содержит адреса");  
}  
catch (AddressException ex) when (ex.Code == ErrorCodes.NoBuilding)  
{  
  
    Console.WriteLine("Адрес не содержит номер дома");  
}
```

Теперь, в зависимости от значения свойства `Code`, исключение будет обработано одним из описанных блоков.

Процессы

Определение процесса

После сборки исходного кода мы получаем бинарный исполняемый файл с расширением `exe`. После того как бинарный файл запускается, операционная система выделяет ему оперативную память и адресное пространство. Приложение, загруженное в память компьютера, называется процессом.

Входные аргументы процесса

Так же, как и методы, процессы могут иметь входные аргументы — аргументы командной строки. Когда мы открываем какой-либо файл по двойному клику, операционная система запускает ассоциированное с этим типом файла приложение и передаёт в качестве аргумента командной строки путь к файлу, который в дальнейшем открывается самим приложением.

Мы также можем добавить обработку входных аргументов в наших программах. Для этого нужно объявить аргумент с типом `string[]` для функции `Main`, и среда выполнения .NET автоматически присвоит ему значение:

```
static void Main(string[] args)  
{  
    Console.WriteLine(args.Length); // 0  
}
```

Так как мы не передаём никакие аргументы нашему приложению, длина массива равна нулю. Чтобы передать входные аргументы, можно воспользоваться командной строкой:

```
C:\Windows\System32\cmd.exe

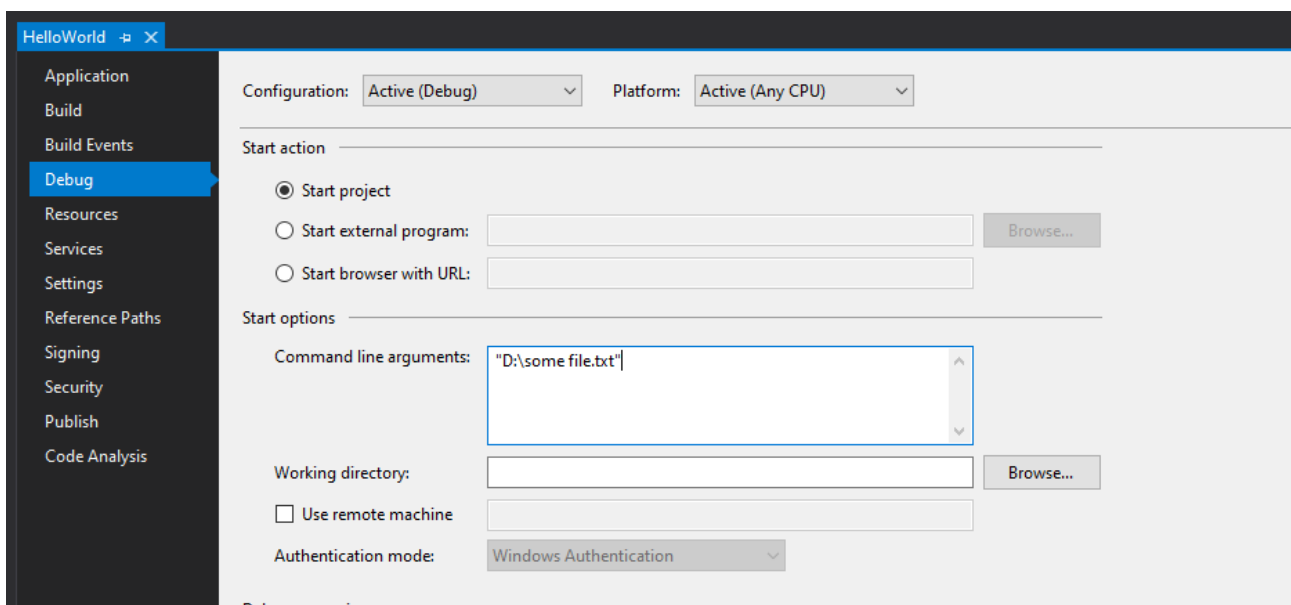
C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\bin\Debug>HelloWorld.exe a b c d
4

C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\bin\Debug>_
```

Этот подход не всегда удобен, особенно в тех случаях, когда мы хотим отладить приложение.

Передача аргументов с помощью Visual Studio

Visual Studio позволяет указать аргументы командной строки, которые будут передаваться приложению при запуске в режиме отладки и без. Для этого необходимо щёлкнуть правой кнопкой мыши по проекту в окне Solution Explorer и выбрать пункт меню Properties. В открывшемся окне перейти на вкладку Debug и в поле Command Line Arguments перечислить входные аргументы, разделённые пробелом. Если необходимо указать аргумент, содержащий пробел, например путь, содержащий пробелы, такой аргумент нужно взять в двойные кавычки:



Изменим содержимое метода `Main` и запустим приложение:

