# ИЗКЛЮЧЕНИЯ. ОБРАБОТКА НА ИЗКЛЮЧЕНИЯ

ЛЕКЦИЯ 2

# ОСНОВНИ ТЕМИ

- Изключенията в ООП
- Изключенията в .NET Framework
- Прихващане на изключения
- Предизвикване (хвърляне) на изключения
- Конструкцията try-catch-finally
- Конструкцията try–finally

# Изключение (exception)

- **Изключение (exception)** в програмирането в общия случай представлява уведомление за дадено събитие, нарушаващо нормалната работа на една програма
- Изключенията дават възможност необичайните събития да бъдат обработвани и програмата да реагира на тях по някакъв начин
- Когато възникне **изключение**, конкретното състояние на програмата се запазва и се търси **обработчик на изключението (exception handler)**
- **Изключенията** се **предизвикват** или **"хвърлят" (throw an exception)** от програмен код, който трябва да сигнализира на изпълняващата се програма за **грешка** или **необичайна ситуация**

# КАКВО Е ИЗКЛЮЧЕНИЕ?

- **Изключението** е индикация за необичайна ситуация, като например прекъсване по време на изпълнение: **деление на нула**, **нарушен достъп до паметта**, **масив извън границите**, и т.н.
- **Изключенията** са аномалии, които се случват по време на изпълнението на програмата
- Те може да са следствие от:
  - **потребителски грешки**
  - **логически грешки**
  - **системни грешки**

# ИЗКЛЮЧЕНИЯТА В ООП

- В обектно-ориентираното програмиране (ООП) изключенията представляват мощно средство за **централизирана обработка на грешки** и необичайни ситуации
- В ООП **кодът**, който извършва дадена операция, обикновено **предизвиква изключение**, когато в него **възникне проблем** и операцията **не може да бъде изпълнена успешно**

# ИЗКЛЮЧЕНИЯТА В ООП

- Методът, който извиква операцията може да **прихване изключението** и **да обработи грешката**
- **или** да **пропусне изключението** и да остави то да бъде **прихванато** от извикващият го метод
- **не е задължително** грешките да бъдат обработвани непосредствено от извикващия код
- това дава възможност **управлението на грешките и необичайните ситуации** да се **извършва на много нива**

# ЙЕРАРХИЯ НА ИЗКЛЮЧЕНИЯТА

- Изключенията в ООП са класове и като такива могат да образуват **йерархии посредством наследяване**
- При прихващането на изключения може да се **обработват наведнъж цял клас от грешки**, а не само дадена определена грешка (както е в процедурното програмиране)
- **Препоръчва се** чрез изключения да се **управлява всяко състояние** на грешка или неочаквано поведение, възникнало по време на изпълнението на една програма

# ИЗКЛЮЧЕНИЯТА В .NET Framework

- Ако **потребителят** (програмистът) не осигури механизъм за разглеждане на тези **изключения**, .NET средата **CLR (Common Language Runtime)** предоставя механизъм по подразбиране, който да прекрати изпълнението на програмата
- **Изключението** прекъсва нормалното протичане на програмата и предизвиква търсене на манипулатор за изключението
- Всички операции от стандартната библиотека на .NET (**Framework Class Library**) **сигнализират за грешки** посредством **хвърляне (throw, raise) на изключение**

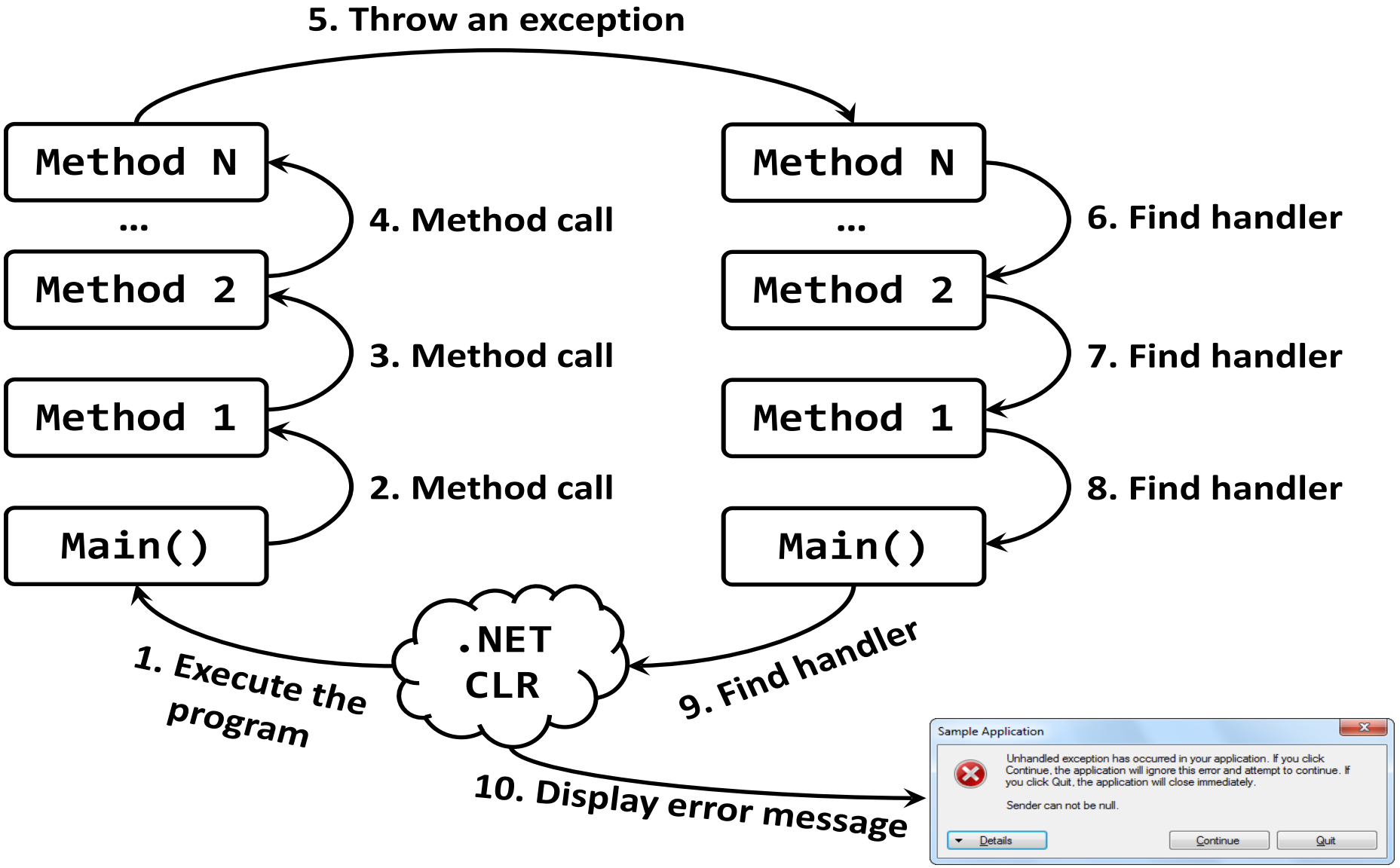
# ИЗКЛЮЧЕНИЯТА В .NET Framework

- **Изключенията** носят в себе си **информация** за настъпилите грешки или необичайни ситуации
- Тази информация може да се **извлича** от тях и е много полезна за **идентифицирането на настъпилия проблем**
- В .NET Framework изключенията пазят в себе си **името на класа и метода**, в който е възникнал проблема
- Ако асемблито е компилирано с грешна (дебъг) информация, **изключенията пазят и името на файла и номера на реда от сорс кода,** където е възникнал проблема

# ИЗКЛЮЧЕНИЯТА В .NET Framework

- Когато възникне **изключение**, изпълнението на програмата спира
- **Изключенията** улесняват писането и поддръжката на **надежден програмен код**, като дават възможност за **обработката** на проблемните ситуации на **много нива**
- В .NET Framework се позволява хвърляне и прихващане на изключения дори **извън** границите на текущия процес
- Всяко изключение в .NET носи т.нар.**stack trace**, който носи информация **къде точно в кода е възникнала грешката**