```
static void Main(string[] args)
{
    Console.WriteLine(args[0]);
}
```

Результат выполнения программы:

```
D:\some file.txt
```

Коды возврата

Кроме аргументов командной строки процесс также обладает кодом возврата или кодом завершения (exit code). Это числовое значение, которое сообщает причину завершения работы процесса. Если процесс штатно завершает свою работу, код возврата равен 0. При завершении работы из-за какой-либо ошибки мы можем задать отличный от нуля код ошибки. Для этого нужно задать тип `int` для возвращаемого значения метода `Main` и задать код завершения с помощью оператора `return`:

```
static int Main(string[] args)
{
    return 22;
}
```

Далее мы рассмотрим использование аргументов командной строки и кодов подробнее.

Создание и запуск процесса

Для создания процессов и управления уже запущенными процессами в .NET существует класс `Process` из пространства имён `System.Diagnostics`. В этом классе определён ряд свойств и методов, позволяющих получать информацию о процессах и управлять ими.

`Id` — уникальный идентификатор процесса в рамках текущего сеанса ОС.

`ProcessName` — имя процесса.

`StartTime` — время запуска процесса.

`VirtualMemorySize64` — объём памяти, выделенный для этого процесса.

`CloseMainWindow()` закрывает окно процесса, который имеет графический интерфейс.

`GetProcesses()` возвращает массив всех запущенных процессов.

`GetProcessesByName()` возвращает процессы по их имени.

`Kill()` завершает процесс.

`Start()` запускает новый процесс.

Следующий пример запускает «Блокнот», а после нажатия любой клавиши завершает процесс и выводит код завершения на экран:

```
Process notepad = Process.Start("notepad.exe");
Console.WriteLine("Нажмите любую клавишу для завершения процесса...");
Console.ReadKey();
notepad.Kill();
Console.WriteLine(notepad.ExitCode); // -1
```

Так как метод `Kill()` завершает процесс в экстренном режиме, код завершения «Блокнота» отличен от нуля (`-1`).

Чтобы дождаться штатного завершения программы, необходимо вызвать метод `WaitForExit()`. В таком случае выполнение нашей программы будет приостановлено до тех пор, пока запущенный процесс не завершит свою работу:

```
Process notepad = Process.Start("notepad.exe");
Console.WriteLine("Для продолжения, закройте окно Блокнота...");
notepad.WaitForExit();
Console.WriteLine(notepad.ExitCode); // 0
```

Передача аргументов командной строки

При запуске процесса мы можем указать аргументы командной строки, которые будут переданы процессу:

```
Process notepad = Process.Start("notepad.exe", @"D:\some file.txt");
notepad.WaitForExit();
Console.WriteLine(notepad.ExitCode);
```

В этом примере «Блокнот» откроет для редактирования файл `"D:\some file.txt"` или предложит его создать, если такой файл не существует.

Чтение стандартного потока вывода

Когда приложение выводит данные в консоль, они сначала помещаются в стандартный поток вывода (Standard Output — `stdout`), а уже затем выводятся в консоль. Есть возможность получить сведения, которые запущенный в нашей программе процесс поместил в стандартный поток вывода.

Чтобы наглядно продемонстрировать работу со стандартным потоком вывода, передачу аргументов и считывание кодов завершения дочерних процессов, мы создадим два консольных приложения. Одно

из них будет выполнять простейшую логику сложения чисел, а другое будет запускать этот калькулятор и выводить полученные сведения.

Создадим новый консольный проект Visual Studio и зададим в качестве названия Calculator.

Содержимое Program.cs приведено ниже:

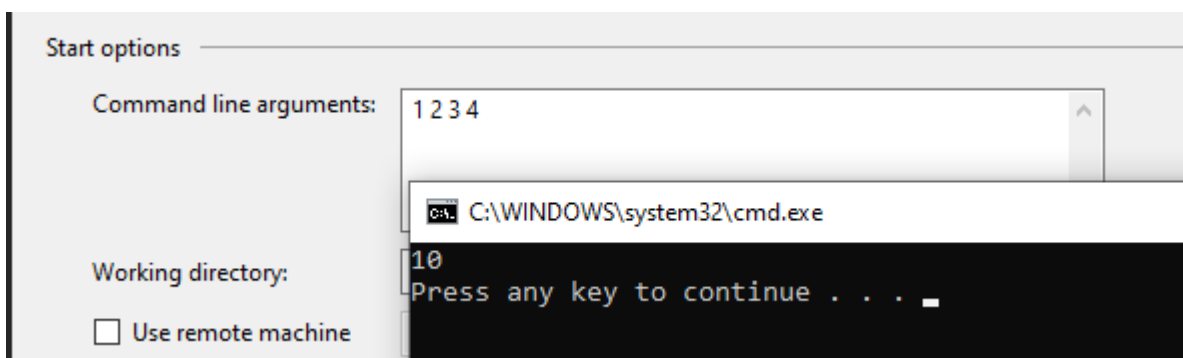
```
using System;