# СТЕК ЗА ИЗВИКВАНЕ НА МЕТОДИТЕ



# СТЕК С ИНФОРМАЦИЯ ЗА ИЗКЛЮЧЕНИЕТО STACK TRACE

- За да се ориентираме в един **stack trace** трябва да можем да го **разчетем правилно** и да знаем неговата структура
- Stack trace съдържа **подробно описание** на естеството на изключението и на **мястото** в програмата, където то е възникнало
- Stack trace съдържа следната информация в себе си:
  - **Пълното име на класа** на изключението;
  - **Съобщение** – информация за естеството на грешката;
  - **Информация за стека** на извикване на методите.
- Информацията е предназначен за анализиране само от програмистите и администраторите, но не и от крайните потребители на програмата

# ОБРАБОТКА НА ИЗКЛЮЧЕНИЯ (Exception handling )

- **Обработка на изключения** – вграден механизъм на .NET framework за **откриване и отстраняване** на грешки по време на изпълнение
- Работата с изключения включва **две** основни операции
  - прихващане на изключения
  - предизвикване (хвърляне) на изключения
- . NET Framework съдържа много стандартни изключения

# ПРИХВАЩАНЕ НА ИЗКЛЮЧЕНИЯ

- **Изключенията** се предизвикват или "хвърлят" (throw an exception) **от програмен код**, който трябва да **сигнализира** на изпълняващата програма за **грешка или необичайна ситуация**
- В езика C#, може да **изхвърлите** (throw) **изключение с конструкцията** *try – catch – finally*

```
try
{
      // Код, който може да предизвика изключение
}

catch (SomeExceptionClass)
{
      // код, отговорен за обработка (хващане) на изключението
}
```

**филтър** за прихващане на изключения

# ПРИХВАЩАНЕ НА ИЗКЛЮЧЕНИЯ

- **Catch** блокът може да посочи т. нар. **филтър за прихващане на изключения** или **да го пропусне**
- **Филтърът** представлява **име на клас**, **поставен в скоби като параметър** на **catch** оператора
- В горния пример филтърът задава прихващане на изключения от класа **SomeExceptionClass** и всички класове, негови наследници

# ПРИХВАЩАНЕ НА ИЗКЛЮЧЕНИЯ

- Ако филтърът бъде пропуснат, се прихващат всички изключения, независимо от типа им:

```
try
{
   // Код, който може да предизвика изключение
}
catch
{
   // Всяко изключение се прихваща
}
```

# ПРИХВАЩАНЕ НА ПОВЕЧЕ ОТ ЕДНО ИЗКЛЮЧЕНИЕ

- Изразът **catch** може да присъства няколко пъти съответно за различните типове изключения, които трябва да бъдат прихванати

```
try
{
   // Do some work that can raise an exception
}
catch (SomeExceptionClass)
{
   // Handle the SomeExceptionClass and its descendants
}
catch (OtherExceptionClass)
{
   // Handle the OtherExceptionClass and its descendants
}
```

# ПЪЛНА КОНСТРУКЦИЯ ЗА ПРИХВАЩАНЕ try/catch/finally

```
try
{
        // Statement which can cause an exception
}
catch (type x)
{
      // Statements for handling the exception
}
finally
{
      // Any cleanup code
}
```

- finally може да се използва за отстраняване на всякакви грешки

# ОСОБЕНОСТИ НА ПРИХВАЩАНЕТО

- В C#, както **блок catch**, така и **блок finally не са задължителни**
- Всеки **try блок** може да съдържа **един или повече catch блока или finally блок**
- Всеки **try блок** може да съдържа **един или повече catch блока и finally блок**
- Ако **не е станало изключение в try блока, управлението се прехвърля директно на finally блока**
- **finally блокът се изпълнява винаги**

# Exception Classes in C#
# System.Exception class

| Exception Class | Description |
|---|---|
| IOException | Handles I/O errors. |
| IndexOutOfRangeException | Array index is out of bounds. |
| ArrayTypeMismatchException | Type of value being stored is incompatible with the type of the array. |
| NullReferenceException | Handles errors generated from deferencing a null object. |
| DivideByZeroException | Division by zero attempted. |
| InvalidCastException | A runtime cast is invalid. |
| OverflowException | An arithmetic overflow occurred. |
| OutOfMemoryException | A call to new fails because insufficient free memory exists. |
| StackOverflowException | The stack was overrun. |

# ИЗПОЛЗВАНЕ НА TRY И CATCH ПРИМЕРИ

```
class Program1
{
   static void Main()
  {
        try
        {
            int value = 1 / int.Parse("0");
            Console.WriteLine(value);
        }
       catch (DivideByZeroException ex)
        {
            Console.WriteLine(ex.Message);
   }
}
```

- Когато горният код се компилира и изпълни ще се изведе следния резултат:

  **Attempted to divide by zero.**

# try блок, към който са асоциирани няколко catch блока - пример

```
static void Main()
{
   string s = Console.ReadLine();
      try
         {
           Int32.Parse(s);
           Console.WriteLine("You entered valid Int32 number {0}", s);
         }
      catch (FormatException)
         {
            Console.WriteLine("Invalid integer number!");
         }
      catch (OverflowException)
         {
            Console.WriteLine("Number too big to fit in Int32!");
         }
}
```

- Програмата очаква да се въведе цяло число. Ако потребителят въведе нещо различно, ще възникне изключение
- В зависимост от типа на изключението се определя кой catch блок се изпълнява

- Извикването на метода **Int32.Parse(s)** може да предизвика различни изключения

- Ако вместо число се въведе някаква произволна комбинация от символи, при извикването на метода **Int32.Parse(s)** ще възникне изключението **System.FormatException**

- Ако потребителят въведе число, по-голямо от максималната стойност за типа **System.Int32**, при извикването на **Int32.Parse(s)** ще възникне **System.OverflowException**

- Всеки **catch** блок е подобен на метод, който приема точно един аргумент от определен тип изключение

- Може да се зададе и променлива:

```
catch (OverflowException ex)
{
  // Handle the caught exception
}
```

# Multiple Catch Blocks

```
class ExcMulti
{
    static void Main()
    {
       // Here, numer is longer than denom.
       int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
       int[] denom = { 2, 0, 4, 4, 0, 8 };
 for (int i = 0; i < numer.Length; i++)
       {
          try
             {
               Console.WriteLine(numer[i] + " / " +
               denom[i] + " is " +
               numer[i] / denom[i]);
             }
          catch (DivideByZeroException)
             {
                Console.WriteLine("Can't divide by Zero!");
             }
          catch (IndexOutOfRangeException)
             {
                Console.WriteLine("No matching element found.");
             }
       }
Console.ReadKey();
   }
 }
```

# Multiple Catch Blocks

- Catch клаузите се проверяват по реда на срещането им в програмата
- Всички останали се игнорират
- Йерархичната същност на изключенията позволява едновременно прихващане (улавяне) и обработка на цели групи изключения

```
catch
{
     Console.WriteLine("Some exception occurred.");
}
```

**Или**

```
catch (IOException e)
{
        // Handle IOException and all its descendants
}
```

Следващият пример демонстрира често откриваема грешка.
**Достъп до индекс на масив извън неговите граници**.

```
class ExcDemo1
{
      static void Main()
      {
          int[] nums = new int[4];
          try
          {
              Console.WriteLine("Before exception is generated.");
              // Generate an index out-of-bounds exception.
              nums[7] = 10;
              Console.WriteLine("this won't be displayed");
          }
          catch (IndexOutOfRangeException)
          {
              // catch the exception
              Console.WriteLine("Index out-of-bounds!");
          }
          Console.WriteLine("After catch block.");
          Console.ReadKey();
      }
}
```

# ИЗПОЛВАНЕ НА finally

```
class DivNumbers
{
     int result;
     DivNumbers()
     {
         result = 0;
     }
     public void division(int num1, int num2)
     {
         try
         {
            result = num1 / num2;
         }
         catch (DivideByZeroException e)
         {
            Console.WriteLine("Exception caught: {0}", e);
         }
         catch (ArithmeticException e)
         {
            Console.WriteLine("Arithmetic eror: {0}", e);
         }
         finally
         {
            Console.WriteLine("Result: {0}", result);
         }
     }
   static void Main(string[] args)
   {
              DivNumbers d = new DivNumbers();
              d.division(25, 0);
              Console.ReadKey();
          }
   }
```

# ИЗПОЛВАНЕ НА finally

- **finally** block се **изпълнява** винаги след напускане на try/catch блока
- **няма значение** какви условия го предизвикват
- винаги ще се извежда следния резултат:

```
Result: 0.
```