namespace Calculator
{
    class Program
    {
        static int Main(string[] args)
        {
            int sum = 0;
            for (int i = 0; i < args.Length; i++)
            {
                sum += Convert.ToInt32(args[i]);
            }
            Console.Write(sum);

            return -sum;
        }
    }
}
```

Калькулятор складывает все переданные входные аргументы, выводит в консоль сумму и затем завершает работу с кодом, равным `-sum`.

Можно проверить работоспособность калькулятора, указав в настройках проекта числовые аргументы командной строки и запустив его без отладки:



Теперь создадим второй консольный проект с таким содержимым метода Main:

```
static void Main(string[] args)
{
    Process process = new Process();
    process.StartInfo.FileName =
@"C:\Users\Vaagn\source\repos\Calculator\Calculator\bin\Debug\Calculator.exe";
    process.StartInfo.Arguments = "1 2 3 4 5 6 7";
    process.StartInfo.UseShellExecute = false;
    process.StartInfo.RedirectStandardOutput = true;
    process.Start();
    string output = process.StandardOutput.ReadToEnd();

    Console.WriteLine(output);
    process.WaitForExit();
    Console.WriteLine(process.ExitCode);
}
```

Здесь мы создаём экземпляр класса `Process`, задаём ему путь до исполняемого файла написанного ранее калькулятора и указываем аргументы командной строки. Чтобы чтение стандартного вывода дочернего процесса стало возможным, нам необходимо установить два флага:

```
process.StartInfo.UseShellExecute = false;
process.StartInfo.RedirectStandardOutput = true;
```

Первый флаг означает, что для дочернего процесса нужно использовать текущее окно родительского процесса, а не создавать новый экземпляр командной строки. В противном случае будет создано отдельное окно и чтение потока вывода будет невозможным (произойдёт исключение).

Второй флаг означает, что стандартный вывод дочернего процесса нужно перенаправить в родительский процесс, а не выводить на консоль.

Далее мы запускаем наш процесс и читаем весь стандартный вывод дочернего процесса в переменную `output`. Это означает, что всё, что дочерний процесс попытается вывести на консоль с помощью `Console.Write()`, будет объединено в строку и помещено в переменную `output`. `Calculator` выводит в консоль сумму аргументов, поэтому в переменной `output` будет храниться данная сумма. Мы специально использовали `Console.Write()` в `Calculator`, так как этот метод не вставляет символы переноса строки.

Затем мы ожидаем завершения дочернего процесса и выводим в консоль код завершения процесса. Так как Calculator сразу завершает свою работу с кодом `-sum`, мы видим следующий результат:

```
28
-28
Press any key to continue . . .
```

Запуск файла в ассоциированной программе

Если вместо пути к исполняемому файлу передать в `Process.Start()` путь на обычный файл, то он будет открыт в ассоциированной с данным типом файлов программе:

```
if (args.Length > 0)
{
    Process.Start(args[0]);
}
```

Если в свойствах проекта указать путь до существующего файла в качестве входного аргумента и запустить программу, указанный файл откроется в программе по умолчанию.

Получение списка процессов

Упомянутые ранее методы `GetProcesses()` и `GetProcessesByName()` возвращают массив объектов `Process`. Например, чтобы получить список процессов редактора «Блокнот», нужно выполнить следующий вызов:

```
Process[] processes = Process.GetProcessesByName("notepad");
```

Для каждого из этих процессов мы можем выполнить методы `Kill`, `WaitForExit`, просмотреть значения свойств `StartupInfo` и так далее.

Практическое задание

Написать консольное приложение Task Manager, которое выводит на экран запущенные процессы и позволяет завершить указанный процесс. Предусмотреть возможность завершения процессов с помощью указания его ID или имени процесса. В качестве примера можно использовать консольные утилиты Windows `tasklist` и `taskkill`.

Используемые источники

1. [Справочник по C#. Оператор try-catch.](#)
2. [Справочник по C#. Оператор try-catch-finally.](#)
3. [Справочник по C#. Оператор throw.](#)
4. [Использование исключений стандартных типов - Framework Design Guidelines.](#)
5. [Process Класс.](#)