- блокът **finally** няма да се изпълни, ако по време на изпълнението на блока **try** средата за изпълнение CLR прекрати изпълнението си!
- всеки **try** блок може да има нула или повече **catch** блокове и максимум един блок **finally**

# Кога да използваме try-finally?

В много приложения се налага да се работи с външни за програмата ресурси:

- **файлове**, **мрежови връзки**, **графични елементи от операционната система, комуникационни канали** (pipes), **потоци от и към различни периферни устройства** (принтер, звукова карта, карточетец и други)
- При **работата с външни ресурси** е важно след като веднъж е заделен даден ресурс, той да бъде **освободен възможно най-скоро,** след като вече не е нужен на програмата
  - Например, ако отворим някакъв файл, за да прочетем съдържанието му (примерно за да заредим JPEG картинка), е важно да го затворим веднага след като го прочетем
  - Ако оставим файла отворен, това ограничава достъпа на останалите потребители като забранява някои операции, като промяна на файла и изтриване

# ОСВОБОЖДАВАНЕ НА РЕСУРСИ

- Блокът **finally** е незаменим при нужда от **освобождаване на вече заети ресурси**

```
static void ReadFile(string fileName)
{
        TextReader reader = new StreamReader(fileName);
        try
        {
                string line = reader.ReadLine();
                Console.WriteLine(line);
        }
        finally
        {
                reader.Close();
        }
}
```

- **шаблон за освобождаване на ресурси** (dispose pattern)

# IDisposable и конструкцията using

- Основната употреба на **интерфейса IDisposable** е за освобождаване на ресурси
- В .NET такива ресурси са **графични елементи, файлове, потоци и т.н.**
- Важен метод в интерфейса **IDisposable** е **Dispose()**
- **Dispose() освобождава ресурсите на класа**, който го имплементира
- Когато ресурсите са **потоци, четци или файлове,** освобождаването им може да се извърши с метода **Dispose()**, който извиква метода им **Close()**
- **Close()** затваря и освобождава свързаните с тях ресурси от операционната система

# IDisposable и конструкцията using

```
StreamReader reader = new StreamReader(fileName);
try
{
        // Use the reader here
}
finally
{
if (reader != null)
{
        reader.Dispose();
}
}
```

- Този пример може да се запише съкратено с помощта на ключовата дума **using** в езика C# по следния начин:

```
using (StreamReader reader = new StreamReader(fileName) )
{
        // Use the reader here
}
```

# КОГА ДА ИЗПОЛЗВАНЕ USING?

- Използвайте using при работа с всички класове, които имплементират **IDisposable**
- Проверявайте за **IDisposable** в MSDN
- Този клас обвива в себе си някакъв ресурс, който е ценен и не може да се оставя неосвободен, дори при екстремни условия
- Ако даден клас имплементира **IDisposable**, значи трябва да се освобождава задължително веднага след като работата с него приключи

# Хвърляне (throwing) на изключение

- Всички изключения до този момент са **генерирани автоматично** от системата по време на работа
- Възможно е **ръчно** да се хвърли изключение, като се използва конструкцията **throw**

```
class ThrowException {
static void Main() {
    try {
        Console.WriteLine("Before throw.");
        throw new DivideByZeroException();
    }
    catch (DivideByZeroException) {
        Console.WriteLine("Exception caught.");
   }
Console.WriteLine("After try/catch statement.");
} }
```

# ПОВТОРНО ХВЪРЛЯНЕ (rethrowing) НА ИЗКЛЮЧЕНИЕ Throw ;

```
class RetrowException {
    public static void Main()
        {
            int[] table = new int[] { 10, 11, 12, 13, 14, 15 };
            int idx = 6;
            M(table, idx);
        }
public static void M (int[] table, int idx)
        {
            try {
                Console.WriteLine("Accessing element {0}: {1}", idx,
table[idx]);
            }
           catch (NullReferenceException) {
                Console.WriteLine("A null reference exception");
                throw;       // rethrowing the exception
            }
            catch (DivideByZeroException)
            {
                Console.WriteLine("Divide by zero");
                throw;       // rethrowing the exception
            }
        }                }
```

# БЛАГОДАРЯ ЗА ВНИМАНИЕТО!